Cornell University
Ithaca, NY 14853

THE ISIS PROJECT: R&D Status and Technical Report

May 4, 1985 - Aug 4, 1985.

Kenneth P. Birman

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 5378, under contact MDA903-85-C-0124 issued by the Department of Army.

Academic Staff

K.P. Birman, Principal Investigator

Students Supported During the Report Period

W. Dietrich
A. El Abbadi
K. Herley
T. Joseph
T. Raeuchle
P. Stephenson

1. Description of Progress

1.1. Overview

This report describes the accomplishments of the *ISIS* project during the six month period February - August 1985. We assume that the reader is familiar with the goals and strategy of the project, summarized in [1]. After a brief summary, we discuss areas where significant progress has been made in greater detail.

As reported previously, we completed a prototype version of the *ISIS* system [2][3][4] the first three months of the project. This software transforms fault-intolerant single-site program specifications into fault-tolerant distributed implementations, and supervises execution of the resulting code. During the second three-month period, several aspects of the system have been enhanced: the interface between external programs and resilient objects, the language used to specify resilient objects, and the command language used to control the system. We have also designed and begun construction of a performance monitoring tool and some application software.

Concurrency is the key to good performance in a distributed system: the less synchronization employed by a system, the less frequently it will be experience delays while waiting for inter-site message transmissions to complete. Recently, we achieved a basic insight into the nature of concurrency in systems like *ISIS*. This has lead to us to redesign the *ISIS* communication primitives [5], resulting in a communication subsystem that achieves very high levels of concurrency, but at the same time makes it *easier* to design high-level software that is correct in the presense of failures. The development of these primitives will probably prove to be our most important achievement of the six-month report period. By achieving high levels of concurrency while simultaneously simplifying concurrent algorithms, they represent a breakthrough in the methodology for developing of large, fault-tolerant systems.

1.2. ISIS System enhancements

The *ISIS* prototype is now largely complete. The system can be broadly split into several major parts:

1. The interface presented to client software (normal C programs executing under UNIX). Client programs treat resilient objects as the glue binding programs together into a distributed, fault-tolerant application system.
2. The object specification language, for use when the predefined types are not suitable for some application.
3. The runtime system, which provides primitives needed by resilient objects while they are executing.

For each of the above areas, we summarize recent work and status.

The "client-object" interface permits normal UNIX programs, written in C, to issue remote procedure calls to *ISIS* resilient objects using a remote procedure-call interface. If desired, multiple calls to *ISIS* can be bundled into a transaction terminating in a *commit* or *abort* (abort is the default if a client fails or the site at which a client is running crashes). In the case of a commit, changes made to data by the transaction become permanent; in the case of an abort, changes are discarded and no other transaction observes data in an intermediate state.

During the past 6 months, software to support the interface has been completed and debugged. For example, assume that a banking program has been implemented as a front-end program that interacts with users, and employs a database object in which accounts and balances are maintained. Fig. 1. illustrates a fragment of code that the front end might use to contact the database and update it, first inserting a new entry, and then initiating a "rebalancing" operation to rebuild secondary indicies for faster access. The rebalancing is done asynchronously. The example first calls the *ISIS* name service to look up the database, called "dbase", and then invokes the object twice. Note that the RPC syntax is a relatively transparent one, unlike some previous RPC proposals for C. Obviously, the procedure is free to use any other C statements or constructs that

are needed. The example does not use the BEGIN() and COMMIT() routines, hence each call executes as a separate transaction. We have built test objects of this sort, and typical calls require about a tenth of a second to return a result, a cost which is independent of the degree to which data is replicated (of course, the update does not finish at remote sites for a longer period of time, which does depend on the degree of replication). Such performance is more than adequate for most applications.

Turning to the object specification language, a number of extensions have been made to the object specification language, which is an extended version of C. These include a cobegin statement for concurrency, a toplevel statement (as in MIT's ARGUS language), remote procedure calls to other objects (including asynchronous ones), dynamic allocation and deallocation of resilient records, and dynamically specified record sizes.

