ISIS: A System for Fault-Tolerant Distributed Computing*

Kenneth P. Birman

TR 86-744
April 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work was suported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Title page

ISIS: A SYSTEM FOR FAULT-TOLERANT DISTRIBUTED COMPUTING[1]

Kenneth P. Birman

Department of Computer Science
Cornell University, Ithaca, New York

[1]This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Abstract page

ABSTRACT

The *ISIS* system transforms abstract type specifications into fault-tolerant distributed implementations, while insulating users from the mechanisms whereby fault-tolerance is achieved. This paper discusses the transformations that are used within *ISIS*, methods for achieving improved performance by *concurrently updating* replicated data, and user-level issues that arise when *ISIS* is employed to solve a fault-tolerant distributed problem. We describe a small set of communication primitives upon which the system is based. These achieve high levels of concurrency while respecting ordering requirements imposed by the caller. Finally, the performance of a prototype is reported for a variety of system loads and configurations. In particular, we demonstrate that performance of a replicated object in *ISIS* can equal or exceed that of a nonreplicated object.

Keywords: Fault tolerant distributed computing, replication, concurrency, atomic broadcast, resilient objects, performance.

1. Introduction

Our basic premise is that the complexity of fault-tolerant distributed programs precludes their design and development by typical programmers. This complexity seems to be inherent: systems achieve fault-tolerance through redundant data or processing, and the distributed agreement and synchronization protocols needed for this purpose are hard to implement. Moreover, high levels of concurrency are required for reasons of performance, making it difficult to reason about correctness in the presense of failures. Alternatives to direct implementation are needed if fault-tolerant systems are to become widely available.

The *ISIS* project seeks to address this need by automating the transformation of fault-*intolerant* program specifications into fault-tolerant implementations, which we call *resilient objects*. In [Birman-a] we first reported this work; the present paper extends the previous discussion, providing considerable additional detail and performance data. *ISIS* works by replicating code and data while ensuring that the resulting distributed program gives behavior indistinguishable from a single-site instantiation of the original specification. Although many systems have been built to assist in the construction of distributed and fault-tolerant software, including ARGUS [Liskov-b], EDEN [Lazowska], CLOUDS [Allchin], LOCUS [Popek], TABS [Spector], and the TANDEM system [Bartlett], *ISIS* goes furthest in insulating users from the details of fault-tolerant programming. Moreover, *ISIS* places few restrictions on programs. In contrast, other systems that execute fault-intolerant specifications fault-tolerantly, such as CIRCUS [Cooper] and AURAGEN [Borg], are restricted to programs that are are fully deterministic given the specification. In particular, concurrent calls to the program are only allowed if the calling sequence is fixed. This sort of restriction effectively disallows the design of a fault-tolerant service that will concurrently be used by more than one caller -- the primary use to which resilient objects will be put.

Rather than implementing a specialized distributed protocol for each algorithm needed in the system, *ISIS* has been built using a communication layer supporting a variety of *broadcast*[2] proto-

cols. While broadcast primitives have been known in the literature for some time [Schneider] [Chang] [Cristian], the primitives we describe are integrated with a failure detection mechanism and provide unusually flexible delivery ordering properties. Within *ISIS*, algorithms are specified as sequences of calls to these protocols, making it feasible to reason about the correctness of our code. *ISIS* would have been far more complex, and our code more error prone, had it been built directly from a lower-level communication mechanism such as asynchronous message passing.

It is sometimes argued that fault-tolerance is best approached by employing specially designed redundant hardware. We believe that software fault-tolerance would be an issue even if this were done. Crashes often stem from user error and obscure bugs in the operating system and associated support software, or in high level application software. Moreover, even special hardware depends on a stable power source and air conditioning, and may have to be shut down from time to time. Thus, failures seem inevitable in any distributed system, and it is important to minimize the resulting disruption. Moreover, we will show that the problem of detecting and reacting to failure is, in its essence, not very different from that of tolerating more benign events, such as online reconfiguration and synchronization in parallel algorithms. By developing a technology for software fault-tolerance, we also gain mechanisms for addressing these other problems.

The structure of this paper is as follows. The next section introduces resilient objects, reviews the object specification language, and shows how an application program can be constructed using *ISIS*. The example presented, a distributed calendar service, required just a few days to design, code, and debug. At a more technical level, we then show how *ISIS* compiles a specification into a fault-tolerant implementation using the communication primitives mentioned above. The paper concludes by discussing the performance of our communication primitives, some sample objects, and of the overall system as the load and configuration are varied.

---

[2]Here, the term "broadcast" refers to a software protocol for sending information from a single source to a set of destination processes. Such broadcasts might take advantage of an ethernet broadcast capability, but can be implemented using other interconnection devices as well.

2. Resilient objects

ISIS extends a conventional operating system by introducing a new programming abstraction, the *resilient object*. Each resilient object is an abstract type that provides some service at a set of sites, where it is represented by *components* to which requests can be issued using remote procedure calls (RPC) [Birrel]. A typical *ISIS* application is constructed by developing conventional front-end programs and interfacing them to one or more such objects. The programmer can also define new, specialized, resilient objects if suitable ones do not already exist. A resilient object can be used for several purposes: as a specialized database for fault-tolerant information storage, as a source of status information through which processes monitor actions underway at remote sites and detect failures or recoveries, and as an intermediary for controlling and synchronizing distributed computations.

The translation of a non-distributed specification into a resilient object is based on several assumptions about the environment in which *ISIS* will be used and what resiliency should mean in this context. These are addressed below.

2.1. Failure assumptions

ISIS runs on clusters of computer systems communicating over a local area network. The network should not be subject to partitioning. Since local networks are often built from ethernets and token rings, consisting of interconnected clusters of sites, this assumption is reasonable. In case partitioning does occur, *ISIS* has been designed to pause when fewer than half the sites in a cluster are known to be operational, thus avoiding incorrect actions. Issues relating to reliable operation in the presense of partitioning failures are being addressed by some of our colleagues [El Abbadi-a] [El Abbadi-b]. We assume that the only way sites fail is by halting (crashing) [Schlicting]; tolerance of more malicious failures would lead to rapid increases in protocol costs at many levels of the system [Strong]. Failure detection and a collection of fault-tolerant broadcast protocols are implemented in software on top of the bare network.

2.2. Properties of resilient objects

Throughout this paper, the term *resiliency* is used to denote *k-resiliency*. A k-resilient object satisfies the following properties:

1. **Consistency**. The external behavior of an is like that of a non-distributed one which executes requests sequentially and to completion, with no interleaving of executions. An object may also implement distributed synchronization operations that introduce additional consistency properties.
2. **Availability**. Let f denote the number of components of an object that fail simultaneously. If $f \leq k$, then operational components continue to accept and process requests.
3. **Progress**. If $f \leq k$, then operations are executed to completion, despite failures.
4. **Recovery**. Because *ISIS* supports replication, two cases can be distinguished:
   a. **Partial**. If $f \leq k$, a failed component restarts automatically when its site recovers.
   b. **Total** If $f > k$, failed components restart automatically when all the failed sites recover.

It should be stressed that k-resiliency is a much stronger property than resiliency of the sort discussed in [Svobodova] or [Liskov-b]. In both of these papers, resiliency denotes the ability of a system to remain in an internally consistent state during normal execution and to automatically recover into such a state after a failure has been resolved. Our approach reflects the additional assumption that data will be replicated, making it feasible (and desirable) to continue to provide services even though a failure has occurred.

2.3. Logical execution model

We decided to model the execution of operations on resilient objects by transactions [Eswaren]. Although this precludes supporting some interesting "non-transactional" resilient types, it is convenient because it permits a programmer to specify resilient objects in a straightforward manner, using a lock-based concurrency control algorithm to enforce the transactional abstraction. A non-transactional model would force the programmer to become much more

involved in the details of synchronization.

Specifically, the execution of each operation gives rise to a transaction, which begins (implicitly) when the procedure specifying the operation is invoked and commits (implicitly) when that procedure returns a result. A procedure can also *abort*, which explicitly causes its actions to be rolled back. Since procedures can call one another, our model is essentially that of Moss' *nested transactions* [Moss], although a wider variety of lock types are available than Moss discussed (Sec. 5.1).

Liskov has observed [Liskov-b] that a mechanism is needed for initiating *top-level* transactions from within other transactions, in order to avoid severe inefficiency. A top-level transaction is one that was invoked by a non-transactional caller; any transaction invoked within some other transaction is said to be a *subtransaction*, and commits or aborts relative to its parent transaction. Liskov points to a case in which the garbage collected during the execution of a subtransaction must be reinstalled if its caller aborts, and argues that this inefficiency can be avoided if a mechanism for initiating a top-level transaction from within other transactions is available. Top-level transactions have other uses as well, notably because they permit increased concurrency when transactionally updating a concurrently accessed data structure. Here, an update may affect both the linkage fields of the data structure itself and the contents of some record, and it is desirable to view these as independent events. By using top-level transactions to update link-fields, other concurrently executing operations are permitted to "pass through" a node while it is still being updated, without waiting until the update transaction has terminated. Operations that try to access the record contents, however, block until the update transaction commits or aborts. For these reasons, we included a top-level transaction mechanism in *ISIS*.

