

# SANDIA REPORT

SAND2000-8219  
Unlimited Release  
Printed February 2000

## Algorithm-dependent Fault Tolerance for Distributed Computing

P. D. Hough, M. E. Goldsby, and E. J. Walsh

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States  
Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Prices available from (703) 605-6000  
Web site: <http://www.ntis.gov/ordering.htm>

Available to the public from  
National Technical Information Service  
U.S. Department of Commerce  
5285 Port Royal Rd  
Springfield, VA 22161

NTIS price codes  
Printed copy: A03  
Microfiche copy: A01



SAND2000-8219  
Unlimited Release  
Printed February 2000

## **Algorithm-dependent Fault Tolerance for Distributed Computing**

**P. D. Hough**

Computational Sciences and Mathematics Research Department

**M. E. Goldsby**

Systems Studies Department

**E. J. Walsh**

Distributed Systems Research Department

Sandia National Laboratories  
Livermore, CA

**Abstract:** Large-scale distributed systems assembled from commodity parts, like CPlant, have become common tools in the distributed computing world. Because of their size and diversity of parts, these systems are prone to failures. Applications that are being run on these systems have not been equipped to efficiently deal with failures, nor is there vendor support for fault tolerance. Thus, when a failure occurs, the application crashes. While most programmers make use of checkpoints to allow for restarting of their applications, this is cumbersome and incurs substantial overhead. In many cases, there are more efficient and more elegant ways in which to address failures.

The goal of this project is to develop a software architecture for the detection of and recovery from faults in a cluster computing environment. The detection phase relies on the latest techniques developed in the fault tolerance community. Recovery is being addressed in an application-dependent manner, thus allowing the programmer to take advantage of algorithmic characteristics to reduce the overhead of fault tolerance. This architecture will allow large-scale applications to be more robust in high-performance computing environments that are comprised of clusters of commodity computers such as CPlant and SMP clusters.

# 1 Introduction

As the size and complexity of current and next generation computers continue to increase, the issue of fault tolerance has become a significant concern in the development of distributed computer systems and applications. Even though fault tolerance on homogeneous systems is a well-studied problem, the issues arising on very large heterogeneous platforms, like Sandia's "Computational Plant" (CPlant), are still largely unexplored. Fault tolerance is a more difficult problem for heterogeneous systems because of their decentralized processor and storage architectures, diverse processor technology (and thus strong nonuniformity in processing powers and computational loads), diverse network topologies, and different operation systems. Moreover, such systems are unlikely to have much support for fault tolerance from manufacturers. Thus, it is left to the developers to address these issues.

As a motivational example, we consider a materials characterization problem. The solution of this problem entails using an iterative optimization algorithm that executes a shock physics simulation at least once each iteration. For a small problem, this simulation takes thirty minutes. Combining this with the expected behavior of the optimization algorithm, we can estimate that solving this problem will require several days of computation time. Figure 1 shows the number of available nodes on CPlant over a two-week period. Notice that on the second day (i.e. 13 days ago), sixteen nodes become unavailable. By the third day, they are once again available, but on the fourth day, eighteen others have failed. If our application is running when these failures occur, we want to be assured that our application will complete. This is the reason why it is essential that fault tolerance capabilities are available.

