# Coordinated Checkpoint from Message Payload in Pessimistic Sender-Based Message Logging

Mehdi Aminian, Mohammad k. Akbari, Bahman Javadi

Department of Computer Eng. and Information Technology
Amirkabir University of Technology, Hafez Ave., Tehran, Iran
{maminian, akbari, javadi@ce.aut.ac.ir}

## Abstract

Execution of MPI applications on Clusters and Grid deployments suffers from node and network failure that motivates the use of fault tolerant MPI implementations. Two category techniques have been introduced to make these systems fault-tolerant. The first one is checkpoint-based technique and the other one is called log-based recovery protocol. Sender-based pessimistic logging which falls in the second category is harnessing from huge amount of messages payloads which must be kept in volatile memory. In this paper we present a Coordinated Checkpoint from Message Payload (CCMP) to reduce the aforementioned overhead. The proposed method was examined by MPICH-V2, a public domain platform implementing pessimistic logging with uncoordinated checkpoint. Experimental results demonstrated the reduction of run-time for NPB benchmarks in both fault-free and faulty environments.

## 1. Introduction

Recently, as the technologies of processors and networks have rapidly been developed, message passing systems consisting of networked computers can provide supercomputer like performance parallel and distributed computing environments. However, as the systems scale up, their failure probability may also be higher.

Especially, if long running applications are executed on the systems, the failure probability becomes significant. Thus, the systems require techniques for supporting fault-tolerance.

Checkpointing and message logging are well-known techniques to build fault-tolerant systems [1]. For consistent recovery from a failure, the checkpointing technique saves the intermediate states of the application into the stable storage that survives the failure, and the message logging technique saves the messages each process has received into the stable storage. After a failure occurs, a process can restore the checkpointed state and regenerate the same computational states with the logged messages. Employing the message logging with periodic checkpointing in distributed systems, the relatively high coordination overhead of the methods relying only on checkpointing can be overcome. Message logging protocols are classified into three categories: pessimistic, optimistic and causal. Although pessimistic method results in high failure-free overhead compared with other approaches because it logs all the messages received before it sends a message but pessimistic message logging approach [9] simplifies recovery procedure of each process in contrast with others.

All the log-based recovery protocols need to store the payloads of the messages exchanged between distributed processes for the recovery purposes. The sender-based method is a low-overhead approach in which each sender saves the payloads of the messages in it's volatile memory. But storing these data in the memory may cause a kind of congestion and low-performance when using uncoordinated checkpoint in message logging techniques.

In this paper we present a new method to decrease the overhead caused by sender-based approach in pessimistic logging. Our method performs a coordinated checkpoint from the payloads of messages which are stored in volatile memory. At the same time the payloads of messages get flushed to the stable storage that causes a reduction in the execution time. In the rest of this paper, in Section 2, we will look at some essential concepts of message logging. Next in Section 3, we introduce our method and finally in Section 4, we demonstrate the performance improvements of MPI programs in faulty and fault-free environments. Finally, in Section 5 we conclude this study.

## 2. Background

The important part of a message logging protocol is to keep the payload of the messages somewhere in the system to have them available at the recovery time. Storing these messages can take place in either receiver or sender side, in the volatile memory or stable storage. In this section we briefly describe various log-based recovery protocols and two important techniques of storing message payloads.

### 2.1. Log-Based Rollback-Recovery Protocols

As opposed to checkpoint-based rollback recovery, log-based rollback-recovery makes explicit use of the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event [4]. Such an event can be the receipt of a message from another process or an event internal to the process. Sending a message, however, is not a nondeterministic event.

Log-based rollback-recovery assumes that all nondeterministic events can be identified and their corresponding determinants can be logged to stable storage [9]. During failure-free operation, each process logs the determinants of all the nondeterministic events that it observes onto stable storage. Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery. After a failure occurs, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding nondeterministic events precisely as they occurred during the pre-failure execution. Because execution within each deterministic interval depends only on the sequence of nondeterministic events that preceded the interval's beginning, the pre failure execution of a failed process can be reconstructed during recovery up to the first nondeterministic event whose determinant is not logged.

