The ISIS Project: Real experience with a fault tolerant programming system

Kenneth Birman
ken@cs.cornell.edu

Robert Cooper*
rcbc@cs.cornell.edu

Department of Computer Science, Cornell University
Ithaca NY 14853, USA

The ISIS project has developed a distributed programming toolkit[2,3] and a collection of higher level applications based on these tools. ISIS is now in use at more than 300 locations world-wide. Here, we discuss the lessons (and surprises) gained from this experience with the real world.

What has been successful in ISIS?

ISIS differs from other process-group-based systems because it integrates group membership changes with communication, and because of the multicast communication primitives we call CBCAST and ABCAST.

Virtual synchrony is a good model. Virtual synchrony underlies those aspects of ISIS that have been most successful. The approach makes it possible for a process to infer the state and actions of remote processes using local state information and events that have been locally observed. Our experiences confirm that using this property, one can often arrive at elegant, efficient solutions to problems that would be difficult to formulate—and extremely complex to implement—on a bare message-passing system.

CBCAST is important but adds complexity. We originally decided to support a causally-ordered CBCAST primitive in addition to the better-known totally-ordered ABCAST primitive because of performance. CBCAST is a one-phase protocol; when used asynchronously the initiator is *not* required to block until remote destinations have received the message. ABCAST has no non-blocking implementations. In the early versions of ISIS (where communication was quite slow), this distinction was huge. Today, ISIS performance has improved to the limits imposed by the underlying message transport facilities, yet CBCAST remains 3 to 5 times faster than ABCAST in all situations. More to the point, applications that invoke ABCAST are delayed for a significant amount of time—long enough to cause a graphics application to stutter visibly, and limiting CPU utilization of multicast-intensive programs to 30-40%. Jointly, we feel that these considerations continue to justify the code and complexity needed to support CBCAST.

What lessons did we learn?

Users want interworking. We have always advertised ISIS as a fundamentally new way to design and program reliable distributed systems. But many of our most enthusiastic users chose to apply ISIS to existing programs, or to use it on only part of their application, using existing standard network protocols for other aspects. One implication is that ISIS must co-exist with old code and other sorts of networking services, a consideration that has forced us to re-engineer parts of the system. A second implication is that for many users, adherence to standard solutions is even more important than functionality, even reliability! A prime example is that most ISIS users insist on using relatively unreliable services such as the Network File System (NFS) and Yellow Pages (YP), even though these can substantially degrade the overall robustness of their application.

The interest in ISIS for interworking has pushed us to port the system to a wide range of hardware and to

*This work is supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG2–593.

offer interfaces from a variety of languages, notably Fortran and Lisp.

On the other hand the existence of appropriate standards, namely the ARPA protocol suite and Unix, has allowed ISIS to be made available on and among a wide range of manufacturers' equipment through the efforts of our research group. In contrast, porting a system like ISIS to a non-Unix environment can be undertaken only as a fully funded commercial operation. The ability to use ISIS without moving to a new programming language, operating system and network protocol suite was crucial for many users.

Performance demands are modest. Performance of the early versions of ISIS was poor, and we expected a great deal of negative feedback in this area. This led to a major effort to improve the performance of multicasting in the most common modes, which has been successful. However, our experience now suggests that rather few ISIS applications are in any way limited by multicast performance. For most people reliability and ease of programming really are more important than pure speed.

We have also found that in cases where speed is important, general protocols will usually be outperformed by specialized solutions tuned for the particular application or hardware environment. A good example of this is in stock and bond trading room systems where fast response and large scale are required of a multicast protocol, but where there is a simple communication structure. In this simple structure many of the more troublesome failure and concurrency conditions cannot occur, and the costs incurred to avoid them can be saved.

Thus the key to satisfying user demands for performance consisted not only of speeding up the basic ISIS protocols, but of providing an interface by which users could plug in their own multicast protocols. Redesigning ISIS so that this interface was simple enough for practical use, while still maintaining the reliability and consistency semantics of ISIS has been challenging.

ISIS programs use lots of groups. Although ISIS places no limits on the number of process groups to which a process may belong, we were surprised to realize that many applications actually use large numbers of process groups. The reason is that process groups with well defined semantics are a very convenient distributed programming abstraction. Many users have adopted an objected-oriented programming style in which a group implements a distributed object and a given process may have several objects. Each object's implementation, including communication and concurrency, can be developed independently. Because ISIS guarantees proper multicast ordering when groups overlap, there is high confidence that objects will behave correctly when combined. In an unordered multicast system such as V, combining two previously disjoint process groups would require extensive algorithm redesign, especially with respect to race conditions and communication.