Another possibility was to support an *undo* mechanism, which would permit the programmer to associate an arbitrary undo action with each operation. If it becomes necessary to abort an operation, the corresponding undo action is executed. Unlike the system-level *abort* feature now supported in *ISIS*, an undo mechanism could be used to back out of actions that have external

side-effects. However, abort is rare in our prototype, and we decided not to implement undo actions at the present time.

3. Object specification language and system interface

In this section we review the language used to specify *ISIS* objects, the interface provided to external callers. A command language is also available, and is used to control the *ISIS* system itself during execution, but is not described in detail here.

3.1. Resilient objects

The *k-resilient types* are a special class of abstract data types [Liskov-a]. Each resilient object instantiates a resilient type and is accessible to holders of a *capability* on it; these are *open* in that they can be freely copied or stored [Dietrich]. Resilient type specifications have the following parts:

1. Declarations for the *resilient data* encapsulated by the type, consisting of one or more indefinite-length arrays or heaps[3] of *resilient records*.
2. Type definitions for the parameters and results of operations.
3. Procedures for manipulating resilient data. These can be given attributes such as *create* (executed when an instance of an object is created), *entry* (accessible to external callers), and *read-only* (does not update resilient data).

Resilient procedures are coded using a version of the *C* programming language. All of *C* is available, as are many operating system calls. The language has been augmented to include a multi-tasking facility for internal concurrency and to provide several new statement types:

[3]Sequentially allocated data structures tend to have "hot spots" which are frequently accessed, reducing potential concurrency [Gawlick]. The heap management facility supports dynamic allocation and deallocation of resilient records within transactions, avoiding a common source of hot-spots. Heap management is done using per-transaction allocation and free lists, which are updated when a heap allocation or free is done and when a subtransaction commits or aborts.

1. *I/O statements*. Resilient data is accessed using *read* and *write* statements, which can also specify a lock to acquire before performing the access. By forcing the programmer to use a special notation when accessing resilient data, the most natural programming style tends to minimize those operations -- and this is also the most efficient way to use *ISIS*.

2. *Remote procedure calls*. A flexible RPC mechanism is provided, including nested, recursive, and asynchronous RPC's, as well as RPC's in which the function to call is a parameter. It should be noted, however, that there are some technical restrictions on the use of recursive and asynchronous RPC's that stem from the model, hence it is not clear whether typical users will make use of either feature. RPC is also used to as an interface to most *ISIS* system functions.

3. *Abort return*. A normal *return* from a resilient procedure is interpreted as a *commit* of the sub-transaction that was being executed. In an *abort return*, the effects of the procedure (and any that it has invoked) are erased.

4. *Cobegin*. A set of *branches* (statements, which do not contain *return* or *abort* statements) are specified for concurrent execution. The cobegin terminates when all its branches terminate. Each *cobegin* branch executes as a task within a single address space. A more flexible *cobegin*, which might provide some form of explicit control over concurrent processing at multiple sites, is under consideration. The statement is currently used to keep a computation active while some branch is blocked (e.g. when acquiring a lock). *ISIS* assumes that the branches of a *cobegin* do not compete for locks on the same data items.

5. *Toplevel*. The statement is executed as a top-level transaction.

3.2. Features for monitoring remote activities

Although resilient objects provide a simple mechanism for concealing replication and distribution, they will also be used to explicitly synchronize and control distributed computations. As a result, features have been added to the language that permit a computation to pause until an update to a replicated variable has been received, or until one of the sites at which an object

resides failure or recovers. A predefined *SITES* variable can be read to determine the full set of sites at which an object resides. A *VIEW* variable gives the subset of *SITES* that are currently operational. By combining these mechanisms, it is easy to build objects with very sophisticated distributed functionality.

3.3. Facilities for use in programs external to ISIS

Non-resilient clients interact with resilient objects through an interface that resembles the one used by objects to communicate with one another, by issuing RPC calls to objects on which they hold or can obtain a capability. *ISIS* supports a globally accessible name space object, which has a well-known capability, and can be used to look up a desired object using a symbolic name.

Normally, each RPC issued by a client executes as a top-level transaction. A client can, however, explicitly combine a series of requests into a transaction. To do this, the program first invokes a *BEGIN* procedure, then performs the operations, and then invokes *COMMIT* or *ABORT*. A client can only have one active transaction at a time.

If a client fails while a transaction is in progress, the default action is to terminate the transaction. Observe that the semantics of termination in this case differ from those for *abort*. This is because an *abort* is explicitly executed by the computation to be aborted, whereas when a client fails it may be necessary to interrupt a computation that is still in progress, or blocked (perhaps deadlocked). To this end, *ISIS* supports a software **kill** signal, which can be issued by a client process, and is automatically issued when a client fails before terminating a transaction. **Kill** cannot be caught or ignored, and terminates a transaction by halting it and its subtransactions and then aborting them.

Transactional systems that lack a mechanism for ensuring continued progress despite failure generally implement a variant of **kill** to terminate transactions that are interrupted when a site at which they executed fails (this gives rise to the *orphan termination* problem discussed in [Moss] [Liskov-b]). Software built using these systems must avoid irreversible actions, like movement of a mechanical arm or dispensing cash from a machine, because the only time such an action can

safely be taken is during the top-level commit (when **kill** can no longer occur). Unfortunately, this tactic makes it hard to implement certain types of operation, for example one that moves a mechanical arm while monitoring *and reacting* to sensor feedback. Here the desired behavior could only be obtained by breaking the operation into a series of separate transactions. The programmer would then implement his own algorithms to cope with failures, a nontrivial undertaking. Because it uses a progress mechanism, *ISIS* never invokes **kill** automatically unless a client-level transaction fails. Thus, an operation such as this is easily implemented within a resilient object. Clearly such an object must be deadlock-free, but this is a minor restriction (for example, locks can simply be acquired in a fixed order).

4. Programming in ISIS

This section summarizes the development of a distributed appointment and calendar system, which was undertaken to exercise *ISIS* and to gain some programming experience using resilient objects. The calendar combines data storage with a dynamic monitoring capability: users who chose to display their schedule on a terminal are shown changes as they occur.

To develop the calendar program, we began by designing and implementing a conventional single-user program with the same functional decomposition that we intended to employ in the fault-tolerant distributed version. This program contains an interactive display module, a command interpreter, and a collection of procedures for managing the calendar data structure. The data structure consists of an array of per-user information (PUI) blocks and, for each user, a linked-list of appointment schedules. The command interface permits a user to define a new group (a set of users and a reason for their meetings) or schedule a meeting, and can provide advice to a user who wishes to schedule a meeting with some group but is unsure what time to pick. A graphic display of a user's schedule is also provided.

Having completed this initial version of the calendar, we modified the calendar database into a resilient object. The object supports operations to fetch or update the PUI for a user or group, fetch or update the entry for a week, and to pause until a change to the schedule for the current

user is detected (this enables the program to refresh the display when the schedule is changed by some other user).

Concurrency control proved to be straightforward to implement. We divided commands into two types: read-only requests and update requests. Each is executed as a transaction. Locking is done on just the PUI blocks, and these are locked in a fixed order, a strategy which is deadlock free.

For reasons of performance, it was desirable to cache information in the interactive front-end programs. A version number was therefore associated with each PUI block and incremented by update transactions. All calendar information except the PUI is cached. Each time a PUI block is referenced, any invalided cache entries for that user are discarded. In practice it seems likely that cache entries will normally be accurate simply because calendars are consulted more frequently then they are updated.

To summarize, we found it easy to implement a nontrivial distributed program using *ISIS*. The result is not of production quality, but the remaining issues are in the calendar interface, not the feasibility of implementing the calendar, and the program was never subject to concurrency related bugs or problems with failure handling and recovery. This would certainly not have been the case in a development starting with the basic UNIX interprocess communication primitives. In fact, our program was converted by the author from a single-site version into a distributed one within a few days. Moreover, the performance of the resulting system is reasonable: although there may be a delay of several seconds before information from an complex update is reflected in remote copies of the calendar, users of the system are unaware of this, since their local calendar reflects updates rapidly. In effect, the program gives a speedy response and then completes the the request in the "background".

5. Runtime issues

In this section we turn to the runtime mechanisms that underlie the implementation of a resilient object. These are nontrivial because of the many "physical" events that can occur during

execution. Our treatment begins by surveying the algorithms without addressing details relating to their implementation using broadcast primitives. Sec. 5.2 introduces the primitives actually employed within *ISIS*, and Sec. 5.3 then shows how some of the algorithms of Sec. 5.1 can be efficiently implemented using them.

