Appears in "Software - Practice and Experience", Vol. 28(10), 1079-1099 (AUGUST 1998)

The STAR Fault Manager for Distributed Operating Environments

Design, Implementation, and Performance

PIERRE SENS

University Paris VI, LIP6 Laboratory / CNRS, 4 place Jussieu, 75252 Paris Cedex 05 - France
(Pierre.Sens@lip6.fr)

AND

BERTIL FOLLIOT

University Paris VII, LIP6 Laboratory / CNRS, 4 place Jussieu, 75252 Paris Cedex 05 - France
(Bertil.Folliot@lip6.fr)

SUMMARY

This paper presents the design, implementation, and performance evaluation of a software fault manager for distributed applications. Dubbed STAR, it uses the natural redundancy existing in networks of workstations to offer a high level of fault tolerance. Fault management is transparent to the supported parallel applications. To improve the response time of fault-tolerant applications, STAR implements non-blocking and incremental checkpointing to perform an efficient backup of process state. Moreover, STAR is application independent, highly configurable. Star actually runs on top of SunOs and is easily portable to UNIX™-like operating systems. The current implementation is based on independent checkpointing and message logging. Measurements show the efficiency and the limits of this implementation. The challenge is to show that a software approach to fault tolerance can efficiently be implemented in a standard networked environment.

KEY WORDS Fault-tolerance Distributed Systems Independent Checkpointing Message Logging Performance

INTRODUCTION

This paper presents the design, implementation and performance evaluation of ***STAR***[30], a fault-tolerant facility to support execution of reliable distributed applications in a network of workstations. For instance, a parallel numerical application may support an arbitrary number of host failures during its execution.

Many systems integrate efficient resource management for distributed applications[3,14,36,41]. Several other systems offer fault management using the natural redundancy of the distributed system without requiring specific hardware support[1,4,22,24,32]. STAR fills this gap and is built totally outside the operating system.

The challenge is to show that software fault tolerance can be efficiently implemented in a standardized environment.

STAR uses checkpointing and rollback recovery. Such techniques are well-known to provide fault tolerance in distributed systems [8,19,20,23]. In the STAR implementation, an independent checkpointing mechanism is used to recover processes[7,16,31]. This implementation is based on a stable storage integrated in STAR. Each process independently saves its state and, when an error is detected, the execution is rolled back and resumed from earlier checkpoints. Because processes do not synchronize themselves for checkpointing, this method generally provides low run-time overhead. However, since the set of checkpoints may not define a consistent global state, the failure of one process may lead to the rollback of other processes (the well-known "domino effect" [10]). To avoid such an effect, our recovery protocol is based on message logging[7,33]. In the general approach, processes log their received messages. When a faulty process restarts, messages are replayed from the log in their original order to deterministically reconstruct its pre-failure state.

STAR fulfills the following goals. First, it allows users' applications to transparently tolerate an arbitrary number of host failures. Secondly, it is designed on top of the Unix operating system without need for any hardware support or kernel modification. Finally, STAR is application independent and flexible with tunable parameters so that each programmer can choose the recovery algorithm geared to his/her application's requirements. When the users starts his/her application, he/she can choose the appropriate message logging strategies (optimistic or pessimistic), he/she can also fix the checkpoint frequency and the replication degree of the stable storage.

STAR has been developed on a set of SUN-Sparc stations connected by Ethernet. The results demonstrate that independent checkpointing is an efficient approach for providing fault tolerance for the chosen applications, namely long-running ones with few message exchanges. We show that software based fault tolerance management is an interesting alternative to specialized hardware or kernel-integrated fault tolerance. Results from[24] as well as our own instrumentation of several parallel applications corroborate this claim.

The remainder of this paper is organized as follows. The next section presents the application, environment and failure models. The two sections following that describe the mechanism of failure detection and the process recovery strategy. We then present the message logging mechanism and our implementation of the stable storage. Finally we give the performance of STAR with several real applications in an academic environment.

ENVIRONMENT

This section presents our target environment and outlines some other systems providing fault tolerance in similar environments.

We consider an application as a dynamic set of communicating processes. A process may use any resource of the network (mostly CPU and files). The only way to exchange information between processes is through a message passing library provided by STAR. File sharing is not allowed. We also make the assumption that processes involved in the parallel computation are *piecewise deterministic*[35,39]; in other words

process execution is divided into a sequence of state intervals each of which is started by a non-deterministic event such as the receipt of a message. The execution within an interval is completely deterministic. In STAR, we assume that message-delivering events are the only source of non-determinism. The application has to handle other sources of non-determinism coming from the Unix system itself (for example due to time sharing) using local synchronization. This assumption is met by many applications, but excludes for example all programs relying on the local time values. To handle this kind of non-determinism, we can extend the message logging scheme by treating each non-deterministic function as a message, logging it and replaying it during recovery[16].

For each program, the user specify the initial execution host. To dynamically benefit of the available processing power, STAR has been integrated into the Gatos process allocation manager[17].

Users applications rely on a *fault-tolerant software layer* providing a reliable access to all external components (processes and files). This layer allows automatic recovery of processes affected by a host failure on any of the remaining valid and compatible hosts. The faults are totally hidden at the applications layer. In this sense, the fault tolerance mechanisms provided by STAR are transparent to the programmer apart from the need to link the final executable with the STAR library. STAR provides a *global naming space* for processes which is used to locate them. Each local fault-tolerant layer maintains a global view of the location of all current processes. We choose to limit migration on set of homogeneous hosts. Process migration among heterogeneous host is complex and appears to restrict significantly the permitted application behaviour[9,28].

STAR is implemented on top of a Unix operating system (SunOs). It works on a set of workstations (hosts) connected by a local area network (Ethernet). We assume that the underlying transport layer provides reliable, sequenced point-to-point communication (TCP). We make the following assumptions about the failure model. The processors are assumed to be fail-silent[25]: a failed node simply stops and all the processes on the node die. Viewed from the communication network, a faulty processor remains silent and cannot receive or send a message. However, a guaranteeing that nodes are fail-silent implies that the nodes are implemented with a perfect self-checking mechanism[25]. Although many techniques are available implementing self-checking hardware[37], at present we do not assume any specific hardware to achieve the fail-silence property. We also consider that failures are uncommon events and so very short recovery delays are not required. Therefore, we favour solutions with a low overhead under normal operation, possibly to the detriment of an increase in recovery time. At present, we make no attempt to detect individual process failures on a node. Future versions of the STAR software will handle finer-grained failures.

STAR consists of a set of servers and a client library (Figure 1). There are three main servers: the *failure manager* in charge of failure detection and restarting processes affected by host failures, the *file server* implementing the stable storage by means of replicated files, and the *communication server* managing interactions between application processes. Each part of the implementation is explained in the following related sections.

