GeoQuorums: Implementing Atomic Memory in Mobile *Ad Hoc* Networks*

Shlomi Dolev[1] Seth Gilbert[2] Nancy A. Lynch[2] Alex A. Shvartsman[32]
Jennifer L. Welch[4]

February 25, 2004

Abstract

We present a new approach, the GeoQuorums approach, for implementing atomic read/write shared memory in mobile *ad hoc* networks. Our approach is based on associating abstract atomic objects with certain geographic locations. We assume the existence of *focal points*, geographic areas that are normally "populated" by mobile nodes. For example, a focal point may be a road junction, a scenic observation point, or a water resource in the desert. Mobile nodes that happen to populate a focal point participate in implementing a shared atomic object, using a replicated state machine approach. These objects, which we call focal point objects, are then used to implement atomic read/write operations on a virtual shared object, using our new GeoQuorums algorithm. The GeoQuorums algorithm uses a quorum-based strategy in which each each quorum consists of a set of focal point objects. The quorums are used to maintain the consistency of the shared memory and to tolerate limited failures of the focal point objects, caused by depopulation of the corresponding geographic areas. We present a mechanism for changing the set of quorums on the fly, thus improving efficiency. Overall, the new GeoQuorums algorithm efficiently implements read and write operations in a highly dynamic, mobile network.

1 Introduction

In this paper, we introduce a new approach to designing algorithms for mobile *ad hoc* networks. An *ad hoc* network uses no pre-existing infrastructure, unlike cellular networks that depend on fixed, wired base stations. Instead, the network is formed by the mobile nodes themselves, which cooperate to route communication from sources to destinations.

Ad hoc communication networks are, by nature, highly dynamic. Mobile nodes are often small devices with limited energy that spontaneously join and leave the network. As a mobile node moves, the set of neighbors with which it can directly communicate may change completely.

The nature of *ad hoc* networks makes it challenging to solve the standard problems encountered in mobile computing, such as location management (e.g., [7]), using classical tools. The difficulties arise from the lack of a fixed infrastructure to serve as the backbone of the network. In this paper, we begin to develop a new approach that allows existing distributed algorithms to be adapted for highly dynamic *ad hoc* environments.

Providing atomic [20] (or linearizable [17]) read/write shared memory is a fundamental problem in distributed computing, with applications in mobile *ad hoc* networks. Atomic memory is a basic service that facilitates the implementation of many higher-level algorithms. For example, one might construct a location service by requiring each

[1]Department of Computer Science, Ben-Gurion University, `dolev@cs.bgu.ac.il`
[2]MIT CSAIL, {`sethg`, `lynch`}`@theory.lcs.mit.edu`
[3]Department of Computer Science and Engineering, University of Connecticut, `alex@theory.lcs.mit.edu`
[4]Department of Computer Science, Texas A&M University, `welch@cs.tamu.edu`
*This work is supported in part by NSF grant CCR-0098305 and NSF ITR Grant 0121277. Part of the work of the first author has been done during visits to MIT and Texas A&M. The first author is partially supported by an IBM faculty award, the Israeli ministry of defense, NSF, and the Israeli Ministry of Trade and Industry. The second and third authors are partially supported by AFOSR Contract #F49620-00-1-0097, DARPA Contract #F33615-01-C-1896, NSF Grant 64961-CS, and NTT Grant MIT9904-12. The fourth author is partially supported by the NSF Grant 9988304, 0311368 and by the NSF CAREER Award 9984774. The fifth author is partially supported by NSF Grant 0098305 and Texas Advanced Research Program 000512-0091-2001.

mobile node to periodically write its current location to the memory. Alternatively, a shared memory could be used to collect real-time statistics, for example, recording the number of people in a building. We present here a new algorithm for atomic multi-writer/multi-reader memory in mobile *ad hoc* networks.

The GeoQuorums Approach. We divide the problem of implementing atomic read/write memory into two parts. First, we define a static system model, the Focal Point Object Model, that associates abstract objects with certain fixed geographic locales. The mobile nodes implement this model using a replicated state machine approach. In this way, the dynamic nature of the *ad hoc* network is masked by a static model. Second, we present an algorithm to implement read/write atomic memory using the Focal Point Object Model.

The implementation of the Focal Point Object Model depends on a set of physical regions, known as *focal points*. The mobile nodes within a focal point cooperate to simulate a single virtual object, known as a focal point object. Each focal point supports a local broadcast service, LBcast, which provides reliable, totally ordered broadcast. This service allows each node in the focal point to communicate reliably with every other node in the focal point. The local broadcast service is used to implement a type of replicated state machine, one that tolerates joins and leaves of mobile nodes. If every mobile node leaves the focal point, the focal point object fails.

The atomic read/write memory algorithm is implemented on top of the geographic abstraction, that is, on top of the Focal Point Object Model. Nodes implementing the atomic memory algorithm use a GeoCast service (as in [26, 4]) to communicate with the focal point objects. In order to achieve fault tolerance and availability, the algorithm replicates the read/write shared memory at a number of focal point objects. In order to maintain consistency, accessing the shared memory requires updating certain sets of focal points, known as quorums [13, 30, 31, 2, 27]. An important and novel aspect of our approach is that the members of our quorums are focal points, not mobile nodes.

The algorithm uses two sets of quorums: (i) get-quorums, and (ii) put-quorums, with the property that every get-quorum intersects every put-quorum.[1] There is no requirement that put-quorums intersect other put-quorums, or get-quorums intersect other get-quorums. The use of quorums allows the algorithm to tolerate the failure of a limited number of focal point objects.

Our algorithm uses a Global Position System (GPS) time service, allowing it to process write operations using a single phase; prior single-phase write algorithms made other strong assumptions, for example, relying either on synchrony [31] or single writers [2]. Our algorithm guarantees that all read operations complete within two phases, but allows for some reads to be completed using a single phase: the atomic memory algorithm flags the completion of a previous read or write operation to avoid using additional phases, and propagates this information to various focal point objects. As far as we know, this is an improvement on previous quorum-based algorithms.

For performance reasons, at different times it may be desirable to use different sets of get-quorums and put-quorums. For example, during intervals when there are many more read operations than write operations, it may be preferable to use smaller get-quorums that are well distributed, and larger put-quorums that are sparsely distributed. In this case, a client can rapidly communicate with a get-quorum, while communicating with a put-quorum may be slow. If the operational statistics change, it may be useful to reverse the situation.

The algorithm presented here includes a limited reconfiguration capability: it can switch between a finite number of predetermined configurations. As a result of the static underlying Focal Point Object Model, in which focal point objects neither join nor leave, this is not a severe limitation. The resulting reconfiguration algorithm, however, is quite efficient compared to prior reconfigurable atomic memory algorithms [25, 14]. Reconfiguration does not significantly delay read or write operations, and, as no consensus service is required (as in [25]), reconfiguration terminates rapidly.

This paper contains three primary contributions. First, we introduce the Focal Point Object Model, a geographic abstraction model which allows simple, static algorithms to be adapted for highly dynamic environments. Second, we provide an implementation of the Focal Point Object Model using mobile nodes. Third, we implement a reconfigurable atomic read/write shared memory, using the static Focal Point Object Model.

An extended abstract of this work was previously published in the 17th International Symposium on Distributed Computing (DISC 2003) [9]. In this paper, we more formally separate the algorithm into two distinct components, defining a *Focal Point Object Model*, which can be used as the basis for other algorithms in mobile networks. We also include complete proofs of correctness that were omitted in the prior version.

[1] Elsewhere, these are usually referred to as read-quorums and write-quorums. The operations performed by the objects in these quorums, however, are not typical read and write operations. Therefore we use the put/get terminology.

Other Approaches. Quorum systems are widely used to implement atomic memory in static distributed systems [13, 30, 31, 2, 12, 16]. More recent research has pursued application of similar techniques to highly dynamic environments, like *ad hoc* networks. Many algorithms depend on reconfiguring the quorum systems in order to tolerate frequent joins and leaves and changes in network topology. Some of these [10, 6, 16, 27] require the new configurations to be related to the old configurations, limiting their utility in *ad hoc* networks. Englert and Shvartsman [11] showed that using any two quorum systems concurrently preserves atomicity during more general reconfiguration. Recently, Lynch and Shvartsman introduced RAMBO [25] (extended in [14]), an algorithm designed to support distributed shared memory in a highly dynamic environment. The RAMBO algorithms allow arbitrary reconfiguration, supporting a changing set of (potentially mobile) participants. The GeoQuorums approach handles the dynamic aspects of the network by creating a geographic abstraction, thus simplifying the atomic memory algorithm. While prior algorithms use reconfiguration to provide fault tolerance in a highly dynamic setting, the GeoQuorums approach depends on reconfiguration primarily for performance optimization. This allows a simpler, and therefore more efficient, reconfiguration mechanism.

Haas and Liang [15] also address the problem of implementing quorum systems in a mobile network. Instead of considering reconfiguration, they focus on the problem of constructing and maintaining quorum systems for storing location information. Special participants are designed to perform administrative functions. Thus, the backbone is formed by unreliable, *ad hoc* nodes that serve as members of quorum groups. Stojmenovic and Pena [29] choose nodes to update using a geographically aware approach. They propose a heuristic that sends location updates to a north-south column of nodes, while a location search proceeds along an east-west row of nodes. Note that the north-south nodes may move during the update, so it is possible that the location search may fail. Karumanchi *et al.* [18] focus on the problem of efficiently utilizing quorum systems in a highly dynamic environment. The nodes are partitioned into fixed quorums, and every operation updates a randomly selected group, thus balancing the load. Lee et al. [21] and Bhattacharya [3] have done simulation studies comparing the use of probabilistic quorum systems and traditional quorum systems in implementing location services for mobile *ad hoc* networks.

Document Structure. The rest of the paper is organized as follows. The system model appears in Section 2. In Section 3, we formally define an atomic object, and provide some machinery to prove that an algorithm implements an atomic object. In Section 4, we define the Focal Point Object Model, and provide a brief overview of the algorithm. Section 5 then presents the Operation Manager, an implementation of read/write atomic memory based on the Focal Point Object Model, and Section 6 proves it correct. Section 7 then presents the Focal Point Emulator, an implementation of the Focal Point Object Model, and Section 8 includes a proof of correctness. Section 9 discusses the performance of the algorithm, and Section 10 concludes and presents some areas for future research.

2 System Model

In this section, we describe the underlying theoretical model, and discuss the practical justifications. Figure 1 defines some of the mathematical notation used throughout this paper.

2.1 Theoretical Model

Our world model consists of a bounded region of a two-dimensional plane, populated by mobile nodes. Each mobile node is assigned a unique identifier from a set, I. The mobile nodes may join and leave the system, and may fail at any time. (We treat leaves as failures.) The mobile nodes can move on any continuous path in the plane, with speed bounded by a constant. We assume there exists at least one node, $i_0 \in I$.

Mobile Nodes. We model the location and motion of specific mobile nodes using a (hybrid) *RealWorld* automaton (see [23, 24] for a formal presentation of hybrid automata). The RealWorld automaton represents a part of the environment, and is outside the control of the algorithm. It maintains in its state the current location of every mobile node.

In order to model nodes joining and leaving the system, the RealWorld automaton also maintains in its state an indication for each mobile node whether it has joined the system or failed. Formally, when the execution begins, the RealWorld automaton is initialized with a set, $A \subseteq I$, of active nodes. Each mobile node is similarly initialized with an indicator as to whether it begins the execution awake or asleep. A new node i is activated when the RealWorld

Mathematical Notation:

- I, the totally-ordered set of *node identifiers*
- $i_0 \in I$, a distinguished node identifier in I that is smaller than all other identifiers in I
- S, the set of *port identifiers*, defined as $\mathbb{N}^{>0} \times OP \times I$, where $OP = \{\mathsf{get}, \mathsf{put}, \mathsf{confirm}, \mathsf{recon\text{-}done}\}$
- O, the totally-ordered, finite set of *focal point identifiers*
- T, the set of *tags*, defined as $\mathbb{R}^{\geq 0} \times I$
- U, the set of *operation identifiers*, defined as $\mathbb{R}^{\geq 0} \times S$
- X, the set of *memory locations*
- For each $x \in X$:
  - V_x, the set of *values* for x
  - $v_{0,x}$, the initial value of x
- M, a totally-ordered set of *configuration names*
- $c_0 \in M$, a distinguished configuration in M that is smaller than all other names in M
- C, totally-ordered set of *configuration identifiers*, defined as $\mathbb{R}^{\geq 0} \times I \times M$
- L, set of locations in the plane , defined as $\mathbb{R} \times \mathbb{R}$

Figure 1: Notation used through this paper.

automaton sends a wakeup_i action to node i, and adds i to A. A node i fails when the RealWorld automaton sends a fail_i action to node i and removes i from A. Throughout this paper, for clarity of presentation we omit these details. Each automaton described in this paper can be trivially transformed to only perform operations when it is awake and not yet failed.

The RealWorld automaton contains a *Geosensor* component that maintains the current location of each mobile node. It also maintains the current real time. The time is represented as a nonnegative real number.

The computation at each mobile node is modeled by a set of asynchronous (hybrid) automata (interacting via shared actions), each formed by augmenting a regular I/O automaton with continuous inputs from the Geosensor component of the RealWorld, along with the fail and wakeup actions.

Focal Points. While we make no assumptions about the motion of the mobile nodes, we do assume that certain regions are usually "populated" by mobile nodes. We assume that there exists a collection of non-intersecting regions in the plane, called *focal points*, such that (i) "most" focal points remain "populated" during an execution, and (ii) the mobile nodes in each focal point are able to implement a reliable, atomic broadcast service. We define these properties more formally in the next paragraph. Condition (i) is used to ensure that sufficiently many focal points remain available. Once a focal point becomes unavailable due to "depopulation", we do not allow it to recover if it is repopulated. Condition (ii) ensures that all mobile nodes within a focal point can communicate reliably with each other, and that messages are totally ordered. We assume that each mobile node knows about all the focal points.

More formally, a focal point consists of a unique identifier, chosen from the set O, and a contiguous geographic region in the plane. No two focal points intersect. A node is in a focal point at some point in the execution if (according to the RealWorld automaton) its location is within the region defined by the focal point. A focal point is *populated* throughout an execution if there exists a sequence, $j_0, \ldots, j_k$ of mobile nodes with the following properties:

- Node j_0 is in the focal point when the execution begins. That is, the location of node j_0 at the beginning of the execution (as determined by the RealWorld automaton) is within the region specified for the focal point.
- For all $\ell < k$, node $j_{\ell+1}$ enters the focal point sufficiently far in advance before node j_ℓ leaves the focal point. Node $j_{\ell+1}$ remains in the focal point sufficiently long after node j_ℓ leaves the focal point.
- Node j_k is in the focal point at the end of the execution.

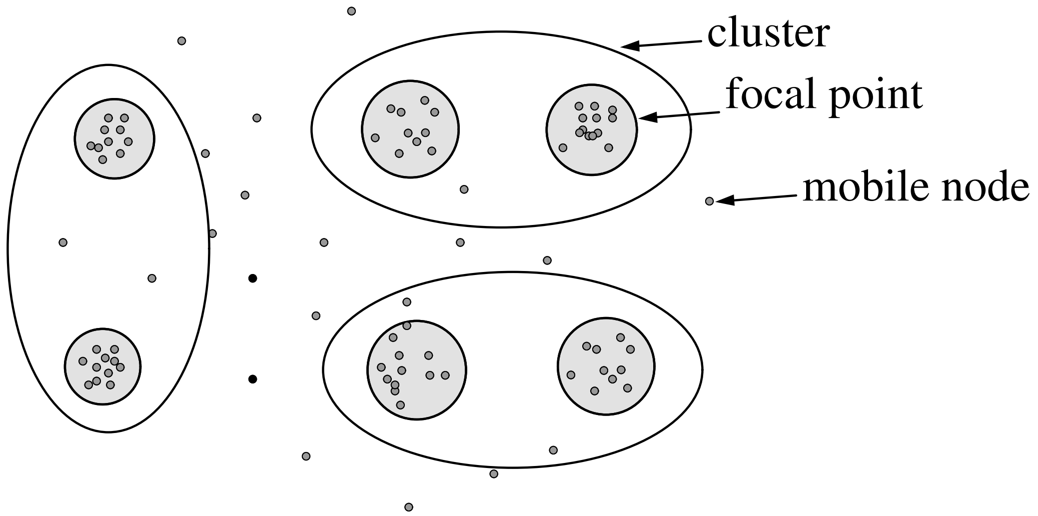

Figure 2: Clusters

Notice that this definition requires that a mobile node joins a focal point "sufficiently prior" to the earlier node leaving and remains in the focal point "sufficiently long" after the earlier node leaves. We intentionally avoid defining "sufficiently prior and "sufficiently long" precisely; however, this time interval must be long enough for the focal point join protocol to complete. This protocol requires two messages to be broadcast, and therefore this "sufficient" time will be proportional to a small number of message broadcast time intervals. If a focal point is populated throughout an execution, it said to be *correct*. If a focal point is not correct, it is said to *fail*. We assume that at most f focal points fail during an execution.

LBcast Service. For each focal point, we assume a reliable, atomic broadcast service. The atomic broadcast service for focal point h, LBcast$_h$, supports the following actions for each mobile node i:

- Input $\mathsf{lbcast}(m)_{h,i}$
- Output $\mathsf{lbcast\text{-}rcv}(m)_{h,i}$

where m is an arbitrary message to be sent. (Notice that the first index of the action indicates the focal point; the second index indicates the node at which the action takes place.) The LBcast service satisfies the usual requirements of a reliable, atomic broadcast service, with the exception that its guarantees hold only for mobile nodes that are in the focal point. Specifically, for any arbitrary focal point, h, the following hold:

***Reliable Delivery*:** Assume that mobile node i performs an $\mathsf{lbcast}(m)_{h,i}$ event using the LBcast$_{h,i}$ service, and that node i is in focal point h when the lbcast occurs. Then for every mobile node, j (potentially the same as node i), that is in focal point h when the message is sent, and remains in the focal point forever thereafter[2] and does not fail (i.e., receive a failure notification from the RealWorld automaton), a $\mathsf{lbcast\text{-}rcv}(m)_{h,j}$ event eventually occurs delivering the message to node j.

***Integrity*:** For any LBcast message m and mobile node i, if an $\mathsf{lbcast\text{-}rcv}(m)_{h,i}$ event occurs, then (1) node i is in focal point h when the message is received, and (2) an $\mathsf{lbcast}(m)_{h,\ell}$ event precedes it, for some mobile node ℓ, and this node ℓ is in focal point h when the lbcast occurs.

***No Duplication*:** For any message m and mobile node i, if at most one $\mathsf{lbcast}(m)_{h,*}$ event occurs, then at most one $\mathsf{lbcast\text{-}rcv}(m)_{h,i}$ event occurs.

[2] Note that after the message is received, node j may of course leave the focal point.

***Total Order*:** There exists a total ordering, $m_1, \ldots, m_k$ of all messages sent on the LBcast$_h$ service during the execution such that if some mobile node, i, receives messages m_r and m_t, then i receives m_r prior to m_t if and only if $r < t$. Notice that there is no requirement that this total ordering relate in any way to the real time order in which the messages were sent.

GeoCast Service. Mobile nodes also depend on a global message delivery service, GeoCast. The GeoCast service delivers a message to a specified destination point in the plane and every node within a certain radius of that destination. Formally, then, the GeoCast service is parameterized by some constant R which determines the size of the destination region. The constant R is chosen to be larger than the radius of the largest focal point, where the "radius" of a focal point is defined in the natural way as the smallest distance such that for some "center" of the focal point the entire region is circumscribed by a circle with the specified radius. Therefore, a GeoCast message can be sent to every mobile node in a focal point by sending a message to the center of the focal point.

