Proceedings of the
13th International Conference on Parallel and Distributed Computing and Systems
Anaheim, California, pp. 480-485, August 2001.

FAULT-TOLERANT CLUSTER MANAGEMENT FOR RELIABLE HIGH-PERFORMANCE COMPUTING

MING LI DANIEL GOLDBERG WENCHAO TAO YUVAL TAMIR

Concurrent Systems Laboratory
Computer Science Department
UCLA
Los Angeles, California 90095
{mli,dangold,wenchao,tamir}@cs.ucla.edu

ABSTRACT

Clusters of COTS workstations/PCs are commonly used to implement cost-effective high-performance systems. A central coordinator/manager is often the simplest way to implement many of the operations required for managing these distributed systems. These operations include scheduling of parallel tasks, coordination of access to limited resources, as well as high-level coordination of fault tolerance mechanisms and interactions with external devices. A key disadvantage of using a central manager is that it becomes a critical single point of failure. The UCLA Fault-Tolerant Cluster Testbed (FTCT) project is focused on the implementation of fault-tolerant management for clusters. Unlike most other cluster management projects, our approach is based on active replication and voting and focused on tolerating failure modes other than fail silent and minimizing interruptions to management operations. We describe key aspects of the design and implementation of the FTCT and provide preliminary evaluation of the overheads incurred by our management mechanisms.

KEY WORDS

Cluster Manager, Multicast, Replication, Recovery

1. Introduction

Operating systems such as Amoeba [17], that are designed from the ground up for distributed systems are best suited for managing clusters and multicomputers. However, the success of cluster computing is based on the use of low-cost, widely-used commercial off-the-shelf (COTS) hardware and software. Thus, in practice, each node of a cluster runs a local copy of what is essentially a uniprocessor OS [1, 4]. Hence, any system-level management must be done by ‘‘middleware,’’ above the OS [10].

The cluster-level management middleware is key to the operation of clusters. For example, it has long been known that for parallel tasks consisting of communicating processes, independent scheduling of the processes on different nodes is inefficient [18]. Cluster-level management is needed for coscheduling (gang scheduling), where the scheduling of processes on different processors is coordinated so that all the processes of a particular task are running at the same time. Similarly, the assignment of processes to processors has to be coordinated across the entire system, taking into account such factors as the load on different nodes or specific resources connected to particular nodes. The cluster management tasks could be decentralized. However, most clusters employ centralized managers since they are simpler to design, implement, and debug [10, 14].

For many applications of clusters (e.g., Internet servers), high reliability and availability are key requirements. In particular, this is the case for the JPL/NASA Remote Exploration and Experimentation (REE) project, whose goal is to use COTS hardware and software to deploy scalable supercomputing technology in space. In this environment, fault tolerance is more critical than for most earth-bound systems due to the much higher fault rate of COTS hardware in space. For cluster management in general but for the REE application in particular, a key problem with centralized management is that the failure of the manager leads to the failure of the entire system. Most cluster managers do not deal with this problem. Some projects, such as Sun’s Grid Engine Software [21], use a cold spare approach, where a backup replica of the manager detects the failure of the primary and takes over its tasks. In this case, the manager failure mode is assumed to be fail-stop [20], i.e., the managers never generate incorrect results. Since the fail-stop assumption is often violated, this approach can result in poor system reliability. Furthermore, recovery of the management functionality on the cold spare can take a long time (for example, up to a minute on Sun’s Grid Engine Software), resulting in unacceptably long service interruptions.

The focus of the UCLA Fault-Tolerant Cluster Testbed (FTCT) project is to develop and evaluate algorithms and implementations of fault tolerant cluster managers. Some of the critical factors driving this work

