An Overview of the Isis Project *

Kenneth P. Birman
Amr El Abbadi
Wally Dietrich
Thomas Joseph
Thomas Raeuchle

TR 84-642
October 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

* This work is supported in part by a grant from the Sperry Corporation.

AN OVERVIEW OF THE ISIS PROJECT*

Kenneth P. Birman, Amr El Abbadi, Wally Dietrich
Thomas Joseph, Thomas Raeuchle

Department of Computer Science
Cornell University, Ithaca, N.Y. 14850

ABSTRACT

The goal of the *ISIS* project is to provide high-level support for fault-tolerant distributed computing by automatically replicating data and code. The extent to which information is replicated and the physical location of information are not specified directly by the programmer, but are instead inferred from a specification, which looks much like a conventional program in an object-oriented language. This novel approach to fault-tolerant software construction requires much less sophistication from programmers than current alternatives. Moreover, optimization techniques that would be too complex for implementation in general purpose applications can be supported by the *ISIS* system. This overview discusses the goals of the project, its current status, and some of the implications of our work.

1. Introduction

Computer applications are increasingly often implemented as distributed systems - systems of multiple processors exchanging information across a communications network. One feature of a distributed system is that processing capability is replicated, and this has added a new dimension to fault tolerance. Whereas the best that a non-replicated system can achieve in the presence of failures is graceful recovery after a fault has been identified and rectified, a replicated system has the potential to offer continuous operation despite failures of some of its components. This type of fault tolerance is almost indispensable in critical applications like medical life-support equipment, and is highly desirable in other areas. Replication of data requires complex protocols to ensure that the various copies of data are kept consistent, and sharing of processing among different units necessitates careful synchronization between them. The picture is further complicated if processors or communications links can fail arbitrarily and independently. The result is that achieving fault tolerance in distributed systems normally requires a great deal of skilled programming effort.

*This work is supported in part by a grant from the Sperry Corporation.

The *ISIS*[1] project, currently being undertaken at Cornell, seeks to place the implementation of fault-tolerant distributed systems within the reach of a relatively unsophisticated programmer. *ISIS* will provide support for the development of k-resilient objects: abstract data types having the additional property that their operations are guaranteed to execute to completion despite the failure of up to k processing sites. To satisfy these requirements, a k-resilient object must be implemented at $k+1$ or more sites, because after k sites fail, there must be at least one operational site at which a copy of the object-data is available, and that is able to continue operation. In *ISIS*, a user specifies and accesses a resilient object as if it were not distributed. The system transforms the specification into a collection of *components:* single-site objects residing at independent sites. The components of an object cooperate to manage replicated data and synchronize their actions, presenting the behavior of a single entity that remains operational despite failures.

To use the system, the programmer will construct front-end programs with the tools available in a conventional host operating system. A procedure-call interface permits these programs to access one or more resilient objects for fault-tolerant back-end control and data storage. The *ISIS* programmer will be able to specify new types of resilient objects using a high level language, in much the same way as he or she would specify a non-resilient one. *ISIS* transforms such a specification into a resilient implementation, inserting calls to a runtime system that takes care of the necessary replication and provides the mechanisms required to maintain consistency of data, to synchronize execution of operations, and to detect and recover from failures.

Certain examples have become standard in the literature of fault-tolerance and can be recast in terms of resilient objects for purposes of comparison. Consider the mail registry shown in Figure 1. Here, the front-end software would consist of a conventional mail program. The resilient object specified in the figure might be used for mail storage, ensuring that mail remains accessible despite up to 3 site-failures. The mail system may be rendered inaccessible by additional site-failures, but mail would never be lost or damaged. The specification is complete as shown; note