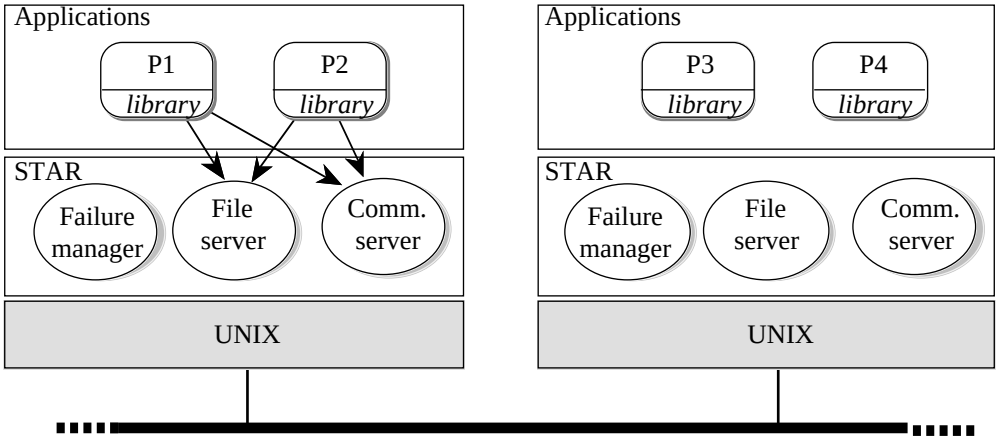

Figure1.STAR architecture

The STAR library offers a RPC-like interface for communication and a standard Unix interface for file access. Programs must use this interface for communication. Other Unix communication functions such as signals and pipes are not supported by STAR. Consequently, portions of code using these functions must be rewritten to use our library communication interface. The STAR library offers the following functions (Figure 2):

- Process creation and termination: at start-up, the **`main`** function registers the new process with STAR. Similarly at the end, the **`exit`** function unregisters the process with STAR.
- Checkpoint and restoration: the **`checkpoint`** function is either implicitly (at regular interval transparently to the application), or explicitly called as indicated in the source code. When a process is restarted, the restore function is automatically called from the **`main`** function.
- File access functions: these functions provide a Unix-like interface to the STAR file manager.
- Communication functions: these functions allow reliable message exchanges implementing message logging strategies.

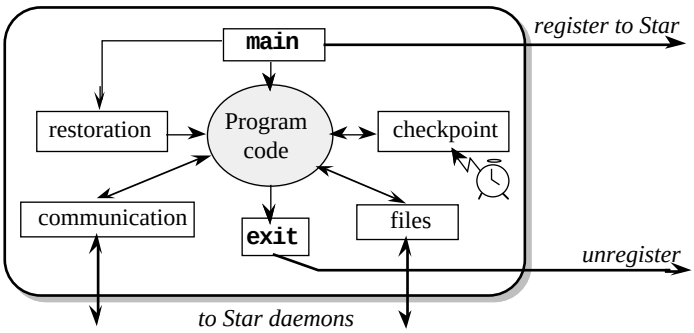

Figure2. Process organization

While STAR can be used in an almost transparent mode (the user has only to define the checkpoint period when he/she starts the application), it also offers the possibility to the user to explicitly include checkpoints supports in the source code. This user-triggered approach allows the application designer to use his/her knowledge about the application to judiciously insert checkpoints (e.g. after performing a significant amount of communication in order to purge the log).

Some process managers already provide fault tolerance in similar environments. Condor[22] is a well-known process manager on top of Unix workstations. It provides a checkpointing facility that allows sequential programs to use the idle time of workstations. Alternatively, checkpoints can be used to tolerate host failures. Recently, an extension proposed by [27] adapts the Condor sequential checkpointing to parallel programs. Libckpt[24] is another portable checkpointing tool for Unix that implements classical performance optimizations such as non-blocking and incremental checkpointing. However, Libckpt is limited to sequential programs. These two projects provide efficient tools for designing reliable applications. However, recovery is not automatically invoked and the fault treatment is not transparent to the user, who must explicitly restart the application. Moreover, checkpoints are not replicated and applications cannot recover from the failure of the file server.

In STAR, we want to provide a self-contained fault manager. STAR processes transparently all the steps of the fault treatment: checkpointing, fault detection, and restarting from valid hosts. Moreover, we do not want to be dependent on a single specific host. For this reason, STAR uses a replicated file manager to implement the reliable storage of critical data, such as checkpoints and message logs.

FAILURE DETECTION

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

Another solution consists of periodically checking the hosts' states[7]. The recovery is invoked as soon as a host does not respond to the checker. This technique allows a fast recovery, but introduces an overhead in the network traffic. This overhead is proportional to the checking rate multiply by the number of checked hosts.

STAR uses a combination of the two methods. The normal traffic is used as in the first method, but in addition, when there is no traffic during a given period, a specific detection message is generated. A naive implementation of this detection would be for each host to check all the other active ones. This solution is not suitable for complex systems with many hosts, since the network would become rapidly overcrowded with detection messages. In order to get an efficient detection message traffic, we organize all the hosts in a *logical ring*. Periodically, each host *only* checks its immediate successor on the ring. The checking process is straightforward and the cost in messages is very low. For a two second detection time, the network overhead on a 10 Mb/s Ethernet is negligible. However, to ensure the coherence of the ring, a two-phase

reconfiguration protocol is executed when adding or removing a host. The cost of the reconfiguration protocol is not significant since host crashes are uncommon events.

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

The implementation of the logical ring is as follows. Each failure manager is linked to its predecessor and successor using the TCP communication protocol. Periodically, the server checks if a normal message has been received from its successor. If no message has been received, it sends a detection message to its successor. If this sending fails, the successor is considered faulty and the server initiates the recovery step*.

PROCESS RECOVERY

The recovery step is invoked as soon as a failure is detected. Processes affected by the failure are immediately restarted on a valid host, unlike some other fault managers where processes can only be restarted after the faulty host is rebooted (as in DAWGS[12] or Arjuna[32]). The process recovery in STAR is done by (1) checkpointing the process to stable storage, (2) restarting the process on a hardware compatible valid host, (3) redirecting communications to the new process location.

Checkpointing a single process

The checkpoint of a single process is a snapshot of the process's address space at a given time. Each checkpoint is saved on stable storage capable of surviving some given number of host failures. To reduce the cost of checkpointing, two complementary techniques can be applied: incremental, and non-blocking checkpointing[15,24].

Incremental methods reduce the amount of data that must be written. Only those pages of the address space modified since the last checkpoint are written to stable storage. The most efficient way to implement incremental checkpointing is to use internal operating system mechanisms such as memory page protection. Thus, the set of modified pages is determined using the dirty bit in each page table entry. This implementation reduces the cost of checkpointing significantly. However, it often must be integrated with the kernel.

Non-blocking methods allow the process to continue executing while its checkpoint is written to stable storage[15]. However, if the process modifies any of its pages during the checkpoint, the resulting checkpoint may not represent the real state of the process. The internal *copy-on-write* memory protection may be used to protect pages during the checkpoint. At the start of the checkpoint, the pages to be written are write-protected. After writing each page to stable storage, the checkpoint manager removes the protection from the page. If a process attempts to modify a protected page, the page is

* We consider that the probability of incorrect failure detection is negligible using TCP. This assumption seems realistic for local area network.

copied to a newly allocated page, and the protection of the original page is removed. The newly allocated page is not accessible to the process. It is used only by the checkpoint manager to finish the checkpoint. This technique improves significantly the response time, between 3.4 times and 4.7 times faster checkpointing according to experimental results presented by [15].