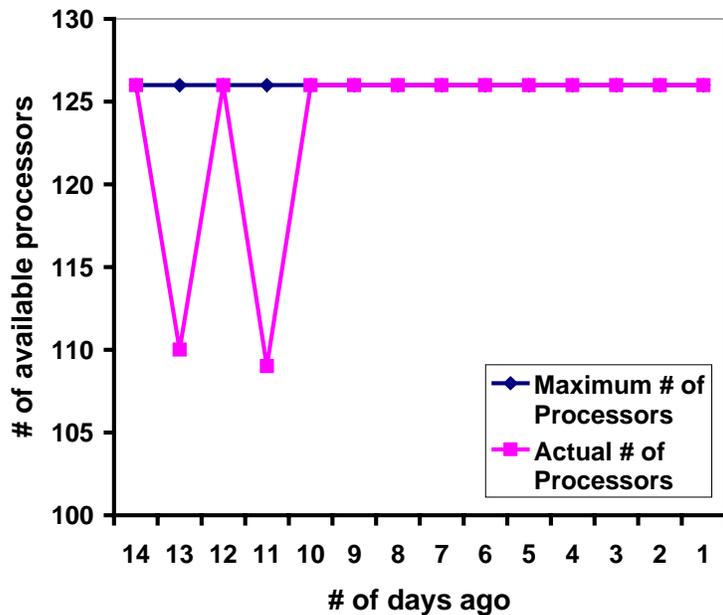


Figure 1: This figure shows the number of processors available on CPlant over a two-week period, starting fourteen days ago.

The criteria for good fault tolerance strategies are effectiveness (good coverage for common failure scenarios) and efficiency (low computational overhead). Some of this can be accomplished in the physical design of the distributed system. For example, disks may be hot swappable, and network routing may be redundant. However, to be fully effective and efficient, fault tolerance must include software that will aid in the detection of and recovery from failures. It is on the development of such software tools that we focus.

Fault tolerance can be broken down into three main layers, as illustrated in Figure 2. The bottom layer consists of a detection system. This layer is responsible for monitoring the state of the distributed system, determining when a failure occurs, and notifying the application. The top layer is the application code. This is usually a large-scale distributed simulation that may run for days. In the event of a failure, the ideal scenario is for it to recover in an intelligent and efficient manner. The middle layer is the communication layer. This is the means by which the detection system communicates with the application, and in some cases, the means by which the application communicates with itself. While all three of these layers are required for effective fault tolerance, our work focuses on the detection layer, described in the next section, and the application layer, described in the third section. The final section of this report contains a discussion of ideas for future extensions of this work.

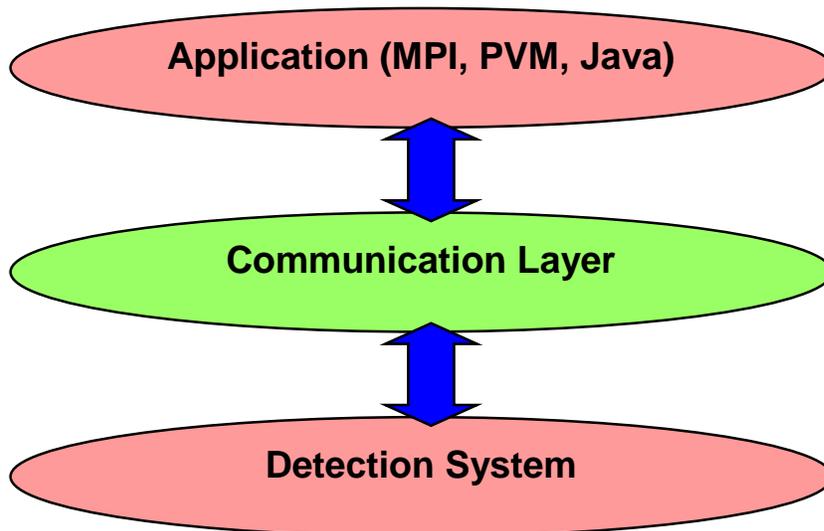


Figure 2: This figure shows the three layers of software required for fault tolerance.

## 2 Fault Detection

There are many types of failures that can occur in a complex distributed system. For example a node can fail, a network connection could fail, or a disk could fail. Furthermore, these failures may occur one at a time, or several may happen at once. In this work, we focus on detecting individual node failures and network failures insofar as they appear to be node failures. To this end, we have found that there are two primary classes of fault detection protocols in the literature.

The first is based on group membership [Birman, 1993]. This is a situation in which each node in a distributed system monitors the state of every other node in the group by direct communication with it. This is an effective method of fault detection on small systems, but since it requires all-to-all communication, network congestion becomes a significant bottleneck as the system becomes larger. An alternative to group membership protocols are gossip-based fault detection protocols [van Renesse, *et al*, 1998]. In this setting, each node communicates with a subset of the other nodes in the system. Its view of the system is determined by a combination of its own information and that received from these other nodes. This requires only selected point-to-point communication. In addition, it is easy to design a hierarchical structure for the gossip-based protocol. For these reasons, a gossip protocol is more scalable than a group membership approach and thus, more appropriate for fault tolerance on a large system. Descriptions of two new variations of gossip protocols follow.

