# Performance Evaluation of Fault Tolerance for Parallel Applications in Networked Environments

Pierre SENS[*] and Bertil FOLLIOT[**]
LIP6 Laboratory , University Paris VI[*], University Paris VII[**]
email: {sens,folliot}@masi.ibp.fr

## Abstract

*This paper presents the performance evaluation of a software fault manager for distributed applications. Dubbed STAR, it uses the natural redundancy existing in networks of workstations to offer a high level of fault tolerance. Fault management is transparent to the supported parallel applications. STAR is application independent, highly configurable and easily portable to UNIX-like operating systems. The current implementation is based on independent checkpointing and message logging. Measurements show the efficiency and the limits of this implementation. The challenge is to show that a software approach to fault tolerance can efficiently be implemented in a standard networked environment.*

## 1. Introduction

Few distributed computing environments offer fault management using the natural redundancy of the distributed system and requiring no specific hardware support [1, 14, 17]. STAR was developed to add to the filling of this gap and is built totally outside the operating system. It also answers to the challenge to show that software fault tolerance can be efficiently implemented in a standardized environment.

Checkpointing and rollback recovery are well-known techniques to provide fault tolerance in distributed systems [11, 12, 13]. With *coordinated checkpointing*, processes coordinate their checkpointing actions such that the collection of checkpoints represents a consistent state of the whole system [5]. When a failure occurs, the system restarts from these checkpoints. Looking at the results of [2], [7], and [13], the main drawback of this approach is that the messages used for synchronizing a checkpoint are an important source of overhead. In *independent checkpointing*, each process independently saves its state. Because processes do not synchronize themselves for checkpointing, 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 (well-known as the domino effect [5]).

In the STAR implementation, an independent checkpointing mechanism is used to recover processes [4, 16, 21]. Our recovery protocols are based on *message logging* [4, 8, 10, 19] to avoid the domino effect. In the general approach, processes log their received messages. A process may recover by restarting from its last checkpoint and then replaying from the log the sequence of messages it originally received. We also present an evaluation of *optimistic message logging* [1, 18, 20] where received messages are buffered in volatile storage and logged to stable storage asynchronously. Unlike pessimistic message logging, this approach allows a process to continue execution before the message is logged.

To improve the response time of fault-tolerant applications, STAR includes several optimizations. First, it implements non-blocking and incremental checkpointing to perform an efficient backup of process state. Secondly, we developed an optimized stable storage based on replicated file system. It appears from other works and our experience, that these optimization methods are very important [7]. These techniques lead to a drastic reduction of the overhead for classical parallel applications.

STAR was implemented on a set of Sparc stations connected by Ethernet. The results demonstrate that independent checkpointing is an efficient approach for providing fault tolerance for the chosen applications, namely long-running ones with small message exchanges. We show that a software based fault tolerant management is an interesting alternative to specialized hardware or kernel-integrated fault tolerance. Results from [13,14] as well as our own instrumentation of several parallel applications corroborate this claim.

The remainder of this paper is organized as follows. Section 2 presents the application, environment and failure models. Sections 3 and 4 describe the mechanism of failure detection and the process recovery strategy. Our implementation of the stable storage is presented in section 5. Then, Section 6 gives the performance of STAR in a real academic environment. We conclude in Section 7.

## 2. Environment

STAR manages fault of processes already allocated by an allocation manager. To provide a complete management of parallel applications STAR was integrated in the Gatos process allocation manager [9]. An application is a dynamic set of communicating processes which may use any resource of the network (mostly CPU and files). The only way to exchange information between processes is through message passing. A further assumption is made 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 [5]. This assumption is met by many applications, but excludes for example all programs relying on the values of the local time. To handle some nondeterminism, we can extend the message logging scheme by treating each nondetermisnistic function as a message, logging it and replaying it during recovery [8].

User applications rely on a *fault-tolerant software layer* providing a reliable access to all external components (processes and files). This layer allows the recovery of processes affected by a host failure in a transparent way on any remaining valid and compatible host. It provides a *global naming space* for processes and files independent of the location. The STAR communication protocol relies on this global naming space to find the location of the target process. This knowledge is updated after each process recovery.

