In proc. of the First European Dependable Computing Conference, Berlin, Germany, October 1994,
Lecture Notes in Computer Science, 852 Springer-Verlag

GATOSTAR: A Fault Tolerant Load Sharing Facility for Parallel Applications

Bertil Folliot* Pierre Sens

MASI Lab / CNRS 818, IBP, University Paris VI
4 place Jussieu
75252 Paris Cedex 05
(France)

Abstract. This paper presents how and why to unify load sharing and fault tolerance facilities. A realization of a fault tolerant load sharing facility, GATOSTAR, is presented and discussed. It is based on the integration of two applications developed on top of Unix: GATOS and STAR. GATOS is a load sharing manager which automatically distributes parallel applications among heterogeneous hosts according to multicriteria allocation algorithms. STAR is a software fault tolerance manager which automatically recovers processes of faulty machines based on checkpointing and message logging. The main advantage of this approach is to increase fault tolerant performance by taking advantage of the load sharing policies when allocating or recovering processes. This unification not only improves the efficiency of both facilities but avoids many redundancies mechanisms between them. Indeed, each facility needs to manage at least three common features: global knowledge of the running processors, a crash detection mechanism and remote process management. The backbone of this unification is a logical ring of communication for host crash detection and for host related information transfer. Thus, all necessary information is acquired with a relatively low cost of messages compared to the two systems taken independently.

1 Introduction

Distributed systems based on local area networks provide new opportunities for developing high-performance parallel and distributed applications. However, because of the dependency of the components, such systems are particularly fragile: the failure of one component may result into the failure of the whole system. A failure is particularly inconvenient for complex long-lived applications. Thus, it is essential to have a *fault-tolerant* management. For portability and export reasons, we are only concerned with user software methods, avoiding all operating system modification and hardware support. Unfortunately, implementing fault tolerance via software is very expensive and it degrades application performance. To limit the fault tolerance cost, a straightforward approach is to use all the power of the distributed environment. Indeed, studies have shown that many hosts are unused or idle at a given time in local area networks of workstations. In such an environment,

* Author's affiliation: UFR d'Informatique, University Paris VII, Paris, France

workstations are free from 33% to 90% of the time [35]. By using a *load sharing* management, users' applications can take advantage of all the hosts' processing power and particularly of the idle ones [4, 26]. Thus, not only the application response time greatly decreases when there is no fault but also the software fault tolerant overhead becomes acceptable.

There are several common mechanisms in the implementation of a load sharing manager (LSM) and of a fault tolerant manager (FTM). The unification of the two systems is more than a simple cooperation between them. Indeed, each system needs to incorporate two main common features. The first one is "membership management" which consists of the global knowledge of running processors and of crash detection mechanisms. The global knowledge of the running processors is needed to maintain a global coherency for the FTM and to allocate processes to the right processors for the LSM. Furthermore, LSM needs some additional information, such as the processors' speed, the hosts' load and the amount of available memory. The crash detection mechanism needs not be efficient for the LSM, and is often very basic (as in DAWGS [11] and Utopia [37]), but is crucial for the FTM. It must be very efficient, in order to react to failures as fast as possible, while its implementation must not overload the network or the workstations.

The last feature concerns distributed process management. This management includes transparent remote execution, when allocating a processor and when restarting a process. Transparency is an important attribute to allow the execution of interactive programs and to not limit the facility only to batch programs. For the LSM, process management should also include a dynamic process migration mechanism to react to unexpected load variation and to let owners of formerly idle workstations repossess their computing power. The migration mechanism is similar to the FTM checkpoint/rollback mechanism. Finally, both FTM and LSM must provide a global naming space in order to support remote communication and remote file access.

A solution that suppresses those redundancies and increases software fault tolerance performance is to *unify* the load sharing and the fault tolerance facilities into a single system. Such an approach is rarely found despite the similarities. In this paper, we present, GATOSTAR, an experience in load sharing and fault tolerance unification. Our main goal is to minimize the cost of both systems by taking advantage of each others' capabilities and by reducing the number of messages exchanged in normal functioning.

GATOSTAR is based on two independent systems realized at the MASI lab: GATOS and STAR. GATOS is a load sharing manager which *automatically distributes parallel applications* among heterogeneous hosts [18, 19]. STAR is a fault tolerance manager which *automatically recovers processes of faulty machines* based on message logging [31]. They are implemented on a set of Sun3 and Sun4 workstations connected by a LAN (Ethernet) and they run on top of Unix (SunOS). The main common properties of STAR and GATOS are:

- *portability*: they were designed and implemented as software layers, without operating system modification,
- *transparency*: users have only to relink with special libraries,

- *parallel application support*: the domain of applications is not limited to independent batch programs.

