Checkpointing Distributed Shared Memory

LUIS M. SILVA JOÃO GABRIEL SILVA

Departamento Engenharia Informática
Universidade de Coimbra
POLO II - Vila Franca
P-3030 - Coimbra
PORTUGAL
Email: {luis, jgabriel}@dei.uc.pt

ABSTRACT

Distributed shared memory (DSM) is a very promising programming model for exploiting the parallelism of distributed memory systems, since it provides a higher level of abstraction than simple message passing. Although the nodes of standard distributed systems exhibit high crash rates only very few DSM environments have some kind of support for fault-tolerance.

In this paper, we present a checkpointing mechanism for a DSM system that is efficient and portable. It offers some portability because it is built on top of MPI and uses only the services offered by MPI and a POSIX compliant local file system.

As far as we know, this is the first real implementation of such a scheme for DSM. Along with the description of the algorithm we present experimental results obtained in a cluster of workstations. We hope that our research shows that efficient, transparent and portable checkpointing is viable for DSM systems.

Keywords: Distributed Shared Memory, Checkpointing, Fault-Tolerance, Portability.

1. INTRODUCTION

Distributed Shared Memory (DSM) systems provide the shared memory programming model on top of distributed memory systems (i.e. distributed memory multiprocessors or networks of workstations). DSM is appealing because it combines the performance and scalability of distributed memory systems with the ease of programming of shared-memory machines.

Distributed Shared Memory has received much attention in the past decade and several DSM systems have been presented in the literature [Eskicioglu95][Raina92][Nitzberg91]. However, most of the existing prominent implementations of DSM systems do not provide any support for fault-tolerance [Carter91] [Keleher94] [Li89] [Johnson95]. This is a limitation that we wanted to overcome in our DSM system.

When using parallel machines and/or workstation clusters the user should be aware that the likelihood of a processor failure increases with the number of processors, and the failure of just one process(or) will lead to the crash or hang-up of the whole application. Distributed systems represent a cost-effective solution for running scientific computations, but at the same

time they are more vulnerable to the occurrence of failures. A study presented in [Long95] shows that we can expect (in average) a failure every 8 hours in a typical distributed system composed by 40 workstations, where each machine exhibits an MTBF of 13 days. If a program that runs on such a system takes more than 8 hours to execute then it would be very difficult to finish its execution, unless there is some fault tolerance support to assure the continuity of the application. Parallel machines are considerably more stable, but even so they present a relatively low MTBF. For instance, the MTBF of a large parallel machine like the Intel Paragon XP/S 150 (with 1024 processors) from Oak Ridge National Laboratory is in the order of 20 hours [ORNL95].

For the case of long-running scientific applications it is essential to have a checkpointing mechanism that would assure the continuity of the application despite the occurrence of failures. Without such mechanism the application would have to be restarted from scratch and this can be very costly for some applications.

In this paper, we will present a checkpointing scheme for DSM systems that despite being transparent to the application it is quite general, portable and efficient. The scheme is quite general because it does not depend of any specific feature of our DSM system. In fact, our system implements several protocols of consistency and models of consistency, but the checkpointing scheme was made independent of the protocols.

The scheme is also quite portable, since it was implemented on top of MPI and nothing was changed inside the MPI layer. This is why we call it DSMPI [Silva97]. The scheme only requires a POSIX compliant file system, and makes use of the *likckpt* tool [Plank95] for taking the local checkpoints of processes. That tool works for standard UNIX machines.

Finally, DSMPI is an efficient implementation of checkpointing. Usually, efficiency depends on the write latency to the stable storage but also on the characteristics of the checkpointing protocol. While the first feature mainly depends on the underlying system, the second one is under our control. We have implemented a non-blocking coordinated checkpointing algorithm. It does not freeze the whole application while the checkpoint operation is being done, as blocking algorithms do. The only problem of non-blocking algorithms over the blocking ones is the need to record in stable storage some of the in-transit messages that cross the checkpoint line. However, we have exploited the semantics of DSM messages and we achieved an important optimization: no in-transit message has to be logged in stable storage. Some results were taken using a distributed stable storage and we have observed a maximum overhead of 6% for a extremely short interval between checkpoints of 2 minutes. With a more realistic interval, in order of tens of minutes or even hours, the overhead would fall to insignificant values.

The rest of the paper is organized as follows: section 2 describes the general organization of DSMPI and its protocols. Section 3 presents the transparent scheme that is based on a non-blocking coordinated checkpointing algorithm. Section 4 compares our algorithm with other schemes. Section 5 presents some performance results, and finally section 6 concludes the paper.

2. OVERVIEW OF DSMPI

This section gives a brief description about DSMPI [Silva97].

2.1 Main Features

DSMPI is a parallel library implemented on top of MPI [MPI94]. It provides the abstraction of a globally accessed shared memory: the user can specify some data-structures or variables to be shared and that shared data can be read and/or written by any process of an MPI application.

The most important guidelines that we took into account during the design of DSMPI were:

1. assure full portability of DSMPI programs;
2. provide an easy-to-use and flexible programming interface;
3. support heterogeneous computing platforms;
4. optimize the DSM implementation to allow execution efficiency.
5. provide support for checkpointing.

For the sake of portability DSMPI does not use any memory-management facility of the operating system, neither requires the use of any special compiler or linker. All shared data and the read/write operations should be declared explicitly by the application programmer. The sharing unit is a program variable or a data structure. DSMPI can be classified as a *structure-based* DSM as opposed to *page-based* DSM systems, like IVY [Li89].

It does not incur in the problem of false sharing because the unit of shared data is completely related to existing objects (or data structures) of the application. It allows the use of heterogeneous computing platforms since the library knows the exact format of each shared data object. Most of the other DSM systems are limited to homogeneous platforms.

DSMPI allows the coexistence of both programming models (message passing and shared data) within the same MPI application. This has been considered as a promising solution for parallel programming [Kranz93].

Concerning absolute performance, we can expect applications that use DSM to perform worse than their message passing counterparts. However, this is not always true. It really depends on the memory-access pattern of the application and on the way the DSM system manages the consistency of replicated data.

We tried to optimize the accesses to shared data by introducing three different protocols of data replication and three different models of consistency that can be adapted to each particular application in order to exploit its semantics. With such facilities we expect DSM programs to be competitive with MPI programs in terms of performance. Some performance results collected so far corroborate this expectation [Silva97].

2.2 Internal Structure

In DSMPI there are two kinds of processes: application processes and daemon processes. The latter ones are responsible for the management of replicated data and the protocols of consistency. Since the current implementations of MPI are not thread-safe we had to implement the DSMPI daemons as separate processes. This is a limitation of the current version of DSMPI that will be relaxed as soon as there is some thread-safe implementation of

MPI. All the communication between daemons and application processes is done by message passing. Each application process has access to a local cache that is located in its own address space where it keeps the copies of replicated data objects. The daemon processes maintain the master copies of the shared objects. DSMPI maintains a two-level memory hierarchy: a local cache and a remote shared memory that is located in and managed by the daemons. The ownership of the data objects is implemented through a static distributed scheme.

2.3 DSM Protocols

We provided some flexibility in the accesses to shared data by introducing three different protocols of data replication and three different models of consistency that can be adapted to each particular application in order to exploit its semantics.

Shared objects can be classified in two main classes: single-copy or multi-copy. The multi-copy class replicates the object among all the processes that perform some read request on it. In order to assure consistency of replicated data the system can use a *write-invalidate* protocol or a *write-update* protocol [Stumm90A]. This is a parameter that can be tuned by the application programmer.

In order to exploit execution efficiency, we have also implemented three different models of consistency:

1. *Sequential Consistency* (SC), as proposed in the IVY system [Li89];
2. *Release Consistency* (RC), that implements the protocol of the DASH multiprocessor [Lenoski90];
3. *Lazy Release Consistency* (LRC), that implements a similar protocol as proposed in the TreadMarks system [Keleher94].