Log-based rollback-recovery protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process, that is, a process whose state depends on a nondeterministic event that cannot be reproduced during recovery. The way in which a specific protocol implements this condition affects the protocol's failure-free performance overhead, latency of output commit, and simplicity of recovery and garbage collection, as well as its potential for rolling back correct processes. There are three flavors of these protocols:

- Pessimistic log-based rollback-recovery protocols guarantee that orphans are never created due to a failure. These protocols simplify recovery, garbage collection and output commit, at the expense of higher failure-free performance overhead.
- Optimistic log-based rollback-recovery protocols reduce the failure-free performance overhead, but allow orphans to be created due to failures. The possibility of having orphans complicates recovery, garbage collection and output commit.
- Causal log-based rollback-recovery protocols attempt to combine the advantages of low performance overhead and fast output commit, but they may require complex recovery and garbage collection.

### 2.2. Receiver-Based Message Logging

With Receiver-Based Message Logging (RBML) [5], the processes participating in a distributed computation log on stable storage the messages that they receive during failure-free operation.

During recovery from a failure, a process restarts from a previous checkpoint and replays the messages in the log to restore the execution to a state that occurred before the failure. As with all message logging protocols, process execution must be deterministic in order for message replay to restore a process to the same state as before the failure. Several techniques exist for recovery, all based on computing the maximum recoverable state using the checkpoints and message logs available on stable storage [2, 8].

### 2.3. Sender-Based Message Logging

The Sender-Based Message Logging (SBML) [6] protocol keeps the contents of the message in the volatile memory of sender. The receiver sends the dependency information to stable storage. In recovery time after the process born again from its recent checkpoint, it wants from reliable media to send it the determinants and according to them, it requests the other processes to resend the payload of desired messages.

## 3. Motivation and Design

Sender-based logging is considered more efficient than receiver-based logging because the copying can take place after sending the message over the network [9]. Additionally, the system may combine the logging of messages with the implementation of the communication protocol and share the message log with the transmission buffers. This scheme avoids the extra copying of the message. Logging at the receiver is more expensive because it is in the critical path of the communication protocol.

Although SBML is the best method in storing the messages in log-based recovery protocols, it suffers from some considerable limitations. Since the payloads are stored in the volatile memory, so this method can tolerate only one fault at a time [2], because, occurring more than one fault would clear the payloads which are needy for troubled processes. Another problem with this protocol is the amount of space consumed to keep these data in the volatile memory.

In pessimistic sender-based message logging protocol the above-mentioned flaws bothers more [7]. In this protocol, a crashed process resumes its activity from checkpoint file that was taken by an uncoordinated checkpoint protocol. In addition to checkpointing the context of the running process in an uncoordinated checkpoint protocol, the whole message payloads must be stored because this information would be useful for other crashed processes to recover from failure. But these savings have their price, imposing a high overhead to run-time. Also occurring simultaneous faults in this environment may cause too much recovery time. Figure 1 depicts usage of uncoordinated checkpoint protocol in pessimistic sender-based logging. In this figure, each checkpoint file contains the entire process information plus message payloads.

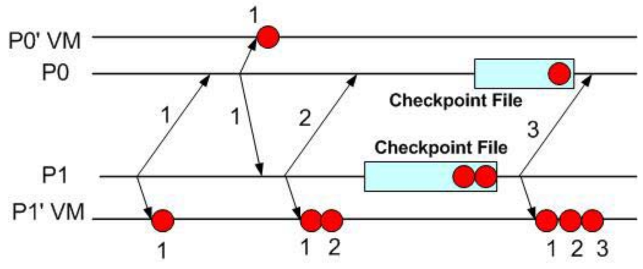


Figure 1- SBML with Uncoordinated Checkpoint

To alleviate these problems in the pessimistic sender-based protocol, we proposed an extension to sender-based message logging in which we use a coordinated checkpoint protocol to flush the payloads of all processes to the stable storage. In the time interval of two consecutive checkpoints the messages are kept in the sender's volatile memory based on traditional sender-based method. By this, the Coordinated Checkpoint from Message Payloads (CCMP) can release the uncoordinated checkpoint server from payloads savings which results in run-time reduction. In the meantime, flushing payloads to checkpoint files also releases a huge amount of volatile memory occupied by traditional sender-based message logging. Besides, simultaneous faults can be recovered faster, applying the CCMP protocol.