STAR lies on top of a Unix operating system including network facilities (SunOS). It works on a set of workstations (hosts) connected by a local area network (Ethernet). We assume that the underlying transport layer provides reliable, sequenced point-to-point communication. The system is composed of fail-silent processors where a failed node simply stops and all the processes on the node die.

STAR consists of a set of servers and a client library. There are three main servers: the *recovery server* in charge of failure management, the *file server* implementing the stable storage by means of replicated files, and the *communication server* managing interactions between application processes.

Each application program must be linked to the STAR library which contains the following functions

- Checkpoint and restoration: the **`checkpoint`** function is either periodically called or explicitly indicated in the source code. When a process is restarted the **`restore`** function is automatically called.
- File access functions: these functions provide a Unix-like interface to the STAR file manager.
- Communication functions: these functions allow reliable message exchanges implementing message logging strategies.

## 3. Failure Detection

The software approach to detect a host crash is often realized by using the normal communication traffic. This method has no overhead, in terms of number of messages, but the failure processing can only occur when one needs to use the faulty host. Thus, the recovery time in case of failure can be very high. Such a method only based on normal communication traffic is not appropriate for a fault manager.

Another solution consists in periodically checking the hosts states [4]. The recovery is invoked as soon as a host does not respond to the checker. This technique allows a fast recovery, but introduces an overhead in the network traffic. This overhead is proportional to the checking rate.

STAR uses a combination of the two methods. The normal traffic is used as in the first method, but in addition, when there is no traffic during a given time slice, a specific detection message is generated. A naive implementation of this detection would be for each host to check all other active ones. This solution is not suitable for complex systems with many hosts, since the network would become rapidly overcrowded by detection messages. In order to get an efficient detection message traffic, we organize all the hosts in a *logical ring.* Periodically, each host *only* checks its immediate successor on the ring. The checking process is straightforward and the cost in messages is very low. However, to insure the coherence of the ring, a two-phase reconfiguration protocol is executed when adding or removing a host. The cost of the reconfiguration protocol is not significant since host crashes are uncommon events.

On each host, the recovery server maintains a global view of the ring. In case of failure, the predecessor of the faulty host can locally determine its new successor. Host insertion in the ring is done in three steps: broadcast of an insertion message, update of the global knowledge, and transmission of the knowledge to the new host. The new host takes place in the ring according to its own host identification. This method supports an arbitrary number of simultaneous failures

The implementation of the logical ring is as follows. Each recovery server is linked to its predecessor and successor using the TCP communication protocol. Periodically, the server checks if a normal message has been received from its successor. If no message has been received, it sends a detection message to its successor. If this sending fails, the successor is considered faulty and the server initiates the recovery step. At present, we make no attempt to detect individual process failures on a node. Future versions of the STAR software will handle also finer-grained failures.

## 4. Process Recovery

The recovery step is invoked as soon as a failure is detected. Processes affected by the failure are immediately restarted on a valid host unlike some other fault managers where processes can only be restarted after the faulty host is rebooted (as in DAWGS [6] or Arjuna [17]). The process recovery in STAR is done by (1) checkpointing process on a stable storage, (2) restarting the process on a hardware compatible valid host, and (3) redirecting communications to the new process location.

### 4.1. Checkpointing a single process

The checkpoint of a single process is a snapshot of the process address space at a given time. Each checkpoint is saved on a stable storage capable of surviving to a given number of host failures. To reduce the cost of checkpointing, STAR's checkpoint mechanism uses both incremental and non-blocking checkpointing.

The Unix `fork()` primitive provides exactly the mechanism needed to implement non-blocking checkpointing. When checkpointing, the STAR library forks a child process which performs its context backup while the parent process returns to executing the application. The fork system call creates a new process with the same address space as the caller. Many implementations of fork use a copy-on-write mechanism to optimize the copying of the parent's address space.

To perform incremental checkpointing, the new child process compares through a pipe its address space with the space of the child process created at the previous checkpoint then it saves only data that have been modified. We show in Section 5 that these two techniques considerably reduce the cost of checkpointing. However, they require a larger amount of memory and result in increased multiprogramming.

### 4.2. Recovery schemes for communicating processes

When processes exchange messages, the simple approach to recovery for independent processes is no longer adequate. In particular, attempts by individual cooperating processes to achieve backward error recovery can result in the well-known domino effect [15].