The paper is organized as follow. Section 2 presents the related work in fault tolerant and load sharing management. We give in Section 3 the application and environment models supported by GATOSTAR. We describe in Section 4 the GATOSTAR failure management and in Section 5 the GATOSTAR load sharing management. Section 6 gives an overview of the prototype implementation, and Section 7 presents some performance measurements of the prototype in a real environment. We conclude in Section 8.

2 Related Work

In this brief survey of the literature, we present first, work related to parallel application placement in local area networks, and then work on fault tolerance by means of checkpointing. Finally, we describe four existing unified load sharing and fault tolerant facilities.

2.1 Parallel application placement in LAN

Parallel program placement has been widely studied. Earlier works concentrated on static allocation, adapted for multiprocessor systems without interference between applications' components [27, 33] or adapted for specific given applications [24, 29]. A well-known parallel application manager for local area networks of workstations is the PVM system [20]. It uses a user-managed host file indicating a list of hosts to run an application. This scheme is adequate for a dedicated environment, but may lead to over-utilization of resources if different users use overlapping sets of hosts. It may also lead to unacceptable interference with hosts supporting interactive users.

Most of the previous works on load sharing deal only with processor load and are not adapted for the allocation of entire parallel applications allocation. They are divided in two groups, dynamic placement (as Radio [4], REM [32], Utopia [37]) where processes are allocated at start-up and stay on the same location, or dynamic process migration (as Condor [26], Mosix [3], Sprite [13]) where processes can move according to overload conditions or the reactivation of workstations by their owners. Utopia is the closest to GATOS. It is based on previous simulation experiments of Ferrari and Zhou [16] about load thresholds values. This system runs on heterogeneous operating systems and applies load sharing. As in GATOS, it takes into account program' needs (processing power, amount of memory) and the heterogeneous environment, but it manages only independent processes without communication facility.

2.2 Fault tolerance by means of checkpointing

Related work in checkpointing is divided in two categories: *consistent* and *independent* checkpointing. With consistent checkpointing, processes coordinate their checkpointing actions such that the collection of checkpoints represents a consistent state of the whole system [10]. When a failure occurs, the system restarts from these checkpoints. According to the results presented in [6] and [15], the main drawback of this approach is that the 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 [25].

In independent checkpointing, each process saves its state without external synchronization. This technique is simple, but since the set of checkpoints may not define a consistent global state, the failure of one process leads to the rollback of other processes. Reliable message logging avoids this classical domino effect.

A substantial body of work has been published regarding fault tolerance by means of checkpointing. The main issues that have been covered are reducing the number of messages required to synchronize a checkpoint [6, 15, 36], limiting the number of hosts that have to participate in taking the checkpoint or in rolling back [22, 25], or using message logging [8, 23, 34].

2.3 Existing load sharing fault tolerant facility

Few experiences have been reported on load sharing systems with a fault tolerance mechanism. The major ones are the systems REM (Remote Execution Manager) [32], Paralex [2], Condor [26], and DAWGS (Distributed Automated Workload sharing System) [11]. These four systems have been developed essentially as load sharing manager, since fault tolerance is a limited extension. To our knowledge, no work has compared and unified the two concepts.

REM and Paralex are load sharing managers that support cooperative applications. REM provides a mechanism by which processes may create, communicate with, and terminate child processes on remote workstations. Active process replication is used to ensure the fault tolerant execution of the child processes. Thus, the fault tolerant property is gained at the expense of computing power. Moreover, failure of the local user's machine implies a failure of the whole computation. As the failure rate in a local area network is low [11], the process replication mechanism is not adapted to a load sharing scheme. Furthermore, the target applications of software fault tolerance are long-lived ones, and users are not supposed to always be connected with the same local machine. Paralex provides a package for distributing cooperative computation across a network. It uses passive process replication to achieve both fault tolerance and load balancing. Since process migration can only be done between replicas, the load sharing capability is strongly limited.

Condor and DAWGS rely on the same principle as GATOSTAR. Processes are initially allocated to an idle host, and a checkpoint/rollback mechanism allows the migration of processes and to restart processes after a host failure. In both systems, migration is essentially used to respect the privacy of workstation owners. The main drawback of Condor is that it only deals with independent processes since Condor processes cannot communicate or synchronize. DAWGS is a complete load sharing fault tolerant facility, including I/O redirection. The two main differences between DAWGS and GATOSTAR are that (1) DAWGS relies on kernel modification and (2) DAWGS fault-tolerant part is not configurable and a process running on a faulty host must wait for the host to restart (there is no easy way to replicate a checkpoint file). The kernel modifications allow full user-transparency without the need to link with a special library, but it highly limits the portability of such a system (and thus, of the supported applications).