## 2.1 Ring implementation

One approach to implementing a gossip-based detection algorithm is to embed it in a ring protocol. In this scenario, we imagine that the nodes are arranged in a ring. Each node has a left and a right neighbor with whom it communicates about the state of the distributed system. The message traffic flows from left to right, as depicted in Figure 3. There are three basic messages used in the ring. The purpose of each message, as well as its associated procedures, are described below.

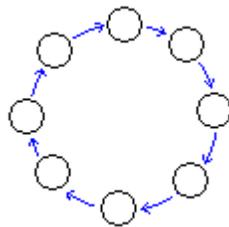


Figure 3: This figure shows a ring of system nodes. The arrows represent the flow of message traffic, directed from left to right.

**Heartbeat:** Each node is responsible for monitoring the state of its right neighbor. In order to accomplish this, it periodically sends a heartbeat message to the right. If the heartbeat is acknowledged, then the right neighbor is still functioning. If not the right neighbor is faulty.

**Failure:** If a node has determined that its right neighbor has failed, it has several responsibilities. The node first breaks its connection with the faulty neighbor, and then it establishes a new connection with the next non-faulty node to the right. This process is illustrated in Figure 4. The node then generates and sends a failure message around the ring. As each node in the ring receives the message, it marks the faulty node and passes the message to the right.

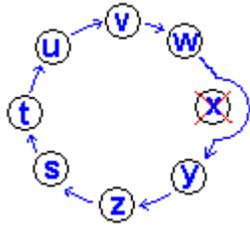


Figure 4: This figure shows the procedure for when a node fails. When node x fails, then node w sends the message “node x failed” to node y, which passes the message to node z, which passes it to node s, etc. The message travels to every node in the ring.

**Recovery:** It is often the case that a faulty node becomes functional again. In this setting, one would like to reincorporate it into the ring. Thus, we have included this capability in our detection system. If a node is marked as faulty, then its left neighbor tries to establish contact with it. If the connection is successful, the recovered node is brought back into the ring by the following process, illustrated in Figure 5. First, the left neighbor breaks its network connection with its current right neighbor. Next, it connects to the recovered node. Finally, the recovered node makes a network connection to its right neighbor. When this process is complete, the left neighbor composes and sends a recovery message around the ring. As each node receives the message, it marks the recovered node as being back in the ring and passes the message to the right.

