Flexible Checkpointing and Efficient Rollback-Recovery for Distributed Computing *

Guy Bernard and Denis Conan

Institut National des Télécommunications
9 rue Charles Fourier 91011 EVRY Cedex France
Phone: +33–1–60 76 {45 67|44 15}
Fax: +33–1–60 76 47 80
e-mail: {bernard|conan}@int-evry.fr

Abstract

This paper addresses the problem of fault tolerant distributed computing in networks of workstations, where distributed applications consist in several processes running on several workstations in parallel, with communication links between them. We consider software fault tolerance achieved by transferring tasks from a workstation that fails to a replacement workstation, in a transparent manner. We develop a flexible checkpointing algorithm and an efficient rollback-recovery algorithm. During fault-free execution, optimistic message logging is minimal and the checkpointing algorithm allows consistent or independent checkpointing. At the beginning of the recovery, the rollback-recovery algorithm constructs the set of orphan messages. The fault tolerance software is fully portable, because it is implemented entirely outside the kernel. The algorithms have been implemented on a network of SUN workstations running SunOS 4.1.2. We give experimental performance results.

Keywords

Portable Software, network of workstations, fault-tolerance, parallel computing.

*Accepted to the *SUUG International Conference on Open Systems: Solution for Open World*, Moscow, Russia, 25-29 April 1994.

1 Introduction

When the number of machines executing a distributed application increases, the program can fail entirely if a single machine executing a part of it fails. Hence, the distributed application becomes more sensitive to machine failures. Moreover, distributed applications are typically long run, so a lot of work (hours or days) can be lost. Therefore, allowing distributed applications to survive machine failures is an important challenge in the process of migrating from centralized to distributed computing.

Several methods can be used to provide fault tolerance in distributed systems. Using a specialized hardware may be efficient, but such a component cannot be easily added to existing systems. Application-specific methods and atomic transactions require the use of a particular programming model. Active replication is well suited for real time systems but require the use of extra processors.

We propose to handle machine failures using checkpointing and rollback-recovery. When a machine failure is detected, a replacement machine is found in the network. Practically, a checkpointing algorithm registers from time to time the state of the distributed application and a recovery algorithm restarts the distributed application from a previous state in case of a machine failure.

The goal of this paper is to investigate the possibility of designing and implementing a portable software tool providing tolerance to n faults for distributed applications in a network of UNIX workstations. By "portable", we mean that existing distributed programs need not be modified in order to be fault tolerant, and that the software tool should run entirely in user space, without requiring any modification of the UNIX kernel. Our work differs from previous one in that: *(i)* we put emphasis on portability, since our target operating system is UNIX; *(ii)* our algorithms take benefit from properties of network of workstations.

The paper is structured as follows. In Section 2, we discuss issues in checkpointing and rollback-recovery. esign choices for a network of UNIX workstations are presented in Section 3, and Section 4 describes the rollback-recovery mechanism. Then, in Section 5, we give experimental results, and we conclude in Section 6.

2 Issues in Checkpointing and Rollback Recovery

The system model consists of several processes running in parallel on different machines, with communication links between the processes [Chan85]. The state of a distributed application is defined as the collection of the local states of the processes of the distributed application, one local state per process. Fault tolerance is achieved by two components. A checkpointing algorithm registers from time to time the state of the distributed application and a recovery algorithm restarts the distributed application from a previous state.

2.1 Checkpointing Algorithms

Since checkpointing algorithms run during fault-free execution, issues in checkpointing are to minimize the time during which processes are inhibited and the quantity of information saved. And yet, checkpoints must contain enough information, avoiding the distributed application to restart from the beginning, i.e. algorithms must deal with the domino effect [Rand75]. Checkpointing algorithms described in the literature divide into two classes: synchronous and asynchronous algorithms which allow consistent and independent checkpoints, respectively.

In synchronous algorithms, processes cooperate for taking their individual checkpoints. Cooperation is achieved by the mean of control messages. Because of communication delays, the collection of individual checkpoints may not form a consistent state of the distributed application. Consistency means that "every message received must have been previously sent" [Chan85]. A consistent state includes the set of messages (if any) sent before the checkpointing of a sending process but not yet received when the checkpoint of the receiving process is

taken. Examples of synchronous checkpointing algorithms that constitute a consistent state are [Chan85, Spez86, Koo87, Li87, Leu88, Tong89, Venk89, Cris91, Elno92a].