STAR's checkpoint mechanism uses both incremental and non-blocking checkpointing. The Unix **`fork()`** primitive provides exactly the mechanism needed to implement non-blocking checkpointing. The `fork` system call creates a new process with the same address space as the caller. When checkpointing, the STAR library forks a child process which performs its context backup while the parent process returns to executing the application. Many implementations of `fork` use a copy-on-write mechanism to optimize the copying of the parent's address space. To perform incremental checkpointing outside the kernel, each child process created at the checkpoint time compares its address space with the space of the child process created at the previous checkpoint. This comparison is performed through a `pipe` and we save only data that have been modified. We show in the performance section that these two techniques considerably reduce the cost of checkpointing compared with full blocking checkpointing. However, they require a larger amount of memory and result in increased multiprogramming. The default checkpoint strategy is the non-blocking and incremental. The user may choose an other checkpoint strategy at link time.

In terms of implementation, periodically or by explicitly calling the **`checkpoint`** function, a process saves its data segment, and stack. Then, a **`setjmp`** is executed to save the register contents. When a process is relaunched on a new host, it first allocates enough memory to contain the saved data and stack segments (a call to the **`sbrk`** function to expand the data segment and to the **`alloca`** function to expand the stack). Then, it loads segments from the checkpoint and finally it executes a **`longjmp`** to restore registers.

Recovery schemes for communicating processes

When processes exchange messages, the simple approach to recovery for independent processes is no longer adequate[21]. In particular, attempts by individual cooperating processes to achieve backward error recovery can result in the well-known domino effect[29]. To recover from a fault, the execution must be rolled back to a consistent state, but rolling back one process could result in an avalanche of rollbacks of other processes before a consistent state is found. For example, when the sender of a message is rolled back, the corresponding receiver is also rolled back. This effect is illustrated in Figure 3.

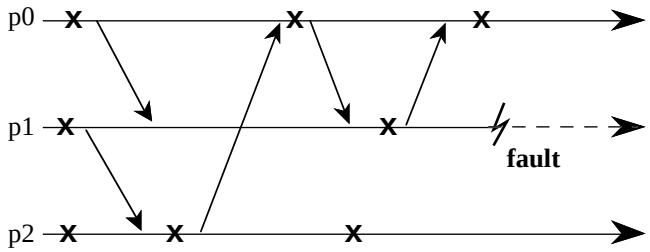

Figure3. Domino effect

The Xs indicate checkpoints and the arrows represent the messages. To recover from the failure of p1, all processes must be rolled back to their initial checkpoints.

A substantial body of work has been published regarding fault tolerance by means of checkpointing. The main issues that have been covered are limiting the number of hosts that have to participate in taking the checkpoint or in rolling back[20], reducing the number of messages required to synchronize a checkpoint[5,16], or using message logging[7,19]. Checkpointing techniques can be classified into two categories: coordinated and independent checkpointing.

With *coordinated 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. Looking at the results of [5,15,23], 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[20]. Alternatively, the number of processes to rollback can be reduced by analyzing the interactions between processes[20,13].

In *independent checkpointing*, each process independently saves its state without any synchronization with the others. *Message logging* was introduced to avoid the domino effect[7,16,18,35]. Logging methods fall into two classes: pessimistic and optimistic. In *pessimistic message logging* the system writes incoming messages to stable storage before delivering them to the application[31,40]. To restart, the failing process is rolled back to the last checkpoint and replies to outgoing messages are returned immediately from the log. The receiver is blocked until the message is logged on stable storage. Alternatively, the system delivers the message to the application immediately but disallows the sending of further messages until the message has been logged[18]. In *optimistic message logging* messages are tagged with information that allows the system to keep track of inter-process dependencies[16,19,33,34,38]. Inter-process dependency information is used to determine which processes need to be rolled back on restart. This approach reduces the logging overhead, but processes that survive a failure may still be rolled back. Alternatively, messages are gathered in the main memory of the sending host and are asynchronously saved to stable storage.

The implementation of STAR is based on independent checkpointing with pessimistic and optimistic message logging. This technique is tailored to applications consisting of processes exchanging few messages. This method totally suppresses the domino-effect and consequently only one checkpoint is needed for each process. We have implemented both pessimistic and optimistic message logging to allow application designers to choose the logging algorithm according to their application requirements. In every case, STAR guarantees that any process state is always recreated.

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

Interactive applications

Most fault-tolerant systems usually consider batch-only applications and do not manage interactive applications. In the Condor system, the application is killed when the user host crashes[22]. Alternatively, the interactivity can be managed in specific hardware architecture, as in the TARGON-32 system, where devices are duplicated[7].

An effort has been made to give STAR the capability to manage *interactive applications.* Each interactive application is associated with a terminal (a user window). The processes of such an application may be rolled back while the terminal is active. A specific process, the Input/Output server, is the link between the user terminal and the application processes. Each interactive process is linked to this I/O server to which it directs its standard input and output messages. Thus, programs outputs are transparently forwarded to the user window, and user inputs are sent to application programs (Figure 4).

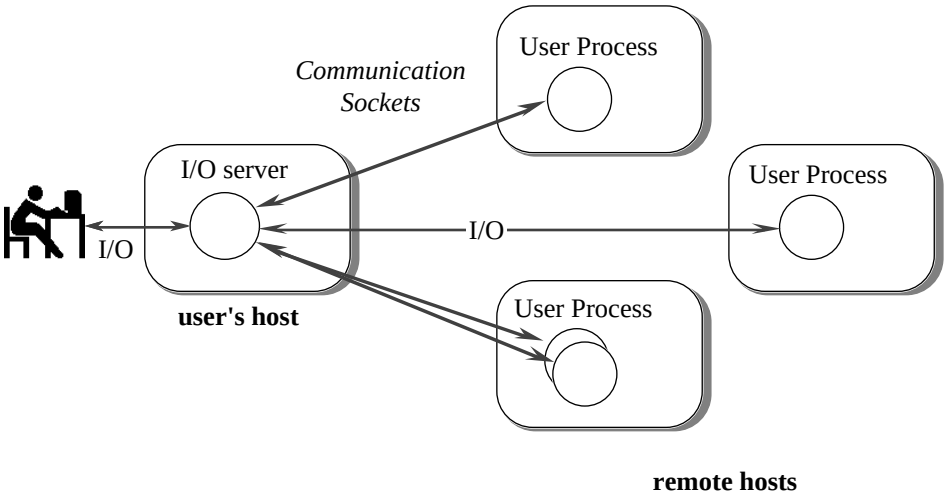

Figure 4. Management of interactive processes

If a remote host fails, the associated processes are restarted on valid hosts and their standard Input/Output are transparently reconnected to the I/O server. If the user host fails (i.e., the I/O server is no longer available), other processes continue their execution but are suspended when they try to access to their standard input/output devices. In such a case, STAR allows the user to login on another valid hosts. A new I/O server is then automatically created and reconnected to all the frozen processes that may then resume their executions.

COMMUNICATION MANAGEMENT

The STAR communication protocol relies on the confining principle: "*a recovered process has no interaction with the others until it reaches the last state before the failure*" and consequently avoids the domino effect. Communications between the checkpoint and the fault point are simulated using data saved in the stable storage.