The current implementation of STAR is based on independent checkpointing with message logging. This technique is tailored to applications consisting of processes exchanging small streams of data. This method totally suppresses the domino-effect and consequently only one checkpoint is needed for each process. We have implemented pessimistic and optimistic message logging to allow application designers to choose the logging algorithm according to their application requirements.

These benefits are obtained at the expense of the space and time required for logging messages. The space overhead is reasonable given the current large disk capacities. Furthermore, at each new checkpoint all messages are deleted from the associated backup (a log is completely deleted after each checkpoint). The main drawback is the Input/Output overhead (i.e., the latency accessing the stable storage, see Section 6).

### 4.3 Communication management

The STAR 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*" and consequently avoids the domino effect. All communications done between the checkpoint and the fault point are locally simulated. Thus, any process may be independently rolled back. To comply with this principle, we use the following techniques:

- Each process *saves all input messages* (message logging, see Section 4.3). A recovered process refers to this backup to access old messages. Thus, old valid senders are not concerned by the recovery of a process. All requests to receive messages are transparently transmitted to the local fault-tolerance layer. This layer directly accesses the backup or waits for messages according to the process state (recovered or not). At the process level there is no difference between receiving a message from the network or from the backup.

- Because processes are deterministic, a recovered process sends again all messages since its last checkpoint. A timestamp on each message allows to detect these retransmissions. Each message has a unique timestamp and is retransmitted with the same timestamp in case of failure. The fault-tolerant layer detects the retransmission by comparing the timestamp of a message with the stamp of the last transmitted message to discard already received messages.

In the optimistic scheme, messages are not directly saved on the stable storage but are kept on the main memory of the sending host. Periodically (when `MaxTransit` messages have been sent), the sending host asynchronously saves all messages on the stable storage. In case of failure, messages addressed to faulty processes are found either on the stable storage or on the main memory of the sending hosts. STAR also provides a sender-based algorithm where all messages are kept in the sending queue and are never saved on stable storage.

## 5. Stable Storage

Stable storage is a key feature in a fault manager. In STAR, a reliable file manager implements stable storage. It is used for file accesses, message backups and checkpoint storage.

In STAR, each file is replicated on separate disks on different hosts. The number of replicated copies is maintained in case of failure (obviously, only if the number of remaining disks is sufficient). Because failures are uncommon events, only a small number of copies is usually necessary (usually 2 for a network of 20 involved workstations). This number is set by the network administrator or by the application designer according to the fault tolerance and performance requirements. To ensure consistency of all copies, the file manager performs a reliable broadcast protocol [3]. A file update is reliably broadcast to all managers having a copy. A read operation is locally done whenever possible.

A reliable file is composed of a set of standard UNIX files replicated on a set of disks. On each host where copies are present, a file server manages accesses to copies. When a file server host fails, the files are copied from a valid host to a new file server thus maintaining the initial replication degree.

Performance of STAR directly depends on the stable storage management. To provide an efficient replicated file access, we take advantage of the pseudo-parallelism offered by the underlying system. Any access to a remote file server is achieved by a specific process located in the client host: the file server proxy. One proxy is associated with each remote server. Local clients and proxies exchange information through a local shared segment of memory. When a client wants to send a request to N servers, it puts the request arguments in the local memory and wakes up the proxies corresponding to the remote servers. Then, proxies read and transmit the request in a pseudo-parallel way.

## 6. Performance Evaluation of STAR

Fig. 1 shows the performance of the STAR file system (SFS) according to different replication degrees for writing and reading a file of 1 Megabyte. These measures were done on a set of Sun 5 and 10 workstations with 32 Mb of memory. A replication degree of 4 means that the file is saved on 4 disks on 4 hosts. We also illustrate the performance of NFS when data are in cache or not. Naturally, NFS measurements do not depend of the replication degree. SFS read has not been optimized since it is only used when recovering. On the other hand, SFS write is especially stressed since it is used during normal running for checkpointing and message logging. We see that in every case SFS writing is very efficient compared to NFS. These good performances are essentially due to the parallelization of servers accesses.