Debugging an *ISIS* object is done using a translator that converts specifications into conventional C procedures that can be linked to calling programs and debugged using normal UNIX

```
import      namespace, dbase;      /* Load interface definitions */
capability  namespace NAMES;       /* Predefined capability on namespace */
capability  dbase DB;              /* Capability on the dbase object */
struct db_entry dbregister(name, date, age, credit_lim)
  char *name, *date, age;
  float credit_lim;
  {
        /* Get capability on dbase object */
        DB = NAMES$nm_find("dbase");
        /* First register the new entry */
        DB$db_register(name, date, age, credit_lim);
        /* Initiate asynchronous data structure rebalancing */
        ASYNC DB$db_balance();
        /* Return the result to the caller */
        return(db_ent);
  }
```

Figure 1: A C procedure that looks up and then calls a resilient object

debugging facilities. This way, by the time an object is actually installed into *ISIS* it will be relatively bug-free.

Figure 2 illustrates the nm_insert and nm_find procedures in the name-space, which manages a directory of symbolic names and capabilities to which they correspond (nm_insert is invoked by the system when a new instance of an object is created). The name space is represented as a list of records, which are scanned sequentially in a do-while loop. The basic rule is that there is only one entry per name. A commit flag is used to indicate whether the entry is valid or invalid, and an invalid entry will be re-used on a subsequent nm_insert for the same name. Better concurrency control is used in the actual namespace object, but in the interests of brevity we have simplified the version given here. The basic strategy is to lock each record before accessing it, using promotable read locks that are converted to write-locks if the entry is actually written. If the transaction calling the nm_insert commits, the entry it has written becomes visible to other callers; if it aborts, other transactions are permitted to read the entry, but detect that its commit flag is clear and hence that it is invalid.

Most C programmers should be able to write object specifications of this sort -- the I/O statement, locking, and the declaration rules are the only things that distinguish this code from a normal C program. A more experienced user is, of course, able to build an object that would give better or more concurrent performance. In contrast, only a few experts can write distributed fault-tolerant programs of the sort that this object "compiles" into, and even fewer could hand-code a really sophisticated namespace with any chance of obtaining a correct result.

Although we have been working actively within the system, most of this effort is relatively technical and hence we defer a detailed discussion to a planned paper on the *ISIS* implementation. One of the more visible appears of progress, however, involves a new command language that greatly increases our control over the *ISIS* system while it is running. Commands allow the user to dynamically add and delete sites, "load" and "unload" types as they are needed, create and delete objects, list the resilient objects known at a site, etc. Figure 3 illustrates the startup file

```
/* The name-space is a list of nm_entry structures */
typedef struct nm_entry nm_entry;
struct nm_entry
{
   int          nm_flag;
   char         nm_name[32];
   cap_t nm_cap;
};

#define   NM_VALID          0x0001          /* Entry in use */
#define   NM_COMMIT  0x0002          /* Has been committed */

/* Definition of the namespace object */
resilient namespace
 created makeone;                          /* Initializes during create */
 entry nm_link, nm_unlink, nm_find;        /* Namespace routines */
 {

   resilient nm_entry names[];             /* The name-space itself */

   proc int
   nm_link(name, cap)
    char *name;
    capability cap;
    {
         register i, n;
         nm_entry ent;

         /* Look for existing entry for this name */
         n = 0;
         do
         {
            /* This I/O statement locks and then reads the n'th namespace record */
            ent <-p names[n++];

            /* A match? */
            if(strcmp(ent.nm_name, name) == 0)
                  if((ent.nm_flag&NM_COMMIT) == 0)
                     break;
                  else
                     /* Name conflicts with a previous entry */
                     abort return(-1);
         }
         while(ent.nm_flag&NM_VALID);

         /* Found entry to use */
         ent.nm_flag = NM_VALID;

         /* Copy name and capability information into record */
         strcpy(ent.nm_name, name);
         ent.nm_cap = cap;

         /* Now write it "provisionally" with the commit bit set */
         ent.nm_flag |= NM_COMMIT;
         names[n] <-w ent;
```

```
        return(0);
}

/* Read-only procedure to look up a name and return a capability */
read_only proc capability
nm_find(name)
  string name;
  {
        register n;
        nm_entry ent;

        /* Search loop, as above, getting read-lock before each I/O */
        n = 0;
        do
        {
           ent <- names[n++];
           if(strcmp(ent.nm_name, name) == 0)
                 if(ent.nm_flag&NM_COMMIT)
                    /* Return the capability on a match with a valid entry */
                    return(ent.nm_cap);
                 else
                    break;
        }
        while(ent.nm_flag&NM_VALID);

        /* Not found or entry was invalid -- return error indication */
        abort return(NULLCAP);
}
```