5.1. Fault-tolerant execution of a request

We use the term *task* to refer to the physical execution of a request by one of the components of a resilient object, designated the *coordinator*. Components are identical: any can be coordinator for any request, which tends to distribute processing load. The components that are passive for a request are designated as *cohorts* and serve as backups -- one takes over if the coordinator fails. Recall from Sec. 2.2 that a task must satisfy several properties to produce a correct logical execution. We now consider these properties individually.

5.1.1. Consistency

Concurrency control [Bernstein] is not automatic in *ISIS*, because it is difficult to infer an efficient concurrency control algorithm without knowledge of the semantics of operations. Therefore, *ISIS* requires that the programmer provide a single-site concurrency control algorithm, which is transformed into a distributed one. The class of concurrency control algorithms supported are the conflict serialization algorithms [Papa], of which 2-phase locking is the best known and easiest to code. Two lock classes are supported (see below); within each class, read, promotable (exclusive) read, previous committed version read, and write locks can be requested.

The previous committed version read-lock is unusual, and deserves further explanation. Locks of this sort permit a read-only transaction to execute concurrently with one that is updating some of the data items it accesses. Denote such a read-only transaction R and an update transaction with which it conflicts U. If R requests a read previous lock on some item x, that lock can be granted even if U already has write-locked x, and the subsequent read request by R satisfied from the last committed version of x. Should U attempt to commit before R does so, U must now be forced to wait until R commits and releases its locks. Then, if R reads other records that U is

updating, it will consistently see the versions committed before U began executing. In effect, R has been serialized before U although it started execution after U, and neither R nor U is forced to block until U reaches its commit point. A similar form of concurrency control was described in [Weihl], where *hybrid atomicity* was introduced to capture the behavior of a special "timestamped" concurrency control method that also permits read-only transactions to run using previous versions of data items. Since many transactions are read-only, previous copy locks should be very valuable in systems such as *ISIS*.

The two lock classes supported by *ISIS* are:

1. Nested 2-phase locks. When a subtransaction commits, the lock is retained by its parent transaction [Moss]; when the top-level commits, it is released.
2. Local 2-phase locks. When the transaction or subtransaction holding the lock commits, it is released.

Nested 2-phase locks are easiest to work with, and probably suffice for most users. On the other hand, by using local locks in conjunction with top-level transactions, it is possible to implement highly concurrent data structures, and this approach was used successfully within the *ISIS* namespace object. The combination of lock classes and types makes *ISIS* exceptionally flexible at the level of concurrency control, and we believe that efficient concurrency control algorithms can be devised for most objects.

5.1.2. Availability

Availability is satisfied by replicating the code and data for each resilient object. Since data accesses are transactional, each item is represented as a stack of versions [Moss], replicated at $k+1$ or more sites. A read-one copy, write-all (operational) copies rule is used when locking or accessing replicated data. An item is updated by pushing a new version on all copies of the corresponding stack, or replacing the top-most version if one already exists for the transaction doing the update. Abort is implemented by popping the top version, and commit by popping the top two

and then pushing the first again. The best known alternative to the read-one, write-all approach is to employ a quorum access rule, where both read and write requests are satisfied by accessing multiple copies [Gifford] [Herlihy]. We rejected this because the latency incurred while waiting for a quorum of responses from remote sites reduces the level of concurrency below that which we attain using the read-one write-all approach, where computation can proceed without delay as soon as the local copy of a data item has been accessed (evidence to support this conclusion is given in Sec. 7). Qurorum methods are preferable if network partitioning is common, because they increase availability [Herlihy] [El Abbadi-a] [El Abbadi-b], however this is not felt to be an issue in the environment for which *ISIS* was designed.

5.1.3. Progress

ISIS ensures that operations progress to completion using a transactional checkpoint-restart scheme [Birman-a] [Birman-b]. Related work on non-transactional checkpoint and restart appears in [Toueg][Chandy]. Each RPC is broadcast to the operational components of an object, and constitutes an *initial checkpoint*. If a coordinator fails while executing the request, one of its cohorts takes over as the new coordinator. It restores its copy of the object to the state at the time of the checkpoint, discarding versions of data items that were written by the transaction being restarted (this requires no communication with other components or objects). The actions of the failed coordinator are then repeated in *restart mode* in order to reestablish the state that existed at the time of the failure.

When an operation is reissued in restart mode, it clearly should not be re-executed -- otherwise, the system state could become inconsistent (e.g. if an increment were done twice). To avoid this, the *results* returned by completed operations are replicated in addition to the updates they perform on replicated data. Specifically, when a coordinator finishes executing a request it broadcasts the result to its cohorts as well as to the caller. The cohorts retain copies of each result under the TID of the transaction, constituting a *final checkpoint*. Because the same TID is used during restart (see Sec. 5.3 for details), when the operation is reissued a copy of the retained

result can be located and returned (this is done by the process that would normally have executed the request in the called object). If none is found, a restart-mode request is rejected. The coordinator performing the restart deduces from this that normal execution should resume.

The storage overhead associated with our method is low. Retained results can be discarded when the parent of a subtransaction commits or aborts, retaining its own result, or when the top-level commits. On the other hand, since top-level statements are re-executed during restart, the results of the top-most action in such a statement must be retained. Also, if a resilient object takes external actions like moving a robot arm, the robot arm must provide a function equivalent to a retained result -- for example, a way that the device driver can determine the command it last executed and in this manner identify a duplicated command (a minor restriction since most devices of this sort contain microprocessors and memory).

While restarting, it is not enough for the new coordinator to determine the results returned by operations that were previously executed. The serialization order must also be the same as was used before the failure -- otherwise, the values read from resilient data items by the new coordinator might differ from those read by the previous one, again leading to inconsistencies. *ISIS* addresses this issue by replicating both read and write locks, so that after a failure the new coordinator holds all the locks acquired by the previous coordinator before it failed. Because replicating read-lock information is potentially inefficient, the approach is to piggyback this data on other messages that could depend on their existence -- RPC requests and updates issued subsequent to the acquisition of the lock. If an RPC or update persists after the crash, the corresponding locking information persists as well. This information is forwarded to the new coordinator before it is informed of the failure, which therefore registers the locks prior to initiating restart [Birman-b]. The reader may be troubled by the apparently complex synchronization requirements of this algorithm: read-locks must be registered before restart begins, and the consistency of the system state must be maintained after failure. We show in the next section that these problems can both be resolved in an elegant manner within the communication primitives themselves.

5.1.4. Recovery

If a partial failure occurs, a failed component can recover by discarding its old state and copying the current state from some operational component. In effect, the components of an object act as dynamic backups, eliminating the need for stable (disk) storage. Later, we will show that the communication primitives can be used to serialize recovery with respect to other operations. To tolerate total failure, an object must save checkpoints and committed versions of the object data on stable storage. When the components that failed last have recovered [Skeen-a], they can resume operation from their stable stores; other components use the partial recovery method.

Some care is needed in deciding what information should be placed in stable storage, since this approach will be costly if access to stable storage must occur frequently. For example, consider a request to insert a data item into a complex data structure. The message containing the request may be tiny, but massive changes to the data structure could result. If *ISIS* were to blindly perform these on a stable representation of the structure, performance would be very poor. On the other hand, if the request itself were logged, the object could restart from failure by replaying the request log. By saving periodic backups of the object state and clearing the log, the cost of replaying it can be kept small. The cost of updating such a log will be minor in comparison with the cost of maintaining the entire object state in a stable form. *ISIS* therefore provides the programmer with a tool for maintaining a log transactionally. It is possible to log any RPC request. The capability and arguments are written to the log and later can be replayed by some other transaction. If a transaction aborts, log entries it has made are deleted. A consequence is that the performance of the stable storage mechanism is not a bottleneck in *ISIS*.

5.2. The communication subsystem

The *ISIS* communication subsystem provides three types of broadcast protocol for transmitting a message reliably from a sender process to some set of destinations. The protocols are *atomic* in an "all or nothing" sense: if any component of an object receives a message, then unless

it fails, all operational [redacted]ts will receive it. Atomic broadcast has often been proposed as a basic primitive from which higher level system services can be constructed, and several protocols for realizing such broadcasts have been reported in the literature [Schneider] [Chang] [Cristian]. Unfortunately, although the number of packets transmitted to deliver a message is low in the published protocols, the latency before message delivery takes place is potentially high in comparison to average intersite message transit times, primarily because they enforce a global message delivery ordering in addition to the atomicity property given above: broadcasts are received in the same order everywhere in the system. Such strong ordering is only needed rarely in *ISIS*. To overcome this problem, our protocols achieve varying degrees of order, and have latency that varies accordingly. Moreover, unlike the previously reported work, our protocols are integrated with a mechanism for dealing with failure and recovery at the level of individual processes. We now summarize our approach, but omit the detailed protocols and correctness proofs, which can be found in [Birman-c]. Fig. 1. illustrates a scenario in which two clients interact with a process group while its membership changes dynamically; the interactions are labeled with the type of primitive that would probably be used.

