INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE
N° 975

# A SURVEY OF RECOVERABLE DISTRIBUTED SHARED MEMORY SYSTEMS

CHRISTINE MORIN, ISABELLE PUAUT

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

# A Survey of Recoverable Distributed Shared Memory Systems

Christine Morin*, Isabelle Puaut*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Solidor

Publication interne n° 975 — Décembre 1995 — 30 pages

**Abstract:** Distributed Shared Memory (DSM) systems provide a shared memory abstraction on distributed memory architectures (distributed memory multicomputers, networks of workstations). Such systems ease parallel application programming since the shared memory programming model is often more natural than the message-passing paradigm. However, the probability of failure of a DSM system increases with the number of sites. Thus, fault tolerance mechanisms must be implemented in order to allow processes to continue their execution in the event of a failure. This paper gives an overview of *recoverable* DSM systems (RDSM) that provide a checkpointing mechanism to restart parallel computations, after a site failure.

**Key-words:** Distributed Systems, Distributed shared memory, Availability, Backward error recovery, Consistent global states

*(Résumé : tsvp)*

*{cmorin,puaut}@irisa.fr

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

# Une synthèse des systèmes à mémoire virtuelle partagée recouvrables

**Résumé :** Les systèmes à mémoire virtuelle partagée offrent à leurs utilisateurs l'illusion d'une mémoire partagée sur les architectures à mémoire distribuée (réseaux de stations de travail, machines parallèles à mémoire distribuée). De tels systèmes facilitent la programmation des applications parallèles, car le modèle de programmation par partage de mémoire est souvent plus naturel que le modèle de programmation par échange de messages. Toutefois, plus le nombre de composants dans un système à mémoire virtuelle partagée augmente, plus la probabilité qu'une défaillance se produise est importante. Des mécanismes de tolérance aux fautes doivent par conséquent être ajoutés aux systèmes à mémoire virtuelle partagée. Ce rapport effectue un tour d'horizon des mécanismes de sauvegarde et restauration de points de reprise dans les systèmes à mémoire virtuelle partagée (mémoires virtuelles partagées *recouvrables*). Ces mécanismes permettent de poursuivre l'exécution d'une application parallèle en dépit de la défaillance d'un site.

**Mots-clé :** Systèmes distribués, Systèmes à mémoire virtuelle partagée, Disponibilité, Retour arrière, États globaux cohérents

# 1 Introduction

Distributed Shared Memory (DSM) systems [NL91] provide the abstraction of a memory space physically shared among processes on loosely coupled architectures such as distributed memory multicomputers or networks of workstations. A DSM is attractive from the programmer's point of view, since it simplifies data partitioning and load balancing, which both are important issues in parallel programming. Nevertheless, as the number of components in these systems grows, the probability that a failure occurs also increases. This is unacceptable, especially when long-running applications are to be executed. As restarting a long-running application from its initial state must be avoided, it is desirable for a DSM to be *recoverable*, and to allow an application to restart its execution from a state saved beforehand (a *checkpoint*), when a failure occurs.

Backward error recovery is a well-known fault tolerance technique [LA90]. It avoids having to restart an application from its initial state in case of failure, by using a previous image of the system to restart the execution. Compared to techniques based on active software replication or hardware redundancy, backward error recovery is the most attractive method to tolerate site failures in a DSM system. Techniques based on hardware redundancy [BGH87, HS87] are too expensive in the context of DSM systems. Active software replication [Bir85] requires a strong synchronization between replica, that would lead in DSM systems to a significant increase of inter-node communications and thus to a high degradation of performance.

In this paper, we review recoverable DSM systems that provide a checkpointing mechanism to restart parallel computations after a failure. The remainder of this paper is organized as follows. General principles of DSM systems are presented in Section 2. Section 3 introduces the main checkpointing strategies that are used in message-passing systems. Section 4 gives an overview of existing propositions aiming at integrating checkpointing in DSM systems. This section, which is the central part of the paper, is not intended to give an exhaustive list of existing recoverable DSM propositions. Rather, it focuses on differences between these propositions and techniques designed for message passing systems. The characteristics of existing Recoverable DSM (RDSM) are summarized in Section 5. Finally, concluding remarks are given in Section 6.

# 2 Distributed Shared Memory

In this section, we give a brief overview of DSM important design issues, and introduce the terminology used in the rest of the paper.

## 2.1 General Principles

Various classes of DSM have been designed depending on the type of shared data (paged virtual address space, variables of a parallel program, typed entities such as objects). These three DSM categories are briefly presented in the following paragraphs.

### Page-based Distributed Shared Memory

The first DSM system proposal, described in [Li86, LH89], allows processes to share a unique paged virtual address space on top of multicomputers. The shared address space is divided in a set of fixed-size blocks called *pages*. The virtual to physical address translation uses standard address translation hardware (Memory Management Unit, or MMU). In the most trivial implementation of this DSM, a unique copy of each page is maintained in the volatile memory of one machine. An access to a page which is not present in the local memory is detected by the address translation hardware which triggers an exception (*page fault*). The page fault treatment consists in transferring the page to the machine requesting it.

Page-based DSM systems give the illusion to the programmer of a shared memory multiprocessor. However, using the address translation hardware requires transfers at a page granularity, which may result in poor performance when the shared data size is less than a virtual memory page (*false-sharing* problem).

### Shared-variable Distributed Shared Memory

Another approach is to share only variables or data structures accessed by several processes, thus avoiding page-based systems' drawbacks. Such schemes are implemented entirely in software and provide a set of shared variables instead of an untyped memory, thus offering the programmer a higher level of abstraction. Additional information associated with shared data (for instance, its type) improves the efficiency of this DSM category. Munin [CBZ91] and Midway [BZ91] are examples of such DSMs which both provide a programming environment in which it is possible to declare shared variables and to synchronize the accesses to these variables.

### Object-based Distributed Shared Memory

A further step in structuring shared data consists in sharing encapsulated data or *objects*. Objects differ from shared data in that each object not only contains data but also methods which provide the only way to access this data (no direct access to object data is permitted). Restricting access to object internal data allows to increase the performance of DSM implementations. Linda [CG86] and Orca [BKT92] are examples of object-based DSM systems.

