Replication and Fault-Tolerance in the ISIS System[1]

Kenneth P. Birman

TR 85-668
March 1985
(Revised September 1985.)

Department of Computer Science
Cornell University
Ithaca, NY 14853

[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 author and should not be construed as an official Department of Defense position, policy, or decision.

REPLICATION [redacted] FAULT-TOLE[redacted]CE IN THE ISIS SYSTEM[1]

Kenneth P. Birman

Department of Computer Science
Cornell University, Ithaca, New York

ABSTRACT

The *ISIS* system transforms abstract type specifications into fault-tolerant distributed implementations while insulating users from the mechanisms used to achieve fault-tolerance. This paper discusses techniques for obtaining a fault-tolerant implementation from a non-distributed specification and for achieving improved performance by *concurrently* [redacted]*g* replicated data. The system itself is based on a small set of communication primitives, which are interesting because they achieve high levels of concurrency while respecting higher level ordering requirements. The performance of distributed fault-tolerant services running on this initial version of *ISIS* is found to be nearly as good as that of non-distributed, fault-intolerant ones.

[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 author and should not be construed as an official Department of Defense position, policy, or decision.

This paper will be presented at the ACM 10th SOSP, Dec. 1985.

1. Introduction

Our basic premise is that the complexity of fault-tolerant distributed programs precludes their design and development by non-experts. This complexity seems to be inherent: systems achieve fault-tolerance [redacted] redundancy, and the distributed agreement and synchronization protocols needed to manage redundant data are hard to implement. Moreover, high levels of concurrency are required to achieve satisfactory performance, making it difficult to reason about correctness in the presense of failures. Alternatives to direct implementation of fault-tolerant systems are needed if fault-tolerance is to gain wide applicability.

The *ISIS* project seeks to address this need by automating the transformation of fault-*intolerant* program specifications into fault-tolerant implementations. This is done 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 there are many systems to assist in the construction of distributed and fault-tolerant software (c.f. ARGUS [Liskov-b], EDEN [redacted], CLOUDS [Allchin], LOCUS [Popek], TABS [Spector], TANDEM [Bartlett]), *ISIS* goes furthest in insulating users from the details of fault-tolerant progr[redacted]. Moreover, *ISIS* places few restrictions on programs. In contrast, other systems that execute programs fault-tolerantly (CIRCUS [Cooper], AURAGEN [Borg]) are restricted to deterministic programs. Such restrictions make it hard to design a fault-tolerant service that can be accessed concurrently from multiple sites -- the primary use anticipated for *ISIS*.

ISIS is also interesting because of the methodology used to build the system itself. In particular, a communication layer supporting a variety of *broadcast*[2] protocols has been implemented. These simplify the description of concurrent algorithms, and make it feasible to reason about the correctness of the system as a whole.

---

[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.

The structure of this paper is as follows. The next section introduces *resilient objects* as a basic unit of fault-tolerance. Subsequent sections review the object specification language and show how fault-tolerant implementations can be expressed in terms of the communication primitives. The paper concludes by discussing the performance of a prototype and the current goals of the project.

2. Resilient objects

ISIS extends a conventional operating system by introducing a new programming abstraction, the *resilient object*. Each resilient object provides some service at a set of sites, where it is represented by *components* to which requests can be issued using remote procedure calls [Birrel]. A typical *ISIS* application would be constructed by developing conventional front-end programs and interfacing them to one or more such objects; in effect, the objects "glue" the resulting distributed program together. The programmer can also define new, specialized resilient objects if suitable ones do not already exist.

The translation of a non-distributed specification into a resilient object is based on several assumptions about the execution environment and what resiliency should mean. These are addressed in the remainder of this section.

2.1. Assumptions about the environment

ISIS runs on *clusters* of computer systems communicating over a local area network, which must not be subject to *partitioning*, whereby the network divides into subgroups of sites between which communication is impossible. Issues relating to cluster interconnection are deferred to a future paper.

We assume that the only way sites fail is by halting (crashing) [Schlicting]. Failure detection and a collection of fault-tolerant broadcast protocols are implemented in software on top of the bare network, as described in subsequent sections. Finally, object specifications are assumed to be correct. Although *ISIS* will not crash when given a buggy specification, the behavior of the result-

ing object is unpredictable. For debugging, a user creates a single-site instance of an object and runs it off-line.