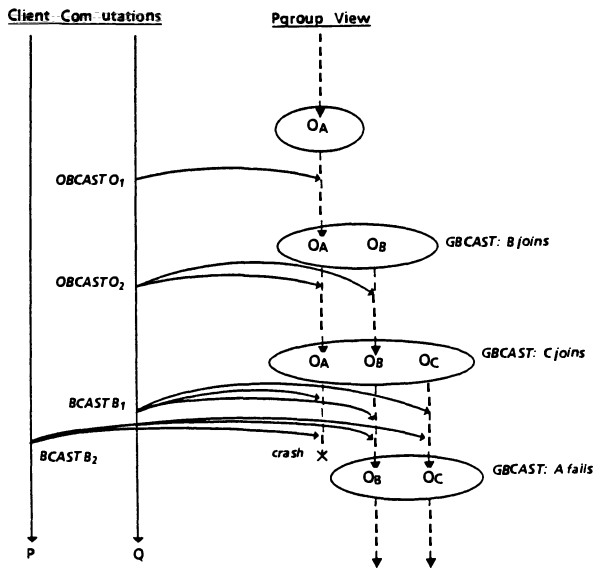


Fig. 1: Two clients interact with a process group using the broadcast primitives.

5.2.1. Broadcast primitives

The GBCAST primitive

GBCAST (group broadcast) is the most constrained, and costly, of the three primitives. We will say that the operational components of an object form a *process group*. *GBCAST* transmits information about failures and recoveries to process group members. A recovering component uses *GBCAST* to inform the operational ones that it has become available. Additionally, when a component fails, the system arranges for a *GBCAST* to be issued to group members on its behalf, informing them of its failure. Arguments to *GBCAST* are a message and a process group identifier (a capability on the resilient object), which is automatically translated into a set of destinations.

Our *GBCAST* protocol ensures that if any process receives a broadcast b before receiving a *GBCAST* g, then all overlapping destinations will receive b before g. This is true regardless of the type of broadcast b. Moreover, when a failure occurs, the corresponding *GBCAST* message is delivered after any other broadcasts from the failed process. Each component can therefore maintain a *view* listing the membership of the process group, updating it when a *GBCAST* is received. Although views are not updated simultaneously (in realtime), all components observe the same sequence of view changes. Moreover, all components will receive a given broadcast message in the same view[4].

Intuitively, the view represents a logical state in which the message arrived simultaneously at all available components. This may not be the same as the set of operational components, because some may still be executing the recovery algorithm (Sec. 5.3.2). A component of a resilient object can take advantage of this to pick a strategy for processing an incoming request, or to react to failure or recovery without running any special protocol first. Although these other components may not have received the message yet or observed the failure or recovery, since the

[4]A problem arises with this definition if a process p fails without receiving some message after that message has already been delivered to some other process q: q's view would show p to be operational; hence, q will assume that p received the message, although p is physically incapable of doing so. However, the state of the system is now equivalent to one in which p did receive the message, but failed before acting on it. In effect, there exists an interpretation of the actual system state that is consistent with q's assumption.

broadcast primitives are atomic they will eventually do so, and since the *GBCAST* ordering is the same everywhere their actions will all be consistent. Notice that *GBCAST* provides an inexpensive way to determine the last site that failed: process group members simply record each new view on stable storage; a simplified version of the algorithm in [Skeen-a] can thus be executed when recovering from failure.

The BCAST primitive

The *GBCAST* primitive is too costly to be used for general communication between the process group members that make up a resilient object. This motivates the introduction of weaker (less ordered) primitives which might be used in situations where a total order on broadcast messages is not necessary. Our second primitive, *BCAST*, satisfies such a weaker constraint. Specifically, it is often desired that if two broadcasts are received in some order at a common destination site, they be received in that order at all other common destinations, even if this order was not predetermined. For example, the *ISIS* heap facility maintains replicated allocation and free lists for each transaction by transmitting each heap operation to all copies; since the operations are done in the same order everywhere, the lists are mutually consistent. The primitive *BCAST*(*msg*, *label*, *dests*), where *msg* is the message and *label* is a string of characters, provides this behavior. Two *BCAST*'s having the same label are delivered in the same order at all common destinations. A *BCAST* having the label '*' is ordered with respect to all other *BCAST*'s. On the other hand, *BCAST*'s with different labels can be delivered in arbitrary order. This relaxed synchronization results in potentially better performance.

The OBCAST primitive

Our third primitive, *OBCAST* (ordered broadcast), is weakest in the sense that it involves less distributed synchronization then *GBCAST* or *BCAST*. *OBCAST*(*msg*, *dests*) atomically delivers *msg* to each operational *dest* so that if one process sends multiple messages to the same destination, they are delivered in the order they were sent. Delivery ordering is unconstrained if two broadcasts originate in different processes or are issued concurrently within a single process. More

specifically, if there exists a chain of message transmissions and receptions or local events by which knowledge could have been transferred from the point at which the first broadcast was issued to the point at which the second one was issued, we consider the broadcasts to be *potentially causally related*, and the delivery ordering will respect the order of transmission. For causally independent broadcasts, the delivery ordering is not constrained.

OBCAST is valuable in *ISIS* because resilient objects employ concurrency control algorithms for distributed synchronization. A consequence is that if two computations communicate concurrently with the same process, the messages are almost always independent ones that can be processed in any order: otherwise, concurrency control would have caused one to pause until the other was finished. On the other hand, order is clearly important within a causally-linked series of broadcasts, and it is precisely this sort of order that *OBCAST* respects.

5.2.2. Other broadcast abstractions

A weaker broadcast primitive is reliable broadcast, which provides all-or-nothing delivery, but no ordering properties. The formulation of *OBCAST* in [Birman-b] actually includes a mechanism for performing broadcasts of this sort, hence no special primitive is needed for the purpose. Additionally, there may be situations in which *BCAST* protocols that also satisfy an *OBCAST* ordering property would be valuable. Although our *BCAST* primitive could be changed to respect such a rule, when we considered the likely uses of the primitives it seemed that *BCAST* was better left completely orthogonal to *OBCAST*. In situations needing hybrid ordering behavior, the protocols of [Birman-b] could easily be modified to implement *BCAST* in terms of *OBCAST*, and the resulting protocol would behave as desired.

5.2.3. Synchronous versus asynchronous broadcast abstractions

Many systems employ RPC internally, as a lowest level primitive for interaction between processes (this type of RPC should not be confused with the high-level RPC primitive used to communicate with and between resilient objects). It should be evident that all of our broadcast

primitives can be used to implement replicated remote procedure calls [Cooper]: the caller would simply pause until replies have been received from all the participants (observation of a failure constitutes a reply in this case). We term such a use of the primitives *synchronous*, to distinguish it from from an *asynchronous* broadcast in which no replies, or just one reply, suffices.

In *ISIS*, *GBCAST* and *BCAST* are normally invoked synchronously, to implement a remote procedure call by one component of an object on all the members of its process group. However, *OBCAST*, which is the most frequently used overall, is almost never invoked synchronously. Asynchronous *OBCAST's* are the source of most concurrency in *ISIS*: although the delivery ordering is assured, transmission can be delayed to take advantage of piggybacking or to schedule I/O within the system as a whole. While the system cannot defer such a broadcast indefinitely, the ability to defer it a little, without delaying some computation by doing so, permits load to be smoothed. As observed above, although concurrency is introduced by the primitive, it respects the delivery orderings on which a computation might depend, and is ordered with respect to failures, so this concurrency does not complicate higher level algorithms. Moreover, the protocol itself is extremely cheap.

A problem is introduced by our decision to allow asynchronous broadcasts: the atomic reception property must now be extended to address causally related sequences of asynchronous messages. If a failure were to leave a "gap" in such a sequence, such that some broadcasts were delivered to all their destinations but others that precede them were not delivered anywhere, inconsistency might result even if the destinations do not overlap. We therefore extend the atomicity property as follows. If process t receives a message m from process s, and s subsequently fails, than the state of t may depend on any message m' received by s before it sent m. Therefore, unless t fails as well, m' must be delivered to its remaining destinations. The cost of the protocols are not affected by this change.

A second problem arises when the user-level implications of this atomicity rule are considered. In the event of a failure, any suffix of a sequence of aysnchronous broadcasts could now

be lost and the system state would still be internally consistent. A coordinator that is about to send a top-level reply or take some action that may leave an externally visible side-effect will therefore need a way to pause until all such broadcasts have actually been delivered. For this purpose, a **flush** primitive is provided within the object specification language. Notice that occasional calls to **flush** do not eliminate the benefit of using *OBCAST* asynchronously. Unless the system has built up a considerable backlog of undelivered broadcast messages, which should be rare, **flush** will only pause while transmission of the last few broadcasts completes. **Flush** is automatically invoked when a log entry is written.

