INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE
N° 913 - Février 1995

Lessons from FTM: an Experiment in the Design and Implementation of a Low Cost Fault Tolerant System

Gilles Muller, Michel Banâtre, Mireille Hue, Nadine Peyrouze et Bruno Rochat

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

Instiitut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu - 35042 Rennes Cedex - France
Tél. : (33) 99 84 71 00 - Fax : (33) 99 38 38 32

Lessons from FTM: an Experiment in the Design and Implementation of a Low Cost Fault Tolerant System

Gilles Muller, Michel Banâtre, Mireille Hue, Nadine Peyrouze et Bruno Rochat

Programme 1- Architectures parralèles, bases de données, réseaux et systèmes distribués

Projet Solidor

Publication interne n˚913 - Février 1995

46 pages

Abstract: This report describes an experiment in the design of a general purpose fault tolerant system, FTM. The main objective of the FTM design was to implement a "low-cost" fault tolerant system that could be used on standard workstations. At the operating system level, our goal was to provide a methodology for the design of modular reliable operating systems, while offering fault tolerance transparency to user applications. In other words, porting an application to FTM had only to require compiling the source code without having to modify it. These objectives were achieved using the Mach micro-kernel and a modular set of reliable servers which implement application checkpoints and provide continuous system functions despite machine crashes. At the architectural level, our approach relies on a high performance stable storage implementation, called Stable Transactional Memory (STM), which can be implemented either by hardware or software. We first motivate our design choices, then we detail the FTM implementation at both architectural and operating system level. We comment on the reasons for the evolution of our stable memory technology from hardware to software. Finally, we present a performance evaluation of the FTM prototype. We conclude with lessons learned and give some assessments.

Key-words: Fault Tolerance, Blocking Consistent Checkpointing, Stable Memory, Modular Operating System, Micro-kernel.

Centre National de la Recherche Scientifique
(Ura 227) Université de Rennes 1 - Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique - unité de recherche de Rennes

Leçons du projet FTM : une expérimentation dans la conception d'un système tolérant les fautes de faible coût

Résumé : Ce document présente une expérimentation dans la conception d'un système tolérant les fautes à vocation générale, le FTM. Notre motivation principale était la conception d'un système de faible coût pouvant être utilisé sur des stations de travail standard. En ce qui concerne le système d'exploitation, notre objectif était de développer une méthodologie de conception de systèmes d'exploitation fiables offrant la transparence de la tolérance aux fautes aux applications utilisateurs. Autrement dit, le portage d'une application sur FTM ne doit nécessiter que la compilation du logiciel source sans avoir à modifier ce dernier. Nos objectifs ont été atteints en utilisant le micro-noyau Mach et un ensemble modulaire de serveurs fiables qui implément les points de reprises des applications et offrent un service système continu, malgré la défaillance d'une machine. Au niveau de l'architecture, notre approche a reposé sur la conception d'une mémoire stable rapide pouvant être mise en œuvre soit par matériel, soit par logiciel. Nous décrivons tout d'abord nos choix de conception, puis nous présentons la mise en œuvre du FTM en ce qui concerne l'architecture et le système d'exploitation. En particulier, nous décrivons l'évolution de la technologie mémoire stable depuis sa mise en œuvre par matériel jusqu'à son implémentation par logiciel. Enfin, nous présentons une évaluation des performances du prototype qui a été réalisé au cours de cette étude. Nous concluons en tirant les leçons de ce projet.

Mots-clé : tolérance aux fautes, points de reprise cohérents, mémoire stable, système d'exploitation modulaire, micro-noyau.

1 Introduction

A fault tolerant computer system is one which is able to deliver a continuous service to its users even if one or several of its subsystems crash. Fault tolerance is obviously a desirable property and the natural answer to the question "who needs a fault tolerant computer?" should be "everybody", as no one wants to suffer from a system outage. Unfortunately, solutions to fault tolerance are based on redundancy techniques [Lee & Anderson 90] and these introduce additional hardware costs and overheads in execution times. As a consequence, very few computers among the many designed are fault tolerant. Designing a computer system is always a trade-off between performance and cost; designing a fault tolerant computer system is even more complicated since the solution is a trade-off between performance, cost and the level of system reliability that users obtain for their applications.

This document describes an experiment in the design of a general purpose fault tolerant system, FTM, developed at IRISA [Banâtre *et al.* 91b]. The main objective of the FTM design was to implement a "low-cost" fault tolerant system that could be used on standard workstations. At the operating system level, our goal was to provide a methodology for the design of modular reliable operating systems while offering fault tolerance transparency to user applications. In other words, porting an application to FTM had only to require compiling the source code without having to modify it. These objectives were achieved using the Mach micro-kernel and a modular set of reliable servers which implement application checkpoints and provide continuous system functions despite machine crashes [Banâtre *et al.* 93]. At the architectural level, our approach relies on the use of a high performance stable storage implementation called Stable Transactional Memory (STM) which can be implemented either by software or hardware.

1.1 Motivation for Fault Tolerance in a Local Area Network

Local area network based distributed systems are often built from a set of workstations or PCs. Such a system architecture is commonly used in office automation, software development and scientific environments. Since LAN-based systems are made up of independent machines, one might expect such architectures to be less sensitive to individual machine failures than a centralized system. Therefore, the basic fault tolerance requirement concerns the availability of the whole system: a single node may fail, but such a failure must not freeze all of the system. In other words, to preserve global availability in a LAN-based system, no basic operating system function should depend on the availability of any single node. Basic functions include naming services (e.g., machines, users), storage services, such as distributed file systems, and distributed lock services. Consequently, one can state the primary fault tolerance requirement for LAN systems as being that *the operating system has to provide an available service paradigm* where an available service is one that can run continuously despite node failures. It should be noted that some services (e.g., internet naming) are already highly available in commonly used distributed systems but rely on *ad-hoc* solutions that cannot be generalized.

In office automation systems, where dedicated tools such as word processors, spreadsheet calculators and publishing tools are used, high application availability is not a general requirement. If a machine crashes, a user only wants to be able to restart his job on another workstation from a recent consistent state without losing too much work. We can formulate this second requirement as being that *the operating system has to make a consistent user state available*. Currently, this facility is sometimes implemented within tools such as text editors which regularly flush temporary text versions to disk. Nevertheless, these solutions rely on *ad-hoc* recovery mechanisms which do not easily integrate into standard operating systems: problems such as atomically replacing an old file version with an updated one are not always managed properly following a crash. Therefore, to permit reuse of existing applications, a third requirement is that *fault tolerance management must be transparent to the end user programmer*. This means that fault tolerance mechanisms have to be integrated into the operating system layers.

In a scientific environment, LAN distributed systems are also used as large virtual multiprocessors to run number crunching computations on many nodes for long durations (up to several days). The temporary unavailability of one of the virtual multiprocessor nodes is acceptable, so long as it only results in a loss of CPU power and that the whole computation is still able to complete successfully. Due to the large number of nodes involved in such computations and to their duration, applications not only have to deal with hardware crashes, but also with the likelihood of system maintenance. In a non-fault-tolerant computing environment, a programmer who wants to cater for such situations has either to cut his large software into small pieces communicating through files or, design, by hand, some kind of disk checkpoint mechanism permitting an interrupted program to be restarted.

We may thus consider that the main fault tolerance requirement for scientific computations is that a node failure and restart be treated transparently to the end user application programmer. If we assume that the operating system provides available process states (i.e., the second need that we had identified), tolerating node maintenance is simple. However, the execution environment of a scientific computation is generally different to that of the office automation system. Firstly, a user job is generally tied to a single workstation with the exception of accesses made to remote files. Second, the saving of a consistent state must be done quite frequently so that the amount of work lost is minimized. By contrast, in scientific computations, there are no strong requirements concerning the time interval between two savings, since the loss of five to ten minutes of work is not very significant in a day-long computation.

Since a LAN architecture is built from off-the-shelf components, a good approach to fault-tolerance in such an environment is to make it "low-cost" so that the solutions can be widely applied. This objective was our major guideline when defining the FTM solutions to the three requirements identified above.

1.2 Content of the paper

This document is structured as follows. Section 2 presents an overview of the FTM approach to low-cost fault tolerance and outlines the main design choices that we made. Section 3 describes the FTM architecture and the evolution of our RAM stable storage technology from hardware boards to a software implementation. Section 4 presents our model for designing reliable modular operating systems and details the checkpointing algorithms. Section 5 is devoted to the programming of reliable servers; we focus in particular on how fault tolerance can be masked from system programmers. Section 6 gives some performance evaluation for applications representative of workstation use. Section 7 presents related work. Section 8 concludes with lessons learned as well as some assessments.

2 How to Implement a Low Cost Fault Tolerant System: the FTM Solution

Our main design choices for FTM, at both architectural and operating system level, were dictated by our objective of building a low cost fault tolerant system. We now detail these choices.

2.1 Minimizing the Hardware Cost of Fault Tolerance

The hardware cost of a fault tolerant system depends mainly on the degree of replication of its components. Thus, to build a low-cost system we chose solutions that minimize the degree of machine replication; this was done in two ways. First, we chose to tolerate only a single hardware failure at a time. This choice is justified by the fact that the reliability of popular computers has increased and that hardware failures are becoming increasingly rare. Consequently, the probability that two machines fail at the same time is acceptably low in a LAN environment. Second, we oriented the operating system towards passive replication and checkpointing. In normal operating mode, all machines of the architecture perform their own jobs; in the event of a crash, jobs that were running on the failed machine are restarted on a backup machine from a previously saved checkpoint. It should be noted that after a failure, the backup machine executes both the failed machine's jobs and its own jobs.

Finally, to minimize the hardware cost, the FTM architecture is not built from special purpose machines but is instead designed to use standard machines: PCs, workstations, industrial CPU boards. As a result of this choice, failure detection relies both on crash detection (software watch-dogs) and on mechanisms built into the machine (Error Checking Codes, Checksums). When an error is detected by these built-in mechanisms, the operating system halts the machine.

2.2 RAM Stable Storage: Architectural Support for Efficient Checkpointing

The implementation of checkpointing requires that checkpoints be stored in a memory which is unaffected by crashes. Stable storage disks [Lampson 81] are usually used for this purpose. Such a solution offers acceptable performance for file meta-data but is inefficient when using small structures such as those of the low level system (e.g., lists, graphs). To permit efficient use of persistent memory within the operating system, we retained the stable RAM technology used in our previous studies into reliable distributed systems: Enchere [Banâtre *et al.* 86] and Gothic [Banâtre *et al.* 88]. A RAM stable storage is based on the use of separate banks of RAM memory located in two independent machines for availability reasons. The FTM RAM stable storage, called Stable Transactional Memory, provides two notions of *stable object* and *transaction*. We define a stable object as a contiguous set of memory words and a transaction as an atomic set of primitive operations executed on stable objects. Two versions of STM were successively designed during the FTM project: a hardware based implementation which provides fine-grained object protection, and a software implementation which does not require any specific hardware and is thus easily portable. The evolution of our STM technology during the project as well as STM features are discussed in section 3.

An STM can be conceptually viewed as a memory possessing two independent access ports (see figure 1). It is used to build a *stable node* which is made up of a primary machine, a backup machine and an STM. In normal operation, the primary processor has exclusive access to the STM and uses it to store checkpoints. If a crash occurs, the backup processor aborts transactions running on the failed machine in order to ensure a consistent state of STM data, and then takes over by restarting the computations from their checkpoints. Stable nodes are coupled in pairs using passive replication: the primary machine of a stable node is the backup machine of the other stable node and visa versa.