[1]ISIS is named after a goddess of Egyptian antiquity, who restored Osiris, god of the underworld, to life after he had been torn into pieces by Set and scattered across the Nile delta.

```
3-resilient mailer created by create exports add_user, send_msg, rcv_msg;
begin;
  resilient
  {
      char:      user_name[10];          /* Name of this user */
      string:    msg_list[20];           /* List of up to 20 messages */
      integer:   msg_first;              /* First on list */
      integer:   msg_free;               /* First free */
  } mail_spool[100];                     /* Up to 100 users */

  resilient integer nusers;              /* Number of users */

  create(): capability;
   {
      nusers := 0;                       /* Initially empty */
      return(SELF);                      /* Return capability ("pointer") to new object */
   }

  inc(n)
   {
      return( mod(n+ 1, 20) );
   }

  add_user(who)
   string: who;
   {
      if(nusers = 100)
         return(ERROR);                  /* Failed: no more slots */
      mail_spool[nusers].user_name := who;
      mail_spool[nusers].msg_first := 0;
      mail_spool[nusers].msg_free := 0;
      return(SUCCESS);
   }

  send_msg(who, msg)
   string: who, msg;
   {
      integer: n, slot;
      for(n = 0; n < nusers; n := n+ 1)
         if(mail_spool[n].user_name = who)
         {
             /* Found user, add mail to list unless full */
             slot := inc(mail_spool[n].msg_free);
             if(slot = mail_spool[n].msg_first)
                return(ERROR);           /* Failed: no room for message */
             mail_spool[n].msg_list[mail_spool[n].msg_free] := msg;
             mail_spool[n].msg_free := slot;
             return(SUCCESS);
         }
      return(ERROR);                     /* User not found */
   }
```

. . . (cont)

. . . (cont)

```
rcv_msg(who): string;
  string: who;
  {
      integer: n;
      string: msg;

      for(n = 0; n < nusers; n := n + 1)
         if(mail_spool[n].user_name = who)
         {
              /* Found user, return next message if any */
              if(mail_spool[n].msg_free = mail_spool[n].msg_first)
                 return('');                     /* No mail */
              msg := mail_spool[n].msg_list[mail_spool[n].msg_first];
              mail_spool[n].msg_first := inc(mail_spool[n].msg_first);
              return(msg);
         }
      return('user unknown');
  }
end
```

Figure 1: Resilient Mail Spooler

---

that code for distributed access, replication, and failure-handling does not have to be included.

It is also possible to specify more sophisticated objects in *ISIS*. An example would be an airline ticketing system. In such an object, it would be desirable for each site to do significant local processing, selling tickets and satisfying customer requests without interacting with other sites. This abstraction is different from that used in the mail object, where all sites share the same view of any distributed data and updates to the data are immediately visible at all sites. The *ISIS* specification language includes the constructs needed to describe such "limited autonomy" in the behavior of components.

2. System Structure

The logical structure of *ISIS* is layered, although the implementation combines layers for reasons of efficiency. In this section, the contents of each layer are described. Section 4 considers implementation issues and discusses Figure 2, which illustrates the architecture that has been adopted.

In a fault-tolerant system, the sorts of failures that are anticipated has a significant impact on the system structure and attainable performance. For *ISIS*, a failure model has been adopted that is typical of local networks containing loosely coupled processors or workstations. Specifically, it is assumed that machines fail by halting or crashing, without engaging in malicious behavior, such as emitting incorrect messages or generating erroneous requests[2]. However, the communication system is assumed to lose messages and introduce delivery delays. Also, *network partitioning* can sometimes occur.

At the lowest level of the *ISIS* system is the *failure detector*. It detects processor and communications link failures and presents an abstraction called *fail-stop processors* to the rest of the system. The only type of failure experienced by a fail-stop processor is a halting failure: the processor stops executing and other processors become aware of this fact. Supporting a fail-stop abstraction is non-trivial because of the many conditions under which a site might appear to have failed when it is still functioning, or only experiencing a temporary pause. The failure detector operates by polling processors periodically and presenting a *view* of the system to each processor. A view describes which of the processing sites are operational and which are not, and any change in the view represents the failure or recovery of one or more sites. The protocol executed by the failure detector ensures that all sites are presented with mutually consistent views. Additionally, the order in which sites are seen to fail is the same everywhere. A first version of the failure detector has been implemented, but sometimes *blocks* (pauses) if network partitioning occurs. An extended failure detector that will operate despite partitioning is being designed.