5.3. Fault-tolerant implementation of selected operations

In this section, implementations are described for some of the operations that occur most frequently within *ISIS*, using the primitives given above. In the interest of brevity, only a small subset of *ISIS* is presented.

5.3.1. Object invocation and request processing; commit and abort

To issue a request to an object, a task first generates the transaction id under which the desired operation should be executed. If a non-resilient process is performing the RPC, a new top-level transaction is created and a unique identifier is assigned as its TID. If a task with TID **x** does a series of RPC's, TID's for the resulting subtransactions are formed by extending **x** with an index: **x.1**, **x.2**, etc. The branches of a cobegin are assigned TID's in the same manner. Finally, if a *toplevel* statement is executed, a TID is generated as for a subtransaction but the prefix is flagged as a top-level event.

Having determined the TID, the caller asynchronously *OBCAST*'s the RPC to the components of the destination object. A *capability management* facility translates the capability into a list of process addresses for transmission[5]. The caller then waits for a single reply, which could come from any component of the object, or even arrive in duplicate because of failures.

[5]An inexpensive protocol to maintain a *cache* of group addressing information, updating it if it is found to be out of date during message transmission, is given in [Birman-c].

Duplicates are discarded.

Upon receiving the RPC, a component must determine if it is the coordinator. All components of each object are statically ordered by site number into a ring. A component computes its *ranking* as the distance along the ring from the site where the RPC originated; the coordinator is defined to be the lowest ranked operational component. This tends to place the coordinator at the same site as the originator, which is desirable because it minimizes the latency incurred before a result can be returned. The new coordinator returns a retained result if one is found. Otherwise, it executes the new request and asynchronously *OBCAST*'s the result to the caller and its cohorts. A cohort watches the coordinator for failure, which it detects by reception of a *GBCAST* message, and recomputes the ranking if one occurs. Since all components have the same view when an RPC is received, and all subsequently see the same sequence of failures and recoveries, the computed rankings are mutually consistent. Note that all necessary synchronization is provided by the communication primitives.

Now, consider task termination. For each task, a *capability list* (CLIST) is maintained, containing the capabilities of objects whose components should be informed when the task commits or aborts. The coordinator uses *OBCAST* to asynchronously send a commit or abort message to the objects in the CLIST. A CLIST is initially empty; a capability is added when an RPC is issued to an object. Additionally, when a reply is received from a committed subtransaction, the CLIST for that subtransaction is piggybacked on the reply and merged with that of the caller (unless the subtransaction executed in a **toplevel** statement, in which case its CLIST is discarded when it commits).

On reception, a commit or abort message for transaction T is delayed if some subtransaction of T is still active. This makes it possible for a subtransaction to reply to its caller before issuing its own commit or abort, a tactic that reduces latency and ensures that at least one copy of the reply will reach the caller (a duplicate might be sent if the coordinator fails after sending the reply but before sending the commit). After all subtransactions have terminated, retained results

corresponding to *T* are deleted, and the local lock manager and version-stack managers are informed of the event. When a **kill** is received, if the coordinator is doing a restart it waits until restart is completed (to ensure that the CLIST is accurate). Since restart is done without blocking, it will terminate. **Kill** is then forwarded to any active subtransactions, and an abort is performed.

5.3.2. Recovery from partial failures

To initiate recovery, a component issues a *GBCAST* to the operational components of the object to which it belongs. When this message is received, any component transfers its state to the recovering one: since the states of the operational components are determined by the messages they have received, and each has received the same set of messages, all are in the same (logical) state. This *GBCAST* can thus be thought of as a synchronous RPC that returns the current state of the object and has the side-effect of modifying the process-group view to include the recovered component. The total ordering of *GBCAST* with respect to other broadcasts provides all the necessary synchronization.

5.3.3. Managing replicated locks

The locking facilities discussed earlier are easily implemented using our broadcast primitives. A read-lock is first obtained locally by the coordinator of a computation. Then, a *read-lock registration* message is asynchronously *OBCAST* to the other copies of the data item. The coordinator immediately continues execution, as if its read-lock were already replicated, although the message may not actually have been delivered anywhere. If the coordinator fails and *any* process had received a message *m* sent after the lock acquisition, the read-lock will be registered before the failure can be "detected" by the cohorts managing other copies of the lock. This follows because the read-lock registration precedes *m* and hence must be delivered despite the failure, whereas (by definition) the *GBCAST* follows *m* and hence must be delivered after the lock registration. Because the read-lock registration message is small and asynchronous, piggybacking such messages on outgoing updates and RPC messages is particularly effective.

Unlike a read-lock, a write-lock must be granted explicitly by all components of an object, except in certain special cases[6] described in [Raeuchle]. Moreover, a write-lock request can be performed correctly whether or not other broadcasts issued by the computation have been delivered, hence the request is not subject to the *OBCAST* type of ordering constraint. Note, though, that if two write-lock requests are issued concurrently (on the same item), they could deadlock simply by being granted in different orders at different sites. This is just the type of ordering problem addressed by *BCAST*. To acquire a write-lock, the request is synchronously *BCAST* using the identifier of the data item as a *BCAST* label. If a component fails during the protocol, the caller withdraws the partially acquired write-lock and then rerequests it. Since the read-lock registration message preceded the *GBCAST* announcing the failure, either it is delivered before the *GBCAST*, or no site received a message from a failed coordinator after it obtained the lock. Because of the withdrawal rule, the write-lock is rerequested *after* the *GBCAST* message is received, so it will be forced to wait if the coordinator held a read lock and that lock survived the crash. Moreover, since *BCAST* is delivered in the same order everywhere, concurrent write-lock requests will not deadlock.

5.3.4. Updating replicated data

Read operations are satisfied from the version stack for the local copy of the data item being accessed. Three implementations are supported for write operations.

Synchronous update.

For this method, *OBCAST* is used to synchronously transmit the new value to all operational components. The method is only used for experimental evaluation of the effect of asynchronous data transmission on performance, as reported in Sec. 7. Note, however, that if *ISIS* used a quorum replication method, both read and write operations would effectively be synchronous. Thus, the performance of synchronous update sheds light on the performance attainable with a

[6]The most important of these is that, since coordinators for a single transaction are run at the same site, after a transaction has acquired a distributed write-lock on an item x, its sub-transactions need only lock x locally.

quorum replication method.

Concurrent update.

Although synchronous update is conceptually simple, costly delays are incurred while waiting for acknowledgements. Using *concurrent update*, data are updated locally by the coordinator, which issues an asynchronous *OBCAST* to inform its cohorts [Joseph-b]. An asynchronous *OBCAST* is also used to commit, at which time locks are released. Since the updates precede the commit and *OBCAST* respects this ordering, any process that obtains a lock will observe the correct version of the data it reads. Thus, the semantics of the synchronous update are preserved but, if few write-locks are needed, the response time is limited by the *local* execution speed of the request! Recall that when concurrent update is in use, it may be necessary to invoke **flush** before returning a result from a top-level operation or taking an action with external side-effects.

Delayed updates

The concurrent update scheme assumes a *pessimistic* write-locking algorithm, which waits for responses from all operational components each time a write-lock is needed. Pessimistic locking permits the programmer to design a deadlock-free object and hence to implement objects that take irreversible actions. However, better performance can sometimes be obtained using an *optimistic* locking algorithm together with *delayed updating*. Write-locks are acquired *locally* by the coordinator, which queues update messages but does not transmit them. When the transaction is prepared to commit, it attempts to acquire these write locks from its cohorts using the protocol of Sec. 5.3.3. The transaction aborts (discarding its queued updates) if deadlock would result. Otherwise, it transmits the updates using *OBCAST*.

Using delayed updates, the possibility of an occasional abort is accepted as an alternative to issuing multiple write-lock requests -- only one distributed concurrency control action is needed, and it occurs at the end of the transaction. Moreover, other transactions can read old versions of any data items being updated (but only at remote sites) and multiple updates could be sent in each message. These benefits have a cost: large amounts of buffering may be needed to support the

technique, and irreversible actions are precluded. The *ISIS* prototype will be used to compare delayed and concurrent update in the future. Both update methods have been proved correct for objects that use conflict-serializability as a correctness constraint [Joseph-a]. An open problem is to investigate the applicability of these techniques in system which employ other correctness constraints.

6. System architecture and Implementation issues

6.1. Communication primitives

We now summarize the architecture and implementation of the *ISIS* communication subsystem. The primitives are built in layers, starting with a "bare" network providing unreliable datagrams. A site-to-site acknowledgement protocol converts this into a sequenced, error-free message abstraction, using timeouts to detect apparent failures. An agreement protocol is then used to convert the site-failures and recoveries into an agreed upon ordering of events. If timeouts cause a failure to be detected erroneously, the protocol forces the affected site to undergo recovery.