## 2.2 Consistency of Shared Data

A trivial DSM implementation consists in storing a copy of each piece of shared data (page, variable or object depending on the target DSM system) on a single machine. However, such an implementation is inefficient, since for instance simultaneous reads are not allowed on a shared piece of data. Hence, in most DSM systems, shared data is replicated in the volatile memory of multiple machines. This raises the problem of maintaining consistency of the multiple copies. A *consistency model* defines a contract between the application programmer and the DSM. It imposes programming rules and specifies the DSM behavior if these rules are respected. The most widely used consistency models are presented in the following paragraphs.

### Sequential Consistency Model

The sequential consistency model [Lam79] ensures that all accesses to shared data are seen in the same order by every process, wherever it is located. It imposes no constraint on the programmer who uses the DSM as a shared memory multiprocessor.

The sequential consistency model is essentially used in page-based DSM systems. Its most widespread implementation, described in [Li86], relies on a write-invalidate protocol. A page may be replicated only if it is protected against writes. When a process attempts to write on a protected page this privilege violation is detected by the memory management hardware and all the copies of the page are invalidated before permitting the write access. The sequential consistency model thus leads to less efficient implementations than weaker consistency models.

### Relaxed Consistency Models

Several consistency models have been proposed to optimize the implementation of shared-variable DSMs. Most of them distinguish accesses to *ordinary* shared variables from accesses to *synchronization* variables, used to protect the accesses to ordinary variables. *Weak Consistency* [DSB86] ensures that (i) accesses to synchronization variables are sequentially consistent, (ii) no access to a synchronization variable is issued in a processor before all previous writes have completed everywhere, and (iii) no memory access (read or write) can be performed until all previous accesses to synchronization variables have been performed. The weak consistency model allows to build implementations that are more efficient than in sequentially consistent systems, since modifications of shared variables need not be observed immediately. The propagation of modifications to shared variables may be delayed until a synchronization variable is accessed. However, the programming model is not identical to the one used on a shared memory multiprocessor, as consistency is not ensured for every memory access but only at synchronization points.

*Release Consistency* [GLL+90] and *Entry Consistency* [BZ91] are two refinements of weak consistency. The former distinguishes *acquire* and *release* accesses to synchronization variables; modifications to a shared variable are observed at the latest when an *acquire* access to a synchronization variable is performed. The latter requires a synchronization variable to be associated with each ordinary variable. Hence, only modifications to the variables associated with a particular synchronization variable need to be propagated when this synchronization variable is accessed.

### 2.3 Locating Shared Data

In page-based and shared-variable DSMs, shared data is replicated in the volatile memory of several machines to reduce the latency of data accesses. A *local directory* maintains information on shared data (page or variable) present locally. To locate all the replicas of a shared piece of data, two approaches may be used: one based on *broadcasting* and the other using a *global directory*.

#### Broadcast-based Schemes

In broadcast-based systems, no information is maintained about the current location of shared data replicas. A message is broadcast to all machines when copies of shared data must be located (for example, on a page fault in a page-based DSM).

#### Directory-based Schemes

In directory-based systems, a global directory maintains information about the current locations of shared data. Three directory organizations have been proposed in [Li86] for a page-based DSM. For each of these schemes, a machine, called *manager*, is associated with each piece of shared data (here a page) and maintains a list of machines having a copy of the page. In the *central manager* scheme, the global directory is maintained by a single machine (all pages have the same manager). In the *statically distributed manager* scheme, the global directory is distributed on a set of machines. The association between a page and its manager is fixed. In the *dynamically distributed manager* scheme, a manager machine is dynamically associated with a page. The global directory entry related to a given page is stored on a single site, called the *page owner* site that may change during the execution (in Li's proposition, based on a write-invalidate protocol, the owner site of a page is the unique site having a writable copy or one of the sites having a read-only copy). An access chain, that may span multiple machines, is used to locate the page's owner.

# 3 Checkpointing in Message-Passing Systems

Backward error recovery [LA90] is a well-known technique that allows processes to continue their execution despite failures. It consists of regularly saving the state of processes on a crash-proof storage support and rolling back processes to this state in the event of a failure. The two following paragraphs give the state of the art of checkpointing techniques, first considering a single process (paragraph 3.1) and then considering a distributed system in which processes communicate exclusively by exchanging messages (paragraph 3.2).

## 3.1 Checkpointing a Single Process

Backward error recovery is a widespread fault tolerance technique because it is almost independent of the failure type, and well-suited to both *transient*[1] and *permanent* failures. Backward error recovery consists in periodically saving the state of each process, which is called a *checkpoint*, on a support called *stable storage* [Lam81] that is not affected by failures. Stable storage ensures persistence and atomic update of data. The persistence property ensures that a piece of data remains accessible despite the occurrence of failures and is not altered by a failure. The atomicity property guarantees that data updates are either successful, or leave data in its initial state. This second property is generally ensured by a two-phase commit protocol [Gra78]. A checkpoint contains all the information that is required to restart a process (application context plus operating system context). When a failure occurs, a process is restarted using its checkpoint stored on stable storage.

The simplest checkpointing algorithm is synchronous: a process is suspended until its checkpoint is saved on stable storage. The time during which the process execution is suspended may be long, especially with disk-based implementations of stable storage. In order to reduce this delay, *asynchronous* checkpointing algorithms (see [LNP90] as an example) have been proposed. They allow processes to continue their execution while saving their checkpoint. In addition, the simplest

[1] Faults which disappear when processes are restarted.

checkpointing algorithms store the whole process state in stable storage, which can result in large checkpointing overhead when the process data size is large. In order to limit the size of checkpoints in stable storage, *incremental* algorithms have been introduced. They only store on stable storage the data that have been modified since the last checkpoint. Both asynchronous and incremental checkpointing schemes usually rely on the use of address translation hardware to detect modified data [AL91].

## 3.2 Checkpointing in a Distributed System

Let us now consider a message-passing distributed system. The set of communicating process checkpoints must form a *consistent* global system state [CL85]. A global system state is said to be consistent if all messages received by a process at a recovery point were sent at the sender's recovery point. This property is illustrated in the time diagram depicted in Fig. 1 for a system consisting of three communicating processes $P_1$, $P_2$ and $P_3$. Message transmissions are represented by arrows and the processes recovery points by circles.