2.3 Modular Operating Systems: the Right Way to Integrate Checkpointing

Our design goal for the FTM operating system was to satisfy three requirements. The first requirement was fault tolerance transparency for the end user application programmer so that he could design or port an existing application without having to write source code for failure handling. Transparency is already provided by many of the current checkpointing techniques [Elnozahy *et al.* 92] [Borg *et al.* 89] used in scientific computations. However, most of these techniques deal only with the memory resource of application processes; this is not sufficient as standard soft-

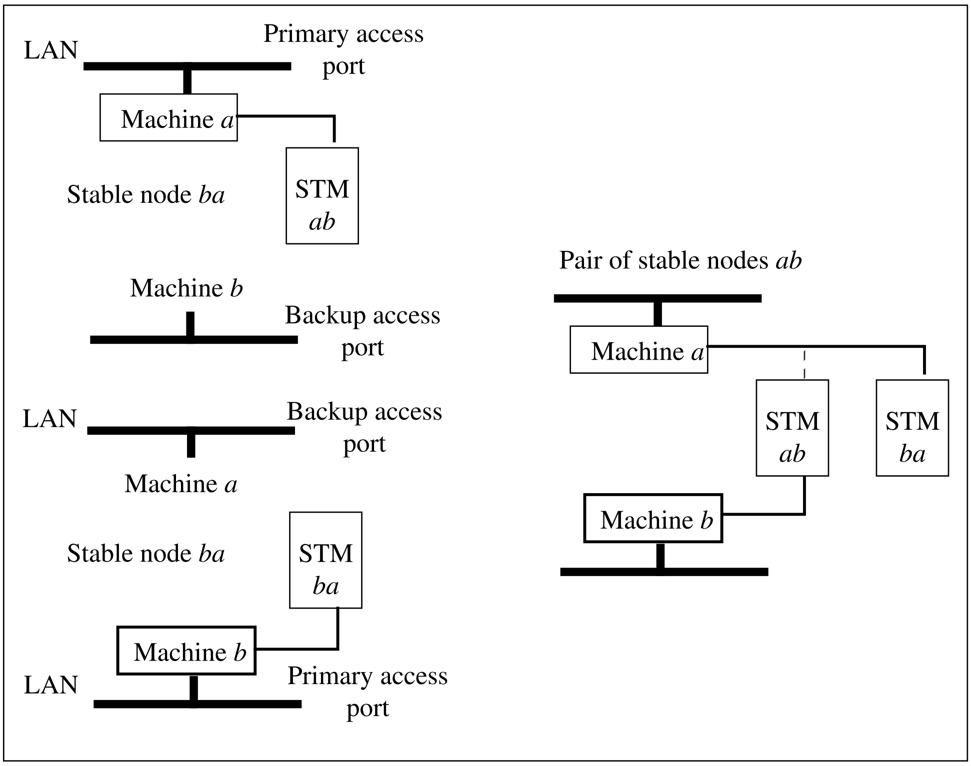

Figure 1 : The pair of stable nodes

ware generally use various other system resources such as files and screens. Thus, our second operating system requirement was that checkpointing be extended to manage all system resources and services. This has been achieved by the provision of a methodology for designing reliable operating systems. Our third requirement was portability for the operating system, that is, it could be easily installed on different hardware platforms. This last objective had many consequences on the operating system design.

Before the appearance of the micro-kernel technology, integration of a new feature, such as fault tolerance, within an operating system was accomplished in one of two ways: a new operating system could be designed from scratch or an existing monolithic system could be extended. The first solution can be efficient but entails a huge amount of work. Moreover, it suffers from portability restrictions, thus causing problems whenever migration to a new generation of machines is required. Modifying a monolithic system solves some portability restrictions since it frees the developer from having to deal with low level hardware. Nevertheless, the resulting software is still dependent upon up-coming kernel releases. Furthermore, interactions between parts of code in a monolithic system are not always well defined; addition and debugging of new code is difficult.

Micro-kernels [Accetta *et al.* 86] [Rozier *et al.* 88] [Mullender *et al.* 90] opened a third way; they provide a low level abstract machine and permit the design of modular operating systems. Micro-kernel operating systems are generally based on the client-server model, that is, the system is built from a set of servers, each implementing a system service. It is possible to add new services, by means of new servers, to an existing system without having to change the kernel. This solution is therefore highly portable.

Given the advantages of micro-kernels, we decided to introduce fault tolerance in a modular operating system while keeping the advantages of the micro-kernel technology. In particular, introducing fault tolerance without sacrificing modularity means that the classical client-server model has to be modified to integrate fault tolerance mechanisms. In FTM, this is achieved through the design of *reliable server*s which implement available system services (see figure 2). A reliable server manages a system resource and is able to restart after a failure from a local checkpoint stored in STM. An application checkpoint is made up of the collection of local checkpoints of the reliable servers created by the application's processes (e.g., memory segment server, screen server, file server). The next paragraph gives the reasons which led us to retain a blocking consistent checkpointing algorithm for creating application checkpoints.

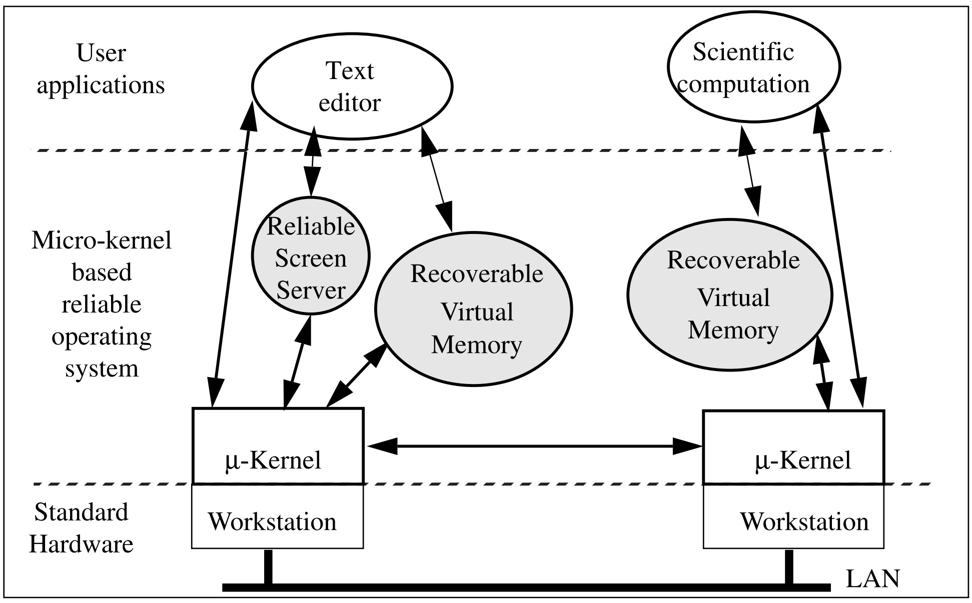

Figure 2 : Structure of the FTM operating system

2.4 Blocking Consistent Checkpointing

Checkpointing has been widely studied by many researchers. These studies fall into two groups: independent (asynchronous) checkpointing and consistent (synchronous) checkpointing. In the independent checkpointing approach, each processor takes checkpoints independently. Messages exchanged between applications are logged during normal execution and are replayed after a crash. Each process executes normally without having to synchronize with others. The drawbacks of independent checkpointing are that it is either domino-effect prone [Wood 81] [Bhargava & Lian 88] [Merlin & Randell 78], or that message replay requires deterministic processes [Strom & Yemini 85] [Borg *et al.* 89] [Goldberg *et al.* 90] [Juang & Venkatesan 91] [Elnozahy & Zwaenepoel 92]. Forcing processes to be deterministic means that the sources of systems' non-determinism, such as multi-threading, interrupts and memory mapped I/O [Gleeson 93], must be removed. Consequently, solutions for deterministic processes are always hardware dependent and lead to restrictions on the services that can be implemented in the operating system.

In the consistent checkpointing approach, processors coordinate their local checkpointing actions so that the global state is guaranteed to be consistent [Chandy & Lamport 85]. When a failure occurs, processors roll back and restart from their preceding checkpoint. Due to our portability requirement, we retained the consistent checkpointing approach since it allowed us to deal with sources of systems' non-determinism and thus its implementation did not require any assumption about the underlying micro-kernel nor about the machine on which the system ran (mono-processor, multiprocessor). Consistent checkpointing techniques also fall in two subgroups: blocking and non-blocking techniques. In blocking techniques [Tamir & Sequin 84] [Koo & Toueg 86] [Leu & Bhargava 88], processes halt and synchronize with each other when saving a local checkpoint. To minimize halt-time duration, several studies have focused on how to reduce the number of dependent processors involved in a checkpoint and the number of messages exchanged [Koo & Toueg 86] [Ahamad & Lin 89].

Non-blocking techniques [Cristian & Jahanian 91] [Li *et al.* 91] [Silva & Silva 92] [Elnozahy *et al.* 92] have been successfully applied to message based parallel scientific applications in distributed systems. When a process takes (or receives) a checkpoint decision, it saves a temporary checkpoint and resumes its execution. Temporary checkpoints are later made definitive when it is known that all processes have saved a temporary checkpoint and that no messages are still in transit. However, when working at the operating system level, non blocking checkpointing makes the management of operating system resources such as memory segments difficult. For instance, application processes have to be halted while the contents of their memory segments are checkpointed in order to ensure that the CPU registers are consistent with the segment. Application processes and server processes have thus to be synchronized during a checkpoint. It should also be noted that non blocking checkpointing is more complex to implement than blocking checkpointing since it requires message logging during the execution of the checkpointing protocol and a termination algorithm for making temporary checkpoints definitive. Due to the these considerations, we chose to implement a blocking checkpointing algorithm.

Though blocking consistent checkpointing possesses the advantage of being simple, its well known drawback is that processes are stopped during the checkpointing protocol. One of the contributions of the FTM project has been to develop strategies for reducing the checkpoint duration by minimizing the number of servers involved in the computation of a global consistent state, and also by reducing the amount of data checkpointed [Muller *et al.* 94]. Reduction of the number of servers involved in the computation of a global consistent state is achieved by keeping track of the dependencies arising from a client's call to a server operation. Minimization of the data to be checkpointed is performed at the application level by the segment server (recoverable virtual memory) and the micro-kernel virtual memory support: only pages modified since the preceding checkpoint are saved in a checkpoint. At the server level, we require that checkpointing only be performed after specific instructions; this permits stack and data segments to be discarded from the servers' checkpoint and thus efficiently reduces the checkpoint size; it need only contain stable data.

In the next three sections, we detail the implementation of several aspects of the FTM project: the STM technology, the reliable client-server model, the checkpointing protocols and the design of reliable servers.

3 Evolution of the RAM Stable Storage Technology: from Hardware to Software

Our initial objective for the stable RAM technology was to design a reliable memory device which protects the consistency and availability of data in the event of a crash. We also wanted to use stable data within servers and this necessitated the ability to efficiently manage groups of objects of arbitrary size, from single word objects to MMU page objects. Disk storage was deemed inappropriate both for performance and access granularity reasons. This led us to introduce a new stable storage design, called Stable Transactional Memory (STM), built from banks of RAM memory with a built-in transaction facility [Banâtre *et al.* 91b].

A RAM stable storage is based on the use of two separate banks of RAM memory located in two independent machines in order to ensure availability. So that STM transactions and stable objects could be managed as simply as standard C++ objects, we build a C++ interface library that hides the implementation of the STM from the programmer [Muller *et al.* 91].

Two versions of STM were successively designed during the FTM project: a hardware based implementation made from two boards and a software implementation not requiring any specific hardware and which is thus easily portable. The hardware-based STM was the first designed: it offers fine-grained object protection against processor crashes and high performance STM transactions. A prototype has been implemented for the Intel IlbxII local bus. Several considerations subsequently led us to redesign this STM in order to make it easier to port to new platforms. This was done by allocating STM RAM banks in processors' main memory and by integrating STM functionality, previously implemented by hardware, into the software C++ library.

3.1 Description of the Hardware-based STM

Our design goal for the hardware-based STM was to develop a high performance stable storage board which internally masks single hardware faults and power failures. In addition, we wanted to detect standard machine errors so we added a hardware mechanism capable of verifying the correctness of processor accesses to memory. This was achieved by integrating the object paradigm into the STM hardware: addresses generated by the processor are checked against the bounds of objects residing in STM.

The hardware STM is made up of two separate boards, one connected to the primary processor, the other to the backup (see figure 3). Each board contains a fail-stop controller and a 32 Mbyte memory bank. The boards are connected together using a fast serial link offering a 16 Mbits/sec bandwidth. To provide efficient access to STM data, stable objects are mapped into the processor's address space and the processor is able to modify the primary bank directly.

3.0.1. Basic Functionality

At any given moment, only the controller of the board attached to the active processor is running. The active controller's task is to implement object protection and to manage transactions. More specifically, an *Open* command first makes the object accessible to the processor, thereby permitting the construction of the list of objects accessed or modified by a transaction. A *Close* command hides the object: any tentative access to a closed object are detected and signaled as an error by the controller. The *Commit* command is executed internally in the STM by the controller using a two-phase protocol:

- Working phase: the processor modifies objects in the primary memory bank.

- Phase 1 (*Precommit*): all objects are closed to prevent further modification.

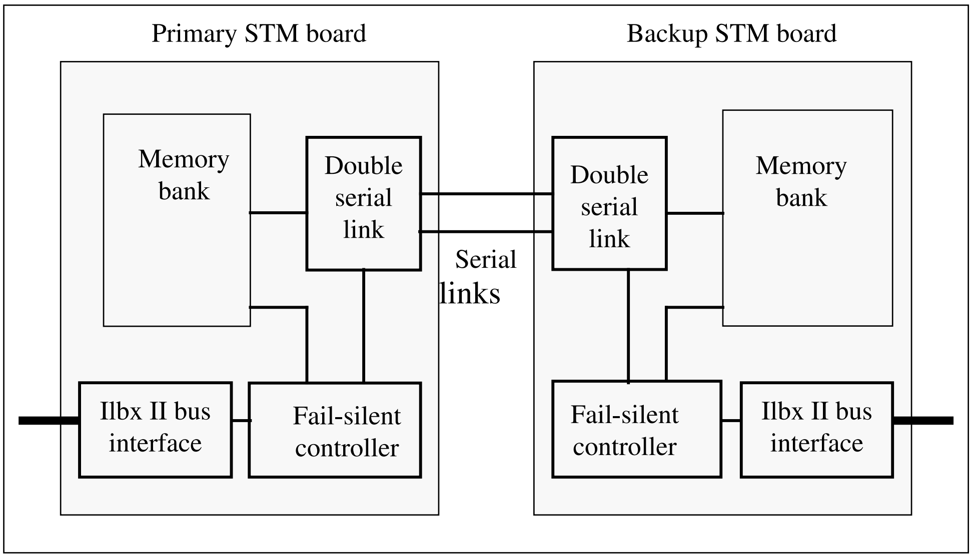

Figure 3 : Structure of the STM boards

- Phase 2 (*Commit*): when receiving the commit command, the controller copies modified objects over the serial link from the primary memory bank to the backup.