Fail-stop processors communicate across a fault-free network, which is provided by filtering all messages through a layer of software running on top of the failure detector. This layer also provides a set of communication primitives for use by higher levels. Specifically, the system supports a set of basic operations that underly most protocols for synchronization and concurrency control. These include *commit* protocols [Skeen], *broadcast* protocols that ensure that all operational members of a set of sites process a message or that none does so, and an *atomic broadcast*

[2]Without such an assumption, costly *Byzantine* protocols are required at every level of the system [Lamport].

protocol, which additionally ensures that if messages m and m' are both broadcast, then all sites process them in the same order [Gray] [Chang].

Higher levels of *ISIS* implement k-resiliency by replicating data and executing operations while preserving the correctness of each object. In essence, this requires that each object behave as if data had not been replicated [Bern-a]. In addition, operations are required to execute *atomically:* to completion or not at all. Finally, resilient objects are accessed concurrently; this calls for a concurrency control algorithm. Many such algorithms are known [Bern-b]; the next section describes a new one that has been developed for ISIS.

For an operation to be able to execute to completion despite processor failures, information about its execution must be distributed among different processors as the execution proceeds so that another site can take over and continue if the site that is executing it fails. This approach is analogous to the use of checkpoints in a non-distributed system. In *ISIS*, a distributed checkpointing mechanism is employed automatically. The method, described in [ISIS-a], determines during execution what information must be distributed to achieve k-resiliency, and when. The number of message exchanges is kept low by transmitting information only when necessary. An important attribute of the approach is that it never requires operations to be aborted and re-executed after a failure. This feature will permit the use of *ISIS* in applications like real-time process control, where programs take irreversible actions from which it is impossible to "back out". In contrast, most other system use concurrency control methods that either do not support replicated data, or require that transactions be rolled back if a a site where they have executed fails [Bern-b].

An interesting consequence of the use of distributed checkpoints is that recovery is less of an issue in *ISIS* than in other systems. An *ISIS* application remains operational despite some number of failures, and recovery after severe failures involving most sites in the system may not be necessary. For example, in control applications, the system would typically be restarted after such an event. *ISIS* distinguishes between the restart of a component of an object that remained operational despite the failure, and the restart of an entire object in which all components have failed. Support for the first form of recovery is simple, and can be done by copying the state of some operational component[3]. In contrast, support for the second form of recovery involves storing checkpoints on *stable storage* and maintaining logs for transactions that are in progress [Gray] [Kohler], and is therefore costly. Since *ISIS* provides the first sort of recovery, the second can be avoided in many objects, improving performance. In other systems, replication is an application-specific issue, hence only the second form of recovery is supported.

The highest level of *ISIS* is concerned with the interface to objects as presented to other objects and to external users of the system, e.g. programs executing under UNIX. Issues include managing a name space for distributed objects and designing a communications system that directs messages to an operational component of an object and enforces access restrictions. Similar problems arise in any distributed system, but they are especially interesting in *ISIS* because a resilient object is directly accessible at multiple sites, placing fault-tolerance requirements on the communication system. At the same time, *ISIS* has an interesting new degree of freedom: it is possible to direct requests to the least loaded processing site, or to migrate data items within objects to adapt data distribution in response to requests. For example, *ISIS* will support a type of object in which data is not replicated everywhere, using a controlling object to channel requests to one of a collection of data storage objects residing at a smaller number of sites, within which data items are completely replicated. Work on more sophisticated objects of this sort is planned for the future.

3. Managing Replicated Data in ISIS

Replication introduces two potential bottlenecks in distributed systems. First, a concurrency control method is needed for arbitrating access to data items. More serious than the basic cost of this algorithm is the latency it can introduce into computations. Specifically, since data is replicated, an update must be delayed until all operational sites at which copies of the data items reside have reached agreement on the order of access to these items, on the assumption that conflicts with reads and other updates may arise. Similarly, a read must be delayed if an

[3]This assumes that the object data is fairly small; other methods could be devised for very large objects.

update might still be "in progress". Below, we describe a new concurrency control method that permits much greater concurrency lower latency than other methods by taking advantage of semantic information obtained by analysis of object specifications. A second potential bottleneck is related to updates of replicated data items. Updates of this sort are potentially slow because, in most systems, it is necessary to wait for confirmation that the physical updates have completed before the logical update is viewed as "finished". A major finding of the project is that such acknowledgements are generally not needed [ISIS-c]. The combination of a good concurrency-control scheme with our replicated-update method leads to a suprising and important result: resilient objects may be able to execute as fast as if they resided at a single site.