It has been shown that the LRC protocol is able to introduce some significant improvements over the other two models. The flexibility provided by DSMPI in terms of different models and protocols is an important contribution to the overall performance of DSMPI.

2.4 Programming Interface

The library provides a C interface and the programmer calls the DSMPI functions in the same way it calls any MPI routine. The complete interface is composed of 18 routines: it includes routines for initialization and clean termination, object creation and declaration, read and write operations, and routines for synchronization like semaphores, locks and barriers. The full interface is described in [Silva97].

3. TRANSPARENT CHECKPOINTING ALGORITHM

When devising the transparent checkpointing algorithm for our DSM system we tried to achieve some objectives, like transparency, portability, low performance overhead, low memory overhead and the ability to tolerate partial and total failures of the system.

Satisfying all the previous guidelines is not an easy task and has not been possible in other proposals. Most of the existing checkpointing schemes for DSM are mainly concerned with transparency. Transparent recovery implemented at the operating system level or at the DSM-layer is an attractive idea. However, those schemes involve significant modifications in the

system which make them very difficult to port to other systems. For the sake of portability, checkpointing should be made independent of the underlying system as much as possible.

3.1 Motivations

Our checkpointing scheme is meant to be quite general because it does not depend on any specific feature of our DSM system. Our system implements several protocols of consistency and models of consistency, but the checkpointing scheme was made independent of the protocols.

The scheme itself offers some degree of portability. It was implemented on top of MPI that *per se* already provides a high-level of portability since it was accepted as the standard for message-passing. Nothing was changed inside the MPI layer. The scheme only requires a POSIX compliant file system, and makes use of the *libckpt* tool [Plank95] for taking the local checkpoints of processes. That tool works for standard UNIX machines.

We are taking transparent checkpoints and this means that it is not possible to assure checkpoint migration between machines with different architectures. However, the checkpoint mechanism itself can be ported to other DSM environments.

3.2 Coordinated Checkpointing

Our guideline was to adapt some of the checkpointing techniques used in message passing systems, since checkpointing mechanisms have been widely studied in message-passing environments. Two methods for taking checkpoints are commonly used: coordinated checkpointing and independent checkpointing. In the first method, processes have to coordinate between themselves to ensure that their local checkpoints form a consistent system state. Independent checkpointing requires no coordination between processes but it can result in some rollback propagation. To avoid the domino effect and to reduce the rollback propagation message logging is used together with independent checkpointing.

Independent checkpointing and message logging was not a very encouraging option, because DSM systems generate more messages than message-passing programs. Thus, we have chosen a coordinated checkpointing strategy for DSMPI. The reasons were manifold: it minimizes the overhead during failure-free operation since it does not need to log messages; it limits the rollback to the previous checkpoint; it avoids the domino-effect; it uses less space in stable storage; it does not require a complex garbage-collection algorithm to discard obsolete checkpoints; and finally, it is the most suitable solution to support job-swapping.

It was shown in [Elnozahi92][Plank94] that coordinated checkpointing is a very effective solution for message-passing systems. Some experimental results have shown that the overhead of synchronizing the local checkpoints is negligible when compared with the overhead of writing the checkpoints to disk.

Implementing the checkpointing algorithm underneath the DSM layer and in a transparent way to that layer is a possible alternative to provide fault-tolerance as was suggested in [Carter93]. However, this would be a very simplistic approach since the DSM system exchanges several messages not related to the application. Such approach would result in extra overhead and it would not exploit the characteristics of the DSM system.

3.3 System Model

We assume that the DSM system only uses message passing to implement its protocols. There is no notion of global time and there is no guarantee that processor clocks are synchronized in some way.

Processors are assumed to be fail-stop: there is no provision to detect or tolerate any kind of malicious failures. When a processor fails it stops sending messages and does not respond to other parts of the system. Processor failures are detected by the underlying communication system (MPI) through the use of time-outs. When one of the processes fails, the MPI layer sends a `SIGKILL` to all the other processes (application and daemons) that will terminate the complete application. MPI makes extensive use of static groups (in fact, the whole set of processes belong at the beginning to a `MPI_WORLD_COMM` group). If some process fails, some collective operations that involve the participation of the processes will certainly hang-up the application. Thus, it makes sense that a failure of just one process should result in the rollback of the entire application.

Communication failures are also dealt by the communication system (MPI). The underlying message-passing system provides a reliable and FIFO point-to-point communication service.

Stable storage is implemented on a shared disk that is assumed to be reliable. If the disk is attached to a central file server all the checkpoints become available to the working processors of the system. If stable storage is implemented on the local disk of every processor (assuming that each host has a disk) then the application can only recover if the failed host is also able to recover.

3.4 Checkpoint Contents

A checkpoint in a DSM system should include: the DSM data (e.g. pages/objects and the DSM directories) and the private data of each application.

Some schemes do not save the private data of processes and thus are not able to recover the whole state of the computation [Stumm90B]. They leave to the application programmer the responsibility of saving the computation state to assure the continuity of the application. Other schemes assume, for the sake of simplicity, that private data and shared data are allocated in DSM [Kermarrec95]. We depart from that assumption and consider that private data is not allocated in the DSM system. Besides, some of the private data like processor registers, program counter and process stack are certainly not part of the DSM data.

In our scheme we have to checkpoint the application processes as well as the DSM daemons since they maintain most of the DSM relevant data. DSM daemons save the shared objects and the associated DSM directories. Saved directories reflect the location of the default and current owners of the shared objects. Some optimizations are made by the system: read-only objects are checkpointed only once, and replicated shared objects are checkpointed only by one daemon (the one that maintains its ownership).

Each process (application or daemon) saves its checkpoint into a separate file. A global checkpoint is composed by N different checkpoint files and a *status_file* that keeps the status of the checkpoint protocol execution. This file is maintained by the checkpointing coordinator and is used during the phase of recovery to determine the last committed checkpoint, as well as to ensure the atomicity in the operation of writing a global checkpoint.

3.5 Checkpointing Algorithm

Since checkpointing schemes for message passing systems are very well stabilized, we decided to adopt one of the most used techniques to implement a non-blocking global checkpointing [Elnozahy92][Silva92].

The main difficulty of implementing non-blocking coordinated checkpoint is to guarantee that the global saved state is consistent. Messages in-transit at the time of a global checkpoint are the main concern for saving a consistent snapshot of a distributed application.

For instance, messages that are sent after the checkpoint of the process sender and are received before the checkpoint of the receiver are called *orphan* messages [Silva92]. This sort of messages violates the consistency of the global checkpoint, and thus should be avoided by the algorithm. Other messages that are sent before the checkpoint of the sender and are received after the checkpoint of the destination process are called *missing* messages. Usually the algorithm should keep track of their occurrence and replay them during the recovery operation.

However, we have considered some features of the DSM system that were important for the implementation of the algorithm, namely: the interaction between processes is not done by explicit messages but through object invocations that are RPC-like interactions; there is both shared and replicated data; the DSM system exchanges some additional messages that are only related to the DSM protocols and do not affect the application directly; and finally, there is a DSM directory that is maintained throughout the system.

These features were taken into account and some of them were exploited to introduce some optimizations. The resulting scheme presents a novel but important feature over those non-blocking algorithms oriented to message-passing: it does not need to record any cross-checkpoint message in stable storage.

The operation of checkpointing is triggered periodically by a timer mechanism. One of the DSM daemons acts like the coordinator (the Master daemon) and is responsible for initiating a global checkpoint and coordinating the steps of the protocol. Only one process is given the right to initiate a checkpointing session in order to avoid multiple sessions and an uncontrolled high-frequency of checkpoint operations. Since there is always one DSMPI daemon that is elected as the Master (during the startup phase) we can guarantee that if this daemon is the checkpoint coordinator, checkpointing will be adequately spaced in time.