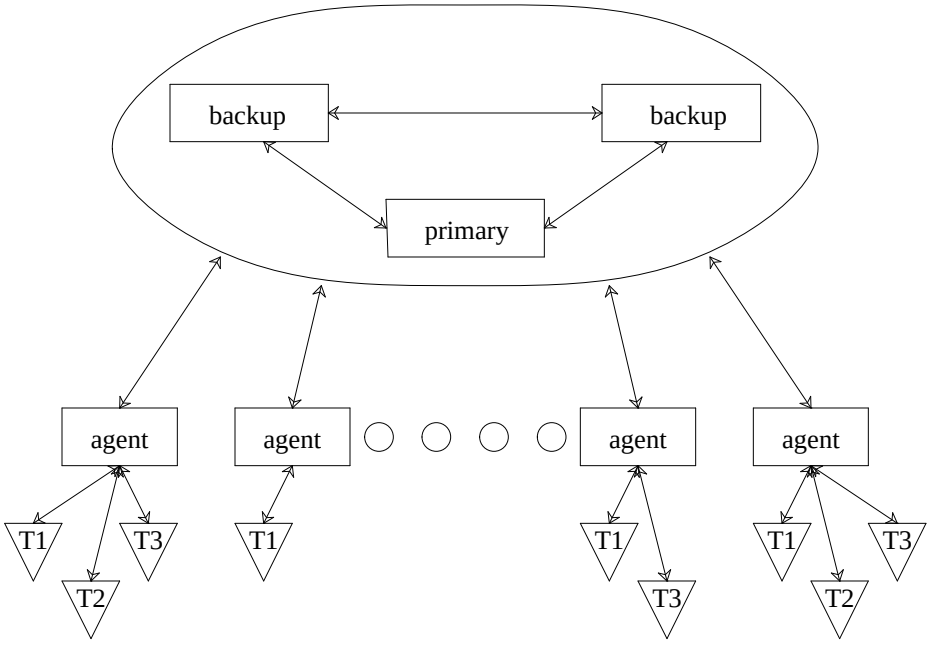

Figure 1: System Structure

are the need to deal with realistic fault models, the need to minimize the performance and power overheads of the fault tolerant mechanisms, and the need to support soft real-time requirements. In addition, the cluster management middleware must provide the mechanisms needed to support, as a separate layer, application-level fault tolerance for critical applications.

This paper describes the design and implementation of the FTCT management layer. The system provides gang scheduling for MPI parallel applications. The management layer is based on authenticated communication (signed messages[13]) and full replication and voting that allow the cluster as a whole (i.e., the management layer) to survive arbitrary single node failures with virtually no impact on application tasks that are not using the failed nodes. The system is currently operational, allowing for preliminary experimental results to be presented.

2. System Overview

The overall structure of the system is shown in Figure 1. The system consists of three components: a group of managers, an agent process on each node, and a library for user applications. The manager group is responsible for resource allocation, scheduling, and coordination of system level monitoring and fault recovery procedures. The agent on each node allows the manager group to control the node and transmits node status information (e.g., heartbeat, identity of terminated processes) to the manager group. From the manager group each agent receives messages to start, kill and schedule user applications as well as to start a new manager replica if a manager fails.

The manager group consists of three manager replicas — a primary replica and two backup replicas. All the manager replicas independently process every message and independently transmit commands to the agents. As described in the next section, the system ensures that messages are delivered to all the manager replicas in the same order and that all manager activities are deterministic. Messages exchanged among managers, and between managers and agents are *authenticated* (signed[13]) to ensure that faulty nodes cannot forge messages from other nodes, even if the message is forwarded by the faulty node. The use of authenticated messages is critical since all messages to the manager group are sequenced by the primary manager replica and forwarded to the other manager replicas.

Agents can act only when receiving identical authenticated commands from at least two manager replicas. Hence, a manager replica that stops or generates incorrect commands cannot corrupt the system. If a manager replica fails, a new manager replica is restarted using the states of the remaining two manager replicas.