The *ISIS* concurrency control algorithm is special in several respects. First, it is desirable to attain as high a level of concurrency as possible to compensate for the delays inherent in updating replicated data. Second, the algorithm should be tolerant to faults. Third, to keep the level of communication low, it should be possible to integrate the concurrency control algorithm with the mechanisms used in *ISIS* to achieve fault tolerance. With these aims in mind, we have developed a novel distributed concurrency control algorithm, described in [ISIS-b], that uses data-flow techniques to predict the data accesses that an operation will make, performing a static analysis of the object specification for this purpose. This information is then exploited to make concurrency control decisions as early as possible and to allow concurrent execution of operations under conditions in which a naive method might force all but one to block. The algorithm takes advantage of commutativity of operations to increase concurrency, and incorporates elements of "optimistic" concurrency control algorithms [Kung], but without permitting cascaded aborts to occur. A special problem that arises in *ISIS* concerns concurrency control decisions for read locks. This is because the checkpointing method permits operations to run forward after a failure, and hence requires that the serialization order be preserved across failures. The *ISIS* concurrency control algorithm accomplishes this by *piggybacking* read-lock information on messages pertaining to updates that may have depended on a read. The effect is that, after a failure, the read locks are known if there is any site at which an update depending on the read has left visible effects. Since read-lock information is transferred by piggybacking on other messages, the cost of the method is minimal.

We now turn to the question of updating replicated data items and checkpoints, which are also a form of replicated data. Our major finding, discussed in [ISIS-c], concerns a method for maintaining the correctness of replicated data items at low *perceived cost.* In that paper, we argue that perceived cost really stems from the forms of latency noted early in this section, and hence that reductions in latency will result in good system performance. The essense of our method is to update replicated data items using broadcasts that execute concurrently with other processing. The task initiating such a broadcast does not wait until it completes. Information about operations that are in progress is piggybacked on concurrency control messages, and is used to ensure that any access to a replicated data item serialized after one of these "concurrent updates" will block if the update has not yet been completed. On the other hand, such an operation will not block if the update *has* been completed, which will generally be the case. Thus, although operations may still be delayed, delays will now tend to be infrequent and of short duration. In effect, the latency normally associated with maintaining replicated data is overlapped with other computing. An implication is that, in *ISIS*, *fault-tolerant objects may attain performance comparable to that of a non-distributed, fault-intolerant object!* This line of research is now leading us to explore the relationship between concurrency in a loosely coupled system such as *ISIS* and parallelism in super-computers, where similar problems arise.

4. Implementation Status

Implementation of an *ISIS* prototype has been underway for more than a year. Our approach has been to construct the system within a network of VAX computers and workstations running 4.2BSD UNIX™. Users view resilient objects much as they view UNIX files, although the operations supported by an object depend on the object type, and a special interface must be used to access them. Object specification are in a high-level language based on *C*, but extended with features borrowed from the *ARGUS* language [ARGUS]. Currently, the *ISIS* failure detector and communication system are operational, as are a concurrency control mechanism and a transac-