2.2. Properties of resilient objects

Throughout this paper, the term *resiliency* is used to denote *k-resiliency*, meaning that the following properties are satisfied:

1. **Consistency**. The object behaves like a non-distributed one which executes requests sequentially and to completion, with no interleaving of executions.
2. **Availability**. Let f denote the number of components of an object that fail simultaneously. If $f \leq k$ then operational [illegible]ents continue to accept and process requests.
3. **Progress**. If $f \leq k$ then operations are executed without blocking, despite failures.
4. **Recovery**. Because *ISIS* supports replication, two cases should be dis[illegible]ed:
   a. **Partial**. If $f \leq k$ a failed component restarts automatically when its site recovers.
   b. **Total** If $f > k$ failed [illegible]ents restart automatically when all the failed sites recover.

2.3. Logical execution model

Although concurrent access to objects will be common, it is hard to reason about the correctness of interleaved executions. This consideration underlies the consistency requirement given above -- intuitively, that a resilient object should always look idle to its users, and that the effect of a correct concurrent execution should be the same as for some correct but non-concurrent one.

Our approach is to treat the execution of each procedure as a *transaction*: a sequence of actions that occurs to completion or not at all. Objects can issue calls to one another, hence these transactions are *nested* in the sense of [Moss]. One way to understand the model is to visualize each procedure as executing within a *scope* defined by its caller. Activities outside this scope are permitted to see only the object state before or after the procedure completes, never while it is taking place. As a convenience to the programmer, any procedure can terminate by executing an

abort statement, which causes all its effects to be erased.

Formally, a transaction is a sequence of one or more *operations*: local computations or remote procedure calls, which execute as *subtransactions*. A transaction terminates by executing a *commit* or *abort*. Commit is implicit when a procedure returns normally, and makes the effects of a *top-level* transaction visible to other transactions, and those of a subtransaction visible to its siblings. Abort must be explicit, and restores the state that existed prior to execution of the transaction or subtransaction. Committed transactions must be serializable: a total order on them should exist such that the state of the system is equivalent to the one that would result if these transactions were executed sequentially in that order [Papadimitrios] [Gray]. Thus, although interleaved executions are permitted, they must produce the same effect as some non-interleaved execution would have produced.

A purely transactional model would result in inefficiencies. For example, if a side-effect of inserting an item into a data structure is to initiate garbage collection, and an invocation is subsequently aborted, the garbage is reinstalled. In *ISIS*, the programmer can avoid this problem by explicitly specifying that an operation should execute as a top-level transaction, as if it had been invoked by an external caller (this feature was proposed in [Liskov-b]).

3. Object specification language and system interface

The *resilient types* are abstract types [Liskov-a] satisfying the properties listed above. 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. 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*.

[3]Sequentially allocated data structures tend to have "hot spots" which are frequently accessed, reducing potential concurrency [Garwick]. The heap management facility is used to dynamically allocate and deallocate resilient records from within transactions, avoiding a common source of hot-spots.

2. Procedures for manipulating the resilient data. These can be given attributes such as *create* (a new instance of the object is allocated for each invocation), *entry* (accessible to external callers), and *read-only* (does no updates).

3. Type definitions for the arguments and results of operations.

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:

1. *I/O statements*. Resilient data is accessed using *read* and *write* statements.

2. *Locking statements*. These are used for concurrency control, as discussed in Sec. 4.1.

3. *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. RPC is also used to invoke most *ISIS* system functions. Each call executes as a transaction.

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

5. *Cobegin*. A set of *branches* (statements) are specified for concurrent execution. The cobegin terminates when all of its branches terminate. *Return* is not permitted in a *cobegin* branch. *Cobegin* branches execute as tasks within a single address space. Currently, these all run at a single site, although a more flexible *cobegin*, which might provide some form of explicit control over replicated processing, is under consideration for the future. The statement is normally used to keep a computation active while some branch is blocked (e.g. when acquiring a lock).

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

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 (the "name space" has a well-known capability). Each call executes as a top-level transaction. A client can also issue a series of requests as a transaction, first calling *BEGIN*, and later *COMMIT* or *ABORT* when the outcome is determined.