If a processor failure occurs during the working phase, the previous version of the objects, which are still in the second memory bank, are copied back onto the first; the update is not successful. If a failure occurs during phase 2, the first memory bank will still contain the new version of the objects, so this phase can be restarted. Only on completion of phase 2 is the update considered successful.

From the operating system standpoint, it can be necessary to group several possibly distributed transactions so that either all transactions commit, or all transactions abort. Transaction grouping is efficiently supported by the STM as part of the *Precommit* phase. In this phase, no object accessed by a transaction is or can be opened. Thus, if the processor fails, objects are still consistent and the transaction does not have to be aborted. When receiving the group decision from the operating system to either commit or abort, the processor forwards the decision to the transaction.

The controller is also able to take decisions. For instance, it can decide that a processor access is faulty and then initiate a reconfiguration by deconnecting the main processor and signaling the failure to the backup. Detection of a processor crash is done by a watch-dog timer which is regularly reset.

3.0.2. Advanced Transaction Support

In order to provide efficiency and convenient support for the operating system, the STM supports multiple concurrent transactions. Multiple concurrent transactions permit any process to have one or more private STM transactions. From the micro-kernel standpoint, the STM can be viewed as an atomicity supporting co-processor: when a scheduling decision occurs, the current transaction of the elected process becomes the activated transaction. Transaction switching can also be effected from the programming level, thus permitting an application to manage stable objects within separate transactions.

3.0.3. Masking a Single Internal Failure

When designing the STM, special care was taken to mask single internal hardware failures. For instance, the two controllers have direct access to the two memory banks and every processor access goes through a controller. If the controller fails, the contents of memory may be destroyed. To prevent this, a controller possesses the fail-silent property: it is made from two identical RISC R3000 single CPU chips that always do the same job. Their outputs are compared before being sent to the memory banks. If any difference is found, the comparator disconnects the chips and declares the fail-stop controller to be in failure. If both chips' outputs are the same, access to the memory banks is permitted.

Access to STM must always be assured to permit service continuation, even in the event of a controller failure. We therefore consider that a failure of the active controller is equivalent to a failure of the processor - the backup controller and processor become active. The backup controller aborts current internal STM operations and restores a consistent object state. In order to do this, all algorithms executed by a controller manipulate robust data structures [Taylor *et al.* 80].

The memory banks are made up of a set of memory chips organized by columns (i.e., each chip is referenced by different address bits). In order to tolerate soft (non permanent) errors when reading memory, an Error Correcting Code (ECC) mechanism is used to correct a false bit. Soft errors are the result of spontaneous alterations of the memory chip and occur from time to time. The ECC is able to correct an error on one bit and detect an error on two. If a chip fails, i.e., the error is hard (permanent), the ECC is able to correct the false bit, deliver the correct memory word value to the processor, but cannot write it back to the memory chip. Hence the ECC cannot recover from subsequent soft failures. In such cases, the *bit steering* technique [Hardell *et al.* 90] is used to copy the contents of the faulty memory chip to a spare one on the board. The fault is reported to the kernel and to the operator, asking the latter to replace the faulty chip during a scheduled maintenance. Using the aforementioned mechanisms, a memory chip failure is masked from the processor.

3.2 Lessons from the Hardware-Based STM

The hardware-based STM was successful from a performance point of view. This was due to the internal controller being able to asynchronously execute complex functions, thus freeing the processor to execute other tasks. The STM provides high level functionality such as transactions and objects, both of which were lacking in our Gothic stable storage design [Banâtre *et al.* 88]. Finally from the programming point of view, the C++ interface to STM means that using stable objects is as simple as using standard C++ objects.

When designing the hardware-based STM, we had the underlying goal of being able to create a modular STM board which could be easily ported to different platforms. For instance, machine specific functions such as the bus interface were designed as a block separate from the memory banks and the controller. Nevertheless, porting the hardware STM is not as simple as we had hoped. The first reason is due to the size of the fail-stop and error detection mechanisms which led our IlbxII bus design to take up a “large” MultibusII board. The second reason is the difficulty encountered in implementing dual powered boards in a standard machine. The third reason is the conflict between the protection mechanism and the evolution of modern processors which require large internal caches to run efficiently: when using the hardware STM, data caches have to be disabled or flushed before closing an object. This is not a drawback for our FTM prototypes since they ran on 68030-based industrial single boards possessing only a small 256 Byte data cache.

For all of the above reasons, we decided to move to a software design which could be independent of the architecture and which could withstand the evolution of machines.

3.3 The Software-Based STM

Our requirements for the software-based STM meant that functions previously performed by the STM hardware had either to be performed by software or else discarded. For instance, STM RAM banks are allocated by software in the standard RAM main memory of the two machines. Unfortunately, checking processor accesses to currently opened objects is not possible unless a trap is generated on each memory access; this would induce an unacceptable execution overhead. Consequently, the concept of fine-grained object protection was abandoned.

From the programmer’s view, there is no difference between the two STM implementations: all functions which were performed by hardware have been moved to the software C++ library. The transaction list of accessed objects is now managed as part of the *Open* procedure. Protection related commands, such as *Close,* which previously hid an object from the processor, are represented by null procedures.

One potential source of performance bottleneck in the software STM is the duration of transaction commit because of the time needed to copy modified objects from the primary machine to the backup machine on transaction commit. To reduce this duration, and to speed up inter-bank copies of objects, we decided to use a fast point to point serial link similar to the one designed for the hardware-based STM.

3.0.4. Implementation of the Fast Serial Link

Using a dedicated point to point serial link means that the use of general purpose communication protocols can be avoided and thus round-trip latency reduced. Today, there exist many ways to implement a point to point serial link. A simple solution consists of using either the Fast Ethernet or the ATM technology which currently offers a physical 100-150 Mbits/sec bandwidth. However, as these technologies were not available when we started the FTM project, we chose to design a fast serial link using a high speed transmitter/receiver [Gazelle 90].

The fast serial link itself is a simple board built from the transmitter/receiver, a 16 Kilobyte receive FIFO and some TTL circuits (see figure 4). The transmitter/receiver registers are directly mapped into the process address space, allowing data transfer from one machine to the other by means of the standard memory move processor instructions. Our implementation for the FTM prototype (e.g., MultibusII board) is based on 200Mbit/sec Gazelle circuits that provide a usable bandwidth of 48 Mbits/sec, which is only limited by the speed of the processor's local bus interface [Muller & Prunault 93].

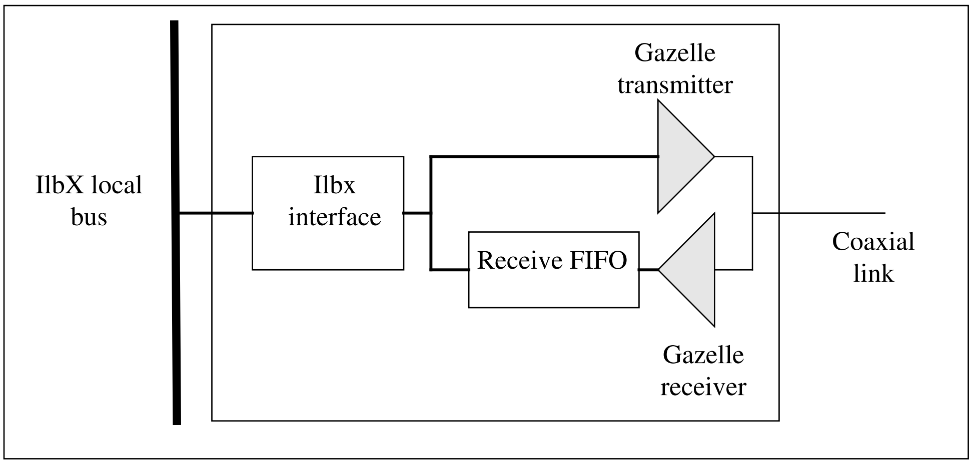

Figure 4 : Structure of a fast serial link board

4.0.1. Management of Objects in STM Banks

In contrast to the hardware-based STM where memory banks were specially designed to mask internal failures and to survive power failures, availability of a RAM bank of the software-based STM is dependent upon the machine in which it is allocated. Management of objects within a transaction is consequently performed as follows (see figure 5):

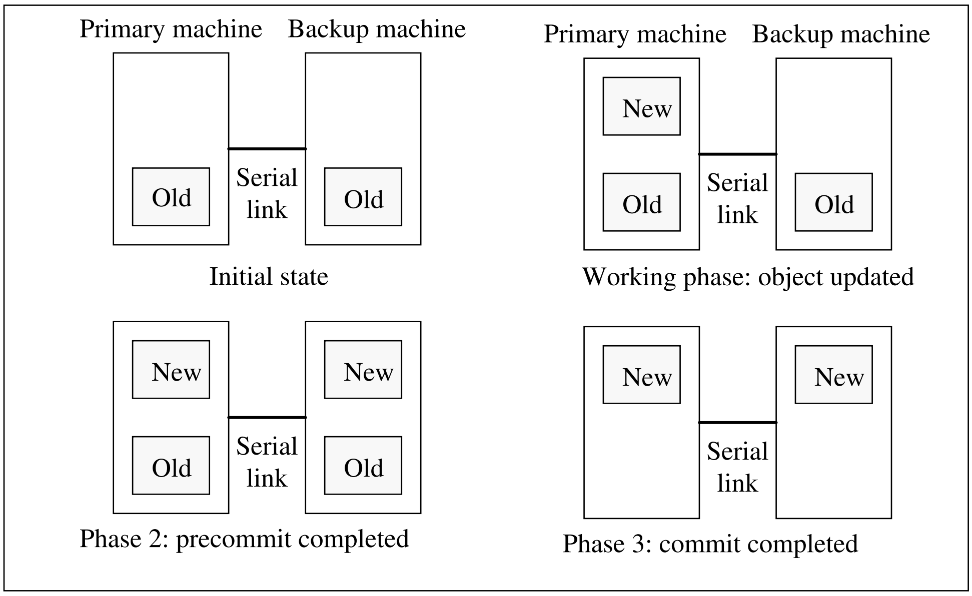

Figure 5 : Object version management using the software STM

- Initial state: the previous transaction commit has been completed.
- Working phase: when the processor modifies an object for the first time, a memory copy is allocated for the new object version.
- Phase 1: modified objects are sent to the backup processor through the serial link. The backup processor also allocates a memory copy for the new object version. When phase 1 completes, the backup machine possesses the old and new versions of all modified objects.
- Phase 2: if the transaction belongs to a group, the transaction waits for the group decision before executing phase 3.
- Phase 3: the final decision, *commit* or *abort,* is propagated to the backup processor. The two processors then delete old versions of objects in the *commit* case, and new versions in the *abort* case.

Assuming that there is only one machine failure at a time, the following situations may arise:

- *failure of the primary machine in the initial phase*; nothing needs to be done since no object is modified.

- *failure of the primary machine in working phase or phase 1*; the transaction is aborted and the backup machine takes over using the old (initial) versions of objects stored in the backup bank.

- *failure of the primary machine in phase 2*; the backup machine possesses sufficient information to commit or abort, and thus resumes waiting for the group decision.

- *failure of the primary machine in phase 3*; the failure does not affect the backup machine which can complete the phase.

- *failure of the backup machine*; at a hardware level, a backup failure does not affect the primary machine which is able to proceed with its work. However, in situations where a computation running on the backup depends on another running on the primary, a failure of the backup requires a software abort of the dependent computations running on the primary machine. As the old object versions on the backup machine are destroyed, a software abort requires the availability of another copy of the old object versions. To deal with such a situation, we keep an old object version on the primary machine when modifying an object. It should be noted that if the software configuration prevents software dependencies between the backup and its primary machine, then keeping the old versions of objects on the primary machine is not necessary.

3.4 Results From the Software-Based STM

From the standpoint of performance, using a separate point to point link instead of the normal local area network, means that network protocols are unnecessary and communication latency with the backup machine is reduced. Furthermore, the implementation of the software STM does not require modification to the kernel memory allocator nor to the scheduler. Allocation of STM objects is completely performed using standard memory allocation primitives.

Finally, the main benefit of the software STM is that it permits the FTM architecture to be built without the use of dedicated hardware. Today, the software-based STM can be implemented with ATM (100-150 Mbits/sec) or fast Ethernet (100 Mbits/sec) without requiring the design of a specific serial link. Consequently, FTM implementations only necessitate the addition of a second network interface to the user machine.