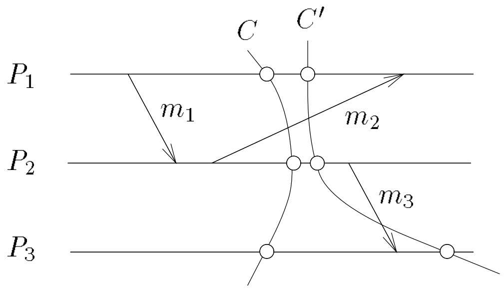


Figure 1: Example of consistent ($C$) and non consistent ($C'$) global states

Only the process states belonging to a consistent global state can be used to restart processes after a failure. For instance, if the three processes of Fig. 1 were restarted from their states belonging to $C'$ global state, process $P_3$ would receive twice the same message ($m_3$).

Two classes of techniques, based on *consistent* or on *independent* process checkpointing, can be used to ensure that processes are restarted from a consistent global system state after a failure.

### Consistent Checkpointing

In consistent checkpointing techniques, processes coordinate the saving of their checkpoints in stable storage to ensure that the set of checkpoints is consistent. The idea of such consistent schemes is to form a barrier beyond which rollback is not required. Consequently, only one checkpoint per process needs to be saved.

Simple implementations of consistent checkpointing synchronize all processes in the system. The optimizations proposed in [KYA86, KT87] consist in synchronizing only processes that have communicated since the last checkpoint. Their implementation requires that information about inter-process communications is logged.

### Independent Checkpointing

In independent checkpointing techniques, each process saves periodically its state on stable storage without any synchronization with other processes. Thus, the set of processes checkpoints does not necessarily form a consistent global state. When a site fails, processes are restarted from one of their previous checkpoints, which is selected so that the set of selected checkpoints is consistent. The consistent global state is computed with an algorithm taking into account inter-process communications. As a consistent state is computed only at recovery time, multiple checkpoints per process must be maintained in stable storage.

In contrast with consistent checkpointing techniques, processes may not be restarted from their last checkpoint. In the worst case, they may be restarted from their initial state when no other consistent global state can be computed. This is called the *domino effect* [Ran75], illustrated in Fig. 2 for a system with two processes. None of the global states represented in the figure by $(X_i, Y_j)$ tuples is consistent except the initial state $(X_1, Y_1)$.

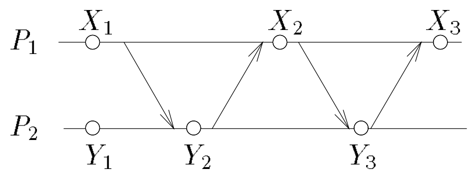


Figure 2: Domino effect

The domino effect can be avoided by forcing a process to save a checkpoint each time it communicates with another one. This trivial solution is realistic only in systems where processes do not communicate frequently. An alternative solution, also aiming at decreasing the amount of data stored on stable storage, consists in logging and replaying messages after a failure. Each process has a single checkpoint saved on stable storage. If the global state formed by the set of process checkpoints is not consistent, it is transformed into a consistent one at rollback time. This is done by sending again the messages that had been sent before the failure. For example, if process $P_1$ in Fig. 2 fails after the saving of state $X_3$, the two processes $P_1$ and $P_2$ will be restarted from their last checkpoint ($X_3$ et $Y_3$). $P_2$ will then send again the message destinated to $P_1$. This technique, adopted in [SY85], requires to log in stable storage messages exchanged between processes. A deterministic process behavior is required to guarantee that processes will exchange the same messages after a rollback that they exchanged before the failure. Any source of non-determinism in a process execution forces the process to save a checkpoint.

# 4 Recoverable Distributed Shared Virtual Memory Systems

This section reviews existing checkpointing propositions in the context of DSM systems. We propose a taxonomy of recoverable DSM systems (RDSMs) based on three orthogonal characteristics: (i) the process checkpointing technique (paragraph 4.2), (ii) the management of DSM data structures, such as directories (paragraph 4.3) and (iii) the storage support used for saving checkpoints (paragraph 4.4).

This classification shows the trade-offs for building a recoverable DSM. Only a few propositions have been evaluated from a performance point of view. Hence, it is very difficult to provide a quantitative comparison of existing RDSMs. Consequently, only a qualitative comparison of existing recoverable DSM is given along the paper. A description of the most widespread system and fault models precedes the presentation of existing RDSM proposals.

## 4.1 System Model

A system made up of a set of machines (or sites) interconnected through a communication network is considered. This model applies to a network of workstations as well as a distributed memory multicomputer. Each site consists of a processor and a volatile primary memory. A permanent storage facility (disk) is shared by all machines in the system and can be directly used to implement stable storage.

The fault model assumed in most recoverable DSM we reviewed is the following. Machines are *fail-stop* [Sch87]: they either work according to their specification, or stop working (i.e., *crash*) without corrupting data. When a machine crashes, the contents of its primary memory is lost. The network and the permanent storage are assumed to be reliable. This is obtained either directly or by the means of fault tolerance mechanisms, like message retransmission or disk mirroring. Moreover, the network is assumed never to partition, so that every machine is always able to communicate with every other site but the faulty ones. No assumption is made on the message delivery order nor on the number of simultaneous site failures. When a different failure model is considered in a particular system, it is explicitly mentioned.

## 4.2 Checkpointing Technique

The application performance when using checkpointing (e.g., the time during which a process is suspended when it saves a checkpoint, the recovery time after a failure) highly depends on the basic process checkpointing technique. The following paragraphs detail solutions based respectively on consistent and independent checkpointing (paragraphs 4.2.1 and 4.2.2).

### 4.2.1 Consistent Checkpointing

Several proposals [CCD+93, JT94, KCG+95, CMP95] use the *global consistent checkpointing* approach. It consists in synchronizing all the processes when a checkpoint is saved. In the context of parallel applications with huge data requirement executing on top of a recoverable DSM, the main advantage of this scheme is that only one checkpoint per process has to be saved on stable storage. An additional benefit is that the checkpointing overhead, due to message exchanges and disk writes, is limited to checkpoint saving operations. Between two checkpoints, there is no perturbation of the normal execution. In particular, no action is required when processors exchange information, in contrast to independent checkpointing (see paragraph 4.2.2). The application programmer can then tune the time overhead of the fault-tolerance mechanisms by choosing an appropriate checkpoint frequency.

As in message-passing systems, the main drawbacks of global consistent checkpointing are due to (i) the cost of process synchronization when saving a set of checkpoints; (ii) the fact that *all* processes are synchronized when storing their state in stable storage. Works aiming at reducing these drawbacks are sketched below.