The *abort* mechanism described above is internal -- it is explicitly executed by a procedure that has detected some exception condition. Also needed is a way to terminate a transaction from the outside, e.g. if a deadlock [illegible] or an operator interrupts a computation for some other reason. *ISIS* supports a software **kill** signal, which can be issued by a client process and cannot be caught or ignored. Associated with each transaction is a unique transaction-id (TID). The **kill** primitive takes a TID as its argument and terminates the corresponding transaction by halting it and its subtransactions, and then forcing them to abort. Additionally, if a non-resilient client fails while executing a transaction, a *kill* is automatically performed on its behalf.

Any system that can be forced to abort must avoid irreversible actions, like [illegible]ate movement of a mechanical arm. In *ISIS*, this is not a problem because the system only invokes *kill* if a deadlock is detected -- thus, a deadlock-free application will never be killed (for example, one that acquires locks in some predetermined order). Our approach is more flexible than that of systems which abort transactions that executed at a site which subsequently failed. Thus, whereas an *ISIS* application could be designed to move a mechanical arm while monitoring and reacting to sensor feedback, in many other systems, this would not be possible: since the arm motion cannot be exactly reversed, to avoid aborts they must defer it until the top-level commit.

The *ISIS* system is itself controlled by a [illegible] language. At start-up, each site [illegible] an initialization file containing configuration information. Subsequently, reconfiguration and other commands can be issued interactively.

4. Runtime issues

4.1. Fault-tolerant execution of a request

A *task* refers to the physical execution of a request by one of the [illegible]ents of a resilient object, designated the *coordinator*. Components are identical: any can be coordinator for any

request, which tends to distribute processing load. The [illegible]ents that are passive for a request are designated as *cohorts* and serve as [illegible] -- one takes over if the coordinator fails. A task must satisfy the properties enumerated in Sec. 2.2 to produce a correct logical execution. We now consider these properties individually.

4.1.1. Consistency

Consistency is a question of event orderings: an acyclic ordering must be established between conflicting events and enforced during execution. To ensure that this is the case, each object implements a *concurrency control* algorithm [Bernstein]. It is difficult to automatically infer a good concurrency control method from an object specification. Therefore, *ISIS* requires that the programmer provide a single-site concurrency control algorithm, which it transforms into a distributed one. *ISIS* provides flexible support for constructing lock-based algorithms, of which 2-phase locking is the best known and easiest to code. Two lock classes are supported; within each class, read, promotable read, "previous committed version" read, and write locks can be requested. The classes 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.

Once an ordering between events is established, it is important to respect it without employing excessive synchronization. As will be seen shortly, the *ISIS* [illegible]unication primitives can provide high levels of concurrency while respecting exactly the message delivery orderings needed to execute a distributed computation correctly.

4.1.2. 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-any 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.

4.1.3. Progress

ISIS ensures that operations progress to completion using a checkpoint-restart scheme [Birman-a]. Each RPC is broadcast to the operational [illegible]ents of an object, and constitutes a *checkpoint*. If the coordinator fails while executing the request, one of its cohorts takes over as the 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 [illegible]unication with other [illegible]ents or objects). The actions of the failed coordinator are then repeated in *restart mode*. When the coordinator issues a call in restart mode that does not repeat some prior call issued by the failed coordinator, the called object detects this (see below) and rejects the request. Normal execution then resumes and the call is reissued.

When the new coordinator reissues an operation during restart, it should not be re-executed -- otherwise, the system state could become inconsistent (i.e. if an increment is performed twice). To avoid this, the *results* returned by operations are replicated in addition to the updates they perform on replicated data. 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. Since the same TID is used during restart, they can find and return a copy of the retained result, or, if none is found, reject the request, which will then be reissued in [illegible] mode. Re[illegible] results are discarded when the parent of a subtransaction commits or aborts, retaining its own result, or when the top-level commits (results of actions in a top-level statement must be retained). Note that if a resilient object takes external actions, like moving a robot arm, the device must provide a function equivalent to re[illegible] results -- for example, a way to

determine the command it last executed.