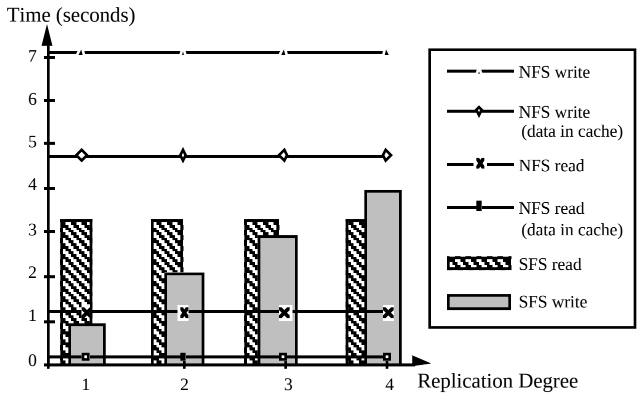

**Fig. 1: Performance of the Star File System**

Fig. 2 illustrates the cost of the optimistic message logging according to the size of the queue on the sending host. 0 means that messages are directly saved on stable storage before delivery. 500 means that messages are queued at the sending host and are asynchronously saved when the queue contains 500 messages. 0 queue size is equivalent to the pessimistic strategy. We also indicate the time to send messages with the sender-based algorithm where all messages are kept in the queue and are never saved on stable storage. These measures were done for 1024 messages of one kilobyte transmitted between two users processes. Messages are saved on two different hosts. The sender-based protocol seems more efficient but in fact it uses too much memory to be applicable in real applications.

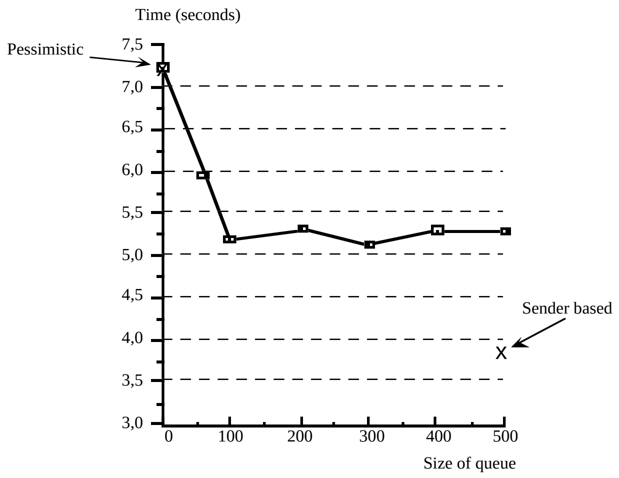

**Fig. 2: Optimistic message logging cost**

Additional experiments were run with STAR ported to a Sun IPC network with an approximate computing power of 12 VAX MIPS and 24 Mb of memory. Figures 3 and 4 present the running times for two independent checkpointing implementations: full checkpointing where all data are written in stable storage and the process is blocked until the checkpoint is over and incremental non-blocking checkpointing where application continues while the checkpoint is written on stable storage and the amount of data to be written is reduced. The cost of checkpointing was measured with different replication degrees (from 1 to 4) and with different process sizes (from 100 to 1150 kilo-bytes). Programs run with a 20 seconds checkpointing interval, a rather short interval. In practice, longer intervals should be used. In that sense, we overestimate the cost of checkpointing and we stress the checkpoint mechanism.

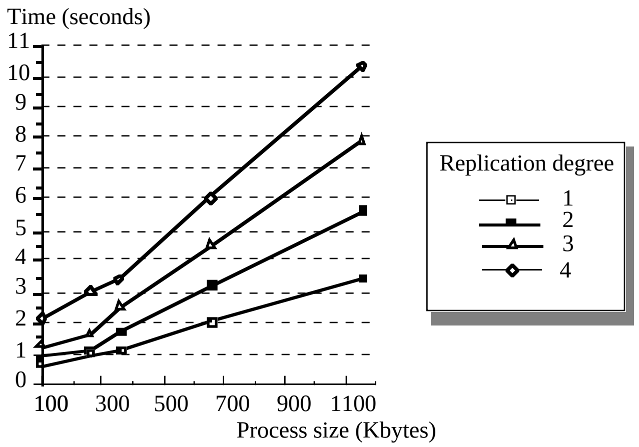

**Fig. 3: Full checkpoint cost**

Fig. 3 shows the cost of full checkpointing, where data and stack segments are entirely copied on stable storage. The process memory usage (i.e., process locality) is not taken into account. The cost linearly depends on the process context size. The time to save 1150 kilo-bytes on four replicated files takes 10.3 seconds. This is about three times slower than to write the same amount of data on a single file (3.4 seconds). The measured time can appear high compared to the performance of the STAR file server presented above. This is mainly due to the difference of machines in term of processing power. Moreover, the short checkpoint period overloads the file servers.