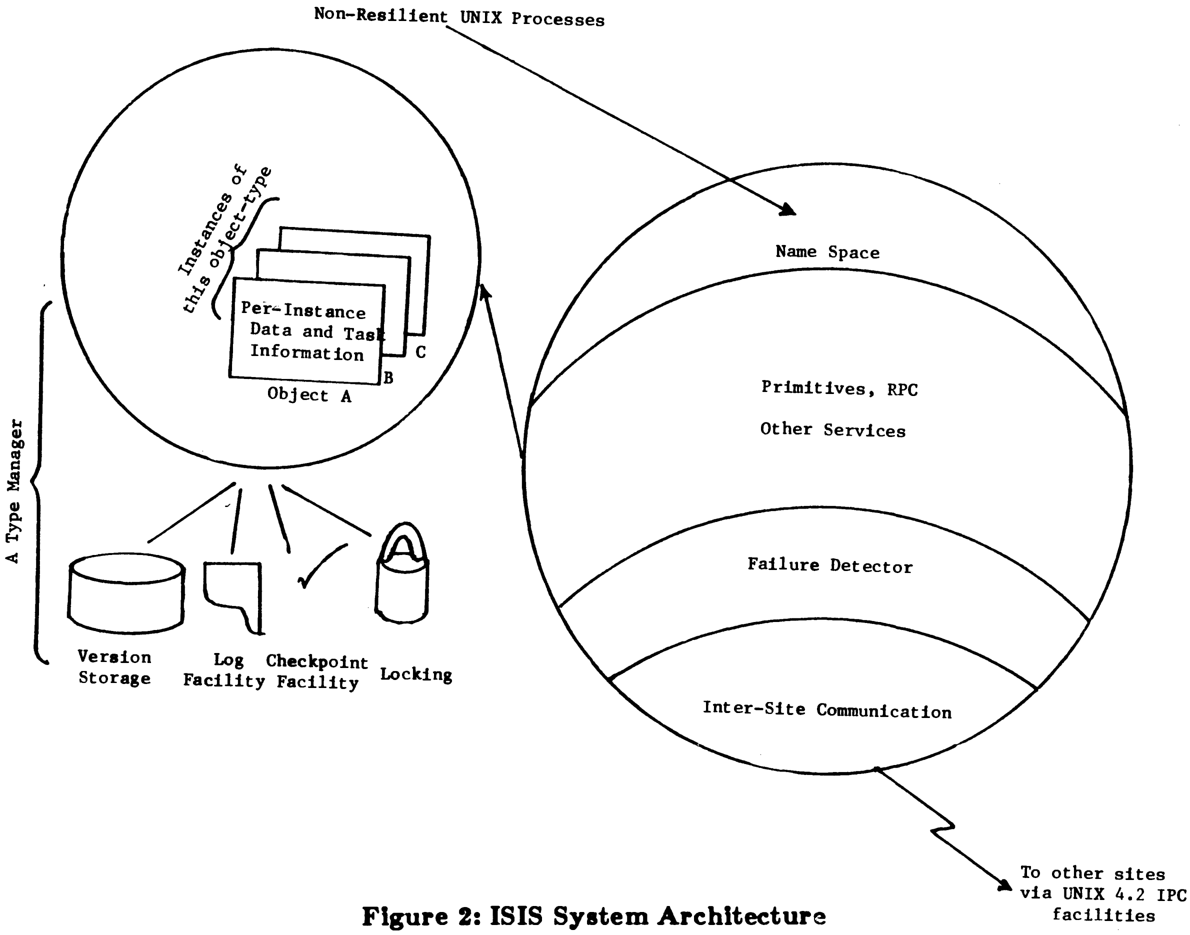

Figure 2: ISIS System Architecture

tional storage system capable of storing multiple versions of each data item. Figure 2 shows how *ISIS* is structured at a typical site, illustrating the *system* process, in which communication-related support is implemented, and a *type manager* process. Support for object creation, capability management, and the name space reside in the system process. Each type manager is responsible for one type of resilient object, and implements storage, concurrency control, and the progress mechanism.

Late in 1984, we expect to have an initial version of the system operational. Using this *ISIS* prototype, simple applications will be constructed, such as a resilient mail and memo management system, and a resilient control program for overseeing computations that are underway at multiple sites, restarting any activities interrupted by a failure at a site that is still operational. However, this system will lack many of the features we believe that *ISIS* should

UNIX is a trademark of AT&T Bell Laboratories.

eventually support. An expanded version is planned for the future.

5. Review of Other Distributed Computing Projects

A number of other projects are investigating issues in fault-tolerance that relate to the research described above. The *ISIS* object specification technique is modeled after ARGUS [ARGUS], a programming language (and runtime system) that allows users to specify new object types and has special support for nested transactions, and recovery. ARGUS supports only single-site objects, while *ISIS* objects are distributed. Nevertheless, *ISIS* employs version storage and locking mechanisms similar to the ones developed for ARGUS. ARGUS was not intended for building applications that remain available despite failures, and, not suprisingly, it would be much harder to implement such an application in ARGUS than in *ISIS*.

