Implementing Sequentially Consistent Shared Objects using Broadcast and Point-To-Point Communication

Alan Fekete† M. Frans Kaashoek‡ Nancy Lynch‡

†Department of Computer Science F09 University of Sydney 2006, Australia.

‡ MIT Laboratory for Computer Science Cambridge MA 02139, U.S.A.

Abstract

A distributed algorithm that implements a sequentially consistent collection of shared read/update objects using a combination of broadcast and point-to-point communication is presented and proved correct. This algorithm is a generalization of one used in the Orca shared object system. The algorithm caches objects in the local memory of processors according to application needs; each read operation accesses a single copy of the object, while each update accesses all copies. Copies of all the objects are kept consistent using a strategy based on sequence numbers for broadcasts.

The algorithm is presented in two layers. The lower layer uses the given broadcast and point-to-point communication services, plus sequence numbers, to provide a new communication service called a *context multicast channel*. The higher layer uses a context multicast channel to manage the object replication in a consistent fashion. Both layers and their combination are described and verified formally, using the I/O automaton model for asynchronous concurrent systems.

1 Introduction

In this paper, we present and verify a distributed algorithm that implements a sequentially consistent collection of shared read/update objects using a combination of (reliable, totally ordered) broadcast and (reliable, FIFO) point-to-point communication. This algorithm is a generalization of one used in the implementation of the Orca distributed programming language [7] over the Amoeba distributed operating system [26].

Orca is a language for writing parallel and distributed application programs to run on clusters of workstations, processor pools and massively parallel computers [7, 25]. It provides a simple shared object model in which each object has a state and a set of operations, classified as either *read* operations or *update* operations. Read operations do not modify the object state, while update operations may do so. Each operation involves only a single object and appears to be indivisible.

More precisely, Orca provides a *sequentially consistent* memory model [19]. Informally speaking, a sequentially consistent memory appears to its users as if it were centralized (even though it may be implemented in a distributed fashion). There are several formalizations of the notion of sequentially consistent memory, differing in subtle ways. We use the state machine definition of Afek, Brown and Merritt [2].

Orca runs over the Amoeba operating system [26], which provides two communication services: broadcast and point-to-point communication. Both services provide reliable communication, even in the presence of communication failures. No guarantees are made by Orca if processors fail; therefore, we do not consider

This research was supported in part by ARPA N00014-92-J-4033, NSF 922124-CCR, and ONR-AFOSR F49620-94-1-0199; also by ONR contract N00014-94-1-0985 and a NSF Young Investigator Award.

processor failures either. In addition, the broadcast service promises delivery of the broadcast messages in the same total order at every destination,[1] while the point-to-point service preserves the order of messages between any sender and receiver. The cost of an Amoeba broadcast, in terms of time and amount of communication, is higher than that of a single point-to-point message. Therefore, it is natural to design algorithms so that point-to-point communication is used whenever possible, i.e., when a message is intended for only a single destination, and broadcast is only used when necessary, i.e., when a message must go to several destinations.

In the implementation of Orca, user programs are distributed among the various processors in the system. The user program consists of threads, each of which runs on a single processor. In this paper, we call these threads *clients* of the Orca system. Each processor may support several clients. Shared objects are cached in the local memory of some of the processors. Each read operation by a client accesses a single copy of the object, while each update operation accesses all copies. The underlying broadcast primitive provided by the Amoeba system is used to send messages that must be sent to several destinations — that is, invocations of update operations for objects that have multiple copies. The underlying point-to-point primitive is used to send messages that have only a single destination, that is, invocations of reads from a site without a local copy of the object, invocations of writes for an object that has only single (remote) copy, and responses to all invocations.

An early version of the implementation used the strategy of caching all shared objects at all processors. This strategy yields good performance for an object that has a high read-to-update ratio, since a read operation needs only to access the local copy of the object. The drawback is that updates must be performed at all copies, using an (expensive) broadcast communication. Experience has shown that there are some objects for which this is not the best arrangement. For example, many applications use a *job queue* object to allow clients to share work; the job queue is updated whenever a client appends information to it about a task that needs to be done, and also whenever a client removes a task from the queue in order to begin work on it. Since all accesses to a job queue are updates, total replication is not an efficient strategy in this case.

Because of objects like these, Orca has been re-implemented to allow more flexibility in the placement of copies. The new implementation allows some objects to be totally replicated and others to have only a single copy. Operations on an object with only a single copy can now be done using only point-to-point messages, though broadcast must still be used for updates on replicated objects. The decision about whether or not to replicate an object is made at run time using information generated by the Orca compiler. The details of this decision process, and also performance measurements to show the benefits of not replicating all objects, can be found in [6].

The naive strategy of allowing each read operation to access any copy of the object and each update operation to access all copies is not by itself sufficient to implement a sequentially consistent shared memory. To see why, consider the execution depicted in Figure 1. The example involves 3 processors, P_1, P_2 and P_3, and two objects, x and y. Object x is replicated on all processors, while object y is stored only on P_2. The figure shows the invocation and response messages for an update of y by P_1, and the broadcast invocation messages for an update of x by P_3. In this execution, P_2's read operations indicate that y is updated before x is, while P_1 reads the new value of x before invoking the update of y. In a centralized shared memory, such conflicting observations are impossible; thus this execution violates sequential consistency.

The new version of the Orca algorithm solves this consistency problem using a strategy based on sequence numbers for broadcasts. These broadcast sequence numbers are piggybacked on certain point-to-point messages and are used to determine certain ordering relationships among the messages.

Our original goal was to verify the correctness of the new Orca algorithm. In the early stages of our work,

[1] A broadcast service with such a consistent ordering guarantee is sometimes called a *group communication* service. Although group communication is widely discussed in the systems literature, there is no general agreement on its definition. In this paper, we sidestep the issue by using the term *broadcast* to indicate a communication to all sites in the system, and *multicast* to indicate a communication to a subset of the sites. This terminology does not say whether the service is provided by hardware or software.

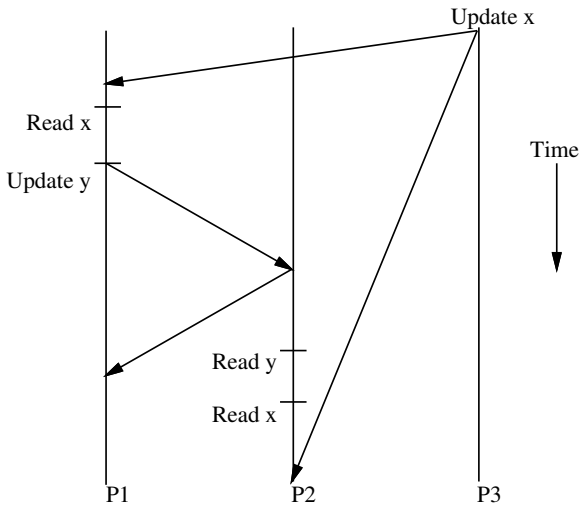

Figure 1: A problem with the naive replication strategy.

however, we discovered a logical error in the implemented algorithm. Namely, broadcast sequence numbers were omitted from some point-to-point messages (the replies returned to the operation invokers) that needed to include them. We produced a corrected version of the algorithm, which has since been incorporated into the Orca system.

The algorithm we study in this paper is our corrected algorithm, generalized beyond what is used in the Orca implementation to allow replication of a shared object at an *arbitrary* collection of processors, rather than just one processor or all processors. There is one way in which our algorithm is less general than the Orca implementation, however: we assume for simplicity that the locations of copies for each object are fixed throughout a program execution, whereas Orca allows these locations to change dynamically, in response to changes in access patterns over time. We discuss the extension of our results to the case of dynamic reconfiguration in Section 7.

We present and verify the algorithm as the composition of two completely separate layers, each a distributed algorithm. The structure of this part of the system is depicted in Figure 2. The lower layer uses the given broadcast and point-to-point communication services, plus broadcast sequence numbers, to implement a new communication service called a *context multicast channel*. A context multicast channel supports multicast of messages to designated subsets of the sites, according to a virtual total ordering of messages that is consistent with the order of message receipt at each site, and consistent with certain restricted "causality" relationships. The guarantees provided by a context multicast channel are weaker than those that are provided by *totally ordered causal multicast channels*, as provided by systems such as Isis [10]. However, the properties of a context multicast channel are sufficiently strong to support the replica management of the Orca algorithm.

The lower layer uses the given point-to-point primitive for each multicast message with a single destination, and the given totally ordered broadcast primitive for each multicast message with more than one destination. (Sites that are not intended recipients simply discard the message.) Sites associate sequence numbers with broadcasts and piggyback the sequence number of the last received broadcast on each point-to-point message. When a point-to-point message reaches its destination, the recipient delays its delivery until the indicated number of broadcasts have been received. (The idea is similar to the one in Lamport's clock synchronization algorithm [18], but we only apply it to a restricted set of events.) We prove that this algorithm correctly implements a context multicast channel.

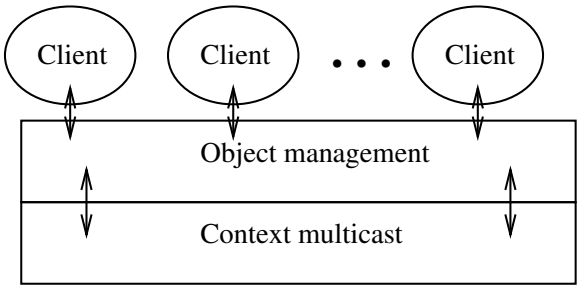

Figure 2: The architecture of the system.

The higher layer uses an arbitrary context multicast channel to manage the object replication in a consistent fashion. Each object is replicated at an arbitrary subset of the sites. A site performs a read operation locally if possible. Otherwise, it sends a request to any site that has a copy and that site returns a response. A site performs an update operation locally if it has the only copy of the object. Otherwise, it sends a multicast message to all sites that have copies, and waits to receive either its own multicast, or else an appropriate response from some other site. We prove that this algorithm, combined with any context multicast system, provides a sequentially consistent memory. Our proof uses a new method based on partial orders.

