The Performance of Independent Checkpointing in Distributed Systems

Pierre Sens

MASI Laboratory / CNRS 818, IBP, Université Paris VI
4, place Jussieu - 75252 Paris Cedex 05, France
email: sens@masi.ibp.fr

Abstract

This paper describes performance measurements of an implementation of independent checkpointing in a network of workstations. Independent checkpointing is a simple technique for providing fault tolerance in distributed system. Because processes do not coordinate during checkpointing, this technique has a low run-time overhead. To avoid the classical domino effect, our implementation relies on a message logging mechanism.

We have measured fault management overhead for different kinds of parallel applications. The costs of checkpointing are very low. However, message logging introduces a sizeable overhead. We compare these results to other works implementing different checkpointing policies, and we show that independent checkpointing is an efficient way to provide fault tolerance for long-running distributed applications composed of processes exchanging small stream of data.

1. Introduction

Checkpointing and rollback recovery are well-known techniques to provide fault tolerance in distributed systems [2, 15, 16]. With independent checkpointing, each process saves its state independently, and when a fault is detected, the execution is rolled back and resumed from earlier checkpoints. Because processes do not coordinate, this method generally provides low run-time overhead. However, since the set of checkpoints may not define a consistent global state, the failure of one process may lead to the rollback of other processes in a well-known domino effect [6].

Proc. of the 28th Hawaii International Conference on Computer on System Sciences - Sofware Technology Track, Maui, Hawaii, Januray 1995 (IEEE Computer Press)

In this paper, we present the performance of an implementation of independent checkpointing: STAR [21]. STAR automatically recovers processes allocated on faulty hosts. It uses a reliable message logging to avoid the domino effect [22, 4].

We implemented our fault-tolerant facility on top of an existing operating system. The operating system support, UNIX, has been chosen for its widely distribution over the world. However, this approach is more costly than using fault tolerance designed with operating system modifications or hardware support. Hardware support is often used for critical fault tolerant applications but it is specialized in given classes of applications and is expensive.

We have evaluated the overhead of fault-tolerant management. This overhead includes both the cost of saving the checkpoint in stable storage and the cost of message logging. Stable storage is provided by a replicated file manager. A set of servers implementes this reliable storage. The cost of checkpointing therefore depends of accessing the file servers. The most important factors affecting the checkpointing performance were the amount of data saved with each checkpoint on stable storage, and the latency in writing data. To reduce the cost of checkpointing, two complementary techniques can be applied: incremental methods, and the possibility for a process to continue executing while its checkpoint is been written to stable storage [9].

This paper is organized as follows. Section 2 describes system and application models; Section 3 presents our checkpointing scheme; Section 4 describes the process recovery protocol and addresses the problem of failure detection; Section 5 presents our implementation of the stable storage using replicated files; Section 6 reports and analyses our performance measurements, and outlines some other similar works. We conclude in Section 7.

2. Models

2.1. Execution model

An application program is a dynamic set of communicating processes. A process may use any resources of the network (CPU and files). The only way to exchange information between processes is through message passing. We make further assumptions that processes involved in the parallel computation are *deterministic*. The state of a process is determined by its starting state and by the sequence of messages it has received [Chandy 85]. This assumption is met by many applications, but excludes all programs using the local time.

2.2. Environment model

Our implementation runs on top of Unix, on a set of workstations linked by a local area network (Ethernet). In such environment the failure probability is low. Clark and McMillin measured the average crash time on a local area network to be once every 2.7 days [7]. Applications concerned by a fault-tolerant management are long-running such as high number factoring, VLSI applications, images processing, or gaussian elimination. Such applications may be executing for hours, days or weeks. In that case, the failure probability becomes significant, and the need for reliability is an important concern.

2.3. Failure model

We make the following assumptions about the failure model:

- The system is composed of fail-silent processors [19], where the failed node simply stops and all the processes on the node die. Viewed from the communication network a faulty processor remains silent and cannot receive or send message. Valid processors are not automatically informed of the failure.
- Only host failures are considered. A host is seen failed if it is not accessible. Network partitioning is not considered.
- Failures are *uncommon events* while very short recovery delays are not required. Thus, we favour solutions with a low overhead in normal functioning.

2.4. Basic architecture

The user's applications rely on a *fault-tolerant software layer* providing accesses to all external components (processes and files). This layer also provides *global naming space* for processes. Thus, the process name is location independent and a process can transparently migrate.