#### Decreasing Process Synchronization Latency

One of the major drawback of global consistent checkpointing is the cost of global process synchronization when saving a checkpoint. While this cost may be found acceptable in massively parallel machines with high-speed interconnection networks, it may not be the case with slower interconnection networks. For this reason, the recoverable DSM proposed in [CMP95] takes advantage of the behavior of many parallel applications in which processes regularly synchronize through barriers, by unifying checkpointing and synchronization mechanisms. In the referenced paper, the processes checkpoints are always saved within synchronization barriers. Hence, the time spent by processes in synchronization is reduced, as barriers are used both for application synchronization and for checkpointing. Moreover, this approach guarantees that no message is in transit during the saving of a checkpoint thus simplifying the checkpointing protocol implementation.

### Decreasing the Number of Processes to Synchronize

Another drawback of global consistent checkpointing is that *all* machines are involved in the saving and restoration of checkpoints. Indeed, to ensure a consistent global system state, only the processes that have communicated with each others since the previous checkpoint, namely *dependent* processes, have to atomically save their state on stable storage. As in message-passing systems, dependency tracking techniques can be used to involve (and thus to synchronize) only the dependent processes in the saving of their state.

In message-passing systems (see Section 3), a process $p_1$ becomes directly dependent of a process $p_2$ if $p_2$ has sent a message to $p_1$. In a similar way, in a DSM, a process $p_1$ becomes directly dependent of a process $p_2$ if $p_1$ accesses a piece of data that has been modified by $p_2$.

A trivial technique mentioned (but not adopted) in [JF94] is to use the same dependency recording techniques in a DSM as in message-passing systems. Such an approach is correct as messages are used for all inter-machine data transfers. However, it introduces more dependencies than strictly necessary. Indeed, not all messages contain shared data, and thus incur a dependency between the sender and the receiver (examples of messages not carrying dependency are messages requesting the invalidation of a shared piece of data). In order to reduce the number of dependencies, the recoverable DSM described in [JT94] records a dependency only when a message containing data that has been modified since the previous checkpoint is transferred. Even if the number of dependencies is drastically reduced by such a technique, unnecessary dependencies are still recorded. As an example, let us consider a page-based DSM implementing sequential consistency through a write-invalidate protocol and storing page location information following a central manager scheme. If the dependency recording mechanism presented in [JT94] was used, any transfer of a message containing modified data transiting through the manager (for instance, any page write fault) would incur new dependencies linking the old and new page owners, as well as the page manager.

These unnecessary dependencies are eliminated in [JF94] which proposes a dependency tracking technique suited to page-based RDSMs. This technique relies on pages timestamps, called *ownership timestamps*. A dependency between the old and the new owner of a page is recorded only when the owner of the page changes.

### 4.2.2 Independent Checkpointing

In RDSMs based on independent checkpointing, each process saves its state on stable storage without synchronizing with the other processes. As in message-passing systems, the two major drawbacks of this policy are the domino effect and the memory occupation due to the need of maintaining multiple checkpoints per process. Existing optimizations of independent checkpointing aiming at reducing these drawbacks are given hereafter.

#### Optimistic Technique to Limit the Domino Effect

So as to reduce the occurrence of the domino effect without eliminating it totally, the recoverable DSM presented in [JF95] uses an optimistic approach based on the periodic saving of the state of each process. Processes save their checkpoints at fixed physical time intervals, and thus approximatively at the same time. Hence, the probability of obtaining a global consistent state is increased, hopefully limiting the amount of computation lost in the event of failure. However, this solution has only been validated by simulation. An implementation in a real system would be more convincing. In the referenced paper, the computation of a consistent global state at rollback time relies on the recording of inter-process communications. The dependency tracking algorithm presented in [JF94], that was briefly described in the previous paragraph is used.

#### Synchronization of Checkpoints and Communications (Communication-induced Schemes)

To avoid the domino effect as well as to reduce the storage requirements for checkpoints in stable storage, several RDSMs ([WF90] [TH90] [JF93] [SZ90] for example) require every process to save its state each time it communicates with another one (*communication-induced* scheme). In the context of a DSM system, this means that a checkpoint must be saved each time modified data is accessed by another process (e.g., transfer of a modified page in a page-based DSM). All RDSMs using such a technique rely on an incremental checkpointing algorithm. Instead of saving the whole process state in stable storage, only data that has been modified since the last checkpoint is saved at each inter-process communication.

An optimization of the base communication-induced scheme can be obtained on shared-variable DSMs implementing a relaxed consistency model. The RDSM described in [JF93] takes advantage of a relaxed consistency model to reduce the checkpoint frequency. The protocol described in the referenced paper considers a release consistency model, for which data modified by a process is propagated to other processes only when a synchronization variable is written. Since all inter-process communications occur when synchronization variables are manipulated, processes must save their state only when such events occur. Thus, the checkpoint frequency is lower than in the base communication-induced scheme.

The checkpointing overhead in communication-induced schemes is directly proportional to the number of inter-process communications. The saving of a checkpoint, which is an expensive operation, is performed each time a communication occurs. The number of inter-process communications is difficult to manage by the application programmer, especially in presence of false sharing (see Section 2). An evaluation of this parameter for real applications is required in our opinion, as it would allow to control the overhead of such techniques. Unfortunately, papers describing this class of techniques do not provide such information.

**Logging Inter-process Communications**

As in message-passing systems, an approach to both eliminate the domino effect and to reduce the storage requirements for checkpoints, is to log inter-process communications (see Section 3). Two issues must be addressed to use such an approach in RDSM systems. The first one is relative to the non-determinism of processes executing on top of a DSM. Indeed, on a given machine the state of a process depends on the reception time of messages originating from other machines (for example, messages related to the treatment of page faults on a node hosting a page manager). The direct use in a DSM of techniques designed for message-passing systems would require the saving of a checkpoint each time a non-deterministic event occurs (for instance, a message reception) leading to a large overhead. The second issue is related to the amount of information that need to be logged, which can be very large in page-based DSMs, as messages can contain virtual memory pages (several kilobytes of data). The following paragraphs focus on these two issues.

