STAR: a Fault-Tolerant System for Distributed Applications

Pierre Sens

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

Bertil Folliot

Blaise Pascal Institute
Paris VII University, France
2, place Jussieu - 75251 Paris Cedex 05
email: folliot@masi.ibp.fr

Abstract

This paper presents a fault-tolerant manager for distributed applications. This manager provides an efficient recovery of hosts' failures on networks of workstations. An independent checkpointing is used to automatically recover application processes affected by host failures. Domino-effects are avoided by means of message logging and file versions management. STAR provides an efficient software failure detection by structuring hosts in a logical ring. Performance measurements in a real environment show the interest and the limits of our system.

1 Introduction

Distributed systems provide new opportunities for developing high-performance parallel and distributed applications. However, because of the components dependency, such systems are particularly fragile: the failure of one component may imply the whole system failure. Many existing systems integrate efficient resources management for distributed applications among a set of processors [4, 17], but few systems offer a fault tolerant mechanism [8, 13, 14].

This paper presents ***STAR***, a system managing fault-tolerant distributed applications in a network of workstations. An application is a dynamic set of communicating programs accessing files. STAR is designed to achieve five main goals:

- *fault-tolerance*: users' applications tolerate simultaneous hosts' failures,
- *transparency*: failure occurrence is transparent to applications,
- *failure confining*: only processes running on faulty hosts are recovered,
- *flexibility*: STAR is application independent and highly configurable,
- *portability*: STAR is designed on top of the UNIX operating system and is easily portable to any other UNIX-like operating systems.

The failure detection delay is a crucial point for fault-tolerant managers. To provide an efficient failure detection, we use a specific structuring of hosts in a *logical ring for host crash detection.*

We use an independent checkpointing mechanism to recover processes [10]. When a failure occurs, concerned processes and their files are restored from checkpoints. A reliable message logging allows to recover only processes allocated on faulty hosts [16, 18] and thus, to avoid all domino effects.

We present in section 2 the application, environment and failure models managed by STAR. Section 3 describes the design of our system. Finally, we give performance measurements of STAR in a real environment.

2 Application, environment and failure models

STAR deals with distributed applications. An application is a dynamic set of communicating processes. A process can use any resources of the network (CPU and files). The only way to exchange information between processes is through message passing. All processes in the system are *deterministic*. The state of a process is

Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing, Dalla Texes, December 1993 (IEEE Computer Press)

determined by its starting state and by the sequence of messages it has received.

STAR works on a set of workstations connected together by a local area network (Ethernet). We consider that the failure probability is low and that there is little interest to manage fault-tolerant for short running applications. Applications concerned by a fault-tolerant management have long execution time such as high number factoring, VLSI applications or image processing. Such applications may run for hours, days or weeks. In that case, the failure probability becomes significant.

We make the following assumptions about the failure model:

- The system is composed of fail-silent processors [15], where the failed nodes simply stop on failing and all the processes on the node die.
- Failures are *uncommon events* while very short recovery delays are not required. Thus, we always favor a solution with a lower overhead in normal functioning.
- We only consider host failures. A host is considered faulty if it is not accessible. We will not consider network partitions in this paper.

3 STAR description

In order to satisfy our flexibility and portability requirements, we designed STAR on top of the UNIX operating system. Users' applications rely on a *fault-tolerant layer* that provides access to all external components (processes and files). This approach is more costly than a fault-tolerance management integrated in the kernel but it has the advantage of not being system dependent.

All accesses to processes or files are done through the fault-tolerant layer. This layer provides a *global naming space* for processes. Thus, the process identification is location independent and a process can transparently migrate.

3.1 Host crash detection

Software host crash detection is rarely described in the literature. The straightforward and most used way to detect a host failure is to wait for a normal host access to fail. This detection method has no overhead but the failure treatment can only occur when one needs to use the faulty host. Thus, the response time in case of failure can be very high because it depends on the network traffic. 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.

We use a combination of these two methods. The normal traffic is used as in the first method, but when the traffic stops between two hosts during a given time (specified by the network administrator, for instance three seconds), we generate a specific detection message.

This combination method raises an other problem: how each host checks each other ? To respect our portability principle, we don't want to use special hardware. We operate a logical structuring of hosts in a *logical ring of detection*. Each host only checks its immediate successor in the ring. The checking process is straightforward and the cost in messages traffic is low.

As for all logical structuring of hosts, a ring reconfiguration protocol must be executed when adding or removing a host [3, 12].

Ring insertion: when a new host joins the fault-tolerant system, it is inserted into the detection ring. The logical ring is scheduled according to the host identifications (host ID). We don't use any centralized coordination. Each host has a global view of all hosts in the ring. When a host wants to be inserted in the ring, it broadcasts its host ID. On reception of this message, each host updates its global knowledge. The logical predecessor connects itself to the new host and this one connects itself to its successor. Finally, the predecessor transmits its global knowledge of the ring to the new host.