### 3.1. The CCMP protocol

In this protocol, each running process sends the whole messages stored in volatile memory between two consecutive checkpoints to the stable storage and flushes the memory too. Checkpointing from payloads is triggered by a checkpoint request from a scheduler. This scheduler sends the request to all processes in the same time intervals to perform the coordinated checkpoint. Figure 2 demonstrates the CCMP protocol in which the Message Payload Checkpoint (MPC) files contain the messages stored in the processes' Volatile Memory (VM). These checkpoint files are gathered by a checkpoint server which is supposed to be run on reliable media.

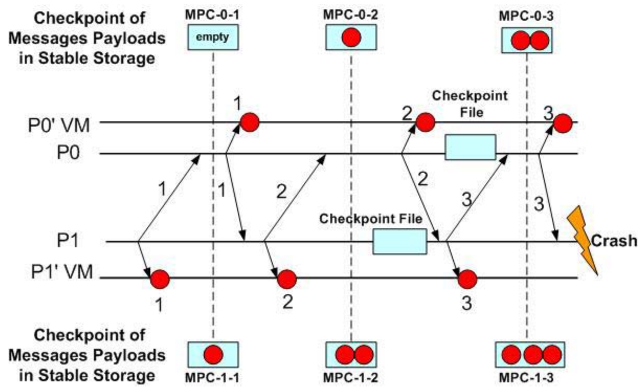

Checkpoint of Messages Payloads in Stable Storage
MPC-0-1
empty
MPC-0-2
MPC-0-3
Checkpoint File
P0' VM
P0
P1
P1' VM
Checkpoint File
Crash
Checkpoint of Messages Payloads in Stable Storage
MPC-1-1
MPC-1-2
MPC-1-3

Figure 2- CCMP Protocol Description

Now, if a crash occurs, the crashed node tries to recover by its last checkpoint file and then requests the dependency information from the stable storage. According to the dependency information, other processes would provide the crashed node with the desired messages as it can be seen in Figure 3. In this phase all other processes would resend the needy messages from their volatile memory, if exists, or from their MPC files if the messages got flushed to the checkpoint server. To avoid garbage messages to be fetched from MPC files, we must specify the latest complete uncoordinated checkpoint from the process context. It means when a process finishes an uncoordinated checkpoint, it notifies all other processes. So other processes will know which MPC files are needed.

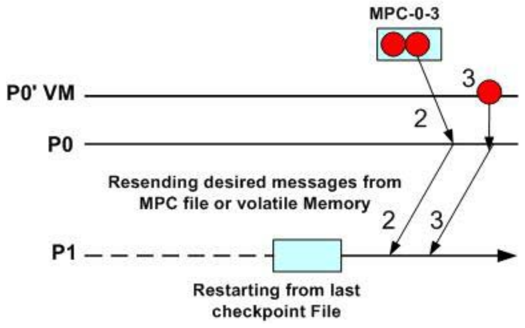

MPC-0-3
P0' VM
P0
P1
3
2
Resending desired messages from MPC file or volatile Memory
2
3
Restarting from last checkpoint File

Figure 3- Re-execution Phase in CCMP

## 4. Performance Evaluation

To test our proposed method (CCMP) some standard benchmarks were run on a cluster platform containing 32 processors under Linux 2.4.29 and using MPICH-V2 tool from LRI [10]. Each node is equipped with a Pentium III processor and 256 MB RAM, including 20 GB hard disk.

The cluster has been used in dedicated mode to ensure a fair comparison between different implementations.

## 4.1. Overview of the MPICH-V2 Architecture

MPICH-V2 implements the pessimistic sender-based protocol on top of MPICH 1.2.5, using a dispatcher, a checkpoint scheduler, some event loggers, checkpoint servers, computing nodes and their communication daemons. Figure 4 presents a typical setup of a running MPICH-V2 system, where the dispatcher, the event logger and the checkpoint scheduler seat on the same computer.