All our specifications and proofs are presented in terms of the I/O automaton model for asynchronous concurrent systems [23]. General results about the composition of I/O automata allow us to infer the correctness of the complete system from our correctness results for the two separate layers.

Many different correctness conditions have been proposed for shared memory, including strong conditions like memory coherence and weaker ones like release consistency. Sequential consistency is widely used because it appears to be closest to what programmers expect from a shared memory system; non-sequentially consistent shared memory systems typically trade programmability for performance. Sequential consistency was first defined by Lamport [19]; in this paper, we use an alternative formulation proposed by Afek et al. [2], based on I/O automata. Other papers exploring correctness conditions for shared memory and algorithms that implement them include [1, 3, 5, 8, 9, 11, 12, 13, 14, 15, 16, 21, 24]. In most of this work, memory is modeled as a collection of items that are accessed through read and write operations. The study of correctness for shared memory with more general data types was initiated by Herlihy and Wing [17]. Sequential consistency and other consistency conditions for general data types has been studied by Attiya and Welch [5] and Attiya and Friedman [4].

The rest of the paper is organized as follows. Section 2 introduces basic terminology that is used in the rest of the paper. Section 3 contains the definition of a sequentially consistent shared memory and introduces our new method for proving sequential consistency. Section 4 contains definitions of multicast channels with various properties, and in particular, the definition of a context multicast channel. Section 5 contains the higher layer algorithm, which implements sequential consistency using context multicast, plus a proof of its correctness. Section 6 contains the lower layer algorithm, which implements context multicast in terms of broadcast and point-to-point messages. Section 7 contains a discussion of dynamic reconfiguration, and some ideas for future work. Finally, in Section 8 we draw our conclusions.

2 Some Basics

2.1 Partial Orders

We use many partial (and total) orders, on events in executions, and on operations. Throughout the paper, we assume that partial and total orders are *irreflexive*, that is, they do not relate any element to itself. Also, we define a partial or total order P to be *well-founded* provided that each element has only finitely many predecessors in P. This assumption is needed to rule out various technical anomalies.

2.2 I/O Automata

The I/O automaton model is a simple labeled transition system model for asynchronous concurrent systems. An I/O automaton has a set of *states*, including some *start states*. It also has a set of *actions*, classified as *input*, *output* or *internal* actions, and a set of *steps*, each of which is a (state, action, state) triple. Finally, it has a set of *tasks*, each of which consists of a set of output and/or internal actions. Inputs are assumed to be always enabled.

An I/O automaton executes by performing a sequence of steps. An execution is said to be *fair* if each task gets infinitely many chances to perform a step. External behavior of an I/O automaton is defined by the set of *fair traces*, i.e., the sequences of input and output actions that can occur in fair executions.

I/O automata can be *composed*, by identifying actions with the same name. The fair trace semantics is compositional. Output actions of an I/O automaton can also be *hidden*, which means that they are reclassified as internal actions. See [23] for more details.

3 Sequentially Consistent Shared Object Systems

In this section, we define a *sequentially consistent shared object system* and give a new method for proving that a system is sequentially consistent. Informally, a system is said to be a sequentially consistent shared object system if all operations receive responses that are "consistent with" the behavior of a serially-accessed, centralized memory. More precisely, the order of events at each client should be the same as in the centralized system, but the order of events at different clients is allowed to be different.

3.1 The Interface

We start by identifying the actions by which the shared object system interacts with its environment (the *clients*). The shared object system receives *requests* from its environment and responds with *reports*. Requests and reports are of two types: *read* and *update*. Each request and report is subscripted with the name of the client involved. Each request and report contains, as arguments, the name of the object being accessed and a unique *operation identifier*. In addition, each update request contains the function to be applied to the object and each read report contains a return value.[2]

Formally, let C be a fixed finite set of *clients*, X a fixed set of *shared objects*, V a fixed set of *values* for the objects, including a distinguished initial value v_0,[3] and Ξ a fixed set of *operation identifiers*, partitioned into subsets Ξ_c, one for each client c. Then the interface is as follows. (Here, c, ξ, x and v are elements of C, Ξ, X, and V, respectively, and f is a function from V to V.)

Input:

[2]There are two ways in which Orca differs from our specification: in Orca, (1) an *update* may return a value and (2) an *update* might block.

[3]We ignore the possibility of different data domains for the different objects.

request-read$(\xi, x)_c$, $\xi \in \Xi_c$
request-update$(\xi, x, f)_c$, $\xi \in \Xi_c$

Output:
report-read$(\xi, x, v)_c$, $\xi \in \Xi_c$
report-update$(\xi, x)_c$, $\xi \in \Xi_c$

If β is a sequence of actions, we write $\beta|c$ for the subsequence of β consisting of *request-read*$_c$, *request-update*$_c$, *report-read*$_c$ and *report-update*$_c$ actions. This subsequence represents the interactions between client c and the object system.

We assume that invocations are *blocking*: a client does not issue a new request until it has received a report for its previous request. This assumption, and the uniqueness of operation identifiers, are assumptions about the behavior of clients. We express these conditions in the following definition: we say that a sequence β of actions is *client-well-formed* provided that for each client c, no two *request* events[4] in $\beta|c$ contain the same operation identifier ξ, and that $\beta|c$ does not contain two *request* events without an intervening *report* event.

The object systems we describe will generate responses to client requests. Here we define the syntactic properties required of these responses. Namely, we say that a sequence of actions is *complete* provided that there is a one-to-one correspondence between *request* and *report* events such that each *report* follows the corresponding *request* and has the same client, operation identifier, object and type.[5] If a sequence β is client-well-formed and complete, then $\beta|c$ must consist of a sequence of pairs of actions, each of the form *request-read*$(\xi, x)_c$, *report-read*$(\xi, x, v)_c$ or *request-update*$(\xi, x, f)_c$, *report-update*$(\xi, x)_c$.

We say that an operation identifier ξ *occurs in* sequence β provided that β contains a *request* event with operation identifier ξ. If β is any client-well-formed sequence and ξ occurs in β, then there is a unique request event in β for ξ. We sometimes denote this event simply by *request*(ξ). Also, if β is client-well-formed and complete, then there is a unique *report* event with operation identifier ξ; we denote it by *report*(ξ). We often refer to an operation identifier as just an *operation*.

If β is a complete client-well-formed sequence of actions, we define the *totally-precedes* partial order, *totally-precedes*$_\beta$, on the operations that occur in β by: $(\xi, \xi') \in$ *totally-precedes*$_\beta$ provided that *report*(ξ) occurs before *request*(ξ') in β. Notice that for each client c, *totally-precedes*$_{\beta|c}$ totally orders the operations that occur in $\beta|c$.

3.2 Definition

Our definition of sequential consistency is based on an *atomic object* [20, 22], also known as a *linearizable object* [17], whose underlying data type is the entire collection of data objects to be shared. In an atomic object, the operations appear to the clients "as if" they happened in some sequential order, and furthermore, that order must be consistent with the totally-precedes order. Specifically, we let *AM*, the *atomic memory automaton*, be just like the serial object automaton M_{serial} defined by Afek, Brown and Merritt [2] for the given collection of objects, except that we generalize it to allow updates that apply functions rather than just blind writes.

In more detail, the actions of *AM* are as follows. (Here, c, ξ, x and v are elements of C, Ξ, X, and V, respectively, and f is a function from V to V.)

Input:
request-read$(\xi, x)_c$, $\xi \in \Xi_c$
request-update$(\xi, x, f)_c$, $\xi \in \Xi_c$

Output:

[4] An *event* is an occurrence of an action in a sequence.

[5] Note that the completeness property includes both safety and liveness conditions.

request-read$(\xi, x)_c$
Effect:
$active(c) := (\textit{read-perform}, \xi, x)$

perform-read$(\xi, x)_c$
Precondition:
$active(c) = (\textit{read-perform}, \xi, x)$
Effect:
$active(c) := (\textit{read-report}, \xi, x, mem(x))$

report-read$(\xi, x, v)_c$
Precondition:
$active(c) = (\textit{read-report}, \xi, x, v)$
Effect:
$active(c) := \textit{null}$

request-update$(\xi, x, f)_c$
Effect:
$active(c) := (\textit{update-perform}, \xi, x, f)$

perform-update$(\xi, x, f)_c$
Precondition:
$active(c) = (\textit{update-perform}, \xi, x, f)$
Effect:
$mem(x) := f(mem(x))$
$active(c) := (\textit{update-report}, \xi, x)$

report-update$(\xi, x)_c$
Precondition:
$active(c) = (\textit{update-report}, \xi, x)$
Effect:
$active(c) := \textit{null}$

Figure 3: Automaton AM.

report-read$(\xi, x, v)_c, \xi \in \Xi_c$
report-update$(\xi, x)_c, \xi \in \Xi_c$
Internal:
perform-read$(\xi, x)_c, \xi \in \Xi_c$
perform-update$(\xi, x, f)_c, \xi \in \Xi_c$

The state of the automaton *AM* consists of:

mem, an array indexed by X of elements of V, initially identically v_0
active, an array indexed by C of tuples or the special value *null*, initially identially *null*

Here, $mem(x)$ represents the current value for object x, and $active(c)$ represents the access by client c that is currently in progress, if any. (The value *null* means that no access is currently in progress.)

The transitions for *AM* are described by the code in Figure 3. We represent the steps for each particular type of action in a single fragment of *precondition-effect* code (i.e., a guarded command). The automaton is allowed to perform any of these actions at any time when its precondition is satisfied; this style allows us to express the maximum allowable nondeterminism.

AM has one task for the output and internal actions of each client. This means that the automaton keeps giving turns to the activities it does in behalf of each client.

Note that every client-well-formed fair trace of *AM* is complete.

Sequential consistency is almost the same as atomicity; the difference is that sequential consistency does not respect the order of events at different clients. Thus, if β is a client-well-formed sequence of actions, we say that β is *sequentially consistent* provided that there is some fair trace γ of *AM* such that $\gamma|c = \beta|c$ for every client c. That is, β "looks like" γ to each individual client; we do not require that the order of events at different clients be the same in β and γ.