4 Operating System Issues

In this section, we describe more precisely issues related to the efficient implementation of consistent checkpointing on top of a micro-kernel using a reliable client-server model.

4.1 The Reliable Client-Server Model

Our computing model basically consists of a set of *reliable server processes* accessed by *user processes*. A server abstracts a system resource (e.g., a memory segment or a console). A server is defined by its *access interface* and its *internal state*, where the access interface lists the operations used to access the internal state. Communication between clients and servers uses the RPC model [Nelson 81].

A process (user or server) is the execution of a sequential program. The process code consists of a sequence of instructions which either manipulate the internal state or perform an RPC. The state is made up of the processor registers, the process stack and some static structures. To get fault tolerance, we associate a *persistent state* (i.e., a local checkpoint), resilient to a single machine crash, with each process. User and server persistent states are both stored in STM. However, their implementations differ for performance reasons since our objective is to reduce the size of persistent states and thus, checkpoint durations

```
                 Server process
Restart ──────────► Init code;
Checkpoint* ──────► Do
                      Receive (call);
                      Operation execution;
                      Return (result);
                    End;
```

Figure 6 : Server structure

Reliable Servers' Persistent State: A server performs a cyclic task (see figure 6). It receives calls, processes them and returns the results. Whenever the server is waiting for a call, the stack and control registers always have a fixed value and thus, their contents can be easily rebuilt by re-executing the **init** code (see figure 6). When checkpointing a global consistent state, we insist that the local checkpointing action can only be performed while the server is awaiting a call. This allows us to efficiently reduce the size of a server's persistent state to include only the static structures stored in STM (i.e., stable objects) by discarding the stack and CPU control registers. A server's state is managed by an

STM transaction which is initialized when executing the first call occurring after a checkpoint and commited at the saving of the next checkpoint.

To restart a reliable server, the STM static structures are first restored. From the values of these structures, the server **init** code is able to determine whether the execution is the first following a restart and thus whether the stack and processor registers should be re-initialized.

User Process' Persistent State: No assumption can be made about the behavior of user processes. Consequently, a user's persistent state must contain the processor registers as well as the stack and data segments. A naïve solution would be to allocate each of these segments in one large STM object. However, this would be inefficient since checkpointing would entail copying a large amount of unmodified data. To reduce the size of user checkpoints, a solution based on the virtual memory management system can be used. This approach, introduced into the Targon system [Borg *et al.* 89], allows the calculation of the set of pages modified since the preceding process checkpoint and thus the set of persistent memory pages to be flushed on checkpointing. Since the micro-kernel technology provides an interface to the MMU through pagers, such a solution can be simply implemented without having to modify the kernel. In FTM, this is done by means of a recoverable virtual memory server which manages a persistent copy of pages in STM [Rochat 92].

Recovery Unit: A *recovery unit* is the sequence of instructions performed by a (user or server) process since the saving of its preceding persistent state. The commitment of a recovery unit is the atomic saving of the process' persistent state and the continuation of this process with a new recovery unit. Aborting a recovery unit leads to the restoration of the process' persistent state and to the restart of the process with a new recovery unit. An STM transaction is associated with each server's recovery unit: the STM objects which contain the server's persistent state are manipulated within the scope of this transaction. Recovery units are uniquely named in the system in order to distinguish successive re-executions of the same process.

Dependencies: To compute a consistent distributed persistent state, it is necessary to keep track of the dependencies that arise when a client, which can be an application or a server, calls a server operation. More precisely, dependencies exist between clients' and servers' recovery units. We define dependencies as follows:

A recovery unit *Ru2* depends on a recovery unit *Ru1* ($Ru_1 \rightarrow Ru_2$), if aborting Ru_1 necessitates the abortion of Ru_2. In other words, if Ru_2 commits then Ru_1 cannot later abort. To ensure the dependency $Ru_1 \rightarrow Ru_2$, it is necessary to commit Ru_1 before Ru_2. For the dependency $Ru_1 \rightarrow Ru_2$, Ru_1 is said to be the *predecessor* of Ru_2 and Ru_2 the *successor* of Ru_1.

Dependency tracking: the programmer of a reliable server must associate one of the two attributes, *read* or *update*, with each operation of the server; this attribute specifies whether the operation modifies the abstract state of the server or not (see section 5). It should be noted that some operations may modify the *real* representation of a server's state without modifying the *abstract* state. For instance, reading a file fills the main memory cache, but leaves the file unchanged. Therefore, attributes of operations are not propagated through nested calls. The following scenarios may arise when a client calls a remote operation:

1. The operation *updates* the state of the server.
2. The operation *reads* the server's state which **has been modified** since the beginning of the server's recovery unit.
3. The operation *reads* the server's state though this state **has not been modified** since the beginning of the server's recovery unit.

In the first scenario, the server state is modified by the client operation. Thus, if the client fails, its operation on the server has to be undone; this gives the dependency $Ru_{client} \rightarrow Ru_{server}$. In the same way, a failure of the server destroys all modifications performed by the client. So we also get the dependency $Ru_{server} \rightarrow Ru_{client}$.

In the second scenario, the client reads a non-persistent state in the server. If the server fails, the state observed by the client is lost, so the recovery unit of the client has to be aborted; there is thus a dependency $Ru_{server} \rightarrow Ru_{client}$. In contrast to the previous case, a failure of the client does not affect the server since no modification of its state has been made by the client.

In the third and final scenario, the client operation reads the persistent state of the server. Thus even if the server fails, the state seen by the client is persistent so no dependency results.

With each recovery unit, we associate a local persistent graph containing the set of recovery units upon which it depends. Dependency tracking is performed within the RPC mechanism; prior to an RPC return, the newly created dependency is added to the caller's and called recovery units' local graphs.

4.2 Implementing Blocking Consistent Checkpointing

Our main requirement in the checkpointing algorithm design was to leave a server programmer free to initiate the saving of a checkpoint. Consequently, a checkpoint protocol can be initiated at any time by any recovery unit, whether it is a server or a user application. In the following, this recovery unit will be designated the *initiator*. To build a global consistent state, the dependencies stored in all graphs are used to determine the set of recovery units which depend on the initiator; this is implemented using a chase protocol [Merlin & Randell 78]. The resulting set is merged into a distributed atomic action with the initiator becoming the coordinator of the atomic action. The atomic action is then committed using the popular two-phase commit protocol [Gray 78]. Consequently, checkpointing is implemented in three phases:

- Phase 1: a consistent state is built by transforming the distributed graph of dependent recovery units into a distributed tree.
- Phase 2: precommit the atomic action (e.g., dependent recovery units) using the previously computed tree.
- Phase 3: commit the atomic action.

Dependencies are also used during crash recovery to determine a previous consistent state for the system and to merge the dependent recovery units into a distributed atomic action. In this case, the resulting action is aborted. It should be noted that due to the dependency rules, the consistent set built for a checkpoint decision is not always the same as the set built for crash recovery.

We wanted the FTM system to operate as much as possible as a standard non fault tolerant system. Thus, in order to allow free sharing of resources, we do not enforce serializability. Consequently, separate client recovery units may be joined within the same consistent set. For example, in figure 7, the first read call from $client_a$ does not create a dependency since the server's recovery unit has just started and so the operation reads the server's persistent state. On the other hand, the second read call from $client_a$ creates a dependency since the server's state has been modified by $client_b$. Consequently, when the server takes a checkpointing decision, the consistent set of recovery units will contain the recovery units of $client_a$, $client_b$ and the server.

In practice, checkpoint decisions are taken either regularly by a library managing application recovery units or by a few servers, e.g., those dealing with I/O. It follows that very few server programmers need to code "checkpoint" instructions and application programmers need never do so. Finally, it should be noted that the initialization of a new recovery unit is transparent to the programmer since it is performed as part of the commit of the previous recovery unit. We now detail the basic checkpoint protocol and extensions which allow us to deal with special cases such as dependency chasing and the merging of multiple initiators. We do not describe the crash recovery protocol since it is similar to the checkpoint protocol.

7.0.1. Basic Checkpoint Protocol

The checkpoint protocol is implemented using three phases: *build_atomic_action*, *precommit* and *commit*. We have chosen to implement these

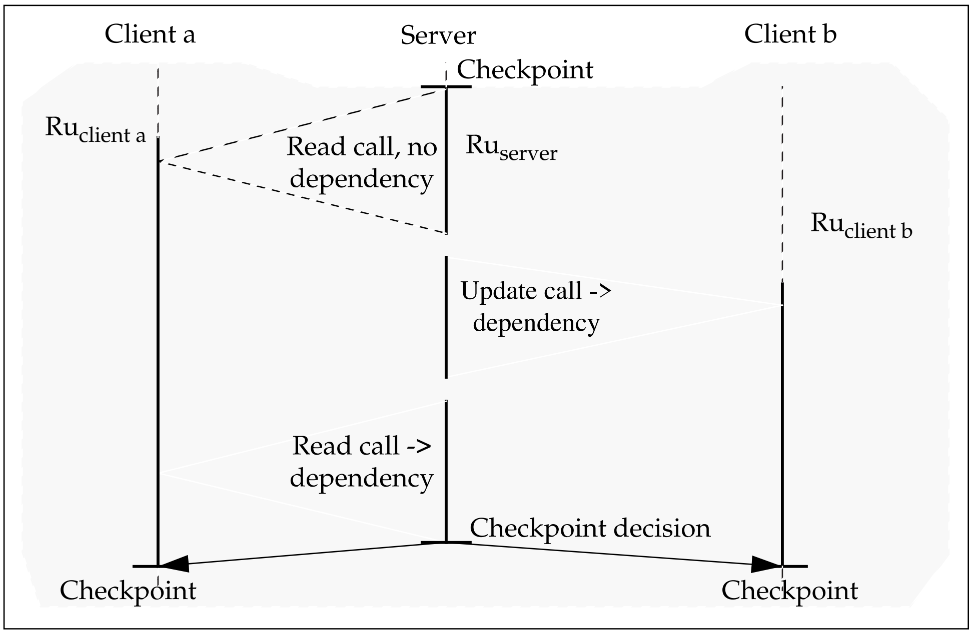

Figure 7 : Building a distributed consistent state

phases in a completely distributed way using waves [Raynal & Helary 90]. The goal of the *build_atomic_action* phase is to compute the consistent distributed state by building a distributed tree of dependent recovery units. In the two other phases, the tree is used to propagate the *precommit/commit* waves.

- Phase 1 (*build_atomic_action*): When a user process enters the phase *build_atomic_action*, it stops to prevent further dependencies being added. However, servers still handle incoming requests. To build the tree, a wave *build_atomic_action* is multicast from the initiator to the recovery units upon which it depends (i.e., its predecessors). When receiving this wave, a recovery unit enters the phase *build_atomic_action*, becomes a *sub-node* of the wave sender, which is called its *node*, and propagates the wave to its own predecessors. If the recovery unit is already a sub-node of another node, it just returns a negative acknowledgment to the sender. When all recovery unit predecessors have replied, a positive acknowledgment is sent to the recovery unit's upper node. Only the predecessors that have replied positively are kept in the tree. The phase *build_atomic_action* terminates when the initiator of the action receives all acknowledgments.

- Phase 2, 3 (*precommit*, *commit*): The *precommit* and *commit* phases are wave implementations of the popular two-phase protocol

[Gray 78]. During the precommit phase, the *precommit* wave is multicast. Upon receipt of this wave, a server stops servicing calls, precommits its associated STM transaction (see details in section 5) and replies with a positive acknowledgment. When the initiator has received all acknowledgments and they are positive, it performs the commit phase by multicasting the *commit* wave. Finally, when receiving the *commit* phase, a recovery unit commits its associated STM transaction and normal execution is resumed with a new recovery unit.

It is possible for the *build_atomic_action* wave to be sent to a recovery unit which has already committed or is executing phase 1 or 2. In such a situation, the recovery unit terminates the current commit and replies with a negative acknowledgment so as not to be added to the new atomic action tree.

7.0.2. Dependency Chasing

As we mentioned above, server recovery units can only be committed while the server is waiting for a call. Thus, before suspending service in the *precommit* phase, we must ensure that all nested calls under execution have completed. In order to do so, the phase *build_atomic_action* integrates a chase protocol.