ISIS could provide more support for this programming style. For instance, ISIS would benefit from an interface definition language that reinforced the notion that a group implements a distributed abstract data type. Also the C++ interface to ISIS could make much more use of the object-oriented features of that language.

Small groups work best. Some of our papers on ISIS assume that all members of each group will cooperate to manage the group state or perform operations on behalf of clients. This is an appropriate model for achieving fault tolerance with small groups of 3 or 4 processes. However, as applications grow large, ISIS users have been forced to employ ad-hoc hierarchical structuring mechanisms to circumvent this limitation. A large group, encompassing perhaps hundreds of processes, is subdivided into many small groups. The small groups provide the reliability; the large group handles scale. There is a significant amount of bookkeeping required to manage such a hierarchical group. This has motivated us to extend ISIS with hierarchical group primitives, and to provide a large-group multicast for the few situations when all the members of a large group need to be contacted.

Users mean something different by "large scale". We expected that many ISIS users would have large networks, and this is indeed the case. However, where we assumed that ISIS itself would ultimately have to scale to large environments, our users needed something entirely different. Large systems are more heterogeneous than we expected, and ISIS is primarily useful in building highly robust centralized services. These centralized services are in fact distributed over a modest number of machines for reliability and performance. These users have thus been far more interested in mechanisms for connecting large numbers of client workstations to a much smaller number of centralized sites running ISIS than in actually running ISIS directly on thousands of client machines.

What did we learn from implementing ISIS?

Implementing ISIS on Unix was a good idea. We resisted the temptation to implement a special purpose operating system kernel for ISIS, despite the performance penalty that decision entailed. This made it easy for others to benefit from our work, and provide us with valuable feedback. With our experience implementing ISIS we now understand which parts of ISIS should be "kernelized" to improve performance. These include the failure detection mechanism, the default multicast transport protocol, and certain aspects of the CBCAST implementation. Most of the ABCAST implementation, and all of the higher level ISIS tools benefit less from inclusion in the kernel. Efficient sharing of message buffers should be directly supported by the kernel.

Modular operating system structures, which allow us to place our code in the kernel in a straightforward manner, are most attractive to us. We are investigating implementing ISIS on Chorus[1].

ISIS should have a modular structure. Continuing this theme, ISIS itself should be structured in terms of separate modules, which can be composed in multiple ways to give differing semantics depending on the needs of the application. For example, one might want to add a real-time communication protocol to ISIS that sacrifices virtual synchrony for timely delivery. Currently, we tend to extend the existing, monolithic system with interfaces supporting such user-specified mechanisms, but as the system grows larger this has grown harder to do.

ISIS semantics need simplifying. The detailed semantics of process groups, particularly for communication, have been extended several times, often in response to feedback from users. For example, the hierarchical group mechanisms mimic the behavior of a single large group but allocate small subgroups to perform each operation, and the basic broadcast interface now supports a *subset multicast*. However these enhancements have complicated the system's implementation and the added complexity of the ISIS interface may result in less reliable programming by our users. Where the user has a choice of primitives with differing semantics, they may choose the wrong one for their purpose. Our next changes to the system will be to unify and thereby simplify some of the multicast and group semantics. We have already removed one feature, that of permitting ABCASTs to arbitrary lists of groups and individual processes, because its effect can be achieved by the subset multicast feature. We will also provide better high-level, problem-oriented tools that choose the right primitive for the user.

The ISIS implementation has proved reliable. There is always concern that a system such as ISIS that enforces consistency throughout a local network may actually reduce reliability. There are two arguments at play here. First, that enforcing consistency whenever a single failure occurs requires all operational sites to participate in some agreement protocol, and second, that the complexity of ISIS itself may be a source of unreliability.

The first argument overstates the problem, because the ISIS recovery protocol typically involves only those sites interested in communicating with a failed site. Those sites, however, must use some timeout interval to determine that a site has failed. Choosing that timeout is a tradeoff between achieving quick failure recovery, and incorrectly deciding that a site that is merely being slow has in fact failed. ISIS allows this timeout parameter to be tuned to a particular environment.

The second argument is a legitimate concern but one that has not proved to be a problem in practice. ISIS appears to be as reliable as any compiler, database, or operating system. And in fact most problems users experience are due to unreliable network naming services, compiler bugs and operating system bugs.

Who uses ISIS?