Each global checkpoint is identified by a monotonically increasing number - the Checkpoint Number (CN). In the first phase of the protocol, the coordinator daemon increments its own CN and broadcasts a "`TAKE_CHKP`" message to all the other daemons and application processes. Upon receiving this message each of the other processes takes a tentative checkpoint, increments the local CN and sends a message "`TOOK_CHKP`" to the coordinator. After taking the tentative checkpoint, every process is allowed to continue with its computation. The application does not need to be frozen during the execution of the checkpointing protocol. This is an important feature to avoid interference with the application and to reduce the checkpointing overhead.

In the second phase of the protocol, the daemon broadcasts a "`COMMIT`" message after receiving all the responses (i.e. "`TOOK_CHKP`") from the participants. Upon receiving a

"COMMIT" message the tentative checkpoints are transformed into permanent checkpoints and the previous checkpoint files are deleted. All these phases are recorded into the *status_file* by the Master Daemon.

Usually, the broadcast message "TAKE_CHKP" is received by all the processes in some order that preserves the causality. However, due to the asynchrony of the system it is possible that some situations may violate the causal consistency.

An important aspect is that every shared object is tagged with the local CN value and every message sent by the DSM system is tagged with the CN of the sender.

The CN value piggybacked in the DSM messages prevents the occurrence of *orphan* messages: if a daemon receives a message with higher CN than the local one, then it has to take a tentative checkpoint before consuming that message and changing its internal state. If later on, it receives a "TAKE_CHKP" message tagged with an equal CN, then it discards that message since the corresponding tentative checkpoint was already taken.

The CN value is also helpful in identifying *missing* messages: if an incoming message carries a CN value lower than the current local one, it means it was sent in the previous checkpoint interval and is a potential *missing* message.

The checkpointing algorithm has to distinguish between the messages used by the read/write operations and the other messages used by the DSM protocols. Using the semantics of DSM protocols we can avoid the unnecessary logging of some potential *missing* messages.

Daemon processes run in a cycle and when they receive a "TAKE_CHKP" message they take a local snapshot of their internal state, including the DSM directories. Application processes get "TAKE_CHKP" orders when they execute DSMPI routines: whenever they read from the local cache, perform some remote object invocation, or access some synchronization variable.

All the invocations on shared objects or synchronization variables involve a two-way interaction: invocation and response. During the period that a process is waiting for a response it remains blocked and thus does not change its internal state. The only interactions that do not fit in this structure are messages related to the DSM protocols. Usually these messages are originated by the daemon processes and do not require an RPC-like interaction. For the sake of clarity, let us distinguish between three different cases:

Case 1: Messages Process-to-Daemon

These interactions are started by an application process that wants to perform a read/write operation into a shared object or gain access to a synchronization variable. Each process maintains a directory with the location of the owners of the objects, locks, semaphores and barriers. If it wants to access any of them it sends an invocation message to the respective daemon. Then it blocks while waiting for the response. Let us consider two different scenarios that should be handled by the algorithm:

1.1 - The process (P_i) is running in the checkpoint interval N, but the daemon (D_k) is still running in the previous checkpoint interval (i.e. CN = N-1);

1.2 - The process (P_i) has its CN equal to (N-1) and the daemon has already taken its N^{th} tentative checkpoint (CN=N);

An example of the first scenario is illustrated in Figure 1. Process P_i already took its N^{th} checkpoint and performs a read access to a remote object that is owned by daemon D_k that has not yet received the corresponding "TAKE_CHKP" message. The "READ" message carries CN equal to N: the daemon checks that and takes a local checkpoint before consuming the message. Instead of a READ operation, it can be a WRITE, LOCK, UNLOCK, WAIT, SIGNAL or BARRIER invocation. All these operations have an associated acknowledge or reply message.

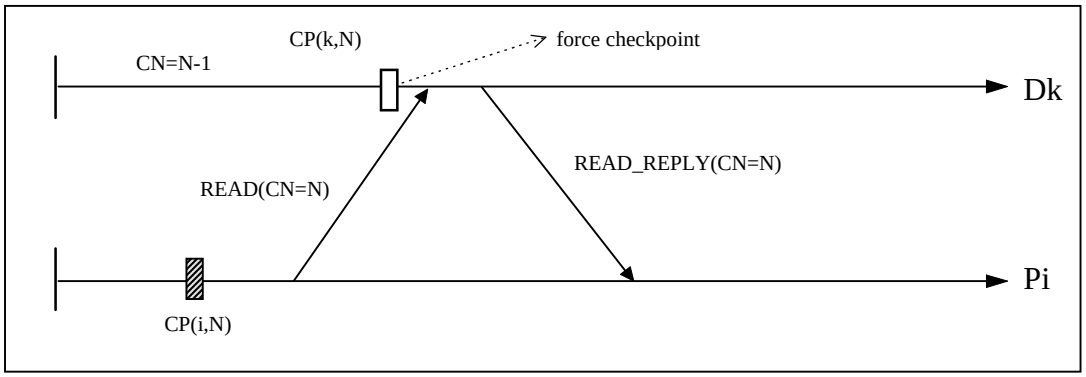

Figure 1: Forcing a checkpoint in the DSM daemon.

Figure 2 represents the second scenario, where the daemon has already taken its N^{th} checkpoint but the process started a read transaction in the previous checkpoint interval. When it receives the "READ_REPLY" the process realizes it has to take a local checkpoint and increment its local CN before consuming the message and continue with the computation.

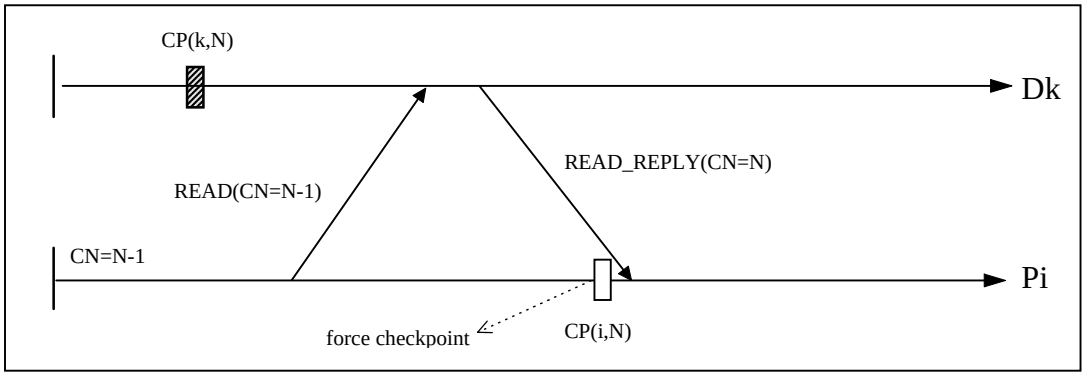

Figure 2: Forcing a checkpoint in the application process.

However, this operation is not enough since the checkpoint CP(i,N) does not record the "READ" invocation message. During recovery the process has to repeat that read transaction again. To do that, the checkpoint routine has to record the "READ" invocation message in the contents of the checkpoint. In some sense, we can say that there is a logical checkpoint immediately before the sending of the "READ" message as represented in Figure 3. At the time of the checkpoint, that invocation message belongs to the address space of the process. Therefore, it is already included in the checkpoint. The only concern was to re-direct the starting point after recovery to some point in the code immediately before the sending of the message. For that purpose, a label was included in every DSMPI routine. After returning from

a restart operation the control flow jumps to that label, and the invocation message is re-sent. Thus, we can assure that the read transaction is repeated from its beginning.