For mobile node i, the GeoCast service supports the following two actions:

- Input $\mathsf{geocast}(m, d)_i$
- Output $\mathsf{geocast\text{-}rcv}(m, d)_i$

where m is an arbitrary message to be sent and $d \in L$ is the destination location.

If the message is destined for all the nodes in a focal point, then d is the center of the focal point. Alternatively, if the message is destined for an individual mobile node j, then d is some location that (hopefully) is near to node j. The GeoCast service has the following properties:

***Reliable Delivery*:** Assume that the mobile node i performs a $\mathsf{geocast}(m, d)_i$ action. Then for every mobile node j that is within distance R of location d when the message is sent, and remains within distance R of location d forever thereafter and does not fail, a $\mathsf{geocast\text{-}rcv}(m, d)_j$ event eventually occurs, delivering the message to node j.

***Integrity*:** For any GeoCast message m and mobile node i, if a $\mathsf{geocast\text{-}rcv}(m, d)_i$ event occurs, then (1) node i is within distance R of location d when the message is received, and (2) a $\mathsf{geocast}(m, d)_\ell$ event precedes it, for some mobile node ℓ.

***No Duplication*:** For any message m and mobile node i, if at most one $\mathsf{geocast}(m)_*$ event occurs, then at most one $\mathsf{geocast\text{-}rcv}(m)_i$ event occurs.

***Delivery Time*:** Each message takes some time > 0 to be delivered.

Configurations. We assume a fixed set of *configurations* that is finite and ordered and known to all mobile nodes. Each configuration is assigned a name in M (the set of configuration names), and is defined to be a set of members and two sets of quorums. Configuration c consists of a set of focal point identifiers in O, $members(c)$, and the following sets of quorums: $get\text{-}quorums(c)$ and $put\text{-}quorums(c)$. Each quorum in $get\text{-}quorums(c)$ and $put\text{-}quorums(c)$ is a subset of $members(c)$, that is, a set of focal point identifiers. The quorums have the following intersection property: if $G \in get\text{-}quorums(c)$ and $P \in put\text{-}quorums(c)$, then

$$G \cap P \neq \emptyset\,.$$

Additionally, for any configuration c and any set of f focal point identifiers, F, there exists $G \in get\text{-}quorums(c)$ and $P \in put\text{-}quorums(c)$ such that:

- $F \cap G = \emptyset$
- $F \cap P = \emptyset$.

In other words, no matter what set of f focal points might fail, there is always at least one get-quorum and at least on put-quorum that does not contain any of the failed focal points. Thus, an algorithm based on these quorums can tolerate f focal points failing.

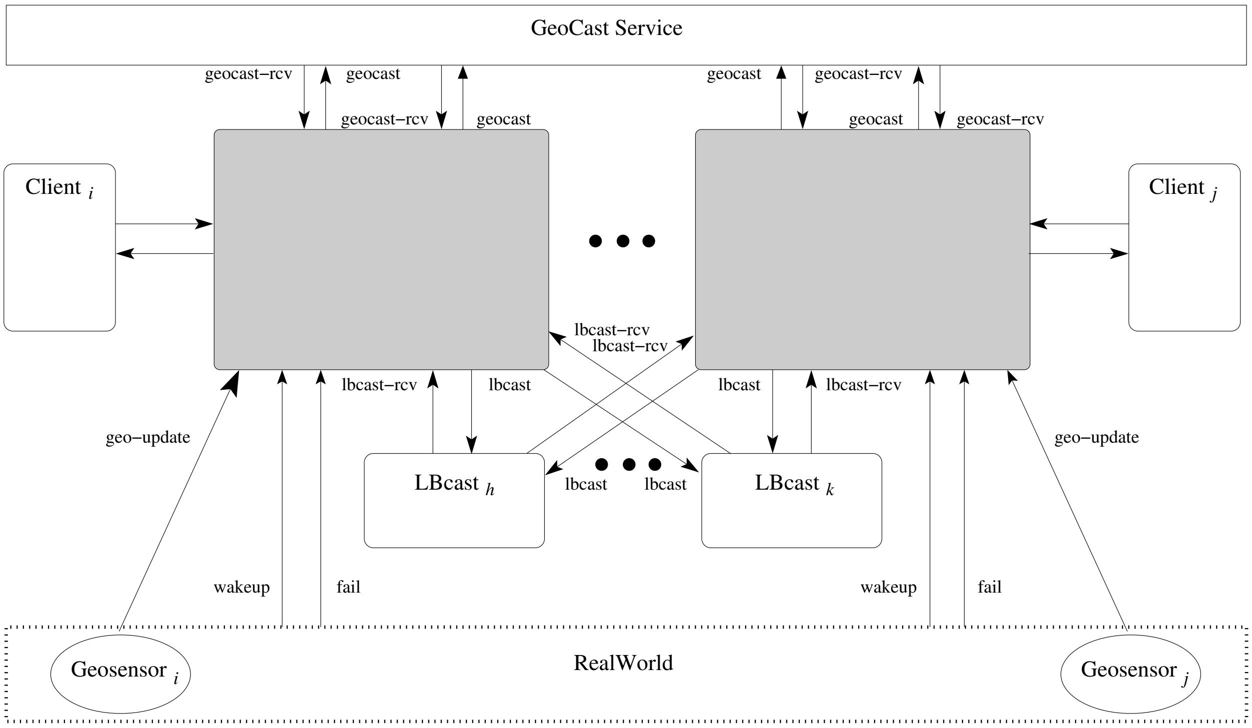

Figure 3: Architecture of the theoretical system model. The shaded boxes represent two mobile nodes, i and j, operating in a highly dynamic environment. Everything inside the shaded area represents programs running on the mobile node. Everything outside the shaded area represents the general system model, including the GeoCast service, multiple LBcast services (one per focal point), and the Geosensor components of the RealWorld for nodes i and j. The RealWorld automaton represents the physical world in which the other automata operate.

Whenever a configuration is used, it is identified by a configuration identifier in C. A configuration identifier consists of three components: a time ($\in R^{\geq 0}$) when the configuration identifier is created, a node identifier ($\in I$) for the mobile node that initiates the use of the configuration, and a configuration name ($\in M$) that identifies the configuration being used. As long as no mobile node proposes more than one configuration at a given instant, then every configuration identifier created during an execution is unique. The configuration identifiers are ordered lexicographically, based first on comparing the time components, then comparing the process identifiers, and then comparing the configuration names.

2.2 Practical Aspects

This theoretical model represents a wide class of real mobile systems. First, there are a number of ways to provide location and time services, as represented by the Geosensor. GPS is perhaps the most common means, but others, like Cricket [28], are being developed to remedy the weaknesses in GPS, such as the inability to operate indoors. Our algorithms can tolerate small errors in the time or location, though we do not discuss this.

Second, the broadcast services specified here are reasonable. Consider the implementation of the LBcast service. If a focal point is small enough, it should be easy to ensure that a single wireless broadcast, with appropriate error correction, reaches every mobile node at the focal point. If the broadcast service uses a time-division/multiple-access (TDMA) protocol, which allocates each node a time slot in which to broadcast, then it is easy to determine a total ordering of the messages. A node joining the focal point might use a separate *reservation channel* to compete for a time slot on the main TDMA *communication channel*, with a fixed, finite number of time slots. This would eliminate collisions on the main channel, while slightly prolonging the process of joining a focal point.

A GeoCast service is also a common communication service in mobile networks: a number of algorithms have been developed to solve this problem, originally for the internet protocol [26] and later for *ad hoc* networks (e.g., [19, 4]).

We propose one set of configurations that may be particularly useful in practical implementations. In this case, we use two configuration c_0 and c_1. We take advantage of the fact that accessing nearby focal points is usually faster than accessing distant focal points. The focal points can be grouped into clusters, using some geographic technique [5]. Figure 2 illustrates the relationship among mobile nodes, focal points, and clusters. For configuration c_0, the $get\text{-}quorums$ are defined to be the clusters. The $put\text{-}quorums$ consist of every set containing one focal point from each cluster. Configuration c_1 is defined in the opposite manner. Assume, for example, that read operations are more common than write operations (and most read operations only require one phase). If the clusters are relatively small and are well distributed (so that every mobile node is near to every focal point in some cluster), then configuration c_0 is quite efficient. On the other hand, if write operations are more common than read operations, configuration c_1 is quite efficient. Our algorithm allows the system to switch safely between two such configurations.

Another difficulty in implementation might be agreeing on the focal points and ensuring that every mobile node has an accurate list of all the focal points and configurations. Some strategies have been proposed to choose focal points: for example, the mobile nodes might send a token on a random walk, to collect information on geographic density [8]. The simplest way to ensure that a mobile node has access to a list of focal points and configurations is to depend on a centralized server, through transmissions from a satellite or a cell-phone tower. Alternatively, the GeoCast service itself might facilitate finding other mobile nodes, at which point the definitive list can be discovered.

3 Atomic Objects

Atomic objects play an important role in this paper. The main result of this paper is an algorithm that implements a highly fault-tolerant read/write atomic object that can tolerate a highly dynamic environment. We also discuss implementing arbitrary atomic objects, using focal points. In this section, we formally define an atomic object, as specified by a *variable type*. We specify the variable type for a *read/write object*, formally describe an *atomic object*, and discuss what it means to *implement* an atomic object.

3.1 Variable Types

An atomic object is specified by a *variable type* that describes its sequential behavior. The definition here is adapted from [22] and [1]. A *variable type*, τ, consists of the following components:

- V, a set of legal values (i.e., states) for the object
- $v_0 \in V$, an initial value (i.e., state) for the object
- $invocations$, a set of invocations
- $responses$, a set of responses
- δ, the transition function, a mapping from:

  $$(invocations \times V) \rightarrow (responses \times V)$$

  that maps every invocation and state to a response and a new state.

We now specify a variable type for a read/write object. (In Section 5 we present a more complicated variable type as part of our algorithm.) A read/write object has the following variable type:

- V, an arbitrary set of values for the atomic object
- $v_0 \in V$, an arbitrary initial value
- $invocations = \{\mathsf{read}\} \cup \{\mathsf{write}(v) : v \in V\}$
- $responses = \{\mathsf{read\text{-}ack}(v) : v \in V\} \cup \{\mathsf{write\text{-}ack}\}$
- δ is defined as follows:

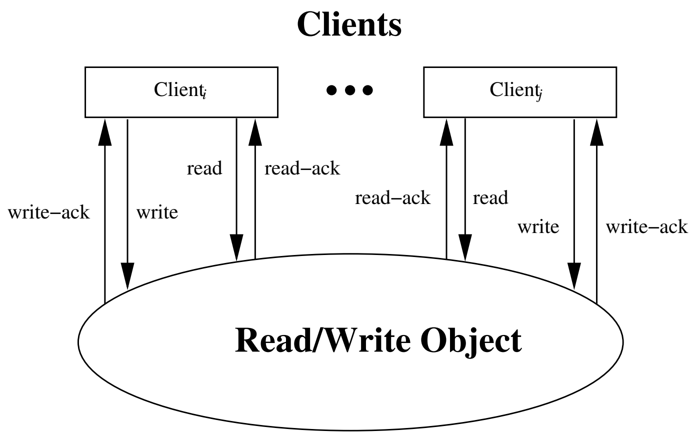

Figure 4: Abstract Read/Write Object

– $\delta(\mathsf{read}, v) \to \langle \mathsf{read\text{-}ack}(v), v \rangle$
– $\delta(\mathsf{write}(v'), v) \to \langle \mathsf{write\text{-}ack}, v' \rangle$

In programmatic style (used later in this paper), the read/write variable type is expressed as the sequential specification presented in Figure 5.

3.2 Canonical Atomic Objects

While the variable type specifies the sequential behavior of an object, it does not specify how the object behaves when it receives concurrent invocations. We therefore specify a canonical atomic object automaton that indicates the legal behaviors of an object of a given variable type. Figure 6 presents the automaton for an atomic object of type

$$\tau = \langle V, v_0, \mathit{invocations}, \mathit{responses}, \delta \rangle$$

with ports in Q, using the I/O automata formalism (see [22] for more details).

The input and output actions are of the form

$$\mathsf{invoke}(\mathit{inv})_p$$

and

$$\mathsf{respond}(\mathit{resp})_p\ ,$$

Read/Write Sequential Specification

State
    value, initially v_0

Operations
read()
    return read-ack(*value*)

write(*new-value*)
    value $\leftarrow$ *new-value*
    return write-ack()

Figure 5: Read/Write Sequential Specification.

Canonical Atomic Object of Type $\tau = \langle V, v_0, invocations, responses, \delta \rangle$, for the set, Q, of ports

Signature:
Input:
invoke$(inv)_p$, $inv \in invocations$, $p \in Q$, the invocations defined by the variable type τ
Output:
respond$(resp)_p$, $resp \in responses$, $p \in Q$, the responses defined by the variable type τ
Internal:
perform$(inv, v, resp, v')_p$, $inv \in invocations$, $resp \in responses$, $v,v' \in V$, $p \in Q$, performs the transitions defined by the variable type τ

State:
$val \in V$, a value, initially v_0
inv-buffer, a set of pairs $\langle inv, p \rangle$ for invocations, $inv \in invocations$, by port $p,\ p \in Q$, initially $\emptyset$
resp-buffer, a set of pairs $\langle resp, p \rangle$ for responses, $resp \in responses$, to port $p,\ p \in Q$, initially $\emptyset$

Transitions:

Input invoke$(inv)_p$
Effect:
$inv\text{-}buffer \leftarrow inv\text{-}buffer \cup \{\langle inv, p \rangle\}$

Output respond$(resp)_p$
Precondition:
$\langle resp, p \rangle \in resp\text{-}buffer$
Effect:
$resp\text{-}buffer \leftarrow resp\text{-}buffer \setminus \{\langle resp, p \rangle\}$

Internal perform$(inv, v, resp, v')_p$
Precondition:
$\langle inv, p \rangle \in inv\text{-}buffer$
$v = val$
$\delta(inv, v) = (resp, v')$
Effect:
$val \leftarrow v'$
$inv\text{-}buffer \leftarrow inv\text{-}buffer \setminus \{\langle inv, p \rangle\}$
$resp\text{-}buffer \leftarrow resp\text{-}buffer \cup \{\langle resp, p \rangle\}$

Figure 6: Canonical Atomic Object Specification

where $inv \in invocations$, $resp \in responses$, and $p \in Q$. Each invocation and response takes place on a port, and each port can support only one operation at a time. Notice that the set of ports of the atomic object is a parameter of the canonical object.

Here Q is a parameter of the canonical automaton; different instantiations of the automaton will uses different sets for Q. In some cases, Q may simply be I, the set of node identifiers, in which case each mobile node has one port on the atomic object. In other cases, Q may be $S = \mathbb{N}^{>0} \times OP \times I$, where OP is some set of operation identifiers (see Figure 1), giving each mobile node a countably infinite number of ports on the object, which allows each mobile node more concurrent access to the object. We will show in more detail why this is useful in Section 5, where we present the Operation Manager, an algorithm that makes use of objects with ports in S.

Figure 4 depicts the atomic object derived from the read/write variable type. In diagrams like Figure 4, for clarity of presentation, instead of writing invoke(read) and the corresponding respond(read-ack), we simply say read and read-ack, as the direction of the arrows makes clear the action involved. We also omit the parameters to the invocations and responses.

Notice that the canonical automaton is not a distributed algorithm; it assumes centralized state that all the nodes can access. It does, however, support concurrent read and write invocations on different ports. The automaton simply performs the operations in some order. This is consistent with the usual notion that an atomic object serializes all of its operations.

The canonical automaton is presented with no liveness conditions. Often (as in [22]), there is an additional fail_p action, for each port in Q, and a "tasks" specification requires that as long as no fail_p action occurs, each invocation on p eventually leads to a response. In this paper, we focus on formally proving the safety properties (in particular, atomicity) of our algorithms, and discuss liveness less formally.

3.3 Implementing Canonical Objects

We say that an automaton, U, is a *well-formed environment* for an atomic object if:

1. Its outputs are exactly the invocations of the object, and its inputs are exactly the responses of the object.
2. In every execution, for every port p, the automaton never performs two consecutive invocations on port p without an intervening response on port p.

We then say that an automaton, S, *implements* the canonical (abstract) object, A, if:

1. S has the same input and output actions as A, the canonical object.
2. If U is a well-formed environment, then any trace of $S \circ U$ is also a trace of $A \circ U$. This implies that S preserves the well-formedness and safety guarantees of A .

(Informally, a trace of an automaton is the sequence of input and output actions occurring in an execution. The symbol $\circ$ represents the composition of automata, as defined in [22])

The most common way of showing that an algorithm implements an atomic object is to show that in every execution there exists a total ordering of the operations with certain properties. This ordering reflects the order in which the operations are performed in the canonical automaton. The following theorem is a variant of Lemmas 13.10 and 13.16 in [22][3].

Theorem 3.1 *Let A be a canonical atomic object of some variable type, and assume that S is an automaton with the same inputs and outputs as A, and that U is any well-formed environment. For every execution α of $S \circ U$ in which every operation completes, assume that the following holds:*

Let Π be the set of operations in α. Assume that there exists a total ordering, $\prec$, on all the operations in Π with the following properties:

1. *The total order is consistent with the external order of invocations and responses. That is, if π completes before π' begins, then $\pi \prec \pi'$.*
2. *Fix some $\pi \in \Pi$. Let $inv_1, inv_2, \cdots, inv_k$ be the invocations of the operations preceding π in the total ordering, indexed according to the total ordering. Let $inv(\pi)$ be the invocation that initiates π, and $resp(\pi)$ be the response that concludes π.*

   Let v be the value of the variable type that results from starting with the initial value, v_0, and processing the following invocations:
$$inv_1, inv_2, \ldots, inv_k \ .$$
   Then the response to operation π is consistent with the object being in state v. That is, consider the $\mathsf{respond}(resp(\pi))$ *event that occurs in α. Then for some value v' of the variable type,*
$$\langle resp(\pi), v' \rangle = \delta(inv(\pi), v) \ .$$

Then $traces(S \circ U) \subseteq traces(A \circ U)$.

[3]Lemmas 13.10 and 13.16 in [22] are presented for a setting with only finitely many ports, while here we allow there to be a countably infinite number of ports. However, nothing in the lemmas or their proofs depends on the number of ports being finite, so the results carry over for our setting.

Since U is an arbitrary environment, this implies that S implements A.

Property 1 requires that the total ordering be consistent with the real-world ordering of operations. Consider the example of a read/write atomic object: this property requires that if a write operation successfully completes and writes some value, val, then a later read operation cannot return an earlier value.

Property 2 requires that the total ordering of operations be consistent with the actual responses sent during the execution, since the total ordering is supposed to represent the order in which operations appear to happen. Consider again the example of a read/write atomic object: Property 2 guarantees that if, in the real execution, a read operation returns some value, val, then the closest preceding write operation (in the total order) must write that same value val.

The proof of Theorem 3.1 is similar to that of Lemma 13.16 in [22]:

Proof (sketch). The proof involves choosing a serialization point for each operation: the earliest point after which the operation has begun and every operation preceding it in the total order has begun, where ties are ordered consistently with the total order. Property 1 ensures that the serialization point occurs before the operation completes and Property 2 ensures that the serialized execution has the same responses as the real execution. □

In the case of a read/write atomic object, it is necessary to determine only a partial ordering of the operations. The following theorem, then, is the analogue of Theorem 3.1, and is proved in [22], Lemmas 13.10 and Lemma 13.16[4]:

Theorem 3.2 *Let A be a canonical atomic read/write object (i.e., an object of the variable type presented in Figure 5), and assume that S is an automaton with the same inputs and outputs as A, and that U is any well-formed environment. For every execution α of $S \circ U$ in which every operation completes, assume that the following holds:*

Let Π be the set of operations in α. Assume that there exists a partial ordering, $\prec$, on all the operations in Π with the following properties:

1. *All write operations are totally ordered, and every read operation is ordered with respect to all the writes.*
2. *The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.*
3. *Every read operation that is ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns v_0.*

Then $traces(S \circ U) \subseteq traces(A \circ U)$.

