Low-Overhead Fault-Tolerance for Java Parallel Applications on Heterogeneous Networked Computers*

Jinho Ahn, Kibom Kim, and Chongsun Hwang
Dept. of Computer Science and Engineering
Korea University
1, 5-ka Anam-dong, Seongbuk-ku, Seoul, 136-701, Korea

Abstract *In this paper, we implement a software package to support low-overhead fault-tolerance for Java parallel applications on Java Message Passing System (JMPS) using a coordinated checkpointing algorithm and an efficient recovery algorithm, which is based on causal message logging for improving asynchrony during recovery. Currently, as the technologies of processors and communication networks have rapidly been developed, large-scale distributed systems, consisting of heterogeneous networked computers, provide high-performance parallel computing environments. But as they scale up, their failure rate may also be higher. Therefore, they require the techniques to support low-overhead fault-tolerance. In this paper, the coordinated checkpointing algorithm and the efficient recovery algorithm are used for the requirement in distributed systems. Also as large-scale distributed systems consist of heterogeneous computers, the software package for providing their fault-tolerance must be platform-independent. Thus, we implement the package in Java, which is a highly portable programming language. Our performance evaluation results show that the software package implemented in this paper is performed at low-cost.*

* This work was supported by the Korea Science and Engineering Foundation under contract 981-0924-126-2

Keywords: distributed systems, fault-tolerance, Java, coordinated checkpointing, causal message logging

1 Introduction

Currently, as the technologies of processors and communication networks have rapidly been developed, large-scale distributed systems, consisting of heterogeneous networked computers, provide high-performance parallel computing environments. Therefore, long-running scientific applications on large-scale distributed systems can concurrently use all the available resources in clusters of heterogeneous computers. But the first problem is that as they scale up, their failure rate may also be higher. Therefore, to solve the problem, they require the techniques to support low-overhead fault-tolerance. The second problem is that as large-scale distributed systems consist of heterogeneous computers, software packages for message passing and fault-tolerance services in distributed systems must be platform-independent.

In this paper, to solve the first problem, the coordinated checkpointing algorithm [5] and the efficient recovery algorithm, based on causal message logging to improve asynchrony during recovery, [1] are implemented. Java provides high-portability and architecture-neutral applicat-

ion interfaces to network at language level. Therefore to solve the second problem, we implement in Java, a software package to support low-overhead fault-tolerance in Java Message Passing System (JMPS). Our performance evaluation results describe that the software package in this paper is performed at low-cost.

The rest of the paper is organized as follows. Section 2 discusses previous researches for supporting fault-tolerance in message passing systems and identifies their problems. Section 3 explains a software package to solve the problems. Section 4 discusses the performance evaluation of the software package explained in Section 3. Section 5 concludes the paper.

2 Related Work

Message passing libraries allow parallel applications to be executed on distributed systems consisting of networked computers. The two most popular high-level message-passing systems for long-running scientific applications are the PVM (Parallel Virtual Machine) from Oak Ridge National Laboratory [7] and the MPI (Message Passing Interface) from MPI Forum [6]. PVM and MPI don't include the techniques to support fault-tolerance in large-scale distributed systems. Previous fault-tolerance packages to solve the problems were implemented for PVM [3] and MPI [8] respectively. But, these packages didn't offer platform-independence in large-scale distributed systems. For example, we assume that after a process *p1* on a computer *c1* saves its checkpoint on stable storage, it fails, and then intends to recover and restart on an available computer *c2* using its previous checkpoint from stable storage. In this case, if *c1*'s architecture and operating system platform are different from those of *c2*, *p1* can not recover on *c2*'s platform from the checkpoint produced on *c1*'s platform using the previous packages to support fault-tolerance. Therefore, we implement a software package in Java to solve the above stated problems.