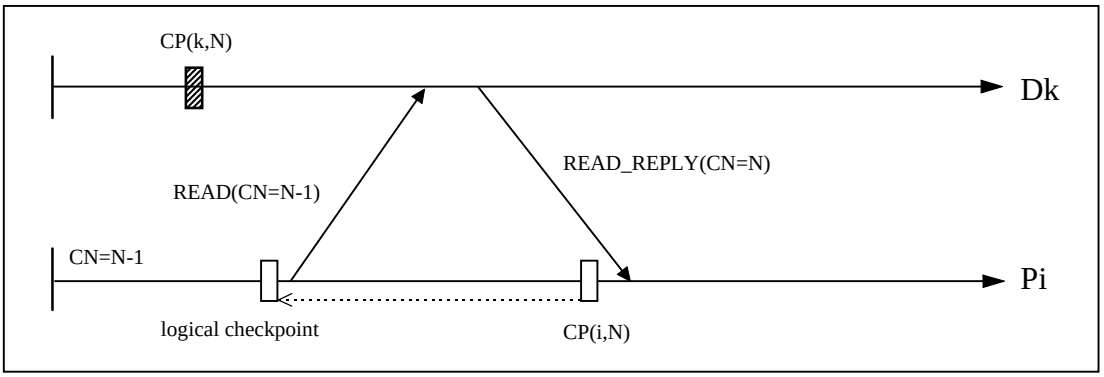

Figure 3: The notion of a logical checkpoint.

Case 2: Messages Daemon-to-Process

These messages are started by the daemon processes and are related to the DSM protocols. Usually they are `INVALIDATE` or `UPDATE` messages, depending on the replication protocol. These messages do not follow an RPC-like structure and thus do not block the sending daemon. Application processes consume these sort of messages when they execute the cache refresh procedure. We can also identify two different scenarios:

2.1- A protocol message that can be a potential *orphan* message;

2.2- A protocol message that can be a potential *missing* message.

The first situation is represented in Figure 4: process P_i receives a message that carries a higher CN than the local one, and is forced to take a checkpoint before proceeding.

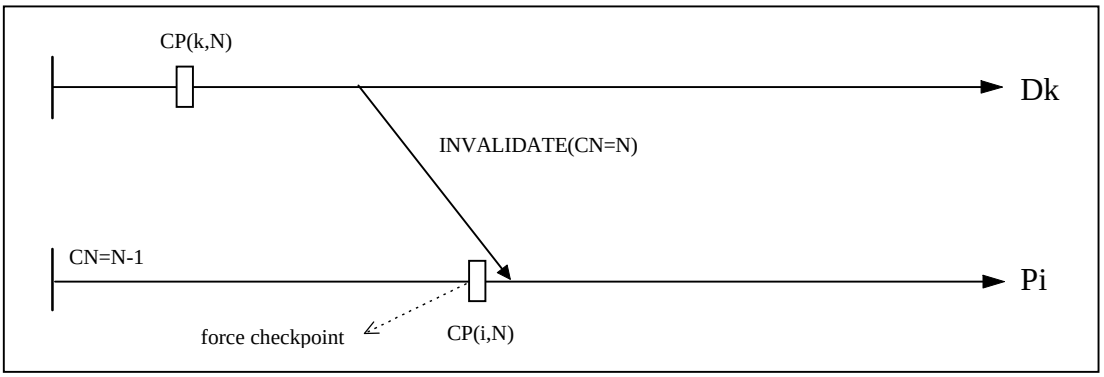

Figure 4: Potential *orphan* message.

The second scenario is illustrated in Figure 5: the `INVALIDATE` or `UPDATE` message is sent in the previous checkpoint interval and received by the application process after taking its local checkpoint. This is, theoretically, the example of a *missing* message and in the normal case it would have to be recorded in stable storage in order to be replayed in case of recovery.

However, we do not need to log these *missing* `INVALIDATE/UPDATE` messages. There is absolutely no problem if the message is not replayed in case of a rollback. The reason is simple: during the recovery procedure, every application process has to clean its local cache.

After that, if the process accesses some shared object it has to perform a remote operation to the owner daemon from where it gets the most up-to-date version of the object.

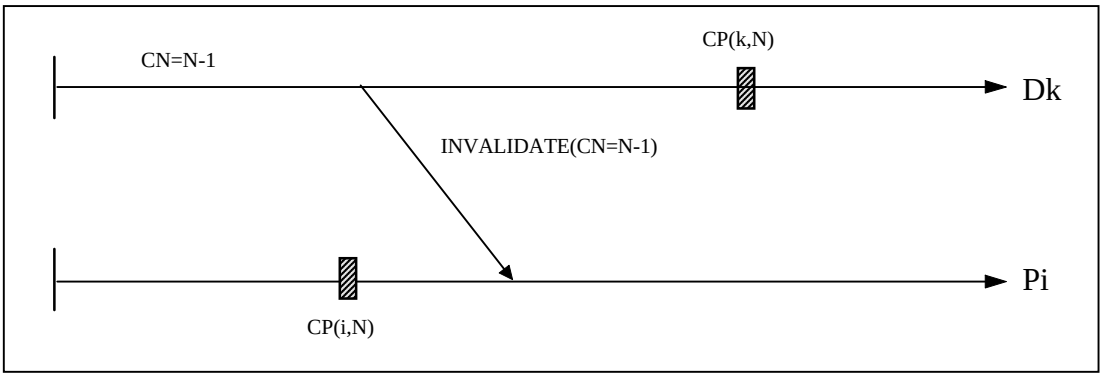

Figure 5: Example of a *missing* message.

Case 3: Messages Daemon-to-Daemon

When using a static distributed ownership scheme daemons only communicate between themselves during the startup phase and during the creation of new objects, that also happens during the startup phase. With a dynamic ownership scheme messages between daemons are sent all the time since the ownership of an object can move from daemon to daemon. The current version of DSMPI follows a static distributed scheme but the next version will provide a dynamic distributed scheme as well.

We will consider a dynamic distributed ownership scheme similar to the one presented in [Li89] where each shared data object has an associated owner that is changing as data migrates throughout the system. When a process wants to write into a migratory object the system changes its location to the daemon associated with that process. As object location can change frequently during the program execution, every process and daemon has to maintain a guess of the *probable owner* for each shared object. When a process needs a copy of the data it sends a request to the *probable owner*. If this daemon has that data object it returns the data, otherwise it forwards the request to the new owner. This forwarding can go further until the current owner is found in the chain of the *probable owners*. The current owner will send the reply to the asking process that receives the data and updates its value for the *probable owner* if the reply was not received from the expected daemon.

Sometimes this scheme can be inefficient since the request may be forwarded many times before reaching the current owner. This inefficiency can be reduced if all the daemons involved in forwarding a request are given the identity of the current owner. We will consider this optimization, which involves the sending of ownership update messages from the current owner to the daemons belonging to the forward chain.

Let us now see what are the implications of this forward-based scheme to the checkpointing algorithm. The messages exchanged between daemons can be divided into three different classes:

3.1- *forward* messages that are sent on behalf of read/write transactions;

3.2- *owner_update* messages to announce the current owner of some object;

3.3- the transfer of some object from a current owner to the next owner.

The first kind of messages follows the same rule stated for Case 1 (i.e. transactions started by a process). That rule was explained with the help of Figures 1, 2 and 3. The only difference is that all the daemon processes involved in the forwarding chain have to apply that rule.

The case of *owner_update* messages are treated in a similar way to Case 2: if the *owner_update* message carries a CN higher than the CN at the destination daemon then that message is a potential *orphan* message and should force a checkpoint at the destination before proceeding. The rule for case 2.1 (explained in Figure 4) applies to this case.

If the *owner_update* message carries a lower CN than the destination daemon it corresponds to potential *missing* message (like in Figure 5). In the normal case, it should be logged to be replayed in case of recovery. However, once again we realized that there is no problem if we do not log messages. The system will still be able to ensure a consistent recovery of the application. During the recovery procedure the distributed directory will be reconstructed through the use of broadcast to update the current ownership of the objects. This means that those *owner_update* messages can be lost during recovery. The object directory will be updated in any case.

The transfer of data from one daemon to the new owner is included in the *forward* protocol: when the reply is sent from the current owner to the original process a copy of the data is also sent (or is first sent) to the daemon associated to that process. This daemon will be the new owner. The rules stated previously are also used in this case.