Periodic heartbeats is one of the two basic mechanisms for determining the status of the system. Specifically, the agents send periodic heartbeats to the manager group and the manager replicas send periodic heartbeats to each other. As discussed later, some additional status information is piggybacked on the heartbeat messages. A faulty agent cannot corrupt the entire system. Hence, during normal operation, it is sufficient to rely on periodic heartbeats to (eventually) detect problems with agents. On the other hand, faulty outputs from the manager replicas must be detected immediately since they can easily corrupt the entire

system. The mechanism used to detect faulty manager replica outputs is the identification of conflicting (or missing) messages from the manager replicas. If an agent receives conflicting messages from the manager replicas, it reports this to the manager group. The manager replicas then enter a diagnostic procedure, where their states are compared and, possibly, a faulty replica is identified and terminated.

```
BEGIN LOOP
    IF there are any expired time events THEN
        push the event into event queue;
    IF there are any delivered messages THEN
        push the message into event queue;
    IF the event queue is not empty THEN
        pop out an event with the highest priority;
        process the event;
    ELSE
        block until new event happens;
END LOOP
```

Figure 2: The Event Loop

The managers and agents are event-driven. An event is a delivered message or a local time event (see Section 2.2). The event loop to process these events is shown in Figure 2. The processing of the events is priority-based. When an event happens, it is assigned a priority and inserted into the queue based on its priority. Normally, events are processed in priority order without preemptions. This can be problematic if the handling of some low priority events is time consuming and should be preempted by newly arrived high priority events. In order to handle this case efficiently, we employ a lightweight user-level thread library that allows low priority event handlers to periodically yield control to the main event loop.

3. Active Replication

As discussed earlier, detection of arbitrary (Byzantine) failures requires multiple active replicas and continuous comparison of the results. In order to avoid a lengthy interruption for recovery when a discrepancy is detected, more than two replicas are needed. With the classic TMR scheme that is currently implemented in FTCT, any single failure can be tolerated since there will always be two manager replicas that agree on each result.

In active replication, all replicas of a process execute and generate outputs independently. In the absence of failures, they must produce the same outputs in the same order — *output consistency*. In order to achieve output consistency, two conditions must be satisfied [3]: *input consistency* — the set of inputs delivered to every correct replicas must be identical and in same order, and *processing consistency* — each replicas must perform consistent operations and produce an identical set of output, in processing a consistent set of input.

In order to achieve processing consistency, the manager replica implementation must be deterministic. Achieving this deterministic behavior is complicated by the need to handle timer events, the fact that messages do not arrive at all the replicas simultaneously, and the priority-based event processing coupled with multithreading that allows a handler to yield control. The next two subsections describe how these problems are addressed.

3.1. Reliable Multicast

The input consistency condition implies that a reliable group communication protocol has to be used to transmit messages to the manager replicas. The multicast protocol must be *reliable* — a message will be delivered to all fault-free replicas or will be delivered to none of them. The protocol must also be *atomic* — all messages, sent by different sources, must be delivered to all fault-free replicas in the exact same order.

Many reliable atomic multicast protocols have been described in the literature and/or implemented in real systems. These protocols vary in complexity, performance, overhead, and ability for fault tolerance. We chose a protocol similar to the group communication protocol used in Amoeba [11], because of its simplicity and efficiency [7]. Specifically, we use a sequencer-based protocol, with the primary manager replica as the sequencer. Senders from outside the manager group communicate with the group by sending messages to the primary manager replica only. The primary manager replica assigns a sequence number to the message and forwards it to all the backup manager replicas. A reliable point-to-point communication protocol is used to transmit the message to the primary replica and then to forward the message to the backup replicas.

The reliability of the multicast is assured by the underlying reliable point-to-point communication as long as the primary manager replica does not fail. However, the primary manager replica may fail before it successfully forwards a message to all the backup replicas. In that case, some of the manager replicas may receive the message while others may not, thus violating the reliability requirement. In order to be able to recover from such primary replica failures, every backup replica maintains a *history buffer* where it stores copies of all the messages it received from the primary manager. When the primary replica fails, all the backup replicas report to each other the highest sequence number that each has received from the primary replica. The one with the highest sequence number becomes the new primary manager, since it is ahead of others. The new primary manager then sends copies of messages which were missed by the other backup managers to each of them.

Hence, a message received by one manager replica will be eventually received by all the replicas, thus meeting the reliability requirement.

The size of the history buffer at each manager replica is limited. Hence, the replica must reclaim the storage of messages once they have been delivered to all the other replicas. This is accomplished by piggybacking the message receive sequence number (RSN) on the heartbeat messages exchanged among the replicas.