Figure 2: Fragment of the ISIS namespace object

used to configure *ISIS* at site "anubis"; commands like these can also be issued interactively while *ISIS* is running. The configuration file defines three *ISIS* site by giving the machine names and ARPANET port addresses at which they can be contacted (as offsets from a base-port number), then defines 5 types and loads one of them. A file for restart information is then defined, and if the system is coldstarting, the namespace object is instantiated. Otherwise, the system is told to restart from the restart file written prior to the crash.

The overall robustness of *ISIS* is steadily increasing in response to a continuing program of testing and development. Performance is good when no failures occur, particularly because of concurrent update techniques which we describe below. Recovery from *partial failures*, in which some sites remain operational has been implemented, and also gives good performance. Still needed is code to handle recovery from *total failures* (all sites fail at once), and *partitioning* (sites

```
/*
 *      ISIS configuration file for site anubis
 */

/* Site numbering and machine names */
mysite  1       anubis
site    2       osiris
site    3       amun

/* INET base port */
baseport 1200

/* Type definitions and corresponding executable files */
typedef 0 names_t       /isis/bin/namespace
typedef 1 btree_t       /isis/bin/btreeobj
typedef 2 file_t        /isis/bin/fileobj
typedef 3 queue_t       /isis/bin/queueobj
typedef 4 stack_t       /isis/bin/stackobj

/* Load the namespace.  Other types loaded as needed */
load    names_t

/* Tell ISIS what restart file to use */
restartfile=/anubis/restart_file

if      coldstart
      /* Coldstart: create namespace object */
        create "names": type=names_t, sites={anubis osiris amun}
else
        /* Otherwise, initiate restart sequence */
        restart
endif
```

Figure 3: Fragment of an ISIS command file

lose the ability to communicate with one another but do not fail). Although both of these are relatively rare events in most networks, we intend to address them eventually.

1.3. Performance monitor

P. Stephenson has designed and is now implementing a distributed performance monitoring program. This tool could be used in any distributed system, but is particularly well suited to obtaining performance information from the *ISIS* system while it runs. The program is table

driven, and employs a graphics interface to the SUN window system for output. It is possible to change the parts of the system being monitored, replay activity during a selected period of time, focus on the detailed behavior of the system while some event is occurring, plot system load and performance, and so forth. Our intention is to use the tool for overall tuning and debugging, particularly as we begin developing data migration algorithms for *ISIS*.

1.4. Applications software development

With the completion of the *ISIS* system, we are now beginning to focus on applications. There will be more to report in this area in the near future. At the moment, only some test software and a distributed game program are operational. One of our long term ideas is to port a medical database system, MDB-1, onto *ISIS*. This database system is available to us because of a collaboration with medical researchers at Columbia University, and is of particular interest because its modular structure is tailored to a resilient object environment.

1.5. Communication primitives

K. Birman and T. Joseph have recently completed a thorough re-examination of the communication problem in *ISIS*, focusing on the communication layer of the system and the ordering relationships between communication events. The solution, described in [5] is remarkable in several respects. First, it preserves a very high level of concurrency, which in a distributed system is the key to obtaining good performance. Most other work on communication primitives has not focused on this issue. Second, the ordering of events seen by users of the primitives is the same at all sites (although not "simultaneous" in the sense of a global clock). This results in simplified high-level algorithms, increasing the level of confidence that can be placed in their correctness. In effect, we have designed a set of communication primitives that eliminates the need for most synchronization by enabling a process to assume that other processes will experience the same sequence of events as it does, unless they fail first.

An example will illustrate the class of problems that arise here. Consider a process p that is updating a replicated data item maintained by a set of *data managers*. Assume that this update is performed using a *reliable broadcast:* if any data manager receives the broadcast and remains operational, all data managers will receive it. If p fails, a data manager could observe any of several outcomes:

1. The update is received prior to detection of the failure.
2. The failure is detected prior to reception of the update.
3. The failure is detected and the update is not delivered (anywhere).

