Making Distributed Applications Fault-Tolerant in Networks of UNIX Workstations[1]

Guy Bernard and Denis Conan
Institut National des Télécommunications
9 rue Charles Fourier 91011 EVRY Cedex France
Phone: +33–1–60 76 45 67 Fax: +33–1–60 76 47 80 e-mail: bernard@int-evry.fr

Abstract

This paper addresses the problem of fault tolerant distributed computing in networks of UNIX 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. We discuss how checkpointing and rollback recovery can be used for this purpose. We show how one can rely on the characteristics of networks of UNIX workstations for designing a flexible checkpointing algorithm and an efficient recovery algorithm. The fault tolerance software is fully portable, because it can be implemented entirely outside of the kernel. Moreover, fault tolerance is transparent to application programmers, who just have to use some predefined communication primitives for describing message exchanges between processes of distributed applications.

[1]Accepted to the Italian Unix Users Conference, Milano, Italy, May 1993.

1 Introduction

Networks of workstations become more and more widespread computing environments, in particular because of their attractive performance/cost ratio. In such an environment, migrating from centralized applications (executing on one machine) to distributed applications (executing simultaneously on several machines of the network) is a natural evolution. However, when the number of machines executing an application increases, the application becomes more sensitive to machine failures, since the program can fail entirely if a single machine executing a part of it fails. Moreover, distributed applications are typically long run, so if the program fails, a lot of work (hours or days) can be lost. Fault tolerance, allowing distributed applications to survive machine failures, is thus 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 by application programmers. Active replication, such as ISIS [Birm87], are well suited for real time systems, but require the use of extra processors for running the replicas of each program.

The goal of this paper is to investigate the possibility of designing and implementing a portable software tool providing fault tolerance 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 is close to the one of Johnson [John89], who implemented a fault tolerance mechanism under the V distributed kernel. The major differences are: *(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 how checkpointing and recovery can be used to achieve fault tolerance. Design choices for a network of UNIX workstations are presented in Section 3, and Section 4 describes the current state of our prototype.

2 Checkpointing and Rollback Recovery

Our computation model of a distributed application consists in several processes running in parallel on different machines, with communication links between some processes. Machine failures are handled in the following way. When a machine failure is detected, a replacement machine is found in the network, and the role that played the failed machine in the distributed computation is transferred to the replacement machine. The replacement machine may be one already involved in the distributed application, but in general it will be a new one, since the number of machines in a network of workstations is generally greater than the parallelism degree of the distributed application. Fault tolerance is thus achieved by two components, a checkpointing algorithm that registers from time to time the state of the distributed application (or parts of this state), and a recovery algorithm that restarts the application from a previous state in case of failure.

Checkpointing algorithms described in the literature divide into two classes: synchronous and asynchronous. In synchronous algorithms, the processes involved in a distributed application cooperate for taking their individual checkpoints. Cooperation is achieved by the mean of control messages. Because of communications delays, the collection of individual checkpoints may not be a consistent state of the distributed application. Consistency means that "every message received must have been previously sent" [Chan85], i.e. the reception of a message cannot be included into the checkpoint of a process if the sending of the message is not already included in the checkpoint of another process. Thus, a consistent state must also include the set of messages that were 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 global state are [Chan85] and [Koo87]. In asynchronous algorithms, processes can take their own checkpoint when they decide so, and *all* interprocess messages are saved ("logged") in stable storage. Messages may be logged by the sending processes or by the receiving processes. No consistent global state is permanently maintained. However, the set of individual 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. Examples of asynchronous checkpointing algorithms are [John87], [Stro88] and [Borg89].

Recovery algorithms involve restarting at least one process (the one(s) that was (were) running on the machine that has failed), from its (their) last checkpoint, on a replacement machine. However, other processes may be involved in the recovery. Of course, since recovery makes use of information built by checkpointing, recovery algorithms are not independent from checkpointing algorithms. With synchronous checkpointing algorithms, a consistent global state is readily available. In this case, synchronous recovery is appropriate: when a failure is detected, all processes are stopped and the whole application restarts from the last consistent state [Koo87]. With asynchronous checkpointing algorithms, recovery can be asynchronous. A restarting process resumes its execution from its last checkpoint, independently from other processes, and logged messages are replayed (i.e., they are not resent by some process, but their content is recovered from stable storage) when needed. Other processes are aware of the recovery only if (and when) they must restart, which occurs when a message that they sent to the failed machine was not acknowledged, and thus could not be stored in stable storage as "sent and received". An example of asynchronous recovery algorithm is described in [Stro85]. However, since asynchronous checkpointing algorithms register more information (messages) than synchronous ones, synchronous recovery with asynchronous checkpointing is possible too [Stro91]. What is needed here is a centralized or distributed computation for determining the set of processes that must restart from their last checkpoint in order to preserve a consistent state.