When our project began, we could only speculate on the sorts of applications that really need an ISIS-like technology. With a community of 300 users, we have a better idea of the market for this type of technology. A substantial percentage of our users appear to have an interest in the technology primarily for evaluation or for instructional use. Excluding this group, our active current users include the following:

Systems integration projects. A number of ISIS users are building systems to fault-tolerantly monitor and control an application built using older technology. A typical user of this sort will have modified a batch application to run continuously in a networked environment, using files and pipes to interconnect the software, and perhaps exploiting simple forms of parallelism such as the ability to run several sequential programs concurrently. Use of ISIS is typically confined to the supervisory program. The need for fault tolerance is primarily to achieve the kind of reliability and consistency that users came to expect on a single mainframe computer. Users do not like the inconsistencies that arise in networks of workstations.

Financial and brokerage firms. These groups are typically attracted by the fault-tolerance aspects of ISIS and its multicasting facilities. They tend to favor ISIS over alternatives because it is a general-purpose system and because source-code is available. Several such groups evaluated ISIS V1.0 and concluded that the multicasting mechanisms were unacceptably slow; the easily extensible, faster protocols in ISIS V2.0 should allay their concerns. Financial systems are typically large, heterogeneous UNIX environments, with a relatively low load of general purpose computing and a high volume of quote-dissemination (multicast) activity.

Factory automation efforts. Several ISIS users are developing automation software for factory floor environments. The reliability requirements in this environment are obvious. This appears to be one of the few settings where users have been drawn to ISIS primarily for its computing model.

Telecommunication switching systems. Several major telecommunication companies are using ISIS to prototype control software for next-generation switching and control systems. Of course, the current implementation of ISIS is not well-tuned for this kind of extremely demanding embedded application, but ISIS does provide an excellent prototyping environment. Later an ISIS-derived technology oriented to real-time environments could be used in the production system.

Distributed applications at Cornell. At Cornell, as elsewhere, many users are working with ISIS as a base technology for building other sorts of applications. Within our department, Keith Marzullo and Mark Wood are developing the META system[4] for monitoring distributed sensors and triggering actions as needed. By using ISIS they are able to focus on the difficult issues of implementing the sensor and actuator database and query system, rather than reimplement many of the ISIS mechanisms. Robbert Van Renesse is building a still higher-level system, for graphically monitoring a distributed application and specifying control actions through a powerful control language and user interface.

Alex Siegel is developing a distributed file system, Deceit[5], that provides file replication, fault-tolerance, and mechanisms for integrating large numbers of separate file servers into a coherent large-scale file system. He uses ISIS within Deceit to keep track of replicated file state, but for compatibility uses an NFS-based protocol to communicate with disk servers and clients and to transfer whole files when a server recovers from failure.

ISIS is used by computer graphics researchers at Cornell to execute large parallel computations on a collection of workstations. By using ISIS this group can concentrate on their graphics algorithms, and avoid the work of maintaining their own library of communication primitives based on Unix sockets. The performance of ISIS is relatively more important than absolute reliability in this application.

Conclusions

If ISIS V1.0 was an immature system aimed, fortuitously, at what proved to be a large potential user community, ISIS V2.0 represents a more considered attempt to adapt our system to the real needs of its existing users. Looking to the future, it is unclear to us where this path will lead, but our hope is that major changes to the ISIS architecture will no longer be needed, permitting our user community to view ISIS as less of a moving target, and our research effort to shift its attention to developing distributed applications. We view the ISIS work as a stepping stone to a new and exciting class of robust, massively concurrent, and tightly integrated distributed systems. It now seems clear that there is a substantial demand for technologies in this area, and that some very interesting systems could be built. Meanwhile, several research projects are exploring support for facilities like the ones in ISIS. It seems only a matter of time before technologies such as ours are widely accepted, standardized, and widely available.

Acknowledgements

The ISIS system architecture has evolved in response to pressures from our users and to accommodate new ideas by group members. While this is too lengthy a list to include here, we acknowledge with gratitude the many contributions that these individuals have made to the system.

References

[1] F. Armand, M. Gien, F. Herrmann, and M. Rozier. Revolution 89 or Distributing UNIX brings it back to its original virtues. Technical Report CS/TR-89-36.1, Chorus systèmes, 6 Avenue Gustave Eiffel, F-78182, Saint-Quentin-en-Yvelines, France, Aug. 1989.

[2] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 123–138. ACM Press, New York, NY 10036, Order No. 534870, Nov. 1987.

[3] K. P. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual, Version 2.0.* Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853, Mar. 1990.

[4] K. Marzullo. Implementing fault-tolerant sensors. Technical Report TR 89-997, Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853, May 1989.

[5] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. Technical Report TR 89-1042, Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853, Nov. 1990.