Since object ownership can change frequently during the execution of the checkpoint protocol, it is necessary to take care and avoid a shared object to be checkpointed more than once. We solve this problem in a very simple way: the current owner is responsible for checkpointing the shared object. If it transfers the object to another daemon after taking its checkpoint, that object will not be checkpointed again. The CN value tagged with each object is used to prevent a migratory object to be checkpointed more than once.

To summarize, our checkpointing algorithm follows a non-blocking coordinated strategy. It avoids the occurrence of *orphan* messages and detects the potential *missing* messages. We do not log any *missing* message in stable storage but the system will still ensure a consistent recovery of the application. To achieve this optimization we have exploited some of the characteristics of the DSM protocols.

Our scheme works for sequential consistency and relaxed consistency models. It is also independent of the replication protocol (*write-update* or *write-invalidate*). This fact allows a wide applicability of this algorithm to other different DSM systems.

3.6 Recovery Procedure

In our case, the application recovery involves the roll back of all the processes to the previous checkpoint. We do not see this as a drawback, but rather as an imposition of the underlying communication system (MPI). Nevertheless, it suits well our goals: to use checkpointing for job-swapping as well, and to tolerate any number of failures. Thus, the recovery procedure is quite simple: all the processes have to roll back to the previously committed checkpoint.

The determination of the last committed checkpointed is obtained from the *status_file*. After restoring the local checkpoints on each process, they still have to perform some actions before re-starting the execution:

(i) the object location directory is constructed and updated through all the processes. In the case of a static distribution, this operation can be bypassed;

(ii) for every shared object defined as multi-copy the owner daemon resets its associated copy-set list;

(iii) each application process cleans its local private cache and updates the object location directory, if necessary.

Only after these steps are processes allowed to resume their computation. Cleaning the private cache of the processes during recovery does not introduce a visible overhead, and allows a simpler operation during the checkpoint operation since some potential *missing* messages exchanged on behalf of the DSM protocols do not need to be logged.

4. Comparison with other schemes

Some other coordinated checkpointing algorithms have been proposed in the literature. The algorithm presented in [Janakiraman94] extends the checkpoint/rollback operations only to the processes that have communicate directly or indirectly with the process initiator. That algorithm uses a 2-phase commit protocol during which all the processes participating in the checkpoint session have to suspend their computations, and all the messages in-transit have to be flushed to their destinations. Their algorithm waits for the completion of all on-going read/write operations before proceeding with the checkpointing protocol. Only after all the pending read/write operations have to be terminated the processors begin sending their checkpoints to stable storage. This may result in a higher checkpoint latency and performance overhead since they use a blocking strategy.

In [Cabillic95] is presented an implementation of consistent checkpointing in a DSM system. Their approach relies on the integration of global checkpoints with synchronization barriers of the application. The scheme was implemented on top of the Intel Paragon and several optimizations were included, like incremental, non-blocking and pre-flushing checkpointing techniques. They have shown that copy-on-write checkpointing can be an important optimization to reduce the checkpointing overhead. In the recovery operation of that scheme all the processes are forced to roll back to the last checkpoint, as in our case. The only limitation of this scheme is that it does not work with all applications: if there is no *barrier()* within an application the system is never able to checkpoint.

[Costa96] also presents a similar checkpointing scheme, that relies on the garbage collector mechanism to achieve a global consistent state of the system. It is based on a full-blocking checkpointing approach.

In [Kaashoek92] was presented a global consistent checkpointing mechanism for the Orca parallel language. It was very easy to implement because that DSM implementation is based on total-order broadcast communication. All the processes receive all broadcast messages in the same order to assure consistency of updates in replicated objects. The checkpointing messages are also broadcasted and inserted in the total order of messages. This ensures the consistency of the global checkpoint. Unfortunately, MPI does not have that characteristic.

[Choy95] presented a definition for consistent global states in sequentially consistent shared memory systems. They have also presented a lazy checkpoint protocol that assures global consistency. However, lazy checkpointing schemes may result in a high checkpoint latency, which is not desirable for job swapping purposes.

Other different recovery schemes not based on coordinated checkpointing were also presented in the literature. Some of them [Wu89][Janssens93] were based on communication-induced checkpointing: every process is allowed to take checkpoints independently but, before communicating with another one, they are forced to checkpoint in order to avoid rollback propagation and inconsistencies. Communication-induced checkpointing is sensitive to the frequency of inter-process communication or synchronization in the application. This may introduce a high performance overhead and an uncontrolled checkpoint frequency.

Another solution for recovery is based on independent checkpointing and message logging [Richard93]. However, we did not find this option very encouraging because DSM systems generate more messages than message passing programs. Even considering some possible optimizations [Suri95], message logging would incur in a significant additional performance and memory overhead.

A considerable set of proposals [Wilkinson93][Neves94][Stumm90B][Brown94] [Kermarrec95] are only able to tolerate single processor failures in the system. While this goal is meaningful for distributed systems, where we can expect that machine failures are uncorrelated, the same is not true for parallel machines where total or multiple failures are as likely as partial failures. We require our checkpointing mechanism to be able to tolerate any number of failures.

Although those different approaches could be interesting for other systems, we did not find them the most suitable for our system and we decided to adopt a coordinated checkpointing strategy.

5. Performance Results

In this section we present some results about the performance and memory overhead of our transparent checkpointing scheme. The results were collected in a distributed system composed of 4 Sun Sparc4 workstations connected by a 10 Mb/s Ethernet.

5.1 Parallel Applications

To conduct the evaluation of our algorithm we used the following six typical parallel applications[1]:

- **TSP**: solves the Traveling Salesman Problem using a branch-and-bound algorithm.
- **NQUEENS**: solves the placement problem of N-queens in a N-size chessboard.
- **SOR**: solves Laplace's equation on a regular grid using an iterative method.
- **GAUSS**: solves a system of linear equations using the method of Gauss-elimination.
- **ASP:** solves the All-pairs Shortest Paths problem using Floyd's algorithm.
- **NBODY:** this program simulates the evolution of a system of many bodies under the influence of gravitational forces.

[1] For lack of space we refer the interested reader to [Silva97] for more details about the applications.

5.2 Performance Overhead

We have made some experiments with the transparent checkpointing algorithm in a dedicated network of Sun Sparc workstations. Every processor has a local disk and access to a central file server through an Ethernet network. To take a local checkpoint of each process we used the *libckpt* tool in its fully transparent mode [Plank95]. None of the optimizations of that tool were used.

Two levels of stable storage were used: the first level used the local disks of the processors, while the second level used a central server that is accessible to all the processors through the NFS protocol. Writing checkpoints to the local disks is expected to be much faster than writing to a remote central disk. However, the first scheme of stable storage is only able to recover from transient processor failures. If a processor fails in a permanent way and is not able to restart, then its checkpoint can not be accessed by any other processor of the network and recovery becomes impossible. The central disk does not have this problem (assuming the disk itself is reliable).

Considering that stable storage is implemented on a central file server, Table 1 shows the time to commit and the corresponding overhead per checkpoint for all the applications. Usually, the time it takes to commit a global checkpoint is higher than the overhead produced. This is because the algorithm follows a non-blocking approach and the application processes do not need to wait for the completion of the protocol. If the algorithm were based on a blocking approach, the overhead per checkpoint would be roughly equal to the whole time it takes to commit. So, in the Table we can observe that, in the overall, the non-blocking nature of the algorithm allows some reduction in the checkpoint overhead.

Application	Size Chkp (Kbytes)	Time Commit (seconds)	Overhead Chkp (seconds)
TSP (18)	2197	152.078	16.330
NQUEENS (13)	2162	25.304	22.612
SOR (512)	5807	61.999	49.599
SOR (1024)	12311	122.591	116.547
SOR (2048)	37607	406.151	402.612
GAUSS (1024)	8500	130.763	128.832
GAUSS (2048)	35495	1136.530	1127.654
ASP (1024)	7905	99.192	98.985
ASP (2048)	20317	275.920	231.527
NBODY (4000)	5561	64.522	38.174

Table 1: Time to commit and overhead per checkpoint using the central disk.