The *ISIS* system design has also been influenced by work of the EDEN project [EDEN], which examines kernel support for single-site objects. Fault-tolerance, however, is a secondary issue in EDEN. Other systems that use single-site objects or processes and support nested transactions include CLOUDS [CLOUDS], TABS [Spec], and LOCUS [LOCUS]. LOCUS provides support for replication in directories and files, but not more general objects, and does not always ensure consistency. In *ISIS*, the consistency of a resilient object is never sacrificed for higher availability. Database systems supporting fault-tolerance include System D [SYSD], ADAPLEX [ADA], System R* [R*], and SDD-1 [SDD1]. Many of the basic ideas relating to version storage and concurrency in abstract type systems were explored by the SWALLOW project [Reed] and in Moss' work on nested transactions [Moss]. At Berkeley, the CIRCUS project is exploring the semantics of replicated procedure calls, a problem with considerable bearing on ISIS [CIRCUS].

A different approach to fault-tolerance is exemplified by the TANDEM NonStop operating system, which shadows each process with a single backup process and thereby ensures that computations will make progress despite failures [TAND]. The DEMOS-MP project uses a similar technique for reliability [DEMOS]. A third effort in this vein is the AUROS system, which proposes a message-spooling mechanism for fault-tolerance [AUROS]. The latter approach is essentially a restricted form of the checkpointing method that we use to ensure that progress will be made.

6. Conclusions

ISIS is an unusual distributed computing project because it has adopted an approach that makes both replication and fault-tolerance transparent. This implies that *ISIS* must address problems that do not arise in other systems. Its advantage is that users are freed from the burden of manually implementing the facilities necessary for fault-tolerant program execution. Moreover, because the system has control over replication and synchronization, it can provide a far higher degree of support for these abstractions than is available in other systems. In particular, optimizations can be supported in the *ISIS* system that would be "application specific" in other settings.

The advantages of high-level programming facilities are well known. In fault-tolerant distributed computing, the case for such an approach is compelling, because the techniques required for fault-tolerant programming are so complex. Only a high-level programming methodology can make distributed computing feasible for typical programmers. On the other hand, any such methodology is open to criticism to the extent that efficiency is sacrificed for generality. The *ISIS* project is attempting to strike a balance, relieving the programmer of much of the complexity of distributed computing, and compensating for the resulting inefficiencies by introducing mechanisms to improve performance.

Completion of the *ISIS* system will make possible the development of a wide range of fault-tolerant distributed software systems – software of a sort that is currently extremely difficult to implement. Consequently, *ISIS*, or at least the *ISIS* approach, may lead to a fundamental change in the perception of distributed programming, and in the sort of services that are routinely expected from operating systems for distributed computing environments. We believe that this approach to fault-tolerant programming will be crutial as distributed computing becomes increasingly prevalent.

REFERENCES

[ADA] Chan, A. et. al. Overview of an Ada compatible distributed database manager. *SIGMOD International Conf. on Management of Data*, San Jose, California, May 1983.

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

[AUROS] Borg, A. *et al* A message system supporting fault tolerance. *Proc. 9th SOSP*, Bretton Woods, (Oct. 1983), 90-99.

[Bayer] Bayer, R., Heller, H., Reiser, A. Parallelism and recovery in database systems. *Trans. on Database Systems* 5, 2 (June 1980), 139-156.

[Bern-a] Bernstein, P., Goodman, N. Concurrency control algorithms for multiversion database systems. *Proc. First Principles of Distributed Computing*, Ottawa, 1982, pp. 209-215.

[Bern-b] Bernstein, P., Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys* **13**, 2 (June 1981), 185-222.

[Bern-c] Bernstein, P., Goodman, N. The failure and recovery problem for distributed databases. *Proc. 2nd PODC*, Montreal, Canada, Aug. 1983, 114-122.

[Chang] Chang, J.M., Maxemchuk, N.F. Reliable broadcast protocols. *To appear: Trans. on Computer Systems*, August 1984.

