Computer Science Department Technical Report CSD-010040
University of California, Los Angeles, October 2001.

The Design and Implementation of a Fault-Tolerant Cluster Manager

Daniel Goldberg, Ming Li, Wenchao Tao, Yuval Tamir
Concurrent Systems Laboratory
Computer Science Department
UCLA
Los Angeles, California 90095
{dangold,mli,wenchao,tamir}@cs.ucla.edu

Abstract

Cluster management middleware schedules tasks on a cluster, controls access to shared resources, provides for task submission and monitoring, and coordinates the cluster's fault tolerance mechanisms. Thus, reliable continuous operation of the management middleware is a prerequisite to the reliable operation of the cluster. Hence, the management middleware should tolerate a wide class of faults with minimal interruptions to management operations. This paper describes design considerations and implementation details of cluster mangement middleware for high performance computing in space, where fault rates are significantly higher than for earth-bound systems. We describe key detection, recovery, and reconfiguration mechanisms for different components of the system. The system is based on centralized decision making. Unlike other systems, the decision making capability is protected by active replication and the ability to restore the decision maker to full operational and fault tolerance capabilities following node failure. The management middleware is used to provide the application tasks with an out-of-band signaling capability that can be a key building block for application-level fault tolerance mechanisms. The middleware described has been implemented as part of the UCLA Fault-Tolerant Cluster Testbed (FTCT) project. Based on measurements of this implementation, we present preliminary evaluation of the overheads incurred by the management middleware.

1. Introduction

Clusters of commercial off-the-shelf (COTS) processors interconnected by high-speed SANs or LANs are increasingly used for cost-effective high-performance parallel computing. Typically, in such systems, every node runs a local copy of an off-the-shelf operating system that is not designed to manage a distributed cluster. Cluster management middleware (CMM) runs between the applications and the operating system and provides resource allocation, scheduling, coordination of fault tolerance actions, and coordination of the interaction between the cluster and external devices. The cluster management middleware is critical to overall system reliability since if the cluster management middleware fails, the system is lost.

Virtually all cluster management middleware is based on centralized decision making. While distributed decision making is possible [2, 13], centralized managers are simpler to design, implement, and debug [12, 18]. A key problem with centralized management is that the failure of the manager leads to the failure of the entire system. Surprisingly, many cluster management systems fail to deal with this problem. Other systems, such as Sun's Grid Engine Software [23], 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 [22], 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.

In order to be able to deal with more realistic fault models and minimize the interruptions to management service, an active replication [8] approach must be used to implement the central manager. The manager is replicated, and each replica performs the same management operations independently. The commands produced by these manager replicas are compared and voted on so that a majority of correct replicas can mask the failure of a minority of faulty replicas. This is the basic approach used in the UCLA Fault-Tolerant Cluster Testbed (FTCT) system.

The FTCT project is motivated by JPL/NASA's Remote Exploration and Experimentation (REE) project. The goal of REE 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. The focus of the FTCT project is on developing and evaluating algorithms and implementations of fault tolerant cluster managers. Some of the critical factors driving this work 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 focuses on the design and implementation of key mechanism for fault-tolerant cluster management middleware. The overall system structure is presented in Section 2. The central decision maker (manager) implemented using active replication is discussed in sections 3 and 4. Section 5 presents the fault detection mechanisms. Section 6 described how we utilize a unique feature of this cluster — the limited availability of a ''hard core'' that can be used as a recovery mechanism of last resort. Detection and recovery from agent failures is addressed in section 7. An out-of-band signaling mechanism for supporting application-level fault tolerance schemes is presented in Section 8. Section 9 discusses preliminary measurements of the overheads incurred by the management system. Related work is presented in Section 10.

2. System Overview

The overall structure of the system is shown in Figure 1. The system consists of four components: a group of managers, an agent process on each node, a library for user applications, and the Space craft Control Computer (SCC). The manager group performs cluster-level decision making. An agent process on each node sends node status information to the manager group, performs commands at the node on behalf of the manager group, and provides an interface between application processes and the cluster management middleware (CMM). A library that is linked with every application process is an important part of the CMM. It provides the mechanisms needed to set up intra-task communication using MPI as well as part of the interface between the application process and the local agent.