Built on this is a layer that supports the various primitives. *OBCAST* has a very light-weight implementation. Each process buffers copies of any messages needed to ensure the consistency of its view of the system. If message m is delivered to process p, and some message m' precedes m, a copy is sent to p also. Thus, if any chain of process-to-process interactions leads to the intended recipient of a message, the message will travel down that chain and can be delivered (duplicate copies are discarded). An inexpensive garbage collector tracks down and deletes superfluous copies after a message has reached all its destinations. By using extensive piggybacking and a simple scheduling algorithm to control message transmission, the cost of an *OBCAST* is kept low -- often, less than one packet per destination. *BCAST* employs a two-phase protocol based on one suggested to us by Skeen [Skeen-b]. This protocol has higher cost than *OBCAST* because delivery can only occur during the second phase; *BCAST* is thus inherently synchronous. Recall, however, that *ISIS* uses *BCAST* primarily for write-lock acquisition, which can be done rarely. Moreover,

BCAST is only needed the first time a computation write-locks a particular variable; subsequent attempts to re-lock it (not uncommon in nested transactions) can be handled locally. *GBCAST* is implemented using a two-phase protocol similar to the one for *BCAST*, but with an additional mechanism that flushes messages from a failed process before delivering the *GBCAST* announcing the failure. Although *GBCAST* is slow, it is used very rarely. More details and correctness proofs appear in [Birman-c].

6.2. Higher level system structure

The *ISIS* prototype was built under UNIX 4.2, and is organized hierarchically, as illustrated in Fig. 2. The lowest level provides the communication primitives described earlier, together with a message "editing" subsystem supporting variable-format messages with symbolically named message-fields. Built on top of this is the a layer supporting concurrent tasks, monitors for mutual exclusion, the transactional version stack, the lock manager, the capability manager [Dietrich],

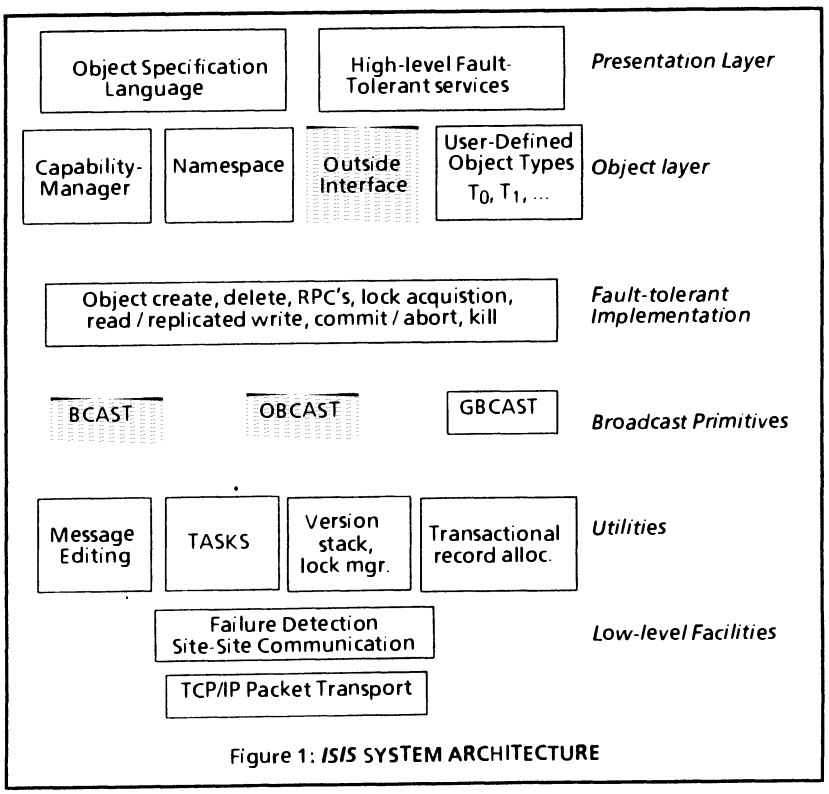


Figure 1: *ISIS* SYSTEM ARCHITECTURE

which maps a capability on an object to a list of sites where its components reside, the name-space, which maps symbolic names to capabilities, and the interface used by external non-resilient processes to issue requests to resilient objects. The capability manager supports dynamic migration of objects, although we do not yet exploit this possibility.

We originally feared that processes and inter-process communication would be the dominant cost factor in *ISIS*. Consequently, a single system process handles functions common to all resilient objects, and a single "type manager" is used for each resilient type. A type manager multiplexes its time between the different instances of its type residing at the site where it is executing; these in turn multiplex their time among currently active tasks. Process creation occurs only when a new type manager must be started (this idea was suggested in [Lazowska].) Commands to interactively load and unload type managers (e.g. when a new type is defined) are provided by the system process.

In retrospect, we feel that the decision to multiplex type managers was an error. The increased code complexity required to keep separate copies of the various data structures used in the type manager for each instance was not justified by the reduced overhead that resulted. In any future version of *ISIS*, each object will be represented by a single process at every site where it resides and the runtime system will be fragmented into multiple processes: a process group manager, a protocols process, a failure detector, communication buffering processes, etc. We now believe that this would reduce complexity and that adverse performance impact can be minimized with careful tuning. We also believe that if *ISIS* is to perform well, it must be moved away from UNIX, and are planning to do so in the future.

7. Performance of the prototype

A prototype of *ISIS* has been operational since January 1985. Performance is reported for a cluster of SUN 2/50 workstations interconnected by a 10-Mbit ethernet (Table 1). Our approach was to evaluate the performance of the communication primitives, the response time for some simple resilient objects, and the overall response time of the system when presented with

concurrent requests at multiple sites. The indexed sequential file, built from a resilient directory and a resilient file, illustrates the overhead associated with nesting.

When we instrumented *ISIS*, we discovered that the performance of our IPC connections was suboptimal, primarily because the version of UNIX we used did not support changes to the IPC buffer size, which was consequently too small to permit effective "windowing". We lacked the

GENERAL PERFORMANCE	COMPONENT TESTED	SUN 2/50		
Site-to-site message	Delay to reception	8ms		
Process-to-process message	Delay to reception	10ms		
RPC to object, same site	Delay until task starts	30ms		
Version stack:	BEGIN / COMMIT	19ms		
(volatile)	BEGIN / READ / COMMIT	20ms		
	BEGIN / WRITE / COMMIT	23ms		
(stable)	BEGIN / COMMIT	467ms		
	BEGIN / READ / COMMIT	493ms		
	BEGIN / WRITE / COMMIT	880ms		
Lock manager	Acquire local lock	0.7ms		
COMMUNICATION PRIMITIVES		**1 site**	**3 sites**	**6 sites**
Failure detector	Timeout	n.a.	7 secs	7 secs
GBCAST	Delay until delivered	n.a.	2 secs	3 secs
OBCAST	Latency	10ms	32ms	44ms
	Turnaround	18ms	165ms	360ms
	Throughput (one task)	10 / sec	6 / sec	3.5 / sec
	Effective Throughput	35 / sec	24 / sec	17 / sec
	System Throughput	> 100 / sec	18 / sec	10 / sec
BCAST	Latency	10ms	180ms	240ms
	Turnaround	20ms	180ms	360ms
	System Throughput	> 100 / sec	20 / sec	12 / sec
Write lock	Delay to acquisition	30ms	220ms	400ms
Log manager	Write log record	55ms	225ms	410ms
RESILIENT OBJECTS*				
Resilient file	READ	13 / sec	11 / sec	11 / sec
	WRITE (synchronous)	4.2 / sec	1.27 / sec	.75 / sec
	WRITE (concurrent)	12.5 / sec	11 / sec	9 / sec
Resilient directory	BIND (synchronous)	2.9 / sec	.9 / sec	.57 / sec
	BIND (concurrent)	6.3 / sec	6.3 / sec	5 / sec
	LOOKUP (read only)	11 / sec	10 / sec	11 / sec
Resilient stack	PUSH (synchronous)	2.7 / sec	1.1 / sec	.52 / sec
	PUSH (concurrent)	9.2 / sec	9.3 / sec	9.6 / sec
	POP (synchronous)	3.3 / sec	1.7 / sec	1.0 / sec
	POP (concurrent)	9.2 / sec	10.5 / sec	8.9 / sec
Indexed seq. file	INSERT (synchronous)	1.6 / sec	.52 / sec	.3 / sec
	INSERT (concurrent)	3.8 / sec	5. / sec	2.3 / sec
	LOOKUP	9.5 / sec	9.5 / sec	9.5 / sec

*Partial recovery mode.

Table 1: Performance in the ISIS Prototype

resources to correct this problem.