Our system has been developed on top of SunOS 4.1.3 (UNIX BSD) system. It consists of a set of servers and provides a client library to user's processes.

3. Checkpointing

3.1. Checkpointing a single process

The checkpoint of a single process is a snapshot of the process's address space at a given time. For reliability reasons, each checkpoint is saved on stable storage. To reduce the cost of checkpointing, two complementary techniques can be applied: incremental methods, and the possibility for a process to continue executing while its checkpoint is written [9].

Incremental methods reduce the amount of data that must be written. Only the pages of the address space that have been modified since the last checkpoint are written to stable storage. The most efficient way to implement incremental checkpointing is to use internal operating system mechanisms. In that case, the set of modified pages is determined using the dirty bit in each page table entry. This mechanism reduces the cost of checkpointing in a sizeable way. However, it must be integrated in the kernel; this is in contradiction with our portability requirement. We have implemented an incremental checkpointing outside the kernel. When a process does a checkpoint, we keep in local memory a copy of its address space. This copy is used at the next checkpoint to find the data that have been modified.

The second method to reduce checkpoint cost allows the process to continue executing while its checkpoint is written on stable storage. However, if the process modifies any of its pages during the checkpoint, the resulting checkpoint may not represent a real state of the process. Two techniques avoid this problem:

- The internal *copy-on-write* memory protection may be used to protect pages during the checkpoint. At the start of the checkpoint, the pages to be written

are write-protected. After writing each page to stable storage, the checkpoint manager removes the protection from the page. If a process attempts to modify a protected page, the page is copied to a newly allocated page, and the protection of the original page is removed. The newly page is not accessible by the process. It is only used by the checkpoint manager to finish the checkpoint. In the Manetho system [9], Elnozahy et al. have implemented a copy-on-write method. They obtain a significant improvement (between 4.7 times and 3.4 times faster checkpointing). However, like the incremental methods, this technique is integrated in the kernel.

- The second solution is the *pre-copying* method. At checkpoint time, pages to be written are copied to a separate area in memory and are then written from there to the stable storage without interrupting the process's execution. This method is simple and can be implemented outside the kernel.

Our checkpoint mechanism uses incremental and pre-copying techniques. To locally copy the process's address space, we use the fork UNIX system call. This routine creates a new process with the same address space as the caller. At the checkpoint time, the process calls the fork function, then the new created process performs its context backup while the caller continues executing. The newly process compares its address space with the space of the process created at the previous checkpoint time and it only saves data that have been modified. We show in Section 6 that these techniques reduce the cost of checkpointing in a sizeable way.

3.2. Checkpointing communicating processes

Numerous approaches to checkpointing and rollback recovery have been proposed in the literature for parallel systems. Checkpointing techniques can be divided into two categories: *consistent* and *independent* checkpointing.

With consistent checkpointing, processes coordinate their checkpointing actions such that the collection of checkpoints represents a consistent state of the whole system [6]. When a failure occurs, the system restarts from these checkpoints. If we compare the results of [2] and [9], the main drawback of this approach is that the messages used for synchronizing a checkpoint were an important source of overhead. Moreover, after a failure, surviving processes may have to rollback to their latest checkpoint in order to remain consistent with recovering processes. Alternatively, Koo and Toueg [16] reduce the number of processes to rollback, by analyzing the interactions between processes.

In the second approach, each process independently saves its state with no synchronization with the others. This technique is simple, but since the set of checkpoints may not define a consistent global state, the failure of one process leads to the rollback of other processes. A *reliable message logging* [14, 23] avoids this classical domino effect. Logging methods fall into two classes: pessimistic and optimistic. Pessimistic message logging suppresses additional rollback by *synchronously* logging messages [18, 4], i.e., the receiver is blocked until the message is logged on stable storage. In this way, all sent messages are logged, and a process in its recovered execution will directly access to the log to receive again messages. Optimistic message logging reduces failure-free overhead by logging recovery information asynchronously [22, 15]. Several messages can be grouped together and written to the stable storage in a single operation to reduce the logging overhead. However, processes that survive a failure may be rolled back.