3 Implementation

3.1 System Architecture

The architectural view of the software system implemented in this paper is like in figure 1. Each node has an operating system and a JVM for the operating system. The following two layers in a

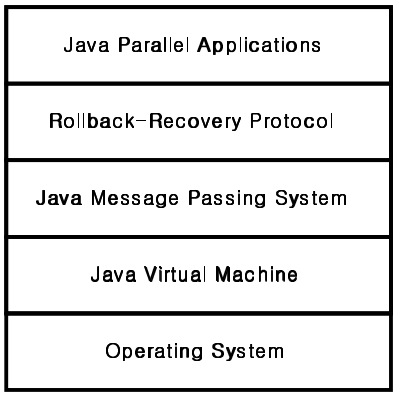

Figure 1. Layers in a node

node are implemented in this paper: the rollback-recovery protocol layer and Java Message Passing System layer. The rollback-recovery protocol layer implements the coordinated checkpointing algorithm and the efficient recovery algorithm. JMPS layer implements reliable ordered message passing between nodes. It ensures that messages between nodes are delivered in First-In-First-Out (FIFO) order, that is, that if a process p sends a message m_1 and then sends a message m_2 to a process q, q must receive m_2 after receiving m_1. Java parallel applications are executed in the highest layer. The details of the rollback-recovery protocol layer and JMPS layer will be explained in section 3.2 and 3.3.

3.2 Java Message Passing System

JMPS is a software library for supporting reliable message passing between nodes. It is implemented in pure Java and follows PVM model. A *JMPSDaemon* process is executed on

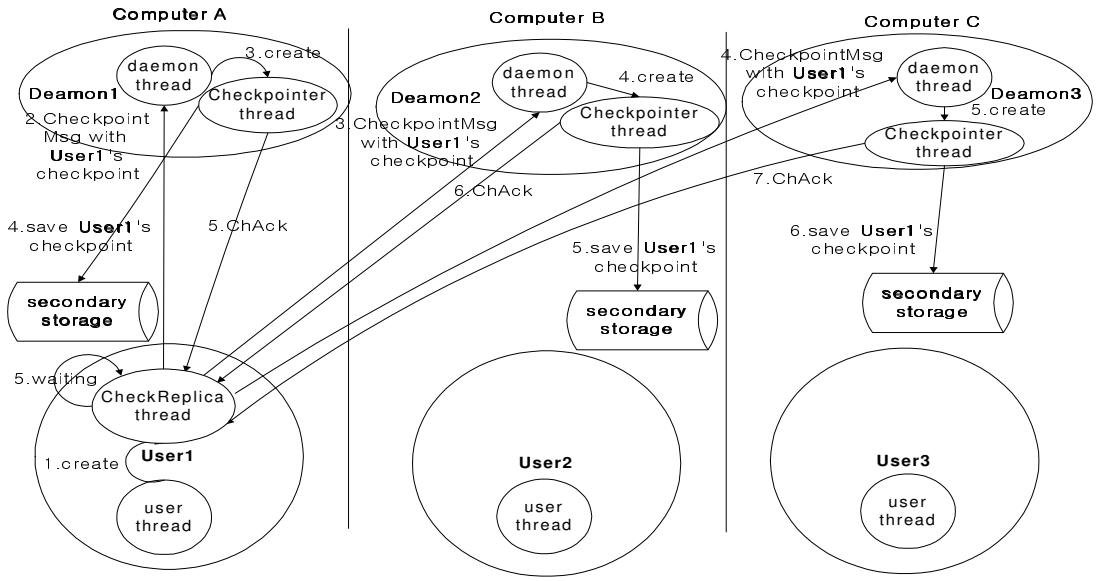

Figure 2. *User1* saves its checkpoint on computer A, B and C

each node to create and manage *JMPSProcess* processes on the node. A Java parallel application consists of one or more *JMPSProcess* processes. A *JMPSController* process adds and deletes *JMPSDaemon* processes and provides users with their information. It also provides users with the status information of *JMPSProcess* processes while they are executed. Each *JMPSProcess* process has a FIFO message queue.