The first set of figures addresses performance of the version store and lock manager. These show that while the version store is very fast in its in-core partial recovery mode, it degrades in the disk-based "stable" storage mode. This supports our decision to favor log-based recovery from total failures, since the use of stable storage is minimized in this manner. Consequently, the resilient objects tested were run in the in-core mode only.

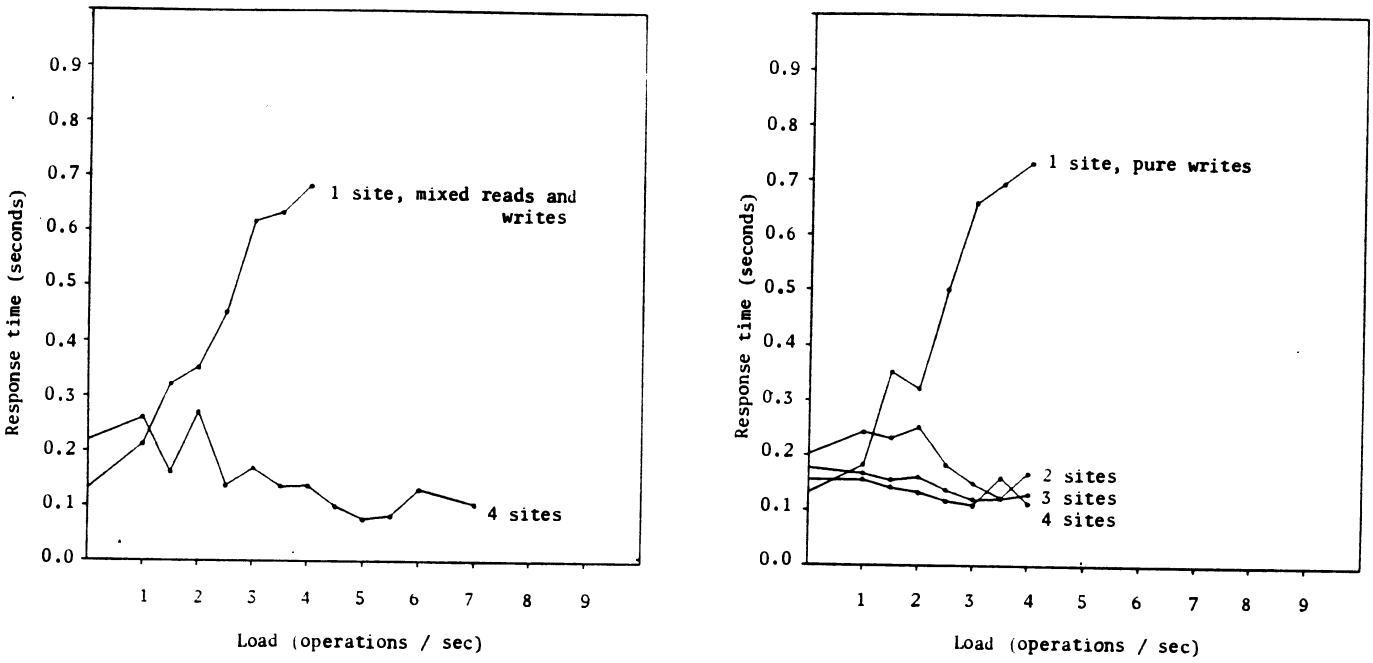
The broadcast primitives are dominated by underlying message-passing costs, but otherwise depend primarily on the number of phases required. In the initial implementation of the primitives, all run in two phases (although the message is delivered during the first one for *OBCAST* and the second for *BCAST*), hence all the primitives give similar performance. The *latency* figure measures the time from message transmission to remote delivery. Because the *OBCAST* implementation we instrumented is not identical to the one described in this paper, the *OBCAST* latency is very high. Moreover, the latency figure turned out to be very hard to measure: using a 60Hz line-clock, which is the only one available on our SUN workstations, elapsed time can only be measured to an accuracy of 16ms. Nonetheless, the *OBCAST* latency (32ms in the 3-site case) is much larger than the inter-site latency (10ms). We found that this results from delay associated with the I/O operation that occurs when an *OBCAST* recipient acknowledges delivery to the initiator. Additional latency is introduced by the small window size, and the inaccurate clock further inflates the *OBCAST* figure. We are confident that after we reimplement the primitives using the algorithms given in [Birman-c], *OBCAST* latency will not be much higher than the site-site latency of 10ms.

Turnaround measures the delay from transmission to reception of a reply from the remote task that received the message, and *throughput* measures the rate at which a single task can issue broadcasts without waiting for acknowledgements, in messages per second. The effective throughput is 3 to 5 times higher than this, because concurrent update permits multiple update messages to be piggybacked on a single packet (notice that the effective throughput decreases

more slowly than the true throughput as the number of sites increases: while waiting for acknowledgments, there is time to generate more concurrent update messages, hence the degree of piggybacking rises). We also measured the *system throughput*, which is the maximum number of *BCAST* or *OBCAST* protocols that can be started per second at a site in a steady state (this figure could be improved by tuning the UNIX scheduling policy). Note that the cost of the protocols rises linearly with the number of destinations, as least when the number of destinations remains small.

Turning to the resilient objects themselves, we see the dramatic performance impact of the concurrent update technique when compared with synchronous update. These tests measured the average cost per operation for a transaction doing 25 operations of the designated type. Concurrency control overhead is higher for the first operation than for subsequent ones, which the system recognizes as being "covered" by previously acquired locks. The amortized cost is therefore low, permitting bursts of 10-12 operations per-second even when updating was being done (again, assuming an otherwise idle system). The fact that concurrent update does better than synchronous update even in the single-site case is because concurrent update is also used to maintain message routing tables in the type managers. Nesting did not introduce any substantial overhead. Within the system, most time is spent sending and receiving messages and in the object itself, executing the requested operation.

Finally, we measured the performance of the file object under a distributed load. Concurrency control was not included, in order to isolate the effect of replication from other factors. Two types of tests were undertaken. First, we considered a "mixed" transaction that performed 3 reads before doing a write and committing. The file object was replicated at 1 and 4 sites, and varying loads of requests were presented randomly at each site. Figure 3a shows the mean response time for several thousand requests, for loads ranging from 0 to 7 operations per second. Each curve stops when the system saturated and began to develop a request backlog. A component of the file object does two sorts of work when processing these requests: computing related



Figures 3a,3b: Responsiveness of a file object as a function of its workload

to the coordinator side of each operation, and work stemming from its role as cohort in requests initiated remotely. The latter involves processing the initial RPC message, the message containing the data for the write, and the commit; the former involves generating these messages and interacting with the external client programs, in addition to executing the operation itself.

The data we plotted was obtained by correlating response time for individual requests with the times at which read and write requests were serviced by the file object. The overall load on the object was deduced by measuring the rate of local reads and writes and adding 4 times the rate of updates received from remote sites, to account for the 3 reads that were done remotely for every update sent out. Note that piggybacking makes it possible for a cohort to do quite a bit of work for each message it receives from the runtime system; in the case of the coordinator this is generally not the case. It is interesting to observe that except when the load on the object was very low, response time in the 4-site case is better than that which can be achieved with a non-replicated object. This effect can be explained by the sharing of coordinator and interface related

activity among the components of the object. Moreover, the maximum capacity of the object to perform operations rises from 4.25 operations per second in the non-replicated case to 7 operations per second. As the load rises, piggybacking increases the efficiency of the system, explaining why the response time drops from about .25 seconds to .1 seconds for a typical operation.

We wondered what would happen if transactions did only writes. Figure 3b shows how response time varies as a function of load for a transaction that does one write and then commits. In the single-site case the performance of this transaction is close to that for the single-site mixed case (writes and reads have comparable local costs); in all the replicated cases, however, response time improves as the object is placed under increasing load (the saturation point is approximately the same, however). This better response time is explained purely by the reduced coordinator-related and front-end work being done by the system. Of course, the cost of running the broadcast protocols rises with the number of sites, and performance would undoubtably drop again for objects replicated at very large numbers of sites.

The major conclusion to draw from the above is that when using concurrent update, the apparent performance of a resilient object accessible from multiple sites can be comparable or better than for a fault-intolerant single-site object of the same type (our experience with the calendar program supports this). Moreover, overall performance is higher in read-intensive settings, provided that requests arrive randomly at the different components and the concurrency control algorithm is good, since reads are done locally. On the negative side, the steadily increasing costs of the protocols, especially *BCAST*, suggests that data should not be replicated to more than 3 or 4 sites because concurrency control overhead could become excessive. This has lead us to implement a data migration mechanism for *ISIS*, which will be described elsewhere [Dietrich]. Our figures demonstrate that *ISIS* is able to provide powerful distributed services at suprisingly low cost. If an effort were made to tune the *ISIS* prototype and the objects themselves, performance could probably be doubled under UNIX, and further improved by moving to a more streamlined operating system.