[CLOUDS] Allchin, J., McKenty, M. Synchronization and recovery of actions. *Proc. 2nd Principles of Distributed Computing*, Toronto, 1983.

[CIRCUS] Cooper, E. Replicated procedure call. *Proc. 3rd PODC*, Vancouver, Canada, Aug. 1984, 220-232.

[EDEN] Almes, G. Integration and distribution in the EDEN system. Department of Computer Science, Univ. of Washington, Tech. Rep. 83-01-02, Jan. 1983.

[ISIS-a] Birman, K., Joseph, T., *et al.* A technique for implementing fault-tolerant objects. Department of Computer Science, Cornell University, Tech. Rep. TR-84-594, (March 1984). To appear, *Proc. 4th SRSD*, Silver Springs, MD, Oct. 1984.

[ISIS-b] Birman, K., Joseph, T., Raeuchle, T. Concurrency Control for Resilient Objects. *Second GI/NTG/GMR Conference on Fault-Tolerant Computing Systems*, also available as Department of Computer Science, Cornell University, Tech. Rep. TR-84-???, (May 1984).

[ISIS-c] Birman, K., Joseph, T. Low-cost management of replicated data in distributed systems. Department of Computer Science, Cornell University, Tech. Rep. TR-84-???, forthcoming (Oct 1984).

[FS] Schlichting, R.D., Schneider, F.B. Fail-Stop processors: An approach to designing fault-tolerant computing systems. *TOCS* **1**, (Aug. 1983).

[Gray] Gray, J.N., Notes on data base operating systems. *Lecture Notes in Computer Science* **60**, Goos and Hartmanis editors, Springer-Verlag, Berlin, 1978.

[Kohler] Kohler, W. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys* **13**, 2 (June 1981), 149-184.

[Kung] Kung, H., Robinson, J. An optimistic method for concurrency control. *TODS* **6**, 2 (June 1981), 213-226.

[Lamport] Lamport, L., Shostak, R., Pease, M. The Byzantine Generals problem. *TOPLAS* **4**, 3 (July 1982), 382-401.

[LOCUS] G. Popek et al, Locus: A network transparent, high reliability distributed system. *Proc. 8th SOSP* Dec. 1981, 169-177.

[Moss] Moss, J.E., Nested Transactions: An approach to reliable distributed computing. Ph.D. thesis, M.I.T. Dept. of EECS, available as M.I.T. Lab. for CS Tech. Report 260, April 1981.

[Papa] Papadimitriou, C.H., The serializability of concurrent database updates. *JACM* **26**, 4 (Oct. 1979), 631-653. *TOCS* **1**, 3 (Aug. 1983), 222-238.

[R*] Lindsay, B. *et al* Computation and communication in R*: A distributed database manager. *Trans. Computer Systems* **2**, 1 (Feb 84), 24-38.

[Reed] Reed, D.P. and Svoboda, L. SWALLOW: A distributed data storage system for a local network. *Proc. International Workshop on Local Networks*, Zurich, Switzerland, August 1980.

[Skeen] Skeen, D. Crash recovery in a distributed database system. (Ph.D. Thesis), ERL Memo M82/45, University of California, Berkeley, May 1982. (Also to appear in: Lecture Notes in Computer Science, Springer-Verlag).

[SDD1] Hammer, M., Shipman, D. Reliability mechanisms for SDD-1: A system for distributed databases. *Trans. on Database Systems* 5, 4 (Dec. 1980), 431-466.

[Spec] Spector, A., Schwartz, P. Transactions: A construct for reliable distributed computing. *Operating Systems Review* **17**, 2 (April 1983), 18-35.

[SYSD] S. Andler *et al*, System D: A distributed system for availability. IBM RJ3313 (39939), San Jose, California, Nov. 1981.

[TAND] J. Bartlett, A NonStop kernel. Tandem Computers Inc., Cupertino, California, June 1981.

[Nicolau] Nicolau, A. Parallelism, Memory Anti-Aliasing and Correctness for Trace Scheduling Compilers. *Ph.D. thesis*, Yale University, 1984.