If A is an automaton that models a shared object system, then we say that A is *sequentially consistent* provided that every client-well-formed fair trace of A is sequentially consistent.

3.3 Proving Sequential Consistency

In order to show that the Orca shared object system is sequentially consistent, we will use a new proof technique based on producing a partial order on the operations that occur in a fair trace. In this subsection, we collect the properties we need, in the definition of a "supportive" partial order.

For each $c \in C$, let β_c be a complete client-well-formed sequence of request and report events at client c. Suppose that P is a partial order on the set of all operations that occur in the sequences β_c. Then we say that P is *supportive* for the sequences β_c provided that it is consistent with the order of operations at each client and orders all conflicting read and update operations; moreover, the responses provided by the reads are correct according to P. Formally, it satisfies the following four conditions:

1. P is *well-founded*.
2. For each c, P contains the order *totally-precedes*$_{\beta_c}$.
3. For each object $x \in X$, P totally orders all the update operations of x, and P relates each read operation of x to each update operation of x.
4. Each read operation ξ of object x has a return value that is the result of applying to v_0, in the order given by P, the update operations of x that are ordered ahead of ξ. More precisely, let $\xi_1, \xi_2, \ldots, \xi_m$ be the unique finite sequence of operations such that (a) $\{\xi_j : 1 \leq j \leq m\}$ is exactly the set of updates ξ' of x such that $(\xi', \xi) \in P$, and (b) $(\xi_j, \xi_{j+1}) \in P$ for all j, $1 \leq j < m$. Let f_j be the function associated with *request*(ξ_j). Then the return value for ξ is $f_m(f_{m-1}(\ldots(f_2(f_1(v_0)))\ldots))$.

The following lemma describes how a supportive partial order can be used to prove sequential consistency.

Lemma 3.1 *For each $c \in C$, let β_c be a complete client-well-formed sequence of request and report events at client c. Suppose that P is a partial order on the set of all operations that occur in the sequences β_c.*

If P is supportive for the sequences β_c, then there is a fair trace γ of AM such that $\gamma|c = \beta_c$ for every c and totally-precedes$_\gamma$ contains P.

Proof: Let P be a supportive partial order. We first show that we can extend P to a total order Q such that Q is also supportive for the sequences β_c. We define Q as follows: suppose ξ and ξ' are operations that occur in β_c and $\beta_{c'}$ respectively. Let $(\xi, \xi') \in Q$ provided that either ξ has fewer predecessors in P than ξ', or else the two operations have the same number of predecessors and c precedes c' in some fixed total ordering of the clients. It is clear by construction that Q is a total order on the operations that occur in the sequences and that $P \subseteq Q$.

To show that Q is supportive, we note that the second and third conditions follow from the fact that P is supportive (since Q contains P).

To show the first condition, we observe that P totally orders all the operations that occur in β_c (for the same c), and so it is not possible for two operations ξ and ξ' that are both in β_c to have the same number of predecessors. (Whichever is later will have a set of predecessors that include all the predecessors of the other, together with the other operation itself and possibly more). It follows that there are at most $n(N+1)$ operations that have $\leq N$ predecessors in P, where n is the number of clients in the system. Now, if an operation has N predecessors in P, then by definition of Q, each of its predecessors in Q must have at most N predecessors in P. Since there are at most $n(N+1)$ such operations, the operation has at most $n(N+1)$ predecessors in Q. This shows the first condition.

Finally the fourth condition holds for Q because it holds for P, and the set of update operations of x that precede a given read of x is identical whether P or Q is used as the order.

Now since Q is a total order in which each element has only a finite number of predecessors, arranging the operations in the order given by Q defines a sequence of operations. We obtain the required sequence γ by replacing each operation in this sequence by its *request* event followed by its *report* event.

We claim that γ has the required properties. The fact that each β_c is well-formed and complete implies that *totally-precedes*$_{\beta_c}$ is a total order on the operations that occur in β_c, and so these operations occur in Q in the same order; since γ is constructed to be well-formed and complete, the events in $\gamma|c$ are the same as the events in β_c, and their order is also the same. Thus $\gamma|c = \beta_c$. By construction, *totally-precedes*$_\gamma$ equals Q which contains P. Finally, γ is a trace of *AM* because the fourth condition ensures that return values are appropriate; the trace is fair since γ is complete. ■

The following lemma is what we actually use later in our proof.

Lemma 3.2 *Suppose that A is an automaton with the right interface for a shared object system. Suppose that, for every client-well-formed fair trace β of A, the following are true:*

1. *β is complete.*
2. *There is a supportive partial order for the sequences $\beta|c$.*

Then A is a sequentially consistent shared object system.

Proof: Immediate by Lemma 3.1. ■

4 Multicast Communication

In this section, we define properties for multicast channels, and in particular, define a context multicast channel.

As in the previous section, we start by identifying the actions by which the multicast channel interacts with its environment; now the environment will be a set of *sites* in a distributed network. The multicast channel receives requests from a site to send a message to a specified collection of sites, and responds by delivering the message to the requested recipients. Thus, the channel provides *multicast messages*. There are two special cases: when the destination set consists of the entire collection of sites (including the sender), the communication is called *broadcast*, and when the destination set contains a single site, the communication is called *point-to-point*.

Formally, let M be a set of *messages*, I be a set of *sites*, and $\mathcal{I}$ be a fixed set of subsets of I, representing the possible destination sets for messages. If $\mathcal{I} = \{I\}$ we say that the channel is *broadcast*, since the only possible destination set includes all the sites. When $\mathcal{I} = \{\{i\} : i \in I\}$ we say the communication system is *point-to-point*, since each destination set consists of a single site. The interface is as follows:

Input:
  mcast$(m)_{i,J}$, $m \in M$, $i \in I$, $J \in \mathcal{I}$
Output:
  receive$(m)_{j,i}$, $m \in M$, $j, i \in I$

The action *mcast*$(m)_{i,J}$ represents the submission of message m by site i to the channel, with J as the set of intended destinations. The action *receive*$(m)_{j,i}$ represents the delivery of message m to site i, where j is the site where the message originates. In each case, the action occurs at site i.

Now we describe various correctness properties for fair traces of multicast channels. First, we require reliable delivery of all messages, each exactly once, and to exactly the specified destinations. Formally, in any fair trace β of any multicast channel, there should be a *cause* function mapping each *receive* event in β to a preceding *mcast* event (i.e., the *mcast* event that "causes" this *receive* event). The two corresponding events should have

the same message contents, the site of the *mcast* should be the originator argument of the *receive*, and the site of the *receive* should be a member of the destination set given in the *mcast*. Furthermore, the *cause* function should be one-to-one on *receive* events at the same site (which means there is no duplicate delivery at the same site). Finally, the destination set for any *mcast* event should equal the set of sites where corresponding *receive* events occur (which means that every message is in fact delivered everywhere it should be).

In addition to these basic properties, there are additional properties of multicast systems that are of interest. These involve a "virtual ordering" of multicasts. We define these properties as conditions on a particular sequence β that we assume satisfies all the basic reliability requirements described just above, and a particular total order T of *mcast* events in β. The first condition is a technical condition: the virtual ordering T is really a sequence, i.e., it does not order infinitely many multicasts before any particular multicast.

Well-Foundedness T is *well-founded*.

The next condition says that the order in which each site receives its messages is consistent with the virtual ordering T. This implies that the order in which any two sites receive their messages is consistent.

Receive Consistency β and T are *receive consistent* provided that the following holds. If π and π' are *mcast* events in β, and a *receive* corresponding to π precedes a *receive* corresponding to π' at some site i, in β, then $(\pi, \pi') \in T$.

The next condition describes FIFO delivery of messages originating at the same site.

FIFO β and T are *FIFO* provided that the following holds. If π and π' are *mcast* events at site i in β, with π preceding π', then $(\pi, \pi') \in T$.

The final condition describes a restricted "causality" relationship, between a multicast that arrives at site and another that subsequently originates at the same site.

Context safety β and T are *context safe* provided that the following holds. If π is any *mcast* event, π' is an *mcast* event at site i, and a *receive* event corresponding to π precedes π' at site i in β, then $(\pi, \pi') \in T$.

Now we define a *context multicast channel* to be any automaton with the proper interface in which every fair trace β satisfies the basic reliability requirements, and also has a total order T such that β and T are well-founded, receive consistent and context safe. (We do not require the FIFO condition.)

In a totally ordered causal multicast channel, every fair trace has a total order guaranteeing the FIFO condition in addition to well-foundedness, receive consistency, and context safety. Thus, any totally ordered causal multicast channel is a special case of a context multicast channel. However, there are communication systems (such as the one described in Section 6) that are context multicast channels but are not FIFO.

5 The Higher Layer

Now we present the replica management algorithm, which uses a context multicast channel to implement a sequentially consistent shared memory (see Figure 4).

5.1 The Algorithm

The algorithm is modeled as a collection of automata P_i, one for each site i in a distributed network. As in the previous section, we let I denote the set of sites. The entire shared object system is, formally, the composition of the site automata $P_i, i \in I$, and a context multicast channel. Each client c is assumed to run at a particular site *site*(c). We let *clients*(i) denote the set of clients that run at site i.[6]

[6] In theoretical work on distributed shared memory, it is common to assume that only one client runs per site. This does not accurately model systems like Orca.

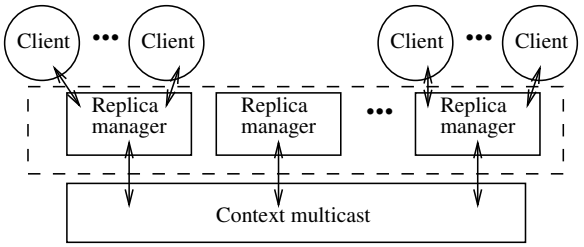

Figure 4: The architecture of the higher layer.