The SCC controls the entire space craft, including communication between the cluster and operators on Earth. The SCC interacts with the cluster through the manager group. Loss of the SCC implies loss of the space craft. Hence, while the entire cluster is built using commercial-

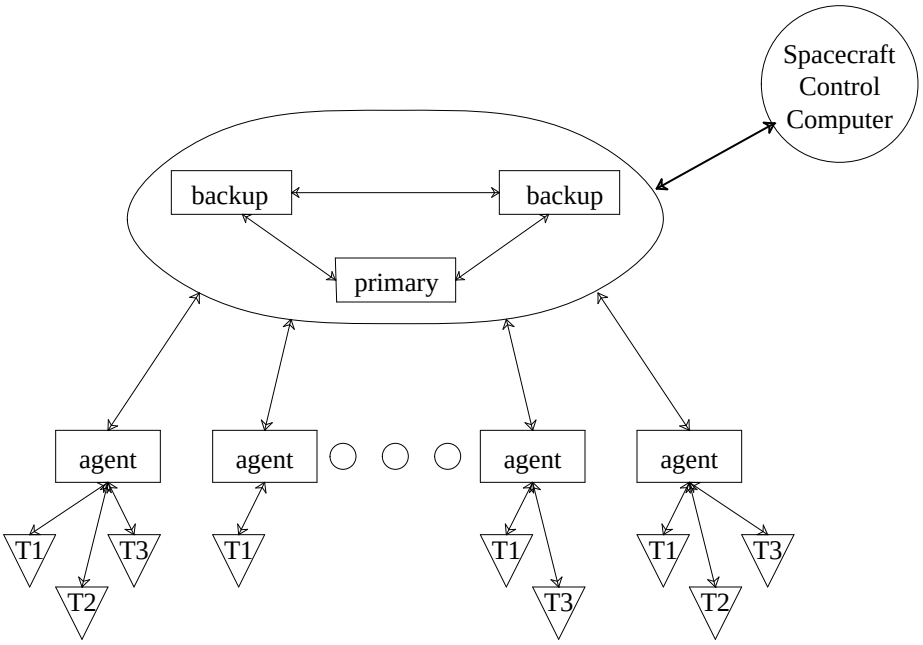

Figure 1: System Structure

off-the-shelf technology, the SCC uses radiation-hard technology and other aggressive fault tolerance techniques to ensure the survival of the space craft. The SCC has the ability to power reset nodes within the cluster. The design of the CMM must take into account the need to interact with the SCC and can take advantage of the existence of this ‘‘hard core.’’ However, the SCC is not designed for high performance and must not be burdened with the routine operation of the cluster. Furthermore, the CMM must not require complex software to run on the SCC since that would increase the probability of software bugs on the SCC and thus reduce the SCC’s reliability. Thus, on rare occasions, the SCC can participate in cluster recovery action. However, this must be a measure of last resort and must be implemented with very simple software on the SCC.

As described earlier, our CMM uses centralized decision making. Furthermore, the interactions of the cluster with the SCC are through the central manager. Thus, correct continuous operation of the manager is critical. The operation of the manager cannot depend on any one node on an assumption that nodes always fail silent. Hence, the manager group is implemented using active replication — a primary replica and two backup replicas. As described later, 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[16]) to ensure that faulty nodes cannot forge messages from other nodes, even if the message is forwarded by the faulty node. Manager replicas transmit commands to the agents. Agents 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.

Agents voting on outputs produced by the manager replicas is one error detection mechanism. Another key detection mechanism is the transmission of hearbeats by the agents to the manager group and the exchange of heartbeats among the members of the manager group.

As in any fault-tolerant system, faults are either masked, as with TMR, or the errors caused by the faults must be detected and then trigger recovery and reconfiguration. Even when faults are masked, they must eventually be detected and ''fixed.'' The CMM does not deal directly with hardware faults that cause the application to behave incorrectly. Detection and recovery from such faults is implemented above the CMM, possibly using some basic services from the CMM. In addition, not all the components of the CMM are protected from faults at the same level. For example, an agent process running on one of the nodes is not critical to the operation of the entire system. Thus, while active replication is used to mask a large class of faults that may affect manager operation, less aggressive mechanisms are used for the agents. The choices made in the design of the FTCT CMM are described in detail in the following sections.