Thus, any process may be rolled back independently. To comply with this principle, we use the following techniques:

1. Each process *saves all input messages*. A recovered process refers to this backup to access old messages. Thus, old valid senders are not concerned by the recovery of a process. All requests to receive messages are transparently transmitted to the local fault-tolerance layer. This layer directly accesses the backup or waits for messages according to the process state (recovered or not). At the process level there is no difference between receiving a message from the network or from the backup.
2. Because processes are piecewise deterministic, a recovered process resends all messages that have been sent since the last checkpoint. A timestamp on each message allows detection of these retransmissions. Each message has a unique timestamp and is retransmitted with the same timestamp in case of failure. The fault-tolerant layer detects retransmissions by comparing the timestamp of a message with the stamp of the last transmitted message in order to discard messages already received.

Each message is stamped with the global name of the sender and a sequence number. The sequence number is incremented at each send and is stored in the process' address space. The stamp uniqueness is ensured by the uniqueness of the global names. The current value of the sequence number is saved within each checkpoint. When a process recovers, the sequence number of its last checkpoint is restored and becomes the current one. Retransmissions are detected by comparing the current number of the sender to the number of its last message sent before the failure. This test is done, without any communication, in the local fault-tolerant layer of the sending host.

One difficulty resides in obtaining the number of the last message sent before the failure. It would be too costly to check for this number in all message logs. Thus, each host keeps in memory the set of the last received stamps (one stamp for each process). When a process is relaunched on a new host, all other hosts send the last stamp received from the recovered process. The number of the last sent message is the maximum of all the received numbers.

There remains the possibility that the maximum of all received numbers is not the real one. That happens if the host (say H_{max}) keeping the real maximum number is down and cannot answer. Then, some messages may be sent again to processes recovering from the H_{max} failure. These retransmitted messages will be detected and discarded by the communication server at the receiving host.

The following describes our implementation of message logging. Communications between hosts are always performed in a reliable way. Point-to-point communications are achieved through a reliable protocol (TCP). On each host, the communication server manages all users' processes communications. Each server uses local processes information for the location resolution and for the detection of retransmissions.

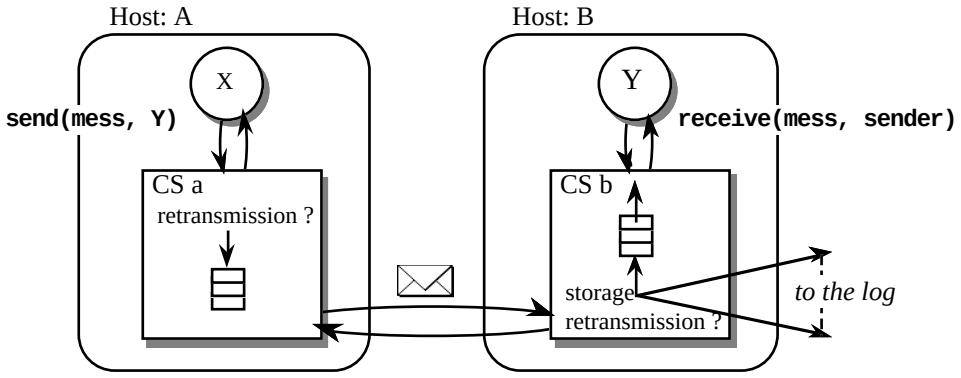

Figure5. Communication protocol

In the pessimistic scheme, the communication from process X on host A to process Y on host B is done in three steps (Figure 5):

1. X transparently calls its local communication server (CSa) with the name of the receiver and the message.
2. If the message is a retransmission (i.e., X is recovering), CSa discards it. Otherwise, CSa finds the location of process Y, based on its global knowledge about running processes, then puts the message in a local queue and sends it to the communication server of the remote host (CSb).
3. If the message received by CSb is a retransmission, CSb discards it. Otherwise, CSb saves it in the log and puts it in the receiver queue (Y), then it sends an acknowledgment to CSa that deletes the message from its sending queue. When Y asks to receive a message, it makes a request to its local communication server (CSb).

When host B fails, all new messages addressed to Y are locally queued at the sending hosts. When the new location of Y is known, all waiting messages are sent. They are deleted when the acknowledgments are received. If host B fails while logging a message, the message is not acknowledged and stays in the sending queue of host A. The latter is retransmitted when Y is recovered.

In the optimistic scheme, messages are not directly saved in stable storage but are kept in the main memory of the sending host. Periodically, the sending host saves all messages to stable storage. In case of failure, messages addressed to faulty processes are found either on the stable storage or on the main memory of the sending hosts. The receiving host only saves in stable storage the order information used to reorder messages in case of failure. The extreme case of optimistic logging is the sender-based algorithm, where all messages are kept in the sending queue and are never saved to stable storage. However, messages saved in volatile memory may be lost upon a failure, but since the processes are piecewise deterministic they resend their messages and the receiving host uses order information to deterministically reconstruct its pre-failure state.

The programmer can choose the message logging scheme (pessimistic or optimistic) when starting the application according to the application requirements. Moreover, STAR allows the disabling of fault-tolerant management to *temporarily suppress the message logging overhead*. Each application process may have two states: reliable or unreliable. In the unreliable state, a process will not be recovered if a failure occurs and

messages sent to an unreliable process are not saved. The programmer can change the process state to optimize part of the computation which has a high rate of communication. An application must explicitly handle failure of an unreliable process using **`checkpoint`** and **`restore`** functions provided by the STAR library. Such processes can be restarted from the beginning, or may coordinate their checkpoints before starting unreliable steps. For instance, consider the following application. At start-up, a master process sends a large matrix to some slave processes. In the reliable scheme, each slave process has a copy of the matrix in its address space *and* a copy in its log. To reduce the space and communication overhead, the application may be set as unreliable while the matrix is transferred. Then, processes may save coordinated checkpoints before starting the reliable computation step.

STABLE STORAGE

Stable storage is a key feature in a fault manager. The STAR file system (SFS) implements stable storage. It is used for file accesses, message backups and checkpoint storage. In several systems, the stable storage is based on specific hardware[2,26]. Some software fault managers use a reliable central file server[4,15]. Alternatively, to protect against a failure of the primary server, data can be duplicated on a backup server. In any case, this solution is not suitable for many environments, where there is no specific reliable host.

We developed a new file system which provides reliability using file replication. We do not use NFS for performance reasons (see Performance section) and because NFS does not provide replication. In SFS, each file is replicated on separate disks on different hosts. For each file, the user defines the initial number of copies (the default number is two). This number is maintained in case of failure (obviously, only if the number of remaining disks is sufficient). With N file copies, N-1 simultaneous failures are tolerated. Because failures are uncommon events, only a small number of copies is usually necessary (usually 2 for a network of 20 workstations). This number is set by the network administrator or by the application designer according to the fault tolerance and performance requirements. To ensure consistency of all copies, the file manager uses an atomic broadcast protocol[6]. A file update is broadcast to all managers that have a copy of it. Since files are not shared between processes, the broadcast protocol doesn't need to be fully ordered: it only assumes fifo channels. Read operations are completed locally whenever possible.