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 local 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. The approach is to piggyback read locking information on other messages that could depend on their existence -- RPC requests and updates issued subsequent to the acquisition of the lock. Then, if any "evidence" to the existence of the lock survives a crash, in the form of an RPC or update that was not wiped out, the locking information needed to ensure consistent restart actions survives as well. After the crash, this information is forwarded to the new coordinator, which registers it before granting any pending write-lock requests. If *all* the evidence pertaining to a read-lock is lost, actions inconsistent with that lock might occur during restart -- but this is acceptable, since data stored at the failed site are discarded during recovery, and the actions taken by the operational part of the system will thus be self-consistent (but see also the discussion of **flush** in Sec. 4.3.4).

4.1.4. Recovery

If a partial failure [illegible], a failed [illegible]ent 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 [illegible], eliminating the need for stable (disk) storage. We will show that the [illegible]unication primitives ensure that recovery is serialized with respect to other operations.

One way to implement total recovery is to save checkpoints and committed versions of the object data on stable storage. When the [illegible]ents that failed last have recovered [Skeen], they resume operation from their stable stores; other components use the partial recovery method. However, this approach is costly. A preferable alternative is to restart from total failure in the initial state and then to reload the objects from application-level [illegible] and logs. *ISIS* provides a

log facility that can save RPC requests for subsequent replay. If a transaction aborts, its log entries are deleted. This permits each application to implement a low-cost recovery method of its own.

4.2. The communication subsystem

We now turn to implementation issues, and particularly the way in which the above techniques can be expressed using a small set of communication primitives [Birman-b]. For brevity, protocols and correctness proofs are not included here.

ISIS employs a communication subsystem providing 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 destination receives a message, then unless it fails, all destinations will receive[4] it.

The BCAST primitive

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, some parts of *ISIS* maintain replicated lists by ensuring that insertions and deletions occur in the same order at each copy. This behavior is obtained using the primitive *BCAST*(*msg*, *label*, *dests*), where *msg* is the message and *label* is a string of characters. Two *BCAST*'s having the same label are ordered in the same way at all common destinations. A *BCAST* having the label '*' is ordered with respect to all other *BCAST*'s.

The OBCAST primitive

If a computation transmits multiple messages to a single destination, they should be processed in the order they were sent. The *ordered broadcast* primitive, *OBCAST(msg, ts, dests)*, respects such "prespecified" delivery orderings. Here, *ts* is a timestamp that can be compared with

[4]Reception can be indirect. If process *t* receives a message *m* from process *s*, and *s* subsequently fails, than the state of *t* may depend on messages received by *s* before it sent *m*. Therefore, unless *t* fails as well, those messages must be delivered to their remaining destinations.

other timestamps. *OBCAST* ensures that broadcasts are received at overlapping destinations in the order prescribed by their timestamps (if the destinations do not overlap, delivery order is unconstrained). *OBCAST* is the source of most concurrency in *ISIS*: although the delivery ordering is assured, transmission can be delayed to take advantage of piggybacking. Since many *ISIS* messages are small and local computing continues in parallel with the *OBCAST*, the resulting speedup can be dramatic. A scheduling algorithm is being designed to optimize transmission decisions.

Timestamps are generated by incrementing a local counter for each broadcast. The counter is also changed to be larger than any timestamp received in a message. Thus, if one broadcast could have affected a second broadcast, the timestamp of the first will be smaller than the timestamp of the second [Lamport]. Note that timestamps generated by unrelated pr[redacted] are comparable, hence it might seem that *OBCAST* is being asked to respect specious orderings. In [Birman-b], we give a *OBCAST* implementation that respects only those orderings that could arise in a resilient computation.

The GBCAST primitive

The basic characteristic of a coordinator-cohort computation is that the cohorts monitor the coordinator for failure. To capture this, we will say that the operational components of an object form a *process group*. The third broadcast primitive, denoted *GBCAST*, transmits information about failures and recoveries to process group members. A recovering [redacted]ent uses *GBCAST* to join the operational ones; when a [redacted]ent fails, some other [redacted]ent issues a failure *GBCAST* on its behalf.

