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 [20]. 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 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.

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

[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.

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, 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.

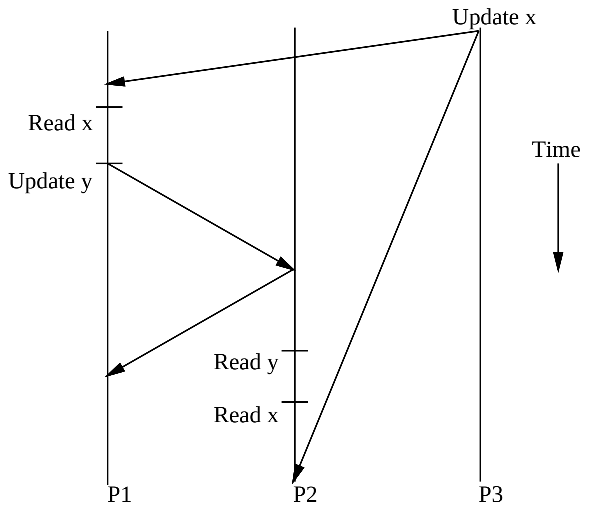

Figure 1: A problem with the naive replication strategy.

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 [19], but we only apply it to a restricted set of events.) We prove that this algorithm correctly implements a context multicast channel.

The higher layer uses an arbitrary context multicast

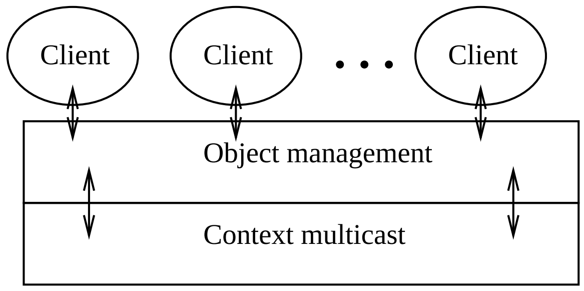

Figure 2: The architecture of the system.

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 [20]; 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, 15, 16, 17, 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 [18]. 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 ideas for future work. Finally, in Section 8 we draw our conclusions.

Because of space limitations, most details of the proofs are omitted here. Full details appear in [14].

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 internal and/or output 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:
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. 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 $\beta|c$.

[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.

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

3.2 Definition

Our definition of sequential consistency is based on an *atomic object* [22], also known as a *linearizable object* [18], 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 the serial object automaton 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. 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 variable $x \in X$, 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 variable 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.

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 $\gamma|c$=β_c for every c and totally-precedes$_\gamma$ contains P.

Proof Sketch: 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.

Now arranging the operations in the order given by Q defines a sequence of operations. Replacing each operation in this sequence by its *request* event followed by its *report* event yields the required sequence γ. ■

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.

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

The action *mcast*$(m, J)_i$ 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 subscript i denotes the site at which the action occurs.

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

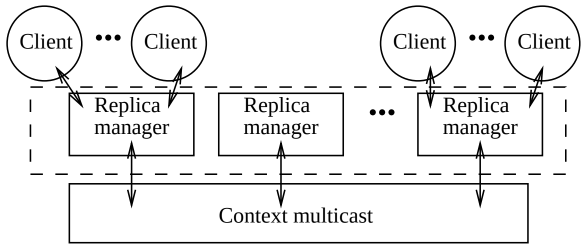

Figure 3: The architecture of the higher layer.

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 3).

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 $\mathit{site}(c)$. We let $\mathit{clients}(i)$ denote the set of clients that run at site i.[5]

The algorithm replicates each object x at an arbitrary (but fixed) subset $\mathit{sites}(x)$ of the sites, one of which is distinguished as the *primary site*, $\mathit{primary}(x)$. 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, $\mathit{sites}(x) \in \mathcal{I}$.

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:

$(\mathit{read\text{-}do}, c, \xi, x)$,

$(\mathit{update\text{-}do}, c, \xi, x, f)$,

$(\mathit{read\text{-}reply}, c, \xi, x, v)$,

$(\mathit{update\text{-}reply}, c, \xi, x)$,

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, $c \in \mathit{clients}(i)$, 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:
- $\mathit{request\text{-}read}(\xi, x)_c$, $\xi \in \Xi_c$
- $\mathit{request\text{-}update}(\xi, x, f)_c$, $\xi \in \Xi_c$
- $\mathit{receive}(m, j)_i$

Output:
- $\mathit{report\text{-}read}(\xi, x, v)_c$, $\xi \in \Xi_c$
- $\mathit{report\text{-}update}(\xi, x)_c$, $\xi \in \Xi_c$
- $\mathit{mcast}(m, J)_i$

Internal:
- $\mathit{perform\text{-}read}(c, \xi, x)_i$, $\xi \in \Xi_c$
- $\mathit{global\text{-}read}(c, \xi, x)_i$, $\xi \in \Xi_c$
- $\mathit{perform\text{-}update}(c, \xi, x, f)_i$, $\xi \in \Xi_c$
- $\mathit{global\text{-}update}(c, \xi, x, f)_i$, $\xi \in \Xi_c$

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 $\mathit{perform\text{-}read}(c, \xi, x)_i$ represents the reading of a local copy of x, whereas $\mathit{global\text{-}read}(c, \xi, x)_i$ represents the sending of a message to another site requesting the value of x. Similarly, $\mathit{perform\text{-}update}(c, \xi, x, f)_i$ represents the local performance of an update (when site i has the only copy of x), whereas $\mathit{global\text{-}update}(c, \xi, x, f)_i$ represents the sending of a message in order to update x.