The sender based pessimistic message logging protocol of MPICH-V2 assumes that the logging of messages is split in two parts. One part uses a sender based logging method storing the messages payload on a non reliable media. The other part (the event logger) is used to store dependency information associated to these messages and must be run on a reliable system.

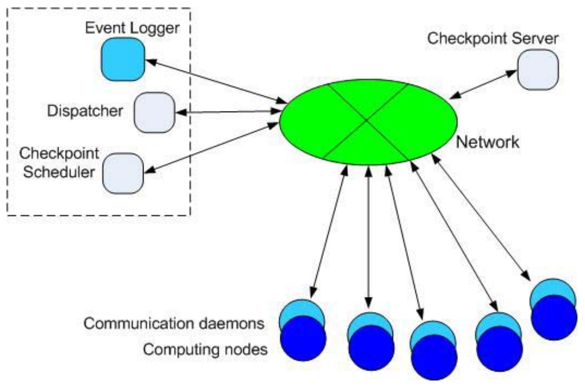


Figure 4- MPICH-V2 Architecture [10]

Each process increments a local logical clock when it sends or receives a message. The message payload logging system is integrated into the communication daemon located on the computing node. Every time a message is sent to a computing node, it is stored locally in a list for further usages (sender based). Moreover the value of the sender logical clock is stored with the message copy.

The event logger is a repository executed on a reliable component of the system. It stores and delivers dependency information about messages exchanged by the computing nodes. The dependency information is composed of four fields associated to every received message: (sender’s identity; sender’s logical clock at emission; receiver’s logical clock at delivery; number of probes since last delivery).

This information is collected during receptions of messages and sent synchronously to the event logger. However, this information must be sent and acknowledged by the event logger before the node can modify the state of another MPI process by performing a send action. In order to implement this, the communication daemon does not send messages before the event logger has acknowledged the reception of the preceding reception events.

We have modified the MPICH-V2 tool by adding one checkpoint server and one checkpoint scheduler to perform the coordinated checkpoint from message payloads. As it can be seen in Figure 5, these two processes communicate with the communication daemon. It should be noted that all the message payloads of two consecutive checkpoints are stored on the communication daemon.

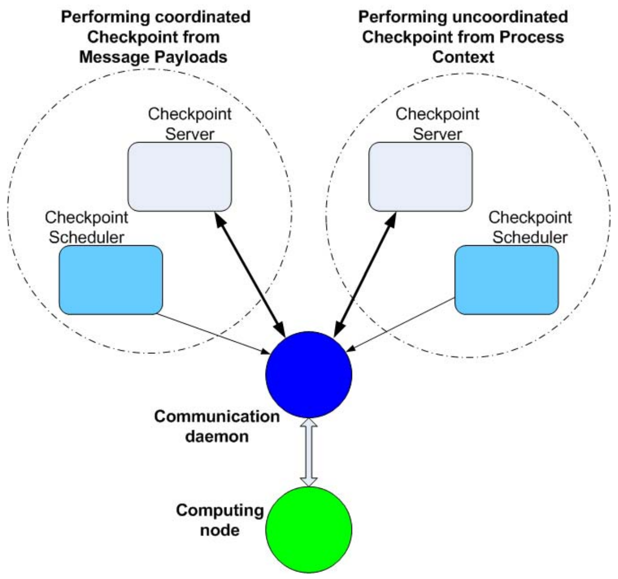

Performing coordinated Checkpoint from Message Payloads
Performing uncoordinated Checkpoint from Process Context
Checkpoint Server
Checkpoint Scheduler
Checkpoint Server
Checkpoint Scheduler
Communication daemon
Computing node

Figure 5- Implementation of CCMP Protocol in MPICH-V2

## 4.2. Fault-free Execution

In order to examine the performance of CCMP protocol on a wide set of well established and optimized MPI programs, NAS Parallel Benchmark ([3]) NPB 2.3 was used. In this regard, we ran each benchmark in two different environments. First environment was the sender-based pessimistic logging with uncoordinated checkpoint (which MPICH-V2 implements it) and the second environment was CCMP protocol in pessimistic message logging with uncoordinated checkpoint.