3.3 Rollback-Recovery Protocol

3.3.1 Checkpointing protocol

In this paper, a coordinated checkpointing protocol [5] is implemented. The first *JMPSProcess* process to be created for a Java parallel application becomes the checkpointing initiator and coordinates other *JMPSProcess* processes to take their checkpoints to be a consistent global state [4]. And like in figure 2, each *JMPSProcess* process distributes its checkpoint on the local disk of every available node through the *JMPSDaemon* process. This approach eliminates a single point failure and a potential performance bottleneck which may occur in the approach using a centralized, large RAID disk. Also we use the technique [9] by which a *JMPSProcess* process can recover to be a consistent state even if any node fails while the process distributes its checkpoint on the local disk of every available node.

3.3.2 Message logging protocol

Causal message logging has the advantages of optimistic message logging during failure-free operation and pessimistic message logging during recovery without their disadvantages [2]: First, each process piggybacks its log of determinants in volatile memory on every message to be sent to each other in order to prevent orphan processes, which are live processes whose states are inconsistent with the recovered state of a failed process. Therefore, causal message logging has the advantage of pessimistic message logging. The determinant of a message includes the identifiers of the sender process and the receiver process, the unequivocal send sequence number assigned to the message by the sender, and the unequivocal receive sequence number assigned to it by the receiver. Second, each process logs all

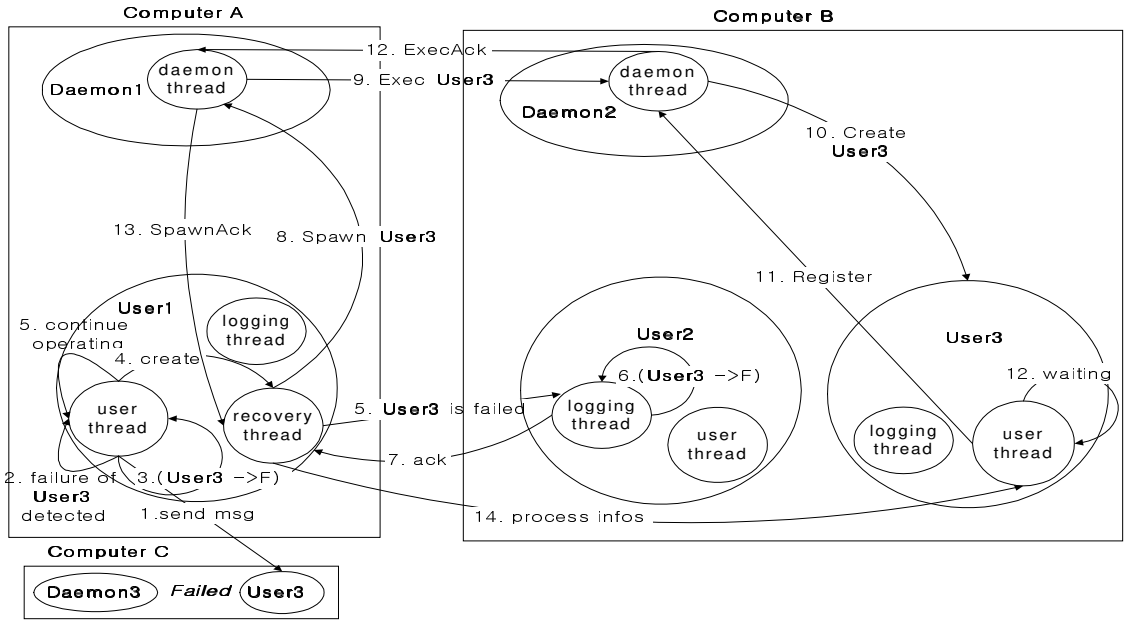

Figure 3. After *User3* has failed on a computer C, it is created on a computer B

determinants piggybacked on every received message in volatile memory, not in stable storage synchronously. Therefore causal message logging has the advantage of optimistic message logging. Causal message logging has the following property: To tolerate *f* concurrent failures, it is sufficient to keep the identifiers of *m*'s sender and receiver, the unique send sequence number assigned to *m* by the sender, and *m*'s content in volatile memory of the sender, and *m*'s determinant in volatile memory of *f+1* processes. In this paper, we implement a causal message logging protocol proposed in [1].

Every *JMPSProcess* process has the following data structures:

• ***incarnation***: It is the number incremented by one whenever a failed process starts to recover. During failure-free operation, each process piggybacks the number on a message sent to each other. In initial state of every process, it is initialized to 0.