The algorithm replicates each object x at an arbitrary (but fixed) subset *sites*(x) of the sites, one of which is distinguished as the *primary site*, *primary*(x). We assume that the set of sites at which each object x is replicated is a possible destination set for the multicast channel, i.e., that for every x, *sites*$(x) \in \mathcal{I}$.

A site automaton P_i performs a read operation on an object x locally if it has a copy of x. Otherwise, it sends a request to any site that has a copy of x and that site returns a response. P_i performs an update operation on x locally if it has the only copy of x. Otherwise, P_i sends a multicast message to all sites that have copies of x, and waits to receive either its own multicast (in case P_i has a copy of x), or else an acknowledgement from the primary site (in case P_i does not have a copy).

Formally, the messages M used in the algorithm are of the following kinds:

(*read-do*$, c, \xi, x$),

(*update-do*$, c, \xi, x, f$),

(*read-reply*$, c, \xi, x, v$),

(*update-reply*$, c, \xi, x$),

where $c \in C$, $\xi \in \Xi$, $x \in X$, $v \in V$, and $f : V \to V$. The "do" messages are the requests to perform the operations, and the "reply" messages are the reports.

The interface of P_i is as follows. (Here, $c \in$ *clients*(i), ξ, x and v are elements of Ξ, X, and V, respectively, and f is a function from V to V. Also, m is an arbitrary message in M, $j \in I$, and $J \in \mathcal{I}$.)

Input:
- *request-read*$(\xi, x)_c$, $\xi \in \Xi_c$
- *request-update*$(\xi, x, f)_c$, $\xi \in \Xi_c$
- *receive*$(m)_{j,i}$

Output:
- *report-read*$(\xi, x, v)_c$, $\xi \in \Xi_c$
- *report-update*$(\xi, x)_c$, $\xi \in \Xi_c$
- *mcast*$(m)_{i,J}$

Internal:
- *perform-read*$(c, \xi, x)_i$, $\xi \in \Xi_c$
- *global-read*$(c, \xi, x)_i$, $\xi \in \Xi_c$
- *perform-update*$(c, \xi, x, f)_i$, $\xi \in \Xi_c$
- *global-update*$(c, \xi, x, f)_i$, $\xi \in \Xi_c$

The input and output actions of P_i are all the actions of all clients c at site i, plus actions to send and receive multicasts. The internal action *perform-read*$(c, \xi, x)_i$ represents the reading of a local copy of x, whereas *global-read*$(c, \xi, x)_i$ represents the decision to sen a message to another site requesting the value of x. Similarly,

perform-update$(c, \xi, x, f)_i$ represents the local performance of an update (when site i has the only copy of x), whereas *global-update*$(c, \xi, x, f)_i$ represents the decision to send a message in order to update x.

P_i has the following state components:

for every $c \in$ *clients*(i):
    status(c), a tuple or *quiet*, initially *quiet*
for every x for which there is a copy at i:
    val$(x) \in V$, initially v_0
buffer, a FIFO queue of (message, destination set) pairs, initially empty

The *status* components keeps track of operations being processed at the site. For example, if *status*(c) = (*update-wait*, ξ, x), it means that P_i has sent a message asking for x to be updated on behalf of operation ξ, and is waiting for to receive either its own message or an acknowledgement before reporting back to client c. (Because of client-well-formedness, status information needs to be kept for at most one operation of c at a time.) The *val*(x) component records the current value of the copy of x at site i. The *buffer* contains messages scheduled to be sent via the multicast channel.

The steps of P_i are given in Figure 5. We have organized the code so that the fragments involved in processing reads (plus the code for *mcast*) appear on the left and the fragments for processing updates appear on the right. Also, the fragments appear in the approximate order of their execution. However, note that the order in which the fragments are presented has no formal significance. As we described earlier, the automaton can perform any of its steps at time when its precondition is satisfied.

The code follows the informal description we gave above. For example, a *perform-read* is allowed to occur provided that the operation has the right status and i has a copy of the object x; its effect is to change the status to record the value read (and the fact that the read has occurred). As another example, a *global-update* is allowed to occur provided that the operation has the right status and i is not the only site with a copy of the object x; its effect is to change the status to record that P_i is now waiting and also to put a message in the *buffer*. The most interesting code fragment is that for *receive*(*update-do*). When this occurs, P_i always updates its local copy of the object x. In addition, if the message received is P_i's own message, then P_i uses this as an indication to stop waiting and report back to the client. On the other hand, if the message received is from a site that does not have a copy of x, and P_i is the primary site for x, then P_i sends a *reply* back to the sender.

The tasks of automaton P_i correspond to the individual output and internal actions. This means that each non-input action keeps getting chances to perform its work.

5.2 Correctness

Let A denote the composition of the site automata P_i and an automaton B that is a context multicast channel, with the *mcast* and *receive* actions hidden. We prove the following theorem:

Theorem 5.1 *A is a sequentially consistent shared object system.*

The proof of Theorem 5.1 is based on Lemma 3.2. The rest of this section is devoted to this proof. For the rest of the section, fix β to be an arbitrary client-well-formed fair trace of A, and let α be any fair execution of A that gives rise to β. Our eventual goals are to show that:

1. β is complete, and
2. there is a supportive partial order P for the sequences $\beta|c$.

request-read$(\xi, x)_c$
Effect:
  status(c) := $(\textit{read-perform}, \xi, x)$

perform-read$(c, \xi, x)_i$
Precondition:
  status$(c) = (\textit{read-perform}, \xi, x)$
  $i \in \textit{sites}(x)$
Effect:
  status(c) := $(\textit{read-report}, \xi, x, \textit{val}(x))$

global-read$(c, \xi, x)_i$
Precondition:
  status$(c) = (\textit{read-perform}, \xi, x)$
  $i \notin \textit{sites}(x)$
Effect:
  add $((\textit{read-do}, c, \xi, x), \{j\})$ to *buffer*
    where j is any element of *sites*(x)
  status(c) := $(\textit{read-wait}, \xi, x)$

receive$((\textit{read-do}, c, \xi, x), j)_i$
Effect:
  add $((\textit{read-reply}, c, \xi, x, \textit{val}(x)), \{j\})$ to *buffer*

receive$((\textit{read-reply}, c, \xi, x, v), j)_i$
Effect:
  status(c) := $(\textit{read-report}, \xi, x, v)$

report-read$(\xi, x, v)_c$
Precondition:
  status$(c) = (\textit{read-report}, \xi, x, v)$
Effect:
  status(c) := *quiet*

mcast$(m)_{i,J}$
Precondition:
  (m, J) is first on *buffer*
Effect:
  remove first element of *buffer*

request-update$(\xi, x, f)_c$
Effect:
  status(c) := $(\textit{update-perform}, \xi, x, f)$

perform-update$(c, \xi, x, f)_i$
Precondition:
  status$(c) = (\textit{update-perform}, \xi, x, f)$
  sites$(x) = \{i\}$
Effect:
  val(x) := $f(\textit{val}(x))$
  status(c) := $(\textit{update-report}, \xi, x)$

global-update$(c, \xi, x, f)_i$
Precondition:
  status$(c) = (\textit{update-perform}, \xi, x, f)$
  sites$(x) \neq \{i\}$
Effect:
  add $((\textit{update-do}, c, \xi, x, f), \textit{sites}(x))$ to *buffer*
  status(c) := $(\textit{update-wait}, \xi, x)$

receive$((\textit{update-do}, c, \xi, x, f), j)_i$
Effect:
  val(x) := $f(\textit{val}(x))$
  if $j = i$ then *status*(c) := $(\textit{update-report}, \xi, x)$
  if $j \notin \textit{sites}(x)$ and $i = \textit{primary}(x)$
    then add $((\textit{update-reply}, c, \xi, x), \{j\})$ to *buffer*

receive$((\textit{update-reply}, c, \xi, x), j)_i$
Effect:
  status(c) := $(\textit{update-report}, \xi, x)$

report-update$(\xi, x)_c$
Precondition:
  status$(c) = (\textit{update-report}, \xi, x)$
Effect:
  status(c) := *quiet*

Figure 5: Automaton P_i.

Since B is a context multicast channel, there is a total order on the *mcast* events satisfying well-foundedness, receive consistency and context safety. We choose one such order and call it T.

If ξ is an operation that occurs in β then since β is client-well-formed, we know that there is a unique *request*(ξ) event in β.

Let ξ be an operation. Then we classify ξ as follows.

1. ξ is a *local read* operation if ξ is a read operation of object x by client c and *site*$(c) \in$ *sites*(x).
2. ξ is a *local update* operation if ξ is an update operation of x by c and $\{$*site*$(c)\} =$ *sites*(x).
3. ξ is a *remote read* operation if ξ is a read operation of x by c and *site*$(c) \notin$ *sites*(x).
4. ξ is a *remote update* operation if ξ is an update operation of x by c and *site*$(c) \notin$ *sites*(x).
5. ξ is a *shared update* operation if ξ is an update operation of x by c, *site*$(c) \in$ *sites*(x) and $\{$*site*$(c)\} \neq$ *sites*(x).
6. ξ is a *global operation* if it is either a remote operation or a shared operation.

The following five lemmas summarize the way the algorithm processes all the different types of operations. Namely, local operations are processed using *perform* events, remote operations are processed using an *mcast*(do) message followed by an *mcast*$(reply)$ message, and shared operations are processed using an *mcast*(do) message only. The proofs are all routine.