The reliable file manager is implemented as follows. A reliable file is composed of a set of standard UNIX files replicated on a set of disks. On each host where copies are present, a SFS file server manages accesses to copies. SFS daemons offer services for reading and writing files. When a file server host fails, the files are copied from a valid host to a new file server. This maintains the initial degree of replication.

The performance of STAR depends directly on the stable storage management. To provide an efficient replicated file access, we take advantage of the pseudo-parallelism offered by the underlying system: any access to a remote SFS server is achieved by a specific process: the SFS server proxy. One proxy is associated with each remote SFS server (Figure 6). A proxy is automatically created when a new SFS server is created. Local clients and proxies exchange information through a local shared memory

segment (SM). When a client wants to send a request to N servers, it puts the request arguments in the local memory and wakes up the proxies corresponding to the remote servers. The proxies read and transmit the request in a pseudo-parallel way.

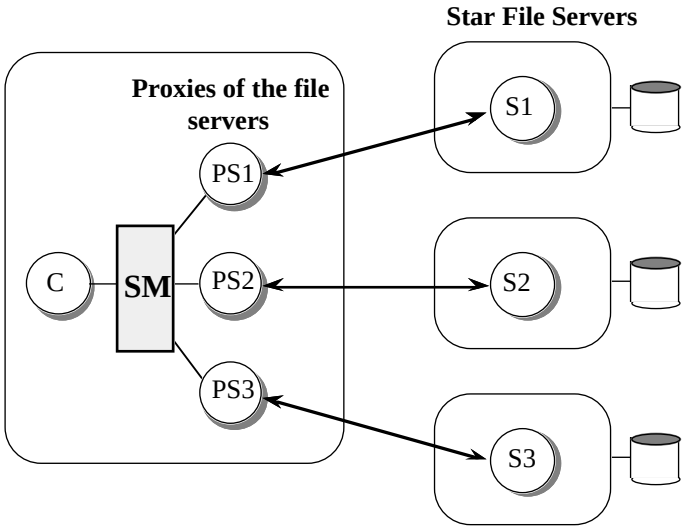

Figure 6. Communications with the file servers

To increase the performance of transfer between client and SFS server, we implemented a protocol specially optimized to deal with bursts of sending in order to transfer large volumes of data. The main performance problem of NFS is due to the use of a synchronous protocol to exchange information between client and server: because the NFS server is stateless, each write request must commit any modified data to stable storage before returning results. Such a protocol is not suitable for transferring large amounts of data, when each send is delayed until the previous acknowledgment is received.

Our protocol is asynchronous: only one acknowledgment is sent for the last message of the transmitted data. To ensure reliability of message transfer we use the TCP connection oriented protocol. At the initialization step, each proxy opens a TCP connection to its SFS server.

PERFORMANCE EVALUATION OF STAR

This section presents the performance of STAR. The implementation runs on an Ethernet network of Sun workstations. The environment has not been modified (all of the usual demons were running). All presented results are averages over a number of trials. These measurements could help system administrators or application designers to choose appropriate parameters (message logging strategies, degree of replication, period of checkpointing) geared to their fault tolerance and performance requirements.

First, we report performance measurements of the basic components then we show performance of parallel applications using STAR.

Performance of STAR components

Performance of Star file system (SFS)

Figure 7 shows the performance of the STAR file system (SFS) with different replication degrees for writing and reading a file of 1 Megabyte. These measurements were taken on a set of Sun 5 and 10 workstations with 32 Mb of memory. A replication degree of 4 means that each file is saved on 4 disks on 4 hosts. We also illustrate the performance of NFS when data are in cache or not. Naturally, NFS measurements do not depend on the replication degree. SFS read has not been optimized since it is only used during recovering. On the other hand, SFS write is especially stressed since it is used during normal running for checkpointing and message logging. We see that in every case STAR writing is very efficient compared to NFS. This good performance is essentially due to the parallelization of server accesses and the use of an asynchronous protocol between client hosts and SFS servers.

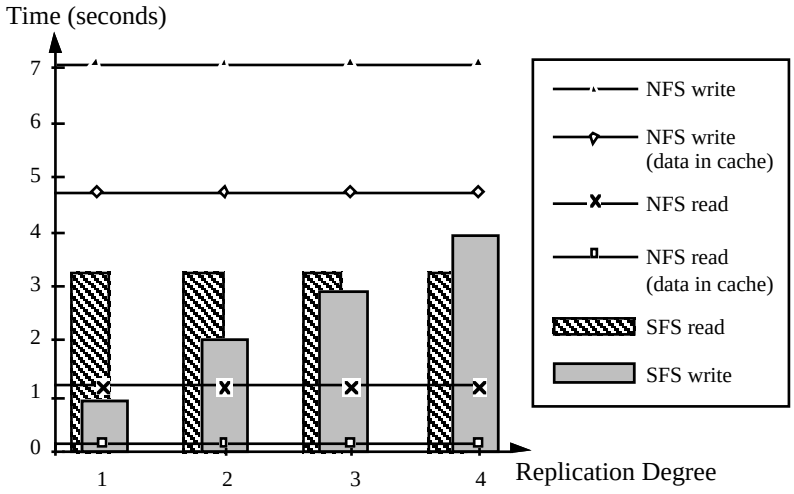

Figure 7. Performance of the STAR File System

Performance of message logging

Figure 8 shows the cost of optimistic message logging as a function of the size of the queue on the sending host. 0 means that messages are directly saved on stable storage before being delivered. 500 means that messages are queued at the sending host and are asynchronously saved when the queue contains 500 messages. 0 queue size is equivalent to the pessimistic strategy. These measurements were taken for 1024 messages of one kilobyte transmitted between two user processes. Messages are saved on two different hosts. We indicate the average sending time. We also monitored the relevant communication servers to determine the maximum amount of memory used during the transfer.