3. Active Replication for the Fault-Tolerant Central Manager

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.

With active replication, all replicas execute and generate outputs independently. In the absence of failures, they must produce the same outputs in the same order. This requires all inputs to be delivered to the replicas in the same order and processing to be deterministic [3]. Hence, a reliable group communication protocol must be used to transmit messages to the manager replicas.

FTCT uses a group communication protocol developed for Amoeba [14], with the primary manager replica serving as the sequencer [17]. This protocol was chosen due to its simplicity and efficiency [9]. Senders from outside the manager group, in this case the agents, 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.

In case the primary dies after sending a message to one backup replica but not the other, each 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 and send copies of messages that were missed by the other backups managers to each of them. The last receive sequence number (RSN) is piggybacked on heartbeat messages to garbage collect the history buffer.

```
Pseudocode for Manager X

rsn_x , rsn_y , rsn_z = the RSN of the last message delivered to X,Y,Z
Initially, rsn_y = rsn_z = NIL

Receive ErrorReportMessage from agent OR
Receive EnterSelfDiagnosisMessage from a manager
        Send CheckRSN(X,rsn_x ) to managers Y,Z
        Set timer event NoRSNCheck for Y,Z
Receive CheckRSN(I,rsn_i )
        if [Received both rsn_y and rsn_z ] then
                Dismiss timer event NoRSNCheck
                if [rsn_x > rsn_y ,rsn_z ] then
                        forward messages with RSN = (rsn_y ,rsn_x ] to Y, (rsn_z , rsn_x ] to Z
                        Take snap shot of state and calculate checksum(c) of the state
                        Send Checksum(X,c) to Y,Z
                Set timer event NoChecksum for Y,Z
Receive Checksum(I,c_i )
        If [not taken checksum yet] then
                Take snap shot of state and calculate checksum(c) of the state
                Send Checksum(X,c) to Y,Z
        If [c_i != c_x ] then
                Send ProposeKill(I) to other manager J
                Set timer event NoProposeKillAck(J)
        If [c_i == c_x for both Y,Z] then
                Nothing wrong with states, continue operating
        If [Second Checksum received] then
                dismiss NoChecksum
Receive ProposeKill(I)
        If [agree manager I is faulty] then
                send ProposeKillAck(agree) back
        else
                send ProposeKillAck(disagree) back
Receive ProposeKillAck(response)
        if [response == agree] then
                kill manager
        else
                continue operating
        Dismiss NoProposeKillAck
Timer event: NoRSNCheck(I), NoChecksum(I), NoProposeKillAck(I) expired
        send ProposeKill(I) to manager J
        Set timer event NoProposeKillAck(J)
```

Figure 2: The Manager Self Diagnosis Procedure

4. Manager Self-Diagnosis

With active triplication, if one replica is faulty the remaining working replicas still form a majority and the group can continue to function correctly. However, the faulty replica must still be identified and repaired or replaced. Otherwise, a future fault in some other component of the system (such as another replica) may combine with the faulty replica and drive the system into a state from which it cannot recover.

Faulty manager replicas can be detected by comparing all the replica outputs. With the FTCT CMM, this is done by the agents. However, agents may be faulty and thus cannot be trusted to correctly identify faulty manager replicas. Instead, an agent that detects an inconsistency reports it by sending point-to-point messages to all the members of the group. This causes the manager group to initiate a procedure to diagnose itself.

The manager self-diagnosis procedure is shown in Figure 2. When a manager receives a self-diagnosis request from an agent it begins by finding out which manager has processed the highest sequence number. If a manager has not yet received messages up to the highest sequence number it will be sent the messages by the manager who is ahead. Each manager then processes messages up to that sequence number, thus each working manager should have the same state at this point. Then, a checksum is taken of the state and compared by each manager. If two managers agree that a third manager's state is different, it is removed and another manager is created.

5. Fault Detection

As mentioned earlier, the long-term survival of the system requires faults to be detected and repaired. FTCT employs three key mechanisms for fault detection: inconsistent outputs from the manager replicas, missing heartbeats, and identification of corrupt, forged, or missing messages. The first mechanism is described in previous sections. Members of the manager group send periodic heartbeats to each other and agents send periodic heartbeats to the manager group. Missing heartbeats indicate a *likely* faults.