The primary manager replica has a critical role in the operation of the cluster since it receives messages from the agents and forwards them to the backup manager replicas. Thus, a faulty primary manager replica is in a position to generate forged messages as well as to modify, discard, or reorder messages it is supposed to forward to the backup replicas. The possibility of the primary manager replica forging agent messages is eliminated through the use of authenticated communication (signed messages[13]). Backup manager replicas discard messages that are not properly authenticated. Authentication coupled with error-detecting codes also allow the backup replicas to detect and discard modified messages.

Discarded or reordered messages lead to agents receiving inconsistent acknowledgements from the manager replicas. Agents report such inconsistencies directly to all the replicas. The manager replicas then initiate a self-diagnosis procedure that can lead to the termination of the faulty replica and the initiation of a replacement.

3.2. Group Time Events

As mentioned earlier, the manager group is event-driven, where events are either arriving messages or time events that are invoked by a local timer. Examples of time events are the termination of a time slice given to a particular parallel task or a timeout triggered when a message to an agent fails to be acknowledged. The time events must be processed by all manager replicas in the same order with respect to other time events and with respect to message arrival event. Hence, the time events must pass through the sequencer (the primary replica) in the same way that messages do. Thus, under normal conditions, the backup replicas do not process time events based on their local times. Instead, the primary replica timer triggers the time event and is then sequenced with the message arrival events and forwarded with an appropriate sequence number to the backup replicas.

The problem with the above approach is that if the primary backup fails the time event may never occur. This problem is solved by scheduling ‘‘backup time events’’ on each of the backup replicas using the backup replica’s own timer. These backup time events are scheduled for some fixed delay after the primary time event is scheduled. Under normal circumstances, the primary time event is triggered on the primary replica and then delivered in sequence to the backup replicas before the backup time events are triggered. The backup replicas then dismiss the corresponding backup time event.

If a backup time event is triggered, that implies a possible faulty primary manager. Hence, the backup replica initiates a manager group self-diagnosis procedure that involves state comparison. As a result of this procedure, one of the replicas may be identified as faulty and terminated. If no faulty replica is identified or if the faulty replica is not the one reporting the problem, the primary (possibly a new primary) sequences the time event and forwards it to the other replicas so that it can be processed as a normal time event.

4. Gang Scheduling

As discussed earlier, the manager group maintains information about the system based on periodic node status information from the agents. New tasks are submitted to the cluster through the manager group. The manager group then assigns the task’s processes to nodes based on the state of the nodes and, possibly, special requirements of particular processes of the task.

Gang scheduling of tasks is done by the manager group using the Ousterhout matrix algorithm [18] with the simple ‘‘first fit’’ policy. When it is time to switch a task, the manager group send the appropriate message to all the agents of the relevant nodes. As in many other cluster managers [10], agents stop and resume processes on their nodes using the SIGSTOP and SIGCONT signals, respectively.

5. Experiments and Results

The FTCT currently consists of eight PCs, each with dual 350 MHz Pentium-II processors, interconnected by a high-speed Myrinet [6] LAN. For these experiments, the nodes operating system is Solaris x86.

The processing and associated power consumption of the manager group and agents are system overhead that should be minimized. This is particularly important for the REE application in space, where the number of nodes and the available power are severely limited. One simple way to measure the overhead is to evaluate the fraction of ‘‘processing’’ that becomes unavailable to the application when it is executing on the same node as a manager replica. The manager overhead is largely dependent on the rate of status report messages (heartbeats) that it must handle. Figure 3 shows the percentage of processor time used by the managers given different event processing rate.

When a manager replica fails the system loses its ability to mask a failure of another manager replica. Hence, a new replica must be instantiated as quickly as