It may be difficult for a data manager to distinguish cases 2 and 3. Moreover, if some managers experience the first outcome and others the second one, the overall system must still be correct. There are several ways that these problems might be addressed. By performing updates using a two-phase commit, agreement can be reached on the action to take after a failure is detected [Skeen-a]. This approach could be slow because it is synchronous. Another possibility is to discard messages arriving from a process that has failed. However, inconsistencies may arise if messages are discarded by one process but retained by another. A third alternative, representative of the general approach of our work, is to construct a broadcast protocol in which the second outcome never occurs. Using the *ISIS* communication primitives, a data manager can perform an update immediately upon receiving the corresponding message, and can take a recovery action immediately after detecting a failure; moreover, every data manager experiences the same sequence of events, or fails first.

What types of events are we including? We have distinguished three kinds "broadcasts" which are issued by one process to a set of destination processes, all having the property that they are delivered to *every* operational destination or *none*, regardless of failures[1]. The three types are:

[1]Although the broadcasts are process-to-process, processes can reside on different sites. These are thus more powerful than site-to-site broadcasts. Note that the term "broadcast" is used here to mean a software message-transmission protocol, not an ethernet broadcast (although such a hardware feature might be useful when implementing some of our protocols).

1. Independently issued broadcasts that should be delivered in a consistent order to any overlapping destinations.
2. Related broadcasts which must be performed in the order they were issued.
3. Broadcasts used to notify processes of failures and recoveries of other processes in their "process group".

Our protocols enable processes to employ consistent strategies when processing messages and reacting to failure or recoveries, without using any special protocols to decide what to do. Moreover, "race conditions" and other anomalies causes by unpredictable message orderings are are ruled out. A more thorough discussion, together with examples illustrating the extent to which the approach simplifies high-level algorithms appears in [5], which is being sent under separate cover.

T. Joseph has developed a model within which this problem can be studied formally [6]. He has found that the technique (and hence our primitives) would be useful in almost any correct message-based system. In future operating systems, we believe that communication primitives such as these will be critical to good performance, and the key to the development of correct, fault-tolerant distributed software.

2. Summary of trips and visits

K. Birman visited the University of Texas at Austin and the NASA Johnson Space Center, where he gave seminars titled "An Overview of the *ISIS* Project". He was accompanied by T. Joseph, who will remain with the project as a Research Associate in the fall, and who spoke about his work on concurrency in message-based systems. Birman also visited with Bill Joy at SUN Microsystems in California regarding a paper on *ISIS* [4] and visited the systems research center at IBM in Palo Alto (only minor local expenses were charged to the grant for this trip). Birman and several students also attended the ACM PODC conference at the end of this reporting period.

3. Status relative to planned effort.

Research is underway in all major areas of the project.

4. Fiscal status.

A summary of expenditures is attached.

5. References

1. An overview of the ISIS project. IEEE Distributed Processing Technical Committee Newsletter, June 1985.
2. K.P. Birman, T. Joseph, T. Raeuchle, A. El Abaddi. Implementing Fault-Tolerant Distributed Objects. *IEEE Trans. on Software Engineering,* TSE-11, No. 4, June 1985.
3. T. Joseph., K. Birman Low cost management of replicated data in distributed computing systems. *To appear:* ACM TOCS.
4. K.P. Birman. Replication and availability in ISIS. *To be presented:* ACM Symp. on Operating Systems Principles (SOSP), Orcas Island, Dec. 1985.
5. K.P. Birman, T. Joseph. Reliable communication in an unreliable environment. Cornell Univ. Dept. of Computer Science Tech. Report TR-85-692, Aug. 1985.
6. T. Joseph. Ph.D. dissertation. *In preparation.*

FISCAL SUMMARY

	Expenditures through 8/4/1985
Summer Salaries (Birman & Nicolau)	18,526
GRA AY sal	11,731
GRA Summer sal	18,050
Applicable employee benfits	1,853
Travel	2,905
Supplies	2,099
Equipment	67,993
Applicable indirect cost	31,653

	Anticipated through 10/4/1985
Secretarial support	
Applicable employee benefits	855
Academic year GRA	24,450
Comp.main	7,200
Publications	1,095
Supplies	301
Comp.supplies	1,333
Applicable indirect cost	7,985
Total to 10/4/85	200,254