Again, since U is an arbitrary environment, this implies that S implements A.

4 GeoQuorums Overview

In this section, we present an overview of the GeoQuorums algorithm. The algorithm consists of two independent components: the Focal Point Emulator, which implements the Focal Point Object Model in a highly dynamic environment (described in Section 2), and the Operation Manager, an algorithm that implements an atomic read/write object in the Focal Point Object Model. The GeoQuorums algorithm is the composition of these two sub-algorithms, resulting in an atomic read/write object, with port set $Q = I$, implemented in a mobile *ad hoc* network.

Our implementation is described for a single object, $x \in X$; the composition of all the read/write objects results in a distributed shared read/write memory.

We first define the Focal Point Object Model (Section 4.1) and then provide a brief overview of the Operation Manager (Section 4.3) and the Focal Point Emulator (Section 4.4). A detailed description of the Operation Manager and a proof of correctness appear in Sections 5 and 6. A detailed description of the Focal Point Emulator and a proof of correctness appear in Sections 7 and 8.

4.1 Focal Point Object Model

The Focal Point Object Model is a simple model that hides much of the highly dynamic behavior of the system. It is therefore much easier to specify correct algorithms for the Focal Point Object Model than for the general highly dynamic environment described in Section 2.

[4] In [22], a fourth property is included, assuming that each operation is preceded by only finitely many other operation. This is unnecessary, as it is implied by Property 2.

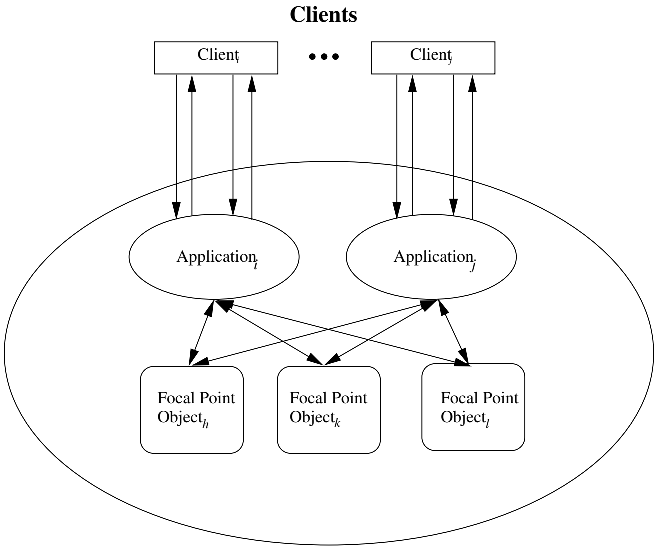

Figure 7: Focal Point Object Model. The application automata are running on the mobile nodes, and receive inputs and send outputs to external clients of the service. The application automata communicate by invoking operations on the focal point objects. (The RealWorld component of the model is omitted for clarity.)

The Focal Point Object Model is a shared memory model. The model consists of two types of entities: application components running on mobile nodes, specified by (hybrid) I/O automata, and atomic objects, specified by variable types. The clients communicate only through their interactions with the atomic objects; there is no message-passing network. We call these objects "focal point objects", hinting at how we later implement them. Each client has a countably infinite number of ports onto each shared object, allowing it to invoke concurrent operations on an object. That is, the port set, Q of each object is $S = \mathbb{N}^{>0} \times OP \times I$, where OP is a set of operations: $\mathsf{get}, \mathsf{put}, \mathsf{confirm},$ and $\mathsf{recon\text{-}done}$. As is usual in shared memory models, the clients can invoke only one operation at a time on each port of an object.

The Focal Point Object Model also guarantees that the response to an invocation on a focal point object comes at a strictly later time (according to the clock available to the mobile nodes) than the invocation. This technical requirement indicates that the clock provides sufficient resolution to measure the time of an operation.

This model is presented schematically in Figure 7, where an arbitrary application interacts with a set of focal point objects. The Focal Point Object Model guarantees that no more than f focal point objects fail in any execution.

4.2 Example Algorithm

As an example of an algorithm that uses the Focal Point Object Model, consider the problem of implementing an *unreliable* atomic read/write memory in the Focal Point Object Model. In this simple example, we use only a single focal point object, obj, with port set $Q = I$. The focal point object implements the read/write memory variable type presented in Figure 5. Let $p = i$, a port for client i on object obj. When a client, i, receives a read request, it simply invokes the read operation on the focal point object. It does this by performing the following action:

$$\mathsf{invoke}(\langle \mathsf{read} \rangle)_p$$

Notice that the operation is invoked on port $p = i$. In this example, there is no reason to use more than a single port per mobile node. Eventually, the focal point object performs the following action:

$$\mathsf{respond}(\langle \mathsf{read\text{-}ack}, val \rangle)_p$$

At this point, the client i can return the value. A write operation proceeds similarly, invoking the write operation on the focal point object.

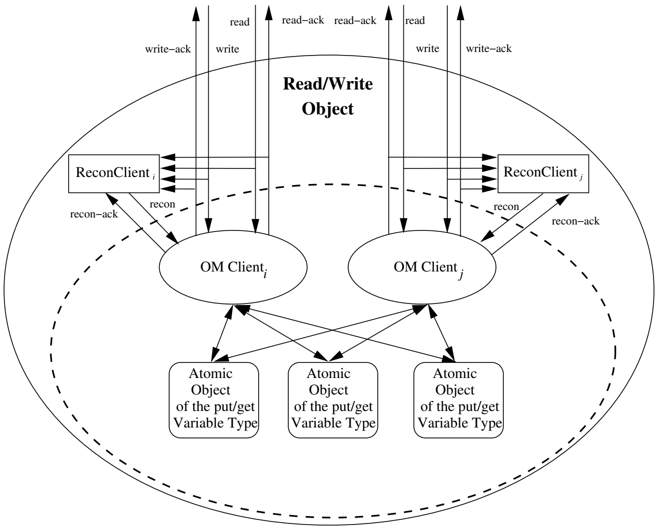

Figure 8: Implementing atomic read/write memory in the Focal Point Object Model. This figure depicts how the Operation Manager is implemented in the Focal Point Object Model (as depicted in Figure 7). The dotted oval depicts the boundary between the Focal Point Object Model and the external clients. The solid oval depicts the boundary of the read/write atomic memory. The application automata of Figure 7 are here the OM Clients (Operation Manager Clients), and the focal point objects are put/get objects. (The RealWorld component of the model is omitted for clarity.)

This simple algorithm solves the problem of implementing *unreliable* read/write atomic memory in the Focal Point Object Model. This algorithm is, effectively, a centralized solution. If the focal point object, obj, fails (either because a focal point itself fails, or because of message loss in the network), then the read/write memory itself fails. In Sections 4.3 and 5 we describe a fault-tolerant algorithm for read/write atomic memory in the Focal Point Object Model.

4.3 Operation Manager Overview

The Operation Manager is an algorithm designed to implement a read/write atomic object (with port set $Q = I$) in the Focal Point Object Model. Using this model significantly simplifies the algorithm. There is no notion of mobility in the Focal Point Object Model, and as a result, the Operation Manager avoids much of the complexity usually associated with an *ad hoc* mobile network. There is no need to handle nodes joining and leaving in any special way, as the only interprocess communication is through the focal point objects. The read/write atomic object is robust, guaranteeing ongoing operation, if at most f focal point objects fail.

The Operation Manager is a quorum-based algorithm for read/write memory. By replicating data at multiple focal point objects, and performing read and write operations on quorums of focal point objects, the Operation Manager ensures that the data is maintained reliably and consistently.

The Operation Manager relies on the variable type of the focal point objects, which we call the put/get variable type. These objects support specially defined operations, put, get, and two others, that allow clients to send information to the objects and retrieve information from the objects, thus exchanging information.

Each read and write operation uses a different port on the focal point objects, so that earlier operations do not interfere with later ones. (In fact, an operation may use two different ports for two different phases during the operation.) This allows for improved performance, since even if a given object is very slow during one operation (perhaps never responding, due to a lost or severely delayed message), it may be used in a later operation.

Figure 8 depicts the various components of the Operation Manager. The dashed black oval represents the boundary interface of the Operation Manager. Notice that this interface includes three operations: read, write and recon. Our goal is to show that the GeoQuorums algorithm implements an atomic read/write object; the recon interface should be hidden from the external environment that will use this is a read/write object.

We therefore assume that there exists a set of reconfiguration automata, one for each mobile node, called $ReconClient_i$, for all i in I. These reconfiguration automata generate recon requests, and receive recon-ack responses. They are not a part of the algorithm presented in this paper, but rather a component specified by a client of the GeoQuorums algorithm. We place only one restriction on the $ReconClient$ automata: the reconfiguration clients are required to respect the environmental well-formedness requirement that a recon request is issued only if there is no ongoing read, write, or reconfiguration. (It would be a relatively simple modification to completely decouple the $ReconClient$ automata from the read/write environment, by allowing concurrent reconfigurations and read/write operations, as is done in RAMBO [25]. We impose this restriction primarily for simplicity of presentation.)

When the Operation Manager is composed with both the $ReconClient$ automata and the focal point objects, the recon and recon-ack actions are hidden, as are the invoke and respond actions on the put/get objects. This results in an external interface consisting only of read/read-ack and write/write-ack actions, as depicted by the solid black oval in Figure 8. This matches the external signature of a read/write object, as specified in Figure 5 (with port set $Q = I$).

4.4 Focal Point Emulator Overview

The Focal Point Emulator implements the Focal Point Object Model in an *ad hoc* mobile network. The nodes in a focal point (i.e., in the specified physical region) collaborate to implement a focal point object. They take advantage of the powerful LBcast service to implement a replicated state machine that tolerates nodes continually joining and leaving. This replicated state machine consistently maintains the state of the atomic object, ensuring that the invocations are performed in a consistent order at every mobile node.

5 Operation Manager

In this section we present the Operation Manager (OM), an algorithm built on the Focal Point Object Model. As the Focal Point Object Model contains two entities, focal point objects and mobile nodes, we present two specifications, one for the objects (each depicted as a "Focal Point Object" in Figure 7) and one for the application running on the mobile nodes (each depicted as an "Application" in Figure 7):

- put/get *variable type* (Figure 10): the variable type of the focal point objects in the Focal Point Object Model.
- *Operation Manager Client* (Figures 9, 11, and 12): an automaton that receives read, write, and recon requests from clients and manages quorum accesses to implement these operations.

Figure 8 depicts the various Operation Manager components. The Operation Manager (OM) is the collection of all the Operation Manager Clients (OM_i, for all i in I). It is composed with the focal point objects, each of which is an atomic object with the put/get variable type.

5.1 The put/get Variable Type

The put/get variable type supports four operations: put, get, confirm, and recon-done. The variable type is specified in Figure 10. The put and get operations are used to set and retrieve the value. We first describe the various state components of the variable type, and then explain the different operations and how they modify the state.

Variable Type State Components. The put/get variable type is used to maintain a $value$, which is therefore the primary component of its state. The variable type also contains a tag component in its state. Each tag consists of a nonnegative real number (the time at which the tag was determined) and a unique process identifier (i.e., $T = \mathbb{R}^{\geq 0} \times I$, see Figure 1). A tag is associated with every value, and the tags determine an ordering on the values that are stored by the put invocations (the only invocations that modify the $value$ component of the state). Ordering these values allows us to order the high-level write operations that create these values, which is necessary for guaranteeing atomic consistency. The put and get invocations take a configuration identifier, $new\text{-}config\text{-}id$, as a parameter (Figure 10, Lines 10 and 19). The put/get variable type includes a $config\text{-}id$ in its state, corresponding to the largest configuration identifier that any put or get invocation has used. The $confirmed\text{-}set$ is a set of tags, indicating whether a tag has been confirmed. We explain later in Section 5.2 when a tag is confirmed. The $recon\text{-}ip$ flag indicates whether the focal point object believes that a reconfiguration is in progress; this is set to true when the object learns about a new configuration, and is set to false when a recon-done indicates that the configuration is fully installed.

Signature:

Input:

$\mathsf{write}(val)_i$, $val \in V$
$\mathsf{read}()_i$
$\mathsf{recon}(cid)_i$, $cid \in C$
$\mathsf{respond}(resp)_{obj,p}$, $resp \in responses(\tau)$, $obj \in O$, $p = \langle *, *, i \rangle \in S$
$\mathsf{geo\text{-}update}(t, l)_i$, $t \in R^{\geq 0}$, $l \in L$

Output:

$\mathsf{write\text{-}ack}()_i$
$\mathsf{read\text{-}ack}(val)_i$, $val \in V$
$\mathsf{recon\text{-}ack}(cid)_i$, $cid \in C$
$\mathsf{invoke}(inv)_{obj,p}$, $inv \in invocations(\tau)$, $obj \in O$, $p = \langle *, *, i \rangle \in S$

Internal:

$\mathsf{read\text{-}2}()_i$
$\mathsf{recon\text{-}2}(cid)_i$, $cid \in C$

State:

confirmed $\subseteq T$, a set of tag ids, initially $\emptyset$
conf-id $\in C$, a configuration id, initially $\langle 0, i_0, c_0 \rangle$
recon-ip, a Boolean flag, initially false
clock $\in \mathbb{R}^{\geq 0}$, a time, initially 0
ongoing-invocations $\subseteq O \times S$, a set of objects and ports, initially $\emptyset$
current-port-number $\in \mathbb{N}^{>0}$, used to invoke objects, initially 1

op, a record with the following components:
type $\in$ {read,write,recon}, initially read
phase $\in$ {idle,get,put}, initially idle
tag $\in T$, initially $\langle 0, i_0 \rangle$
value $\in V$, initially v_0
recon-ip, a Boolean flag, initially false
recon-conf-id $\in C$, a configuration id, initially $\langle 0, i_0, c_0 \rangle$
acc $\subseteq O$, a set of data objects, initially $\emptyset$

Figure 9: Operation Manager Client Signature and State for Node i in I, where τ is the put/get variable type.

Variable Type Transitions. The put/get variable type supports four types of invocations and responses. A put invocation includes three parameters: the $new\text{-}value$, a value to be stored in the state, as well as $new\text{-}tag$, a tag, and $new\text{-}config\text{-}id$, a configuration identifier. The put invocation modifies the $value$ component of the state only if the invocation's tag, $new\text{-}tag$, is larger than the tag stored in the state (i.e., the tag of the last successful put invocation, Figure 10, Line 11). The put invocation also modifies *config-id* if the invocation's configuration identifier, $new\text{-}config\text{-}id$, is larger than the *config-id* stored in the state (Line 14). Whenever the put invocation causes *config-id* to be modified, we assume that a reconfiguration is in progress and set *recon-ip* to true (Line 16).

A put invocation results in a put-ack response. The response includes the configuration identifier stored in the state, $config\text{-}id$, and an indication of whether a reconfiguration is in progress, $recon\text{-}ip$.

A get invocation includes one parameter: $new\text{-}config\text{-}id$, a configuration identifier. The get invocation modifies the state only if the invocation's configuration identifier, $new\text{-}config\text{-}id$, is larger than the *config-id* stored in the state. In this case, the *config-id* component of the state is set to the invocation's configuration identifier, $new\text{-}config\text{-}id$. As in the case of a put invocation, if the *config-id* is modified, *recon-ip* is set to true.

A get invocation results in a get-ack response. This response includes the tag and $value$ stored in the state, as well as the *config-id* and an indication of whether a reconfiguration is in progress, that is, *recon-ip*. It also includes a boolean flag indicating whether the tag is confirmed. That is, it returns true if the tag is in the $confirmed\text{-}set$ stored in the state. Effectively, this indicates whether a confirm invocation has previously indicated that the tag is confirmed.

A confirm invocation takes one parameter: a $new\text{-}tag$. The $confirmed\text{-}set$ component of the state is modified, adding the tag $new\text{-}tag$ to this set. The confirm invocation results in a confirm-ack response.

A recon-done invocation includes a single parameter: a $new\text{-}config\text{-}id$, a configuration identifier. The $recon\text{-}ip$ component of the state is modified if the configuration identifier, $new\text{-}config\text{-}id$, matches the $config\text{-}id$ stored in the state. In this case, $recon\text{-}ip$ is set to false. This indicates that the configuration associated with that configuration identifier is installed, that is, that the reconfiguration that proposed the configuration identifier is complete. This invocation results in a recon-done-ack response. (Note that if $new\text{-}config\text{-}id$ is not equal to $config\text{-}id$, stored in the state, the invocation is ignored. While it may improve performance to allow the recon-done action to modify $config\text{-}id$, we do not do this in the interests of simplicity.)

5.2 Operation Manager Client Specification

The Operation Manager Client uses the atomic objects (with the put/get variable type) provided by the Focal Point Object Model as replicas, invoking put operations to update the focal point objects and get operations to retrieve the value (and associated information) from the focal point objects. Replication allows the Operation Manager Clients to

---

put/get **Variable Type** τ

State

- *tag* $\in T$, initially $\langle 0, i_0 \rangle$
- *value* $\in V$, initially v_0
- *config-id* $\in C$, initially $\langle 0, i_0, c_0 \rangle$
- *confirmed-set* $\subseteq T$, initially $\emptyset$
- *recon-ip*, a Boolean, initially false

Operations

```
put(new-tag, new-value, new-config-id)
    if (new-tag > tag) then
        value ← new-value
        tag ← new-tag
    if (new-config-id > config-id) then
        config-id ← new-config-id
        recon-ip ← true
    return put-ack(config-id, recon-ip)

get(new-config-id)
    if (new-config-id > config-id) then
        config-id ← new-config-id
        recon-ip ← true
    confirmed ← (tag ∈ confirmed-set)
    return get-ack(tag, value, confirmed, config-id, recon-ip)

confirm(new-tag)
    confirmed-set ← confirmed-set ∪ {new-tag}
    return confirm-ack()

recon-done(new-config-id)
    if (new-config-id = config-id) then
        recon-ip ← false
    return recon-done-ack()
```

Figure 10: Definition of the put/get Variable Type τ.