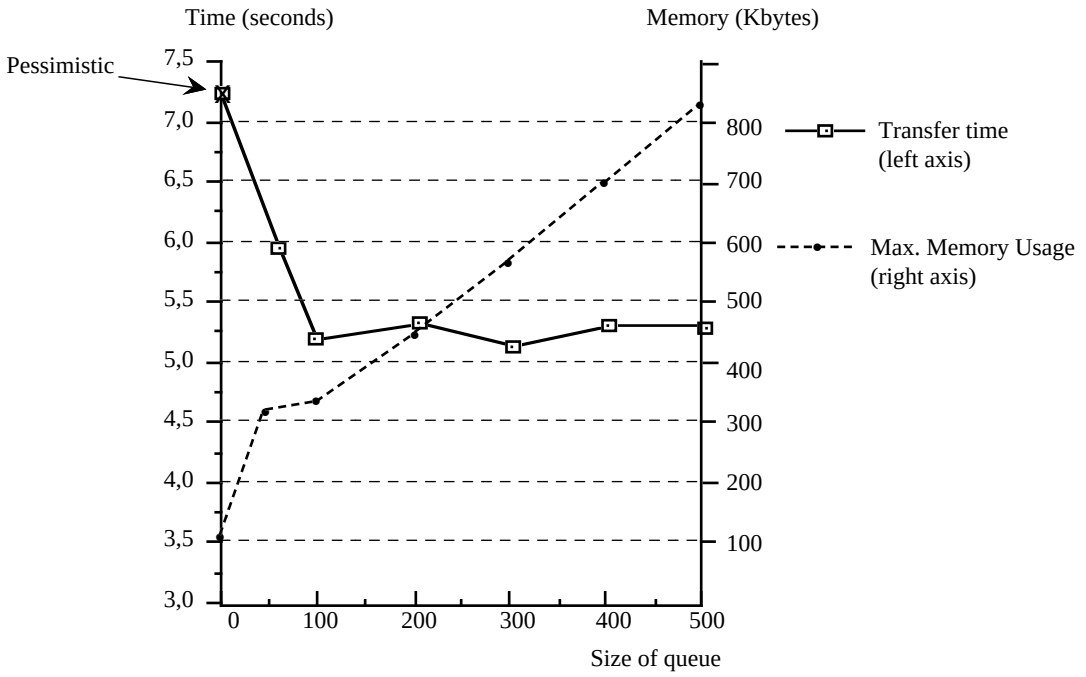

Figure 8. Optimistic message logging cost

We observe a sizeable reduction of the sending time when at least one hundred messages are queued. Below this number, the message saving happens too frequently and the size of saved data is too small to obtain a speed-up due to the asynchronous SFS protocol. We also note a stabilization of the sending time when the queue contains more than one hundred messages. This effect could result from an increase of the average load of the communication servers, which consequently become less able to satisfy application requests. The memory usage partly illustrates this effect. Consequently, the application designer has to tune the queue size in order to find a good compromise between the asynchronous storage and memory usage. One hundred messages appears to be a good value and was chosen as a default value for the message queue size.

We also measured the time to send messages with the sender-based algorithm where all messages are kept in the queue and are never saved on stable storage. The average sending time is only 3.8 seconds but the maximum memory usage is 1.2 Megabytes. In fact the sender-based protocol uses too much memory to be useful for real applications. In such a case, the queues on sending hosts will become too large and the demand-paging system may slow down the application, depending on the size of the main memory available.

Performance of checkpointing

Additional experiments were run with STAR ported to a Sun IPC network with an approximate computing power of 12 VAX MIPS and 24 Mb of memory. The following three figures present the running times of a simple sequential sort program

using three independent checkpointing implementations: full checkpointing (where all data are written in stable storage and the process is blocked until the checkpoint is over), full non-blocking checkpointing (where the application continues to execute while the checkpoint is written on stable storage), and incremental non-blocking checkpointing (reducing the amount of data to be written). The cost of checkpointing has been measured with different replication degrees (from 1 to 4) and with different process sizes (from 100 to 1150 kilo-bytes). Programs run with a rather short 20 second checkpointing interval. In practice, longer intervals should be used. In that sense, we overestimate the cost of checkpointing to evaluate the checkpoint mechanism in a loaded situation. At the end of this section, Table II presents an evaluation with a more realistic checkpoint interval.

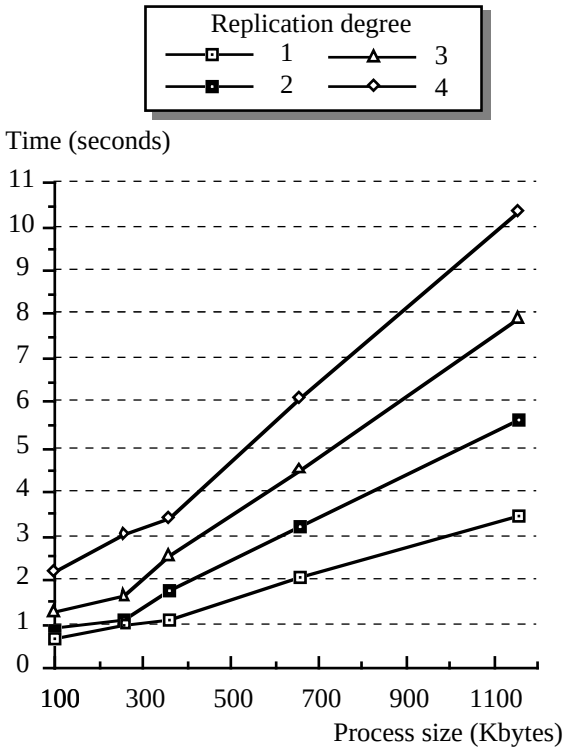

Figure 9. Full checkpoint cost

Figure 9 shows the cost of full checkpointing, where data and stack segments are entirely copied to stable storage. The cost linearly depends on the process context size. The time to save 1150 kilo-bytes on four replicated files takes 10.3 seconds. This is about three times slower than writing the same amount of data on a single file (3.4 seconds). The measured time can appear high compared to the performance of the STAR file server presented above. This is due to the difference of machines in terms of processing power. Moreover, the short checkpoint period overloads the file servers.

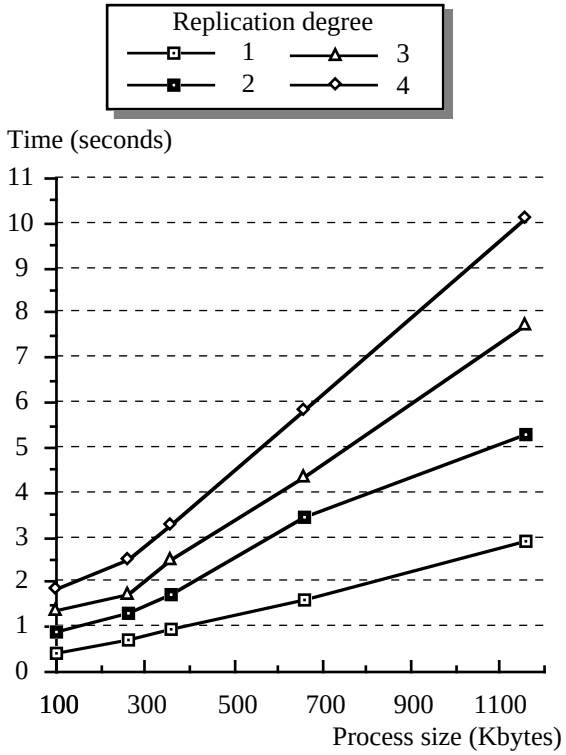

Figure10. Non-blocking checkpoint cost

With blocking checkpointing, the performance degradation is dependent on the amount of data to be saved, due to the latency in accessing file servers. Figure 10 shows the checkpoint cost with the full non-blocking method. It might seem surprising to see no significant reduction of the checkpoint cost. In fact, this method consumes much CPU time and memory capacity, and the host resources become saturated because of the small checkpoint period. Non-blocking checkpointing provides better results for larger processes with larger checkpoint intervals, as we will see with the parallel applications (see Table II).