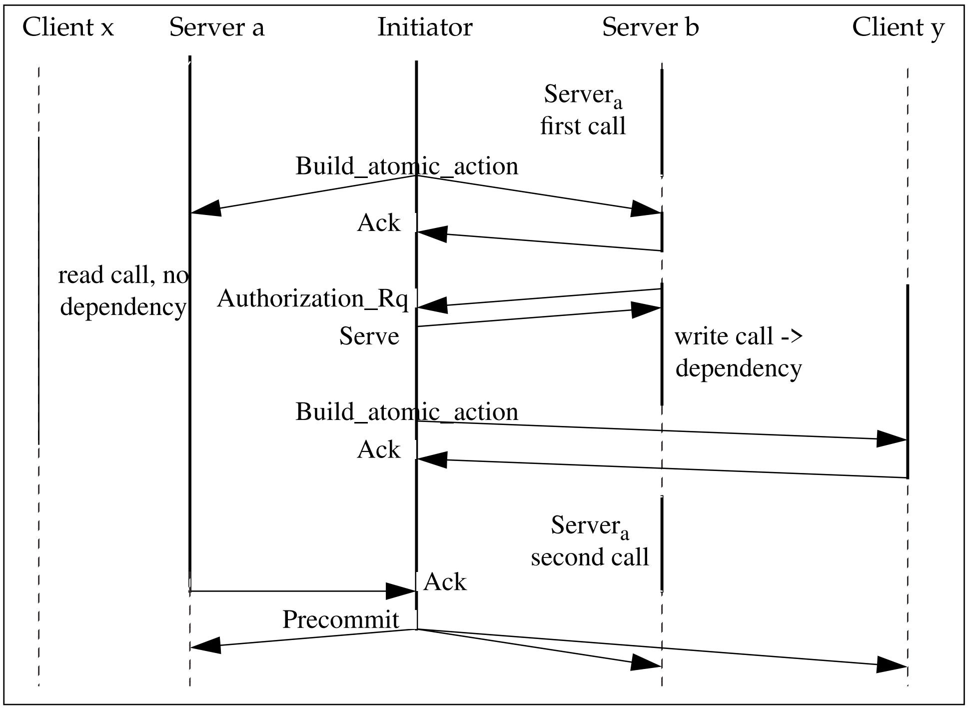

Figure 8 : Chasing recovery units

When a reliable server enters the phase *build_atomic_action*, it will request authorization from the initiator to service further operations. The initiator replies *serve* while in phase *build_atomic_action*, otherwise it replies *defer*. The authorization mechanism permits handling of incoming operations up to the building of a distributed consistent state. For instance, in figure 8, chasing permits $server_a$ to execute its second call to $server_b$, which in turn permits it to complete $client_x$'s call. Deferred operations will be serviced by the reliable server after execution of the commit protocol.

If any dependency is added during the phase *build_atomic_action* as a result of initiator authorization, the dependent recovery unit is directly connected to the initiator who immediately sends a *build_atomic_action* message to the new dependent recovery unit. For instance, in figure 8, $client_y$'s recovery unit will be added to the dynamic action since the *build_atomic_action* phase has not completed. To optimize the authorization mechanism, the initiator is not called by a recovery unit if the latter has not sent up its acknowledgment to its node since the recovery unit is obviously aware that the *build_atomic_action* phase has not yet terminated.

4.2.1 Merging Multiple Initiator Graphs

As there is no fixed top-level initiator, a recovery unit may participate in several simultaneous dynamic actions. When this occurs, the trees have to be merged and a single initiator has to be elected. To merge two trees, we choose to keep as initiator, the initiator whose recovery unit possesses the smallest UID and then to connect the other initiator as a sub-node of the new one.

Merging is detected whenever a recovery unit receives a *build_atomic_action* message from a different initiator while in phase 1 (see figure 9, step1). The recovery unit (Ru_2) then sends a *merge* message to the initiator possessing the highest unique identifier (Ru_3), asking it to become dependent on the other initiator (see figure 9, step2). The recovery unit which receives a *merge* message loses its initiator status and sends an *add_dependency* message to the elected initiator (e.g., Ru_1 in figure 9, step3). The latter then adds this recovery unit to its list of dependent recovery units. As in a normal *build_atomic_action* phase, the dependent recovery units (Ru_2, Ru_3) become nodes of the dynamic atomic action tree only after receiving all acknowledgments from their successors.

The merging algorithm also handles the situation where a recovery unit has already lost its initiator status when a message *merge* or *add_dependency* is received. Receipt of multiple *merge* messages occurs when several trees are merged simultaneously (see figure 10). When receiving an additional *merge* message, the recovery unit (e.g., Ru_5 in figure 10, step 3) forwards it to the initiator possessing the highest unique identifier (e.g., Ru_3 in figure 10, step 4). Similarly, if a recovery unit receives

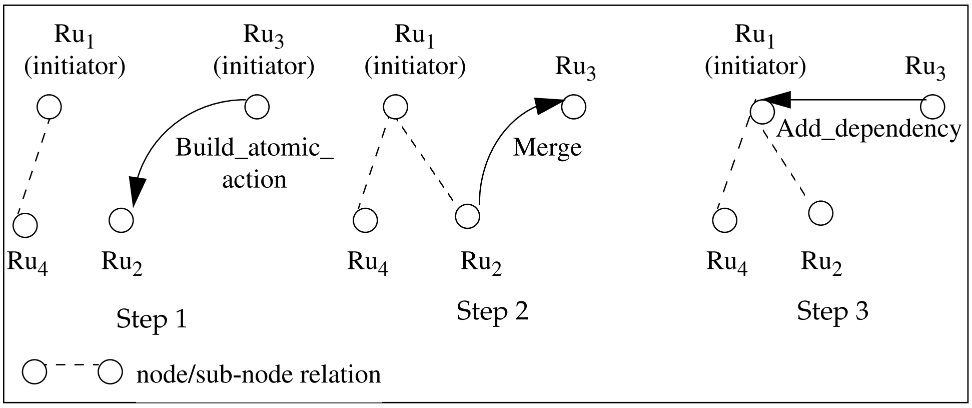

Figure 9 : Merging multiple initiator graphs

an *add_dependency* message after losing its initiator status, it forwards the message to its current initiator.

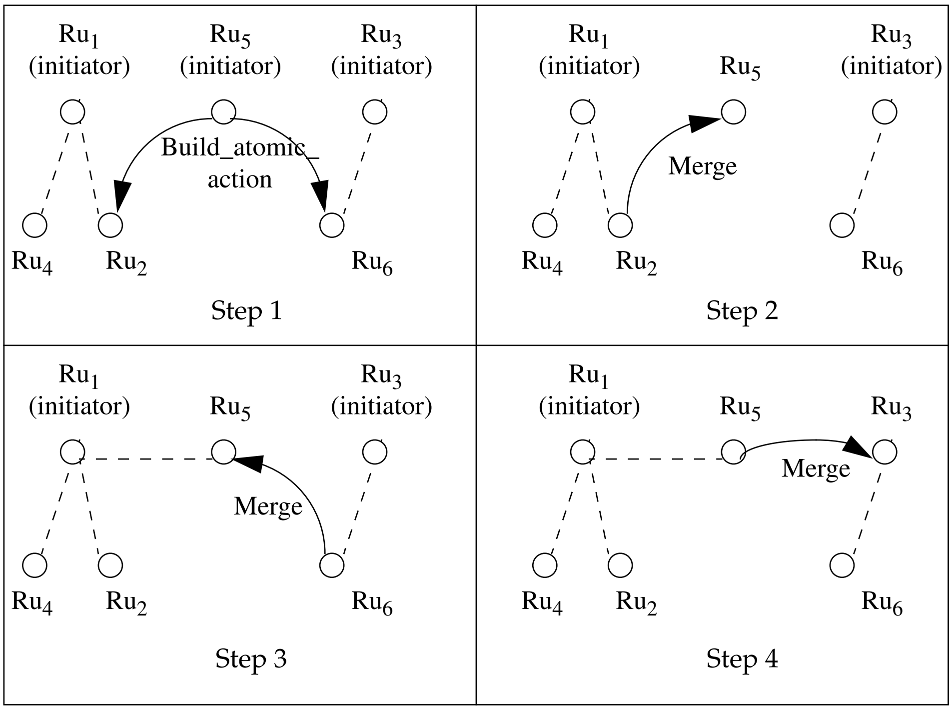

Figure 10 : Concurrently merging multiple graphs

10.0.1. Dealing with Crashes While Committing

When a recovery unit crashes while running the commit protocol, two decisions may be taken at restart, depending on whether the recovery unit state is restored from persistent memory or not. If the recovery unit has performed the *precommit* phase, it completes the commitment of the distributed action by waiting for the final (*commit*/*abort*) decision from the initiator. If the recovery unit is in the *build_atomic_action* phase, the building of the atomic action is cancelled and the crash recovery protocol is started: the tree of recovery units which are dependent on the failed unit (i.e., its successors) is built. It should be noted that this new set is generally not the same as in the previous distributed action.

Due to crashes, messages in a wave may be lost. To tolerate this, the protocol keeps a list of recovery units to which a wave has already been sent in STM. If no acknowledgment from a recovery unit is received within a finite time, the wave is resent. To avoid unpredictable effects due to multiple receptions of the same wave, the message is simply discarded though a re-acknowledgment is returned.

4.3 Implementation of the FTM Model on Mach 3.0

The FTM model has been implemented on top of Mach 3.0 [Accetta *et al.* 86] [Loepere 93] by adding specific mechanisms to deal with failures. Normal Mach entities such as tasks, ports, threads and memory objects are destroyed by a processor failure. To restart applications and reliable servers safely after a crash, we have developed the following three stable entities: *stable tasks*, *stable memory objects* and a *persistent ports*.

Stable task: a stable task defines an execution environment. It is used to implement reliable servers and user application tasks. As with standard tasks, a stable task contains access rights and a virtual address space. Should a processor fail, the environment of the stable task is restored at restart time on the backup processor. Stable memory objects are then remapped at their previous address and a thread is restarted from the **init** code of reliable server tasks (see section 4.1) or, for applications, from the address in the saved program counter.

Stable memory object: a stable memory object can be viewed as a logical STM. It allows access to the real STM and allocation of stable objects. A stable memory object is private to a stable task and is always remapped at the same logical address when restarting after a crash.

Persistent port: a persistent port has a naming function. It allows FTM entities (i.e., stable tasks, stable memory objects) to possess a unique name which resists crashes. Persistent ports are implemented using a UID and a specific name server. In a primitive design [Banâtre *et al.* 91a], sending and receiving messages on a port had the property of being atomic and all messages were stored in STM. Later on, we relaxed theses two properties as they led to an expensive design and were only useful for managing acknowlegements within the checkpoint and crash recovery protocols. In our current implementation, acknowlegements are managed in STM by the protocols themselves.

These three stable entities have been implemented in libraries and did not require kernel modifications.

5 Programming Reliable Servers

As for application programmers, our goal for programming reliable servers was to mask fault tolerance from a server designer, that is, programming a reliable server should be no more difficult than programming an ordinary one. Nevertheless, achieving the degree of transparency obtained for applications programming is not possible since the purpose of reliable servers is to implement the applications' checkpoints. Furthermore, for servers such as those dealing with I/O devices, complete transparency is not desirable as the programmer may want to perform specific actions during the saving of a checkpoint or the recovery from a crash. For example, when designing a reliable window server, the programmer must insert a *refresh_window* procedure in the restart code of the recovery unit.

Object-oriented languages provide important features such as abstract data types, generic functions and inheritance. The inheritance mechanism enables classes to inherit features of ancestor classes. For these reasons, we chose to program reliable servers using the object oriented language C++. We use inheritance for two purposes: first, to allocate the server state in STM, and second, to provide default checkpoint/crash recovery procedures while allowing for their redefinition. Such a technique has also been used in the projects ARJUNA [Dixon & Shrivastava 87] and AVALON [Eppinger *et al.* 91].

5.1 Reliable Server Structure

To offer transparency, our approach relies on an automatic generation of a reliable server from a C++ server written by the programmer. A reliable server *xx_reliable_server* implements both the standard server operations which manage the resource *xx,* and the checkpointing/crash recovery protocols. Consequently, when a request message is received by the reliable server (see figure 11), the operation called may be either a server operation of the resource *xx*, or an operation that implements a protocol wave or acknowledgment and manages the recovery unit. Our idea is thus to automatically create the code of a reliable server by merging the access interface of the standard *xx* server with the protocol interface of the recovery unit.