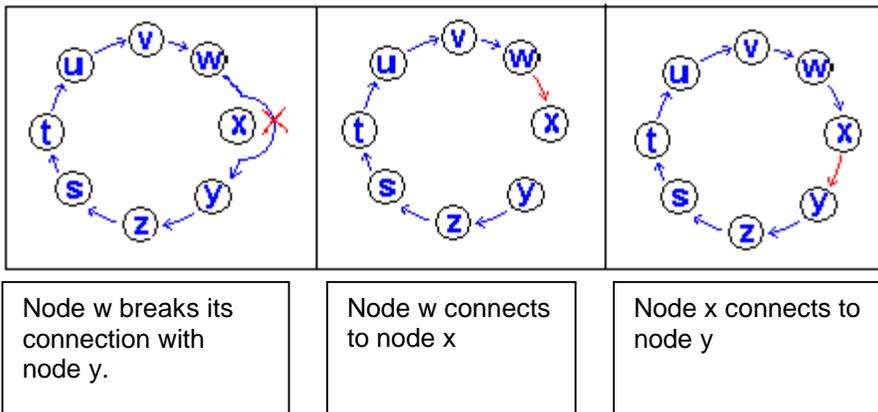
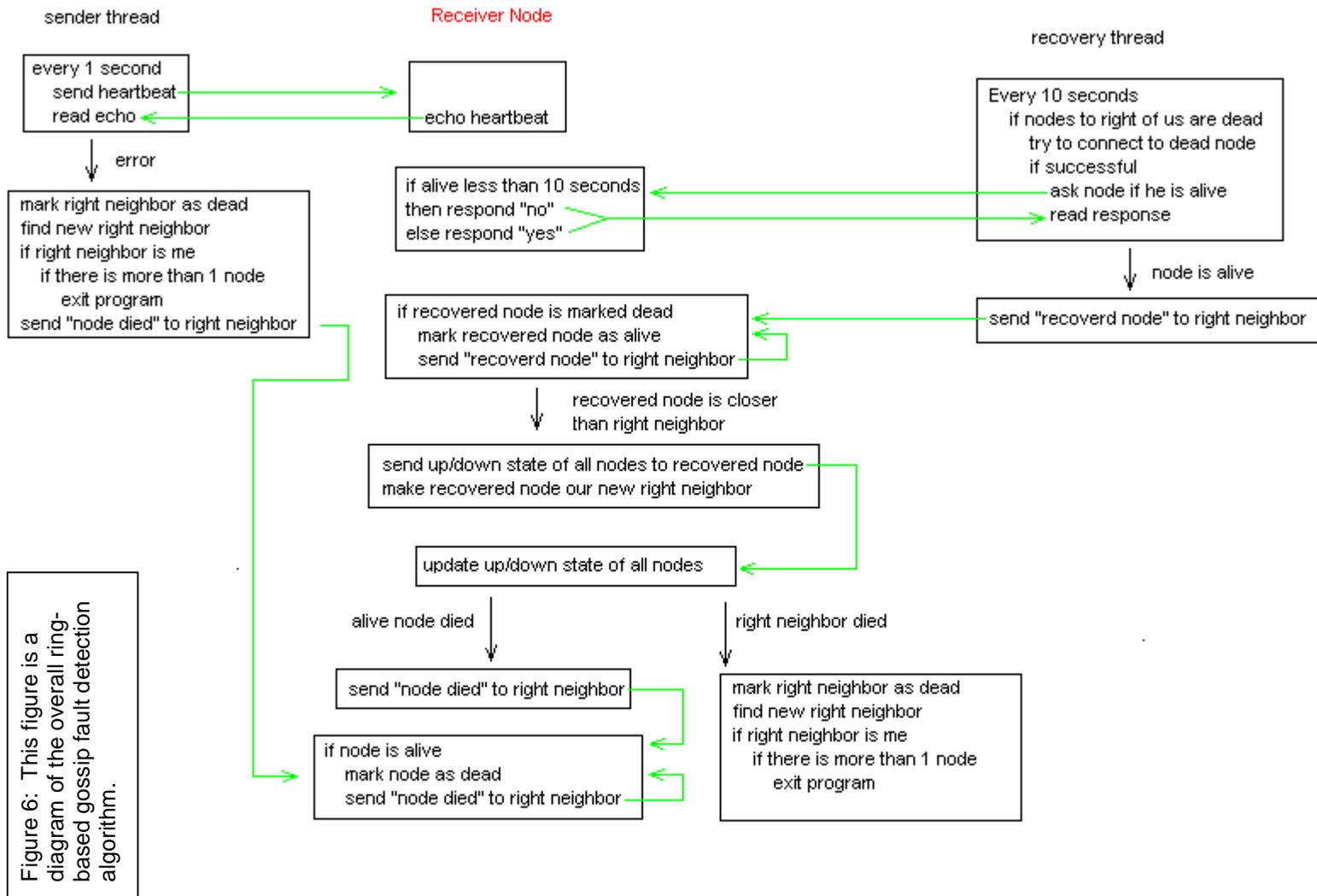


Figure 5: This figure shows the recovery process.

There are many intricacies involved in tying these messages and procedures together into a full-scale fault detection protocol, but they can be boiled down to three primary threads of logic. These threads and their interactions are summarized in Figure 6. More details are available on request.