Our *GBCAST* protocol ensures that *GBCAST* events are totally ordered with respect to *OBCAST* and *BCAST* events, and that when a failure occurs, the corresponding *GBCAST* 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. The ordering property then implies that although [redacted]ents do not receive broadcast simultaneously (in wall-clock time), they do receive each message in the same view. This lead us to formu-

late algorithms implementing the techniques described earlier in terms of *logical* orderings that the protocols can implement, rather than explicit synchronization. A result is that it remains easy to establish the correctness of the algorithms despite concurrency in the communication layer.

4.3. Fault-tolerant implementation of selected operations

4.3.1. RPC calls and reception; commit and abort

To issue an RPC, a task first generates a transaction id under which the execution will occur. If a non-resilient process is perf[illegible] 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. 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 *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 reply.

Upon receiving the RPC, a [illegible]ent must determine if it is the new coordinator. All components of each object are statically ordered by site number into a ring. A [illegible]ent 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 [illegible]ent. This tends to place the coordinator at the same site as the originator, which is desirable because it maximizes the potential degree of asynchrony between the [illegible]ents at a site and their cohorts. The new coordinator [illegible] a retained result if one is found. Otherwise, it executes the new request and *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 then recomputes the ranking. Since all [illegible]ents have the same view when an RPC is received, and all subsequently see the same sequence of failures and recoveries, the conclusions reached by components are consistent. Note that all necessary syn-

[5]An inexpensive protocol for maintaining a *cache* of group addressing information, and to update it if it is found to be out of date during message transmission, is given in [Birman-b].

chronization 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 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 (a subtransaction can reply to its caller before issuing its own commit or abort, hence the parent's commit could arrive before the child's computation is finished). 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 by a coordinator for a task, it waits until restart is completed (to ensure that the CLIST is accurate), forwards the signal to any active subtransactions, and then initiates an abort.

4.3.2. 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 [redacted]ent transfers its state to the recovering one: since the state 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 state. Note that all the necessary synchronization is provided by the *GBCAST* primitive.

4.3.3. Managing replicated locks

The locking methods discussed earlier are easily realized using our broadcast primitives. A read-lock is first obtained locally by the coordinator of a computation. Then, a *read lock registration* message is *OBCAST* to the other copies of the data item. The coordinator immediately con-

tinues 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 has received a message *m* sent after the lock acquisition, the read-lock will be registered before the failure is "detected" by the cohorts [redacted] 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 occur after the lock registration. Because there is no urgency to send a read-lock registration message, piggybacking is particularly cost-effective (hence the description of Sec. 4.1.3).

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. On the other hand, if two write-lock requests on the same date item are issued concurrently, they could deadlock 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 *BCAST* using the identifier of the data item as a label. If a component fails during the protocol, the partially acquired write-lock is withdrawn and then re-requested. Since the read-lock registration message preceded the *GBCAST* announcing the failure, either it is delivered before the *GBCAST*, or no site has received a message sent by the process after it obtained the lock. The write-lock is re-requested *after* the *GBCAST* message is received, so it will be forced to wait on the read-lock if evidence that the transaction held a read lock survives the crash (in the form of some action it took that reflects the data it read). Moreover, since *BCAST* is delivered in the same order everywhere, concurrent write-lock requests will not deadlock.

[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 can do local locking on **x** using the same registration technique as was described for read-locks.

4.3.4. 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 transmit the new value to all operational [redacted]ents. The coordinator then waits for acknowledgement that the operations have been completed (or a failure occurs, as signalled by arrival of a *GBCAST*).

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 uses *OBCAST* to inform its cohorts without waiting for acknowledgments [Joseph-b]. Recall that *OBCAST* is also used to commit, which releases locks. Since the updates precede the commit and *OBCAST* respects this ordering, any process that obtains a lock will observe the correct version of any 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!

To preserve the consistency constraint on executions, two additional issues must be addressed. First, when a top-level transaction that has done updates is ready to reply to its caller, the reply must be delayed until all [redacted]ents have received the initial RPC -- otherwise, a failure could cause the RPC and all the updates it generated to be lost, invalidating the result returned to the caller. *ISIS* is able to satisfy this constraint without introducing any overhead. Similarly, if an object using concurrent update performs actions with external side-effects, a failure could erase any [redacted] concurrent updates together with any associated read-lock information, although the external actions that depended on this information would persist. Then, during restart, although the actions of the object would be internally consistent, they might be inconsistent with the externally observed behavior of the system before it failed. For such cases, a **flush** primitive is provided, which the object calls before taking any external actions. **Flush** blocks until all [redacted]

updates have been delivered.