Operation Manager Client Transitions

Output invoke($\langle$get, *config-id*$\rangle$)$_{obj,p}$
Precondition:
  $p = \langle current\text{-}port\text{-}number, \text{get}, i\rangle$
  $\langle obj, p\rangle \notin ongoing\text{-}invocations$
  $obj \notin op.acc$
  $op.phase = \text{get}$
  $config\text{-}id = conf\text{-}id$
Effect:
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \cup \{\langle obj, p\rangle\}$

Output invoke($\langle$put, *tag*, *value*, *config-id*$\rangle$)$_{obj,p}$
Precondition:
  $p = \langle current\text{-}port\text{-}number, \text{put}, i\rangle$
  $\langle obj, p\rangle \notin ongoing\text{-}invocations$
  $obj \notin op.acc$
  $op.phase = \text{put}$
  $tag = op.tag$
  $value = op.value$
  $config\text{-}id = conf\text{-}id$
Effect:
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \cup \{\langle obj, p\rangle\}$

Output invoke($\langle$confirm, *tag*$\rangle$)$_{obj,p}$
Precondition:
  $p = \langle k, \text{confirm}, i\rangle$
  $\langle obj, p\rangle \notin ongoing\text{-}invocations$
  $tag \in confirmed$
Effect:
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \cup \{\langle obj, p\rangle\}$

Output invoke($\langle$recon-done, *config-id*$\rangle$)$_{obj,p}$
Precondition:
  $p = \langle k, \text{recon-done}, i\rangle$
  $\langle obj, p\rangle \notin ongoing\text{-}invocations$
  $recon\text{-}ip = \text{false}$
  $config\text{-}id = conf\text{-}id$
Effect:
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \cup \{\langle obj, p\rangle\}$

Input respond($\langle$get-ack, *tag*, *value*, *confirmed*, *new-cid*, *new-rip*$\rangle$)$_{obj,p}$
Effect:
  if $(\langle current\text{-}port\text{-}number, \text{get}, i\rangle = p)$ **then**
    $op.acc \leftarrow op.acc \cup \{obj\}$
    if $(tag > op.tag)$ **then**
      $op.tag \leftarrow tag$
      $op.value \leftarrow value$
    if $(new\text{-}cid > conf\text{-}id)$ **then**
      $conf\text{-}id \leftarrow new\text{-}cid$
      $op.recon\text{-}ip \leftarrow \text{true}$
      $recon\text{-}ip \leftarrow new\text{-}rip$
    else if $(new\text{-}cid = conf\text{-}id)$ **then**
      $recon\text{-}ip \leftarrow recon\text{-}ip \wedge new\text{-}rip$
    if $(confirm = \text{true})$ **then**
      $confirmed \leftarrow confirmed \cup \{tag\}$
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \setminus \{\langle obj, p\rangle\}$

Input respond($\langle$put-ack, *new-cid*, *new-rip*$\rangle$)$_{obj,p}$
Effect:
  if $(\langle current\text{-}port\text{-}number, \text{put}, i\rangle = p)$ **then**
    $op.acc \leftarrow op.acc \cup \{obj\}$
    if $(new\text{-}cid > conf\text{-}id)$ **then**
      $conf\text{-}id \leftarrow new\text{-}cid$
      $op.recon\text{-}ip \leftarrow \text{true}$
      $recon\text{-}ip \leftarrow new\text{-}rip$
    else if $(new\text{-}cid = conf\text{-}id)$ **then**
      $recon\text{-}ip \leftarrow recon\text{-}ip \wedge new\text{-}rip$
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \setminus \{\langle obj, p\rangle\}$

Input respond($\langle$confirm-ack$\rangle$)$_{obj,p}$
Effect:
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \setminus \{\langle obj, p\rangle\}$

Input respond($\langle$recon-done-ack$\rangle$)$_{obj,p}$
Effect:
  $ongoing\text{-}invocations \leftarrow ongoing\text{-}invocations \setminus \{\langle obj, p\rangle\}$

Figure 11: Operation Manager Client invoke/respond Transitions for Node i.

Operation Manager Client Transitions

Input write$(val)_i$
Effect:
  $current\text{-}port\text{-}number \leftarrow current\text{-}port\text{-}number + 1$
  $op \leftarrow \langle \mathsf{write}, \mathsf{put}, \langle clock, i\rangle, val, recon\text{-}ip, \langle 0, i_0, c_0\rangle, \emptyset\rangle$

Output write-ack$()_i$
Precondition:
  $conf\text{-}id = \langle time\text{-}stamp, pid, c\rangle$
  if $op.recon\text{-}ip$ **then**
    $\forall c' \in M, \exists\, P \in put\text{-}quorums(c') : P \subseteq op.acc$
  else
    $\exists\, P \in put\text{-}quorums(c) : P \subseteq op.acc$
  $op.phase = put$
  $op.type = write$
Effect:
  $op.phase \leftarrow idle$
  $confirmed \leftarrow confirmed \cup \{op.tag\}$

Input read$()_i$
Effect:
  $current\text{-}port\text{-}number \leftarrow current\text{-}port\text{-}number + 1$
  $op \leftarrow \langle \mathsf{read}, \mathsf{get}, \perp, \perp, recon\text{-}ip, \langle 0, i_0, c_0\rangle, \emptyset\rangle$

Output read-ack$(v)_i$
Precondition:
  $conf\text{-}id = \langle time\text{-}stamp, pid, c\rangle$
  if $op.recon\text{-}ip$ **then**
    $\forall c' \in M, \exists\, P \in put\text{-}quorums(c') : P \subseteq op.acc$
  else
    $\exists\, P \in put\text{-}quorums(c)$ **such that** $P \subseteq op.acc$
  $op.phase = \mathsf{put}$
  $op.type = \mathsf{read}$
  $v = op.value$
Effect:
  $op.phase \leftarrow \mathsf{idle}$
  $confirmed \leftarrow confirmed \cup \{op.tag\}$

Internal read-2$()_i$
Precondition:
  $conf\text{-}id = \langle time\text{-}stamp, pid, c\rangle$
  if $op.recon\text{-}ip$ **then**
    $\forall c' \in M, \exists\, G \in get\text{-}quorums(c') : G \subseteq op.acc$
  else
    $\exists\, G \in get\text{-}quorums(c) : G \subseteq op.acc$
  $op.phase = \mathsf{get}$
  $op.type = \mathsf{read}$
  $op.tag \notin confirmed$
Effect:
  $current\text{-}port\text{-}number \leftarrow current\text{-}port\text{-}number + 1$
  $op.phase \leftarrow \mathsf{put}$
  $op.recon\text{-}ip \leftarrow recon\text{-}ip$
  $op.acc \leftarrow \emptyset$

Output read-ack$(v)_i$
Precondition:
  $conf\text{-}id = \langle time\text{-}stamp, pid, c\rangle$
  if $op.recon\text{-}ip$ **then**
    $\forall c' \in M, \exists\, G \in get\text{-}quorums(c') : G \subseteq op.acc$
  else
    $\exists\, G \in get\text{-}quorums(c)$ **such that** $G \subseteq op.acc$
  $op.phase = \mathsf{get}$
  $op.type = \mathsf{read}$
  $op.tag \in confirmed$
  $v = op.value$
Effect:
  $op.phase \leftarrow \mathsf{idle}$

Input recon$(conf\text{-}name)_i$
Effect:
  $conf\text{-}id \leftarrow \langle clock, i, conf\text{-}name\rangle$
  $recon\text{-}ip \leftarrow \mathsf{true}$
  $current\text{-}port\text{-}number \leftarrow current\text{-}port\text{-}number + 1$
  $op \leftarrow \langle \mathsf{recon}, \mathsf{get}, \perp, \perp, \mathsf{true}, conf\text{-}id, \emptyset\rangle$

Internal recon-2$(cid)_i$
Precondition
  $\forall c' \in M, \exists\, G \in get\text{-}quorums(c') : G \subseteq op.acc$
  $\forall c' \in M, \exists\, P \in put\text{-}quorums(c') : P \subseteq op.acc$
  $op.type = \mathsf{recon}$
  $op.phase = \mathsf{get}$
  $cid = op.recon\text{-}conf\text{-}id$
Effect:
  $current\text{-}port\text{-}number \leftarrow current\text{-}port\text{-}number + 1$
  $op.phase \leftarrow \mathsf{put}$
  $op.acc \leftarrow \emptyset$

Output recon-ack$(c)_i$
Precondition:
  $cid = op.recon\text{-}conf\text{-}id$
  $cid = \langle time\text{-}stamp, pid, c\rangle$
  $\exists\, P \in put\text{-}quorums(c)$ **such that** $P \subseteq op.acc$
  $op.type = \mathsf{recon}$
  $op.phase = \mathsf{put}$
Effect:
  if $(conf\text{-}id = op.recon\text{-}conf\text{-}id)$ **then**
    $recon\text{-}ip \leftarrow \mathsf{false}$
  $op.phase \leftarrow \mathsf{idle}$

Input geo-update$(t, l)_i$
Effect:
  $clock \leftarrow t$

Figure 12: Operation Manager Client read/write/recon and geo-update Transitions for Node i.

guarantee fault-tolerance, tolerating the failure of up to f focal point objects. Figure 8 depicts the implementation of a read/write atomic object in the Focal Point Object Model.

Signature. We first describe the signature of the Operation Manager Client, contained in Figure 9. The external signature consists of read, write, and recon actions, to initiate the appropriate operations, and read-ack, write-ack, and recon-ack actions to indicate that the operation have completed.

The external signature also includes invoke and respond actions, to communicate with the focal point objects. Each of these actions is performed on some port, p, for some object, obj.

There are also two internal actions: read-2 and recon-2. The first of these begins the second phase of a read operation. The latter begins the second phase of a reconfiguration operation. (We describe these operations later in this section.)

The Operation Manager Clients are composed with two sets of automata: the focal point objects and the $ReconClient$ automata. The invoke/respond actions allow the Operation Manager to communicate with the focal point objects. The recon and recon-ack allow the Operation Manager to communicate with the $ReconClient$ automata.

State. The state of an Operation Manager Client consists of two parts: some general state that is maintained throughout the execution, and the op record, which maintains state specific to an ongoing operation.

The *confirmed-set* is a set of tags associated with operations that have completed. That is, if a tag is in *confirmed-set*, then some read or write operation associated with that tag has completed.

The *conf-id* is the largest configuration identifier that the Operation Manager Client has received. The Operation Manager Client receives configuration identifiers from respond actions for get-ack and put-ack responses.

The *recon-ip* flag indicates whether the Operation Manager Client believes that a reconfiguration is in progress. The Operation Manager Client sets this flag to true whenever it receives a new, larger configuration identifier (from a respond action), and sets it to false when it receives an indication that the reconfiguration is complete (also from a respond action).

The $clock$ is the current real time, as specified by the Geosensor component of the RealWorld.

The *ongoing-operations* is a set of objects and ports, indicating that an operation has been invoked on the specified port of that object, and that a response has not yet occurred. This is used to ensure the well-formedness condition that atomic objects require: there is only one operation ongoing at any given time on a given port of a given object.

Each invocation of a focal point objects uses a port, which consists of a sequence number, an operation identifier, and a node identifier. The *current-port-number* stores the sequence number component of the port. An invocation by node i, then, uses the port identified by $\langle current\text{-}port\text{-}number, op, i\rangle$, where op is either put, get, confirm, or recon-done.

Every time a new phase of an operation is begun, the *current-port-number* is incremented. Since only one operation can take place on a port at a time, incrementing the port number allows the new phase to perform invocations, even if old invocations on the prior port have not completed.

The op record maintains information specific to a given operation. The $op.type$ field indicates the type of the ongoing operation. The $op.phase$ field indicates the phase of the operation. (Operations may go through two phases: a get phase and a put phase; a write operation performs only a put phase.) The $op.tag$ field indicates the largest tag discovered during the get phase of an operation. The $op.value$ field indicates the value associated with that tag.

The $op.recon\text{-}ip$ field indicates whether a reconfiguration is in progress. Notice that, unlike the general *recon-ip* flag, the $op.recon\text{-}ip$ flag is never reset to false until the phase completes. Once a reconfiguration occurs concurrently with some phase of an operation (and some Operation Manager Client receives information about this reconfiguration), the $op.recon\text{-}ip$ flag is set to true for the rest of the phase.

The $op.recon\text{-}conf\text{-}id$ field is used to keep track of the configuration being installed by an ongoing reconfiguration. While the reconfiguration occurs, a new reconfiguration may be initiated at some other mobile node. This may cause *conf-id* to be modified. The $op.recon\text{-}conf\text{-}id$, however, is not modified until the ongoing reconfiguration is complete.

The $op.acc$ set is an accumulator that maintains the set of object identifiers of objects that have performed a respond during the phase of an operation. A phase completes when $op.acc$ contains a large enough set of object identifiers; in particular, it completes when object identifiers associated with appropriate quorums are contained within the $op.acc$ set.

Read/Write Operations. The code for read/write operations is presented in Figures 11 and 12. We first explain how a write operation proceeds, and then go on to explain read operations.

Each read and write operation consists of either one or two phases. A write operation requires only a single phase, a "put phase" that propagates the new value to at least one quorum of focal point objects. Some read operations require only a single phase, a "get phase", that retrieves the value from at least one quorum of focal point objects. Other read operations require two phases: a "get phase" followed by a "put phase".

Assume that the read or write operation is initiated at node i. During each phase of the operation, node i invokes put and get operations on the focal point objects. Each invocation and subsequent response uses a port.

Each phase of each operation uses a unique port. When a phase begins, node i chooses a new port to use during that phase by incrementing the $\mathit{current\text{-}port\text{-}number}$ (for example, Figure 12, Line 3).

The choice of port serves two purposes. First, it ensures that the Operation Manager Client respects the well-formedness requirement of the focal point objects. Only one operation may occur at a time on each port of each object. By choosing a new port for each phase, we ensure that node i can perform invocations during that phase while respecting the well-formedness requirement.

Second, the use of a unique port for each phase allows, node i to be sure that any response received on port p is the result of an invocation during the phase associated with port p. Any response on any other port (i.e., a port that is not identified by $\mathit{current\text{-}port\text{-}number}$) is ignored (see Figure 11, Line 42, for example), since it results from an earlier (completed) phase.

A write operation begins at node i when OM_i receives a $\mathsf{write}(val)_i$ request. Node i then examines its clock to choose a new tag for the operation (Figure 12, Line 4). For example, if the write is initiated at time t, then the tag is chosen to be $\langle t, i\rangle$. At this point, the $\mathit{current\text{-}port\text{-}number}$ is incremented, choosing port p for this phase of the operation.

The OM_i automaton then begins a put phase, which performs put invocations on the focal point objects (Figure 11, Lines 11–21). We allow invocations to happen, nondeterminstically, on all the focal point objects. In some cases, of course, there is no need to contact all the focal point objects; it is simpler, however, to allow invocations on all of them, and assume that in an optimized implementation invocations occur only on a small set of focal point objects.

The phase completes when the OM_i automaton receives sufficient responses from the objects for operations that it invoked on port p (Figure 11, Lines 57–67). Assume that when the operation begins, the automaton is in the configuration identified by $cid = \langle *, *, c\rangle$ (i.e., $cid = \mathit{conf\text{-}id}_i$). If all responses indicate that c is the most recent configuration identifier, and as a result no reconfiguration is in progress, then the operation terminates when OM_i receives at least one response from each object in some $P \in \mathit{put\text{-}quorums}(c)$ (Figure 12, Line 12).

On the other hand, if any response indicates that a reconfiguration is in progress, then OM_i waits until it receives responses from objects in quorums of every configuration. Specifically, the phase completes when for every configuration c' in M, there is some quorum, $P \in \mathit{put\text{-}quorums}(c')$ such that every object in P has responded to node i during the phase (Figure 12, Line 10).

After the operation the OM_i may notify objects that the tag has been *confirmed*, indicating that the previous operation is complete (Figure 11, Lines 23–29). The confirm invocation uses the port $\langle \mathit{current\text{-}port\text{-}number}, \mathsf{confirm}, i\rangle$, thus ensuring that it does not conflict with put and get invocations.

When the OM_i automaton receives a read request, it first begins a get phase (Figure 12, Line 22) and performs get invocations on the atomic objects (Figure 11, Lines 1–9). Again, assume that when the operation begins, the automaton is in the configuration identified by $cid = \langle *, *, c\rangle$ (i.e., $cid = \mathit{conf\text{-}id}_i$).

As for write operations, if all responses indicate that c is the most recent configuration identifier, then the get phase terminates when OM_i receives a response from each object in some $G \in \mathit{get\text{-}quorums}(c)$. Otherwise, the phase completes when for every configuration c' in M, OM_i receives a response from each object in some quorum $G \in \mathit{get\text{-}quorums}(c')$.

At this point, OM_i chooses the value associated with the largest tag from any of the responses. If the chosen tag has been *confirmed*, then the operation completes (Figure 12, Lines 54–66). Otherwise, OM_i begins a second phase, a put phase, which is identical to the protocol for the write operation (Figure 12, Lines 38–52).

The knowledge of the *confirmed* tags is used to short-circuit the second phase of certain read operations. The second phase is required only when a prior operation with the same tag has not yet completed. By notifying objects when the tag has been confirmed, the algorithm allows later operations to discover that a second phase is unnecessary.

Reconfiguration. The code for the reconfiguration algorithm is presented in Figure 11 (where Lines 31–38 are used by the reconfiguration mechanism, while the rest is used also by the read/write mechanism) and Figure 12, Lines 68–97.

The reconfiguration algorithm differs from the reconfiguration processing presented in the RAMBO algorithm [25, 14]. The new algorithm eliminates the *Recon* service and the associated consensus service, while limiting the number of configurations the system can support. In RAMBO, an arbitrary new configuration can be proposed, while upgrading to the new configuration requires knowledge about all active preceding configurations. The *Recon* service in RAMBO uses consensus to agree on the order of configurations, while the configuration-upgrade operation in RAMBO uses the knowledge of the order and local information about active configurations.

The new reconfiguration algorithm works with a known finite set of possible configurations. The algorithm does not use consensus because all possible preceding configurations are known. The configuration identifiers determine a total ordering on the installed configurations, however it is not necessary that a mobile node be aware of all prior configuration identifiers in the total order. It is sufficient for the reconfiguration algorithm to simply contact all configurations in order to ensure that all configurations preceding it in the total order are contacted. Because this simplification obviates the need for a consensus service, it significantly improves efficiency.

A reconfiguration operation is a two-phase operation similar to a two-phase read operation; it includes a get phase and a put phase. In each phase it requires contacting appropriate quorums of objects from certain configurations.

A reconfiguration begins when the Operation Manager Client receives a $\mathsf{recon}(c)$ input, where c names one of the configurations in M. For the sake of this discussion, assume that the recon is initiated at mobile node i.

First, the Operation Manager Client chooses a new, unique configuration identifier, by examining the local clock, and using its node identifier (i.e., node i) and the name of the new configuration (i.e., configuration c). Specifically, if the $\mathsf{recon}(c)_i$ occurs at time t, then the configuration identifier is $cid = \langle t, i, c \rangle$ (Figure 12, Line 70). At the same time, node i sets its $\mathit{conf\text{-}id}_i$ to the new configuration identifier ($\langle t, i, c \rangle$) and sets $\mathit{recon\text{-}ip}_i$ to true, to indicate that a reconfiguration is in progress (Figure 12, Line 71).

The Operation Manager Client then chooses a new port, incrementing $\mathit{current\text{-}port\text{-}number}_i$ (Figure 12, Line 72). This starts a get phase. During the get phase, $\mathsf{invoke}(\mathsf{get}, \ldots)_{obj,p}$ events occur (Figure 11, Lines 1–9) for objects obj in quorums of all configurations in M.

When a $\mathsf{respond}(\mathsf{get\text{-}ack}, \ldots)_{obj,p}$ event occurs (on the same port p), obj is added to $op.acc$. The phase completes when i has received a response from every object in at least one put-quorum and one get-quorum of each configuration in M.

At this point, a $\mathsf{recon\text{-}2}(cid)_i$ occurs (Figure 12, Lines 75–85), a new port, p', is chosen (Figure 12, Line 83), and the put phase begins. During the put phase, $\mathsf{respond}(\mathsf{put}, \ldots)_{obj,p'}$ events occur (Figure 11, Lines 11–21) for objects obj in quorums of the new configuration, c.

When $\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_{obj,p'}$ events occur (on the same port p'), obj is added to $op.acc$. The phase completes when i has received responses from every object in at least one put-quorum of the new configuration, c (Figure 12, Line 91).

At this point, a $\mathsf{recon\text{-}ack}(cid)_i$ occurs (Figure 12, Lines 87–97), ending the reconfiguration.