We adopt *independent checkpointing with pessimistic message logging*. This logging is particularly adapted for application composed of processes exchanging small streams of data. In that case the message logging overhead becomes low. The cost of the logging is proportional to the number of messages. This method has the following advantages:

- elimination of the domino-effect,
- checkpoint operation can be performed unilaterally by each process and any checkpointing policy may be used,
- only one checkpoint is associated with each process,
- checkpoint cost is lower than in consistent checkpointing and recovery is implemented efficiently because all interprocess communications need not be replayed.

The benefits listed above are gained at the expense of the space that is required for storing the message logs and the time to log messages. The space overhead is reasonable given the large disk capacity.

To avoid the domino effect, our communication protocol relies on the confining principle: "a recovered process has no interaction with the others until it reaches the last state before the failure." All communications done between the checkpoint and fault point are locally

simulated. To respect this principle, we use the following techniques:

- Each process *reliably saves all received messages*. A recovered process refers to this backup to access to old messages. Thus, old valid senders are not concerned by the recovery of a process. All requests to receive messages are transmitted to the local fault-tolerance layer. This layer directly accesses to the backup or waits for messages according to the user's process state (recovered or not). At the process level there is no difference between a message reception from the network or from the backup.
- Because processes are deterministic, a recovered process sends again messages since its last checkpoint. A stamp on each message allows *to detect these retransmissions*. Each message has a unique stamp and is retransmitted with the same stamp in case of failure. The fault-tolerant layer detects the retransmission by comparing the stamp of the message with the stamp of the last transmitted message. This method avoids the reception of an old message from a recovered process.

Message logs are replicated on several hosts. We show in the performance section that the time of message logging linearly depends on the replication degree. For a replication degree of two, a message takes 1.5 more time to be delivered than the same message sent without backup. This is the main drawback of our system.

4. Failure recovery

The failure management requires two essential mechanims: failure detection and process recovery.

4.1. Failure detection

A straightforward method for software host crash detection is to wait for a normal host access failure. This detection method has no overhead but the failure treatment can only occur when one needs to use the faulty host. Thus, the failure recovery time can be very high because it directly depends on the network traffic. At worst, if no process communicates with a given host, this latter crash will never be detected. For this reason, such a method is not adapted to an efficient failure processing.

A second detection method is to periodically check the hosts states. The recovery is invoked as soon as a host does not respond to the checker. This technique provides a good recovery time but introduces an overhead in the network traffic depending on the check period.

We use a combination of the two previous methods. The normal traffic is used as in the first method, but when the traffic is too low between two hosts, we generate detection messages. An obvious software method is for each host to check all other ones. This solution leads to a big number of detection messages. It is not suitable for complex systems including many hosts where the network would become rapidly overcrowded by detection messages. A better solution is for one host to check the validity of only one other. For this reason, we organize hosts in a *logical ring of detection*. Each host independently scans its immediate successor in the ring. The checking process is straightforward and the cost in messages is relatively low. However, a ring reconfiguration protocol must be executed when adding or removing a host [4]. To perform an efficient reconfiguration protocol, each has a global view of the ring. Host insertion in the ring is done in three steps: broadcasting of an insertion message, updating of the global knowledge and transmission of the knowledge to the new host.

4.2. Processes recovery

Our recovery scheme is composed of three main steps. First, we identify processes that were running on faulty hosts. This step requires global knowledge of all processes locations. In STAR, each host has a local copy of all locations. The set of copies is updated each time a process is created, moved or terminated.

The second step allocates new processes on a valid host, usually the host which has detected the failure. This may overload the detecting host if a big number of processes were running on a faulty host, or if the same host detects successively several failures. The successor of STAR integrates a load balancing facility for dynamic allocation of processes when recovered [11]. This integration is described in [12].

Finally, the last step consists in restoring process context from their checkpoints. Then, executions are replayed from the checkpoint time to the crash time. During this period, communications are locally simulated (see Section 3).

5. Stable storage

The stable storage is implemented using standard UNIX files. To ensure fault-tolerant file accesses, files are duplicated on several disks belonging to different hosts. For user's processes, replicated files are necessary but not sufficient. When a process rolls back to its last checkpoint, it needs to see used files at the state of the checkpoint time. To solve this problem, we manage versions of duplicated files: each file in use has an old version (also replicated) corresponding to the last checkpoint of the process.

All accesses to the storage are achieved through a specific file manager. This manager has the following properties:

- *fault tolerance*: each file is duplicated on separate disks,
- *consistency*: file copies are kept identical,
- *version management*.