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 (by acquiring write-locks in a predetermined order), 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 and 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 a single locking 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.

5. System architecture

5.1. Communication primitives

A detailed discussion of the communication subsystem is given in [Birman-b]; we now summarize its architecture briefly. The primitives are build 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 [redacted] 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. *BCAST* employs a two-phase protocol based on one by Skeen [Skeen-b]. *OBCAST* has a very light-weight implementation, based on the idea of flooding the system with copies of a message: Each process buffers copies of the 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 as well (duplicates are discarded). A garbage collection phase 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. *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. More details and correctness proofs appear in [Birman-b].

5.2. Higher level system structure

ISIS is best viewed as an adjunct to a conventional operating system, providing persistent fault-tolerant services which complement conventional fault-intolerant progr[illegible] support. Our prototype was built under 4.2 UNIX. Performance is limited [illegible]y by UNIX overhead, and is reported in the next section.

The system is organized hierarchically, as illustrated in Fig. 1. The lowest level of *ISIS* provides the communication primitives described earlier, together with a 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 [Lampson], the transactional version stack, and the lock manager. These are used by the implementations of the algorithms of Sec. 4. A number of system services are implemented as resilient objects: the capability manager [Dietrich], 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 pr[illegible] to issue requests to resilient objects.

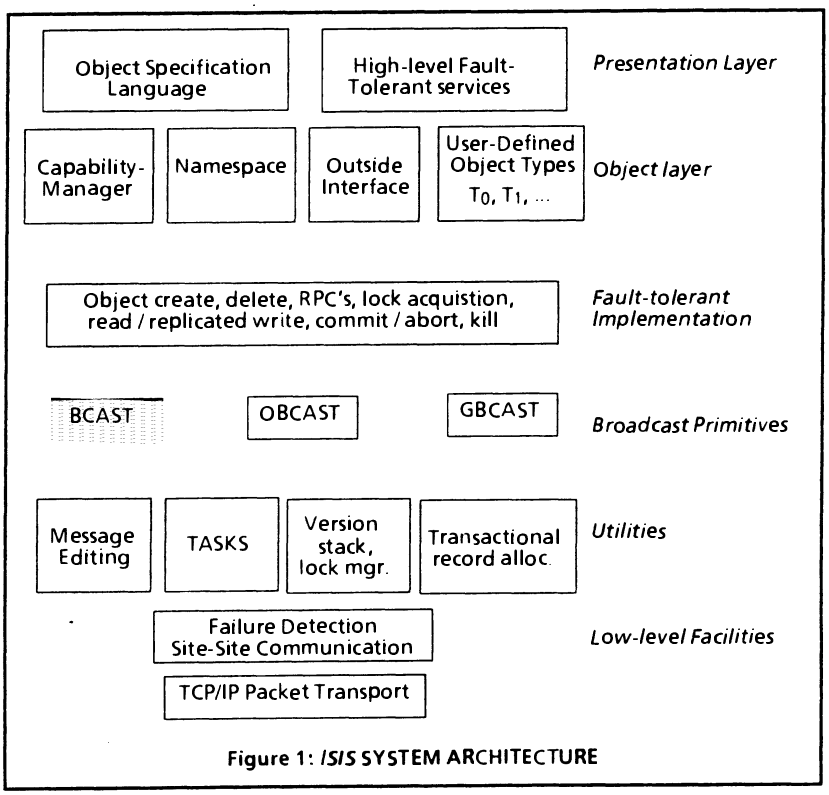

Figure 1: *ISIS* SYSTEM ARCHITECTURE

In UNIX, processes and inter-process communication are expensive. 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.

6. 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, and then to measure the response time for some simple resilient objects. The indexed sequential file, built from a resilient

directory and a resilient file, illustrates the overhead associated with nesting.