If $\mathit{conf\text{-}id}_i$ is equal to $op.\mathit{recon\text{-}conf\text{-}id}$, then $\mathit{recon\text{-}ip}_i$ is set to false (Figure 12, Line 96). Otherwise, a new configuration with a larger configuration identifier has been discovered by node i, and a reconfiguration for this new configuration identifier may be in progress elsewhere. Therefore, in this case, $\mathit{recon\text{-}ip}_i$ is left unchanged.

When a reconfiguration is not in progress, node i may notify focal point objects that the reconfiguration for a certain configuration identifier is done, with recon-done invocations (Figure 11, Lines 31–38).

Finally, notice that the reconfiguration algorithm proceeds in the same way, regardless of whether the newly proposed configuration (i.e., the configuration with name c) is the same as the old configuration: whenever the new configuration identifier is different from the old one, a reconfiguration occurs.

6 The Operation Manager Implements a Read/Write Object

In this section, we show that the Operation Manager guarantees atomic consistency. We show that the Operation Manager correctly implements an atomic read/write object by showing that a partial ordering of operations exists with the properties required by Theorem 3.2. We first define some notation, in Section 6.1. We then define a partial order, in Section 6.2. Next, we prove some preliminary lemmas, in Section 6.3. We then outline the main part of the proof in Section 6.4, and then move on to the main body of the proof in Section 6.5.

6.1 Notation

We first define some notation that we use during the proof. Throughout this section, we fix α to be an execution of the entire system: the Operation Manager, the focal point objects, the reconfiguration clients, and the well-formed environment, U. Additionally, we assume that every read and write operation in α completes. Let Π be the set of read and write operations in α.

There are two ways in which a read operation may conclude: after two phases (see Figure 12, Lines 24–36), or after a single phase (see Figure 12, Lines 54–64). In the first case, at the end of the read operation when the read-ack occurs, $op.phase$ = put, indicating that a "put" phase has completed. In the second case, at the end of the read operation, $op.phase$ = get, indicating that only a single phase, a "get" phase, has completed. In this case, $op.tag$ is in the set *confirmed* immediately before the read completes, so the operation completes after only the get phase.

Every read operation begins with a read action and ends with a read-ack action. We say that a read operation $\pi \in \Pi$ that takes place at node i is a *two-phase read* operation if a read-2_i event occurs between the read_i event and $\mathsf{read\text{-}ack}_i$ event. Operation π is a *one-phase read* operation if no $\mathsf{read}2_i$ event occurs.

We now associate a configuration identifier with each phase of a read or write operation, π, based on the value of the $conf\text{-}id$ of the operation's initiator at the end of that phase. Specifically, if π is a one-phase read operation initiated by node i, then the "get configuration" of π, $get\text{-}conf\text{-}id(\pi)$, is the value of $conf\text{-}id_i$ when π's $\mathsf{read\text{-}ack}_i$ event occurs, ending the get phase. If π is a two-phase read operation, then $get\text{-}conf\text{-}id(\pi)$ is the value of $conf\text{-}id_i$ when π's read-2_i event occurs, ending the get phase. (If π is a write operation, then π has no get phase, so $get\text{-}conf\text{-}id(\pi)$ is undefined.)

If for some operation π, $get\text{-}conf\text{-}id(\pi) = \langle t, i, c\rangle$, then we define $get\text{-}conf(\pi)$ to be c, the name of the configuration identified by the $get\text{-}conf\text{-}id(\pi)$. We say the $get\text{-}conf(\pi)$ is the "get configuration" of π.

If π is a two-phase read operation (respectively, a write operation), then the "put configuration identifier" of π, the $put\text{-}conf\text{-}id(\pi)$, is the value of $conf\text{-}id_i$ when π's $\mathsf{read\text{-}ack}_i$ (respectively, $\mathsf{write\text{-}ack}_i$) event occurs. If π is a reconfiguration operation, then $put\text{-}conf\text{-}id(\pi)$ is the value of $op.conf\text{-}id_i$ when the $recon\text{-}ack$ event occurs. (If π is a one-phase read operation, then π has no put phase, so $put\text{-}conf\text{-}id(\pi)$ is undefined.)

If for some operation π, $put\text{-}conf\text{-}id(\pi) = \langle t, i, c\rangle$, then we define $put\text{-}conf(\pi)$ to be c, the name of the configuration identified by the $put\text{-}conf\text{-}id(\pi)$. We say the $put\text{-}conf(\pi)$ is the "put configuration" of π.

Next, we associate a "recon-in-progress" flag with each phase of a read or write operation, based on the value of $op.recon\text{-}ip$ at the end of that phase. Specifically, if π is a one-phase read operation initiated by node i, then we define $get\text{-}rip(\pi)$ to be the value of $op.recon\text{-}ip_i$ when π's $\mathsf{read\text{-}ack}_i$ event occurs, ending the get phase. If π is a two-phase read operation, then $get\text{-}rip(\pi)$ is the value of $op.recon\text{-}ip_i$ when π's read-2_i event occurs, ending the get phase.

If π is either a two-phase read operation or a write operation, then we define $put\text{-}rip(\pi)$ to be the value of $op.recon\text{-}ip_i$ when π's $\mathsf{read\text{-}ack}_i$ or $\mathsf{write\text{-}ack}_i$ event occurs.

The $get\text{-}rip$ and $put\text{-}rip$ flags indicate whether node i detects a reconfiguration in progress during the get or put phase of the operation. It is sufficient to consider the value of the $op.recon\text{-}ip$ flag at the end of the phase, since the flag is never set to false during the phase: none of the invoke/respond actions set $op.recon\text{-}ip$ to false, only the write, read, read-2, recon, and recon-2 event might have this effect, if $recon\text{-}ip$ is true. If at any time during the phase $recon\text{-}ip$ is set to true, which happens only during a respond event, then $op.recon\text{-}ip$ is set to true at the same time (for example, Figure 11, Lines 49 and 50), and it is therefore true at the end of the phase.

During the proof, if s is a state during the execution and obj is a focal point object, we use the terminology $s.obj$ to refer to the state of the object. If x is a component of the state of the object, we use the terminology $s.obj.field$ to refer to the *field* component of the object. For example, $s.obj.tag$ refers to the tag of the object obj in state s.

6.2 Partial Order

We now construct an appropriate partial ordering, and then show that it meets the necessary requirements of Theorem 3.2. For a read or write operation, $\pi \in \Pi$, initiated at mobile node i, we define $tag(\pi)$ as follows: $tag(\pi) = op.tag_i$ immediately after the acknowledgment of π occurs, that is, when the $\mathsf{read\text{-}ack}_i$ or $\mathsf{write\text{-}ack}_i$ event occurs. (In fact, the tag is often fixed earlier in the operation, as we show in Lemma 6.1.) For a reconfiguration operation, ρ, we define $tag(\rho) = op.tag_i$ immediately after the recon-2 event occurs. We then define the partial order $\prec$:

- For any two operations π_1 and π_2:

$$\text{if } tag(\pi_1) < tag(\pi_2) \text{ then } \pi_1 \prec \pi_2 \ .$$

- For any write operation π_1, and any read operation π_2:

$$\text{if } tag(\pi_1) = tag(\pi_2) \text{ then } \pi_1 \prec \pi_2 \ .$$

We show in Theorem 6.16 that this partial order, $\prec$, satisfies the three conditions of Theorem 3.2. The key condition to prove about the partial ordering is that it is consistent with the ordering of operations in α. That is, we need to show Property 2 of Theorem 3.2, that if π_1 and π_2 are two operations, and π_1 completes before π_2 begins, then π_2 does not precede π_1 in the partial order.

6.3 Preliminary Lemmas

Before beginning the main part of the proof, we prove a few preliminary lemmas. First we examine when during an operation the tag of the operation is fixed. Then we prove some general lemmas about the propagation of tags and values during a put phase and the retrieval of tags and values during a get phase.

Recall that for operation π at node i, $tag(\pi)$ is defined as the value of $op.tag_i$ when the operation completes. In fact, if the operation has a put phase, the tag is fixed prior to the put phase of the operation.

Lemma 6.1 *If π is a write operation at node i, then $tag(\pi) = op.tag_i$ immediately after the* write_i *event. If π is a two-phase read operation, then $tag(\pi) = op.tag_i$ immediately after the* $\mathsf{read\text{-}2}_i$ *event.*

Proof. Assume π is a write operation. In this case, OM_i performs only put invocations. Notice that the

$$\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_i$$

action does not update $op.tag_i$ Therefore $op.tag_i$ does not change after the write_i event until the $\mathsf{write\text{-}ack}_i$ event that concludes the operation and defines the $tag(\pi)$.

Assume π is a read operation. Similarly, after the $\mathsf{read\text{-}2}_i$ event, the OM_i only performs put invocations, so again $op.tag_i$ does not change after the $\mathsf{read\text{-}2}_i$ event, until the $\mathsf{read\text{-}ack}_i$ that concludes the operation and defines the $tag(\pi)$. ☐

We next note that the tag component of the focal point object's state is nondecreasing:

Lemma 6.2 *For every focal point object, obj, the tag of obj is nondecreasing. If s and s' are two states during execution α, and s precedes s', then $s.obj.tag \leq s'.obj.tag$.*

Proof. Immediate by examination of the code that modifies tag. The tag is modified only in Figure 10, Line 13, which is executed only if $new\text{-}tag > tag$. ☐

Next we consider how tag information is propagated during read and write operations to focal point objects. We show that after the put phase of an operation completes, there exists a specific quorum of objects each of which has a tag no smaller than that of the operation.

Lemma 6.3 *Let π be a two-phase read operation, a write operation, or a reconfiguration that occurs at node i. Then there exists a put-quorum, P, in $put\text{-}conf(\pi)$ such that for every object, obj, in P,*

$$tag(\pi) \leq obj.tag$$

anytime after π completes.

Proof. This lemma follows from the termination condition of the put phase of an operation. Assume that when the put phase of π begins (i.e., immediately after the write, read-2, or recon-2 event), $p = \langle current\text{-}port\text{-}number, \mathsf{put}, i \rangle$, the port number that is used throughout the phase. Also, assume that $cid = put\text{-}conf\text{-}id(\pi) = \langle *, *, c \rangle$.

We divide the proof of into two subcases: the case where $put\text{-}rip(\pi) = \mathsf{false}$, and where $put\text{-}rip(\pi) = \mathsf{true}$.

First, consider the case where $put\text{-}rip(\pi) = \mathsf{false}$. Recall that if π is a read or write operation, then cid, the $put\text{-}conf\text{-}id(\pi)$, is equal to the configuration identified by $conf\text{-}id_i$ when the operation completes; if π is a reconfiguration, then cid is equal to the configuration identifier $op.conf\text{-}id_i$ when the operation completes. (Notice that our use of c is consistent with the notation used in Figure 12, Lines 8, 26 and 90.)

Then the precondition for the put phase ending is that there exists a put-quorum $P \in \mathit{put\text{-}quorums}(c)$ such that $P \subseteq op.acc_i$ (see Figure 12, Lines 12, 30 and 91.

An object obj is added to $op.acc$ only when a

$$\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_{obj,p}$$

event occurs (Figure 11, Lines 57–67). The focal point object model guarantees that each respond event is caused by a unique preceding invoke event:

$$\mathsf{invoke}(\mathsf{put}, t, v, cid)_{obj,p}$$

Since the invocation takes place on port p, this means that it must occur after the beginning of the put phase. Therefore, the tag, t, is in fact equal to $tag(\pi)$, the tag at the beginning of the put phase, by Lemma 6.1 and the definition of $tag(\pi)$. The focal point object model guarantees that at some point between the invocation and the response, the put transition was executed on the object's state, thus ensuring that the tag of the object is no smaller than t.

We conclude, then, by Lemma 6.2, that for each object, $obj \in P$, $tag(\pi) \leq obj.tag$ after operation π completes.

We now consider the case where $\mathit{put\text{-}rip}(\pi) = \mathsf{true}$. Assume, then, that π is a two-phase read operation or a write operation. In this case, the precondition for the put phase ending is that for every configuration c', there exists a put-quorum $P \in \mathit{put\text{-}quorums}(c')$ such that $P \subseteq op.acc_i$ (see Figure 12, Lines 10 and 28). Fix $c' = c$.

By the same argument as before, we can conclude that for every object, $obj \in P$, $tag(\pi) \leq obj.tag$ when operation π completes. □

When $\mathit{put\text{-}rip}$ is true there is a stronger version of this lemma for read and write operations: there exists at least one put-quorum for each configuration where every object in the put-quorum has a tag no smaller than the tag of the operation.

Lemma 6.4 *Let π be a two-phase read operation or a write operation that occurs at node i. If $\mathit{put\text{-}rip}(\pi) = \mathsf{true}$, then for every $c \in M$, there exists a put-quorum, $P \in \mathit{put\text{-}quorums}(c)$, such that for every object, obj, in P,*

$$tag(\pi) \leq obj.tag$$

anytime after π completes.

Proof. This lemma follows from the termination condition of the put phase of an operation. Assume that when the put phase of π begins (i.e., immediately after the write, read-2, or recon-2 event), $p = \langle \mathit{current\text{-}port\text{-}number}, \mathsf{put}, i \rangle$, the port that is used throughout the phase. Fix any arbitrary $c \in M$.

The precondition for the put phase ending is that there exists a put-quorum $P \in \mathit{put\text{-}quorums}(c)$ such that $P \subseteq op.acc_i$ (see Figure 12, Lines 10 and 28).

An object, obj is added to $op.acc$ only when a

$$\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_{obj,p}$$

event occurs (Figure 11, Lines 57–67). The focal point object model guarantees that each respond event is caused by a unique preceding invoke event:

$$\mathsf{invoke}(\mathsf{put}, t, v, c)_{obj,p}$$

Since the invocation takes place on port p, this means that it must occur after the beginning of the put phase. Therefore, the tag, t, is in fact equal to $tag(\pi)$, the tag at the beginning of the put phase, by Lemma 6.1 and the definition of $tag(\pi)$. The focal point object model guarantees that at some point between the invocation and the response, the put transition was executed on the object's state, thus ensuring that the tag of the object is no smaller than t.

We conclude, then, by Lemma 6.2, that for each object, $obj \in P$, $tag(\pi) \leq obj.tag$ after operation π completes. Since for every $c \in M$ there exists such a put-quorum, P, the lemma holds. □

We next show that a get phase effectively retrieves information on the tags from a quorum of a certain configuration.

Lemma 6.5 *Let π be a two-phase read operation that occurs at node i. Then there exists a get-quorum, G, in $\mathit{get\text{-}conf}(\pi)$ such that for every object, obj in G,*

$$obj.tag \text{ when } \pi \text{ begins is } \leq tag(\pi)\ .$$

Proof. This lemma is similar to Lemma 6.3, and follows from the termination condition of the get phase of an operation.

Assume that when the get phase of π begins (i.e., immediately after the read event), $p = \langle \mathit{current\text{-}port\text{-}number}, \mathsf{get}, i \rangle$, the port that is used throughout the phase. Also, assume that $cid = \mathit{get\text{-}conf\text{-}id}(\pi) = \langle *, *, c \rangle$.

We divide the proof into two subcases: the case where $\mathit{get\text{-}rip}(\pi) = \mathsf{false}$, and where $\mathit{get\text{-}rip}(\pi) = \mathsf{true}$.

First, consider the case where $\mathit{get\text{-}rip}(\pi) = \mathsf{false}$. Recall that cid, the $\mathit{get\text{-}conf\text{-}id}(\pi)$, is equal to the configuration identifier $\mathit{conf\text{-}id}_i$ when the get phase of the operation completes. (Notice that our use of c is consistent with the notation used in Figure 12, Line 40.)

Then the precondition for the get phase ending is that there exists a get-quorum $G \in \mathit{put\text{-}quorums}(c)$ such that $G \subseteq op.acc_i$ (see Figure 12, Line 44).

An object, obj is added to $op.acc$ only when a

$$\mathsf{respond}(\mathsf{get\text{-}ack}, t, v, \ldots)_{obj,p}$$

event occurs (Figure 11, Lines 40–55). The focal point object model guarantees that each respond event is caused by a unique preceding invoke event:

$$\mathsf{invoke}(\mathsf{get}, \ldots)_{obj,p}$$

Since the invocation takes place on port p, this means that it must occur after the beginning of the get phase. The focal point object model guarantees that the get transition occurs sometime after the invocation and prior to the response. Therefore, the tag t in the response is greater than or equal to $obj.tag$ when the invocation occurs. We therefore conclude, by observing Figure 11, Lines 44–45, that $obj.tag \le \mathit{tag}(\pi)$ when the phase begins.

We now consider the case where $\mathit{get\text{-}rip}(\pi) = \mathsf{true}$. In this case, the precondition for the get phase ending is that for every configuration c', and in particular for the case where $c' = c$, there exists a get-quorum $G \in \mathit{get\text{-}quorums}(c')$ such that $G \subseteq op.acc_i$ (see Figure 12, Line 42).

By the same argument as before, we can conclude that for every object, $obj \in G$, $obj.tag \le \mathit{tag}(\pi)$ when the phase begins. □

Again, in the case where $\mathit{get\text{-}rip}(\pi)$ is true, we can show a stronger property: the get phase retrieves tag information from at least one get-quorum of each configuration.

Lemma 6.6 *Let π be a two-phase read operation that occurs at node i. If $\mathit{recon\text{-}ip}_i = \mathsf{true}$ at the end of the get phase, then for every configuration $c \in M$, there exists a get-quorum, $G \in \mathit{get\text{-}quorums}(c)$ such that for every object, obj in G,*

$$obj.tag \textit{ when } \pi \textit{ begins is } \le \mathit{tag}(\pi)\ .$$

Proof. This lemma is similar to Lemma 6.4, and follows from the termination condition of the get phase of an operation.

Assume that when the get phase of π begins (i.e., immediately after the read event), $p = \langle \mathit{current\text{-}port\text{-}number}, \mathsf{get}, i \rangle$, the port that is used throughout the phase. Fix any arbitrary $c \in M$.

The precondition for the get phase ending is that there exists a get-quorum $G \in \mathit{get\text{-}quorums}(c)$ such that $G \subseteq op.acc_i$ (see Figure 12, Line 42).

An object, obj is added to $op.acc$ only when a

$$\mathsf{respond}(\mathsf{get\text{-}ack}, t, v, \ldots)_{obj,p}$$

event occurs (Figure 11, Lines 40–55). The focal point object model guarantees that each respond event is caused by a unique preceding invoke event:

$$\mathsf{invoke}(\mathsf{get}, \ldots)_{obj,p}$$

Since the invocation takes place on port p, this means that it must occur after the beginning of the get phase. The focal point object model guarantees that the get transition occurs sometime after the invocation and prior to the response. Therefore, the tag t in the response is greater than or equal to $obj.tag$ when the invocation occurs. We therefore conclude, by observing Figure 11, Lines 44–45, that $obj.tag \le \mathit{tag}(\pi)$ when the phase begins. Since for every $c \in M$ there exists such a get-quorum, G, the lemma holds. □

6.4 Outline of the Operation Manager Proof

Our goal is to show that if we have two operations, π_1 and π_2, and π_1 completes before π_2 begins, then $\pi_2 \not\prec \pi_1$. We break this proof into a number of cases:

1. Operation π_2 is a write operation (Lemma 6.7).
2. Operation π_2 is a read operation and operation π_1 is either a two-phase read operation or a write operation.
   (a) $\textit{put-ip}(\pi_1) \vee \textit{get-ip}(\pi_2) = \mathsf{true}$. Either the put phase of π_1 or the get phase of π_2 detects a reconfiguration in progress (Lemma 6.8).
   (b) $\textit{put-ip}(\pi_1) \vee \textit{get-ip}(\pi_2) = \mathsf{false}$. Neither the put phase of π_1 nor the get phase of π_2 detect a reconfiguration in progress.
      i. $\textit{put-conf-id}(\pi_2) = \textit{get-conf-id}(\pi_2)$. The put configuration identifer of π_2 is equal to the get configuration identifer of π_2 (Lemma 6.9).
      ii. $\textit{put-conf-id}(\pi_1) > \textit{get-conf-id}(\pi_2)$. The put configuration identifier of π_1 is larger than the get configuration identifier of π_2 (Lemma 6.11).
      iii. $\textit{put-conf-id}(\pi_1) < \textit{get-conf-id}(\pi_2)$. The put configuration identifier of π_1 is smaller than the get configuration identifier of π_2 (Lemma 6.12).
3. Operation π_2 is a read operation and operation π_1 is a one-phase read operation (Lemma 6.14).