In asynchronous algorithms, processes can take their own checkpoint when they decide so and, in order to avoid the domino effect - except in [Bhar88, Wang92] where the domino effect does occur -, every interprocess messages are saved ("logged") in stable storage. These messages would be re-played during the recovery so that the distributed application resumes execution from the instant of the failure. Messages may be processed by receivers before or after they are logged: this is optimistic message logging [Stro85, John87, Stro88, John89, Sist89, John91, Juan91, Alvi93] and pessimistic message logging [John87, Borg89, John89], respectively. Optimistic message logging is more efficient during fault-free execution but leads to a more complex recovery mechanism. Messages may be logged by senders [John87, Stro88, John89, Juan91] or by receivers [Stro85, Borg89, Sist89, John91] or by other processes than senders and receivers [Alvi93]. Sender-based message logging allows "spooling" and thus seems to be more efficient than receiver-based one. Therefore, the set of local checkpoints and the set of logged messages constitute the raw material that can be used to restart a distributed application in a consistent way after a failure.

2.2 Rollback-Recovery Algorithms

When a failure occurs, some processes have to rollback. Issues in rollback-recovery are to minimize the work to undo and to begin the rollback-recovery as soon as possible for each process which must rollback. Two characteristics of rollback-recovery algorithms correspond respectively to these two issues.

mechanisms using independent checkpointing, optimistic message logging and detecting orphan messages before the beginning of rollback-recoveries [John89, Alvi93], and, *(vi)* a mechanism using consistent checkpointing, optimistic message logging and antecedence graphs for the computation of the maximum recoverable state [Elno92c].

3 Design Choices

As far as we know, there exists six complete implementations of fault tolerance software, using checkpointing and recovery in a loosely coupled distributed system [Bhar88, Borg89, John89, Elno92b, Wang92, Alvi93]. However, in [John89, Elno92b], the implementation was done in the V distributed kernel. Targon/32 [Borg89] is a local area network of machines with three processors, connected via a dual bus, and requires atomic three-way message delivery. Using independent checkpointing only [Bhar88, Wang92], the domino effect does occur and the scheme also necessitates keeping all the checkpoints in secondary storage, even if partial logging allows to garbage discardable ones [Wang92]. Finally, in [Alvi93], the fault-tolerance software is implemented outside the kernel but uses Sun LWP to manage program concurrency, and "it can only withstand a sequence of (process crash; process recovery) pairs". This is why we decided to design, implement and evaluate a portable fault tolerance tool for networks of UNIX workstations.

We have taken in consideration the following properties of networks of UNIX workstations: *(i)* the TCP protocol achieves reliable and FIFO interprocess communications; *(ii)* a broadcast mechanism is provided; *(iii)* machine failures are scarce and independent; *(iv)* a distributed file system, such as NFS, is available; *(v)* local clocks may be (at least loosely) synchronized [*] and *(vi)* when a file server fails, all the processes running pieces of code stored on that server can no longer run when page faults occur. The last property implies that a file server involved in a distributed application may be considered as reliable. Note that the problem of the vulnerability of the file server can be solved by replication on several machines [Borg89, John89, Elno92b]. The assumption of a reliable file server has important consequences. This makes possible to maintain on it a global knowledge about the state of distributed applications. This can be used for designing a simple and flexible checkpointing algorithm and a simple and efficient recovery algorithm.

4 Rollback-Recovery Mechanism

We concentrate on distributed applications which have deterministic behaviors. The checkpointing algorithm allows both independent and consistent checkpoints, and the rollback-recovery algorithm computes the maximum recoverable state and consequently detects orphan messages before the beginning of rollbacks.

4.1 Message Logging

Message logging is sender-based. Messages sent since last checkpoints are kept in main memory [John87]. In case of a failure, a process can ask fault-free processes the needed messages. "Spooling" can be perform if volatile message logging takes too much memory space. With this simple scheme, tolerance to n faults is nearly reached: the scheme doesn't support more than two adjacent processes failing at the same time [†][Juan91]. If two adjacent processes failed, needed messages which were lost are reconstructed in due course by the processes.