To improve the fault tolerance degree, all copies of a same file are kept identical. If there are N file copies, then N-1 simultaneous failures are tolerated. To ensure the equality of each copy, the file manager performs a reliable broadcast protocol [3]. This solution can appear greedy but failures are assumed as uncommon events, only a small number of copies is necessary (most of the time 2 copies are enough). The number of copies depends on the environment: the number of hosts and the crash probability. This number is given by the network administrator when he starts the STAR servers. An application designer may change this value according to his fault tolerance and performance needs.

The file manager also manages versions of duplicated files. Since each process needs only one checkpoint, only one old version is needed for each file in use. When a process rolls back, old versions of all files in use replace the current ones. In the current version of STAR, we consider that files are not shared. The management of shared files is a complex problem, and requires a distributed database manager. A future extension would integrate an existing and open log manager as Camelot [8], Clio [10] or KitLog [20].

The file manager is implemented as a set of file servers. A file server is associated with each copy. The file functions of the user's library coordinate accesses to file servers to keep the consistency property.

6. Performances

In this section, we present the performance of our implementation of independent checkpointing. All measurements have been done on a set of Sparc Station 1 with 24 Mb of memory connected by Ethernet. The environment has not been modified (usual demons were running). All results presented in this paper are averages over a number of trials.

6.1. Application Programs

We chose three long-running, compute-intensive applications representing different memory usage and communications patterns:

- The gauss application performs gaussian elimination with partial pivoting on a 1024 x 1024 matrix. The matrix is distributed among several processes. At each iteration of the reduction, the process which holds the pivot sends the pivot column to all other processes.
- The multiplication application multiplies two square matrixes of size 1024 x 1024. The computation is distributed among several processes. No communication is required other than reporting the final solution.
- The fft application computes the Fast Fourier Transform of 32768 data points. The problem is distributed by assigning each process an equal range of data points. Like the previous application, no communication is required other than reporting the final solution.

6.2. Applications requirements

Table 1 presents running time, communication, and memory requirements for the three applications when run without fault-tolerant management (without checkpointing and message logging). In all applications the computation has been distributed among four processes. Moreover, a specific process, the master, coordinates computing processes. Thus, each application is composed of five processes distributed on five hosts.

Gauss and matrix multiplication require a sizeable amount of data stressing the checkpoint and state restoration mechanisms. Moreover, the gauss application exhibits a large amount of communications especially stressing the message logging. The fft application is long-running and requires a medium amount of data.

Application programs	Running Time (seconds)	Per Process Memory (Kbytes)	Per Process communication (Kbytes)
gauss	344	1704	2700
matrix multiplication	723	2688	0.06
fft	1177	1200	0.06

Table 1: Application requirements

6.3. Checkpointing overhead

This section presents the running times of the applications programs when run with independent checkpointing with a 2-minutes checkpointing interval. This interval seems quite low. In practice, longer intervals should be used. In that sense, we overestimate the cost of checkpointing. The periodic checkpointing routine is implemented as the interrupt service routine for UNIX alarm(T) system call, where T is the checkpoint interval.

We evaluate the cost of the three checkpointing policies with two different replication degrees of the stable storage: one (i.e., each backup is composed of one UNIX file) and two (i.e., each backup is composed of two identical UNIX files). Checkpoints and message logs are stored on Sparc Station 10 with 32 Mb of memory.

Full checkpointing. Table 2 presents the running times for the basic checkpointing policy. All data are written in the stable storage and the process is blocked until the checkpoint is over.

Low communicating applications (matrix multiplication and fft) have a relatively low overhead. On the other hand, the intensive communicating application (gaussian elimination) has a very high overhead. This result is the direct consequence of the message logging cost.

Pre-copy checkpointing. We use pre-copy to avoid blocking the processes while the checkpoint is written on the stable storage. Table 3 presents the running time with pre-copy checkpointing.

With full checkpointing, the performance degradation is dependent of the amount of data to be saved, due to the latency in accessing file servers. For applications with a large address space, pre-copy provides an important reduction. For instance, the overhead of gauss and matrix multiplication applications respectively decreases of 18.12 % and 10.45 % with a replication of two degrees. For the application with a smaller address space such as fft, we obtain no measurable overhead reduction.