A reliable server for a resource *xx* is built from three main C++ classes (see figure 11): *xx*, *Recovery_unit* and *xx_reliable_server*. The *xx* class implements the *xx* resource and is written by the programmer. The *Recovery_unit* class is a built-in class that implements the recovery unit and the distributed protocols. The *xx_reliable_server* class is generated from the access interface (i.e., operations on the resource) and from the protocol interface (i.e., waves and acknowledgments). The *xx_reliable_server* class inherits from a built-in *Generic_reliable_server* class which

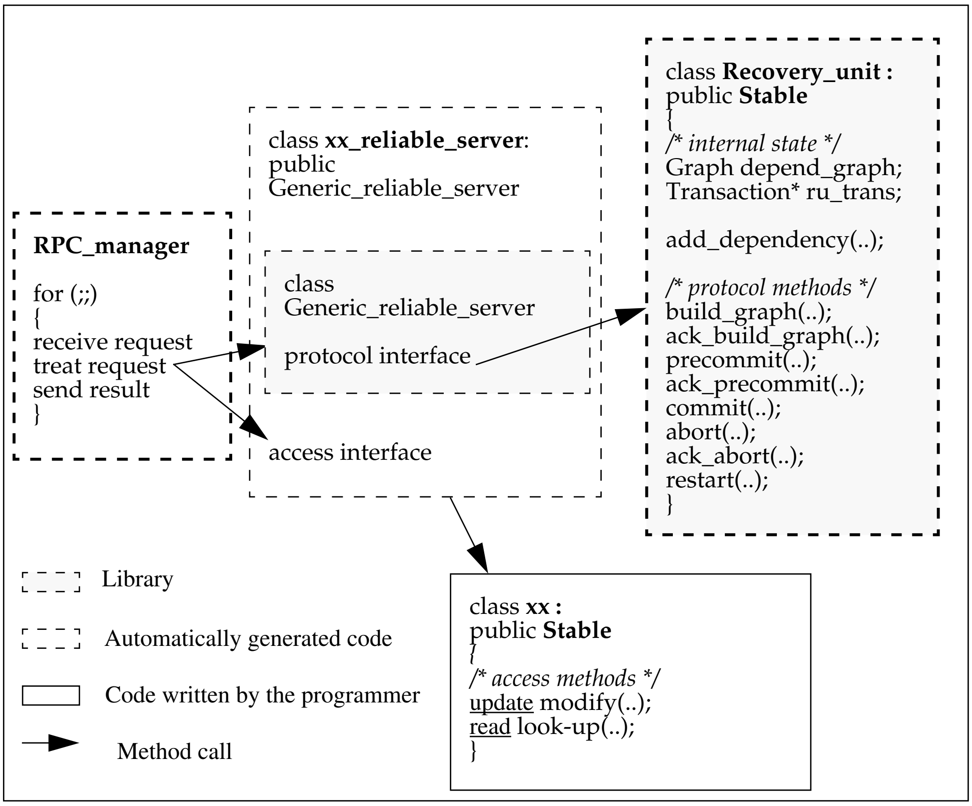

Figure 11 : Reliable server structure

offers a default implementation of the protocol interface. The purpose of calling protocols through an inherited class is to permit the programmer to redefine the operations if he wishes to execute specific actions during the protocol's execution.

The reliable server interface is called by RPC using communication ports. Translation from a system RPC to the C++ reliable server interface is performed by a stub, the *RPC_manager*, which is also generated in C from the definition of the *xx* class.

5.2 Programming a Reliable Server: the Window Server Example

To design the *window* class, the programmer specifies the internal state and the access methods of the resource (see figure 12). In our example, the internal state is an array of characters; access methods read or write a character at a given position in the window, and refresh it. Persistence of the resource state is achieved through inheritance by the *window* class from the *Stable* class: *window*'s instances are then allocated in STM. The programmer should also give hints to the reliable server generator: all access methods need to be prefixed with the read or update keyword depending on whether or not the methods look-up or modify the resource's internal state.

```
class window : public Stable
{
/* internal state */
/* array of characters within the window of size H*W */
char A[H*W];

public:

/* access methods */
/* read or write a character at a given position in the window */
read void read(char& c,int x, int y);
update void write(char c,int x, int y);
/* refresh the window */
read void refresh();
}
```

Figure 12 : Definition of the *window* class

The *window_reliable_server* class is generated from the *xx* class using a preprocessor*. The code is generated as follows (see figure 13): first, the *window_reliable_server* class inherits protocols from the *Generic_reliable_server* class. Second, for each method of the resource, an equivalent method is created with the addition of parameters which permit the caller's recovery unit to be identified and to return the dependency. The appropriate method of the *resource* object (i.e., the window) is then called within the scope of the server's recovery unit.

In the case of the reliable window server, the window needs to be refreshed as part of the crash recovery protocol. To do this, the programmer must redefine the restart method of the protocol to include the refresh-window command.

* This is currently implemented using a modified version of the Mig tool provided with the Mach micro-kernel.

```
/* Inheritance of the class Generic_reliable_server to add protocol interface */
class window_reliable_server : public Generic_reliable_server
{
public:

/* access window interface */
void read(char& c,int x, int y,ident_ru_t client_ru,
dependency_t& depend);
void write(char c,int x, int y,ident_ru_t client_ru,
dependency_t& depend);
void refresh(ident_ru_t client_ru,dependency_t& depend);
}


/* Generated code the access write method */
void window_reliable_server::write(char c,int x, int y,
ident_ru_t client_ru,
dependency_t& depend)
{
  /* Execution of the access method within the recovery unit scope */
depend=activate_recovery_unit(UPDATE, client_ru);
resource->write(c,x,y);
deactivate_recovery_unit(UPDATE);
}



/* Redefinition of the restart method in order to insert the refresh of the window */
void window_reliable_server::restart()
{
/* Execution of the default restart method */
Generic_reliable_server::restart();

/* Then execution of the access refresh method */
resource->refresh();
}
```

Figure 13 : Definition of the *window_reliable_server* class

5.3 Recovery Unit Implementation

An STM transaction is associated with each recovery unit; both possess the same lifetime. The commit operation of a recovery unit is thus straightforwardly implemented by committing its associated STM transaction. This leads to a copy, from one bank to the other, of the STM objects modified within this transaction. Due to the multi-transactional functionality of the STM, it is necessary to activate the recovery unit transaction before calling an access method of the resource. This is implemented by the *activate_recovery_unit* method of the *Generic_reliable_server* class (see figure 14). This method also computes the dependency between the client's recovery unit and the local recovery unit and atomically adds the dependency to the dependency graph.

```
class Generic_reliable_server : public Stable
{
Recovery_unit* recovery_unit;        /* Pointer to the C++ object which implements
                                          the server recovery unit */
Stable* resource;                    /* Pointer to the C++ stable object which implements
                                          the managed resource */
ident_ru_t      server_ru;           /* server recovery unit identifier */
boolean_t modified;                  /* tells if the resource internal state being modified */

/* delimite the recovery unit scope within which access methods are
executed */
dependency_t activate_recovery_unit(method_t method,
ident_ru_t client_ru);
void deactivate_recovery_unit(method_t method);

public:

/* protocol interface */
void build_graph (ident_ru_t ru,ident_ru_t initiator, ident_ru_t father);
void ack_build_graph (ident_ru_t ru, boolean_t son);
void precommit ();
void ack_precommit (ident_ru_t ru);
void commit ();
void abort (ident_ru_t ru,ident_ru_t initiator, ident_ru_t father);
void ack_abort (ident_ru_t ru,boolean_t son);
void restart ();
}
```

Figure 14 : Definition of the *Generic_reliable_server* class

The *recovery_unit* class implements the distributed protocols, the local graph of dependant recovery units and also contains information recording the replies received from other recovery units during checkpointing. The persistence and consistency of this information despite failures is achieved by inheriting from the *Stable* class and accessing the graph within a short term STM transaction which is independent of the recovery unit's one.

6 Performance Analysis

To analyze the performance of the FTM system, we ran several applications typical of workstation use: scientific computations and interactive office tools. The scientific programs were represented by two long running computations: a prime number calculator and the *mp3d* program of the SPLASH test suite [Singh *et al.* 91]. The office tools were represented by the well-known text editor, *micro-emacs*.

What we wanted to analyze was the fault tolerance penalty, i.e., the overhead paid for fault tolerance in a non-stop execution of programs. It should be noted here that the penalty is not measured in the same manner for the two kinds of applications. In scientific computations, we are interested in the execution overhead in time, while for office tools the penalty is the checkpoint duration, i.e., the time which the user cannot interact with the program.

6.1 Test Environment

The tests described in this section compare the execution of the same programs, on the same machine, under a Mach 3/BSD (MK75) environment and under the FTM system. Porting to the FTM environment did not require any change to the source programs. Nevertheless, to enable proper tests and analysis of the result, we instrumented the test programs by placing a “checkpoint” instruction at specific locations in the code. We measured the fault tolerance overhead in time, as well as several other parameters such as the mean checkpoint size and duration so that all sources of overhead could be defined.

The tests were run on a single pair of stable nodes using our prototype based on a MultibusII single board computer; this is as powerful as a Sun 3/60. The size of the main memory is 20 Mbytes. Computations run only on the primary machine of the pair, the backup machine only being used to implement the STM backup bank.

The software configuration used by applications is made up of the application process itself, a Recoverable Virtual Memory (RVM) server and a reliable screen server (only used with micro-emacs). The RVM server manages three segments for the application process: the data segment, the stack segment and the heap segment. Since a recovery unit is associated with the application’s process, the screen resource and each segment, a checkpoint protocol involves cooperation between 4 recovery units (5 in the micro-emacs example). In our configuration, the checkpoint decision is always taken by the application recovery unit which plays the coordinating role. As this configuration is quite simple, the execution of the checkpointing protocol leads to the exchange of 5 messages between a recovery unit and the coordinator: 2 for the build tree phase, 2 for the precommit phase and 1 for the commit phase. Consequently, the number of Mach messages exchanged during a checkpoint is 15 (20 in the micro-emacs example).

When checkpointing a segment, a minimum of 3 messages is exchanged between the RVM server and the Mach kernel: a flush request from the server to the kernel, 1 data return message for each 256 Kilobytes of contiguous data from the kernel to the server (e.g., Mach maximum size transfer), and an end of flush message from the kernel to the server. Since the size of a memory page is 8 Kilobytes, there is one return message for each set of 32 pages sent.

6.2 Prime Number Calculation

The prime number calculation program is based on the well known Erathostene’s sieve. This program possesses the property of having a small working set since the modified data are the pages containing the prime numbers produced. As an integer is 4 bytes long, one memory page contains 2048 prime numbers. Figure 15 shows the time to calculate 400 000 prime numbers, when varying the numbers of checkpoints taken. We chose to take checkpoints whenever a power of 2 pages was produced. The checkpoint size can thus be precisely calculated: it equals the previous multiple, plus 2 pages for the stack and data segments.

The execution time overhead of FTM varies from 3.6% to 16%, depending on the amount of computation the user accepts to loose in the event of a crash. The minimum overhead is obtained when checkpointing every 266 seconds; this correspond to a 25% loss of work in the event of a crash (4 checkpoints). The 16% worst-case overhead is obtained when checkpointing every 12 seconds; this corresponds to a 1% loss of work.

Figure 15 : Summary of 400 000 prime numbers calculation

# of CKPT	Exec time	Page faults	Exec overhead	Time between CKPT	Data_ return _msg	Mean CKPT duration	CKPT size (pages)
Mach	1028 s						
FTM, 0	1036 s	197	0.78 %				
FTM, 4	1065 s	197	3.6 %	266 s	4	7.25 s	2 + 64
FTM, 7	1070 s	197	4.09 %	153 s	3	4.8 s	2 + 32
FTM, 13	1076 s	197	4.67 %	83 s	3	3 s	2 + 16
FTM, 25	1092 s	197	6.23 %	43.6 s	3	2.24 s	2 + 8
FTM, 49	1125 s	197	9.44 %	25 s	3	1.81 s	2 + 4
FTM, 98	1192 s	197	16 %	12 s	3	1.59 s	2 + 2

6.3 Mp3d

Mp3d is a scientific program coming from the SPLASH test suite. This scientific computation solves a problem in rarefied fluid flow simulation. Details can be found in [Singh *et al.* 91]. We ran *mp3d* with two different numbers of molecules, 10 000 (see figure 16) and 30 000 (see figure 17), for 300 time-steps (see figure 18).

Checkpoints were taken every n time-steps in order to vary the amount of work lost after a crash.

Figure 16 : Summary of *mp3d* execution for 300 time-steps with 10 000 molecules

Number of CKPT	Exec. time	Page faults	Exec. overhead	Time between CKPT	Data_return_msg	Mean CKPT duration	Mean CKPT size
Mach	831 s						
FTM, 0	835 s	103	0.5 %				
FTM, 5	853 s	103	2.6 %	170 s	7.8	3.6 s	819 Kbytes
FTM, 10	864 s	103	4 %	86 s	7	2.9 s	732 Kbytes
FTM, 20	881 s	103	6 %	44 s	6.6	2.3 s	689 Kbytes
FTM, 43	923 s	103	11 %	21 s	6.4	2 s	665 Kbytes
FTM, 100	1028 s	103	23 %	10 s	6.3	1.9 s	654 Kbytes

The minimum overhead measured (for 5 checkpoints) is 2.6% and 2% for 10 000 and 30 000 molecules respectively. The maximum overhead (for 100 checkpoints) is 13.7% and 23% for 10 000 and 30 000 molecules respectively. The gradient of time overhead, dependent on the number of checkpoints, raises when the number of molecules is lower (e.g., see gradients for the 10 000 and the 30 000 molecules execution in figure 18). This is due to the fact that the *mp3d* execution time increases faster than the process' working space. Consequently, the checkpoint duration is proportionally smaller for larger numbers of molecules.