So as to tolerate n faults, the history of processes is logged in main memory of the file server machine. The daemon process running on the file server machine is called "the controller". This

[*] Local clocks can be loosely synchronized by some utility, such as *rdate(8)*.

[†] Two processes are adjacent if they communicate with each other.

global knowledge is called "the archives" of the distributed application and is defined as the collection of sets of triplets {sending sequence number, receiving sequence number and receiving process identifier}, one set per process. Sending and receiving sequence numbers are linked so that, equal to n, they indicate that it is the nth events ; an event is to send or to receive a message. Archives may be sent to the controller at any time, such as after sending some given number of messages or consuming some given amount of time since the last archive logging. Again, if archives take too much space in main memory, they can be spooled out.

So, a process sends a message with, added into it, a sending sequence number, and continues its computation. The receiver gives a receiving sequence number to the message and returns an acknowledgment with, added into it, the receiving sequence number. On the reception of the acknowledgment, the sender can archive the message, or, archive logging may occur later.

4.2 Checkpointing Algorithm

The checkpointing algorithm is composed of four phases. Checkpointing a process involves two entities which are the process itself and the fault tolerance software, implemented as a daemon process in user space. In the first phase, the process saves its local state, writes a checkpoint file on the server's disk, and then resumes its execution. In parallel, the daemon process completes the algorithm. This way, the checkpointing delay as perceived by processes is minimal. In the remaining of the paper, we don't distinguish between processes of the distributed application and daemon processes. In fact, this is always daemon processes which act.

In the second phase, processes save some messages in secondary storage. As our definition of needed messages is less restrictive than the one given in [Stro88], the following lemma is correct : "provided that messages are retained on stable storage until no longer needed, then once a message is not needed, it remains not needed for ever, and hence may be discarded". Here, stable storage means that messages are either in volatile memory or spooled in secondary storage. By default, processes save in secondary storage all the messages sent before the beginning of the checkpoint. We mark messages and acknowledgements sent during the checkpointing algorithm. Thus, when a checkpointing process receives marked messages or marked acknowledgements, it means that a peer process is also checkpointing. Unmarked messages which are acknowledged marked are messages sent before the checkpoint of a process and received after the checkpoint of another process. These messages belong to a consistent cut [Chan85] and they are needed. For the same two processes, unmarked messages which are acknowledged unmarked are not needed and thus need not be saved in secondary storage. Those messages will be garbaged either from the volatile memory or from the file created when spooling.

In order to maximize the probability to meet the condition leading to optimize message saving and since we consider that machine failures are scarce, we introduce a delay between local state saving and message saving ; it is called the "message saving time". If processors clocks are approximately synchronized, if checkpointing algorithms are run periodically and if the flow of sending messages is piecewise constant, we obtain loosely synchronized checkpointing [Tong89, Cris91]. The last condition means that a process must send (respectively receive) at least one message or acknowledgement to (respectively from) other processes so that it takes part in the consistent cut. However, loosely synchronized checkpointing may interrupt a process at undesirable time. So, we could ask application programmers for predicates which would demand a delay in checkpointing. Such predicate could be for instance "I'm entering a graphical display sequence, so inhibit checkpointing".

In the third phase, a checkpoint is "surely taken" if and only if all the checkpoints belonging to the consistent cut are effectively taken. This verification consists of a dialogue with the controller. At the end of message saving, each process sends a message to the controller notifying that it took a checkpoint with such-and-such optimization. This control message is called a "checkpointing notification". Let A be the set of processes belonging to the same consistent cut and a_i be the set of processes that process p_i knows as belonging to the consistent cut. Then, when the checkpointing notification from process p_i arrives at the controller, A is modified to be equal to $A \cup a_i$. The

checkpoints of the processes belonging to the consistent cut relative to A are committed when the controller has received a checkpointing notification from all the processes belonging to A. Then, the controller sends an "end-checkpointing notification" to all processes of A. In case of independent checkpointing, processes don't wait for any end-checkpointing notification. This third phase may last a long time. The wait delay or "end-checkpointing delay" can be equal to the message saving delay multiplied by $|P|$; let P be the set of processes of the distributed application. Thus, processes only wait the end-checkpointing notification for a given end-checkpointing delay.