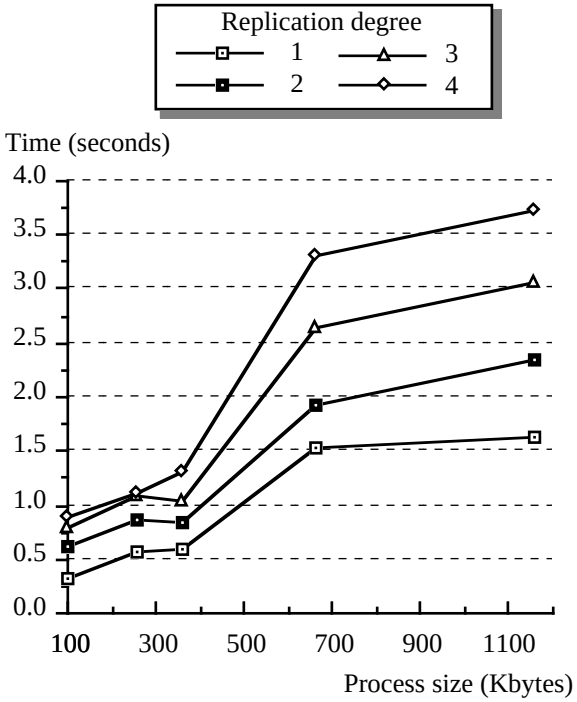

Figure 11. Incremental checkpoint cost

Figure 11 shows the costs associated with incremental checkpointing. In the previous checkpointing methods, the amount of data written on stable storage was significant even though only a small part of the data changed between successive checkpoints. With incremental checkpointing, we observe a sizeable reduction of the checkpoint overhead. An incremental checkpoint is about three times faster than a full one. For the smallest program the checkpoint cost is less than 1 second for any degree of the replication. We can observe that the curves are not linear because the time to take a checkpoint depends on the process's memory usage.

Performance of rollback/recovery

Now we consider the recovery time of a process after a failure. This time includes the time to restart a process and to recover its state from its last checkpoint.

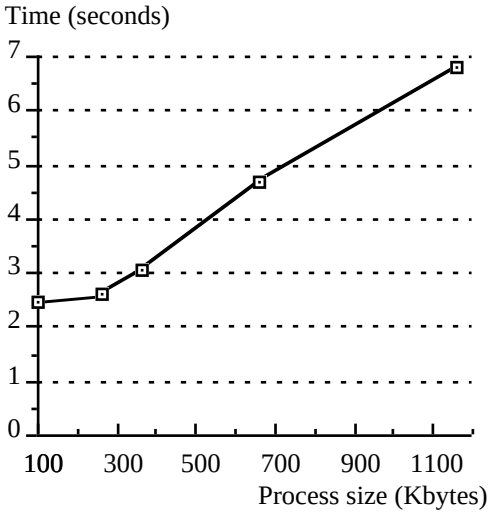

Figure 12. Restore context cost

The restoration time is large compared to the checkpoint cost. In fact, the restoration step is much more complex. It must identify the faulty process, reconfigurate the ring, create a new process, restore its context. Finally, all servers update their global view with the new process location. Repeated failures incurs an extra-overhead due to the detection period, the restoration time and the time to rollback. The time costs of one fault may be evaluated as :

mean recovery time = detection period / 2 + restoration time + checkpoint period /2

The cost of faults is therefore linear with the number of failures.

Performance of applications using STAR

To tally the performance of STAR under a working load we chose three long-running, compute-intensive applications exhibiting different memory usage and communications patterns:

1. The gauss application performs gaussian elimination with partial pivoting on a 1024 x 1024 matrix. The matrix is distributed among several processes. At each iteration of the reduction, the process which holds the pivot sends the pivot column to all other processes.
2. The multiplication application, called matmul, multiplies two square matrixes of size 1024 x 1024. The computation is distributed among several processes. No communication is required other than reporting the final solution.
3. The fft application computes the Fast Fourier Transform of 32768 data points. The problem is distributed by assigning each process an equal range of data points. Like the previous application, no communication is required other than reporting the final solution.

Table I presents running time, communication, and memory requirements for the three applications when run without fault-tolerance management (i.e. without checkpointing and message logging). Each application is distributed on five hosts: one host executes a master process and the four other hosts execute the computational processes.

Table I. Applications requirements

Applications	Application Running Time (seconds)	Per Process Memory (Kbytes)	Per Process sending data (Kbytes)
gauss	344	1704	2700
matmul	723	2688	0.06
fft	1177	1200	0.06

Gauss and matmul require a sizeable amount of data, stressing the checkpoint and state restoration mechanisms. Moreover, the gauss application exhibits a large amount of communications especially stressing the message logging. The fft application is long-running and requires a medium amount of data.

Table II presents the running times of the applications programs when run with independent checkpointing and pessimistic message logging. Applications run with a 2-minute checkpointing interval. Checkpoints and logs are duplicated.

Table II. Parallel applications evaluation

Applications	Full checkpoint		Full Non-blocking checkpoint		Incremental checkpoint	
	Running Time (sec.)	Percentage of overhead	Running Time (sec.)	Percentage of overhead	Running Time (sec.)	Percentage of overhead
gauss	567	64.82	505	46.80	457	32.85
matmul	844	16.79	768	6.34	748	3.57
fft	1244	5.75	1228	4.36	1194	1.50

In spite of checkpoint optimizations, we observe a high overhead for the gauss application. In fact this application is not a good candidate for message logging approaches, specifically because of its communication rate. The overhead due to message logging for this application is 14.53%. The cost of message logging represents half of the global overhead when we apply incremental checkpointing.

For all three applications, incremental checkpointing provides a sizeable reduction of the overhead. Comparing to the full non-blocking checkpointing, we obtain reduction in overheads of between 24% and 63%. This difference is partly due to different communication rates. Moreover, applications can be divided into two categories: applications with an address space that is modified with high locality (matrix multiplication and fft applications) and applications with an address space that is modified almost entirely between any two checkpoints (gauss application). For applications of the first category, incremental checkpointing is very successful (79% of

reduction for matrix multiplication and 75% of reduction for fft). For the applications in the second category, incremental checkpointing is less effective (about 49% of reduction for the gauss application).

CONCLUSIONS

This paper presented the STAR fault manager for distributed applications in a standard workstation environment. The basic software components of STAR are (1) an efficient host crash detection mechanism based on a logical ring, (2) three checkpointing mechanisms (full, non-blocking, incremental), (3) a restoration mechanism, and (4) a configurable stable storage using replicated files and a proxy mechanism to reduce the latency in accessing file servers. The implementation is based on independent checkpointing, and avoids the domino effect by using reliable storage of messages. STAR is flexible with tunable parameters so that applications can adapt the fault management according to the load and failure rate of the system.

STAR has been developed on a set of SUN-Sparc stations connected by Ethernet. We have reported performance measurements of the basic software components. The results demonstrate that independent checkpointing is an efficient approach for providing fault tolerance for the specific applications studied, i.e., long-running with few message exchanges. We have also shown that a software based management of fault tolerance is an interesting alternative to specialized hardware or kernel-integrated solutions. Results from [24], as well as our own instrumentation of distributed applications, corroborate this claim. Furthermore, it appears from other work and our own experience, that some optimization methods are very important: in particular non-blocking and incremental checkpointing[11,15]. Either technique incorporated within STAR leads to a drastic reduction of the overhead for classical parallel applications.