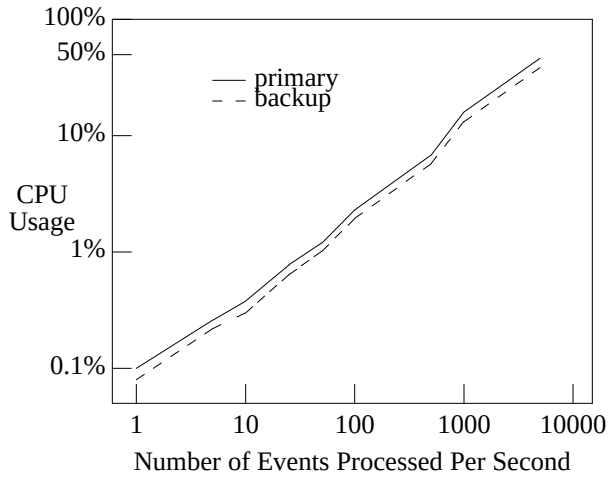

Figure 3: The overhead of a manager.

possible. We measured the recovery time from a manager replica failures — the time from when a manager replica failure is detected until a new manager starts and restores a complete (triplicated) manager group to normal operation. The results are shown in Table 1 and exclude the operating system time to load the executable from disk. Out of the time reported in the table, about 40 millseconds are spent initializing the communication system (GM) on Myrinet. Hence, the recovery time can be reduced significantly by maintaining an initialized ‘‘cold’’ manager replica, ready to accept the state from other manager replicas and become active.

Failed Manager	**Recovery Time** (msec)
primary	63
backup	60

Table 1: Manager recovery time (excluding the time to load the executable from disk).

During recovery from a manager replica failure, the functionality of the manager group is maintained by the remaining replicas. However, if the failed replica is the primary, communication from the agents will fail for a short period since the agents will continue to send messages to the primary until the agents are informed of the identify of the new primary replica. Our preliminary measurements show that, once the failure of the primary replica is detected, the time to identify and advertise the identity of the former backup replica that is now the primary replica is approximately 3.7 milliseconds.

6. Related Work

Over the past decade, a number of resource management systems for cluster computing have been implemented [10, 19, 21, 23]. A survey of 20 research and commercial cluster management systems can be found in [2]. Excluding the management fault tolerance features, the basic functionality of the FTCT is currently similar to the functionality of the GLUnix system [10]. While various projects mention the possibility of active replication of the managers for fault tolerance, none that we have seen report actually implementing and evaluating an actively replicated manager.

The Delta-4 [3] project designed an open architecture for dependable distributed systems through the use of atomic multicast protocol and specialized hardware. Active, passive and semi-active replication techniques can be used to achieve fault tolerance. Chameleon [12] provides an adaptive software infrastructure to satisfy different levels of availability requirements for user applications. Chameleon has a centralized manager, and passive replication is used to tolerate the failures of this manager.

Reliable group communication has served as the basis for many fault-tolerant distributed systems, such as ISIS [5], Horus [22], Totem [15], and Transis [9]. Several systems, such as AQuA [8] and Eternal [16] provide fault tolerance for distributed CORBA applications by using replicated objects.

7. Conclusion and Future Work

We have designed and implemented middleware that provides fault-tolerant cluster management using active replication. Based on this approach, the management layer can survive a much larger class of failures than other cluster management systems implemented on COTS hardware and software. Such middleware is critical enabling technology for the deployment of cost-effective supercomputing in space applications. Our preliminary overhead measurements indicate that if error detection latency of a few hundred milliseconds is acceptable, a central manager running on a relatively slow CPU can handle a cluster with a few tens of nodes with processing overhead of only a few percent. Lower detection latencies and/or larger clusters will require a hierarchical mechanism for collecting status reports.

The system is still in development. Future work will include topics such as the evaluation of mechanisms for recovery from agent failure, the integration of application level recovery mechanisms, and the characterization and optimization of the (soft) real-time performance of the system.

Acknowledgements

This work is supported by the Remote Exploration and Experimentation (REE) program of the Jet Propulsion Laboratory.