In the fourth phase, processes unmark previously marked messages (if any) and initiate a timer for the next checkpoint, and the controller discards some checkpoints. We apply the same definition of a needed checkpoint as the one in [Stro88] and the same lemma. For this calculus, we add a dependency vector [Stro85] to each process. Now, when a process p receives an interprocess message m from a process q, p computes the qth index of the dependency vector equal to the sending sequence number of m. Archives not corresponding to needed checkpoints are garbaged from the main memory of the file server machine.

In conclusion, our checkpointing algorithm may be either independent or consistent depending upon whether checkpointing algorithms run periodically or not. For more flexibility, checkpointing algorithms can also begin after processes send some given number of messages.

4.3 Recovery Algorithm

The rollback-recovery algorithm is patterned on two-phase-commit protocols. The initiator of the first phase is the first process which detects a machine failure, and the controller is the initiator of the second phase. Since the file server machine is reliable, a three-phase-commit protocol is never needed.

In the first phase, a process notifies all machines about the failure, and all the processes stop sending messages. Then, the controller takes the control of the distributed application and is responsible for the recovery. Archive saving is necessary before the beginning of orphan message detection in order that it can be taken for sure that orphan messages won't appear in the near future.

The controller computes the maximum recoverable state. The maximum recoverable state is defined as the set of sending sequence numbers of the messages which are first undone, one number per process. ue to space limitations, we don't insert the rules for the computation of the recoverable state.

From the maximum recoverable state, the recovery set is the set of processes causally depending on orphan messages and which must rollback. Then, the recovery line is the set of checkpoints from which dead processes and faulty processes resume their computation.

The file sever machine then selects replacement machines [‡] for dead processes. ead and faulty processes are restarted on replacement machines and on the same machines, respectively. The following information maintained by the controller is used for the reconfiguration: communication links known with archives, file names of the checkpoints known by the dialogue played during the checkpointing algorithm. Fault-free processes re-send undone messages to rolling back processes (i.e., one message sent per interprocess message undone). We prefer this alternative to the one consisting of processes asking for messages during recoveries (i.e., two messages sent per interprocess message undone). The missing messages are orphan messages and will be reconstructed in due course by rolling back processes. At the end of the first phase, rolling back processes archives are marked to be re-done up to the maximum recoverable state.

In the second phase, the controller commits the overall recovery, and processes resume their execution. uring rollbacks, the controller manages rollbacks by the way of archive saving.

Finally, when failures occur during the rollback-recovery algorithm, the algorithm restarts from the beginning or from the end of the maximum recoverable state computation. When failures

---

[‡] Several algorithms have been designed for finding a suitable workstation (e.g, [Bern91]).

occur during rollbacks, the rollback-recovery algorithm run excluding rolling back processes. And, when failures occur during checkpointing, checkpointing is finished without any optimization for fault-free processes which have already saved their local state on secondary storage.

5 Experimental results

5.1 Implementation Details

The checkpointing and rollback-recovery algorithms have been implemented in a network on Sun4 workstations running SunOS 4.1 (25 MHz clock rate, 16 Mbytes of main memory) with a Sun SPARC670MP Server (40 MHz clock rate, 64 Mbytes of main memory, three 1.3 Gbytes SCSI disk drives with 5 Mbytes/s transfer rate each).

The starting point was a set of communication primitives for distributed computing and the corresponding runtime [Bern89]. The runtime has been modified in order to include fault tolerance software. On every workstation, a daemon process runs in user space. This daemon acts as an intermediary between application processes for their communication. When an application process has a message to send to another application process, the message is first passed to the daemon process that performs the appropriate tasks related to fault tolerance (message numbering, volatile message logging, archive saving, checkpointing, failure detection, recovery). On the receiving side, messages are buffered by the daemon until being requested by application processes. Communication between daemons are based on the TCP protocol. Machine failures are detected by loss of TCP connections.