6.5 Proving the Operation Manager Correct

We now proceed to examine the various cases, as outlined above.

Case 1: Write Operation. We first consider the case where π_2 is a write operation:

Lemma 6.7 *If π_1 is a read or write operation, and π_2 is a write operation, and π_1 completes before π_2 begins, then $\pi_1 \prec \pi_2$.*

Proof. Assume that operation π_2 occurs at node i. The result follows immediately by the choice of $tag(\pi_2)$. The tag $op.tag_i$ is chosen during the $\mathsf{write}(v)_i$ action (see Figure 12, Line 4). It is chosen using the real-time clock (along with a process identifier to break ties). The tag of π_1 must have been chosen at the beginning of a prior write operation, or must be the initial value. If the tag of π_1 is the initial value, then the tag of π_2 is necessarily larger. Assume, then, that the tag of π_1 originates with a prior write operation.

This prior write operation must take some time strictly greater than zero to complete, since the write operation requires performing at least one invocation on a focal point object and receiving a response from that invocation. The focal point object model guarantees that each operation consisting of an invocation and a response on a focal point object takes some time to complete: the invocation and the response do not occur at the same time. Therefore the write operation must take some time strictly greater than zero to complete. Since π_2 begins after π_1 ends, it begins at some time strictly greater than zero after the prior write operation begins.

This ensures that $tag(\pi_1) < tag(\pi_2)$, which immediately implies that $\pi_1 \prec \pi_2$, as desired. $\square$

For the rest of the proof we assume that π_2 is a read operation.

Case 2: Two-Phase Read and Write Operations. We now consider the case where π_1 is either a two-phase read operation or a write operation, and π_2 is a read operation. We postpone until later the case where π_1 is a one-phase read operation.

There are two subcases to consider, depending on whether at least one of the flags $\textit{put-rip}(\pi_1)$ or $\textit{get-rip}(\pi_2)$ is true (Case 2(a)), or both flags are false (Case 2(b)).

We first consider the case where at least one of the put phase of π_1 or the get phase of π_2 detects a reconfiguration in progress (Case 2(a)). That is, if i initiates operation π_1 and j initiates π_2, then we assume that at least one of the following two conditions holds:

- At the end of the put phase of π_1, $op.recon\text{-}ip_i = \mathsf{true}$.
- At the end of the get phase of π_2, $op.recon\text{-}ip_j = \mathsf{true}$.

Lemma 6.8 (Case 2(a)) *Assume operation π_1 is a two-phase read or write operation at node i. Assume that π_2 is a read operation initiated at node j, and that π_1 completes before π_2 begins. Assume that at least one of the following is true:*

- *$put\text{-}rip(\pi_1) = \mathsf{true}$, or*
- *$get\text{-}rip(\pi_2) = \mathsf{true}$.*

Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result $\pi_2 \not\prec \pi_1$.

Proof. In this case, at least one of the two nodes detects a reconfiguration in progress: node i during the put phase and/or node j during the get phase. We divide this case into two subcases, depending on whether it is node i or node j that detects the reconfiguration.

Subcase 1:. First, assume that:

$$put\text{-}rip(\pi_1) = \mathsf{true} \ .$$

This implies that node i detects the reconfiguration during the put phase of π_1.

Choose $c' = get\text{-}conf(\pi_2)$. Lemma 6.4 guarantees that there exists a put-quorum, $P \in put\text{-}quorums(c')$, such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ when π_1 completes (since it guarantees this for every $c' \in M$).

Lemma 6.5 guarantees that there exists a get-quorum, G, in $get\text{-}quorums(c')$, the get configuration of π_2, such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins.

Then there must exist an object, $obj \in G \cap P$, since both are quorums of the same configuration c' and one is a get-quorum and the other is a put-quorum.

We already know that $tag(\pi_1) \leq obj.tag$ when π_1 completes. And $obj.tag$ when π_2 begins is $\leq tag(\pi_2)$. Since π_1 completes before π_2 begins, we conclude that $tag(\pi_1) \leq tag(\pi_2)$.

Subcase 2:. Next, assume that:

$$get\text{-}rip(\pi_2) = \mathsf{true} \ .$$

This implies that node j detects the reconfiguration during the get phase of π_2.

Choose $c' = put\text{-}conf(\pi_1)$. Lemma 6.6 guarantees that for every $c' \in M$, there exists a get-quorum, $G \in get\text{-}quorums(c')$, such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins (since it guarantees this for every $c' \in M$).

Lemma 6.3 guarantees that there exists a put-quorum, P, in $put\text{-}quorums(c')$, the put configuration of π_1, such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ when π_1 completes.

Then there must exist an object, $obj \in G \cap P$, since both are quorums of the same configuration c' and one is a get-quorum and the other is a put-quorum.

We already know that $tag(\pi_1) \leq obj.tag$ when π_1 completes. And $obj.tag \leq tag(\pi_2)$ when π_2 begins. Since π_1 completes before π_2 begins, we conclude that $tag(\pi_1) \leq tag(\pi_2)$. □

In the next case (Case 2(b)), we assume that neither the put phase of operation π_1 nor the get phase of operation π_2 detects the reconfiguration in progress. Thus for the next set of lemmas, we assume that $put\text{-}rip(\pi_1)$ and $get\text{-}rip(\pi_2)$ are both false. This case has three subcases, depending on the relationship of the put configuration identifier of π_2 and the get configuration identifier of π_1. First, we assume that these configurations identifiers are the same.

Lemma 6.9 (Case 2(b).i) *Assume that operation π_1 is a two-phase read operation or a write operation at node i, and that π_2 is a read operation at node j. Assume that π_1 completes before π_2 begins.*

Also, assume that $put\text{-}rip(\pi_1)$ and $get\text{-}rip(\pi_2)$ are both false *and that:*

$$put\text{-}conf\text{-}id(\pi_1) = get\text{-}conf\text{-}id(\pi_2) \ .$$

Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result, $\pi_2 \not\prec \pi_1$.

Proof. Let $cid = put\text{-}conf\text{-}id(\pi_2) = get\text{-}conf\text{-}id(\pi_1)$. Assume that $cid = \langle *, *, c \rangle$.

Lemma 6.3 guarantees that there exists a put-quorum, P such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ when π_1 completes.

Lemma 6.6 guarantees that there exists a get-quorum, $G \in get\text{-}quorums(c)$ such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins.

Since P is a put-quorum and G is a get-quorum of the configuration identified by c, there must exist some object, $obj \in P \cap G$.

We already know that $tag(\pi_1) \leq obj.tag$ when π_1 completes. And $obj.tag \leq tag(\pi_2)$ when π_2 begins. Since π_1 completes before π_2 begins, we conclude that $tag(\pi_1) \leq tag(\pi_2)$. □

We now consider the case (Case 2(b).ii) where the put configuration identifier of π_1 is larger than the get configuration identifier of π_2, that is, $put\text{-}conf\text{-}id(\pi_1) > get\text{-}conf\text{-}id(\pi_2)$. It turns out that this case cannot occur.

We first need to show that when the *recon-ip* flag at node i is set to false, this correctly indicates that the configuration identified by *conf-id* is fully installed, meaning that the reconfiguration that created *conf-id* has completed.

Since we assume in this case (Case 2(b).ii) that $get\text{-}rip(\pi_2)$ is false, this lemma shows that the configuration identified by $get\text{-}conf\text{-}id(\pi_2)$ is fully installed prior to the start of π_2.

Lemma 6.10 *Let α' be a prefix of α, and let c be some configuration that is not the initial configuration:*

$$c \neq \langle 0, i_0, c_0 \rangle \ .$$

Assume that at the end of α', $conf\text{-}id_i = cid = \langle *, *, c \rangle$ and $recon\text{-}ip_i =$ false. Then for some node j, a

$$\mathsf{recon\text{-}ack}(cid)_j$$

event occurs in α'.

Proof. Assume, without loss of generality, that α' is the shortest prefix of α such that for any node k, $conf\text{-}id_k = cid$ and $recon\text{-}ip_k =$ false at the end of α'.

There are only two ways in which i can have $conf\text{-}id_i = cid$ and $recon\text{-}ip_i$ set to false: either i performs a recon-ack$(cid)_i$ action (see Figure 12, Line 96), or i receives a put-ack or get-ack response from an object specifying configuration c and $new\text{-}rip =$ false (see Figure 11, Lines 50 and 64). (The recon$(c)_i$ event does result in $conf\text{-}id_i = cid$, however $recon\text{-}ip_i$ is set to true.)

Assume, however, that i receives a put-ack or get-ack from some object, obj, specifying configuration $new\text{-}cid$ and $new\text{-}rip$ set to false. Then a

$$\mathsf{invoke}(\mathsf{recon\text{-}done},\ new\text{-}cid)_{obj, \langle *, j' \rangle}$$

must occur prior to the put-ack or get-ack from obj, as this is the only event that can set $obj.recon\text{-}ip$ to false.

But we assumed that i was the first node to be in this state (i.e., α' is the shortest prefix ending with some node i in this state), so this recon-done invocation cannot occur. Therefore i must perform a $recon\text{-}ack(cid)_i$. The node i, then, satisfies the required properties of node j. □

We can now show that the get configuration identifier of π_2 is always greater than or equal to the put configuration identifier of π_1. Therefore, the second case (Case 2(b).ii) can never occur.

Lemma 6.11 (Case 2(b).ii) *Assume operation π_1 occurs in α at node i before operation π_2 begins at node j. Assume that π_1 is a two phase read or write operation, and π_2 is a read operation.*

Assume that $put\text{-}rip(\pi_1)$ and $get\text{-}rip(\pi_2)$ are both false. *Then $put\text{-}conf\text{-}id(\pi_1) \leq get\text{-}conf\text{-}id(\pi_2)$.*

Proof. If $put\text{-}conf\text{-}id(\pi_1) = \langle 0, i_0, c_0 \rangle$ (the smallest possible value for a configuration identifier), then clearly this result is true. Assume, therefore, that $put\text{-}conf\text{-}id(\pi_1) > \langle 0, i_0, c_0 \rangle$.

It is clear that $recon\text{-}ip_i$ is false at the end of the put phase of π_1, since $op.recon\text{-}ip_i =$ false: whenever $recon\text{-}ip_i$ is set to true, so is $op.recon\text{-}ip_i$, and $op.recon\text{-}ip$ is not reset to false until the phase is completed.

Lemma 6.10 then indicates that for some node, k, a

$$\mathsf{recon\text{-}ack}(put\text{-}conf\text{-}id(\pi_1))_k$$

event occurs prior to the end of the second phase of π_1. In particular, the recon-ack occurs prior to the beginning of π_2.

In order for the recon to complete, a recon-2 must occur. This event completes the get phase of reconfiguration. The precondition of recon-2 requires that for every configuration $c' \in M$, there exists a quorum $P \in \mathit{put\text{-}quorums}(c')$ such that $P \subseteq op.acc$. This implies that each object in quorum P responds to a get invocation in the first phase of the recon operation. (Notice that during a reconfiguration, there are get invocations made on objects in put-quorums. This is the one exception to the general rule that get operations are invoked on objects in get quorums and put operations are invoked on objects in put quorums.)

Choose $c' = \mathit{get\text{-}conf}(\pi_2)$, and let $P \in \mathit{put\text{-}quorums}(c')$ be the put-quorum (described above) contacted by node k prior to the recon-2 event, and therefore prior to the start of π_2.

When the get phase of π_2 completes, there exists some put-quorum of objects, $G \in \mathit{put\text{-}quorums}(\mathit{get\text{-}conf}(\pi_2))$, such that every object, $obj \in G$ has responded to a get invocation during the first phase of π_2.

There must be some object obj in $G \cap P$, as both G and P are quorums of the same configuration, $\mathit{get\text{-}conf}(\pi_2)$, and one is a get-quorum and the other is a put-quorum.

Recall that the reconfiguration in question is installing the configuration $\mathit{put\text{-}conf\text{-}id}(\pi_1)$. As a result of the invocation of object obj during the get phase of the reconfiguration, it is clear that at the end of the get phase, $\mathit{put\text{-}conf\text{-}id}(\pi_1) \leq obj.\mathit{config\text{-}id}$.

As a result of the response of object obj during the get phase of π_2, it is clear that $obj.\mathit{config\text{-}id} \leq \mathit{get\text{-}conf\text{-}id}(\pi_2)$ at the beginning of the get phase.

Therefore, we conclude that:

$$\mathit{put\text{-}conf\text{-}id}(\pi_1) \leq \mathit{get\text{-}conf\text{-}id}(\pi_2) \ .$$

□

The next case to consider is when the put configuration identifier of π_1 is strictly smaller than the get configuration identifier of π_2. This is the most complicated part of the proof, and relies on showing that an intervening reconfiguration operation – the one that creates configuration $\mathit{get\text{-}conf\text{-}id}(\pi_2)$ – relays information from π_1 to π_2.

Lemma 6.12 (Case 2(b).iii) *Assume operation π_1 is a two-phase read or write operation at node i. Assume that π_2 is a read operation initiated at node j, and that π_1 completes before π_2 begins.*

Also, assume that $\mathit{put\text{-}conf\text{-}id}(\pi_1) < \mathit{get\text{-}conf\text{-}id}(\pi_2)$. Finally, assume that $\mathit{put\text{-}rip}(\pi_1)$ and $\mathit{get\text{-}rip}(\pi_2)$ both equal false. *Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result $\pi_2 \not\prec \pi_1$.*

Proof. Some reconfiguration must occur in order to create configuration $\mathit{get\text{-}conf\text{-}id}(\pi_2)$. We first identify the reconfiguration, ρ, that creates the new configuration. We then show that $tag(\rho) \leq tag(\pi_2)$. Finally, we show that $tag(\pi_1) \leq tag(\rho)$, concluding the proof.

Notice that $\mathit{get\text{-}conf\text{-}id}(\pi_2) \neq \langle 0, i_0, c_0 \rangle$, since it is strictly larger than $\mathit{put\text{-}conf\text{-}id}(\pi_1)$, and $\langle 0, i_0, c_0 \rangle$ is the smallest possible value for configuration identifier $\mathit{put\text{-}conf\text{-}id}(\pi_1)$.

Since $op.\mathit{recon\text{-}ip}_j =$ false at the *end* of the get phase of π_2, this means that $\mathit{recon\text{-}ip}_j =$ false at the *beginning* of the get phase of π_2: this is because no action resets $op.\mathit{recon\text{-}ip}_j$ to true during an operation.

Consider the prefix α' of α whose last event is the read event that begins π_2. Then by Lemma 6.10, there exists some node k that performs a recon-ack$(\mathit{get\text{-}conf}(\pi_2))_k$ in α', that is, prior to the read$_j$ of π_2. Let ρ be the recon operation concluding with this recon-ack event.

We now show that $tag(\rho) \leq tag(\pi_2)$. Lemma 6.3 guarantees that there exists some put-quorum, P, in the put configuration of ρ such that for each object, $obj \in P$, $tag(\rho) \leq obj.tag$ at the end of the reconfiguration. Note that P is in $\mathit{put\text{-}quorums}(c)$, where

$$\langle *, *, c \rangle = \mathit{put\text{-}conf\text{-}id}(\rho) = \mathit{get\text{-}conf\text{-}id}(\pi_2) \ .$$

Lemma 6.5 guarantees that there exists some get-quorum, $G \in \mathit{get\text{-}quorums}(c)$ such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins.

Since G is a get-quorum and P is a put-quorum of the configuration identified by $\mathit{get\text{-}conf\text{-}id}(\pi_2)$, there exists an object, $obj_1 \in G \cap P$.

We have already shown that $tag(\rho) \leq obj_1.tag$ when ρ completes. And we have already shown that $obj_1.tag$ when π_2 begins is $\leq tag(\pi_2)$. Since ρ completes before π_2 begins, we conclude from Lemma 6.2 that $tag(\rho) \leq tag(\pi_2)$.

We next show that $tag(\pi_1) \leq tag(\rho)$. Consider the

$$\text{recon-2}(get\text{-}conf\text{-}id(\pi_2))_k$$

event that occurs as part of reconfiguration ρ, ending the get phase of the reconfiguration. The precondition for the recon-2 action requires that for every configuration $c' \in M$, there exists a get-quorums $G \in get\text{-}quorums(c')$ such that $G \subseteq op.acc_k$ when the event occurs. This implies that for every object, $obj \in G$, an

$$\text{invoke}(\text{get}, \ldots)_{obj,p}$$

event and a

$$\text{respond}(\text{get-ack}, \ldots)_{obj,p}$$

event occur during the get phase, where p is the port number during the get phase of the reconfiguration. As part of this get operation, a perform event occurs at the automaton for obj.

Choose $c' = put\text{-}conf\text{-}id(\pi_1)$, and let G be the get-quorum determined by the end of the get phase of the reconfiguration. Lemma 6.3 guarantees that there exists some put-quorum, $P \in put\text{-}quorums(c')$ such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ at the end of π_1.

Since G is a get-quorum and P is a put-quorum of the configuration identified by c', there exists some object, $obj_2 \in G \cap P$. We know that $tag(\pi_1) \leq obj_2.tag$ when π_1 completes, since an invocation during the put phase ensure that $obj_2.tag$ is at least $tag(\pi_1)$. And we know that $obj_2.tag \leq tag(\rho)$ when ρ begins.

At this point, however, we do not know which event came first: the invocation during the put phase of π_1 or the response during the get phase of ρ.

Since obj_2 is an atomic object, it must process these two invocations – doing perform steps in the canonical automaton – in one order or the other. Assume that obj_2 processes the invocation by ρ first, that is, the perform step in response to ρ precedes the perform step in response to π_1. In this case, the response to π_1 includes a configuration identifier no smaller than $get\text{-}conf\text{-}id(\pi_2)$, the configuration being installed by ρ. As a result:

$$put\text{-}conf\text{-}id(\pi_1) \geq get\text{-}conf\text{-}id(\pi_2) \ .$$

This contradicts our assumption that the put configuration identifier of π_1 is less than the get configuration identifier for π_2.

Therefore, we can conclude that the invocation of obj_2 for π_1 precedes the response of obj_2 for π_2. Therefore, $tag(\pi_1) \leq tag(\rho)$. Combining the two inequalities, we conclude that

$$tag(\pi_1) \leq tag(\pi_2) \ ,$$

which implies that $\pi_2 \not\prec \pi_1$. □

Case 3: One-Phase Read Operations. We now address the case of single-phase read operations. We assume that π_1 is a one-phase read operation, that is, it does not have a put phase.

Notice that in Lemma 6.12, we depended significantly on π_1 propagating its tag to a put-quorum of objects. Since a one-phase read operation has no put phase, we cannot use this.

Instead, we rely on the fact that a one-phase read operation, π occurs only when the tag of operation π is *confirmed*, indicating that another two-phase operation, π' has already propagated the tag of π to a put-quorum.

We first need a lemma showing that if a tag, t, is confirmed, then there exists a two-phase operation that propagated tag t to a put-quorum.