The results of our benchmarking are illustrated in Figure 6. As it can be seen, this figure reveals that the run-time of CCMP method for small benchmarks (i.e., IS, CG, MG, and EP) is more than the traditional sender-based method. But for large benchmarks the CCMP protocol can obtain better result to reduce the execution time and this is due to activation of uncoordinated checkpoint. Therefore, applying CCMP protocol, on the contrary of traditional sender-based method, we are not required to save the messages along with process context. This means that we have a big saving in the run-time, and the improvement is something between 5% and 10%.

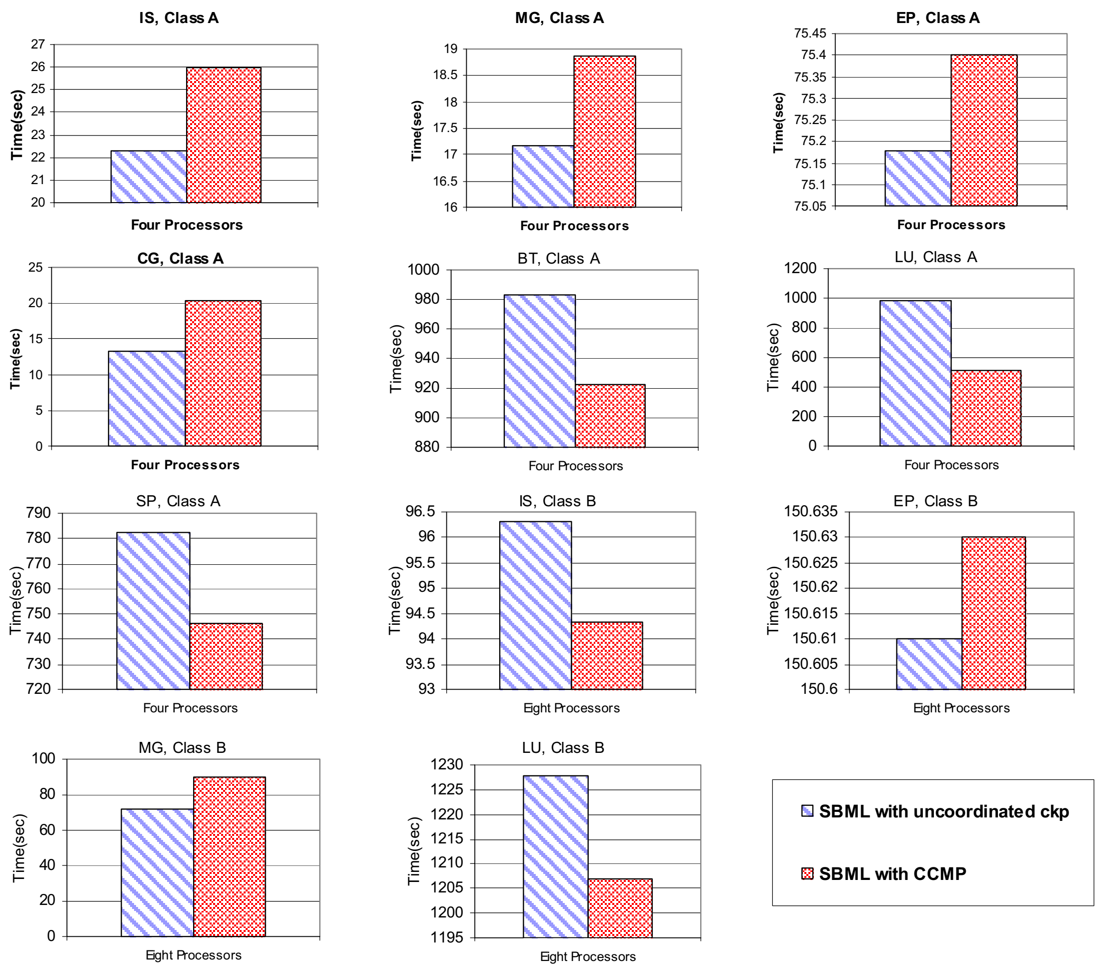


Figure 6- NPB Benchmark Results

### 4.3. Faulty Execution

The next evaluation is measuring the performance degradation of the BT and SP benchmarks when fault occurs during the execution. Figure 7 presents the execution time of BT and SP for the class A dataset size using 4 computing nodes. The test is done for the both situation, the CCMP and the traditional sender-based methods. We simulate faults by sending a termination signal to a randomly selected MPI process. The execution resumes immediately from the checkpoint file provided by the uncoordinated checkpoint server. If no checkpoint file is available, the MPI process restarts the execution from the beginning. In the traditional sender-based environment each crashed process must fetch its checkpointed messages payloads from the uncoordinated checkpoint server after it's born from the last checkpoint and other processes resend their previous messages from the volatile memory. But in the CCMP method the crashed process only requests the others to resend the messages saved in the MPC files or volatile memory (if exists). It should be noted that on the contrary of previous method we do not need to fetch any messages to the volatile memory.

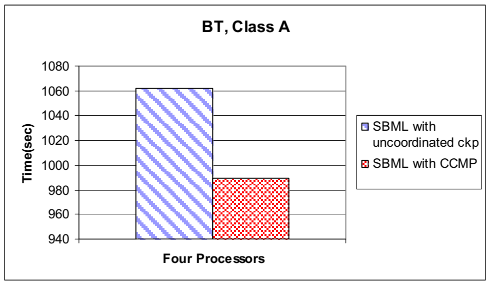
However, as illustrated in the Figure 7, the average time of recovery is much better in our approach and can recover the crashed process faster.

## 5. Conclusions

We proposed a coordinated checkpoint method from message payloads for the pessimistic logging. In our method we tried to take coordinated checkpoint from the stored messages payloads in the volatile memory and flush them to stable storage.