Ring reconfiguration: when a host detects its successor's failure, it initiates a reconfiguration step. The detector connects itself to the first valid successor and broadcasts the new structure of the ring.

3.2 Processes recovery

Our recovery mechanism is based on a checkpointing method. The related works in checkpointing are divided in two classes: *consistent* and *independent* checkpointing.

In the first approach, processes coordinate their checkpoints such that the set of checkpoints forms a snapshot of a consistent global state of the system [6]. When a failure occurs, processes rollback to their last checkpoints. The main drawback of this approach is that messages used for synchronizing a checkpoint are 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 [11].

We adopt the second approach where processes save their states independently. This technique is simple, but since the checkpoint of processes may not define a consistent global state, the failure of one process leads to rollback other processes. To avoid this classical domino effect, we use a *reliable message logging* [10, 18]. This logging is particularly adapted for applications composed of processes exchanging small streams of data. In that case, the message logging overhead becomes negligible.

In order to ensure fault-tolerant file accesses, users' files are duplicated on several disks belonging to several hosts. Replicated files are not sufficient for users' processes. 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 also manage versions of duplicated files (see section 3.4).

3.3 Communication management

The STAR communication protocol is based on the local simulation of interactions for the recovered processes. It relies on the confining principle: "*a recovered process has no interaction with the others until it reaches the last state before the failure*". To respect this principle, we use message logging of all received messages and we detect retransmissions from recovered processes.

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. The reception of messages is done by the fault-tolerant layer. This layer accesses to the backup or waits for messages according to the user 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 determinists, a recovered process sends again all messages since its last checkpoint. The sending is done by the fault-tolerant layer. In this layer, a stamp on each message allows *to detect retransmissions of messages*. 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 a message with the stamp of the last transmitted message. This method avoids the reception of an old message from a recovered process.

With this protocol, the fault transparency and fault confining goals are respected.

3.4 File management

The file manager is the central point of STAR. It is used for users' files, for messages backups and for checkpoints storing. This manager has the following properties:

- *fault tolerance*: each file is duplicated on separate disks,
- *coherency*: file copies are kept identical,
- *version management*: each used file has an old version (also duplicated) corresponding to the last checkpoint of the process.

To ensure coherence of all copies, the file manager performs a reliable broadcast protocol [2]. A file's update is atomically broadcasted to all managers keeping a copy. A read operation is done locally whenever possible.

The file manager also manages versions of duplicated files. There are two versions of used files: a current one for all accesses and a checkpoint one created at the checkpoint time. When a process rolls back, old versions of used files replace current ones.

3.5 Failures Treatment

The failure treatment begins after the ring reconfiguration. First, faulty processes (processes that were running on the faulty host) are identified, then new processes are created on a valid host and their states are restored from checkpoints.

Each host knows all information concerning the applications' processes. This knowledge allows a local identification of faulty processes. It is updated each time a process is created, terminated or recovered.

After the identification step, we choose a valid host to reallocate faulty processes. In the first version of STAR, processes are relaunched on the host which has detected the failure. This may overload the detecting host, if a big number of processes were running on a faulty site, or if the host detects successively several failures. The second version of STAR includes a load balancing capability for dynamic reallocation of processes when recovered [5].

4 Performances

In this section, we present the performance of STAR. We evaluate the cost of the communication protocol, checkpoints and failure recovery. These measures will help system administrators or application designers to choose the good parameters (various degrees of replications, period of checkpointing) according to their fault-tolerance and performance requirements.

All measures have been done in a set of Sparc Station 1 with 24 Mb of memory connected by Ethernet. The environment has not been modified (usual daemons were running).

4.1 Checkpoint evaluation

The figure 1 shows the checkpointing cost with different replication degrees. The replication degree is the number of identical copies of the checkpoint files. This cost directly depends on the process context size. We indicate the checkpointing time for programs with different size of local data.

For a 100 kilobytes' data program and for a replication degree of two, the cost of a checkpoint is 1.4 seconds. For the same size and a replication degree of four, the cost is 2.7 seconds.

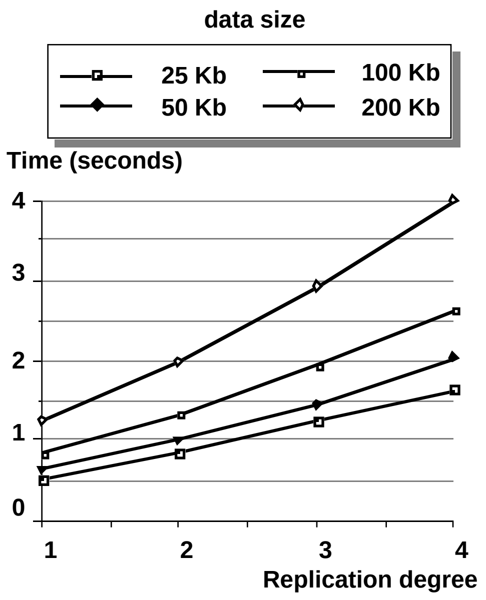

Figure 1: Checkpoint cost

Such performances are relatively good compared to other systems. For instance, DAWGS [7] does a similar checkpointing with only one replication degree (i.e., a faulty process must wait for the faulty host to be restarted). In this system, the checkpoint time is about 3.5 times slower (1.84 seconds for only 25 Kbytes) than the checkpoint time in STAR.