3 Environment and Application Model

3.1 System environment

GATOSTAR relies on a BSD Unix operating system (SunOS). It works on a set of workstations connected by a local area network. In this environment, failures are uncommon events. Clark and McMillin measured the average crash time in a local area network to be once every 2.7 days [11]. Moreover, most of the Unix systems do not allow real time programming and thus, very short recovery delays can not be provided. Therefore, we favor solutions with a low overhead under normal operation, possibly to the detriment of an increase in recovery time.

We consider that the system is composed of fail-silent processors [28], where the faulty nodes simply stop and the remote nodes are not notified. In this first version, we do not consider the network partition problem.

3.2 Application model

We have adopted the classic application model where an application is a set of communicating processes connected by a *precedence graph*. The graph contains all qualitative information needed to execute the application and helpful for the load sharing manager: precedences and communications between processes, and accessed files. This information can be provided by the application programmer both by an execution graph (a file containing the graph description using a parallel application grammar), and/or by dynamic operations that allow, while running, to modify the execution graph.

We are interested in applications running for extended periods of time such as high number factoring, VLSI applications, image processing, etc. This kind of application may be executing for hours, days or weeks. In that case, the failure probability becomes significant, and load sharing and the need for reliability are important concerns.

For users performance requirements, all processes of an application do not have to be fault-tolerant. In that case, the application designer specifies for each process if it must be fault-tolerant or not. All fault tolerant processes are deterministic and exchange information through message passing. The state of a process is determined by its starting state and by the sequence of messages it has received.

This application model is well adapted to fault tolerance and load sharing support. All GATOSTAR processes are initially allocated on unused hosts. The fault-tolerant GATOSTAR processes not only support hosts failures, but they can also be migrated when host load or process behavior conditions change.

4 Failure Management

In this section, we describe the fault-tolerant facility of GATOSTAR: firstly, the host crash detection mechanism, and then the process recovery and message logging management. Finally, the reliable file management is presented.

4.1 Host crash detection: the logical ring

A classical method for software host crash detection is to wait for a normal host access failure. This detection method has no overhead but the treatment of the failure 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 processing of failure recovery.

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 a non-negligible overhead in the network traffic depending on the frequency of the check.

In GATOSTAR, we use a combination of the two previous methods. The normal traffic is used as in the first method, but in addition, when there is no traffic between two hosts during a given time slice, we generate a specific detection message. A basic method is, 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 structure hosts in a *logical ring of detection*. Periodically, each host *only* checks its immediate successor in the ring. The checking process is straightforward and the cost in messages is relatively low. Furthermore, this ring and the detection messages are used to transfer host load information between hosts (see Section 6).

In this method, a ring reconfiguration protocol is executed when adding or removing a host. 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 host identification.

4.2 Process recovery

A process recovery is basic if the process has no interaction with any other and does not access files. In that case, it can independently be recovered. Difficulties arise when a process communicates with others and/or accesses files. The recovery method uses checkpoint/rollback management [25]. By checkpointing, user's processes save their states to survive their local host failures. We adopt independent checkpointing with pessimistic message logging. This technique is tailored to applications consisting of processes exchanging small streams of data. The cost of logging is proportional to the number of messages (see Section 7). This method has the following advantages:

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

The benefits listed above are obtained at the expense of the space that is required for storing the message logs and the time it takes to log messages. The space overhead is reasonable given large disk capacities. Each checkpoint is replicated on several hosts. The default number of replicas is given by the network administrator. This number depends on the probability of simultaneous host crashes in the system and may be changed according to programmers' needs. We show in the performance section that the time of message logging depends linearly on the replication degree. A message sent and saved on two different disks takes about twice the time of a normal send (i.e., with one backup).

Failure recovery is divided into four steps: identification of the faulty processes (processes which were running on the faulty host), selection of a set of valid hosts using allocation algorithms (see Section 5), creation of processes on these hosts, and finally, restoration of their states using checkpoints.

4.3 Reliable file management

The file manager is a key feature in a fault-tolerant system. In GATOSTAR, it allows file accesses, message backups and checkpoint storage. Our file manager has the following features:

- *Fault tolerance*: each file is replicated on separate disks, with the number of replicate copies being maintained in case of failure (obviously, only if the number of remaining disks is sufficient).
- *Consistency*: file copies are kept identical.
- *Version management*: each used file has an old version (also replicated) corresponding to the last checkpoint of the process.