Coding is used to detect message corruption. In addition, as mentioned earlier, all the messages transmitted by members of the CMM are authenticated. The use of authenticated messages is critical to detecting any attempt by the primary manager replica to modify messages to the manager group. All messages to the manager group are acknowledged *independently* by the replicas. Hence, if the primary manager replica modifies, discards or reorders messages before sending them to the backup manager replicas, the agents will detect the problem as inconsistent or missing acknowledgements from members of the manager group.

Exchange of heartbeats is a key mechanism for fault detection. The design of the heartbeat mechanism is driven by two key considerations: minimizing false alarms and minimizing the overhead of heartbeats. The frequency of the heartbeats is determined by a tradeoff between the ability to tolerate faults with minimum disruption and the overhead during normal operation. Frequent heartbeats reduce fault detection latency and thus reduce possible disruption/delay that may be caused by a fault. However, the overhead of heartbeats during normal operation increases with increasing heartbeat frequency.

The frequency of heartbeats is determined based on the experimental overhead measurements (Section 9) and the needs of the application. Except for heartbeat messages, all other messages transmitted among the CMM components use a reliable communication with explicit acknowledgements. In order to minimize the overhead of heartbeats, they are transmitted without any delivery guarantees. This can clearly lead to false alarms. This possibility is dealt with by requiring explicit ‘‘probing’’ to be initiated once a heartbeat is missed.

6. Use of the Space Craft Control Computer for Recovery

As described so far, the system is not capable of tolerating all faults even in the manager replicas. For example, if the primary manager replica forwards a message to one backup and not the other, the agents will detect inconsistent acknowledgements from the manager replicas. However, there is no way to tell whether the the primary failed to forward the message to the

backup (i.e. the primary is faulty) or the backup is faulty and simply dropped the message that was forwarded by the primary. The self-diagnosis procedure (Section 4) in this case will result in the fault-free backup being killed since the primary replica and the other backup will have consistent states. Since this situation is a result of the faulty primary replica, the problem could persist once a new backup replica is instantiated. Hence, the system could enter an endless cycle of detection of inconsistencies among the replicas, self-diagnosis, manager replica termination, and initialization of a new manager replica.

The problem described above could be avoided with a more complex protocol that may involve additional replicas and/or additional message exchanges among the replicas for each message received by the manager group [6]. However, this additional overhead for each message during normal operation need not be incurred if the problem is extremely unlikely to occur and, in the worst case, there is some other way to resolve it. For the example above, the problem will occur only if the primary will persist in dropping messages to one of the backups for multiple ‘‘reincarnations’’ of the backups but will otherwise continue to operate normally — generating heartbeats and passing the self-diagnosis procedure. Hence, this is an unlikely situation. However, if it does occur, we can take advantage of the SCC to resolve it.

As described in Section 2, the SCC can be assumed to be fault free but cannot take part in frequent or complex cluster operations. The ability of the SCC to power reset the system can be extremely useful for recovery from catastrophic failure, such as multiple nodes failing simultaneously due to a radiation burst. In order to facilitate this, the cluster manager group send periodic heartbeats to the SCC. If the SCC detects multiple missing heartbeats, the entire cluster is reset. This functionality requires a ‘‘hard core’’ cannot be replaced by more complex algorithms (e.g., in [7] hard core watchdog timers are used).

Since the ‘‘hard core’’ is needed anyway and actually exists for our application, we use it to resolve the problematic example describe above and other similar scenarios. Specifically, whenever the manager replicas enter self-diagnosis, they individually report this fact to the SCC. Since we use authenticated messages, this report cannot be forged. The SCC will reset the system if it detects that the rate of multiple manager replicas entering self-diagnosis exceeds some threshold. All the problematic (Byzantine) cases that can occur with our simple replication scheme are detected and resolved by the SCC in this way. It should be noted that manager self-diagnosis is rare and the SCC software required to perform the additional monitoring is very simple, thus meeting the limitations on the use of the SCC explained in Section 2.