Elnozahy et al [8] have implemented a consistent checkpointing with also one replication degree. They obtain results according to the application complexity. For a relatively simple application, such as a distributed gaussian elimination, the checkpoint time to save 200 Kbytes of data is about 6 seconds (5 times slower than in STAR). For a complex application, such as a computation of a grid, the checkpoint time is about 10 seconds (about 8 times slower than in STAR). In consistent checkpointing, the cost of the checkpoint depends not only on the process size but also on the synchronization between involved hosts.

Bhargava et al [1] also 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).

Borg et al [3] have implemented a fault-tolerant version of UNIX based on three-way atomic message transmission: the TARGON/32 system. They measured the performance on only two machines. In that case, the performance turns out to be 1.6 times that of a standard UNIX. This system is totally transparent to users, but implies more overhead than in STAR.

4.2 Recovery evaluation

We have evaluated the cost to restore faulty processes. Measures have been done with different processes sizes. They are quite linear. For a 25 Kbytes process, the average restoration time is 2.2 seconds. For a 50 Kbytes process, it is 2.5 seconds, for a 100 Kbytes process, it is 3.2 seconds, and for a 200 Kbytes process, it is 4.1 seconds.

This restoration time seems important compared to the checkpoint cost. In fact, the restoration step is much more complex. It includes the time to identify the process, to reconfigurate the ring, to create a new process, to restore its context and eventually to update the global knowledge.

In the TARGON/32 system the average recovery time for a process is 5-15 seconds.

4.3 Communication protocol evaluation

We have evaluated the cost of the STAR communication protocol according to the replication degree of backups. The replication degree is the number of identical copies of the backup. The cost to send one message with one backup is 1.6 times slower than without backup. It is 2.2 times slower for two backups, 2.6 times slower for three backups, and 3 times slower for four backups.

Process that only communicates would see their performances at least divided by two. The communication protocol is the weak point of our system, and more generally, of all systems based on message logging. Thus, STAR is particularly adapted for long running application composed of processes exchanging small streams of data.

5 Conclusion

We have presented the STAR fault-tolerant system for distributed applications. It uses an independent checkpointing of processes. By a reliable storage of processes' messages, the well-known domino effect is avoided. STAR provides an efficient host crash detection with a logical structuring of host in a ring.

STAR has been developed on a set of Sparc stations connected by Ethernet. We gave some performance measures that show the efficiency of our system and that prove that a fault tolerant system is viable for a workstation model. The limit concerns highly communicating programs that suffer from the reliable logging cost. Otherwise, the STAR overhead is negligible.

We developed a new version to take benefit of a load balancing manager (the GATOS System [4, 9] developed at the MASI Lab.). GATOS allows to use efficiently all the available processing power. It is used by STAR when starting or recovering a process. The mean improvement observed by using GATOS is from 20 to 50% (depending on a large number of parameters, such as the parallelism degree, the CPU needs and the overall network load).

The scalability of STAR is limited by the size of a LAN (20-30 hosts). In large distributed systems, the overhead of information distribution becomes too important. A future extension will be to manage fault-tolerance in wide area network.

References

[1] 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.

[2] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272-314, August 1991.

[3] 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.

[4] 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.

[5] R. Boutaba, B. Folliot, and P. Sens. Efficient Resources Management in Local Area Networks. In *Proc. of the International Conference on Advanced Information Processing Techniques for LAN and MAN Management*, IFIP WG6.4, Versailles, Avril 1993.

[6] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, 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] 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 92.

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

[10] 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.

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

[12] G. Le Lann. Synchronization in Local Area Networks: an Advanced Course. *Lecture Notes in Computer Science*, (184), Springer-Verlag, pp 361-395, 1983.

[13] D. Powell. Delta 4: A Generic Architecture for Dependable Distributed Computing. *Research Reports ESPRITS, Project 818/2252 Delta 4* Vol. 1, Springer-Verlag Edition, 1992.

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

[15] R. D. Schlichting and F. B. Schneider. Fail Stop Processors: An Approach to Designing Distributed Computing Systems. *IEEE Transactions on Computer Systems*, 1(3):222-238, August 1983.

[16] 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.

[17] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communication of the ACM*, Vol 33, pp. 46-63, December 1990.

[18] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery. *Information Processing Letters*, 25:295-303, July1987.