User programs are written in a standard programming language (such as C) and use a set of predefined primitives for communication between remote parts of the distributed application. These primitives are implemented in a library that handles the internal communication with the local daemon. So, application programmers have just to link their source module with the library, without having to worry about fault tolerance aspects.

A mechanism that was designed for checkpointing process state for the purpose of process migration [Alar92] is used for checkpointing application process state. It also runs out of the kernel. Thus, the whole fault tolerance software (daemon process code, checkpointing code, communication primitives) is fully portable on standard (Berkeley) UNIX systems.

5.2 Experimental results

Like in [Alvi93], we have measured the performance of our fault-tolerance software with a loop simulating a token ring with four workstations. Application computational overhead is then near to zero and almost all the measured time is devoted to interprocess communication. The tests perform 1,250 loops, so, the total number of messages sent and received is thus 5,000. Messages contain 0 byte, 1024 bytes or 2032 bytes of data, with a 16-byte header (message type - i.e., messages are interprocess or control messages -, sending sequence number, a flag used for instance to mark messages, and data length).

As could be expected, fault-tolerance is expensive because of communication operations. Each interprocess (application-to-application) message necessitates the sending of four internal messages. By using LWP, as in [Alvi93], interprocess communication delays would be roughly divided by two, since each interprocess communication message would necessitate only two internal messages. However, remind that one of our objectives is to evaluate the overhead of fault-tolerance using checkpointing and rollback-recovery provided by a fully portable tool.

We measured the overhead of message logging both on workstations and on the file server, and the cost of checkpointing.

On workstations, the overhead is quantified by the increase in application response time. For 0 byte (respectively 1024 bytes and 2032 bytes) messages, the overall time is ranging from $14.86s$ (respectively $24.30s$ and $33.52s$) using pessimistic message logging to $14.40s$ (respectively $24.90s$

and 33.14s) using no message logging. Without the fault-tolerance software tool, the overall time is ranging from 3.90s (respectively 10.55s and 16.14s). Message logging overhead is very low, ranging from 2.9 % with 0 bytes messages to 1.2 % with 2032 bytes messages. With an ideal communication environment, assuming only one internal message sent for one application-to-application message, the message logging overhead would be at most $4 \times 2.9 = 11.6\%$, yet for pessimistic message logging. These results are better than the one presented in [Borg89, John89, Alvi93]. The reason is the use of volatile archive saving one the file server instead of second storage archive saving.

On the file server, the overhead of message logging is quantified by the increase of CPU utilization. Figure 1 shows the CPU utilization of the file server versus the message logging frequency. It can be seen that pessimistic message logging is somehow expensive. However, if application