References

1. T. E. Anderson, D. E. Culler, and D. A. Patterson, ‘‘A Case for NOW (Networks of Workstations),’’ *IEEE Micro* **15**, pp. 54-64 (February 1995).
2. M. A. Baker, G. C. Fox, and H. W. Yau, ‘‘A Review of Commercial and Research Cluster Management Software,’’ *Technical Report*, Northeast Parallel Architectures Center, Syracuse University (June 1996).
3. P. A. Barrett, ‘‘Delta-4: An Open Architecture for Dependable Systems,’’ *IEE Colloquium on Safety Critical Distributed Systems*, London, UK, pp. 2/1-7 (October 1993).
4. D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, ‘‘Beowulf: A Parallel Workstation for Scientific Computation,’’ *24th International Conference on Parallel Processing*, Oconomowoc, WI, pp. 11-14 (August 1995).
5. K. P. Birman, ‘‘The Process Group Approach to Reliable Distributed Computing,’’ *Communications of the ACM* **36**(12), pp. 36-53 (December 1993).
6. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, ‘‘Myrinet: A Gigabit-per-Second Local Area Network,’’ *IEEE Micro* **15**, pp. 29-36 (February 1995).
7. F. Cristian, R. de Beijer, and S. Mishra, ‘‘A Performance Comparison of Asynchronous Atomic Broadcast Protocols,’’ *Distributed Systems Engineering* **1**(4), pp. 177-201 (June 1994).
8. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz, ‘‘AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects,’’ *Proceedings of the 17th IEEE symposium on Reliable Distributed Systems*, West Lafayette, IN, pp. 245-253 (October 1998).
9. D. Dolev and D. Malki, ‘‘The Transis Approach to High Availability Cluster Communication,’’ *Communications of the ACM* **39**(4), pp. 64-70 (April 1996).
10. D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, ‘‘GLUnix: A Global Layer Unix for a Network of Workstations,’’ *Software - Practice and Experience* **28**(9), pp. 929-961 (July 1998).
11. M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal, ‘‘An Efficient Reliable Broadcast protocol,’’ *ACM Operating Systems Review* **23**(4), pp. 5-19 (October 1989).
12. Z. Kalbarczyk, R. K.Iyer, S. Bagchi, and K. Whisnant, ‘‘Chameleon: A Software Infrastructure for Adaptive Fault Tolerance,’’ *IEEE Transactions on Parallel and Distributed Systems* **10**(6), pp. 560-579 (June 1999).
13. L. Lamport, R. Shostak, and M. Pease, ‘‘Byzantine Generals Problem,’’ *ACM Transactions on Programming Languages and Systems* **4**(3), pp. 382-401 (July 1982).
14. M. J. Litzkow, M. Livny, and M. W. Mutka, ‘‘Condor - A Hunter of Idle Workstations,’’ *8th International Conference on Distributed Computing Systems*, Washington DC, pp. 104-111 (June 1988).
15. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, and T. P. Archambault, ‘‘The Totem system,’’ *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, pp. 61-66 (June 1995).
16. L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. Tewksbury, and V. Kalogeraki, ‘‘The Eternal System: An Architecture for Enterprise Applications,’’ *Proceedings of the 3rd International Conference on Enterprise Distributed Object Computing*, Mannheim, Germany, pp. 214-222 (September 1999).
17. S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, ‘‘Amoeba: A Distributed Operating System for the 1990s,’’ *IEEE Computer* **23**(5), pp. 44-53 (May 1990).
18. J. K. Ousterhout, ‘‘Scheduling Techniques for Concurrent Systems,’’ *3rd International Conference on Distributed Computing Systems*, Miami/Fort Lauderdale, FL, pp. 22-30 (October 1982).
19. S. H. Russ, K. Reece, J. Robinson, B. Meyers, R. Rajan, L. Rajagopalan, and C.-H. Tan, ‘‘Hector: An Agent-Based Architecture for Dynamic Resource Management,’’ *IEEE Concurrency* **7**(2), pp. 47-55 (April-June 1999).
20. F. B. Schneider, ‘‘Byzantine Generals in Action: Implementing Fail-Stop Processors,’’ *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).
21. Sun Microsystems, ‘‘Sun Grid Engine Software,’’ http://www.sun.com/software/gridware/.
22. R. van Renesse, K. P. Birman, and S. Maffeis, ‘‘Horus: A Flexible Group Communication System,’’ *Communications of the ACM* **39**(4), pp. 76-83 (April 1996).
23. S. Zhou, X. Zheng, J. Wang, and P. Delisle, ‘‘Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems,’’ *Software - Practice and Experience* **23**(12), pp. 1305-1336 (December 1993).