Lemma 5.2 *Suppose that ξ is a local read operation of object x by client c that occurs in α. Let $i =$ site(c). Then:*

1. *α contains exactly one event perform-read$(c, \xi, x)_i$. This event follows request(ξ) in α.*
2. *α contains exactly one event of the form report-read$(\xi, x, v)_c$ (for some v), and this event follows perform-read$(c, \xi, x)_i$.*
3. *α contains no other events involving ξ.*

Lemma 5.3 *Suppose ξ is a local update operation of object x that occurs in α. Let $i =$ site(c). Then:*

1. *α contains exactly one event of the form perform-update$(c, \xi, x, f)_i$. This event follows request(ξ).*
2. *α contains exactly one event of the form report-update$(\xi, x)_c$, and this event follows perform-update$(c, \xi, x, f)_i$.*
3. *α contains no other events involving ξ.*

Lemma 5.4 *Suppose ξ is a remote read operation of object x by client c that occurs in α. Let $i =$ site(c). Then:*

1. *α contains exactly one event global-read$(c, \xi, x)_i$. This event follows request(ξ).*
2. *α contains exactly one event of the form mcast$($read-do$, c, \xi, x)_{i,\{j\}}$ (for some j), and this event follows global-read$(c, \xi, x)_i$.*
3. *α contains exactly one event of the form receive$($read-do$, c, \xi, x)_{i,j}$, and this event follows mcast$($read-do$, c, \xi, x)_{i,\{j\}}$.*
4. *α contains exactly one event of the form mcast$($read-reply$, c, \xi, x, v)_{j,\{i\}}$ (for some v), and this event follows receive$($read-do$, c, \xi, x)_{i,j}$.*

5. α *contains exactly one event of the form receive*$(\textit{read-reply}, c, \xi, x, v)_{j,i}$*, and this event follows mcast*$(\textit{read-reply}, c, \xi, x, v)_{j,\{i\}}$.

6. α *contains exactly one event report-read*$(\xi, x, v)_c$*, and this event follows receive*$(\textit{read-reply}, c, \xi, x, v)_{j,i}$.

7. *There are no other events in* α *that involve* ξ.

Lemma 5.5 *Suppose* ξ *is a remote update operation of object* x *by client* c *that occurs in* α*. Let* $i = \textit{site}(c)$*,* $J = \textit{sites}(x)$*, and* $j = \textit{primary}(x)$*. Then:*

1. α *contains exactly one event global-update*$(c, \xi, x, f)_i$*. This event follows request*(ξ).

2. α *contains exactly one event of the form mcast*$(\textit{update-do}, c, \xi, x, f)_{i,J}$*, and this event follows global-update*$(c, \xi, x, f)_i$.

3. *For every* $j' \in J$*,* α *contains exactly one event receive*$(\textit{update-do}, c, \xi, x, f)_{i,j'}$*, and each of these events follows mcast*$(\textit{update-do}, c, \xi, x, f)_{i,J}$.

4. α *contains exactly one event mcast*$(\textit{update-reply}, c, \xi, x)_{j,\{i\}}$*, and this event follows receive*$(\textit{update-do}, c, \xi, x, f)_{i,j}$.

5. α *contains exactly one event receive*$(\textit{update-reply}, c, \xi, x))_{j,i}$*, and this event follows mcast*$(\textit{update-reply}, c, \xi, x)_{j,\{i\}}$.

6. α *contains exactly one event report-update*$(\xi, x)_c$*, and this event follows receive*$(\textit{update-reply}, c, \xi, x)_{j,i}$.

7. *There are no other events in* α *that involve* ξ.

Lemma 5.6 *Suppose* ξ *is a shared update operation of object* x *by client* c *that occurs in* α*. Let* $i = \textit{site}(c)$*,* $J = \textit{sites}(x)$*, and* $j = \textit{primary}(x)$*. Then:*

1. α *contains exactly one event global-update*$(c, \xi, x, f)_i$*. This event follows request*(ξ).

2. α *contains exactly one event of the form mcast*$(\textit{update-do}, c, \xi, x, f)_{i,J}$*, and this event follows global-update*$(c, \xi, x, f)_i$.

3. *For every* $j' \in J$*,* α *contains exactly one event receive*$(\textit{update-do}, c, \xi, x, f)_{i,j'}$*, and each of these events follows mcast*$(\textit{update-do}, c, \xi, x, f)_{i,J}$.

4. α *contains exactly one event report-update*$(\xi, x)_c$*, and this event follows receive*$(\textit{update-do}, c, \xi, x, f)_{i,i}$.

5. *There are no other events in* α *that involve* ξ.

The previous lemmas can be used to draw some conclusions about the order of events in α.

Lemma 5.7 *Suppose that* $(\xi, \xi') \in \textit{totally-precedes}_{\beta|c}$ *for some client* c *at site* i.

1. *If* ξ *and* ξ *are both local, then the perform event for* ξ *precedes the perform event for* ξ' *in* α.

2. *If* ξ *is local and* ξ' *is global, then the perform event for* ξ *precedes the mcast of the do message for* ξ' *in* α.

3. *If* ξ *is local and* ξ' *is shared, then the perform event for* ξ *precedes the receipt of the do message for* ξ' *by* i*, in* α.

4. *If* ξ *is shared and* ξ' *is local, then the receipt of the do message for* ξ *by* i *precedes the perform event for* ξ' *in* α.

5. *If ξ is remote and ξ' is local, then the receipt of the reply message for ξ precedes the perform event for ξ' in α.*

6. *If ξ is shared and ξ' is global, then the receipt of the do message for ξ by i precedes the mcast of the do message for ξ' in α.*

7. *If ξ and ξ' are both shared, then the receipt of the do message for ξ by i precedes the receipt of the do message for ξ' by i, in α.*

8. *If ξ is remote and ξ' is global, then the receipt of the reply message for ξ precedes the mcast of the do message for ξ' in α.*

Proof: We prove Part 2. By Lemma 5.2 or 5.3, the *perform* for ξ precedes *report*(ξ). By the definition of *totally-precedes*$_{\beta|c}$, this in turn precedes *request*(ξ'). By Lemma 5.4, 5.5 or 5.6, this precedes the *mcast* of the *do* message for ξ'.

The proofs of the other parts of the lemma are similar. ■

Among the facts shown by the previous lemmas is that in fair client-well-formed executions, each operation has a report event following the request. Therefore, we have reached the first of our two goals:

Lemma 5.8 *β is complete.*

Now we turn to our second goal, of producing a supportive partial order P for the sequences $\beta|c$. We begin by providing some terminology for discussing the crucial actions of the various operations, namely, those actions that actually affect or use the values of object copies.

Let x be an object, and let i be any element of *sites*(x). Thus, there is a replica of x at site i. From the transition relation, we see that the events that can modify this replica are those of the form *perform-update*$(c, \xi, x, f)_i$ and *receive*(*update-do*$, c, \xi, x, f)_{j,i}$. We say that each of these is a *modification* of x at i *on behalf of* ξ. The only other events that use this replica are those of the form *perform-read*$(c, \xi, x)_i$ and *receive*(*read-do*$, c, \xi, x)_{j,i}$. We say that each of these is a *lookup* of x at i *on behalf of* ξ. A *crucial event* for x at i *on behalf of* ξ is either a modification or a lookup. Note that for any site i and operation ξ, α contains at most one crucial event at i on behalf of ξ. We also have some guarantees of when crucial events must occur:

Lemma 5.9 *Let ξ be an update operation of object x. Then for every site i in sites(x), there is a modification of x at i on behalf of ξ.*

Lemma 5.10 *Let ξ be a read operation of object x. Then there is some site i in sites(x) such that there is a lookup of x at i on behalf of ξ.*

Now we can define P. It is defined to be the transitive closure of the union of the following relations:

1. *mcast-order.*

   This relates any two global operations that occur in α, ordering them in the total order T provided by the context multicast channel B.

   That is, each global operation gives rise to a unique *mcast*(*read*) or *mcast*(*update*) event. If ξ and ξ' are global operations that occur in α, then we define $(\xi, \xi') \in$ *mcast-order* provided that the *mcast* event of ξ precedes the *mcast* event of ξ' in T.

2. For each site i, *crucial-order*$_i$.

   This relates any two (local or global) operations that both perform crucial events at site i in α, ordering them in the order of their crucial events.

   That is, if ξ and ξ' are operations that occur in α, then we define $(\xi, \xi') \in$ *crucial-order*$_i$ provided that α contains a crucial event at i on behalf of ξ and a later crucial event at i on behalf of ξ'. (Note that these events may be for different objects.)

3. For each client c, the *totally-precedes*$_{\beta|c}$ order on operations invoked by c, which totally orders the operations of client c.

It turns out that most of the work is devoted to showing that P is a partial order; it is then easy to show that P is supportive for the sequences $\beta|c$.

In order to show that P is a partial order, we show that all its constituent orders give the same order for global operations. This involves a case analysis, using the receive consistency and context safety properties. Then the combined order P just inserts local operations in appropriate places in the sequence of global operations.

The following lemmas demonstrate relationships between the different constituent partial orders.

Lemma 5.11 *Suppose that ξ and ξ' are local or shared operations of the same client c that occur in α. Let $i = site(c)$. If $(\xi, \xi') \in$ totally-precedes$_{\beta|c}$, then $(\xi, \xi') \in$ crucial-order$_i$.*

Proof: Four of the parts of Lemma 5.7 together imply that a crucial event at i on behalf of ξ precedes a crucial event at i on behalf of ξ', in α. Therefore, $(\xi, \xi') \in$ *crucial-order*$_i$. ■

Lemma 5.12 *Suppose that ξ is a remote operation. Then T orders the mcast of the do message for ξ before the mcast of the reply message for ξ.*

Proof: Let j be the site performing the *mcast* of the *reply* for ξ. By Lemma 5.4 or 5.5, the *mcast* of the *reply* for ξ must be preceded by the receipt by j of a *do* message for ξ. Then the context safety property implies that T orders the *mcast* of the *do* message for ξ before the *mcast* of the *reply* message. ■

Lemma 5.13 *Suppose that ξ and ξ' are two global operations and i is any site. If $(\xi, \xi') \in$ crucial-order$_i$ then $(\xi, \xi') \in$ mcast-order.*

Proof: For global operations on behalf of which a crucial event occurs at site i, *crucial-order*$_i$ is the order in which i receives the multicast *do* messages. By receive consistency, this is the same as the order given by T to the *mcast* events, which is exactly the order given to the operations by *mcast-order*. ■

Lemma 5.14 *Suppose that ξ and ξ' are two global operations of the same client c. If $(\xi, \xi') \in$ totally-precedes$_{\beta|c}$ then $(\xi, \xi') \in$ mcast-order.*

Proof: If ξ is shared, then by Lemma 5.7, the receipt by i of the *do* message for ξ precedes the *mcast* of the *do* message for ξ' in α. Then the context safety property implies that T orders the *mcast* of the *do* message for ξ before the *mcast* of the *do* message for ξ'. Thus, $(\xi, \xi') \in$ *mcast-order*.