The time to take a checkpoint depends basically on four factors: (i) the size of the checkpoint; (ii) the access time to stable storage; (iii) the synchronization structure of the application; (iv) and the granularity of the tasks.

Checkpoint operations are only performed inside DSMPI routines. This means that if an application is very asynchronous and coarse-grain it takes some time more to perform a global checkpoint when compared with a more synchronous application. These factors are important but, in practice, the dominant factor is actually the operation of writing the checkpoint files to

stable storage. Reducing the size of the checkpoints is a promising solution to attenuate the performance overhead. Another way is to use a stable storage with faster access.

Table 2 shows the overhead per checkpoint considering the two different levels of stable storage. As can be seen, the difference between the figures is considerable: in some cases it is more than one order of magnitude. Using the Ethernet and the NFS central file server is really a bottleneck for the checkpointing operation. Nevertheless, it ensures a global accessible stable storage device where checkpoints can be made available even in the occurrence of a permanent failure of some processor.

Application	Size Chkp (Kbytes)	Overhead Chkp (sec) (local)	Overhead Chkp (sec) (central)
TSP (18)	2197	3.889	16.330
NQUEENS (13)	2162	4.457	22.612
SOR (512)	5807	1.138	49.599
SOR (1024)	12311	2.255	116.547
SOR (2048)	37607	7.780	402.612
GAUSS (1024)	8500	1.186	128.832
GAUSS (2048)	35495	4.284	1127.654
ASP (1024)	7905	1.496	98.985
ASP (2048)	20317	3.483	231.527
NBODY (4000)	5561	1.067	38.174

Table 2: Overhead per checkpoint (local vs central disk).

Table 3 shows the difference in the overall performance overhead considering the two levels of stable storage and different intervals between checkpoints. We present the results for the SOR application, that was executed for an average time of 4 hours.

Application	Interval between chkp	Overhead Chkp (%) (local)	Overhead Chkp (%) (central)
SOR (512)	2 min	0.940	40.980
	5 min	0.371	16.188
	10 min	0.181	7.922
	20 min	0.086	3.788
SOR (1024)	2 min	1.863	96.313
	5 min	0.736	38.039
	10 min	0.360	18.615
	20 min	0.172	8.902
SOR (2048)	2 min	6.429	332.714
	5 min	2.539	131.408
	10 min	1.242	64.306
	20 min	0.594	30.755

Table 3: Total performance overhead (local vs. central disk).

The average overhead for checkpointing can be tuned by changing the checkpoint interval. In Table 3 we can see that the maximum overhead observed when using the local disk was 6.4%. The corresponding overhead with the central file server was up to 332%. This shows that if we consider a distributed stable storage scheme the performance can become interesting.

Nevertheless, two minutes is a very conservative interval between checkpoints. Long-running applications do not need to be checkpointed so often and 20 minutes is a more acceptable interval. For this case, the performance overhead when using the local disks was 0.6%, which is a very small value. The same interval with the central disk as stable storage presented an overhead of 30.7 %.

An interesting strategy would be the integration of both stable storage levels: that is, the application is checkpointed periodically to the central server, and in the meantime it can also be checkpointed to the local disks of the processors. If the application fails due to a transient perturbation and all the processors are able to restart, then they can recover from the checkpoints saved in each local disk (if this one correspond to the last committed checkpoint). If some of the processors is affected by a permanent outage then the application can be restarted from the last checkpoint located in the central disk.

A possible solution to make the distributed stable storage scheme resilient to a permanent failure of one processor, is to implement a sort of logical ring where each processor should copy its local checkpoint file to the next processor's disk. This can be done after the global checkpoint being committed and in a concurrent way. This lazy update scheme would not introduce any delay in the commit operation: only some additional traffic in the network, that can be regulated if we use a token-based policy and perform each remote checkpoint file copy in a sequential way. Obviously, if we want to tolerate *n* permanent processor failures we have to replicate each checkpoint file by *n+1* locals disks of the network.

We measured the performance overhead when using both levels of stable storage and some of the results are presented in Figure 6. For each checkpoint in the central disk we performed *K* checkpoints to the local disks. The factor *K* was changed from 0 up to 10. Figure 6 shows the overhead reduction for the SOR application with 512x512 grid points.

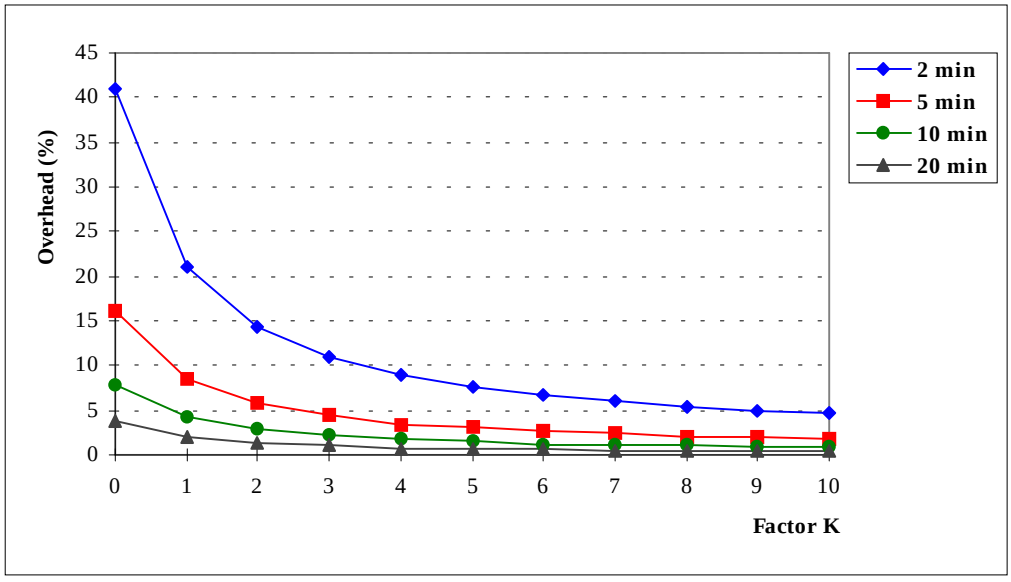

Figure 6: Two-level stable storage (SOR 512).

For instance, if the user wants an overhead lower than 5% then the factor *K* should be 9, 3, 1 and 0 when using a checkpoint interval of 2, 5, 10 and 20 minutes, respectively.

If a permanent failure occurs in one of the processors of the system then in the worst case the application will loose approximately 20 minutes of computation in any of the four previous cases. The advantage still goes for an interval of 2 minutes and *K* equal to 9, since in the occurrence of a transient failure it will lose less computation.

Figure 7 shows the corresponding values for the SOR application with 1024x1024 points.

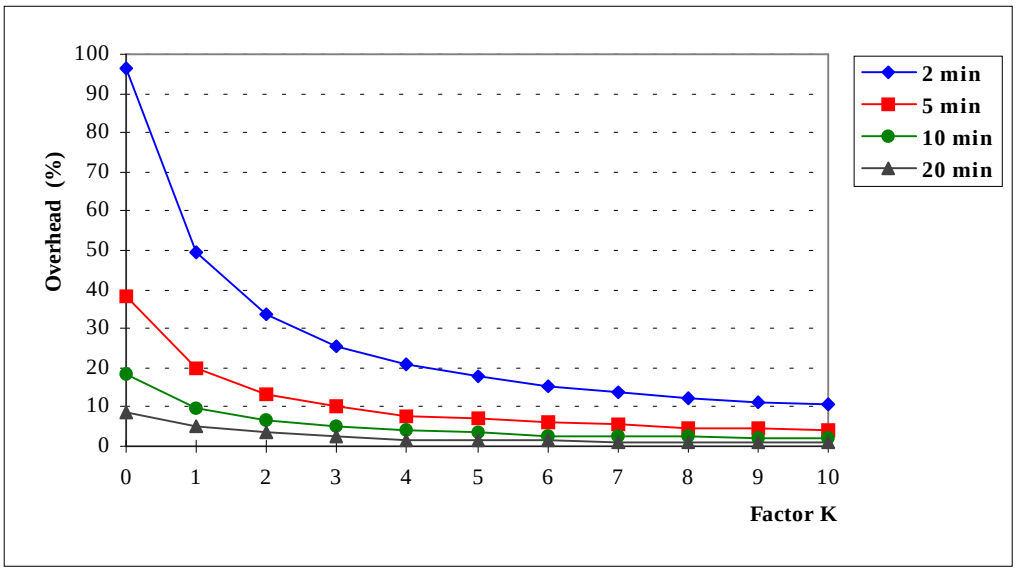

Figure 7: Two-level stable storage (SOR 1024).