3 Design Choices

As far as we know, only one complete implementation of fault tolerance software, using checkpointing and recovery in a loosely coupled distributed system, has been achieved [John89], but the implementation was done under the V distributed kernel. This is why we decided to design, implement and evaluate a portable fault tolerance tool for networks of UNIX workstations.

Networks of UNIX workstations present some properties: *(i)* the TCP protocol achieves reliable and FIFO interprocess communication; *(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 processes running on workstations pieces of code stored on that server can no longer run when page faults occur.

We have taken these properties into consideration for designing a portable fault tolerance software tool. The last property implies that a file server involved in a distributed application may be considered as reliable, since no fault tolerance software running on workstations will ever be able to handle the impossibility of running application code when the binary files can no longer be accessed if the server crashes.

Assuming a file server is a reliable machine has important consequences in our design. It makes possible to maintain on the server a global knowledge about the state of distributed applications. This can be used for designing an efficient recovery algorithm.

The checkpointing algorithm should be as lightweight as possible, for performance reasons (it will be run several times during the execution of a distributed application). On the other hand, the recovery algorithm may be more complex, since machine failures are scarce: most of the time, it will never be run.

3.1 Checkpointing Algorithm

Synchronous checkpointing algorithms have two drawbacks: first, they may interrupt a process at undesirable times (for instance, a process should not be interrupted in the middle of a graphical display sequence); second, they involve a large number of messages for synchronization purposes. Thus, we believe that an asynchronous checkpointing algorithm is more appropriate.

Message logging is done by the sending process rather than by the receiving process. The reason is that if message logging is done by the receiving process, messages must be logged on a remote machine in order to be retrievable if the receiving machine fails, involving extra communication costs. In our algorithm, the content (i.e., the data field) of the messages sent after the last checkpoint is kept in main memory on the sending machine until the next checkpoint, and the archives of sending processes (i.e., sending sequence number, receiving sequence number and receiving process identifier - not message data) are kept in main memory on the file server machine. This is the scheme used in [Stro88], except that no disk access is required for storing and retrieving the archives. When a process takes a checkpoint, it can be taken for sure that the messages it received before the checkpoint will no longer be requested if a failure occurs, and the corresponding data and archives can be discarded from main memory of the sending workstation and of the server.

Checkpointing a process involves in fact two entities (see Section 4): the application process itself, and the fault tolerance software, implemented as a daemon process in user space. The application process itself takes its checkpoint strictly speaking (i.e., its state), writes a checkpoint file on the server's disk, and then resumes its execution, while in parallel the daemon process completes the algorithm by moving, from main memory of the workstation to the server's disk, the logged messages. This way, the checkpointing delay as perceived by a distributed application is minimal.

Moreover, if two processes are checkpointing simultaneously, the set of messages that must be moved to the server's disk can be reduced (since it can be known that some messages will never be requested [Stro88]). This information can be used to optimize message logging, by reducing both disk access time and communication costs. Each process can know that a peer process is concurrently checkpointing because we mark application messages sent during checkpointing (no explicit control messages are needed). In fact, our checkpointing algorithm is flexible. Any process can decide to checkpoint at any time, independently of other processes (in this case, all messages in main memory are moved on the server's disk if no other process is currently checkpointing), or checkpoints can be loosely synchronized, as is proposed in [Tong89]. For instance, a distributed application can be started with previsional common checkpointing times for some or all workstations[1]. When it is "time to checkpoint", each daemon process invites local application processes to checkpoint, and wait a (small) delay before moving the logged messages to the server's disk, with the hope that during this delay marked application messages will be received, in which case the set of messages moved to the server will be reduced. In other words, it is not necessary that *all* processes synchronize before local checkpointings take place, but in case of simultaneity the algorithm is optimized.