## 2.2 An implementation based on randomness

Another approach to implementing gossip-based detection protocols involves a random communication pattern. In other words, a node does not communicate with a fixed set of nodes as in the ring protocol. Instead, a node randomly chooses another node in the system with whom to communicate. It sends its information on the state of the system, and the receiving node merges the incoming information with its own in order to obtain an updated state. This also differs from the ring protocol in that the message contains the state of the entire



system rather than information only about particular nodes. Our collaborators in the High-performance Computing and Simulation (HCS) Laboratory at the University of Florida have implemented such a protocol and have conducted thorough simulative studies.

In addition to developing the basic protocol described above, the researchers in the HCS Lab have developed a number of improvements and enhancements. Improvements include building a hierarchy into the fault detection protocol and piggybacking protocol messages on the message traffic of the application. Studies indicate that these techniques improve the scalability of the basic gossip protocol. The most notable enhancement is a scalable consensus algorithm. This allows the nodes in the distributed system to agree on the state of the entire system so that appropriate recovery action can be taken.

For a detailed account of the work done at the HCS Laboratory, see [Burns, *et al*, 1999; Ranganathan, *et al*, 1999].

### **3 Algorithm-dependent Fault Recovery**

There are two fundamentally different methodologies for fault recovery. The traditional approach is to make fault tolerance as transparent to the user as possible by using techniques such as checkpointing the entire state of the application or replicating processes. The advantages to this approach are that it requires very little work on the part of the application developers, and it is usually easy to incorporate. This type of fault tolerance has worked well in loosely coupled distributed applications with little or no interdependence between processes. The nature of scientific applications is somewhat different. In a typical application, each process requires information from other processes in order to proceed. Thus, the success of the application depends on all of the processes running to completion. While checkpointing and replication are adequate means of ensuring completion, they place limitations on the throughput and the availability of the distributed system. Furthermore, checkpointing can incur a substantial overhead, and the checkpoints are often usable only on processors with the same architecture and operating system as the processor where they were taken. These limitations are unacceptable to the developers of scientific applications, and therefore, a different approach is required.

A more efficient approach to fault recovery is application dependent. This involves exploiting the characteristics of an application in order to implement fault recovery in an efficient manner. While this is more intrusive to the application, it can allow for substantial computational savings. Replication can be eliminated, and the architecture-dependence of checkpointing can be avoided. Since each application is different, it is not a trivial task to develop one set of generic tools for this type of fault tolerance. However, it is possible to provide examples for various classes of applications. We have implemented fault tolerance in two very different applications. These are described below.

### **3.1 Asynchronous Parallel Direct Search (APDS) Optimization**

One type of optimization method that is popular for use with engineering problems is known as Parallel Direct Search (PDS) [Dennis and Torczon, 1991]. This is an inherently parallel optimization method that has demonstrated robustness in typical engineering settings. PDS has a couple of features that lend it to fault tolerance. The first is the inherent parallelism. This results in loose coupling between processes, which makes implementing fault tolerance easier to implement. The second is the fact that PDS can lose some information and still be guaranteed to find a solution. This means that there may be no overhead associated with some failures. There is one drawback to PDS, however. It is an iterative algorithm with a synchronization point at each iteration. This not only makes implementing fault tolerance difficult, but it is also generally undesirable in a heterogeneous computing environment.

To eliminate the synchronization problem, we implement PDS in an asynchronous manner. Not only is this more suitable for the heterogeneous environment, but it allows the flexibility necessary to incorporate fault tolerance. In fact, a high degree of fault tolerance comes at a very low cost. The asynchronous PDS is implemented in a peer-to-peer fashion, so there is no single point of failure in the algorithm. If a node fails, it is either ignored or the process is restarted on another node with only a small packet of information received from another process. The result is an algorithm whose fault tolerance is limited only by the communication architecture underlying it and that requires no checkpointing whatsoever, resulting in extremely low overhead for fault recovery. Numerical experiments run on CPlant have shown that, unlike the original PDS, this asynchronous implementation runs to completion in the presence of failures, and it requires less time than the original version. For more details, see [Hough, *et al*, 2000].