Figure 17 : Summary of *mp3d* execution for 300 time-steps with 30 000 molecules

Number of CKPT	Exec. time	Page faults	Exec. overhead	Time between CKPT	Data_return_msg	Mean CKPT duration	Mean CKPT size
Mach	2406 s						
FTM, 0	2415 s	246	0.4 %				
FTM, 5	2455 s	246	2.0 %	491 s	12	8 s	1964 Kbytes
FTM, 10	2470 s	246	2.6 %	247 s	10	5.5 s	1746 Kbytes
FTM, 20	2513 s	246	3.6 %	125 s	10	4.9 s	1637 Kbytes
FTM, 43	2568 s	246	6.7 %	59 s	10	3.5 s	1578 Kbytes

Figure 17 : Summary of *mp3d* execution for 300 time-steps with 30 000 molecules

Number of CKPT	Exec. time	Page faults	Exec. overhead	Time between CKPT	Data_ return _msg	Mean CKPT duration	Mean CKPT size
FTM, 100	2737 s	246	13.7 %	27 s	9.8	3.2 s	1551 Kbytes

Another observation that can be made is that *mp3d* modifies nearly all of its working space during each time-step. A consequence of this is that the checkpoint size does not decrease proportionally with the number of checkpoints but becomes constant.

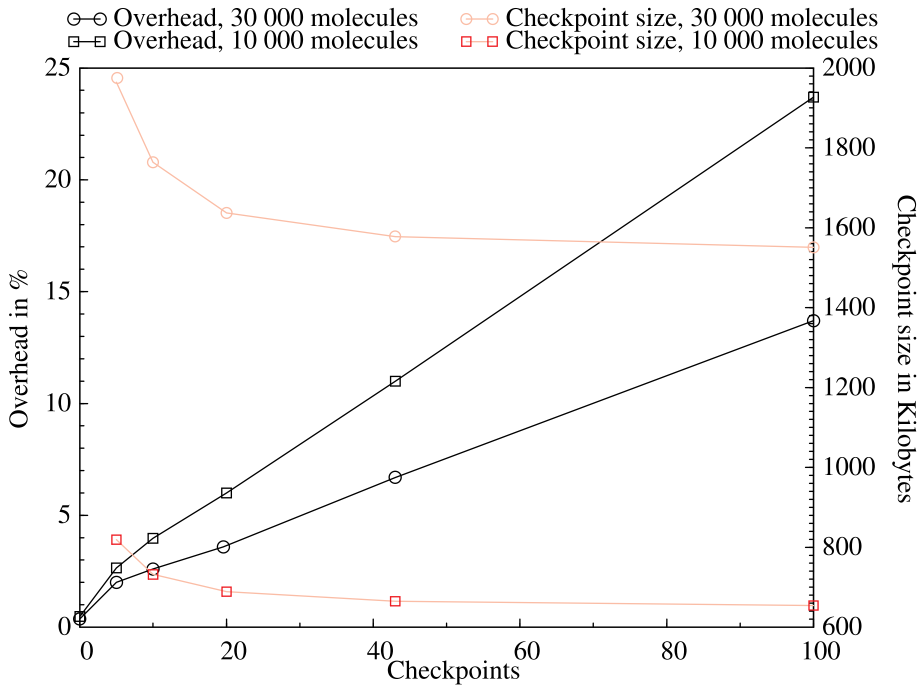

Figure 18 : Execution of *mp3d*

6.4 The *Micro-Emacs* Text Editor

*Micro-emac*s is a well-known editor that runs on many platforms from Unix to PC. We chose to port this editor to the FTM environment since it is representative of many of the interactive tools that can be found on the network. Even if some modern office tools already implement some form of regular checkpointing using disk files, this is not the common case. Moreover, if the operating system does not provide a transactional file system, the checkpoint has to be coded in an ad-hoc fashion [Schmuck & Wyllie 91]. Thus, file consistency and availability are not always ensured if a user's machine crashes. Consequently, we think that the operating system should provide support for masking workstation failures.

The *Micro-emac*s editor is representative of programs that manage not only memory but also I/O. After crash recovery, it is necessary that the user sees on his/her screen, characters consistent with the internal state of the text file. In order to provide this behavior, we designed a reliable screen server that emulates a VT100 console. The VT100 server is simply implemented by storing a copy of the screen in STM. When it is restarted after a crash on the backup machine, the screen is updated with the old version of the screen stored in the backup bank.

To port *Micro-emacs* to the FTM environment, we compiled it with appropriate libraries and initialized the FTM environment to save a checkpoint every 15 seconds. To measure the fault tolerance penalty, the important criterion is the checkpoint duration, since the execution time overhead is not meaningful. During checkpointing, an application process is suspended and thus cannot interact with the user. This is due to the fact that the memory segments have to be consistent with the CPU registers of the application process and hence must not be modified. Our measurements show a checkpoint duration of about 1 second, with a 30% variation in the result due to scheduling. Even if the text is not modified, three pages are modified between checkpoints. Whenever text is entered by the user, with the exception of loading a file, 8 pages are modified leading to a mean increase of 0.1secs. One should note that these durations are still perceptible to the human user on our older machines.

Figure 19 : Summary of *micro-emacs* execution

	Page faults	Time between checkpoints	Data return messages	Checkpoint duration	Mean checkpoint size
No text modification	10	15 s	3	0.98 s < d < 1.3 s	3 pages
With text modifications	10	15 s	5	1 s < d < 1.4 s	8 pages

6.5 Sources of Inefficiency and Possible Improvements

We can observe that running a program under the FTM environment without checkpointing leads to an initial overhead compared to the same program running on the pure Mach/BSD environment. This is due to the use of an external pager (the RVM server). When the program initializes data in a new page, the kernel requests the page from the RVM server which in turn replies with a data unavailable message. This leads to an exchange of 2 messages per new page. Therefore, the initial overhead is proportional to the size of the process's working space, i.e., the number of page faults during execution.

Another observation that can be made for the *prime* and *mp3d* programs is that the checkpoint duration for a small number of checkpoints is slightly longer than the duration for a large number of checkpoints, even if the checkpoint sizes are the same. In fact, the penalty comes from the creation of the first checkpoint and is due to the allocation and initialization of internal RVM structures in STM. Currently, this part of the RVM server has not been optimized and so performance improvement can be expected by tuning the page initialization code.

When few pages are saved during a checkpoint, its duration directly depends on the number of messages exchanged and also on the number of process switches induced. In our implementation, a message exchanged between two recovery units is implemented by a single Mach message. As some Mach servers, such as the RVM server, manage several recovery units for the same application, a possible optimization would be to group together messages of the same wave within a single Mach message. By also applying this optimization to acknowledgments, the number of Mach messages exchanged in a test configuration for segment recovery units would be divided by three. This would give us 5 messages instead of 15 (10 instead of 20, in the micro-emacs example).

7 Comparison with Other Work

Our work builds on numerous previous results which are closely related to techniques for building reliable services and approaches that provide checkpointing environments for scientific computations. This section discusses these related works.

The idea of exploiting a dependency tracking system to reduce the number of computations involved in a checkpoint or a recovery operation appears first in [Koo & Toueg 86] for consistent checkpointing and in [Strom & Yemini 85] for independent checkpointing. This technique is now reused in most of the checkpointing systems. An RPC based dependency tracking system was introduced in [Lin & Ahamad 90], in the framework of a distributed reliable object-based system. The main difference between this study and ours is that, in their work, attributes of operations are calculated from source code: a *read* operation should not modify any data in the object. Therefore, *update* attributes are propagated through nested calls that may transform an upper *read* operation into an *update* one. In our work, attributes have to be specified by the programmer and are only valid at a single call level. Thus, our scheme allows the number of dependent servers to be reduced. Moreover, our dependency tracking system permits the concurrent execution of several applications, something not described in [Lin & Ahamad 90].

In [Leu & Bhargava 89], Leu and Bhargava have shown that consistent checkpointing can be modeled as a distributed transaction (e.g., ARGUS [Liskov *et al.* 87], CAMELOT [Eppinger *et al.* 91], RELAX [Schumann *et al.* 89], ARJUNA [Dixon & Shrivastava 87] [Little 91], QuickSilver [Haskin *et al.* 88] [Schmuck & Wyllie 91]). In transactional systems, the system programmer must still explicitly define the units of work, and hence, the fault-tolerance techniques are not transparent. Moreover, locking resources limits the sharing of objects by several clients. In order to allow for an early release of locks before the transaction commit, the RELAX system manages a dependency graph of transactions sharing uncommitted resources. Also, recent transactional systems, such as QuickSilver [Schmuck & Wyllie 91], separate synchronization from failure atomicity and allow servers to implement their own concurrence control policy.

One of the major issues when designing a reliable operating system is to make programming no more difficult than programming a classical system. This implies the need to be able to mask distributed protocols and the management of reliable data from the programmer. Our solution relies on the use of inheritance in the C++ object-oriented programming language. This technique was first introduced in the projects ARJUNA [Dixon & Shrivastava 87] and AVALON [Eppinger *et al.* 91]. While permitting the redefinition of object creation (e.g., data storage allocation), our (GNU) C++ compiler is not sufficiently flexible to allow the redefinition of method invocation. This implies that the programmer needs to write a call to the *Open* method before accessing a stable object whereas these instructions should be easily integrated into a stable object's method invocation. Some recent variants of C++ [S. Chiba 93] do provide access to method invocation and object creation and thus offer the ability to modify them in the language itself: this property is known as *reflection*. This approach is currently being investigated to design secure and reliable object-based applications using the Fragmentation-Redundancy-Scattering technique [J.C. Fabre 94].

Among the other studies in the checkpointing domain, very few have published real measurements. One major contribution was reported in [Elnozahy *et al.* 92], with the measurement of the checkpointing overhead of the failure-free execution of several parallel/distributed scientific programs. This study provides better results than ours since the mean overhead measured was 1% for taking checkpoints every 2 minutes. The common link between this study and ours is that the MMU is used to determine the modified pages in an application's working set. The major difference is that their algorithm is non-blocking while ours is blocking. This is the first reason for our lower result. A second reason is the fact that our test programs run on a single machine and do pure computations without waiting for messages. Therefore, there is no time wasted which could otherwise be used to flush checkpoints in the background.

The third reason of our lower performance is that they modified the system kernel (V-kernel). In contrast, the FTM implementation is effected completely on top of Mach; no modifications were made to the micro-kernel. Consequently, our implementation induces additional context switches (i.e., between applications and servers) which would not exist in an in-kernel implementation. Nevertheless, this permits an easy port of our system to any existing Mach platform. Finally, it should be noted that the FTM model is not restricted to the management of memory; it has been designed to be general, by allowing the support of reliable servers able to handle I/O, such as our VT100 screen server.

8 Lessons and Conclusion

Our goal for the FTM was to provide solutions that permit the construction of "low-cost" reliable operating systems with minimal assumptions about the underlying machine and kernel. These objectives led us to choose the micro-kernel technology and a blocking consistent checkpoint approach for the FTM reliable client-server model. We now draw lessons from our experience.

8.1 Assessments of the Architecture

At the architectural level, we wanted to develop a convenient memory device which permitted the design of reliable operating systems. This led us to introduce a new stable storage design, called Stable Transactional Memory, built from banks of RAM memory with a built-in transaction facility. We first built a hardware implementation of the STM which provided fine-grained object protection and high performance transactions. Despite its good performance, this hardware design suffered from portability restrictions and this was inconsistent with our goal of using standard of-the-shelf machines.

We consequently introduced a software-based STM relying on a high speed serial link. As such a link can be implemented in several manners, the software based STM is machine independent and is thus able to survive workstation evolution. As a consequence, the FTM code is highly portable and can be run on any mono-processor or multiprocessor Mach platform since the only machine dependent code is related to the serial link implementation. Moreover, several configurations can be chosen for the two machines making up a pair of stable nodes, according to the kinds of faults that must be tolerated:

- two machines in an office connected to the same network. This simple configuration only permits machine failures to be tolerated,
- two machines in separate offices with different sources of external power, possibly connected to separate network segments. This configuration handles faults such as air-condition and power supply breakdown as well as network partitioning, as a result of which other nodes become unreachable,

- two machines in separate buildings. This configuration handles catastrophic disasters such as fire.

Finally, since the two machines of a stable node can be active at the same time, the hardware cost of fault tolerance boils down to the cost of duplicating the main memory (triple or quadruple for modified objects according to the software configuration) and adding the serial link.

8.2 Assessments of the Operating System

Our main objective for the FTM operating system was to offer fault tolerance transparency to the end user application programmer, so that he could design or port an existing application without writing source code related to failure handling. Furthermore, we wanted our approach to be general purpose so that it could be applied to all system resources and services. This has been achieved with the design of the reliable client-server model. An application checkpoint is built from the persistent states of all the servers called by the application since the preceding checkpoint. For instance, the memory part of the application checkpoint is managed by a recoverable memory segment server. It should be noted that the micro-kernel pager paradigm has permitted this recoverable server to be implemented in an efficient way by only saving the pages modified since the last checkpoint.

19.0.1. Analysis of the Reliable Client-Server Model

The advantages and drawbacks of the reliable client-server model result directly from our design choices. The first advantage is that the model preserves the benefits of micro-kernel based operating systems: modularity and compatibility between the communication model and distribution. Modularity means that new system functionality can be added to an existing operating system without having to modify the system. This property is ensured by the checkpointing algorithm and the reliable server design methodology, since a consistent state is dynamically computed when checkpointing and consequently no static software configuration has to be specified. Compatibility with distribution is also preserved since a reliable server may be on a different machine to the client. The second advantage of the reliable client-server model is that it does not introduce RPC performance degradation: the number of messages involved in a client-server call in FTM is the same as for a traditional micro-kernel based system. The only difference is that the length of messages is increased by the addition of the recovery unit's UID. Consequently, the cost of fault tolerance is only noticeable when a checkpoint is being taken.