3.2 Recovery Algorithm

The checkpointing algorithm registers (in publicly readable files) on the server's disk the state of the processes (checkpoints strictly speaking) and the messages that can be requested after a failure. These informations can thus be retrieved from any workstation in the network.

The main problem of asynchronous recovery is that cascading recoveries may occur. An asynchronous recovery procedure begins with restarting one process alone. During its execution, this process can discover that it must wait for another process to restart too (for resending a message that was not logged), and so on. Thus, the global recovery delay after a failure may be quite long. For instance, with ten processes and one hour between checkpoints, it can be ten hours in the worst case.

[1]Assuming local clocks are approximatively synchronized by some utility, such as *rdate(8)*.

In order to avoid cascading recoveries, our recovery algorithm is synchronous. This assumes that global information about distributed application state is available. With our checkpointing algorithm, there is a machine which centralizes this information (the server). Thus, even though checkpointing is basically asynchronous, we can design a synchronous recovery algorithm. It works in the following way. When a failure occurs, the server is notified by the machine that detected the failure. The server stops all running processes of the distributed application (i.e., interprocess communications are inhibited, but local computation goes on). It then determines, from information locally stored, (list of processes involved in a distributed application, along with their communication links, the machines they are running on, the file names of their last checkpoint, the file names of logged sent messages, and message archives) the minimal set of processes that have to restart from their last checkpoint (processes that were running on the machine that failed, and maybe some others). The server then selects a replacement machine[2], the appropriate communication links are reopened from this machine. Last, all processes involved in the recovery restart simultaneously from their last checkpoint on a signal delivered by the server. This way, the global recovery time is the longest of the individual process recovery times, and not their sum, like it can be in asynchronous algorithms.

Requested messages are retrieved either from main memory of workstations or from the server's disk (their location is known from information contained in the archives), rather than being resent by application processes themselves that would have restarted from their own checkpoint. This has two advantages. First, unuseful restarts are avoided. An extra process must restart only when the failure occurred between a message sending and the update of the corresponding archive on the server. Second, the same messages are received in the same order by the process that was running before the failure and by the process that restarts on a replacement machine, thus the re-execution is equivalent (i.e., gives the same results) to the old one.

4 Current State and Implementation Details

The checkpointing and recovery algorithms have been implemented in a network on Sun4 workstations running SunOS 4.1. 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 runs a daemon process 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, message storage in main memory, archive update on the server, failure detection, recovery). On the receiving side, messages are buffered by the daemon until being requested by application processes.

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

Communication between daemons are based on the TCP protocol. Machine failures are detected by loss of TCP connections. A special daemon process runs on the file server, for achieving archive management and recovery.

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 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.

The prototype is now in operation. We are currently evaluating the cost of fault tolerance (overhead in interprocess communications, checkpointing delay, recovery delay) on real distributed applications. Further work is also needed for correctly handling input/output from/to the external world (e.g., writing a data record on a tape should not be repeated if the writer process is restarted after a failure).

5 Conclusion

We have designed and implemented a software fault tolerance mechanism for distributed computing in a network of UNIX workstations. This mechanism runs entirely in user space and requires no modification to the standard UNIX kernel, thus 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. Our checkpointing algorithm is flexible, since it can be used in an asynchronous way or in a fully or partially synchronous way, and efficient because checkpointing is done in parallel by two UNIX processes. Our recovery algorithm minimizes the set of processes that must restart after a failure and recovery duration is optimal. This is possible because global information is centralized on a single machine that can be considered as reliable: a file server. Both checkpointing and recovery algorithms rely on the existence of a network file system, which permits access to permanent data from any workstation.

It should be noted that our tool may be used for other purposes than fault tolerance. If workstations are dedicated to a user, this one may require the full availability of his/her workstation for doing some intensive computation. In this case, all foreign processes involved in a distributed application and running on his/her workstation can be moved on another workstation, as if a failure occurred.

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.

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

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

[Birm87] K.P. Birman and T.A Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. 11th ACM Symposium on Operating Systems Principles*, Austin (USA), November 1987.

[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. Distributed Snapshots: Determining Global States od Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

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

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

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

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

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

[Stro91] R.E. Strom, A.P. Goldberg, A. Gopal, and A. Lowri. Restoring Consistent Global States of Distributed Computations. *Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 26(12), December 1991.

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