If there are N copies of a file, then N-1 simultaneous failures are tolerated. Because failures are uncommon events, only a small number of copies is necessary (about 2 or 3). This number is set by the network administrator or by the application designer according to the fault-tolerance and performance requirements. To ensure coherence of all copies, the file manager performs a reliable broadcast protocol [7]. A file update is reliably broadcast to all managers having a copy. A read operation is locally done whenever possible.

The file manager also manages versions of replicated files. Since each process needs only one checkpoint, only one old (replicated) version is needed for each file in use. When a process rolls back, old versions of files in use replace current ones. Shared files management has not been considered. Such management is complex, and requires a distributed database manager [5]. Another extension would be to integrate an existing and open log manager, as in Camelot [12], Clio [17] or KitLog [30].

5 Process Allocation

The process allocation in GATOSTAR is not limited to independent programs. It is mostly geared towards parallel applications and takes into account processes' behavior. We present in this section an overview of a multicriteria load sharing

algorithm and how these algorithms can be adapted to react to unexpected load variation by means of a dynamic migration mechanism.

5.1 Process placement

To launch new processes and to restart processes of faulty hosts, GATOSTAR uses the GATOS load sharing manager [18, 19]. This manager automatically distributes parallel applications among heterogeneous hosts. To be executed, application processes need resources (CPU, memory, file access, communication) that vary from one process to another. To evaluate the need of each process, we keep track transparently of previous executions and/or we allow application designers to give appropriate indications.

In order to improve the overall system performance, we operate dynamic process allocation according to the following criteria: host's load, process execution time, required memory, and communication between processes. In the application description, the programmer can specify appropriate allocation criteria for each program according to its execution needs. He specifies the set of hosts involved in the load distribution and provides different versions for each program in order to deal with system heterogeneity.

The following section describes the algorithm for the first of the criteria. For a complete description, see [9, 19].

Selecting the Less Loaded Host. Allocating processes according to the hosts' load allows to benefit from the idle workstations power. To take into account the heterogeneous character of system hosts, the load of a host is generally considered as directly proportional to CPU utilization and inversely proportional to its CPU speed [37]. On each host, the load sharing manager locally computes an average load and uses failure detection messages to exchange load information. Thus, no extra communication is needed when allocating programs. For a given program, the selection of a host is carried out, as follows:

- For every version of the program, compiled for a different CPU architecture, the host with the lightest load among the ones belonging to the compatible class is selected. This gives the list of the best hosts.
- Then, the host (say H) with the lightest load is selected in that list, and the corresponding version of the program is retained. If two (or more) hosts have the lightest load, the one with the highest processor speed is chosen.

The selected version is executed on H if H's load is below the *overload threshold*. Otherwise, the program is run on the originating host as all hosts in the system are heavily loaded. The main advantages of using a threshold are:

- A user working on a loaded host will not have an extra load originated by other users.
- Processes are not placed on remote hosts when all hosts are heavily loaded. In such a situation, running processes on remote hosts would be even more expensive than running them on their local one.

The main difficulty is how to find a good overload threshold value. A number of simulation studies on load sharing have been performed. Eager et al. [14] show that the overload threshold depends on the average load of the system and is between 1 and 2. For Alonso and Cova [1] this threshold depends on the number of users and is between 0.4 and 1.6. Bernard and Simatic have chosen the value of 0.8 for their Radio system [4]. These results and our experience with the GATOS system within the MASI lab. lead us to set the overload threshold value to 1.

Another problem is to get global state information. Indeed, every host cannot have exact knowledge about the load of the other hosts: first because transmitting this information over the network takes time, and secondly there is a delay between the instant when a host receives the information and the instant it uses the information to decide where to place a process. Large load fluctuations can occur unexpectedly. If the delays mentioned above are significant, a program can be placed on a host which was lightly loaded at the placement time but has become heavily loaded. This problem is solved by re-placing the program, if the load of the selected host is higher than the overload threshold. The load fluctuations coming after the initial placement can only be solved by migrating a sub-set of processes (see next section).

5.2 Process migration

Process migration is complex and difficult to realize. Initially, GATOS did not incorporate such a scheme. The GATOSTAR system incorporates migration by means of the checkpoint/restart mechanism developed for STAR.

Fault-tolerant processes of GATOSTAR are linked with a special library (see Section 6.1) that contains software code to checkpoint and to restart processes. When GATOSTAR wants to migrate a process, it sends a signal to the process, caught by the GATOSTAR library code that generates a checkpoint. Then, GATOSTAR stops the process and restarts it on the selected host.