Applications programs	One replication degree		Two replication degree	
	Running Time (seconds)	Percentage of overhead	Running Time (seconds)	Percentage of overhead
gauss	445	29.36	567	64.92
matrix multiplication	750	3.78	844	16.79
fft	1223	3.94	1244	5.75

Table 2: Full checkpoint overhead

Applications programs	One replication degree		Two replication degree	
	Running Time (seconds)	Percentage of overhead	Running Time (seconds)	Percentage of overhead
gauss	427	24.32	505	46.80
matrix multiplication	758	4.96	768	6.34
fft	1220	3.74	1228	4.36

Table 3: Pre-copy checkpoint overhead

Incremental checkpointing. We see that pre-copy significantly reduces the cost of checkpointing. However, the amount of data written on stable storage is important whereas a small part of the data change between two checkpoints. The goal of incremental checkpointing is to reduce the amount of data to be written. The table 4 indicates the overhead obtained with incremental checkpointing.

For the three applications, incremental checkpointing provides a sizeable reduction of the overhead. Comparing to the pre-copy checkpointing, we obtain a reduction of the overhead from 24% to 256% with one replication degree and from 42% to 190% with two replication degree. Applications can be divided into two categories: applications with an address space that is modified with high locally (matrix multiplication and fft applications) and applications with an address space that is modified almost entirely between any two checkpoints (gauss application). For the applications in the first category, incremental checkpointing is very successful (more than 77 % of reduction for matrix multiplication and 190 % of reduction for fft with a replication of two degrees). For the applications in the last category, incremental checkpointing is less effective (about 42 % of reduction for the gauss application with a replication of two degrees).

6.4. Message logging overhead

We have evaluated the cost of message logging for the intensive communicating application. The overhead of message logging for the gauss application is about 6.83 % for a replication of one degree and 14.53 % for a replication of two degrees. The cost of message logging represents about a third of the global overhead.

More generally, we have evaluated the cost of the STAR communication protocol according to the replication degree of the log. The cost to send one message with one backup is 1.2 times slower than without backup. It is 1.5 times slower for two backups, 2 times slower for three backups, and 2.4 times slower for four backups.

6.5. Related work

A substantial body of work has been published regarding fault tolerance by means of checkpointing. The main issues that have been covered are reducing the number of messages required to synchronize a checkpoint [2, 9, 17, 24], limiting the number of hosts that have to participate in taking the checkpoint or in rolling back [1, 13, 16], or using message logging [4, 14, 23]. However, there are very few studies on performances of checkpointing.

Applications programs	One replication degree		Two replication degree	
	Running Time (seconds)	Percentage of overhead	Running Time (seconds)	Percentage of overhead
gauss	411	19.62	457	32.85
matrix multiplication	744	3.00	748	3.57
fft	1189	1.05	1194	1.50

Table 4: Incremental checkpoint overhead

Bhargava et al. [2] give some performance measurements of consistent checkpointing. In their environment, the messages needed for synchronizing a checkpoint implied an important overhead. Authors have limited their study to small size of programs (4 to 48 kilobytes).

Elnozahy et al. [9] have implemented consistent checkpointing on an Ethernet network of Sun 3/60 workstations. They measured the cost of synchronized checkpointing for different distributed applications and obtained good results. With a 2 minutes checkpoint interval, their checkpointing increased the running time of application by about 1 %. The worst overhead measured was 5.8 %. However, Manetho has been designed inside the V-System and cannot be easily distributed.

Borg et al. [4] has implemented a fault-tolerant version of UNIX based on three-way atomic message transmission: the TARGON/32 system. This system is specific to a hardware architecture. They measured the performance on only two machines. In that case, the performance turns out to be 1.6 times slower than a standard UNIX. The recovery time for a process is 5-15 seconds.

8. Conclusion

We have presented an implementation of independent checkpointing. A reliable storage of messages avoids the well-known domino effect. The reliable storage is achieved through a replicated file manager. Our system provides an efficient host crash detection with a logical structuring of host in a ring.

We have developed on a set of Sparc stations connected by Ethernet. We gave some performance measures that show the efficiency of our system and we showed that a fault tolerant system is viable for a workstation model. These measures showed that the total cost of our fault management is low for long-running applications with small message exchanges.

It appears from other works and our experience, that some optimization methods are very important: pre-copy checkpointing and incremental checkpointing [9]. Pre-copy checkpointing avoids to block the process while the checkpoint is written on stable storage. For applications with a large address space, it decreases the overhead from 18% to 10%. Incremental checkpointing reduces the amount data written on stable storage during each checkpoint. Comparing to pre-copy, it decreases the overhead from 14% to 2.8 %.