The drawback of our approach is that checkpointing needs to be implemented in three phases. This is the price paid for preserving modularity and allowing any server or client to initiate a checkpoint. Our current protocol takes care of complex situations, such as chasing and multiple graph merging, which may seldom appear in the first phase when building the consistent global checkpoint. Nevertheless, the general case is simpler and can be described as a single application communicating with a recoverable virtual memory and screen server. Moreover, most of the time the checkpoint decision is taken by the application recovery unit and this reduces the duration of the *build_atomic_action* phase since the application stops before the servers do. Consequently, many optimizations can be studied to improve performance: topology optimization and depth reduction of the recovery unit commit tree, and execution of the checkpoint protocol in two phases when all application processes have been stopped before servers.

19.0.2. Blocking Consistent Checkpointing: Did we Make the Right Choice?

When designing the FTM model, our objective was to minimize the hardware cost of fault tolerance. This led us to choose a checkpoint-based system model; this means that computations performed since previous checkpoints are lost when a crash occurs. Thus, not all types of applications can be run on the FTM system. For instance, applications managing I/Os whose results should not be destroyed by a crash need to save a checkpoint at each I/O. The extent to which the FTM model can be used depends on how often checkpoints can be saved or, more precisely, on how long it takes to save a checkpoint. We now analyze the limits of our current prototype and estimate what those limits would be when the prototype is run on modern machines.

One might expect that the major checkpoint penalty is due to the checkpoint protocol. However, our experiment shows that this is generally not true. For instance, in our scientific computation oriented tests, the time overhead is mainly due to the saving of modified memory pages during checkpointing. This time is directly proportional to the serial link speed and to the machine's memory bandwidth. These parameters are independent of the consistent checkpointing choice. The protocol overhead becomes the main source of overhead when very few pages are flushed in a checkpoint. Such a situation arises for office tools, such as our *micro-emacs* example, for which checkpoint duration is under 1.4sec.

From the user's perspective, the important overhead is the time during which he is unable to interact with his application. It should be noted, that a 1.4sec duration is still perceptible to the human user. Two comments can be expressed about this result. First, our FTM prototype has not yet been optimized; several techniques, such as those described in sections 6.5 and 8.2.1, can be used to improve performance. Second, our prototype machine is a slow machine, equivalent to an out-dated Sun3.

Extrapolating existing results to an up to date machine is always difficult. In the case of FTM, performance depends on several parameters: CPU speed, memory and serial link bandwidth. On a modern off-the-shelf PC (e.g., 486/66 Mhz-PCI), the Mach micro-kernel runs 4 to 7 times faster than on our prototype. Current ATM boards offer a physical bandwidth of 155Mbits; at user level, a 50-70 Mbits bandwidth is reachable using standard TCP/IP. As we said above, the checkpoint protocol overhead becomes important when small amounts of data is sent over the serial link, so performance then mainly depends on the micro-kernel speed (e.g., messages exchange and scheduling). Consequently, with current off-the-self hardware, we hope to gain a factor of 4 for office tools' checkpoint durations. This would lead to a minimum checkpoint duration of about 0.3sec. In our *micro-emacs* example, that would make checkpointing nearly transparent to the human user.

Acknowledgements

We wish to thank C. Bryce, V. Issarny, I. Puaut and P.A. Lee from Newcastle University for their pertinent comments and patient reviews of early versions of this document. J.P. Routeau helped us to implement FTM on top of Mach and to improve system performance; we are grateful to him for this.

References

[Accetta *et al.* 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, & M. Young. Mach: A new kernel foundation for Unix development. In *Proc. of Usenix 1986 Summer Conference*, pages 93–112, July 1986.

[Ahamad & Lin 89] M. Ahamad & L. Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proc. of 8th Symposium on Reliable Distributed Systems*, pages 2–11, Seattle (WA), October 1989.

[Banâtre *et al.* 86] J.P. Banâtre, M. Banâtre, G. Lapalme, & Fl. Ployette. The design and building of enchere, a distributed electronic marketing system. *Communications of the ACM*, 29(1):19–29, January 1986.

[Banâtre *et al.* 88] J.P. Banâtre, M. Banâtre, & G. Muller. Ensuring data security and integrity with a fast stable storage. In *Proc. of 4th International Conference on Data Engineering*, pages 285–293, Los Angeles, February 1988.

[Banâtre *et al.* 91a] M. Banâtre, P. Heng, G. Muller, & B. Rochat. How to design reliable servers using fault tolerant micro-kernel mechanisms. In *USENIX Mach Symposium*, pages 223–231, Monterey, California, November 1991.

[Banâtre *et al.* 91b] M. Banâtre, G. Muller, B. Rochat, & P. Sanchez. Design decisions for the FTM: A general purpose fault tolerant machine. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems*, pages 71–78, Montréal (Canada), June 1991.

[Banâtre *et al.* 93] M. Banâtre, P. Heng, , G. Muller, N. Peyrouze, & B. Rochat. An experience in the design of a reliable object based system. In *Proc. of the 2th Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[Bhargava & Lian 88] B. Bhargava & S.R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach. In *Proc. of 7th Symposium on Reliable Distributed Systems*, pages 3–12, Columbus (OH), October 1988.

[Borg *et al.* 89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, & W. Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[Chandy & Lamport 85] K.M. Chandy & L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Cristian & Jahanian 91] F. Cristian & F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proc. of*

10th Symposium on Reliable Distributed Systems, pages 12–20, Pisa (Italy), September 1991.

[Dixon & Shrivastava 87] G.N. Dixon & S.K. Shrivastava. Exploiting type inheritance facilities to implement recoverability in object based systems. In *Proc. of the 6th Symposium on Reliability in Distributed Software and Database Systems*, pages 107–114, Williamsburg, March 1987.

[Elnozahy & Zwaenepoel 92] E.N. Elnozahy & W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[Elnozahy *et al.* 92] E.N. Elnozahy, D.B. Johnson, & W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. of 11th Symposium on Reliable Distributed Systems*, pages 39–47, Houston (TX), October 1992.

[Eppinger *et al.* 91] J.L. Eppinger, L.B. Mummert, & A.Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, San Mateo, 1991.

[Gazelle 90] Gazelle. *Hot Rod High Speed Serial Link Data Sheet*. Gazelle Microcircuits, Inc, Santa Clara (CA), 1990.

[Gleeson 93] B.J. Gleeson. Fault tolerance: Why should I pay for it. In M. Banâtre & P.A. Lee, editors, *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, volume 774 of *LNCS*, pages 66–77, Le Mont Saint-Michel (France), June 1993.

[Goldberg *et al.* 90] A. Goldberg, A. Gopal, K. Li, R. Strom, & D.F. Bacon. Transparent recovery of mach applications. In *USENIX Mach Workshop*, pages 169–183, Burlington (VT), October 1990.

[Gray 78] J. Gray. *Notes on Database Operating Systems.*, volume 60 of *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[Hardell *et al.* 90] W.R. Hardell, D.A. Hicks, L.C. Howell, W.E. Maule, R. Montoye, & D.P. Tuttle. Data cache and storage control units. In *IBM RISC System/6000 Technology*, pages 44–51. IBM, 1990.

[Haskin *et al.* 88] Roger Haskin, Yoni Malachi, Wayne Sawdon, & Gregory Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[J.C. Fabre 94] B.Randell J.C. Fabre, Y. Deswarte. Designing secure and reliable applications using fragmentation-redundancy-scattering: an object-oriented approachcattte. In K. Echtle, D. Hammer, & D. Powell, editors, *Dependable Computing - EDCC1*, volume 852 of *LNCS*, pages 21–38, Berlin (Germany), October 1994. Springer Verlag.

[Juang & Venkatesan 91] T.TY Juang & S. Venkatesan. Crash recovery with little overhead. In *Proc. of 13th International Conference on Distributed Computing Systems*, pages 454–461, Arlington (TX), May 1991.

[Koo & Toueg 86] R. Koo & S. Toueg. Checkpointing and rollback recovery for distributed systems. In *Proc. of Fall Joint Computer Conference*, pages 1150–1158, Dallas, 1986.

[Lampson 81] B. Lampson. Atomic transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.

[Lee & Anderson 90] P.A. Lee & T. Anderson. Dependable computing and fault-tolerant systems, vol. 3. In J.C.Laprie A. Avizienis, H. Kopetz, editor, *Fault Tolerance : Principles and Practice*. Springer Verlag, New York, 1990.

[Leu & Bhargava 88] P. Leu & B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proc. of 4th International Conference on Data Engineering*, pages 154–163, Los Angeles (CA), February 1988.

[Leu & Bhargava 89] P. Leu & B. Bhargava. A model for concurrent checkpointing and recovery using transactions. In *Proc. of 9th International Conference on Distributed Computing Systems*, pages 423–430, 1989.

[Li *et al.* 91] K. Li, J.F. Naughton, & J.S. Plank. Checkpointing multicomputer applications. In *Proc. of 10th Symposium on Reliable Distributed Systems*, pages 1–10, Pisa (Italy), September 1991.

[Lin & Ahamad 90] L. Lin & M. Ahamad. Checkpointing and rollback-recovery in distributed object based systems. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, pages 97–104, Newcastle Upon Tyne (UK), June 1990.

[Liskov *et al.* 87] B. Liskov, D. Curtis, P. Johnson, & R. Scheifler. Implementation of Argus. In *Proc. of 11th ACM Symposium on Operating Systems Principles*, pages 111–122, 1987.

[Little 91] M.C. Little. *Object Replication in a Distributed System*. Computing laboratory, University of Newcastle upon Tyne, September 1991.

[Loepere 93] K. Loepere. *OSF MACH 3 Kernel Final Draft Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, May 1993.

[Merlin & Randell 78] P.M. Merlin & B. Randell. State restoration in distributed systems. In *Proc. of 8th International Symposium on Fault-Tolerant Computing Systems*, pages 129–134, Toulouse, June 1978.

[Mullender *et al.* 90] S.J. Mullender, G.Van Rossum, A.S. Tanenbaum, R.Van Renesse, & H.Van Staveren. A distributed operating system for the 1990s. *IEEE Computer*, pages 44–53, May 1990.

[Muller & Prunault 93] G. Muller & Y. Prunault. Conception et réalisation d'un lien série haut débit. Technical report, *IRISA*, 1993.

[Muller *et al.* 91] G. Muller, B. Rochat, & P. Sanchez. A stable transactional memory for building robust object oriented programs. In *EuroMicro 91*, pages 359–364, Vienna (Austria), September 1991.

[Muller *et al.* 94] G. Muller, M. Hue, & N. Peyrouze. Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment. In K. Echtle, D. Hammer, & D. Powell, editors, *Dependable Computing - EDCC1*, volume 852 of *LNCS*, pages 491–508, Berlin (Germany), October 1994. Springer Verlag.

[Nelson 81] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1981.

[Raynal & Helary 90] M. Raynal & J.M. Helary. *Synchronization and Control of Distributed Systems and Programs*. Wiley series in parallel computing, 1990.

[Rochat 92] B. Rochat. *Une approche à la construction de services fiables dans les systèmes distribués*. Phd. thesis, université de Rennes I, February 1992.

[Rozier *et al.* 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, & W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[S. Chiba 93] T.Masuda S. Chiba. Designing an extensible distributed language with meta-level architecture. In *Proceedings of the ECOOP '93*, volume 707 of *LNCS*, pages 483–502. Springer Verlag, July 1993.

[Schmuck & Wyllie 91] F. Schmuck & J. Wyllie. Experience with transactions in quicksilver. In ACM, editor, *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 239–253, October 1991.

[Schumann *et al.* 89] R. Schumann, R. Kroger, M. Mock, & E. Nett. Recovery management in the RelaX distributed transaction layer. In *Proc. of 8th Symposium on Reliable Distributed Systems*, pages 21–28, Seattle, October 1989.

[Silva & Silva 92] L.M. Silva & J.G. Silva. Global checkpointing for distributed programs. In *Proc. of 11th Symposium on Reliable Distributed Systems*, pages 155–162, Houston (TX), October 1992.

[Singh *et al.* 91] J.P. Singh, W.D. Weber, & A. Gupta. Splash : Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, *Computer Systems Laboratory, Stanford University*, April 1991.

[Strom & Yemini 85] R.E. Strom & S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Tamir & Sequin 84] Y. Tamir & C. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. of 1984 International Conference on Parallel Processing*, pages 32–41, August 1984.

[Taylor *et al.* 80] D.J. Taylor, D.E. Morgan, & J.P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, SE-6(6):585–594, November 1980.

[Wood 81] W.G. Wood. A decentralised recovery control protocol. In *Proc. of 11th International Symposium on Fault-Tolerant Computing Systems*, pages 159–164, Portland (OR), June 1981.