8. Future research

The *ISIS* project is now entering its third year. Two major problems are receiving attention: mechanisms for increasing availability during partitioning, and an investigation of the limits of concurrency in systems subject to ordering-based correctness constraints [Joseph-a]. Simultaneously, we are examining uses for *ISIS* in high-level programming tools, which might constitute the interface to a new generation of operating system services. Also being studied are facilities for dealing with real-time events, replicated processing (as opposed to replicated data), and demand-based data migration within k-resilient objects replicated at more than $k+1$ sites. We would also like to build some sort of application system using *ISIS* as its base, for example a critical care system for medical environments [Birman-d].

Resilient objects are too high-level for many purposes. For example, if all updates to a given variable originate at a single site, there are cheaper ways to maintain that variable then to adopt a general purpose transaction mechanism. Recognizing this, we now expect to use *ISIS* in an environment that would also permit programmers to work directly with fault-tolerant process groups. Users could then construct fault-tolerant software using whichever tools seem most convenient.

A basic problem is that *ISIS* provides a type of service and exhibits a collection of requirements which are very different from those seen in most contemporary distributed programs. For example, UNIX assumes that interactions between processes will be through RPC or virtual circuits, whereas communication in fault-tolerant distributed systems is strongly biased towards broadcast protocols. A result is that UNIX is simply not very good at running our software (the V system [Cheriton] might be more reasonable, although we have yet not considered porting *ISIS* to it). Clearly, it is beyond our resources to conduct research into both fault-tolerance and operating system design. It thus seems appropriate to call for renewed research into primitives and computational models at the operating system level, and for greater cooperation between the designers of these two mutually dependent classes of system.

9. Conclusions

This paper presented an overview of the *ISIS* project and reviewed the techniques it uses to obtain fault-tolerant implementations from abstract type specifications. The good performance of a prototype supports our belief that the approach will be viable in diverse situations. Moreover, a novel communication architecture leads to a system structure within which correctness arguments are straightforward despite the presense of failures and concurrency.

We believe that a new generation of high-level computing facilities, including *ISIS*, is now emerging. Much as virtual memory changed the engineering of very large systems in a fundamental way, these facilities will fundamentally change the way that distributed software is developed, and will thereby enable research in areas for which existing programming methodologies are inadequate. As the complexity and sheer size of distributed systems continues to grow, facilities of this sort will be indispensable.

10. Acknowledgement

Wally Deitrich, Amr El Abbadi, Tommy Joseph, Thomas Raeuchle, and Pat Stephenson all made many contributions to the work reported here. We are also grateful to Dale Skeen, who founded the project with us in 1981, and to Fred Schneider for his careful reading of an early draft.

11. References

[Allchin] Allchin, J., McKendry, M. Synchronization and recovery of actions. *Proc. 2nd ACM SIGACT/SIGOPS Principles of Distributed Computing*, Montreal, Canada, 1983.

[Bartlett] Bartlett, J. A NonStop Kernel. *Proc 8th Symposium on Operating Systems Principles*, Dec. 1981.

[Bernstein] Bernstein, P., Goodman, N. Concurrency control algorithms for replicated database systems. *ACM Computing Surveys* **13**, 2 (June 1981), 185-222.

[Birman-a] Birman, K. Replication and fault-tolerance in the ISIS system. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles*. Orcas Island, Washington, Dec. 1985, 79-86.

[Birman-b] Birman, K., *et. al.* Implementing fault-tolerant distributed objects. *IEEE TSE-11*, 6, (June 1985), 502-508.

[Birman-c] Birman, K., Joseph, T. Reliable communication in an unreliable environment. Dept. of Computer Sci-

ence, Cornell Univ., TR 85-694, Aug. 1985.

[Birman-d] Birman, K. *et al.* MDB-1: A database system for medical applications. In *Proc. IEEE Computers and Cardiology* (Sept.1984), 309-312.

[Birrel] Birrel, A., Nelson, B. Implementing remote procedure calls. *ACM TOCS* **2**, 1 (Feb 1984), 39-59.

[Borg] Borg, A. *et. al.* A message system supporting fault-tolerance. *Proc. 9th Symposium on Operating Systems Principles*, Bretton Woods, NH (Oct. 1983), 90-99.

[Chandy] Chandy, K., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems* **3**, 1 (Feb 1985), 63-75.

[Chang] Chang, J., Maxemchuck, M. Reliable broadcast protocols. *ACM TOCS* **2**, 3 (Aug. 1984), 251-273.

[Cheriton] Cheriton, D. The V Kernel: A software base for distributed systems. *IEEE Software* **1** 12, (1984), 19-43.

[Cooper] Cooper, E. Replicated procedure call. *Proc. 3rd ACM Symposium on Principles of Distributed Computing.*, August 1984, 220-232. (May 1985).

[Cristian] Cristian, F. *et al* Atomic broadcast: From simple diffusion to Byzantine agreement. IBM Technical Report RJ 4540 (48668), Oct. 1984.

[Dietrich] Dietrich, W. Ph.D. dissertation, forthcoming.

[El] El Abbadi, A., Skeen, D., Cristian, F. An efficient fault-tolerant protocol for replicated data management. *Proc. 4th ACM Symp. on PODS*, Portland, Oregon, March 1985, 215-229.

[El] El Abbadi, A., Toueg, S. Availability in partitioned, replicated databases. TR 85-721, Dept. of Computer Science, Cornell University, Dec. 1985. To appear: *Proc. 5th ACM Symp. on PODS*, Boston MA, March 1986.

[Eswaren] Eswaren, K.P., *et al* The notion of consistency and predicate locks in a database system. *Comm. ACM* **19**, 11 (Nov. 1976), 624-633.

[Gawlick] Gawlick, D. Processing "hot spots" in high performance systems. Amdahl Corp., Sunnyvale, 1984.

[Gifford] Gifford, D. Weighted voting for replicated data. *Proc. 7th ACM-SIGOPS Symposium on Operating Systems Principles*, December, 1979.

[Gray] Gray, J. Notes on database operating systems. *Lecture notes in computer science* **60**, Good and Hartmannis, eds., Springer-Verlag 1978.

[Herlihy] Herlihy, M. A quorum consensusa replication method for abstract data types. *ACM TOCS* **4**, 1 (Feb 1986), 32-53.

[Joseph-a] Joseph, T. Low cost management of replicated data. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca (Dec. 1985).

[Joseph-b] Joseph, T., Birman, K. Low cost management of replicated data in fault-tolerant distributed systems. *ACM TOCS* **4**, 1 (Feb 1986), 54-70.

[Lamport] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM* **21**, 7, July 1978, 558-565.

[Lazowska] Lazowska, E. *et al* The architecture of the EDEN system. *Proc. 8th Symposium on Operating Systems Principles*, Dec. 1981, 148-159.

[Liskov-a] Liskov, B., Zilles, S.N. Programming with abstract data types. *SIGPLAN notices* **12**, 2 (Apr. 1974), 50-59.

[Liskov-b] Liskov, B., Scheifler, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM TOPLAS* **5**, 3 (July 1983), 381-404.

[Moss] Moss, E. Nested transactions: An approach to reliable, distributed computing. Ph.D. thesis, MIT Dept of EECS, TR 260, April 1981.

[Papa] Papadimitrou, C. The serializability of concurrent database updates. *JACM* **26**, 4 (Oct. 1979), 631-653.

[Popek] Popek, G. *et al.* Locus: A network transparent, high reliability distributed system. *Proc. 8th Symposium on Operating Systems Principles*, Dec. 1981, 169-177.

[Raeuchle] Raeuchle, T. Ph.D. dissertation, forthcoming.

[Schlicting] Schlicting, R, Schneider, F. Fail-stop processors: An approach to designing fault-tolerant distributed

computing systems. *ACM TOCS* **1**, 3, August 1983, 222-238.

[Schneider] Schneider, F., Gries, D., Schlicting, R. Fault-tolerant broadcasts. *Science of Computer Programming* **4**, 1 (Jan. 1984).

[Skeen-a] Skeen, D. Determining the last process to fail. *ACM TOCS* **3**, 1, Feb. 1985, 15-30.

[Skeen-b] Skeen, D. A reliable broadcast protocol. *Unpublished.*

[Spector] Spector, A., *et al* Distributed transactions for reliable systems. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles*, Dec. 1985, 127-146.

[Strong] Strong, H.R., Dolev, D. Byzantine agreement. *Digest of papers, Spring Compcon 83*, San Francisco, CA, March 1983, 77-81.

[Svobodova] Svobodova, L. Resilient distributed computing. *IEEE TSE* **TSE-10**, 3 (May 1984), 257-268.

[Toueg] Toueg, S., Babaoglu, O. On the optimal checkpoint selection problem. *SIAM J. Computing* **13** 3 (Aug. 1984).

[Weihl] Weihl, W. Data dependent concurrency control and recovery. *Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983, 63-75.