Lemma 6.13 *Let α' be a prefix of α, and assume that at some node i, at the end of α', the tag $t \in confirmed_i$. Then there exists an operation, π, in α' such that $tag(\pi) = t$ and π is either a two phase read-operation or a write operation.*

Proof. Without loss of generality, assume that α' is the shortest prefix of α such that at the end of α', for some node i, $t \in confirmed_i$.

There are two ways in which a tag can be added to the confirmed set of i: either a response from some object indicates that a tag is confirmed (see Figure 11, Line 54), or i itself completes a two-phase operation and adds t to the confirmed set (see Figure 12, Lines 36 and 17).

In the first case, this implies that there exists some object, obj, that has $t \in obj.confirmed\text{-}set$ at some point in α'. However, this would imply that some other node k had performed a confirm invocation on obj (see Figure 10, Line 27) in α', as this is the only way in which a tag can be added to an object's confirmed set.

This, then, implies that $t \in confirmed_k$ in α', when the confirm invocation occurs. This violates the assumption that α' is the shortest prefix to end with some node, i, containing t in $confirmed_i$.

Therefore, node i must perform a read-ack$_i$ or write-ack$_i$ in α' that adds t to $confirmed_i$ (see Figure 12, Lines 36 and 17). The value $op.tag_i$ must be equal to t, because t is added to $confirmed_i$. Further, if the operation is a read operation, then a precondition of the read-ack$_i$ is that $op.phase =$ put, implying that it is a two-phase read operation. The node i, then, satisfies all the required properties of node j. □

Now we can show that with one-phase reads, the partial ordering induced by the tags is consistent with the real ordering of the operations:

Lemma 6.14 (Case 3) *Assume operation π_1 is a one-phase read operation, and occurs at node i. Assume that π_2 is a read operation initiated at node j, and that π_1 completes before π_2 begins. Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result $\pi_2 \not\prec \pi_1$.*

Proof. Since π_1 is a one-phase read operation, $tag(\pi_1) \in confirmed_i$ when the read-ack$_i$ event occurs (see Figure 12, Line 63). Recall that $tag(\pi_1)$ is the value of tag_i when the read-ack$_i$ event occurs.

By Lemma 6.13, a two-phase operation π' must complete prior to the read-ack$_i$ event of π_1 and $tag(\pi_1) = tag(\pi')$, the tag_i when the read-ack$_i$ event occurs.

Since π' completes before the end of π_1, it also completes before π_2 begins. Therefore, by Lemma 6.12, $tag(\pi') \leq tag(\pi_2)$; as a result, $tag(\pi_1) \leq tag(\pi_2)$. □

Main Result. Combining the lemmas from all the cases (Lemma 6.7, Lemma 6.8, Lemma 6.9, Lemma 6.11, Lemma 6.12, and Lemma 6.14), we conclude:

Theorem 6.15 *If π_1 and π_2 are two operations, and π_1 completes before π_2 begins, then $\pi_2 \not\prec \pi_1$.*

We now claim that the Operation Manager, composed with the focal point objects, the ReconClients, and a well-formed environment, implements a read/write atomic object.

Theorem 6.16 *Let U be a well-formed environment. Let S be the composition of the Operation Manager, the focal point objects, the ReconClients and the environment, U, where all input and output actions are hidden except for* read, read-ack, write, *and* write-ack. *Let A be the canonical atomic read/write object, an object of the variable type presented in Figure 5. Then:*

$$traces(S \circ U) \subseteq traces(A \circ U)) \ .$$

Proof. First, notice that S has the appropriate input and output actions. Next, we go through the three conditions required by Theorem 3.2:

1. Follows from the uniqueness of the tags: each is chosen by examining the local clock, and using process identifiers to break ties.

2. Follows from Theorem 6.15.

3. Follows from the way in which the partial order is defined, as a read operation is ordered directly after the write operation whose value it returns.

□

7 Focal Point Emulator

In this section we present an algorithm to implement the Focal Point Object Model. The algorithm allows mobile nodes moving in and out of focal points, communicating with distributed clients through the GeoCast service, to implement

Signature:

Input:
geocast-rcv($\langle$invoke, *inv*, *oid*, *loc*$\rangle)_{obj,i}$, *inv* $\in$ *invocations*, *oid* $\in U$, *loc* $\in L$
lbcast-rcv($\langle$join-req, *jid*$\rangle)_{obj,i}$, *jid* $\in T$
lbcast-rcv($\langle$join-ack, *jid*, *v*$\rangle)_{obj,i}$, *jid* $\in T$, $v \in V$
lbcast-rcv($\langle$invoke, *inv*, *oid*, *loc*$\rangle)_{obj,i}$, *inv* $\in$ *invocations*, *oid* $\in U$, *loc* $\in L$
geo-update$(l, t)_{obj,i}$, $l \in L$, $t \in R^{>0}$

Output:
geocast($\langle$response, *resp*, *oid*, *loc*$\rangle)_{obj,i}$, *resp* $\in$ *responses*, *oid* $\in U$, *loc* $\in L$
lbcast($\langle$join-req, *jid*$\rangle)_{obj,i}$, *jid* $\in T$
lbcast($\langle$join-ack, *jid*, *v*$\rangle)_{obj,i}$, *jid* $\in T$, $v \in V$
lbcast($\langle$invoke, *inv*, *oid*, *loc*$\rangle)_{obj,i}$, *inv* $\in$ *invocations*, *oid* $\in U$, *loc* $\in L$

Internal:
join$()_{obj,i}$
leave$()_{obj,i}$
simulate-op$(inv)_{obj,i}$, *inv* $\in$ *invocations*

State:
fp-location $\in 2^L$, constant, locations defining the focal point under consideration
clock $\in R^{\geq 0}$, the current time, initially 0, updated by the geosensor
location $\in L$, the current physical location, updated by the geosensor
status $\in$ {idle,joining,listening,active}, initially active if node is in $fp\text{-}location$ and idle otherwise
join-id $\in T$, unique id for current join request, initially $\langle 0, i_0 \rangle$
lbcast-queue, a queue of messages to be sent by the LBcast, initially $\emptyset$
geocast-queue, a queue of messages to be sent by the GeoCast, initially $\emptyset$
anwered-join-reqs, set of ids of join requests that have already been answered, initially $\emptyset$
val $\in V$, holds current value of the simulated atomic object, initially v_0
pending-ops, queue of operations waiting to be simulated, initially $\emptyset$
completed-ops, queue of operations that have been simulated, initially $\emptyset$

Figure 13: FPE Server Signature and State for Node i and Object obj of variable type $\tau = \langle V, v_0, invocations, responses, \delta \rangle$

an atomic object (with port set $Q = S$) corresponding to a particular focal point. We refer to this algorithm as the Focal Point Emulator (FPE).

Figure 15 depicts the various components of the FPE. The clients (on the left) send invocations and receive responses from the FPE Client, which simply attaches a tag to every request and then broadcasts the request (using the GeoCast service) to a focal point, where it is received by multiple FPE Servers. The FPE Servers then coordinate among themselves, using the LBcast service to determine an ordering of the requests. They then update their local replicas of the data object, and broadcast a response back. The FPE Client then removes duplicates, and sends the response to the client. The FPE Client runs on every mobile node that wants to access this particular atomic object; the FPE Server runs on every node and is active when that node is in the focal point corresponding to the atomic object.

FPE Client. The code for the FPE Client is presented in Figure 16. The FPE Client has three basic purposes. First, it ensures that each invocation receives at most one response (eliminating duplicates). Second, it abstracts away the GeoCast communication, providing a simple invoke/respond interface to the mobile node. Third, it provides each mobile node with multiple ports to the focal point object; the number of ports depends on the atomic object being implemented.

FPE Server. The FPE Server is the automaton that allows the mobile nodes in a focal point to simulate a single replica. The FPE Server implements a replicated state machine, using the totally ordered local broadcast to ensure consistency. Figure 13 contains the signature and state of the FPE. The remaining code for the FPE Server is in Figure 17.

When a node enters the focal point, it broadcasts a join-request message using the LBcast service and waits for a response. The other nodes in the focal point respond to a join-request by sending the current state of the simulated object using the LBcast service. As an optimization, to avoid unnecessary message traffic and collisions, if a node observes that someone else has already responded to a join-request, then it does not respond. Once a node has received the response to its join-request, then it starts participating in the simulation, by becoming active.

When a node receives a GeoCast message containing an operation invocation, it resends it with the LBcast service to the focal point, thus causing the invocation to become ordered with respect to the other LBcast messages (which are join-request messages, responses to join requests, and operation invocations). Since it is possible that a GeoCast is received by more than one node in the focal point, there is some bookkeeping to make sure that only one copy of the same invocation is actually processed by the nodes. We include an optimization that if a node observes that an invocation has already been sent with the LBcast service, then it does not do so.

Active nodes keep track of operation invocations in the order in which they receive them over the LBcast service. Duplicates are discarded using the unique operation ids. The operations are performed on the simulated state in order. After each one, a GeoCast is sent back to the invoking node with the response. Operations complete when the invoking node remains in the same region as when it sent the invocation, allowing the GeoCast to find it.

When a node leaves the focal point, it re-initializes its variables.

A subtle point is to decide when a node should start collecting invocations to be applied to its replica of the object state. A node receives a snapshot of the state when it joins. However by the time the snapshot is received, it might be out of date, since there may have been some intervening messages from the LBcast service that have been received since the snapshot was sent. Therefore the joining node must record all the operation invocations that are broadcast after its join request was broadcast but before it received the snapshot. This is accomplished by having the joining node enter a "listening" state once it receives its own join-request message; all invocations received when a node is in either the listening or the active state are recorded, and actual processing of the invocations can start once the node has received the snapshot and has the active status.

A precondition for performing most of these actions is that the node is in the relevant focal point. This property is covered in most cases by the integrity requirements of the LBcast and GeoCast services, which imply that these actions only happen when the node is in the appropriate focal point.

8 The FPE Implements the Focal Point Object Model

In this section, we show that the Focal Point Emulator correctly implements the Focal Point Object Model. We focus on a single focal point, interacting with some environment. Since atomic objects can be composed, it suffices to show that the focal point implements an atomic object. Fix obj to be this focal point object.

The body of the proof consists primarily of determining a total ordering on all operations of the atomic object

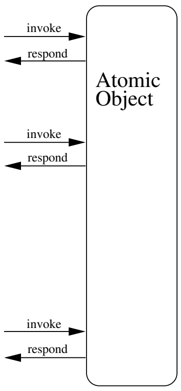

Figure 14: Individual Focal Point Object, which implements an atomic object that responds to invocations.

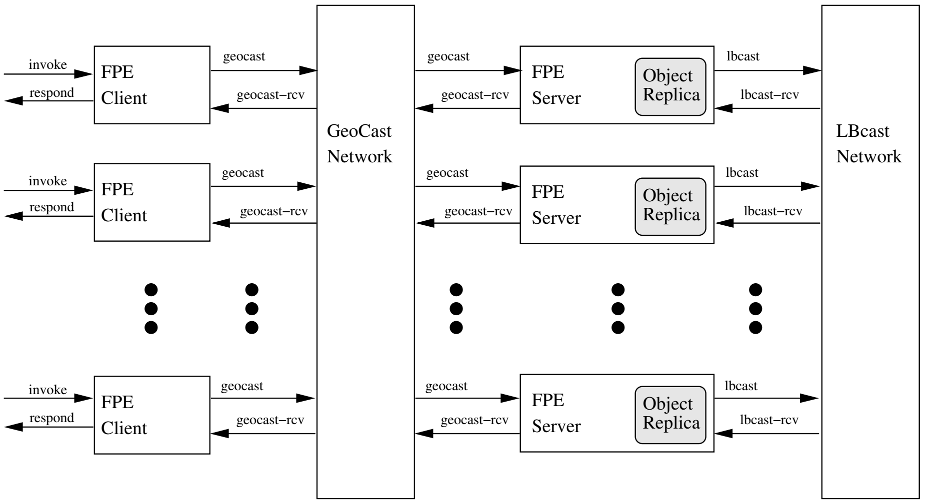

Figure 15: Components of the Focal Point Emulator. When an FPE Client (on the left) receives an invocation, it sens the request to all the FPE Servers using the GeoCast service. At least one of the FPE Servers broadcasts the request to all the other FPE Servers in the focal point, using the LBcast service. Each FPE Server then updates its Object Replica as a result of the request. At this point, at least one FPE Server send a response to the FPE Client, using the GeoCast service.

for an arbitrary execution in which all the operations complete, as per Theorem 3.1. Let α be some such execution. The total ordering is shown to have the necessary properties, allowing us to conclude that the algorithm correctly implements an atomic object. In particular, we need to show that the devised ordering is consistent with the real-time ordering of operations, and that the ordering is consistent with the responses sent during the execution.

Let A be the abstract system consisting of an atomic object (of an arbitrary variable type). See Figure 14. Let S be the system consisting of a set of FPE Clients, composed with the GeoCast network, composed with an FPE Server, each containing an object replica. See Figure 15.

Let U be a well-formed environment. Our goal, then, is to show that $traces(S \circ U) \subseteq traces(A \circ U)$. We consider an arbitrary execution, α, of the system $S \circ U$ in which every operation completes. Let Π be the set of operations in α. Since every operation completes, we know that for every $\pi \in \Pi$, for some $p \in P$, an $\mathsf{invoke}(\pi)_{obj,p}$ event occurs, followed by a $\mathsf{respond}(\pi)_{obj,p}$ event.

Preliminaries. We now define some preliminary notation, and then specify a total ordering of operations in execution α on the atomic object.

Each operation, $\pi \in \Pi$, is assigned a unique identifier by the FPE Client before it is GeoCast to the FPE Servers (see Figure 16, Line 3). We refer to this identifier as $id(\pi)$.

The LBcast service guarantees a total ordering on all messages sent within a focal point. For any execution, or prefix of an execution, β, let $IM(\beta)$ be the set of invoke messages sent (and later delivered) by the LBcast service in β.

(For the purposes of discussion, we will consider a "message" to consist of the tuple: $\langle inv, oid, loc \rangle$, the last three parameters in a $\mathsf{lbcast}(\langle \mathsf{invoke}, inv, oid, loc \rangle)$ action or in a $\mathsf{lbcast\text{-}rcv}(\langle \mathsf{invoke}, inv, oid, loc \rangle)$ action whose first parameter is invoke.) T

he set $IM(\beta)$ contains one message for every $\mathsf{lbcast}(m)$ event in β. Notice that this same message is delivered by multiple $\mathsf{lbcast\text{-}rcv}(m)_i$ events, each at a different node i.

The definition of the LBcast service guarantees that there exists a total ordering of the messages in $IM(\alpha)$ that is consistent with the order in which each mobile node receives the messages. That is, if node i receives two messages m_r and m_t in that order, then $m_r \prec m_t$ in this ordering.

Each of the invoke messages sent by the LBcast service include an identifier, oid. Let $id(m)$ be the identifier associated with message m. This identifier is closely related to the identifier associated with operation, $id(\pi)$ that led to this message.

For all of the invoke messages, the oid is set to the unique identifier of an operation associated with that message. (See Figure 17, Line 8, and notice that the enqueued LBcast message uses the identifier from the received GeoCast request). This allows us to say that certain LBcast messages are associated with each operation. However, it also means that these identifiers are not unique: two LBcast messages may have the same identifier, because the two messages are created as a result of the same operation.

We define a total ordering on operations $\pi \in \Pi$ as follows, using the ordering determined by the LBcast service. Let π_i and π_j be two operations in Π. For operation π_i, let m_1 be the first message in $IM(\alpha)$, the LBcast messages sent in α, associated with operation π_i: $id(m_1) = id(\pi_i)$. The message m_1 is the first message delivered by LBcast in α that was generated by operation π_i.

Similarly, for operation π_j, let m_2 be the first message in $IM(\alpha)$ such that $id(m_2) = id(\pi_j)$. We say that $\pi_i \prec \pi_j$ if $m_1 \prec m_2$. Since operation identifiers are unique, and an operation can only be associated in this way with a single message (i.e., the first), this defines a total ordering.

Properties of the Total Order. The most difficult property of Theorem 3.1 to show is Property 2. We need to show that the total ordering is consistent with the responses sent by the FPE Servers.

Let α' be any finite prefix of execution α, and let i be any node. Of all the LBcast messages ordered in $IM(\alpha')$, choose m to be the largest message received by node i in α' and not in $pending\text{-}ops_i$ at the end of α'. More formally, choose m such that:

1. $\forall m' \in IM(\alpha'),\ m' \prec m$
2. $\mathsf{lbcast\text{-}rcv}(m)_i$ occurs in α'
3. $m \notin \ell state(\alpha').pending\text{-}ops_i$.

Here $\ell state(\alpha')$ denotes the last state in α'. The message m is, in effect, the most recent message that has been processed by node i (if i has completed the join protocol): all prior messages have been received, added to $pending\text{-}ops_i$, and removed; all later messages are in $pending\text{-}ops_i$, or have not yet been received.

We define $\gamma(\alpha', i)$ to be the state of the atomic object after beginning in the initial state, v_0, and processing all the invoke messages for node i in $IM(\alpha')$, stopping at message m (while skipping the "duplicate" messages, referring to the same operation, that might occur in $IM(\alpha')$). In particular, this means that $\gamma(\alpha', i)$ is the state after processing the operations, $\pi_1, \pi_2, \ldots, \pi_t$, where π_t is the most recent operation that i has processed in α'.

We show that for all prefixes α' of execution α, if node i has completed the join protocol, then the state of the replica at node i is equal to $\gamma(\alpha', i)$. That is, the state of the replica at node i is consistent with all prior operations in $IM(\alpha')$ having occurred. If node i has itself received all the messages in $IM(\alpha')$, this claim is immediate. In the case that node i has joined the focal point during the execution, however, node i may have only received some suffix of the sequence. As a result, the main difficulty in proving the invariant is showing that the join protocol works, i.e. that after a node sets its status to active, it has correctly acquired a good snapshot of the state of the world.

Invariant 1 *Let α' be any finite prefix of execution α. If:*

$$\ell state(\alpha').status_i = \mathsf{active}\ ,$$

then:

$$\ell state(\alpha').val_i = \gamma(\alpha', i)\ .$$

Proof. We show this by induction on the length of α'. For the base case, consider the initial state of the system, before any actions occur in α. Let β' be this empty prefix of α. If i is not in any focal point, then $status_i = \mathsf{idle}$, and the result is trivial. If i is in some focal point, then the state of val_i is v_0. However, in this case $IM(\beta')$ is empty, and as a result $\gamma(\beta', i)$ also equals v_0.

For the inductive step, let s and s' be the states before and after the new event, respectively. Let β be the previous prefix of α, that is, $s = \ell state(\beta)$. Let β' be the new prefix of α, that is, $s' = \ell state(\beta')$.

We know, inductively, that for any finite prefix α' of β, for any node j, if $\ell state(\alpha').status_j = \mathsf{active}$, then:

$$\ell state(\alpha').val_j = \gamma(\alpha', j))\ .$$

We need to show that

$$\ell state(\beta').val_i = \gamma(\beta', i) \; .$$

We now consider the various actions relevant to this claim.

- $\mathsf{lbcast\text{-}rcv}(\mathsf{invoke}, inv, oid, loc)_i$: (See Figure 17, Lines 24–28.)
  Recall that the sequence $IM(\beta', i)$ only includes messages that have been sent in β', received by i, and are no longer in $pending\text{-}ops_i$. As a result of this action, the message, m, is added to $pending\text{-}ops_i$, and therefore the sequence $IM(\beta', i)$ is equal to $IM(\beta, i)$, and as a result, $\gamma(\beta', i) = \gamma(\beta, i)$. By induction, we already know that $\gamma(\beta, i) = \ell state(\beta).val_i$. The state of the replicated object is also unchanged, that is:

  $$\ell state(\beta').val_i = \ell state(\beta).val_i \; .$$

  The result then follows.