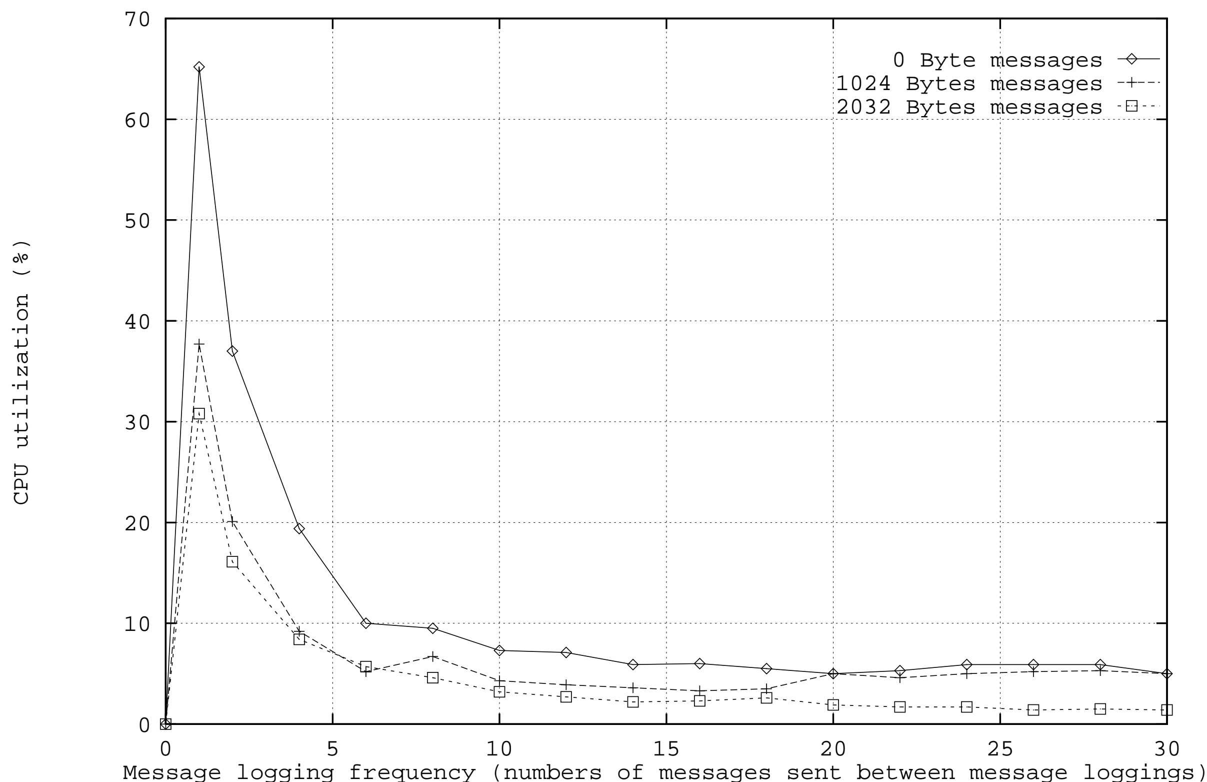

Figure 1: *CPU Utilization of the File Server Workstation versus Message Logging Frequency*

programmers choose optimistic message logging with a message logging frequency greater than 10 messages between message loggings (considered as a suitable value in [John89]), the CPU utilization is less than 7 % for four workstations. So, archive saving on the file server workstation doesn't slow down significantly the file server.

The cost of checkpointing is evaluated by the time necessary to take a checkpoint. In our test, each checkpoint of the distributed application occupies 50,000 bytes in secondary storage (data and stack - process code is not needed [Alar92]). The message logging frequency is 10 messages between message loggings in order to experiment optimistic message logging. Messages are empty but the conclusions remain with 1024 bytes and 2032 bytes messages. Moreover, checkpoints are periodically taken and loosely synchronized in order to minimize the number of messages saved in secondary storage. Figure 2 shows the local state saving time, the message saving time, the communication time and the overall time versus the number of checkpoints taken during the execution. The local state saving time is the time for saving the local state in secondary storage. It increases in a linear way. Let l be the local state saving time in seconds and c be the number