The already existing load sharing algorithms, described in the previous sections, will be extended to process migration according to the applications and environment evolution. Environment evolution results from foreign activities: a user logs in or sends a remote job to an already loaded host. Application evolution stems from variations in resource usage (processor occupation, communication between processes and memory demands). Since mechanisms to monitor such variations were already integrated in GATOS, the adaptation of the multicriteria load sharing algorithm should not be very difficult. The next section describes the extension of the first multicriteria algorithm to a dynamic scheme.

Migration according to host load. The three main questions that must be answered when migrating processes dynamically are: when to move a process, which process to choose, and where to migrate the chosen process. Our proposed answer to the first and the third questions is based on two opposite load thresholds: a *migration* and a *reception threshold*. When at least one local fault tolerant process is running, the GATOSTAR server periodically computes two conditions (1) a faster workstation load is under the reception threshold, (2) the local host load is above the migration threshold and at least one workstation load is under the reception threshold (remember that the load is inversely proportional to CPU speed). The first condition

allows the potential use of fast available hosts, and the second one allows a decrease in the overload of the local station. If one or both conditions are true, the server enters the process selection step.

In the process selection step, the local server chooses a fault-tolerant GATOSTAR process that has been executed locally for at least a given *local execution time*. This minimum required execution time prevents a process from spending all its time in migration from one host to another. If there is more than one process that can be migrated, execution graph information (containing programs description) is used. The selected process is the one that asks for CPU time and that is supposed to have the longest remaining execution time.

Like in case of the overload threshold (the one used when a host is initially chosen), "good" values must be set for the migration and reception thresholds. Alonso and Cova [1] and Bernard and Simatic [4] empirically give the same value to the migration threshold, respectively 1.75 and 1.8, while Harbus [21] uses values between 2.5 and 5! This threshold seems to depend on the behavior of the migrated processes, on the ratio (available processing power / mean number of processes), and on user's behavior. There are few studies on the reception threshold. Most of the time, it is assimilated with the overload threshold. Further experiments will help us to choose appropriate values.

6 GATOSTAR Implementation

This section describes the implementation of the unified prototype of GATOSTAR. First we describe the interface, then the current implementation and finally the evolution of the GATOS and STAR systems.

6.1 GATOSTAR interface

The GATOSTAR interface is composed of commands (start/stop/visualize applications…), and functions stored in the GATOSTAR library.

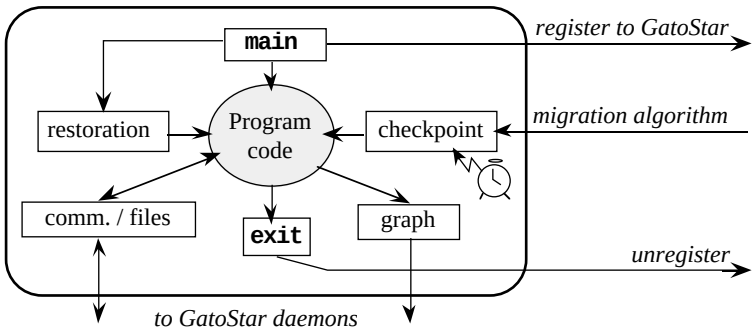

Fig.1. Process organization

Each program must be linked to the GATOSTAR library. This library contains the following functions (Figure 1):

- "**main**" and "**exit**": at start-up, the **main** function registers the new process to GATOSTAR. Similarly at the end, the **exit** function signals GATOSTAR.
- Checkpoint and restoration: the **checkpoint** function is either called periodically, or explicitly indicated in the source code (by using **s_checkpoint**), or called by the GATOSTAR migration algorithm. When a process is relaunched, the **restore** function is automatically called from **main**.
- File access functions: which provide a Unix-like interface to the GATOSTAR file manager.
- Communication functions: which allow reliable message exchanges.
- Graph functions: which allow modification of the execution graph (program creation/destruction).

6.2 GATOSTAR architecture

The current prototype is composed of three daemons running on each host: a *load sharing manager* (LSM, part of GATOS), a *fault tolerance manager* (FTM, part of STAR) and a *ring manager* (RM). The three daemons locally communicate through a *local shared memory* (Figure 2).