On the other hand, if ξ is remote, then by Lemma 5.7, the receipt by i of the *reply* for ξ precedes the *mcast* of the *do* message for ξ', in α. Then the context safety property implies that T orders the $msmcast$ of the *reply* message for ξ before the *mcast* of the *do* message for ξ'. But Lemma 5.12 implies that T orders the *mcast* of the *do* message for ξ before the *mcast* of the *reply* message for ξ. So by transitivity, T orders the *mcast* of the *do* message for ξ before the *mcast* of the *do* message for ξ'. Again, $(\xi, \xi') \in$ *mcast-order*. ■

Lemma 5.15 *Suppose that ξ and ξ' are global operations. If $(\xi, \xi') \in P$, then $(\xi, \xi') \in$ mcast-order.*

Proof: If ξ and ξ' are directly related by one of the constituent relations, then the result is exactly the conclusion of Lemma 5.13 or 5.14.

Next, we consider the situation when ξ and ξ' are related through a chain of local operations, which must therefore all be at a single site i. Let these local operations be named $\xi_1, \xi_2, \ldots, \xi_m$. Lemma 5.11 shows that for each k, $(\xi_k, \xi_{k+1}) \in$ *crucial-order*$_i$. Since *crucial-order*$_i$ is transitive, we have either $m = 1$ or $(\xi_1, \xi_m) \in$ *crucial-order*$_j$. We divide the argument into cases, depending on which constituent relations give the initial and final edges in the chain.

1. $(\xi, \xi_1) \in$ *crucial-order*$_i$.

   Then by transitivity, $(\xi, \xi_m) \in$ *crucial-order*$_i$. That is, the receipt by i of the *do* message for ξ precedes the *perform* event for ξ_m. We consider subcases.

   (a) $(\xi_m, \xi') \in$ *crucial-order*$_i$.
   Then by transitivity, $(\xi, \xi') \in$ *crucial-order*$_i$, so Lemma 5.13 implies that $(\xi, \xi') \in$ *mcast-order*.

   (b) $(\xi_m, \xi') \in$ *totally-precedes*$_{\beta|c}$ for some c.
   Then $i =$ *site*(c) and ξ' is an operation of c. Lemma 5.7 implies that the *perform* for ξ_m precedes the *mcast* of the *do* message for ξ'. Thus, the receipt by i of the *do* message for ξ precedes the *mcast* (by i) of the *do* message for ξ'. Context safety then implies that $(\xi, \xi') \in$ *mcast-order*.

2. $(\xi, \xi_1) \in$ *totally-precedes*$_{\beta|c}$ for some c.

   Then $i =$ *site*(c) and ξ is an operation of c. If ξ is a shared update, then Lemma 5.11 implies that also $(\xi, \xi_1) \in$ *crucial-order*$_i$, in which case the earlier cases apply. So we may assume that ξ is a remote operation.

   Then Lemma 5.12 implies that T orders the *mcast* of the *do* message for ξ before the *mcast* of the *reply* message for ξ. And Lemma 5.7 implies that the receipt by i of the *reply* for ξ precedes the *perform* event for ξ_1, which either equals (in case $m = 1$) or precedes the *perform* event for ξ_m. Thus, the receipt by i of the *reply* for ξ precedes the the *perform* event for ξ_m. We consider subcases.

   (a) $(\xi_m, \xi') \in$ *crucial-order*$_i$.
   Then the *perform* event for ξ_m precedes the receipt by i of the *do* message for ξ'. Therefore, the receipt by i of the *reply* message for ξ precedes the receipt by i of the *do* message for ξ'. Then the receive consistency property implies that T orders the *mcast* of the *reply* message for ξ before the *mcast* of the *do* message for ξ'.

   (b) $(\xi_m, \xi') \in$ *totally-precedes*$_{\beta|c}$
   Then by Lemma 5.7, the *perform* event for ξ_m precedes the *mcast* of the *do* message for ξ'. Therefore, the receipt by i of the *reply* for ξ precedes the *mcast* of the *do* message for ξ', in α. By context safety, T orders the *mcast* of the *reply* message for ξ before the *mcast* of the *do* message for ξ'.

   Thus, in either case, T orders the *mcast* of the *reply* message for ξ before the *mcast* of the *do* message for ξ'. Then by transitivity, T orders the *mcast* of the *do* message for ξ before the *mcast* of the *do* message for ξ'. Thus, $(\xi, \xi') \in$ *mcast-order*.

Thus, if ξ and ξ' are related through a chain of local operations, then $(\xi, \xi') \in$ *mcast-order*.

Finally, if the chain between the operations ξ and ξ' includes other global operations, then we can divide it into segments each starting and ending with a global operation but containing no other global operations. The argument just made applies to each segment, which shows that the global operations in the whole chain are themselves a chain related by *mcast-order*. Transitivity of *mcast-order* yields that $(\xi, \xi') \in$ *mcast-order*. ■

Lemma 5.16 *The relation P is a partial order.*

Proof: Suppose not. Then there is a cycle of length at least 2 consisting of operations, each related to the following by one of the constituent relations. If the cycle contains any global operation ξ, then we have $(\xi, \xi) \in P$, which implies that $(\xi, \xi) \in$ *mcast-order* by Lemma 5.15; this contradicts the fact that *mcast-order* is an irreflexive partial order. If, on the other hand, every operation in the cycle is local, then all must be at a single site i, and by Lemma 5.11, there must be a cycle in *crucial-order*$_i$, which is also a contradiction. ■

Finally, we show that P is supportive.

Lemma 5.17 *P is supportive for the sequences $\beta|c$.*

Proof: We first show that P is well-founded, that is, that each operation has finitely many predecessors in P. Note that in each constituent relation, each operation has finitely many predecessors. So if this property does not hold in P, Konig's lemma implies the existence of an infinite chain of direct predecessors. If infinitely many operations in this chain are global, then Lemma 5.15 gives an infinite chain of predecessors in *mcast-order*, contradicting the well-founded property of the multicast service. On the other hand, if only finitely many operations in the chain are global, then we can start far enough along the chain and get an infinite chain of local operations. But then all these local operations must occur at the same site, say i. Then Lemma 5.11 yields an infinite chain of predecessors in *crucial-order*$_i$ which is impossible because α contains only a finite number of events that are before any given event. It follows that P is well-founded.

The construction immediately guarantees that, for any client c, P contains *totally-precedes*$(\beta|c)$.

Now we show that P relates all the "conflicting" operations (that is, a read and an update, or two updates) on a single object x. Suppose that ξ and ξ' are distinct operations of object x and that at least one is an update. Then Lemmas 5.10 and 5.9 together imply that there is some site i at which there are crucial events for x on behalf of both ξ and ξ'. This implies that ξ and ξ' are related by *crucial-order*$_i$, and therefore are related by P.

Finally, we argue that each read operation returns the right value. Suppose ξ is a read operation of object x. Then by Lemma 5.10, there is a site i at which a lookup event is performed on behalf of ξ. The algorithm ensures that the return value of ξ is exactly the cumulative effect of all the modifications performed on the copy of x at i before the lookup event. By Lemma 5.9, these modifications are exactly those that arise from the collection of update operations to x that are ordered before ξ by *crucial-order*$_i$. Also by Lemma 5.9, every update operation to x is related to ξ by *crucial-order*$_i$. Thus the collection of update operations to x that are ordered before ξ by *crucial-order*$_i$ is exactly the set of update operations to x that precede ξ in P. This shows that ξ receives the specified return value. ■

Proof: (of Theorem 5.1)
Lemmas 5.8, 5.16, 5.17, and 3.2 combine to imply that A is a sequentially consistent shared object system. ■

6 Lower Layer

Now we present the algorithm that constructs a context multicast channel based on a combination of totally ordered broadcast and point-to-point communication (see Figure 6).

We fix an arbitrary message alphabet M, set I of sites, and set $\mathcal{I}$ of destination sets; we will implement a context multicast channel for M, I and $\mathcal{I}$.

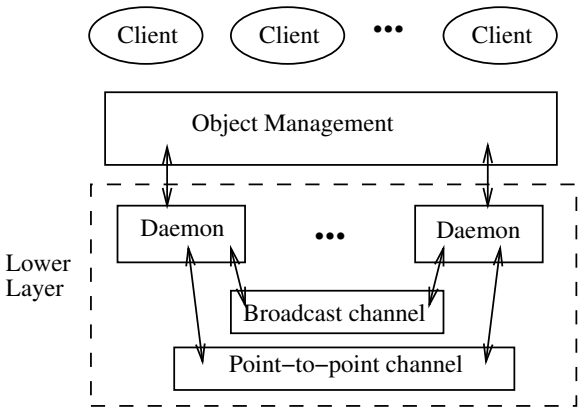

Figure 6: The architecture of the lower layer.

6.1 The Algorithm

The implementation is constructed as the composition of the following automata: *BC*, a reliable, totally-ordered broadcast channel,[7] *PP*, a reliable, point-to-point channel, and a collection D_i, one for each $i \in I$, of *daemon* automata that multiplex between the two lower-level services.

Both *BC* and *PP* are multicast channels, as defined in Section 4, and both have I as their set of sites. The broadcast channel *BC* has only one possible destination set, namely, I itself, while the point-to-point channel *PP* has exactly the singleton sets $\{i\}, i \in I$, as destination sets. Both satisfy the basic reliability requirements for multicast channels. In addition, we assume that *BC* is itself a context multicast channel – each of its fair traces has an ordering that is well-founded, receive consistent and context safe.[8] We do not assume anything additional about *PP*. In order to distinguish the *mcast* and *receive* events for *BC*, *PP*, and the channel being implemented, we superscript each action of *BC* and *PP* by the channel name.