### **3.2 Infrastructure for Distributed Enterprise Simulation (IDES)**

Another class of applications is represented by the Infrastructure for Distributed Enterprise Simulation (IDES) [Johnson, *et al*, 1998]. This is a distributed simulation framework that is capable of handling massive enterprise simulations with large numbers of simulation entities. One example of its use is in simulating the nuclear weapons complex. Unlike PDS, IDES must recover any information that it loses; however, it has two features that are conducive to fault tolerance. It periodically comes to a synchronized state in which there is no communication in progress, which provides a natural opportunity for checkpointing; and it is written in Java, so Java serialization can be used to checkpoint and recover the program state. Another feature is that it can accept communication from external programs. This is used to resolve differences between real time and virtual time, but it can also be used to allow IDES to communicate with the fault detection services.

We have implemented a fault-tolerant version of IDES. As part of this work, we developed a more general tool that can be used to capture periodic checkpoints of an application and recover from a failure by restoring the checkpoints. The current version of the tool makes use of the application's ability to bring its communication to quiescence. Details on the fault-tolerant version of IDES will appear in a forthcoming technical report.

## 4 Conclusions

As the trend toward heterogeneous distributed computing platforms, like CPlant, continues, fault tolerance will continue to be an important issue that should not be ignored. Though we have developed a number of successful tools and strategies, there are still many challenges to be addressed. A brief summary of questions we plan to address follows.

1. We can currently detect node failures; however, when a symmetric multi-processor machine is included in the distributed system, this provides insufficient coverage of application failures. Therefore, we will be extending our fault detection protocols to include the case when SMP machines are included in the distributed system.
2. A failure may occur when an application is doing file I/O. We will investigate the inclusion of file I/O in the recovery scheme.
3. The ring-based gossip protocol as it is currently design is not scalable. In order to correct this problem, we will embed in a hierarchy of rings, drawing on experience from the Totem project [Moser, *et al*, 1996].
4. There are many distributed applications at Sandia that are concerned about fault tolerance. We will consider fault tolerance issues arising in some of these applications, such as the Lilith and the Simulation Intranet projects.
5. In order to be a complete fault-tolerant system, a communication substrate must be included between the detection and application layers. We will investigate various communication architectures and incorporate that which is most appropriate.
6. Multi-agent systems can significantly improve the performance of applications for which they are appropriate. Fault tolerance is a potential candidate for a multi-agent solution, so we will be evaluating the appropriateness of such a solution.

## 5 References

K. P. Birman, "The Process Group Approach to Reliable Distributed Computing", *Communications of the ACM*, Volume 36, Number 12, pp. 37-53, December 1993.

J. E. Dennis, Jr. and V. Torczon, "Direct Search Methods on Parallel Machines", *SIAM Journal on Optimization*, Volume 1, Number 4, pp. 448-474, 1991.

P. D. Hough, T. G. Kolda, and V. J. Torczon, "Asynchronous Parallel Pattern Search for Nonlinear Optimization", *Technical Report SAND2000-8213*, Sandia National Laboratories, Livermore, California, January 2000. (*Submitted to SIAM Journal on Scientific Computing.*)

M. M. Johnson, A. S. Yoshimura, M. E. Goldsby, C. L. Janssen, and D. M. Nichol, "Infrastructure for Distributed Enterprise Simulation", *Technical Report SAND98-8224*, Sandia National Laboratories, Livermore, California, January 1998.

L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System", *Communications of the ACM*, Volume 39, Number 4, April 1996.

M. W. Burns, A. D. George, B. A. Wallace, "Simulative Performance Analysis of Gossip Failure Detection for Scalable Distributed Systems", *Cluster Computing*, Volume 2, Number 3, pp. 207-217, 1999.

S. Ranganathan, A. D. George, R. W. Todd, and M. C. Chidester, "Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters", submitted to *Cluster Computing*, 1999.

R. van Renesse, Y. Minsky, M. Hayden, "A Gossip-Style Failure Detection Service", In *Proceedings of Middleware '98*, 1998.