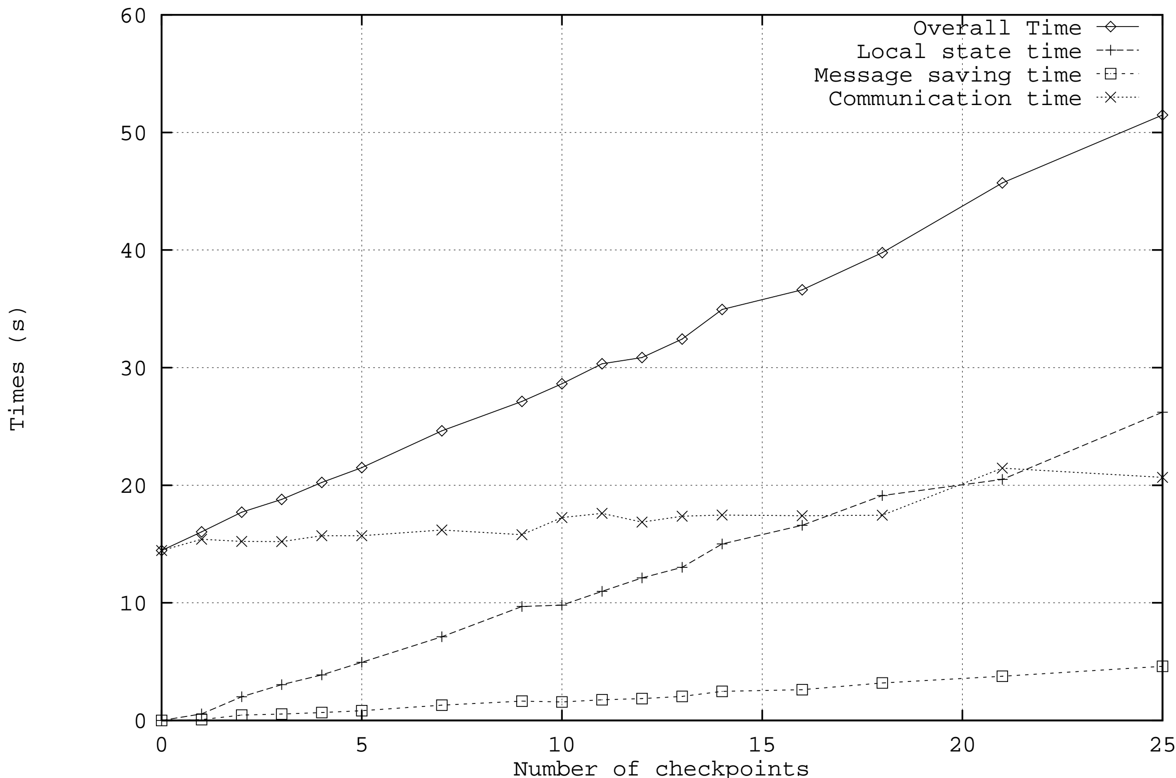

Figure 2: *"1,250 Loops" Application Time versus Checkpointing Frequency*

of checkpoints taken, we have the equation: $l = 1.04 \times c$. In fact, the CPU utilization during checkpoints averages 25 ms for a 50 Kbytes local state. The difference between elapsed times and CPU utilization times is due to the slowness of write operations on server's disks. The message saving time is the time for saving needed messages in secondary storage. It also increases in a linear way. Let m be the message saving time in seconds, we have the equation: $m = 0.17 \times c$. The number of needed messages saved in secondary storage averages 4 messages. This result says that checkpointing was always loosely synchronized. The message saving time increases slowly because a file is created for saving needed messages even if there is no needed messages. However, the communication time increases in a monotonic way but not in a linear way. Let *comm* be the communication time in seconds, we can roughly define the stair case function: $comm = 15.6$ for $1 \leq c < 10$, $comm = 17.3$ for $10 \leq c < 20$, and $comm = 21.0$ for $20 \leq c \leq 25$. It demonstrates that the network acts as a bottleneck. We haven't shown checkpointing frequency greater than 50 messages between two checkpoints - i.e., 25 checkpoints taken - because this doesn't seem realistic.

6 Conclusion

We have designed and implemented a fault tolerance mechanism using message logging, checkpointing and rollback-recovery.

In networks of UNIX workstations, a workstation can be considered as reliable. Practically, this workstation is the file server. Assuming a file server is a reliable machine has important consequences. This makes possible to maintain on it a global knowledge of the system, such as the history of distributed applications (call "the archives"). Then, archive logging on main memory of the file server machine is added to volatile and optimistic message logging.

In checkpoints, needed messages are saved in secondary storage with local states of processes. These two savings are concurrent. We mark messages and acknowledgements sent between the end of local state saving and the beginning of message saving. So, all processes may know that other processes are checkpointing at the same time, and thus, processes can optimize the number of messages saved in secondary storage. Application programmers can consequently choose between either using independent checkpointing or using loosely synchronized checkpointing.

The recovery algorithm is designed as a two-phase-commit algorithm. In the first phase, a process notifies all processes of the distributed application about the failure, and the file server machine daemon process prepares the recovery. ead processes restart from their recovery checkpoint and receive messages from fault-free processes. In the second phase, the file server machine notifies processes to resume their execution and manages the rollbacks.

We have implemented a software tool in a network of UNIX workstations. This mechanism runs entirely in user space and requires no modification to the standard UNIX kernel. Thus, it is fully portable on any network of UNIX workstations. Fault tolerance is transparent to the application programmer, who simply uses high-level primitives for describing message exchanges between remote parts of a distributed application.