*Non Determinism of Message Reception*

In the recoverable DSM proposed in [FT94], the issue of non determinism of message reception is addressed by adding specific hardware counting the number of instructions executed by each process (namely, the process *age*). Each event, in particular a message reception, is timestamped with the current age of the receiving process. In normal (fault-free) functioning, each computing process, called a *primary* process has a *backup* process located on another machine. Primary processes periodically save their state on stable storage, so that the size of the message log is bounded. For each communication (a page migration between two machines), a copy of the page as well as the age of all processes on the source machine are sent to the backup machine, which logs this information. At rollback time after a failure, the backup process re-execute the instructions of the primary process starting from the last checkpoint. Re-execution proceeds until either the process age equals the age of one of the log's items or the log is empty. Page faults occurring during the re-execution are solved using the contents of the log's items, which indicate which version of the page should be used. The algorithm only requires processes to have a deterministic behavior regarding their accesses to shared memory (i.e., all processes perform memory accesses in the same order at recovery time after a failure as before the failure).

In [RIS93], the non determinism of message receptions is dealt with by logging every memory access. For every memory access, the accessed data is logged if it has not been logged yet. At recovery time after a failure, the processes of the faulty machine are restarted from their last checkpoint and all their memory accesses are satisfied with the previously logged page values. As in the algorithm described in [FT94], this technique does not require message reception times after a failure to be the same as in the failure-free execution. The only requirement is that processes have a deterministic behavior regarding their accesses to shared memory.

Note that for these two algorithms, an action is performed on each memory access. This may be expensive for DSMs relying on standard address translation hardware, as it is then necessary to raise an exception for each access to shared data.

In order to limit the amount of information to log, the recoverable DSM presented in [SJF95] logs the pages contents only in the event of a read or write page fault and counts the number of accesses that do not generate a fault.

The proposal described in [NCG94] takes advantage of the entry consistency model [BZ91], where every piece of shared data is protected by synchronization variables and are only transfered between machines when synchronization variables are accessed. As accesses to synchronization variables happen at deterministic times, it is sufficient to log the object contents when associated synchronization variables are accessed.

*Decreasing the Amount of Information to Log*

All log-based techniques rely on the saving in stable storage of data exchanged between processes. In DSM relying on paging hardware, transfered data contain virtual memory pages, thus resulting in saving a large amount of data. Various techniques have been proposed in order to reduce the amount of saved data. In [FT94], instead of logging the page contents, a version number of the page is saved. Each process maintains a list of the old versions of each page in volatile memory. This list is used in the recovery step after a failure to satisfy page faults. As only a single simultaneous failure is assumed in the referenced paper, it is sufficient to ensure that the faulty machine is different from the one used for recovery. This kind of technique is well-suited to systems for which network accesses are faster than disk accesses.

### 4.2.3 Other Techniques

Some proposals of RDSMs rely on specific error recovery techniques. Two of them are sketched below.

The recoverable DSMs described in [Bro93], [BW94] and [BW95] rely on snooping inter-machine communications in order to store the information required to restart processes after a failure. These algorithms are particularly well-suited to DSMs executed on top of broadcast networks like Ethernet. In the simplest algorithm, called *Single Snooper* (SS), a unique machine snoops all DSM messages on the network (page fault messages, page invalidation messages). The snooping machine stores the most recent version of each page and the identity of its current owner. Two optimizations of the basic algorithm, called respectively *Multiple Snooper* (MS) and *Integrated Snooper* (IS) are also proposed in the referenced papers. The MS algorithm eliminates the bottleneck of a unique snooper by creating multiple snooping machines, distinct from the compute machines. In the IS algorithm, any machine can simultaneously act as a snooper and as a compute machine. For both MS and IS schemes, all dirty pages of one node must be flushed atomically to their snooping node when a modified page migrates from this node to another. As several snooping nodes may be involved, a distributed commit protocol must be implemented.

A transactional recoverable DSM is presented in [FCNL94]. Memory accesses are performed within transactions. Modifications to shared data are logged using a traditional database logging technique. The contribution of this work consists in using data contained in the log to maintain consistency of shared data (log-based consistency). Since the log contains all the information required to manage consistency, it is used both for recovery and for consistency management. Hence, the same optimization strategy is used for both functions.

## 4.3 Managing Distributed Shared Memory Data Structures

Processes private data must be saved to allow processes to restart their execution after a failure. In addition, DSM internal data structures, like local directories, global directory and list of pending page fault requests can also be saved on stable storage. This paragraph discusses the impact that the saving of DSM data structures has on applications performance, especially during recovery. For the sake of clarity, we consider here a page-based DSM using a central manager in which the global directory maintains for each page the identity of all machines having a copy.

### 4.3.1 Not Saving the DSM Data Structures

A first approach, taken in [RIS93, JT94, KCG+95, JF95, SJF95], does not save local and global directories when processes save a checkpoint. The amount of information to be saved is thus limited. However, the recovery protocol after a failure is more complex and less efficient, as directories must be reconstructed at recovery time. The reconstruction of the local directory, which has to be performed after a site failure is quite simple. If processes are restarted by loading all shared pages in volatile memory, every directory entry must be set to a value that indicates that the corresponding page is loaded in memory. When shared pages are loaded on demand in memory after a failure, it is sufficient to set all the directory entries to *invalid* to indicate that a page is not present in memory.

The global directory is lost when the manager site fails. A new manager must then be elected in the event of a permanent failure before reconstructing the global directory. The global directory can be reconstructed either before or after application processes are restarted. The former case imposes a cooperation of all sites at recovery time to communicate to the new manager the identity of pages for which they have a copy. In the latter case, each time an action is performed on a given page (for instance, on page fault) all machines having a copy of this page must be found. A search message must be broadcast to all sites thus increasing the network traffic after a failure.

Note that some volatile DSM data structures (like for instance the list of pending page fault requests, or the locks maintained at the manager node) are lost when a failure occurs. Consequently, all RDSMs choosing not to save the DSM data structures require processes to save their checkpoint when the contents of the DSM data structures are predictable (for instance, when there are no pending page fault requests).

### 4.3.2 Saving the DSM Data Structures

In order to limit the time spent to reconstruct directories after a failure, directories can be saved with a checkpoint. The main problem is then to ensure that directories values and shared data are mutually consistent.

This problem is inherently solved in systems based on consistent checkpointing. For instance, in [CMP95], all computational processes and the manager process are coordinated to store their respective state (process state, local and global directories), thus ensuring that shared data and DSM data structures are consistent with each other.