P_i has the following state components:

for every $c \in \mathit{clients}(i)$:
  $\mathit{status}(c)$, a tuple, initially *quiet*
for every x having a copy at i:
  $\mathit{val}(x) \in V$, 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 $\mathit{status}(c) = (\mathit{update\text{-}wait}, \xi, x)$, 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 $\mathit{val}(x)$ 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 Figures 4 and 5. 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 steps at any time its precondition is satisfied; thus, this style allows maximum nondeterminism in the description of the algorithm. We have organized the code so that the fragments involved in processing reads (plus the code for

[5]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.

```
*request-read*$(\xi, x)_c$
Effect:
    *status*$(c)$ := $($*read-perform*$, \xi, x)$

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

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

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

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

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

*mcast*$(m, J)_i$
Precondition:
    $(m, J)$ is first on *buffer*
Effect:
    remove first element of *buffer*
```

Figure 4: Automaton P_i to perform read operations.

```
*request-update*$(\xi, x, f)_c$
Effect:
    *status*$(c) := ($*update-perform*$, \xi, x, f)$

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

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

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

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

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

Figure 5: Automaton P_i to perform update operations.

mcast) appear in Figure 4, while the fragments for processing updates appear in Figure 5. Also, the fragments appear in approximate order of their execution. However, the order in which the fragments are presented has no formal significance.

The code follows the informal description we gave above. For example, a *perform-read* can 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* can 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. Also, 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.

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

Proof Sketch: We use Lemma 3.2. Let β be an arbitrary client-well-formed fair trace of A, and let α be any fair execution of A that gives rise to β. We must show that β is complete, and that there is a supportive partial order for

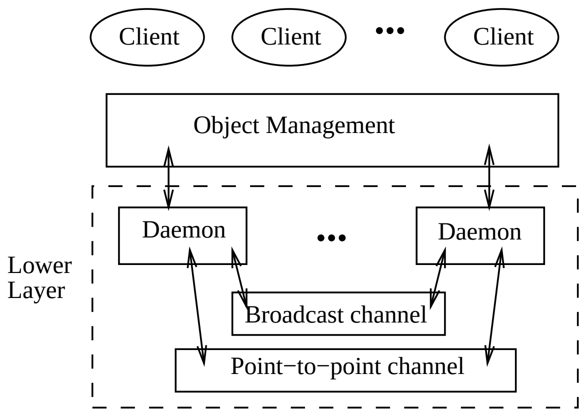

Figure 6: The architecture of the lower layer.

the sequences $\beta|c$. Completeness is argued by tracing the steps involved in the various operations in α, using a case analysis.

We construct a partial order P as the transitive closure of the union of several orders.

1. The *mcast* order relates any two operations whose processing in α includes the multicast of a *read-do* or *update-do* message; it orders them in the order T provided by the context multicast channel.

2. For each site i, the *access*$_i$ order relates any two operations that both perform accesses to copies of objects at site i, (that is, *perform*, *receive*(*read-do*) or *receive*(*update-do*) events at site i) in α; it orders them in the order of their accesses in α.

3. For each client c, the *totally-precedes*$(\beta|c)$ order, which totally orders the operations of client c.

To show that P is in fact a partial order, we show that all its constituent orders give the same order for operations whose processing uses multicasts. This involves a case analysis, based on the receive consistency and context safety properties. Then the combined order P just inserts operations that are performed locally in appropriate places in the sequence of operations that use multicasts.

To show that P is supportive, the key is that for any two conflicting operations (that is, a read and an update, or two updates) on a single object, there must be some copy that is accessed by both. ■

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 $\mathcal{I}$.

6.1 The Algorithm

The implementation is constructed as the composition of the following automata: *BC*, a reliable, totally-ordered broadcast channel,[6] *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.[7] 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*$_i$ 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$(m, J)_i$
receive$^{BC}((m, J), j)_i$
receive$^{PP}((m, k), j)_i$
Output:
receive$(m, j)_i$
mcast$^{BC}((m, J), I)_i$
mcast$^{PP}((m, k), \{j\})_i$

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.

[6] 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.

[7] 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.

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 follows.

```
mcast$(m, J)_i$
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$^{PP}((m, k), \{j\})_i$
Precondition:
    $(m, \{j\})$ is first on buffer
    $k$=seqno
Effect:
    remove head 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
```

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.

Theorem 6.1 *C is a context multicast channel.*

Proof Sketch: Let 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 total order T on the *mcast* events in β.

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.

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 it is straightforward to show that T is a well-founded total order, and that it guarantees the needed properties of receive consistency and context safety. ■

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. Eight 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] W.W. 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] A. Fekete, M.F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. Technical Report TM 518, MIT Laboratory for Computer Science, May 1995.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[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 publishers, Scheduled for 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.