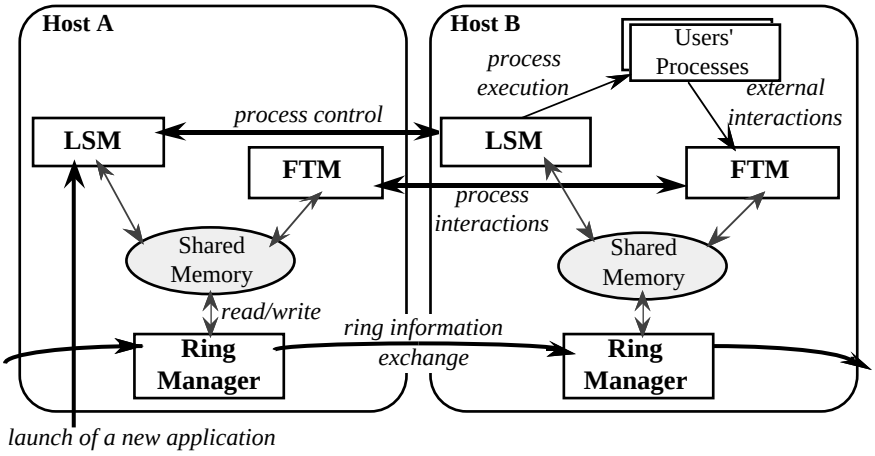

Fig. 2. GATOSTAR architecture

The LSM is responsible for process allocation and for local load update. LSMs communicate with each other to exchange process control information, such as remote execution of a program or remote termination. The FTM is responsible for recovering faulty processes and for process location management. Users' processes call their local FTM when communicating or accessing a file. Process interaction orders are routed through FTM servers. The RM is responsible for checking its logical ring successor and for collecting load information. It updates a local copy of the active hosts with all information needed by the LSM. The list of hosts and the set

of GATOSTAR processes are in local shared memory and accessible by the three daemons. Its global coherence is maintained by the set of RMs.

For example, consider a user starting a parallel application on host A (Figure 2). The host A LSM selects some host (host B in this example) and sends a remote execution request. Then, Host B LSM starts the processes. Each newly created process registers itself to the local FTM that reliably broadcasts this information to all the FTMs. On reception of such a message, each FTM updates the local shared memory.

Information update. Periodically each RM sends a detection message to its successor. Each LSM periodically computes the local load information and writes it in the local memory. This information indicates the load, the number of active processes, the memory used and the disk activity. The RM piggybacks load information in failure detection messages, and updates the local memory (except for the local host) when it receives a message from its predecessor. Local memory information is also updated when receiving processes messages with piggybacked information.

Actions on host crash. When a failure occurs, the ring manager of the predecessor host detects it (the connection is down), and initiates the recovery in three steps:

(i) The RM broadcasts the host ID of the faulty host. On reception of a failure message, each RM updates its local memory.

(ii) The RM updates the ring configuration.

(iii) The RM signals the local FTM (by a local signal). On reception of the signal, the FTM finds out in the shared memory the processes that were running on the faulty host and asks the local LSM to recover them.

6.3 From GATOS and STAR to GATOSTAR

The membership management of GATOS was redundant with the global knowledge and crash detection mechanisms of STAR. It has been suppressed, and the host related information is now included in detection messages. All global information needed by GATOS and STAR is now unified and stored in local memory, and a local protocol has been defined to access these data. In STAR, processes were relaunched on the detecting host. This simplistic allocation strategy has been replaced by a call to the LSM.

GATOSTAR includes a new process allocation algorithm based on the migration mechanism provided by STAR. The GATOSTAR user's interface is composed of all STAR functions (process control, checkpoint/restoration, communication and file access) and the graph functions of GATOS (qualitative and quantitative descriptions).

The main unification problem is that GATOS is not designed to be fault-tolerant. Indeed, each application is managed by a dynamically chosen GATOS server (usually the local one) which centralizes all application information. If a GATOS server fails, its managed applications descriptions are lost. The STAR servers have no such problem as they have global knowledge about the running processes. To give to each GATOS server the fault-tolerance property, two obvious mechanisms can be used: (1) to replicate centralized information between several servers or (2) to store this

information in reliable storage (by means of the STAR replicated file management). This will be taken into account in the next version of GATOSTAR.

7 Performance

We present in this section performance measurements of the process allocation strategy and of the failure management mechanism. These measurements were done in a real system within MASI lab (research/university). The supporting environment is composed of a set of Sun workstations connected by a LAN (Ethernet). The workstations are Sparc station 1's with 16 megabytes of memory.

Process allocation. The following measures give the process allocation cost:

- The overhead to allocate a program at start-up is less than 100 milliseconds.
- Consider five sites and an application composed of four parallel programs. On average, a classical allocation algorithm (taking into account just the hosts' loads) divides by two the response time compared to a local execution. The minimum response time is close to the optimum (a speed-up of 3.9 on four hosts).
- Multicriteria allocation taking into account not only the host load but also the application's expected execution time decreases the response time from 14% to 20% compared to classical algorithms. Figure 3 gives execution speed up according to application complexity for a system composed of three hosts. For example, application A (2.10M 1.50M), means 2 programs performing 10 millions iterations in parallel with one program performing 50 millions iterations.