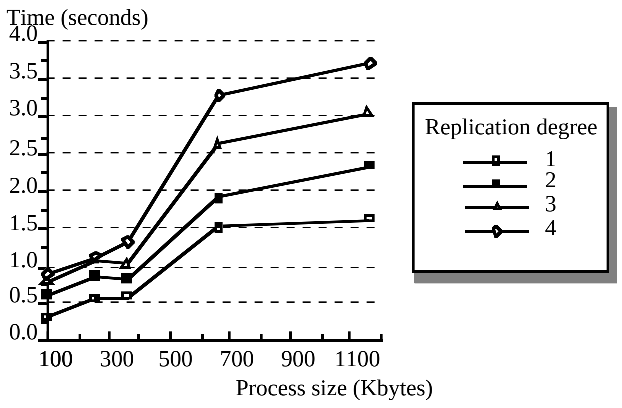

**Fig. 4: Incremental checkpoint cost**

Fig. 4 presents the cost obtained with non-blocking and incremental checkpointing. In the previous checkpointing methods, the amount of data written on stable storage was important whereas a small part of the data changes between two checkpoints. We observe a sizeable reduction of the checkpoint overhead. An incremental checkpoint is about three times faster than a full one. For the smallest program the checkpoint cost is below 1 second for all the replication degrees. Note that the curves are not linear because the time to take a checkpoint depends on the process memory usage.

The performance figures shown above are quiet good compared to that of other software fault managers. For a 2.6 megabytes program (a matrix multiplication) the mean time to perform a checkpoint is in STAR 4 times faster than `libckpt` [14]. This difference is mainly due to the use of the inefficient mprotect system call to implement incremental checkpointing. The system DAWGS [6] does checkpointing with only replication of degree one. In a network of 10 workstations with an approximate computing power of 3 VAX MIPS, the checkpoint time is about 1.84 seconds for 25 Kbytes.

Fig. 5 presents the recovery time of a process after a failure. This time includes the time to relaunch a process and to recover its state from its last checkpoint. This time is close to linear with the process size. For a 100 Kbytes process, it is 2.5 seconds, and for a 1150 Kbytes process, it is 6.8 seconds.

The restoration time can appear important compared to the checkpoint cost. In fact, the restoration step is much more complex. It must identify the process, reconfigurate the ring, create a new process, restore its context and finally update the global knowledge.

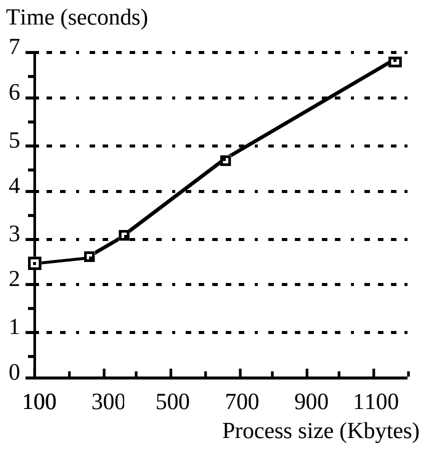

**Fig. 5 : Restore context cost**

To tally the performance of STAR under a working load 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, called matmul, 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.