Each automaton D_i processes the messages that are submitted by the environment via $mcast_{i,J}$ events. To process a message that is destined for more than one site, D_i broadcasts the message and its intended destination set, using the broadcast channel *BC*. When this message reaches a site j, automaton D_j delivers it to the environment if j is among the intended destinations; otherwise, D_j discards it. To process a message intended for one site only, D_i piggybacks on it the sequence number of the broadcast most recently received at site i, and then sends the embellished message directly to its destination using the point-to-point channel *PP*. After this message reaches its destination, it is delivered to the environment, but only after multicasts with the same and lower sequence numbers have been delivered. The interface of D_i is as follows. (Here, $m \in M$, $j \in I$, $J \in \mathcal{I}$, and k is a nonnegative integer.)

Input:
- $mcast(m)_{i,J}$
- $receive^{BC}(m, J)_{j,i}$
- $receive^{PP}(m, k)_{j,i}$

Output:
- $receive(m)_{j,i}$
- $mcast^{BC}(m, J)_{i,I}$
- $mcast^{PP}(m, k)_{i,\{j\}}$

[7] We model this broadcast channel as a single automaton. This could itself be implemented as a collection of automata, one per site, communicating through a still lower-level service.

[8] In fact, since each message is received by every site including the sender itself, and each *receive* event occurs after the corresponding *mcast* event, any total order in a broadcast system that is receive consistent must also be well-founded and context safe.

mcast$(m)_{i,J}$
Effect:
add (m, J) to *buffer*

mcast$^{BC}(m, J)_{i,I}$
Precondition:
(m, J) is first on *buffer*
$|J| > 1$
Effect:
remove first element of *buffer*

mcast$^{PP}(m, k)_{i,\{j\}}$
Precondition:
$(m, \{j\})$ is first on *buffer*
$k =$ *seqno*
Effect:
remove first element of *buffer*

receive$^{BC}(m, J)_{j,i}$
Effect:
seqno := *seqno* + 1
if $i \in J$ then add (m, j) to *msgs*
add to *msgs* (in any order) all (m', j')
such that $(m', j',$ *seqno*$) \in$ *ppwait*
remove from *ppwait* all $(m', j',$ *seqno*$)$

receive$^{PP}(m, k)_{j,i}$
Effect:
if $k \leq$ *seqno* then add (m, j) to *msgs*
else add (m, j, k) to *ppwait*

receive$(m, j)_i$
Precondition:
(m, j) is first on *msgs*
Effect:
remove first element of *msgs*

Figure 7: Automaton D_i.

D_i has the following state components:

buffer, a queue of (message, destination set) pairs, initially empty
msgs, a queue of (message, site) pairs, initially empty
ppwait, a multiset of (message, site, nonnegative integer) triples, initially empty
seqno, a nonnegative integer, initially 0.

The *buffer* component is used like *buffer* in P_i in the higher layer algorithm; it contains messages scheduled to be sent via the underlying communication services. The *msgs* component keeps track of messages that are scheduled for delivery to the environment, each with an indication of its site of origin. The *ppwait* component keeps track of point-to-point messages that are destined for site i, but that are waiting for the receipt of the broadcast with the appropriate sequence number. Finally, component *seqno* records the number of broadcasts received so far.

The code for D_i appears in Figure 7.

6.2 Correctness

Let C denote the composition of the site automata D_i together with *BC* and *PP*, with the actions of *BC* and *PP* hidden. We prove the following theorem:

Theorem 6.1 *C is a context multicast channel.*

The proof of Theorem 6.1 occupies the rest of this section. For the rest of this section, fix β be an arbitrary fair trace of C, and let α be any fair execution of C that gives rise to β. We define a relation T on the *mcast* events in β, and show that T is a total order with the required properties: β and T are well-founded, receive consistent, and context safe.

First, if π is any *mcast* event, then we define its epoch, *epoch*(π). If π is a multi-destination *mcast*, then *epoch*(π) is the value assigned to the state component *seqno* when π's *receive*^{BC} occurs at any site. (Receive consistency of *BC* and the fact that all sites receive each broadcast, imply that this value is uniquely defined.) Also, if π is any single-destination *mcast* event, say with destination set $\{i\}$, then *epoch*(π) is the maximum of the following two numbers: (a) the sequence number piggybacked on π's point-to-point message (this is the value of *seqno* at the sender when the corresponding *mcast*^{PP} occurs) and (b) the value of *seqno* at site i when the corresponding *receive*^{PP} occurs at D_i.

Notice that since the state component *seqno* is incremented exactly in each *receive*^{BC} event, it follows that the range of the function *epoch* is a prefix of the positive integers, and that each integer in this prefix is the *epoch* of exactly one multi-destination *mcast* event (as well as of zero or more single-destination *mcast* events).

We now define T as the relation on *mcast* events in α which is the transitive closure of the union of several individual relations.

1. The *multi-multi* order relates any two multi-destination *mcast* events in α; it orders them according to their *epoch*'s.

2. The *multi-single* relation orders a multi-destination *mcast* event π in α before a single-destination *mcast* event ϕ in α if *epoch*$(\pi) \leq$ *epoch*(ϕ).

3. The *single-multi* relation orders a single-destination *mcast* event ϕ in α before a multi-destination *mcast* event π in α if *epoch*$(\phi) <$ *epoch*(π).

4. The *single-single* order relates any two single-destination *mcast* events in α that have the same *epoch*; it orders them in the order of their *receive* events as they occur in α.

Then we must show that T is a well-founded total order, and that it guarantees the needed properties of receive consistency and context safety.

From the individual relations defined above we see that T respects the order determined by *epoch* numbers, and among events with the same epoch, T places the unique multi-destination *mcast* at the beginning.

Lemma 6.2 *Whenever* $(\pi, \pi') \in T$*, then* *epoch*$(\pi) \leq$ *epoch*(π')*. Also, whenever* $(\pi, \pi') \in T$ *and* π' *is a multi-destination mcast, then* *epoch*$(\pi) <$ *epoch*(π').

There is a partial converse to Lemma 6.2:

Lemma 6.3 *Whenever* *epoch*$(\pi) <$ *epoch*(π')*, then* $(\pi, \pi') \in T$.

Proof: Consider any two *mcast* events, π and π', in α, with *epoch*$(\pi) <$ *epoch*(π'). If both are multi-destination, then $(\pi, \pi') \in$ *multi-multi*; if π is multi-destination and π' is single-destination, then $(\pi, \pi') \in$ *multi-single*; and if π is single-destination and π' is single-destination, then $(\pi, \pi') \in$ *single-multi*. The remaining case is where both are single-destination. Then, taking ψ to be the unique multi-destination *mcast* event with *epoch*$(\psi) =$ *epoch*(π') we see that π is ordered before ψ by *single-multi* and ψ is ordered before π' by *multi-single*. Thus in every case $(\pi, \pi') \in T$. ■

Now we are ready to show in turn that β and T have the properties required of them.

Lemma 6.4 T *is a partial order.*

Proof: By Lemma 6.2, any cycle of edges each from one of the constituent relations (which generate T) would have to involve a collection of events all of which had the same *epoch*, and further, none could be a multi-destination *mcast*. But this means that all the edges must be from the relation *pp* which is itself a partial order. This contradiction shows that T is acyclic, and so, since it is by construction transitive and irreflexive, it is a partial order. ■

Lemma 6.5 *T is a total order.*

Proof: Consider any two distinct *mcast* events, π and π', in α. If *epoch*$(\pi) \neq$ *epoch*(π') then Lemma 6.3 shows that π and π' are related by T. On the other hand, suppose the events have the same epoch; thus at most one can be multi-destination. If neither is multi-destination then they are related by *single-single*, while if one is multi-destination and the other is single-destination then they are related by *multi-single*. Thus in every case π and π' are related by T. ■

Lemma 6.6 *T is well-founded.*

Proof: Note that at a given site i, there can be an infinite number of *mcast* events with the same epoch N only if there are exactly N *receive*^{BC} events at i. Since each broadcast is received at every site, this only happens if there are exactly N multi-destination messages in the execution. As there are a finite number of sites, there must be only finitely many *mcast* events in each epoch except possibly the last. As epoch values are non-negative integers, the predecessors of any *mcast* event π can include only a finite number of events with lower epochs, one multi-destination *mcast* with the same epoch, and those single-destination events with the same epoch which precede π in α. As α is a sequence, this latter collection is also finite. Thus T is well-founded. ■

Lemma 6.7 *T is context safe.*

Proof: Suppose that at i, *receive*$(m, j)_i$ is followed by *mcast*$(m', J)_i$. We divide the argument into cases.

Suppose both m and m' are intended for multiple sites, then at i the algorithm shows that the *receive*^{BC} for m precedes the event *receive*$(m, j)_i$. Similarly, *mcast*$(m', J)_i$ precedes the *mcast*^{BC} for m'. That is we must have the *receive*^{BC} for m before the *mcast*^{BC} for m'. Context safety of TMC now shows that at every site, the *receive*^{BC} for m precedes the *receive*^{BC} for m'. Thus the value of *seqno* assigned at any site during the *receive*^{BC} for m is less than the value of *seqno* assigned during the *receive*^{BC} for m'. That is, the epoch of the *mcast* of m is less than the epoch of the *mcast* of m'; so that the *mcast* events are ordered appropriately by T.