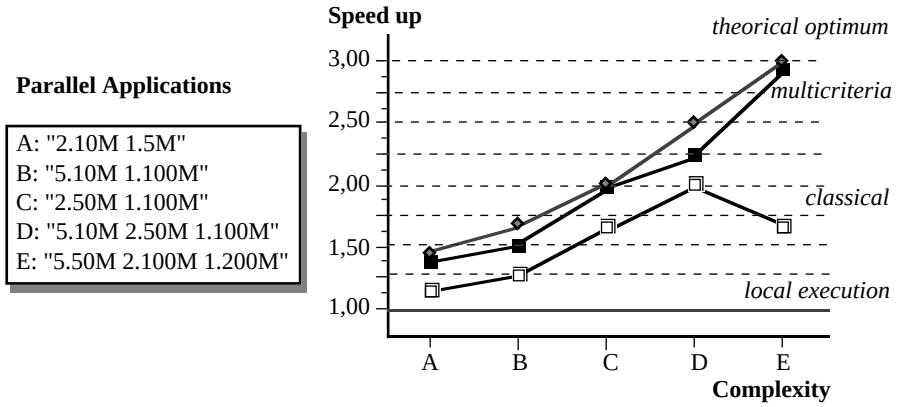

Fig. 3. Speed up according to application complexity

Message logging. We have evaluated the cost of the STAR communication protocol according to the replication degree of the log. 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.

Checkpoint. We present in Figure 4 the checkpoint cost according to different replication degree of the checkpoint file. A checkpoint for a process of 100 kilobytes and for a replication degree of two is done in 1.4 seconds. For a process of the same size and a replication degree of four, the cost is 2.7 seconds.

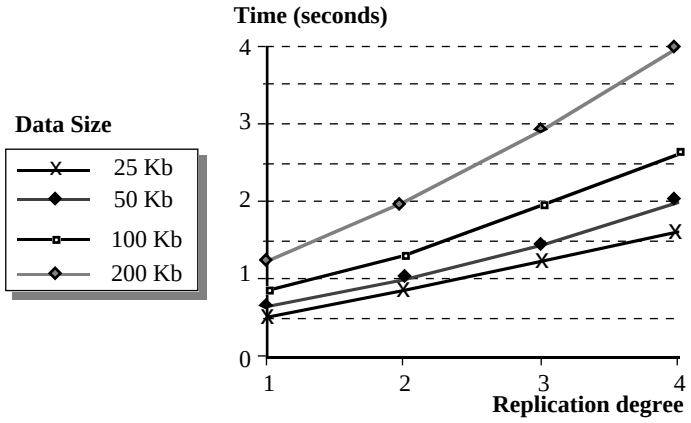

Fig. 4. Checkpoint cost

The previous measures show that the total cost of our fault management is low for the specific studied applications, i.e. long-running ones with small message exchanges.

7.1 Performance comparison with other systems

The mean time for a placement request is 5 seconds in REM [32], 500 ms in Radio [4], and 12 seconds in DAWGS [11]. These high allocation times are essentially due to highly communicating algorithms, while GATOS (recall its 100 ms allocation time) does not need any extra communication when allocating hosts.

The checkpoint time in DAWGS is quite high: 1.84 seconds for a 25 Kbytes process. It manages only a replication of degree one and, furthermore, processes running on a faulty host must wait for the host to be restarted before being recovered.

Elnozahy et al. [15] have implemented a consistent checkpointing on an Ethernet network of Sun 3/60 workstations with only replication of degree one. Their performance results depend on 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. For a complex application, such as a computation on a grid, the checkpoint time is about 10 seconds. In consistent checkpointing, the cost of the checkpoint depends not only on the size of the data but also on the synchronization between involved hosts. To reduce the checkpoint overhead, the authors recommend a copy-on-write method. This technique implies a consequent increase in performance. With a 2 minutes checkpoint interval, their checkpointing increased the running time of an application by about 1%. The worst overhead measured was 5.8 %.

Bhargava et al. [6] give some performance measurements of consistent checkpointing. In their environment, the messages needed for synchronizing a checkpoint implied an important overhead. The authors have limited their study to small size programs (4 to 48 kilobytes).

Borg et al. [8] 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. According to the authors, the performance turns out to be 1.6 times slower than on a standard Unix. The recovery time for a process is 5-15 seconds.

8 Conclusion

This paper has described the advantages of fault tolerance and load sharing unification. This unification improves parallel applications response time, and decreases the overhead of the software fault tolerance facility and of the overall system utilization by eliminating redundant algorithms. Our system takes advantage of the overall system resources and allows the distribution of the applications processes on the most appropriate hosts in the network. Our process allocation policy takes into account both information about the system state and resource needs for the execution of applications. A checkpoint mechanism allows the recovery of processes after hosts failures. The system consistency is maintained by a reliable message logging.