7. Recovery from Agent Crashes and Hangs

If an agent crashes or hangs, the management system loses its ability to interact with the node, i.e., to schedule new tasks on the node, terminate or suspend existing tasks, etc. Using the SCC, this situation can be easily resolved. Specifically, when the manager group detects the missing heartbeats from the agent, it requests from the SCC to power reset (reboot) the node. While this does make the node usable again, all application processes running on the node at the time are lost. The purpose of the mechanism described in this section is to reduce the probability of losing the application processes simply because the agent process fails.

In order to avoid losing all running application processes when the agent process crashes, a mechanism is needed for a new agent process to take over. This can be done using active

replication or even a warm spare scheme. Active replication is not used due to the associated performance overhead and complexity. As discussed earlier (Section 2), such an aggressive mechanism is not justified for the agent. A warm space scheme, involves significant storage overhead — the agent process is a complex program (with associate libraries — over 15,000 lines of C++ code) that occupies 4MB of memory (Table 1).

The scheme used in FTCT achieves the benefits of a warm spare with far less overhead and higher reliability. The scheme is based on a simple *agent keeper* process running on each node whose sole function is to monitor the agent process and initiate a replacement when failure is detected. The agent keeper process is simpler (270 lines of C code) and smaller (Table 1). Hence, it is less likely to crash than a running agent and less likely to be corrupted than a warm spare agent.

	Agent (KB)	**Agent Keeper** (KB)
Binary Size:	1200	230
Runtime Size:	4000	360

Table 1: Comparison in size between agent and agent keeper.

During normal operation, each agent sends to the manager group key information about the state of the node. In particular, this includes the identifiers of local processes and corresponding system-wide task IDs of all the application processes on the node. In order to minimize the complexity of the agent keeper, the agent and agent keeper communicate using shared memory. During normal operation the agent increments a variable in this shared memory that the agent keeper reads. This variable serves as a heartbeat between the two processes. If the agent fails to increment the variable then the agent keeper concludes that the agent has failed and starts a new agent. When the new agent is started, it obtains the node state information from the manager group and takes over control of the local node. It should be noted that the SCC-based reboot scheme are still used if both the agent and agent keeper fail.

8. An Out-of-Band Signaling Mechanism for Applications

Application-level fault tolerance schemes can be facilitated by a mechanism that allows the application processes to exchange messages out-of-band from the normal application MPI interprocess communication. For example, an acceptance test in an application process may fail and the desired roll-forward recovery mechanism may require informing all the processes of the task, which are running on other nodes. At this point, the processes on the other node may not be receiving messages (e.g., a process may be stuck in an infinite loop). The out-of-band communication mechanism implemented in FTCT allows a process of a task to cause an asynchronous message to be delivered to all other members of the task.

In order to implement the out-of-band signaling feature, the FTCT CMM provides application processes with the ability to ‘‘signal’’ the local agent. The agent propagates the asynchronous message by sending it to the manager group, which forwards it to the agents on all the nodes running processes of the same parallel task. Each agent at such a nodes uses UNIX signals to interrupt the application process and then deliver the message.

In order to maintain system integrity in the face of application code bugs, the FTCT CMM can run the processes of different tasks as different users. Application processes are protected from each other and CMM processes are protected from application processes. The CMM agents can obtain UNIX *root* privilege and initiate application processes under any user. The key challenge in implementing the out-of-band signaling mechanism is to provide a way for user applications to signal the agent without violating the integrity of the system. UNIX signals cannot be used since they require that the sender and receiver processes have the same used ID. This cannot be done while providing protection to the agent process from errant application processes.

In addition to the protection issue above, another difficulty that must be faced by the application-agent communication mechanism is that is must be possible to reestablish application-agent communication following the agent recovery procedure described in the previous section. The FTCT CMM solves this problem using UNIX named FIFOs. When the agent forks a process, it creates a named FIFO with a name based on the process ID of the application process. The agent and application process can communicate through these FIFOs. Following agent recovery (Section 7), the recovered agent process obtains the application process IDs from the manager group and is able to reopen the named FIFOs to those processes. The agent process includes a special thread whose sole function is to monitor the state of the FIFOs using *select*.