- $\mathsf{simulate\text{-}op}(inv)_i$: (See Figure 17, Lines 36–43.)
  First, when this action occurs, $status_i = \mathsf{active}$, since that is a precondition to this action (see Figure 17, Line 38). The invoke operation removes the message from the set $pending\text{-}ops_i$, and therefore adds the associated invocation to $IM(\beta, i)$. That is, $IM(\beta', i) = IM(\beta, i) \cup \{m\}$. Therefore:

  $$\gamma(\beta', i) = \delta(m, \gamma(\beta, i)) \; ,$$

  since the total ordering of the LBcast service ensures that the message m is ordered after every message in $IM(\beta, i)$.

  The state of val_i was initially $\gamma(\beta, i)$ in state s, by induction. After this action, val_i is set to:

  $$\delta(m, \gamma(\beta, i)) \; ,$$

  as this is the definition of how the object responds to invocations. This maintains the desired invariant.

- leave_i: (See Figure 17, Line 45.)
  In this case, $status_i \leftarrow \mathsf{idle}$, and the claim is trivially true.

- $\mathsf{lbcast\text{-}rcv}(\mathsf{join\text{-}ack}, jid, v)_i$: (See Figure 17, Lines 17–22.)
  This is the main interesting case of this proof. In this action, node i sets $status$ to active (see Figure 17, Line 21), if the message is a response to an outstanding join-req previously sent by i. (If this is not the case, then this action causes no change to $status_i$, $pending\text{-}ops_i$, or val_i, and the invariant is trivially maintained.)

  We know, then, that some node, j, must have previously performed an $\mathsf{lbcast\text{-}send}_j$ action to send a copy of the state of val_j to i. More specifically, an

  $$\mathsf{lbcast\text{-}rcv}(\mathsf{join\text{-}req}, jid)_j$$

  action occurs, causing the message $\langle \mathsf{join\text{-}ack}, jid, val_j \rangle$ to be added to the LBcast queue. This then leads to an lbcast of this message, which is received by this action. The LBcast service guarantees that every message received was earlier sent, and the message existing in the queue $lbcast\text{-}queue$ ensures that the message was received. Let β'' be the prefix of β ending with the

  $$\mathsf{lbcast\text{-}rcv}(\mathsf{join\text{-}req}, jid)_j$$

  action.

  Inductively, we know that the state of the replicated object sent by j (i.e., the current value v of val_j) is equal to $\gamma(\beta'', j)$.

  Since val_i is set to v when the join-ack message is received, we only need to show that:

  $$\gamma(\beta', i) = \gamma(\beta'', j) \; , \tag{1}$$

  and we can conclude that $val_i = \gamma(\beta', i)$, as desired.

Let m be the largest message in $IM(\beta')$ received by i that is not in $s'.pending\text{-}ops_i$. Every message ordered by the LBcast service prior to i's join request precedes m. This is because i must receive its own join request, as it entered the focal point prior to sending the join request, and i does not respond to its own join request.

Next, every message that i receives in β' that is ordered after i's join request comes after m in the LBcast message ordering. This is because when i receives its own join request, it sets its $status_i$ to listening (see Figure 17, Line 13). Every message received after the join request is therefore added to $pending\text{-}ops_i$ (as $status_i$ is listening). No messages are removed from $pending\text{-}ops$, as all such actions have $status_i = \mathsf{active}$ as a precondition (see Figure 17, Line 38).

Therefore m must correspond to i's join request. As a result, $\gamma(\beta', i)$ is the state reached after processing all messages prior to m, i.e., prior to i's join request.

Notice, though, that i's join request is exactly the last message processed by j in β'' before sending a response to j. In particular, then, $\gamma(\beta'', j)$ is the state reached on processing every message in $IM(\beta''$ prior to i's join request.

Therefore, Equation 1 holds, and the lemma holds in state s'.

The rest of the cases are straightforward, having no effect on the status of i, the elements of $pending\text{-}ops_i$, or the state of the replicated object. ☐

We can now show that the two properties required by Theorem 3.1 hold for the total ordering we have defined, and as a result, the Focal Point Emulator, S, is a correct implementation of the Focal Point Object Model, A.

Theorem 8.1 *For all executions, $traces(S \circ U) \subseteq traces(A \circ U)$.*

Proof. We consider the properties from Theorem 3.1 in order:

1. This follows from Invariant 1. Let $\pi \in \Pi$ be an operation. The client that performs the response for π does so because it receives a GeoCast with a response for π from a server, say i, for the first time. Let α' be the prefix of α ending just before the simulate-op action that caused i to enqueue the GeoCast response. The value v determined by starting in state v_0 and handling all the invoke(π_r) operations prior to π is exactly $\gamma(\alpha', i)$: every preceding operation is associated with an earlier LBcast message (by the way in which the total ordering is defined); at the time when node i invokes operation π, it has removed all prior messages from the $pending\text{-}ops$ queue, and therefore the message associated with operation π is exactly the largest message that i has received in α' that is not in $pending\text{-}ops_i$. Invariant 1 shows that the state of val_i prior to operation π is equal to $\gamma(\alpha', i)$. Therefore the response to operation π is exactly that obtained by applying π to $\gamma(\alpha', i)$.

2. Let message m_1 be the first message associated with operation π_i, and let message m_2 be the first message associated with operation π_j. Since π_i completes before π_j begins, the message m_1 must be received before the message m_2 is sent, and therefore m_1 precedes m_2. This then implies that π_i precedes π_j, as needed.

Therefore this theorem follows from Theorem 3.1. ☐

9 Performance Discussion

The performance of the GeoQuorums algorithm stems from the performance of the Operation Manager and the performance of the Focal Point Emulator.

We first examine the performance of read and write operations, as executed by the Operation Manager. Assume that the Focal Point Object Model guarantees that all invocations result in a response within time T. Then if no more than f focal point objects fail, each read or write operations takes at most time $2T$: each operation requires at most two phases, and each phase can be completed in time T, as all the objects can be invoked concurrently. A write operation, however requires on a single phase. Similarly, many read operations only require a single phase. These operations requires at most time T. Similarly, a reconfiguration operation takes at most time $2T$.

We next discuss the performance of the Focal Point Emulator to determine the maximum time required for an invocation to receive a response. We consider a focal point object that does not fail, and we assume that a mobile node

Signature:

Input:
$\mathsf{invoke}(\mathit{inv})_{obj,p}$, $\mathit{inv} \in \mathit{invocations}$, $p \in Q$
$\mathsf{geocast\text{-}rcv}(\langle \mathsf{response}, \mathit{resp}, \mathit{oid}, \mathit{loc}\rangle)_{obj,i}$, $\mathit{resp} \in \mathit{responses}$, $\mathit{oid} \in U$
$\mathsf{geo\text{-}update}(l, t)_{obj,i}$, $l \in L$, $t \in R^{>0}$

Output:
$\mathsf{geocast}(m)_{obj,i}$, $m \in \mathsf{invoke} \times \mathit{invocations} \times U \times L \times L$
$\mathsf{respond}(\mathit{resp})_{obj,p}$, $\mathit{resp} \in \mathit{responses}$, $p \in Q$

State:
$\mathit{fp\text{-}location} \in 2^L$, a constant, the locations of the focal point
$\mathit{clock} \in R^{\geq 0}$, the current time, initially 0, updated by the geosensor
$\mathit{location} \in L$, the current physical location, updated by the geosensor
$\mathit{ready\text{-}responses} \subseteq Q \times \mathit{responses}$, a set of an operation responses, initially $\emptyset$
geocast-queue, a queue of messages to be geocast
$\mathit{ongoing\text{-}oids} \subseteq U$, a set of operation identifiers, initially $\emptyset$

Transitions:

Input $\mathsf{invoke}(\mathit{inv})_{obj,p}$
Effect:
  $\mathit{new\text{-}oid} \leftarrow \langle \mathit{clock}, p\rangle$
  Enqueue$(\mathit{geocast\text{-}queue}, \langle \mathsf{invoke}, \mathit{inv}, \mathit{new\text{-}oid}, \mathit{location}, \mathit{fp\text{-}location}\rangle)$
  $\mathit{ongoing\text{-}oids} \leftarrow \mathit{ongoing\text{-}oids} \cup \{\mathit{new\text{-}oid}\}$

Input $\mathsf{geocast\text{-}rcv}(\langle \mathsf{response}, \mathit{resp}, \mathit{oid}, \mathit{loc}\rangle)_{obj,i}$
Effect:
  if $(\mathit{oid} \in \mathit{ongoing\text{-}oids})$ **then**
    $\langle c, p\rangle \leftarrow \mathit{oid}$
    $\mathit{ready\text{-}responses} \leftarrow \mathit{ready\text{-}responses} \cup \{\langle p, \mathit{resp}\rangle\}$
    $\mathit{ongoing\text{-}oids} \leftarrow \mathit{ongoing\text{-}oids} \setminus \{\mathit{oid}\}$

Input $\mathsf{geo\text{-}update}(l, t)_{obj,i}$
Effect:
  $\mathit{location} \leftarrow l$
  $\mathit{clock} \leftarrow t$

Output $\mathsf{geocast}(m)_{obj,i}$
Precondition:
  Peek$(\mathit{geocast\text{-}queue}) = m$
Effect:
  Dequeue$(\mathit{geocast\text{-}queue})$

Output $\mathsf{respond}(\mathit{resp})_{obj,p}$
Precondition:
  $\langle p, \mathit{resp}\rangle \in \mathit{ready\text{-}responses}$
Effect:
  $\mathit{ready\text{-}responses} \leftarrow \mathit{ready\text{-}responses} \setminus \{\langle p, \mathit{resp}\rangle\}$

Figure 16: FPE Client for Client i and Object obj of variable type $\tau = \langle V, v_0, \mathit{invocations}, \mathit{responses}, \delta\rangle$

Focal Point Emulator Server Transitions

```
Internal join()_{obj,i}
Precondition:
    location ∈ fp-location
    status = idle
Effect:
    join-id ← ⟨clock, i⟩
    status ← joining
    Enqueue(lbcast-queue, ⟨join-req, join-id⟩)

Input lbcast-rcv(⟨join-req, jid⟩)_{obj,i}
Effect:
    if ((status = joining) ∧ (jid = join-id)) then
        status ← listening
    if ((status = active) ∧ (jid ∉ answered-join-reqs)) then
        Enqueue(lbcast-queue,⟨join-ack,jid,val⟩)

Input lbcast-rcv(⟨join-ack, jid, v⟩)_{obj,i}
Effect:
    answered-join-reqs ← answered-join-reqs ∪ {jid}
    if ((status = listening) ∧ (jid = join-id)) then
        status ← active
        val ← v

Input geocast-rcv(⟨invoke, inv, oid, loc, fp-loc⟩)_{obj,i}
Effect:
    if (fp-loc = fp-location) then
        if (⟨inv, oid, loc⟩ ∉ pending-ops ∪ completed-ops) then
            Enqueue(lbcast-queue, ⟨invoke, inv, oid, loc⟩)

Input lbcast-rcv(⟨invoke, inv, oid, loc⟩)_{obj,i}
Effect:
    if ((status = listening ∨ active) ∧
        (⟨inv, oid, loc⟩ ∉ pending-ops ∪ completed-ops)) then
            Enqueue(pending-ops, ⟨inv, oid, loc⟩)

Internal simulate-op(inv)_{obj,i}
Precondition:
    status = active
    Peek(pending-ops) = ⟨inv, oid, loc⟩
Effect:
    (val,resp) ← δ(inv,val)
    Enqueue(geocast-queue, ⟨response, resp, oid, loc⟩)
    Enqueue(completed-ops, Dequeue(pending-ops))

Internal leave()_{obj,i}
Precondition:
    location ∉ fp-location
    status ≠ idle
Effect:
    status ← idle
    join-id ← ⟨0, i_0⟩
    val ← v_0
    answered-join-reqs ← ∅
    pending-ops ← ∅
    completed-ops ← ∅
    lbcast-queue ← ∅
    geocast-queue ← ∅

Output lbcast(m)_{obj,i}
Precondition:
    Peek(lbcast-queue) = m
Effect:
    Dequeue(lbcast-queue)

Output geocast(m)_{obj,i}
Precondition:
    Peek(geocast-queue) = m
Effect:
    Dequeue(geocast-queue)

Input geo-update(l, t)_{obj,i}
Effect:
    location ← l
    clock ← t
```

Figure 17: FPE Server Transitions for Client i and Object obj of variable type $\tau = \langle V, v_0, invocations, responses, \delta \rangle$

performing an invocation does not move too far until a response is received. In particular, such a mobile node moves no further than distance R, as defined by the GeoCast service.

The performance of the focal point objects is directly dependent on the performance of the two communication services. Assume that every GeoCast message is delivered within time d_G, and every LBcast message is delivered within time d_{LB}; let $d = d_G + d_{LB}$. Then every invocation receives a response within time $2d$: each phase takes at most two round-trip messages. (An extra round of communication may be caused by the discovery during the first round that a reconfiguration is in progress.)

We then conclude that if a mobile node, i, initiates a read or write operation, and no more than f focal points fail, and node i does not move further than distance R after the operation begins (until the operation completes), then the operation completes within time $8d$. A write operation, or a one-phase read operation, completes within time $4d$.

The algorithm as specified also allows the implementation to trade-off message complexity and latency. In each phase of a read or write operation, the node initiating the operation must perform invocations on a quorum of focal point objects; each invocation is going to cause message traffic in the network. It can achieve this goal by performing invocations on all focal point object concurrently, thereby ensuring the fastest result, at the expense of a high message complexity. Alternatively, the node can invoke only the focal point objects in a single quorum. If some of these focal point objects have failed, and they do not all respond, the node can perform invocations on another quorum, and continue until it receives a response from every object in some quorum. This leads to lower message complexity, but may take longer.

10 Conclusions and Future Work

We have presented a new approach, the GeoQuorums approach, to implementing algorithms in mobile *ad hoc* networks. We have presented a geographic abstraction model, the Focal Point Object Model, and an algorithm, the Focal Point Emulator, that implements it using mobile nodes. We have presented the Operation Manager, which uses the static model to implement an efficient, reconfigurable atomic read/write memory.

The GeoQuorums approach transforms a highly dynamic, *ad hoc* environment into a static setting. This approach should facilitate the adaptation of classical distributed algorithms to *ad hoc* networks.

We also believe that our approach will be useful in studying hybrid networks, consisting of both mobile nodes and fixed infrastructure. In areas where there are non-mobile, fixed participants, simpler and more efficient versions of the FPE can be used. When nodes enter areas with no infrastructure, the more dynamic algorithm can seamlessly take over.

A major open area of research, then, is to determine other uses of the Focal Point Object Model, and other algorithms that rely on focal point objects. For example, it seems possible to build mobile routing services using focal point objects to maintain routes. Similarly, the use of the Focal Point Object Model should facilitate the design of resource allocation and task allocation algorithms.

There are many open questions relating to the geographic abstraction itself.

It would be a natural extension to allow focal points to recover. Currently, once a focal point object fails, it cannot recover, even when new mobile nodes reenter it. Allowing a focal point to recover would improve the fault-tolerance of the system.

We would also like to determine whether we can implement something more general than atomic objects using focal points. For example, it seems possible to implement arbitrary automata using a similar technique to that of the Focal Point Emulator.

It is an interesting question to consider the implementation of the $ReconClient$ automata. Since there are only a finite number of configurations to choose from, it should be possible for the mobile nodes to determine which configuration is optimal for a given set of read and write operations. Using the techniques of competitive analysis it may be possible to determine the optimal configuration for a sequence of read and write operations, even without knowing the sequence in advance.

It would also be interesting to consider other implementations of the Focal Point Object Model. The implementation presented in this paper has two major drawbacks. First, it requires the participation of every mobile node in a focal point. It is easy to relax this requirement, by allowing a mobile node to choose not to participate, as long as it is sure that other nodes are participating. It may improve performance and energy efficiency to allow only a small number of mobile nodes – in particular, those that tend to reside the most within the focal point – to maintain the focal point. If a very reliable node remains continuously in the focal point (for example, a fixed, non-mobile base station), then there is no reason to burden too many other nodes. Second, the Focal Point Emulator relies on a powerful local broadcast service, LBcast. Considering the highly local nature of a focal point, there may be implementations of the Focal Point Object model that rely only on a weaker communication service.

We would therefore like to consider implementations of the Focal Point Object Model that depend on a leader election algorithm to determine a primary copy of the data in a focal point object. It may also be possible to implement a focal point object based on the Rambo algorithms ([25]). This would be useful in settings where global time is not available, or where a local broadcast service was unavailable.

We have assumed a static definition of focal points and configurations. It remains an open question to determine how to choose a good set of focal points, how to construct a map of focal points in a distributed fashion, and how to modify the set of focal points dynamically.

References

[1] Paul Attie, Nancy Lynch, and Sergio Rajsbaum. Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical Report LCS-TR-877, MIT, 2002.

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

[3] Sangeeta Bhattacharya. Randomized location service in mobile ad hoc networks. In *Proceedings of the 8th International ACM Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 66–73, sep 2003.

[4] T. Camp and Y. Liu. An adaptive mesh-based protocol for geocast routing. *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, pages 196–213, 2002.

[5] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[6] Danny Dolev, Idit Keidar, and Esti Yeger Lotem. Dynamic voting for consistent primary components. In *Proc. of the Sixteenth Annual ACM Symp. on Principles of Distributed Computing*, pages 63–71. ACM Press, 1997.

[7] S. Dolev, D. K. Pradhan, and J. L. Welch. Modified tree structure for location management in mobile environments. *Computer Communications: Special Issue on Mobile Computing*, 19(4):335–345, April 1996.

[8] S. Dolev, E. Schiller, and J. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 70–79, 2002.

[9] Shlomi Dolev, Seth Gilbert, Nancy Lynch, Alex Shvartsman, and Jennifer Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceeding of the 17th International Conference on Distributed Computing*, October 2003.

[10] 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.

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

[12] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

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

[14] 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.

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

[16] Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *Trans. on DB Systems*, 12(2):170–194, 1987.

[17] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[18] G. Karumanchi, S. Muralidharan, and R. Prakash. Information dissemination in partitionable mobile ad hoc networks. In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pages 4–13, 1999.

[19] Young Bae Ko and Nitin Vaidya. Geotora: A protocol for geocasting in mobile ad hoc networks. In *Proc. of the IEEE Intl. Conference on Network Protocols*, pages 240–249, November 2000.

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

[21] Hyunyoung Lee, Nitin Vaidya, and Jennifer L. Welch. Location tracking using quorums in mobile ad-hoc networks. *Ad Hoc Networks*, 1(4):371–381, nov 2003.

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

[23] Nancy Lynch, Roberto Segala, and Frits Vaandraager. Hybrid I/O automata. *Information and Computation*, 185(1), August 2003.

[24] Nancy Lynch, Roberto Segala, and Frits Vaandraager. Hybrid I/O automata. Technical Report LCS-TR-827d, MIT, August 2003.

[25] 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.

[26] J. C. Navas and T. Imielinski. Geocast – geographic addressing and routing. In *ACM/IEEE Intl. Conference on Mobile Computing and Networking*, pages 66–76, 1997.

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

[28] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proc. of the 6th ACM MOBICOM*, pages 32–43, August 2000.

[29] I. Stojmenovic and P. E. V. Pena. A scalable quorum based location update scheme for routing in ad hoc wireless networks. Technical Report TR-99-11, Computer Science, SITE, University of Ottawa, December 1999.

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

[31] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, 1987.