In systems based on independent checkpointing, special care must be taken to ensure that directories and shared data are mutually consistent. In [TH90], each DSM transaction (page owner change, addition of a page fault request in a waiting list) is executed within a distributed atomic action. The main feature of the implementation of the distributed atomic actions proposed in the referenced paper is the use of a *Unilateral Commit* protocol to commit atomic actions. This implementation is less expensive than a traditional two-phase commit protocol. It ensures weak consistency of DSM data structures stored on stable storage. The atomic actions updating data stored on stable storage are made up of sub-actions (one per site involved) that are independently committed. An identifier is associated with each sub-action, permitting to know its outcome in the event of a failure.

## 4.4 Storage Support for Checkpoints

So far, the existence of a stable storage for saving checkpoints has been assumed. Three different strategies may be used to implement stable storage in a recoverable DSM. The first and most classical one is to save checkpoints on disk. Such an approach is described in paragraph 4.5. Although disk drives are cheap and provide non volatile storage, they exhibit low throughput and high latency. Moreover, some sites (e.g. nodes of parallel machines) may not have a disk. To overcome these problems, a few recoverable DSM proposals use the machines volatile memory to store checkpoints.

The use of this second strategy in RDSMs is presented in paragraph 4.6. Finally, in paragraph 4.7, we briefly describe other approaches where disks and site volatile memories are jointly used to store checkpoints.

## 4.5 Saving of Checkpoints on Disk

In most recoverable DSMs, checkpoints are saved on disk [CMP95, JT94, WF90]. Disk failures problems are not addressed in the proposals we have studied.

In [WF90], the stable disk server uses a twin-page technique allowing incremental saving of checkpoints and efficient recovery after a site failure. Modified memory pages can be copied onto disk at any time. The recovery algorithm does not require disk transfers to restore the last checkpoint and to discard invalid data. In the proposed technique, two timestamped versions of each page are maintained on disk. Additional data (checkpoint and rollback counters) stored on disk, is also managed to identify the correct version of page on read and write accesses. At any time, one of the two twin-pages contains data belonging to the current checkpoint, while the other one contains either (i) current data that have been updated since the last checkpoint, (ii) invalid data if a rollback has occured or (iii) obsolete data belonging to an old checkpoint. A vector is associated with each pair of pages on disk. It contains the identity $p$ of the process which has performed the last update on the page, the time of this last update, and $p$'s checkpoint and rollback counters (CS and RS). By comparing the vector information with the information associated with each process, the counters CS and RS, which are also saved on disk, the disk server knows on a read request which version of the page can be read and on a write request which one can be overwritten.

## 4.6 Saving of Checkpoints on Volatile Memory

In a network of workstations where sites are failure-independent, sites memory can be used to implement a stable storage device. The stability of data is guaranteed by its replication in the memory of two distinct sites. In the recoverable DSM proposed in [Fle90, KCG+95, Wil93, SZ90] checkpoints are stored on volatile memory. With such an approach, the architecture scalability is preserved. When a site is added to the system, more memory is available but the disk space is not necessarily increased. Moreover, the throughput of volatile memories is much higher than the disk one.

As an illustration of this approach, we describe the page-based recoverable DSM presented in [KCG+95]. The persistence of checkpoints is ensured by replicating checkpoint pages in the memory of two distinct sites. As both current[2] and checkpoint pages are stored on volatile memory, they can be all managed by the consistency protocol which is extended with: (i) new states to identify checkpoint pages, and (ii) new transitions representing the saving and restoration of a checkpoint. An advantage is that checkpoint pages can be read by processors as long as they have not been modified since the last checkpoint. When a checkpoint is saved (using a traditional two-phase commit algorithm), two checkpoint pages are created for each current page that has been modified since the last checkpoint (incremental scheme). The first checkpoint page is obtained by simply changing the state of the current page and the second one by replicating the page in another memory. For replicated pages, an optimization consists in choosing one of the replica to become the second checkpoint page, thus avoiding a page transfer and the creation of an additional copy. The rollback algorithm consists in invalidating all current pages and restoring checkpoint pages as current ones. The absence of fixed physical location of DSM pages greatly simplifies the reconfiguration step necessary after a permanent failure. Lost pages can indeed be reallocated in any valid site without any address modification. They are subsequently migrated or replicated using standard mechanisms provided by the DSM system.

The recoverable DSM described in [Wil93] is similar on some aspects to the above proposal but is only applicable to DSM systems using a directory-based consistency protocol. Checkpoint pages are indeed identified by pointers stored in directories instead of using states of the consistency protocol. Thus, they can only be used for recovery purpose. To save a checkpoint, processes are blocked while data that has been modified since the last checkpoint is marked. They are restarted as soon as this operation is finished. Checkpoint pages are eventually created asynchronously using the copy-on-write mechanism.

In approaches described in [KCG+95, Wil93], recovery data is explicitly exported by the sites to a remote memory. Brown suggests a dual method in [Bro93]. A specific site, called *snooper*, maintains a database containing pages and directories in its volatile memory. It snoops the network to update its information. When a modified page migrates, all modified pages of the originating machine are atomically flushed to the snooper site.

Stumm and Zhou have also proposed to save recovery data on volatile memory [SZ90]. Each time a page is transmitted to a remote site, a copy is also kept locally, in order to ensure the replication of each shared page on two distinct sites. In addition, if a dirty page migrates from site $S_1$ to site $S_2$, all modified pages of $S_1$ are atomically copied to $S_2$ memory. A sequence number

[2] Current pages are those used by processors for computation.

is associated with each saved state to be able to compute a global consistent system state in the event of a rollback. The major drawbacks of this approach are the large amount of data that must be saved on volatile memory, and the high checkpoint frequency (an atomic flush of dirty pages is required at each page migration). Moreover, a garbage collector is required to discard obsolete data. These characteristics are in our opinion incompatible with a realistic implementation.

### 4.7 Saving of Checkpoints on both Disk and Volatile Memory

Some propositions use an intermediate approach, in the sense that checkpoints are temporarily saved on volatile memory before being copied onto disk.

In [TH90], each site has a local disk which is assumed to be reliable and is used as a log support. At the end of a set of modifications of shared data, which is completed within a transaction, the log is saved on volatile memory. The migration of a dirty page causes the flush of the log onto disk. A similar approach is chosen in [SJF95].

In the scheme proposed in [Wil93], checkpoints are normally saved on the nodes volatile memories to support a single node failure. However, they can additionally be saved on disk in order to protect the system against double faults and catastrophic failures. The disk checkpoint, called *persistent checkpoint*, is created in the background from the checkpoint existing in the nodes volatile memory. Recovery data which is copied onto disk does not remain in volatile memory. The persistent checkpoint only concerns shared data and cannot be used to restart active processes in the event of a failure. The creation of a persistent checkpoint is triggered either by an explicit request from the application programmer, when a lack of volatile memory is detected, or periodically by the system.