In our implementation (see Section 9), The latency of an asynchronous message originating from an application process until it reaches another application process is 2.5-4ms. This involves writing data through one fifo, a message being sent to the primary manager, the message being forwarded to the backups, an agent receiving two consistent copies, and finally writing the data into the fifo and signaling the user application.

9. Experiments and Results

The FTCT currently consists of eight PCs, each with dual 350 MHz Pentium-II processors, interconnected by a high-speed Myrinet [5] LAN. For these experiments, the nodes operating system is Solaris x86. Interprocessor communication for the CMM is implemented on top of a portable *communication layer* that is implemented on top of Myricom's GM library. Altogether, the FTCT CMM currently consists of over 30,000 lines of C++ code — the communication layer, the agents, the managers, the CMM reliable message package, and the interface between MPI-CH and the communication layer. The system executes MPI applications and implements the mechanisms described in the previous sections.

A simple way to measure the overhead incurred by running the manager replica is to evaluate the fraction of ''processing'' that becomes unavailable to the application when it is executing on the same node. The manager replica 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 manager replicas given different event processing rate.

Much of the manager replica overhead described above is simply the overhead for the context switches and receiving a message. Figure 4 shows the overhead of processing incoming messages using Myricom's GM API. The top line shows the overhead including context switches while the bottom line excludes context switches. In general, context switches will take

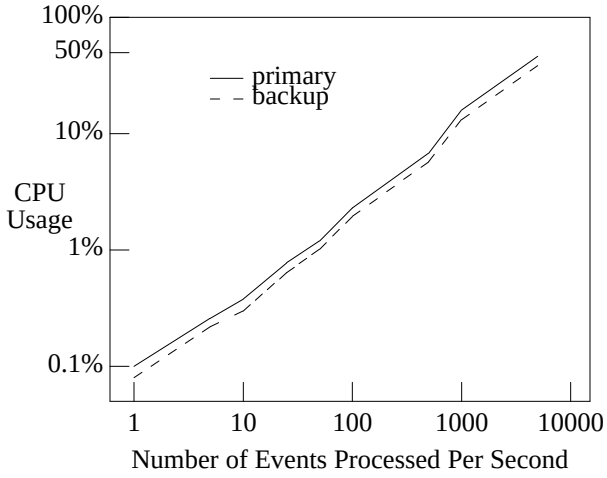

Figure 3: The overhead of a manager.

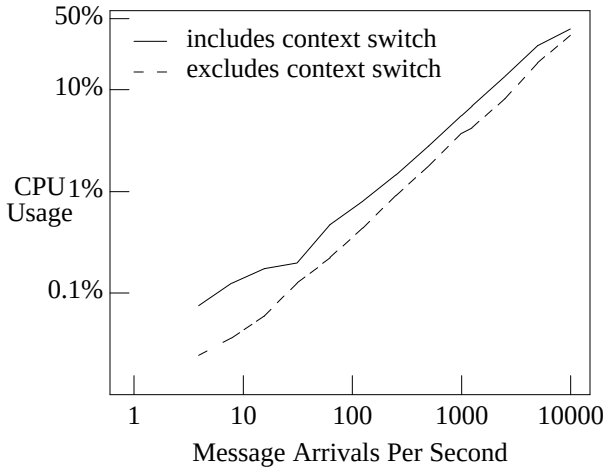

Figure 4: The overhead for receiving small messages

place for every message received since the node is almost always executing an application process with the manager replica or agent process blocked waiting for a message.

Timer events are another source of overhead for the agents and managers [17]. Figure 5 shows the overhead to handle timer events. This includes the operating system's timer management code and invoking the program's alarm handler.

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 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 2 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

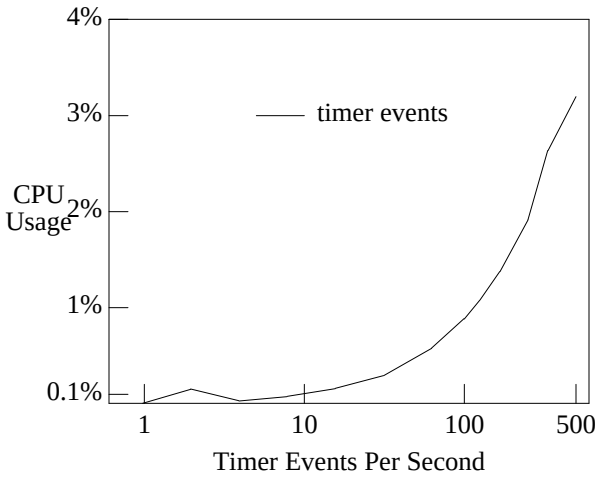

Figure 5: The overhead for handling timer events

replica, ready to accept the state from other manager replicas and become active.