The same analysis can be done, considering a watermark of 10% for the performance overhead: when checkpointing the application with an interval of 2, 5, 10 and 20 minutes the factor K should be 11, 4, 1 and 0, respectively. If the user requires an overhead lower than 5% then K should be 8, 4 and 1, with an interval of 5,10 and 20 minutes, respectively.

6. CONCLUSIONS

As far as we know, this is the first implementation of a non-blocking coordinated algorithm in a real DSM system. DSMPI provides different protocols and models of consistency, and our algorithm works with all of them. The checkpointing scheme is general-purpose and can be adapted to other DSM systems that use any protocol of replication or model of consistency.

Some results were taken considering a distributed stable storage scheme and we have observed a maximum overhead of 6% for an interval between checkpoints of 2 minutes. With a checkpoint interval of 20 minutes the performance overhead was 0.6%. The same interval with the stable storage implemented in a central NFS-file server presented an overhead of 30.7 %.

The algorithm herein presented offers an interesting level of portability and efficiency. Though, we plan to enhance some of the features of DSMPI in the next release that will be implemented on MPI-2. We look forward for a thread-safe version of MPI in order to re-

design the DSMPI daemons and implement some of the optimization techniques proposed in [Cabillic95]. We hope that this line of research would give some contribution to a standard and flexible checkpointing tool that can be used in real production codes.

Acknowledgments

The work herein presented was conducted when the first author was a visitor at EPCC (Edinburgh Parallel Computing Centre). The visit was made possible due to the TRACS programme. The first author was supported by JNICT on behalf of the "Programa Ciência" (BD-2083-92-IA).

7. REFERENCES

[Brown94] L.Brown, J.Wu. "Dynamic Snooping in a Fault-Tolerant Distributed Shared Memory", Proc.14th Int. Conf. on Distributed Computing Systems, pp. 218-226, 1994

[Cabillic95] G.Cabillic, G.Muller, I.Puaut. "The Performance of Consistent Checkpointing in Distributed Shared Memory Systems", Proc. 14th Symposium on Reliable Distributed Systems, SRDS-14, 1995

[Carter91] J.Carter, J.Bennet, W.Zwaenepoel. "Implementation and Performance of Munin", Proc. 13th ACM Symposium on Operating Systems Principles, pp. 152-164, 1991

[Carter93] J.B.Carter, A.Cox, S.Dwarkadas, E.N.Elnozahi, D.Johnson, P.Keleher, S.Rodrigues, W.Yu, W.Zwaenepoel. "Network Multicomputer Using Recoverable Distributed Shared Memory", Proc. COMPCON'93, 1993

[Choy95] M.Choy, H.Leong, M.H.Wong. "On Distributed Object Checkpointing and Recovery", Proc. of the ACM Principles of Distributed Computing, PODC95, 1995

[Costa96] M. Costa, P.Guedes, M.Sequeira, N.Neves, M.Castro "Lightweight Logging for Lazy Release Consistency Consistent Distributed Shared Memory", Proc. 2nd Usenix Symposium on Operating Systems Design and Implementation, pp 59-73, Seattle, October 1996

[Elnozahi92] E.N.Elnozahi, D.B.Johnson, W.Zwaenepoel. "The Performance of Consistent Checkpointing", Proc. 11th Symp. on Reliable Distributed Systems, pp. 39-47, 1992

[Eskicioglu95] M.R. Eskicioglu. "A Comprehensive Bibliography of Distributed Shared Memory", Technical Report TR-95-01, Univ. of New Orleans, May 1995

[Janakiraman94] G.Janakiramam, Y.Tamir. "Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers", Proceedings of the 13th Symposium on Reliable Distributed Systems, SRDS-13, pp. 41-51, October 1994

[Janssens93] B.Janssens, W.K.Fuchs. "Relaxing Consistency in Recoverable Distributed Shared Memory", Proceedings 23rd Fault-Tolerant Computing Symposium, FTCS-23, pp. 155-163, June 1993

[Johnson95] K.L.Johnson, M.F.Kaashoek, D.Wallach. "CRL: High-Performance All-Software Distributed Shared Memory", Proc. of the 15th Symposium on Operating Systems Principles, 1995

[Kaashoek92] M.F.Kaashoek, R.Michiels, H.Bal, A.Tanenbaum. "Transparent Fault-Tolerance in Parallel Orca Programs", Proc. Symposium on Experiences with Distributed and Multiprocessor Systems III, pp. 297-312, 1992

[Keleher94] P.Keleher, A.Cox, S.Dwarkadas, W.Zwaenepoel. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Proc. Winter 94 USENIX Conference, pp. 115-131, January 94

[Kermarrec95] A.M.Kermarrek, G.Cabillic, A.Gefflaut, C.Morin, I.Puaut. "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", Proc. 25th Fault-Tolerant Computing Symposium, FTCS-25, pp. 289-298, July 1995

[Kranz93] D.Kranz, K.Johnson, A.Agarwal. "Integrating Message-Passing and Shared-Memory: Early Experience", Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 54-63, May 1993

[Lenoski90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Henessy. "The Directory-based Cache Coherence Protocol for the DASH Multiprocessor", Proc. 17th Annual International Symposium on Computer Architecture, pp. 148-159, 1990

[Li89] K.Li, P.Hudak. "Memory Coherence in Shared Virtual Memory Systems", ACM Transactions on Computer Systems, Vol. 7, No. 4, pp. 321-359, November 1989

[Long95] D.Long, A.Muir, R.Golding. "A Longitudinal Survey of Internet Host Reliability", Proc. 14th Symposium on Reliable Distributed Systems, pp. 2-9, September 1995.

[MPI94] MPI Forum. MPI Forum. "A Message Passing Interface Standard". May 1994, Available on `netlib`.

[Neves94] N.Neves, M.Castro, P.Guedes. "A Checkpoint Protocol for an Entry Consistent Shared Memory System", Proc. of the 13th ACM Symposium on Principles of Distributed Computing, 1994

[Nitzberg91] B.Nitzberg, V.Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms", IEEE Computer, Vol. 24 (8), pp. 52-60, 1991

[ORNL95] data available in: `http://www.ccs.ornl.gov/`

[Plank94] J.Plank, K.Li. "Performance Results of ickp - A Consistent Checkpointer on the iPSC/860", Proc. of the Scalable High-Performance Computing Conference, Knoxville USA, pp. 686-693, 1994

[Plank95] J.Plank, M.Beck, G.Kingsley, K.Li. "Libckpt: Transparent Checkpointing Under Unix", Usenix Winter 1995 Technical Conference, January 1995

[Raina92] S. Raina. "Virtual Shared Memory: A Survey of Techniques and Systems", Technical Report CSTR-92-36, University of Bristol, December 1992

[Richard93] G.Richard III, M.Singhal. "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory", Proceedings 12th Symposium on Reliable Distributed Systems, SRDS-12, pp. 58-67, October 1993

[Silva92] L.M. Silva, J.G.Silva. "Global Checkpoints for Distributed Programs", Proc. 11th Symp. on Reliable Distributed Systems, pp. 155-162, Houston USA, 1992

[Silva97] L.M.Silva, S.Chapple, J.G.Silva. "Implementation and Performance of DSMPI", *Scientific Programming Journal*, Vol.6, No. 2, April 1997, John Wiley & Sons.