REFERENCES

1. L. Alvisi, B. Hoppe, and K. Marzullo, 'Nonblocking and Orphan-Free Message Logging Protocols', *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, 1993, pp. 145-154.
2. M. Banâtre, G. Muller, B. Rochat, and P. Sanchez, 'Design Decisions for FTM: a General Purpose Fault Tolerant Machine', *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, June 1991, pp. 71-78.
3. A. Barak, A. Braveman, I. Gilderman, and O. Laden, 'Performance of PVM with the MOSIX Preemptive Process Migration Scheme', *Proceedings of the 7th Israeli Conference on Computer Systems and Software Engineering*, June 1996.
4. G. Bernard and D. Conan, 'Flexible Checkpointing and Efficient Rollback-Recovery for Distributed Computing', *Proceedings of the Society UNIX User Group International Conference, Open Systems: Solution for Open World*, April 1994.
5. B. Bhargava, S-R. Lian, and P-J. Leu, 'Experimental Evaluation of Concurrent Checkpointing and Rollback-recovery Algorithms', *Proceedings of the International Conference on Data Engineering*, March 1990, pp. 182-189.
6. K.P. Birman and T. Joseph, 'Reliable Communication in the Presence of Failures', *ACM Transactions on Computer Systems*, **5**,47-76, (February 1987).
7. 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).

8. D. Briatico, A. Ciuffoletti, and L. Simoncini, 'A Distributed Domino-Effect Free Recovery Algorithm', *Proceedings of the 4th Symposium on Reliable Distributed Systems*, October 1984, pp. 207-215.
9. G. Cabillic and I. Puaut, 'Startdust: an Environment for Parallel Programming on Networks of Heterogeneous Workstations', *Journal of Parallel and Distributed Computing*, **40**,(1),65-80, (January 1997).
10. K.M. Chandy and L. Lamport, 'Distributed Snapshots: Determining Global States of Distributed Systems', *ACM Transactions on Computer Systems*, **3**(1):63-75, (1985).
11. Y. Chen, K. Li, and J.S. Planck, 'CLIP: A Checkpointing Tool for Message-passing Parallel Programs', *Proceedings of High Performance Networking and Computing*, November 1997.
12. H. Clark and B. McMillin, 'DAWGS - a Distributed Compute Server Utilizing Idle Workstations', *Journal of Parallel and Distributed Computing*, **14**,175-186, (February 1992).
13. F. Cristian and F. Jahanian, 'A Timestamp-based Checkpointing Protocol for Long-lived Distributed Computations', *Proceedings of 10th Symposium on Reliable Distributed Systems*, September 1991, pp. 12-20.
14. F. Douglis and J. Ousterhout, 'Transparent Process Migration: Design Alternatives and the Sprite Implementation', *Software - Practice and Experience*, **21**,(8),757-785, (1991).
15. E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, 'The Performance of Consistent Checkpointing', *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992.
16. E.N. Elnozahy and W. Zwaenepoel, 'On the Use and Implementation of Message Logging', *Proceedings of the 24th International Symposium on Fault-Tolerant Computing Systems*, June 1994, pp. 298-307.
17. B. Folliot and P. Sens, 'GATOSTAR: A Fault-tolerant Load Sharing Facility for Parallel Applications', *Proceedings of the First European Dependable Computing Conference, Lecture Notes in Computer Science 852*, October 1994, pp. 581-598.
18. D. B. Johnson and W. Zwaenepoel, 'Sender-Based Message Logging', *Proceedings of the 7th Symposium on Fault Tolerant Computing Systems*, June 1990, pp. 97-104.
19. 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).
20. R. Koo and S. Toueg, 'Checkpointing and Rollback-Recovery for Distributed Systems', *IEEE Transactions on Software Engineering*, SE-**13**,(1),23-21, (January 1987).
21. P.A. Lee and T. Anderson, 'Fault Tolerance, Principles and Practice, Second revised edition', *Dependable Computing System Vol. 3*, Springer-Verlag, 1990.
22. M. Litzkow and M. Solomon, 'Supporting Checkpointing and Process Migration outside the UNIX Kernel', *Proceedings of the Usenix Winter Conference*, January 1992.
23. G. Muller, M. Hue, and N. Peyrouze, 'Performance of Consistent Checkpointing in a Modular Operating System: results of the FTM Experiment', *Proceedings of the First European Dependable Computing Conference,, Lecture Notes in Computer Science 852*, October 1994, pp. 491-508.
24. J.S. Plank, M. Beck, G. Kingsley, and K. Li, 'Libckpt: Transparent Checkpointing under Unix', *USENIX Winter 1995 Technical Conference*, January 1995.
25. D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck, 'The Delta-4 Approach to Dependability in Open Distributed Computing Systems', *Proceedings of the 18th International Symposium on Fault-Tolerant Computing Systems*, 1988, pp. 246-251.
26. D. Powell (Ed.), 'Delta 4: A Generic Architecture for Dependable Distributed Computing',*Research Reports ESPRITS*, Berlin, Germany, Springer-Verlag, 1991.
27. J. Pruyne and M. Livny, 'Managing Checkpoints for Parallel Programs', *Proceedings of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, April 1996.
28. B. Ramkumar and V. Strumpen, 'Portable Checkpointing for Heterogeneous Architecture', *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems*, June 1997, pp. 58-67.
29. B. Randell, 'Design Fault Tolerance', *The Evolution of Fault-Tolerant Computing Vol. 1*, Springer-Verlag, pp. 251-270, (1987).
30. P. Sens and B. Folliot, 'STAR: A Fault Tolerant Systems for Distributed Applications', *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, December 1993, pp. 656-660.
31. P. Sens, 'The Performance of Independent Checkpointing in Distributed Systems', *Proceedings of the 28th Hawaii International Conference on System Science*, January 1995, pp. 525-533.

32. S.K. Shrivastava and D.L. McCue, 'Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment', *IEEE Transactions on Parallel and Distributed Systems*, April 1994, pp. 421-432.
33. A. P. Sistla and J. L. Welch, 'Efficient Distributed Recovery using Message Logging', *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, August 1989.
34. S.W. Smith, D.B. Johnson, and J.D. Tygar, 'Completely Asynchronous Optimistic Recovery with Minimal Rollbacks', *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, June 1995.
35. R.E. Strom and S.A. Yemini, 'Optimistic Recovery in Distributed Systems', *ACM Transactions on Computer Systems*, **3**,(3),204-226, (August 1985).
36. 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*, **33**,46-63, (December 1990).
37. J. Wakerly, 'Error Detecting Codes, Self-Checking Circuits and Applications', *Elsevier North-Holland*, (1978).
38. Y.M. Wang and W.F. Fuchs, 'Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems', *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992, pp. 147-154.
39. Y.M. Wang, O.P. Damani, and V.K. Garg, 'Distributed Recovery with K-Optimistic Logging', *Proceedings of the 17th International Conference on Distributed Computing Systems*, May 1997, pp. 60-67.
40. J. Xu and R.H.B Netzer, 'Adaptive Independent Checkpointing for Reducing Rollback Propagation', *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, December 1993, pp. 754-761.
41. S. Zhou, J. Wang, X. Zheng, and P. Delisle, 'Utopia: a Load-sharing Facility for Large Heterogeneous Distributed Computing Systems', *Software - Practice and Experience*, **23**,(2),1305-1336, (December 1993).