• ***ProcessTable***: It is the vector which records the state flag(*state*) and incarnation number(*inc*) of every process in the system. Each element, consisting of the state flag and the incarnation number of a process, of the vector is initialized to *'Live'* and 0 in order.

• ***Log***: It is the set which records the determinants of the received messages maintained by a process.

3.3.3 Recovery protocol

In this paper, we implement an efficient recovery algorithm to improve asynchrony during recovery [1]. We intend to explain the recovery algorithm in details using figure 3 and 4. If a process *User3* on a computer *C* fails and a process *User1* intends to communicate *User3* like in figure 3, *User1* becomes to know *User3*'s failure. In this case, *User1* modifies *User3*'s state flag in *User1*'s *ProcessTable* to '*Fail*', create a recovery thread to allocate a process *User3* on other available computer *B*, inform the other live processes of *User3*'s failure, collect their information from them, and send the information to *User3*. And a user thread in *User1* continues executing its computation. After *User1*'s recovery thread has informed the other live processes of *User3*'s failure, it waits for their replies. After having receiving their replies, *User1*'s recovery thread

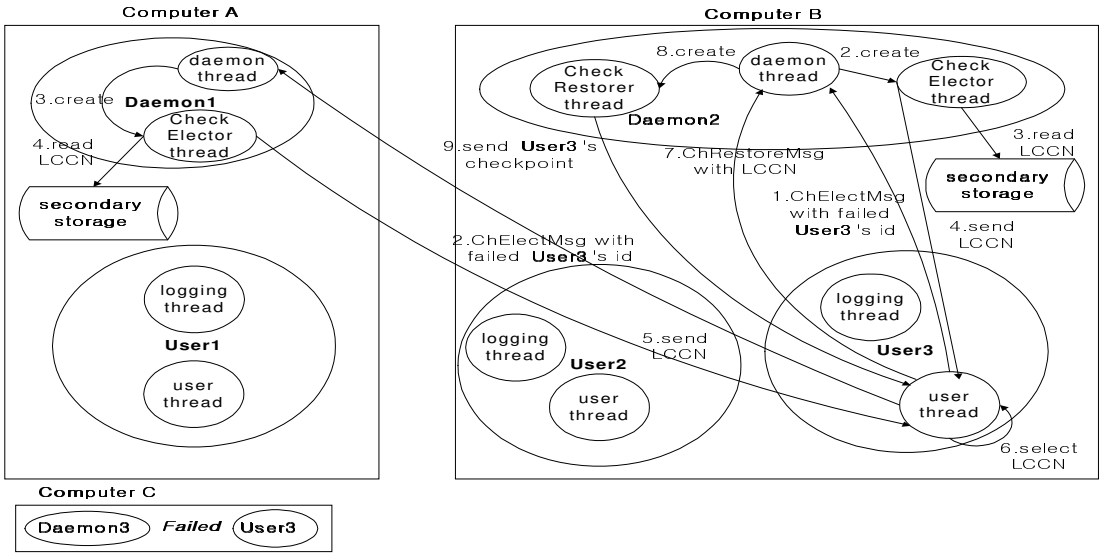

Figure 4. *User3* restores its latest checkpoint

sends *JMPSDaemon1* the request for spawning *User3*. After *JMPSDaemon1* has negotiates with *JMPSDaemon2* for which computer *User3* is allocated on, *JMPSDaemon2* creates a process for *User3*. When *JMPSDaemon2* replies to *User1*'s recovery thread through *JMPSDaemon1*, the recovery thread sends to *User3*'s user thread the information of the other processes and is destroyed. And then the user thread obtains *User3*'s latest checkpoint from *JMPSDaemon*s like in figure 4; First, the user thread requests the number of *User3*'s latest checkpoint(LCCN) to *JMPSDaemon1* and *JMPSDaemon2*. After receiving LCCNs from the *JMPSDaemon*s respectively, the user thread selects the biggest LCCN among the LCCNs and requests its latest checkpoint to the *JMPSDaemon* sending the biggest LCCN. When receiving the request from the user thread, the *JMPSDaemon* creates a *CheckRestorer* thread, which sends *User3*'s latest checkpoint to *User3*'s user thread. After restoring latest checkpoint, *User3*'s user thread sends *User1* and *User2* a message for requesting their state flags and incarnation numbers respectively. And then, logging threads of *User1* and *User2* send their state flags and incarnation numbers to *User3*'s thread. After receiving them, *User3*'s user thread sends live processes, that is, *User1* and *User2* a message for requesting the determinants related to only *User3* respectively. And then, logging threads of *User1* and *User2* send the requested determinants to *User3*'s thread. After *User3* has collected the determinants from all the live processes, it replays all the messages in a previous failure-free execution using the determinants, completes its recovery, and continues being executed as a live process. The *User3*'s recovery doesn't block the execution of *User1* and *User2*'s user thread.

4 Performance

To evaluate the performance of the software package in this paper, we used the testbed that was a set of 4 hosts connected by 10 megabit ethernet. One among the hosts was Sun Enterprise 3000 containing six UtraSPARK processors and being running Solaris 2.6. The other hosts were three PCs containing a 100 MHz Pentium processor and being running Windows 95 respectively. The version of Java used in this paper was JDK 1.2. Applications used in this experiment are as follows:

Table 1. The execution times of applications without and with checkpointing and logging

Application	No checkpointing and logging	checkpointing and logging
PQuickSort	6,598 seconds	7,198 seconds
PMatrixMult	14,789 seconds	16,090 seconds

• PQuickSort : a parallel program that sorts 2,000,000 keys stored in files.
• PMatrixMult : a parallel program that computes a matrix multiplication operation, $M_3=M_1\times M_2$ (The size of a matrix is 1024×1024).

Five processes were created for executing a PQuickSort application and each process checkpointed 20 times. Ten processes were created for executing a PMatrixMult application and each process checkpointed 45 times. Like in table 1, the difference between the execution time for a PQuickSort and a PMatrixMult with and without checkpointing and logging was just 600 and 1,301seconds, that is, the overhead to support fault-tolerance was about 8.3% and 8.1% of each total execution time respectively. Therefore the experimental results show that the software package implemented in this paper is performed at low cost in order to support fault-tolerance.

5 Conclusions

In this paper, we implemented a software package to support fault-tolerance for Java parallel applications on JMPS using a coordinated checkpointing algorithm and an efficient recovery algorithm, which is based on causal message logging for improving asynchrony during recovery. It solves the first problem that as distributed systems scale up, their failure rate may also be higher and the second problem that as distributed systems consist of heterogeneous computers, software packages for message passing and fault-tolerance services in distributed systems must be platform-independent. The performance evaluation results show that the software package implemented in this paper is performed at low cost in order to support fault-tolerance in distributed systems. We will intend to measure clearly the recovery cost of the software package, and develop more applications according to communication and stable storage access pattern and demonstrate its suitability for supporting low-overhead fault-tolerance in distributed systems.

References

[1] J. Ahn, K. Chung, K. Kim, and C. Hwang. Efficient Rollback-Recovery based on Causal Message Logging for improving Asynchrony during Recovery. *In Proceedings of High Performance Computing Symposium*, *SCS,* 305-310, April, 1999.

[2] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149-159, Feb., 1998.

[3] J.Leon, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: a portable package for distributed programming with transparent. *Technical Report CMU-CS-93-124*, Carnegie Mellon University, Feb., 1993.

[4] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *Technical Report CMU-CS-96-181*, Carnegie Mellon University, 1996.

[5] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*. 13(1):23-31, January, 1987.

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM:*

Parallel Virtual Machine. MIT Press, 1994.

[8] G. Stellner. CoCheck: Checkpointing and process migration for MPI. *In Proceedings of International Parallel Processing Symposium, IEEE,* April, 1996.

[9] F. Cristian, S. Mishra, and Young S. Hyun. Implementation and Performance of a Stable-Storage Service in Unix. *In Proceedings of the 15th Symposium on Reliable Distributed Systems, IEEE,* 1996.