We currently develop a new version to take benefit of a load balancing manager (the GATOS System [11] developed at the MASI Lab). GATOS allows to use efficiently all the available processing power. STAR uses GATOS placement algorithms to choose the best host to restart faulty processes [5]. GATOS uses detection messages of the logical ring to exchange load information. It also takes advantage of the STAR checkpoint/rollback mechanism to migrate processes located on overloaded hosts.

9. References

[1] M. Ahamad and L. Lin. Using Checkpoints to Localize the Effects of Faults in Distributed Systems. In *Proc. of the 8th Symposium on Reliable Distributed Systems*, pp. 1-11, October 1989.

[2] B. Bhargava, S-R. Lian, and P-J. Leu. Experimental Evaluation of Concurrent Checkpointing and Rollback-Recovery Algorithms. In *Proc. of the International Conference on Data Engineering*, pp. 182-189, March 1990.

[3] K.P. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5:47-76, February 1987.

[4] A. Borg, W. Blau, W. Craetsch, F. Herrmann, and W. Oberle. Fault Tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1-24, February 1989.

[5] R. Boutaba and B. Folliot. Load Balancing in Local Area Networks. In *Proc. of the Networks'92 International Conference on Computer Networks, Architecture and Applications*, Trivandrum, India, pp. 73-89, October 1992.

[6] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, February 1985.

[7] H. Clark and B. McMillin. DAWGS - A Distributed Compute Server Utilizing Idle Workstations. Journal of Parallel and Distributed Computing, 14:175-186, February 1992.

[8] D.S. Daniels. Distributed Logging for Transaction Processing. *PhD Thesis, Technical Report CMU-CS-89-114*, Carnegie-Mellon University, Pittsburg, PA (USA), December 1988.

[9] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, October 1992.

[10] R.S. Finlayson. A Log File Service Exploiting Write-once Storage. *PhD Thesis, Technical Report STAN-CS-89-1272*, Stanford University, Stanford, CA (USA), July 1989.

[11] B. Folliot. Distributed Applications in Heterogeneous Environments. In *Proc. of the European Forum for Open Systems*, Tromsø, Norway, pp. 149-159, May 1991.

[12] B. Folliot and P. Sens. GATOSTAR: A Fault-tolerant Load Sharing Facility for Parallel Applications. In *Proc. of the First European Dependable Computing Conference*, Berlin, Germany, October 1994. Lecture Notes on Computer Science, Springer-Verlag.

[13] S. Israel and D. Morris. A Non-intrusive Checkpointing Protocol. In *Proc. of the Phoenix Conference on Communications and Computers*, pp. 413-421, 1989.

[14] D.B. Johnson and W. Zwaenepoel. Sender-based Message Logging. In *Proc. of the 7th Symposium on Fault Tolerant Computing Systems*, pp. 97-104, June 1990.

[15] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11(3):462-491, September 1990.

[16] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23-21, January 1987.

[17] T.H. Lai and T.H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25:153-158, May 1987.

[18] M.L. Powell and D.L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proc. of the 9th ACM Symposium on Operating Systems Principles*, pp. 100-109, 1983.

[19] D. Powell, G. Bonn, D. Seaton, P. Verissimo, F. Waeselynck. The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proc. of the 18th International Symposium on Fault-Tolerant Computing Systems*, Tokyo, Japan, pp. 246-251, 1988.

[20] M. Ruffin. KITLOG: a Generic Logging Service. In *Proc. of the 11h Symposium on Reliable Distributed Systems*, Houston, Texas, pp. 139-146, October 1992.

[21] P. Sens and B. Folliot. Star: A Fault Tolerant System for Distributed Applications. In *Proc of the 5th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, pp. 656-660, December 1993.

[22] A.P. Sistla and J.L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proc. of the 8th Annual ACM Symposium on Principles of Distributed Computing*, August 1989.

[23] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204-226, August 1985.

[24] Z. Tong, R.Y. Kain, and W.T. Tsai. A Lower Overhead Checkpointing and Rollback Recovery Scheme for Distributed Systems. In *Proc. of the 8th Symposium on Reliable Distributed Systems*, pp. 12-20, October 1989.