Performance tests show that message logging perform well because of archive logging on the main memory of the file server machine, and that loosely synchronized checkpointing can easily be obtained.

References

[Alar92] E. Alard and G. Bernard. Preemptive Process Migration in Networks of UNIX Workstations. In *Proc. 7th International Symposium on Computer and Information Sciences*, Antalya, Turkey, 2-4 November 1992.

[Alvi93] L. Alvisi, B. Hoppe, and k. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proc. 23th IEEE Symposium on Fault Tolerant Computing*, 1993.

[Bern89] G. Bernard, A. uda, Y. Haddad, and G. Harrus. Primitives for istributed Computing in a Heterogeneous Local Area Network Environment. *IEEE Transactions on Software Engineering*, SE-15(12), ecember 1989.

[Bern91] G. Bernard and M. Simatic. A ecentralized and Efficient Algorithm for Networks of Workstations. In *Proc. EurOpen Spring '91*, Tromso, Norway, 20-24 May 1991.

[Bhar88] B. Bhargava and S.-R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery in istributed Systems - An Optimistic Approach. In *Proc. 7th IEEE Symposium on Reliable Distributed Systems*, 1988.

[Borg89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[Chan85] K.M. Chandy and L. Lamport. istributed Snapshots: etermining Global States od istributed Systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[Cris91] F. Cristian and Farnam Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived istributed Computations. In *Proc. 10th IEEE Symposium on Reliable Distributed Systems*, 1991.

[Elno92a] E.N. Elnozahy, .B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proc. 11th IEEE Symposium on Reliable Distributed Systems*, 1992.

[Elno92b] E.N. Elnozahy and W. Zwaenepoel. Implementation and Performance of Transparent Rollback-Recovery in Manetho. Technical report 92-197, epartment of Computer Science, Rice University at Houston, Texas, 1992.

[Elno92c] E.N. Elnozahy and W. Zwaenepoel. Manetho : Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computer*, 41(5), May 1992.

[John87] .B. Johnson and W. Zwaenepoel. Sender-based Message Logging. In *Proc. 17th IEEE Symposium on Fault Tolerant Computing*, 1987.

[John89] .B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* Ph thesis, Rice University, ecember 1989.

[John91] .B. Johnson and W. Zwaenepoel. Transparent Optimistic Rollback Recovery. *ACM Operating Systems Review*, 25(2), April 1991.

[Juan91] T.T-Y. Juang and S. Venkatesan. Crash Recovery With Little Overhead (Preliminary Version). In *Proc. 11th IEEE International Conference on Distributed Computing Systems*, 1991.

[Koo87] R. Koo and S. Toueg. Checkpointing and Rollback Recovery for istributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[Leu88] P-J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in istributed Systems. In *Proc. 4th IEEE International Conference on Data Engineering*, 1988.

[Li87] H.F. Li, T. Radhakrishnan, and k. Venkatesh. Global State etection in Non-FIFO Networks. In *Proc. 7th IEEE International Conference on Distributed Computing Systems*, 1987.

[Rand75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[Sist89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[Spez86] M. Spezialetti and Kearns P. Efficient istributed Snapshots. In *Proc. 6th IEEE International Conference on Distributed Computing Systems*, 1986.

[Stro85] R.E. Strom and S.A. Yemini. Optimistic Recovery in istributed Systems. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[Stro88] R.E. Strom, .F. Bacon, and S.A. Yemini. Volatile Logging in n-Fault-Tolerant istributed Systems. In *Proc. 18th IEEE Symposium on Fault Tolerant Computing*, 1988.

[Tong89] Z. Tong, R.Y. Kain, and W.T. Tsai. A Low Overhead Checkpointing and Rollback Recovery Scheme for istributed Recovery. In *Proc. 8th IEEE Symposium on Reliable Distributed Systems*, 1989.

[Venk89] S. Venkatesan. Message-Optimal Incremental Snapshots. In *Proc. 9th IEEE International Conference on Distributed Computing Systems*, 1989.

[Wang92] Y-M. Wang and W.K. Fuchs. Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems. In *Proc. 11th IEEE Symposium on Reliable Distributed Systems*, 1992.