We have presented our experience in implementing GATOSTAR based on two existing packages GATOS and STAR. The logical ring used in STAR for fault tolerance recovery is also used in GATOS to get global knowledge of the system state. The principal advantages of GATOSTAR are the following:

- it provides programmers with the ability to build easily fault-tolerant applications with automatic load sharing,
- it reduces overhead in the load sharing and fault tolerance capabilities and particularly the cost in messages (compared to the two independent systems),
- the fault tolerance part uses dynamic load sharing to choose the “best” hosts when restarting faulty processes,
- the load sharing part uses detection messages of the logical ring to transmit load information, and chooses appropriate hosts when needed.

Measures of performances show that GATOSTAR improves applications response time and increases the availability of the network resources. Moreover, to improve the response time of long lived applications, an extension of the multicriteria allocation algorithm has been proposed and will be soon realized. It is based on the checkpoint/rollback mechanism to migrate process when the system load becomes particularly unbalanced.

In the future, our goal is for our system to go beyond the local area network limit. GATOSTAR is currently designed with a local area network in mind (up to 30 machines). Indeed, the logical ring limits the number of hosts. In larger distributed computer systems, including several connected LANs, the load acquisition time and

the overhead of information distribution will become too important. We are now studying a method to reduce information flow exchange between LANs.

Acknowledgments

We would like to thank Jean-Loup Baer for its comments on earlier drafts of this paper. We also wish to thank the referees for their suggestions.

References

1. R. Alonso and L.L. Cova. Sharing Jobs Among Independently Owned Processors. In *Proc. of the 8th International Conference on Distributed Computing Systems*, San Jose, California, pp. 365-372, February 1988.

2. O. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems. In *Proc. of International Conference on Supercomputing*, Washington D.C., pp. 178-187, July 1992.

3. A. Barak and O.G. Paradise. MOS - A Load-balancing Unix. In *Proc. of the Usenix Technical Conference - Summer 1986*, pp. 414-418, 1986.

4. G. Bernard and M. Simatic. A Decentralized and Efficient Algorithm for Networks of Workstations. In *Proc. of the European Conference for Open Systems Spring '91*, Tromsø, Norway, pp. 139-148, May 1991.

5. P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185-221, June 1981.

6. 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.

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

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

9. 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.

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

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

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

13. F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757-785, 1991.

14. D. L. Eager, E. D. Lazoska, and J. Zahorjan. Adaptative Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE12(5):662-675, May 1986.

15. E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, October 92.

16. D. Ferrari and S. Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. *Performances '87*, Bruxelles, Belgium, pp. 515-528, December 1987.

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

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

19. B. Folliot. Méthodes et Outils de Partage de Charge pour la Conception et la Mise en Œuvre d'Applications dans les Systèmes Répartis Hétérogènes. *PhD Thesis, Research Report 93-27*, IBP, University Paris 6, France, December 1992.

20. G. A. Geist and V. S. Sunderam. Network Based Concurrent Computing on the PVM System. *Journal of Concurrency: Practice and Experience*, 4(4):293-311, June 1992.

21. R.S. Harbus. Dynamic Process Migration: To Migrate or Not To Migrate. *Technical Note* CSRI-42, University of Toronto, July 1986.

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

23. 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.

24. C. Kim and H. Kameda. Optimal Static Load Balancing of Multi-class Jobs in a Distributed Computer System. In *Proc. of the 10th International Conference on Distributed Computing Systems*, Paris, France, pp. 562-569, May 1990.

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

26. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference on Distributed Computing Systems*, San José, California, pp. 104-111, January 1988.

27. V.M. Lo. Task Assignement to Minimize Completion Time. In *Proc. of the 5th International Conference on Distributed Computing Systems*, pp. 239-336, 1985.

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

29. C.C. Price and S. Krishnaprasad. Software Allocation Models for Distributed Computing Systems. In *Proc. of the 4th International Conference on Distributed Computing Systems*, San Fransisco, pp. 40-48, May 1984.

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

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

32. G. C. Shoja, G. Clarke, and T. Taylor. REM: A Distributed Facility For Utilizing Idle Processing Power of Workstations. In *Proc. of the IFIP Conference on Distributed Processing*, Amsterdam, October 1987.

33. H. S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85-93, January 1977.

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

35. M. M. Theimer and K. A. Lantz. Finding Idle Machines in a Workstation-based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444-1458, November 1989.

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

37. S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Technical Report* 257, Computer Systems Research Institute, Toronto University, Canada, April 1992.