GENERAL PERFORMANCE	COMPONENT TESTED	SUN 2/50		
Site-to-site message	Delay to reception	40ms		
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		
Failure detector	Timeout	n.a.	7 secs	7 secs
GBCAST	Delay until delivered	n.a.	2 secs	3 secs
COMMUNICATION PRIMITIVES		**1 site**	**3 sites**	**6 sites**
OBCAST	Latency	10ms	49ms	60ms
	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
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

Although our figures are for lightly loaded *ISIS* systems, the multi-process structure of the system still incurred substantial scheduling and interprocess [illegible]unication overhead. Also, the performance of IPC connections is suboptimal, [illegible]y because the SUN version of UNIX does not yet support changes to the IPC buffer size.

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 argues in favor of log-based recovery from total failures (the default), since stable storage is thereby avoided.

The broadcast primitives are d[illegible]ted by underlying message-passing costs, but otherwise depend [illegible]y 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 *GBCAST*), hence all the primitives give similar performance. The *latency* figure measures the time from message transmission to remote delivery. *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 thro[illegible]t 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. The throughput figures indicate that substantial system loads could be accommodated.

Turning to the resilient objects themselves, note the dramatic performance improvement performance achieved when using concurrent 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 rates 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. Note that nesting does 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.

The main observation to draw from the above is that when using concurrent update, the performance of a resilient object accessible from multiple sites is not far from that of a fault-intolerant single-site object of the same type (the *ISIS* figures in the latter case are in line with those reported for other systems supporting non-replicated objects). Moreover, overall performance will be much higher in read-intensive settings, provided that requests arrive randomly at the different components, since reads can be done locally by any [redacted]ent. Thus, *ISIS* is able to provide powerful distributed services at suprisingly low cost -- at worst, a k-resilient object performs nearly as well as a single-site instantiation of the underlying abstract type, and at best its overall performance could exceed that of a single-site object by a factor of $k+1$. Although that this is not suprising, in view of the way that concurrent update works, it is certainly encouraging. 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.

7. Future research

The *ISIS* project is now entering its third year. Two major problems are receiving attention: interconnection of *ISIS* clusters and the resulting partitioning issues, and an investigation of the limits of concurrency in systems subject to ordering-based correctness constraints [Joseph]. On a

more pragmatic level, we are examining uses for *ISIS* in high-level progr[redacted] 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-c]. Finally, we confront the problem of integrating *ISIS* more fully into an operating system. *ISIS* provides a type of service and exhibits a collection of support requirements which are very different from those seen in other distributed systems. Design of an operating system that is good at providing both conventional and fault-tolerant services thus remains an open problem.

8. 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 [redacted]unication architecture leads to a simple system structure within which correctness arguments are straightforward despite the presense of failures and concurrency.

We believe that, along with other systems for fault-tolerant software development, *ISIS* represents the beginning of a new generation of very high-level operating systems facilities. 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 progr[redacted] methodologies are inadequate. As the complexity and sheer size of distributed systems continues to grow, facilities of this sort will be indispensable.

9. Acknowledgement

Wally Deitrich, Amr El Abbadi, Tommy Joseph, Thomas [redacted]e, and Pat Step[redacted] 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 a prior version of this paper.

10. References

[Bartlett] Bartlett, J. A NonStop Kernel. *Proc 8th SOSP*, Dec. 1981.

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

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

[Birman-b] Birman, K., Joseph, T. Reliable communication in an unreliable environment. Dept. of Computer Science, Cornell Univ., TR 85-694, Aug. 1985.

[Birman-c] Birman, K. *et al.* MDB-1: A new 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 SOSP*, Bretton Woods, NH (Oct. 1983), 90-99.

[Borr] Borr, A. Robustness to crash in a distributed database: A non-shared memory multi-processor approach. Tandem Computers, Inc., 1984.

[Cooper] Cooper, E. Replicated distributed programs. Ph.D. dissertation, Computer Science Department, Univ. of California, Berkeley (May 1985).

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

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

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

[Joseph-a] Joseph, T. Ph.D. dissertation, forthcoming.

[Joseph-b] Joseph, T., Birman, K. Low cost management of replicated data in fault-tolerant distributed systems. To appear, *ACM TOCS*.

[Lampson] Lampson, B., Redell, D. Experience with processes and monitors in MESA. *CACM* **23**, 2 (Feb. 1980), 105-117.

[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 SOSP*, 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.

[Papadimitrou] 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 SOSP*, 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.

[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.*