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

Table 2: 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.

10. Related Work

Over the past decade, a number of resource management systems for cluster computing have been implemented [12, 21, 23, 25]. A survey of 20 research and commercial cluster management systems can be found in [1]. Excluding the management fault tolerance features, the basic functionality of the FTCT is currently similar to the functionality of the GLUnix system [12]. 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 [15] 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 [4], Horus [24], Totem [19], and Transis [11]. Several systems, such as AQuA [10] and Eternal [20] provide fault tolerance for distributed CORBA applications by using replicated objects.

11. Conclusion and Future Work

We have designed and implemented fault-tolerant cluster management middleware based on an actively-replicated central manager. The system uses simplex agent processes running on each node and supports parallel applications that communicate using MPI. A combination of heartbeats, voting, and reliable communication protocols are used for fault detection. A simple ‘‘agent keeper’’ mechanism is used to reduce the potential impact of agent crashes. The system supports an out-of-band signaling mechanism for parallel application tasks as a building block for application-level fault tolerance mechanisms.

Based on this approach, the management layer can survive a much larger class of failures than most other cluster management systems implemented on COTS hardware and software. The system takes advantage of the limited availability of a ‘‘hard core’’ to achieve these fault tolerance capabilities using simple and efficient replication algorithms. The middleware developed 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 fault injection experiments, checkpointing and rollback, reliable communication for user applications, 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. 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).
2. A. Barak and O. La’adan, ‘‘The MOSIX Multicomputer Operating System for High Performance Cluster Computing,’’ *Journal of Future Generation Computer Systems* **13**(4-5), pp. 361-372 (March 1998).
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. K. P. Birman, ‘‘The Process Group Approach to Reliable Distributed Computing,’’ *Communications of the ACM* **36**(12), pp. 36-53 (December 1993).

5. 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).
6. M. Castro and B. Liskov, ‘‘Practical Byzantine Fault Tolerance,’’ *Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, pp. 173-186 (February 1999).
7. M. Castro and B. Liskov, ‘‘Proactive Recovery in a Byzantine-Fault-Tolerant System,’’ *Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, pp. 273-287 (October 2000).
8. M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, ‘‘Active Replication in Delta-4,’’ *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing Systems*, Boston, MA, pp. 28-37 (July 1992).
9. 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).
10. 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).
11. D. Dolev and D. Malki, ‘‘The Transis Approach to High Availability Cluster Communication,’’ *Communications of the ACM* **39**(4), pp. 64-70 (April 1996).
12. 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).
13. T. P. Graf, R. G. Assini, J. M. Lewis, E. J. Sharpe, J. J. Turner, and M. C. Ward, ‘‘HP Task Broker: A Tool for Distributing Computational Tasks,’’ *Hewlett-Packard Journal* **44**(4) (August, 1993).
14. 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).
15. 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).
16. L. Lamport, R. Shostak, and M. Pease, ‘‘Byzantine Generals Problem,’’ *ACM Transactions on Programming Languages and Systems* **4**(3), pp. 382-401 (July 1982).
17. M. Li, D. Goldberg, W. Tao, and Y. Tamir, ‘‘Fault-Tolerant Cluster Management for Reliable High-Performance Computing,’’ *International Conference on Parallel and Distributed Computing and Systems*, Anaheim, CA, pp. 480-485 (August 2001).
18. 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).
19. 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).
20. 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).
21. 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).
22. F. B. Schneider, ‘‘Byzantine Generals in Action: Implementing Fail-Stop Processors,’’

ACM Transactions on Computer Systems **2**(2), pp. 145-154 (May 1984).

23. Sun Microsystems, ‘‘Sun Grid Engine Software,’’ http://www.sun.com/software/gridware/.
24. 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).
25. 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).