Table 1 presents running time, communication, and memory requirements for the three applications when run without fault-tolerant management (without checkpointing and message logging). Gauss and matmul require a sizeable amount of data stressing the checkpoint mechanism. 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 | Running Time (seconds) | Per Process Memory (Kbytes) | Per Process communication (Kbytes) |
|---|---|---|---|
| gauss | 344 | 1704 | 2700 |
| matmul | 723 | 2688 | 0.06 |
| fft | 1177 | 1200 | 0.06 |

**Table 1: Application requirements**

Table 2 presents the running times of the applications programs when run with independent checkpointing and message logging. Applications run with a 2-minutes checkpointing interval. Checkpoints and logs are duplicated. For the three applications, incremental checkpointing provides a sizeable reduction of the overhead. Comparing to the non-blocking checkpointing, we obtain a reduction of the overhead from 42% to 190%. Applications can be divided into two categories: applications with an address space that is modified with high locality (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). For the applications in the last category, incremental checkpointing is less effective (about 42 % of reduction for the gauss application). Furthermore, the cost of message logging for the gauss application represents a half of the global overhead.

## 7. Conclusions

This paper has presented an evaluation of the STAR fault manager for distributed applications in a standard workstation environment. The current implementation is based on independent checkpointing, and avoids the domino effect by using message logging. From the basic components, other fault-tolerant techniques can be implemented according to the needs of the supported parallel applications.

We reported performance measurements of the basic software components. The results demonstrate that independent checkpointing is an efficient approach for providing fault tolerance for the specific studied applications, i.e., long-running ones with small message exchanges. We have also shown that a software based fault tolerant management is an interesting alternative to specialized hardware or kernel-integrated fault tolerance. Results from [13] as well as own instrumentation of distributed applications corroborate this claim.

## References

[1] L. Alvisi, K. Marzullo. Message Logging: Pessimistic, optimistic, and Causal. In *Proc. of the 15th International Conference on Distributed Computing System, June 1995.*

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

| | Full checkpoint | | Non-blocking checkpoint | | Incremental checkpoint | |
|---|---|---|---|---|---|---|
| | Running Time (sec.) | Percentage of overhead | Running Time (sec.) | Percentage of overhead | Running Time (sec.) | Percentage of overhead |
| gauss | 567 | 64.92 | 505 | 46.80 | 457 | 32.85 |
| matmul | 844 | 16.79 | 768 | 6.34 | 748 | 3.57 |
| fft | 1244 | 5.75 | 1228 | 4.36 | 1194 | 1.50 |

**Table 2: Parallel Applications Evaluation**

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

[7] 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.

[8] E.N. Elnozahy, W. Zwaenepoel. On the Use and Implementation of Message Logging. In *Proc. of the 24th International Symposium on Fault-Tolerant Computing Systems*, pp. 298-307, Austin, Texas (USA), June 1994.

[9] 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, K. Echtle, D. Hammer, D. Powell (Ed), Lecture Notes in Computer Science 852, pp. 581-598, 1994.

[10] 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.

[11] 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.

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

[13] G. Muller, M. Hue, N. Peyrouze. Performance of Consistent Checkpointing in a modular Operating System: results of the FTM Experiment. In *Proc. of the First European Dependable Computing Conference*, Berlin, Germany, October 1994, K. Echtle, D. Hammer, D. Powell (Ed), Lecture Notes in Computer Science 852, pp. 491-508, 1994.

[14] J.S. Plank, M. Beck, G. Kingsley, K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proc. of USENIX Winter 1995 Technical Conference* , New Orleans, Louisiana(USA), January 1995.

[15] B. Randell. Design Fault Tolerance. *The Evolution of Fault-Tolerant Computing Vol. 1*, A. Avizienis, H. Kopetz, J-C. Laprie (Ed), Springer-Verlag, pp. 251-270, 1987

[16] P. Sens. The Performance of Independent Checkpointing in Distributed Systems. In *Proc. of the 28th Hawaii International Conference on System Science*, pp. 525-533, Maui, Hawaii, January 1995.

[17] S.K. Shrivastava, D.L. McCue. Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment. *IEEE Transactions on Parallel and Distributed Systems,* pp. 421-432, April 1994.

[18] S.W. Smith, D.B. Johnson, J.D. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. In *Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing*, Pasadena, CA (USA), June 1995

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

[20] Y.M. Wang, W.F. Fuchs. Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems. In *Proc. of the IEEE 11th Symposium on Reliable Distributed Systems*, pp. 147-154, October 1992.

[21] J. Xu, R.H.B Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. In *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing,* pp. 754-761, December 1993.