Suppose both m and m' are intended for single sites, then at p we must have the *receive* for m before the *mcast*^{PP} for m'. Now the epoch of the *mcast* event for m' is greater than or equal to the tag placed on m', which is the value of *seqno* at the time of the *mcast*^{PP} event for m'; this value must be at least as great as the value of *seqno* at the time of the *receive* for m. The use of the *ppwait* queue ensures that the value of *seqno* when the *receive* for m occurs is at least as great as the tag which was attached to m, and (because *seqno* never decreases) it is also at least as great as the value of *seqno* when the *receive*^{PP} for m occurs. By definition we see that the value of *seqno* at the *receive* for m is at least as great as the epoch of the *mcast* for m. Combining these observations, we see that the epoch of the *mcast* event for m' is greater than or equal to the epoch of the *mcast* for m. If these epochs are not equal, then it was shown in Lemma 6.3 that T orders the *mcast* events appropriately, while if the epochs are equal, then we note that the *receive* for m' must occur later in α than the *mcast*, and hence later still than the *receive* for m. Again, T orders the events appropriately.

Suppose that m is intended for multiple destinations while m' is intended for a single site. At p we must have the *receive*^{BC} for m before the *mcast*^{PP} for m'. Now the epoch of the *mcast* event for m' is greater than or equal

to the tag placed on m', which is the value of *seqno* at the time of the *mcast*^{PP} event for m'; this value must be at least as great as the value of *seqno* assigned during the *receive*^{BC} for m, which is the epoch of m. Thus the *multi-single* relation orders the *mcast* events appropriately, as so T also orders them appropriately.

Suppose that m is intended for a single site, while m' is intended for multiple sites. then at p we must have the *receive* for m before the *mcast*^{BC} for m', which itself occurs before the *receive*^{BC} event for m' at site i itself. Now the epoch of the *mcast* event for m' is the value assigned to *seqno* during the *receive*^{BC} event for m', which is stricty greater than the value of *seqno* at the time of the *receive* for m. The use of the *ppwait* queue ensures that the value of *seqno* when the *receive* for m occurs is at least as great as the tag which was attached to m, and (because *seqno* never decreases) it is also at least as great as the value of *seqno* when the *receive*^{PP} for m occurs. By definition we see that the value of *seqno* at the *receive* for m is at least as great as the epoch of the *mcast* for m. Combining these observations, we see that the epoch of the *mcast* event for m' is strictly greater than the epoch of the *mcast* for m. Thus the *single-multi* relation (and so also T) order the events appropriately.

The above cases cover all possibilities, showing that T is context safe. ■

Lemma 6.8 *T is receive consistent.*

Proof: We first note that the order of *receive* events at a site is the same as the order of entry into the queue *msgs* at that site. A multi-destination message m enters the *msgs* queue during the corresponding *receive*^{BC} step, in which *seqno* is first assigned to be the epoch of the *mcast* event for m. Also the *ppwait* queue is used so that a single-destination message m enters the *msgs* queue during an event after which the value of *seqno* is equal to the epoch of the *mcast* event for m. Now suppose that *receive*$(m, j)_i$ precedes *receive*$(m', j')_i$ at site i in α. Let us consider cases.

Suppose m and m' are multi-destination messages, so each enters the *msgs* queue during the corresponding *receive*^{BC} event. Thus the *receive*^{BC} for m precedes the *receive*^{BC} for m', and since *seqno* never decreases, the epoch of m is less than the epoch for m'. Thus *multi-multi* (and so also T) orders the *mcast* event for m before the *mcast* event for m'.

Suppose m and m' are both single-destination messages. Since the epoch of each is the value of *seqno* at the step when it enters the *msgs* queue, and since *seqno* never decreases, the epoch of m must be less than or equal to the epoch of m'. If the epochs are equal, then it is immediate that *single-single* orders the corresponding *mcast* events in the same order as the *receive* events. On the other hand, if the epoch of m is less than the epoch for m', then we saw in Lemma 6.3 that T orders the *mcast* event for m before the *mcast* event for m'.

Suppose that m is intended for multiple destinations while m' is intended for a single site. Since m enters the *msgs* queue during the step when *seqno* is first set to equal the epoch of m, and m' enters the *msgs* queue in a later event after which *seqno* equals the epoch of m', we have that the epoch of m is less than or equal to the epoch of m'. It is immediate that *multi-single* orders the *mcast* event for m before the *mcast* event for m'.

Suppose that m is intended for a single site, while m' is intended for multiple sites. Since m enters the *msgs* queue during a step after which *seqno* is equal the epoch of m, and m' enters the *msgs* queue during a later event in which *seqno* is first assigned to be the epoch of m', we have that the epoch of m is strictly less than the epoch of m'. It is immediate that *single-multi* orders the *mcast* event for m before the *mcast* event for m'.

In every case, we see that T orders the *mcast* event for m before the *mcast* event for m'. ■

Proof: (of Theorem 6.1) The properties of T have been shown in Lemmas 6.5, 6.6, 6.7, and 6.8. ■

We note that it is possible to improve the efficiency of the algorithm for all or one-site replication. For example, we tag each point-to-point message with the sequence number of the last broadcast received (in a *receive*^{BC} event) before the point-to-point message is sent (in a *mcast*^{PP} event). Alternatively, we could tag it with the sequence number of the last broadcast message that is passed to the environment at site i before the point-to-point message is submitted by the environment at site i. This can be smaller than the tag used above, so that the

destination site might delay the message for a shorter time. In this respect, our version of the algorithm follows the Orca implementation.

7 Discussion

We have presented a new algorithm for implementing a sequentially consistent shared object system in a distributed network. The algorithm is based on the one used in the Orca system, but generalizes it to allow objects to be partially replicated. Replicated objects are kept consistent using a context multicast system, which is a new communication service that can be implemented using a combination of totally ordered broadcast and point-to-point communication. We have presented this algorithm in two layers, and have carried out a complete correctness proof using this decomposition. In the course of our work, we found a logical error in the implementation of the Orca system that had not yet manifested itself in execution; as a result, the Orca implementation has been modified to correct this error.

This work opens up many avenues for future research. First, some simple extensions to our results can be made. For example, we could allow concurrent invocations of operations by the same client instead of requiring clients to block. In order to handle this case, we need to adjust our definition of sequential consistency to eliminate the *client-well-formedness* condition, to modify the algorithm to maintain sets of active operations, and to make minor changes in our proofs.

Another extension to our work is to incorporate objects with more general kinds of operations than just *read* and *update*.

A more serious extension is to allow for dynamic changes to the locations of object copies. As we noted in Section 1, Orca allows object locations to change dynamically, in response to changes in access patterns. There are several different schemes possible for managing such changes; most of these maintain the safety properties expressed by our results, but cause violations to the liveness conditions (e.g., an operation might not be able to find the needed copies because they are continuously moving). It remains to describe and verify existing schemes using our framework, and to develop and verify new schemes that preserve the liveness condition.

Still another extension is to use weaker communication primitives. In some process group systems such as the present implementation of Isis, consistent ordering is not guaranteed between all messages, but only between messages with a common destination. We would like to consider how to build a shared object system using this primitive together with point-to-point messages. For all these extensions we expect that much of the machinery developed in this paper can be reused.

8 Conclusions

Implementations for distributed systems such as Orca are complicated, because of the many possible interleavings of events of concurrent threads. It is generally difficult to be sure that such implementations are correct. Formal modeling and verification in the style we have presented here can provide great help in understanding and verifying such systems. Our modeling and verification of Orca has already contributed to the Orca project by identifying and correcting an error and by giving the designers extra confidence in the corrected implementation. In addition, the structures we have provided should provide useful documentation and assistance in future system modification.

More broadly, our work can be seen as a first step in the development of a practical theory for distributed shared memory systems. Such a theory should consist of a body of abstract component specifications, abstract algorithms, theorems about how the various abstract notions are related, and application-specific proof methods. Our contributions to this theory include our specifications for a sequentially consistent shared memory system

and for various kinds of multicast channels, our higher layer and lower layer algorithms and their correctness theorems, and our lemmas that show how to prove sequential consistency. However, our work is only a first step — we believe that much more work of the same kind, based on formal modeling of real systems and applications, is needed to complete the job.

Acknowledgements

We like to thank Henri Bal, Wilson Hsieh, Victor Luchangco, and Tim Ruhl for their comments on drafts of this paper.

References

[1] S.V. Adve and M.D. Hill. Weak ordering - a new definition and some implications. Technical Report TR-902, University of Wisonsin, Madison, WI, Dec. 1989.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM. Trans. on Programming Languages and Systems*, 15(1):182–205, Jan. 1989.

[3] M. Ahamad, P.W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proc. Eleventh International Conference on Distributed Computing Systems*, pages 274–281, Arlington, TX, May 1991.

[4] H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 679–691, 1992.

[5] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[6] H.E. Bal and M.F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *Proc. Eigth Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 162–177, Washington, DC, Sept. 1993.

[7] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. on Soft. Eng.*, 18(3):190–205, March 1992.

[8] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. Second Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Seattle, WA, March 1990.

[9] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, CMU, Pittsburgh, PA, Sept. 1991.

[10] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comp. Syst.*, 5(1):47–76, Feb. 1987.

[11] L.M. Censier and P. Feautrier. A new solution to cache coherence problems in multicache systems. *IEEE Trans. on Computers*, pages 1112–1118, Dec. 1978.

[12] Collier. *Reasoning about Parallel Architectures*. Prentice Hall Publishers, Englewood Cliffs, NJ, 1992.

[13] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, Feb. 1988.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. Seventeenth Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, May 1990.

[15] P. Gibbons and M. Merritt. Specifying non-blocking shared memories. In *Proc. Fourth ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, 1992.

[16] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc. Third ACM Symp. on Parallel Algorithms and Architectures*, pages 292–303, 1991.

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

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

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 28(9):690–691, Sept. 1979.

[20] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1(2):77–101, 1986.

[21] R.J. Lipton and J.S. Sandberg. Pram: a scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Princeton, NJ, Sept. 1988.

[22] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1995.

[23] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[24] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proc. Fourteenth Annual International Symposium on Computer Architecture*, pages 234–243, Pittsburg, PA, June 1987.

[25] A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, Aug. 1992.

[26] A.S. Tanenbaum, M.F. Kaashoek, R. van Renesse, and H.E. Bal. The Amoeba distributed operating system - a status report. *Computer Communications*, 14(6):324–335, Aug. 1991.