## 5 Summary

A summary of the properties of existing recoverable DSM propositions is presented in the following table. The first part of each line gives the context in which the work has been realized (target DSM, system model). By default, a set of *fail-stop* processors interconnected by a reliable communication network and using a shared reliable disk server is assumed. Likewise, the opposite is explicitly stated, the target DSM is a page-based DSM implementing the sequential consistency model through a write-invalidate protocol. The second part of each line details the characteristics of each proposal (basic checkpointing algorithm, storage support for checkpoints and the way DSM internal data structures are dealt with). Finally, the third part of each line provides information on the performance of the described RDSM in terms of additional memory occupation (volatile memory and disk), number of messages and number of sites involved in the rollback after a failure. An empty performance box indicates that papers describing the proposition do not give enough information on this topic.

The recoverable DSM appearing in the table are presented by chronological order of publication. Few details are provided for the oldest publications ([Fle90] or [SZ90] as examples) since these papers focus on the issues of building recoverable DSMs rather that giving complete solutions. Concerning the algorithms of the RDSM presented in [SZ90], only the fourth and most sophisticated one is given in our summary. For [JF93], only the protocol exploiting the lazy release consistency protocol is detailed (LRC2 algorithm in the referenced paper) as it is the most representative of the class of solutions presented in this paper.

## 6 Conclusion

This paper surveys existing recoverable DSMs. We propose a taxonomy of RDSMs based on three orthogonal criteria. The main one is the checkpointing algorithm. The traditional checkpointing techniques for message-passing systems have been adapted more or less successfully to DSM systems taking into account their characteristics (message traffic, message size, consistency protocol). The two other criterions are the decision of saving or not DSM data structures and the way the stable storage is implemented.

A great deal of research has been performed in the last five years on RDSM systems. It appears that many approaches are described in the literature but few have been deeply evaluated. It would be now interesting to compare quantitatively the various proposals in a common framework. Research in the RDSM area should now focus on implementation in large-scale systems and experimentation with realistic applications.

| Target | | | Method | | | Performance | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **Target DSM** | **Hypothesis** | **Method** | **Support** | **DSM data** | **Memory** | **Messages** | **Involved sites** |
| [SZ90] | Shared data location by broadcast | Single failure at a time | Independent - recovery point established for each communication | Volatile memory | Not applicable | Multiple copies in volatile memory (a garbage collector is needed to destroy old copies) | Additional messages each time a modified data migrates | Only one site |
| [Fle90] | Mirage [FP89] (statically distributed manager) | Single failure at a time | Independent - recovery point established for each communication | Volatile memory | Replicated | Two copies of data and directories in volatile memory | | |
| [WF90] | Centralized manager | Transient and permanent failures | Independent - recovery point established for each communication | Disk | Not saved | recovery points on disk | Additional messages at each inter-process communication | A single site, except in the event of the manager failure |
| [TH90] | Statically distributed manager | Per site stable storage (disk) | Independent - recovery point established for each communication | Disk | Saved within transactions | Recovery points on disk - volatile memory log copied on disk each time a communication occurs | | A single site |
| [CCD+93] | Munin [CBZ91] (multiple consistency protocols) | | Pure consistent scheme | Disk | Not considered | Recovery points on disk | Synchronization messages for the establishment of recovery points | All sites |
| [JF93] (LRC2 algo.) | Lazy release consistency (data is transferred when synchronization variables are accessed) | | Independent - a recovery point must be established at each synchronization point | Disk | Not considered | Recovery points on disk | Additional messages for each access to a synchronization variable | A single site |
| [Bro93] (SS algo.) | Dynamically distributed manager | Broadcast network - no failure tolerated during the restart of a snoopy site | Snooping of communications | Volatile memory and disk of the snoopy site | Saved | Two copies of data and directories | No additinal message | Faulty site and snoopy site |

| **Target** | | | **Method** | | | **Performance** | | |
|---|---|---|---|---|---|---|---|---|---|
| **Ref.** | **Target DSM** | **Hypothesis** | **Method** | **Support** | **DSM data** | **Memory** | **Messages** | **Involved sites** |
| [Bro93] (MS algo.) | Dynamically distributed manager | Broadcast network - No failure tolerated during the recovery of snoopy sites | Snooping of communications | Volatile memory and snoopy sites' disks | Saved | Two copies of data and directories | No additional message | Faulty site and relevant snoopy sites |
| [Bro93] [BW94] [BW95] (IS algo.) | Dynamically distributed manager | No simultaneous failure of a site and of a snoopy site managing one of its pages | Snooping of communications | Volatile Memory and snoopy sites' disks | Saved | Two copies of data and directories | Twice the number of DSM messages when the network does not provide broadcast | Set of snoopy sites that have copies of lost pages |
| [Wil93] | Arius system [SW92] (multiple consistency protocols, Arius system's directories) | Single failure at a time | Consistent - management of dependencies to limit the number of processes to synchronize | Volatile memory | | | | Dependent sites |
| [RIS93] | Dynamically distributed manager | Single failure at a time | Independent - Log of memory accesses | Volatile memory and disk | Not saved | Checkpoints and log of communications on disk | No additional message | A single site |
| [NCG94] | Entry consistency - adaptation of K. Li dynamically distributed manager | Process determinism for memory accesses | Independent - log of modified data at synchronization points | Disk | Saved | Log and checkpoints | Messages piggy-backed on standard DSM messages | Dependent sites |
| [FT94] | Centralized manager | Single failure at a time - Process determinism for memory accesses - hardware mechanism for measuring the age of processes | Independent - log of data at each communication | Disk | Local directory saved | Checkpoint on disk - list of messages and obsolete pages in volatile memory | Additional messages when processes communicate | All sites |
| [JT94] | ad hoc directories | FIFO links between sites | Consistent - management of dependencies to limit the number of synchronized processes | Disk | Not saved | Checkpoints on disk | | All sites |

| Target | | | Method | | | Performance | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **Target DSM** | **Hypothesis** | **Method** | **Support** | **DSM data** | **Memory** | **Messages** | **Involved sites** |
| [FCNL94] | Transactional DSM | | Use of transaction management data structures (log) to managed shared data consistency. | Disk | Not applicable | Checkpoints and log of modifications on disk | | |
| [CMP95] | Myoan [CPP94] (multiple consistency protocols) | | Consistent - unification of synchronization barriers and checkpoint saving | Disk | Saved | Checkpoints on disk | Messages required for synchronization when checkpoints are established | All sites |
| [KCG+95] | | Single failure at a time | Consistent | Volatile memory | Not saved | One or two copies of each page in volatile memory | Messages required for synchronization when checkpoints are established | All sites |
| [SJF95] | | Process determinism for memory accesses | Independent - Log of data when communications occur | Disk | Not saved | Checkpoint on disk - log in volatile memory but copied on disk when a communication occurs | Messages required to force the log to be copied on disk when a communication occurs | A single site |
| [JF95] | | | Independent - dependency management - domino effect limited by the fixed period for checkpoint establishment | Disk | Not saved | Several checkpoints on disk (until a garbage-collector is activated) | No additional message | Dependent sites |