The experimental results of NAS benchmarks on the MPICH-V2 platform showed that CCMP protocol can reduce the run-time in contrast with sender-based pessimistic logging in fault-free and faulty modes. This improvement is due to the time consumption of the uncoordinated checkpoint in traditional sender-based method to save the messages payloads along with the context of the process.

Consistency between uncoordinated and coordinated checkpointing and applying different policies to each scheduler can be considered as a future work for this project.



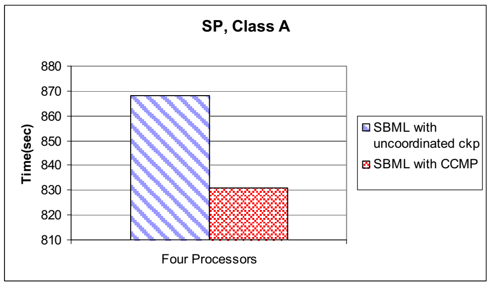


Figure 7- Re-Execution Result

## 6. Acknowledgments

We would like to thank A.Jalalzadeh and S. M. Shojaei for remarks and discussions. This Work was done in Iranian High Performance Computing Research Center (IHPCRC). The tests were done on AkuCluster system provided by IHPCRC.

## 7. References

[1] W. Gropp and E. Lusk, "Fault tolerance in MPI programs," *Special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.

[2] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," *17th international symposium on fault tolerant computing*, pp. 14-18, July 1987.

[3] D. H. Bailey, T. Harris, et al, "The NAS parallel benchmarks 2.0," Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995.

[4] E. Strom and S. Yemini, "Optimistic recovery in distributed systems**,"** *ACM Transactions on Computer Systems,* vol. 3, no. 3, pp. 204-226, August 1985.

[5] E. N. Elnozahy and W. Zwaenepoel, " On the use and implementation of message logging, " *In Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pp. 298-307, June 1994.

[6] D.B. Johnson, "Distributed System Fault Tolerance Using Message Logging and Checkpointing," PhD thesis, Rice University, December 1989.

[7] A. Bouteiller, T. Herault, et al, "MPICH-V: a Multiprotocol Fault Tolerant MPI ," *To appear in International Journal of High Performance Computing and Applications.*

[8] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," *In Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pp. 223-238, August 1989.

[9] M. Elnozahy, L. Alvisi, Y. M. Wang and D. B. Johnson, "A survey of rollback-recovery protocols in message passing systems," Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[10] A. Bouteiller, F. Cappello, et al, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," *Proceedings of High Performance Networking and Computing (SC2003)*, Phoenix USA, IEEE/ACM, November 2003.