## References

[AL91] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[BGH87] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer systems. In A. Avizienis, H. Kopetz, and J.C. Laprie, editors, *The Evolution of Fault-Tolerant Computing*, volume 1, pages 55–76. Springer Verlag, 1987.

[Bir85] K.P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. of 10th ACM Symposium on Operating Systems Principles*, pages 79–86, Washington, December 1985.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in Orca. *IEEE Transactions on Software Engineering*, 18:190–205, March 1992.

[Bro93] L. Brown. *Fault Tolerant Distributed Shared Memories*. PhD thesis, Florida Atlantic University, December 1993.

[BW94] L. Brown and J. Wu. Dynamic snooping in a fault-tolerant distributed shared memory. In *Proc. of 14th International Conference on Distributed Computing Systems*, pages 218–226, Poznan, Poland, June 1994.

[BW95] L. Brown and J. Wu. Fault tolerant distributed shared memories. *The Journal of Systems and Software*, 29(2):149–165, May 1995.

[BZ91] B. N. Bershad and M. J. Zekauskas. Midway : Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Research Report CMU-CS-91-170, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, September 1991.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CCD+93] J. B. Carter, A. L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B Johnson, P. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel. Network multicomputing using recoverable distributed shared memory. In *Proc. of the IEEE International Conference CompCon'93*, February 1993.

[CG86] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4:110–129, May 1986.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CMP95] G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory systems. In *Proc. of the 14th Symposium on Reliable Distributed Systems*, pages 96–105, Bad Neuenahr, Germany, September 1995.

[CPP94] G. Cabillic, T. Priol, and I. Puaut. MYOAN: an implementation of the KOAN shared virtual memory on the Intel Paragon. Research Report 812, IRISA, March 1994.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, June 1986.

[FCNL94] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating coherency and recoverability in distributed systems. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[Fle90] B. D. Fleisch. Reliable distributed shared memory. In *Proc. 2nd Workshop on Experimental Distributed Systems*, pages 102–105, 1990.

[FP89] B. D. Fleisch and G. J. Popek. Mirage : a coherent shared memory design. In *Proc. of 12th ACM Symposium on Operating Systems Principles*, Operating System Review, pages 211–223, December 1989.

[FT94] T. Fuchi and M. Tokoro. A mechanism for recoverable shared virtual memory. Manuscript. University of Tokyo, May 1994.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In

*Proc. of 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[Gra78] J. Gray. *Notes on Database Operating Systems.*, volume 60 of *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[HS87] E. S. Harrison and E. Schmitt. The structure of system/88, a fault-tolerant computer. *IBM Systems Journal*, 26(3):293–318, 1987.

[Int93] Intel Corporation. *Paragon User's Guide*, 1993.

[JF93] B. Janssens and W. K. Fuchs. Relaxing consistency in recoverable distributed shared memory. In *Proc. of 23rd International Symposium on Fault-Tolerant Computing Systems*, pages 155–163, Toulouse, France, June 1993.

[JF94] B. Janssens and W. K. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, pages 34–41, Dana Point, CA, October 1994.

[JF95] B. Janssens and W. K. Fuchs. Ensuring correct rollback recovery in distributed shared memory systems. *Journal of Parallel and Distributed Computing*, October 1995.

[JT94] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, pages 42–51, Dana Point, CA, October 1994.

[KCG+95] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc. of 25th International Symposium on Fault-Tolerant Computing Systems*, pages 289–298, June 1995.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[KYA86] K.H. Kim, J.H. You, and A. Abouelnaga. A scheme for coordinated execution of independently design recoverable distributed processes. In *Proc. of 16th International Symposium on Fault-Tolerant Computing Systems*, pages 130–135, Vienne, Autriche, July 1986.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, second revised edition, 1990.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[Lam81] 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.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–357, November 1989.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, 1986.

[LNP90] K. Li, J.F. Naughton, and J.S. Plank. Real-time concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice Parallel Programming (PPOPP), SIGPLAN notices*, volume 25, pages 79–88, 1990.

[NCG94] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proc. of 13th International Symposium on Principles of Distributed Computing*, August 1994.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, August 1991.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[RIS93] G. G. Richard-III and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proc. of the 12th Symposium on Reliable Distributed Systems*, pages 58–67, 1993.

[Sch87] F. B. Schneider. The fail-stop processor approach. In *Concurency control and reliability in distributed systems*, pages 370–394. Barghava, 1987. Chapter 13.

[SJF95] G. Suri, B. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *Proc. of 25th International Symposium on Fault-Tolerant Computing Systems*, Pasadena, CA, June 1995.

[SW92] A. Saulsbury and T. Wilkinson. The design of a unifying 64-bit distributed architecture. Technical report, Swedish Institute of Computer Science, 1992.

[SY85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[SZ90] M. Stumm and S. Zhou. Fault tolerant distributed shared memory algorithms. In *Proc. of 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, Dallas, Texas, December 1990.

[TH90] V. O. Tam and M. Hsu. Fast recovery in distributed shared virtual memory systems. In *Proc. of 10th International Conference on Distributed Computing Systems*, pages 38–45, Paris, France, May 1990.

[WF90] K. L. Wu and W. K. Fuchs. Recoverable distributed shared memory: Memory coherence and storage structures. *IEEE Transactions on Computers*, 34(4):460–469, April 1990.

[Wil93] T. J. Wilkinson. *Implementing Fault Tolerance in a 64-bit Distributed Operating System.* PhD thesis, City University of London, July 1993.