[Stumm90A] M.Stumm, S.Zhou. "Algorithms Implementing Distributed Shared Memory", IEEE Computer, Vol. 23 (5), pp. 54-64, May 1990

[Stumm90B] M.Stumm, S.Zhou. "Fault-Tolerant Distributed Shared Memory Algorithms", Proc. 2nd IEEE Symp. on Parallel and Distributed Computing, pp. 719-724, Dec. 1990

[Suri95] G.Suri, B.Janssens, W.K.Fuchs. "Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory", Proceedings 25th Fault-Tolerant Computing Symposium, FTCS-25, pp. 279-288, June 1995

[Wilkinson93] T.J.Wilkinson. "Implementing Fault-Tolerance in a 64-bit Distributed Operating System", PhD Thesis City University, July 1993

[Wu89] K.L.Wu, W.K.Fuchs. "Recoverable Distributed Shared Virtual Memory: Memory Coherence and Storage Structures", Proceedings 19th Fault-Tolerant Computing Symposium, FTCS-19, pp 520-527, 1989

Short Biographies

Luís M. Silva graduated in Computer Engineering at the University of Coimbra in 1990, received his MSc in Computer Enginnering from the Technical University of Lisbon in 1993 and the PhD in Computer Science from the University of Coimbra in 1997. Since May 1997 he is an Assistant Professor of Computer Engineering at the University of Coimbra, Portugal. His research interests include parallel processing, high-performance computing, distributed algorithms and fault-tolerance.

João Gabriel Silva received a degree in Electrotechnical Engineering and a PhD in Informatics from the University of Coimbra, Portugal, in 1980 and 1988 respectively. Since 1988 he has been an Assistant Professor of Computer Engineering at the same University. He also coordinates the research group on Dependable Systems, where he investigates experimental techniques for the validation of fault-tolerant systems by fault injection, behaviour based error detection, and fault-tolerant parallel programing.

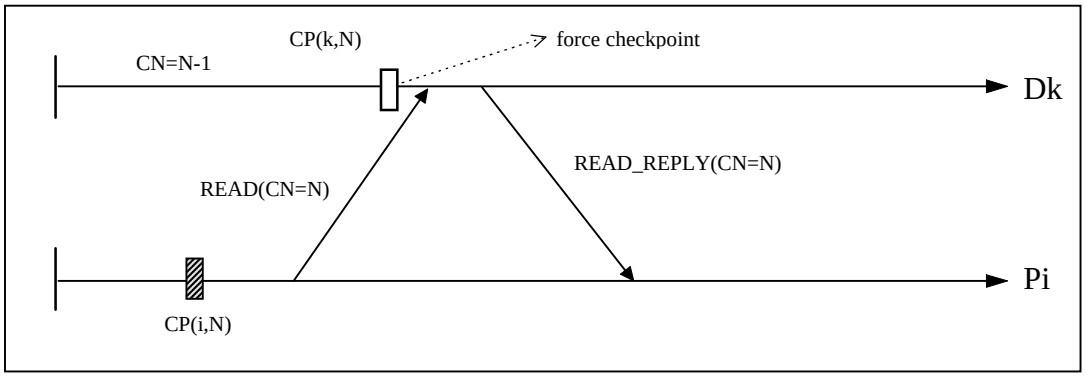

Figure 1: Forcing a checkpoint in the DSM daemon.

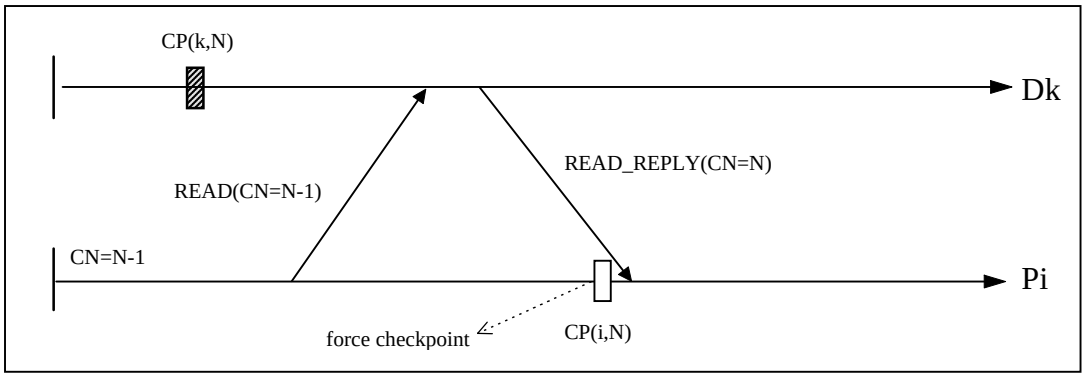

Figure 2: Forcing a checkpoint in the application process.

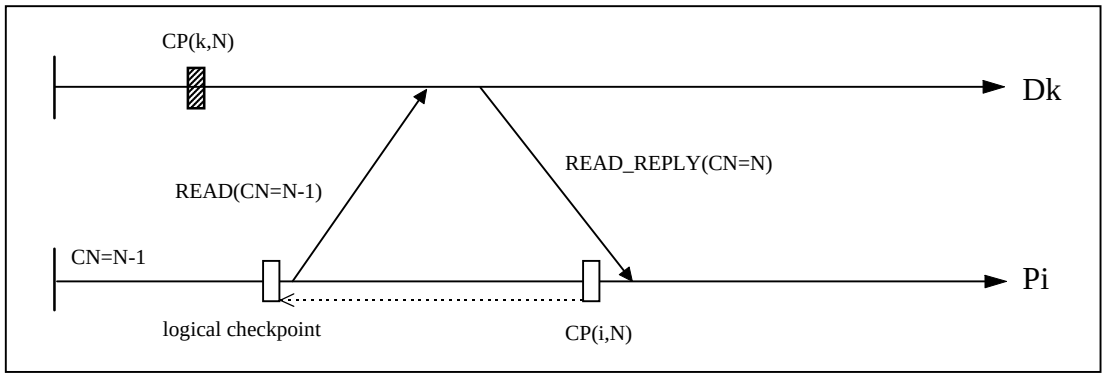

Figure 3: The notion of a logical checkpoint.

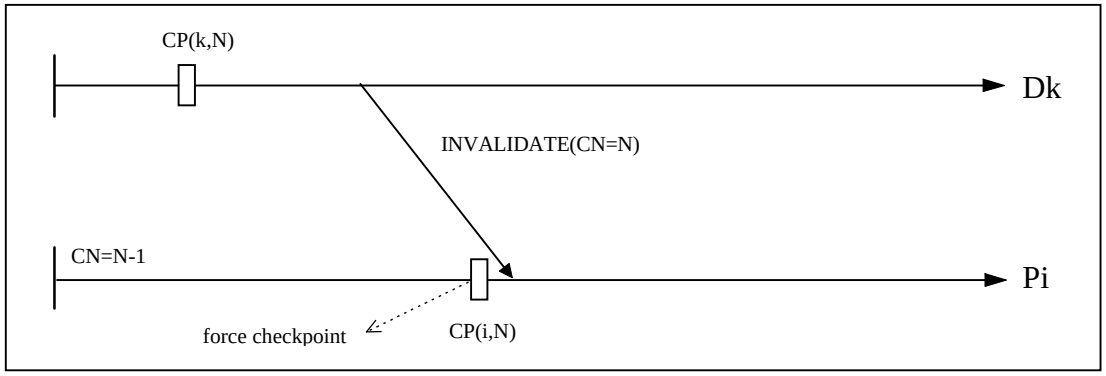

Figure 4: Potential *orphan* message.

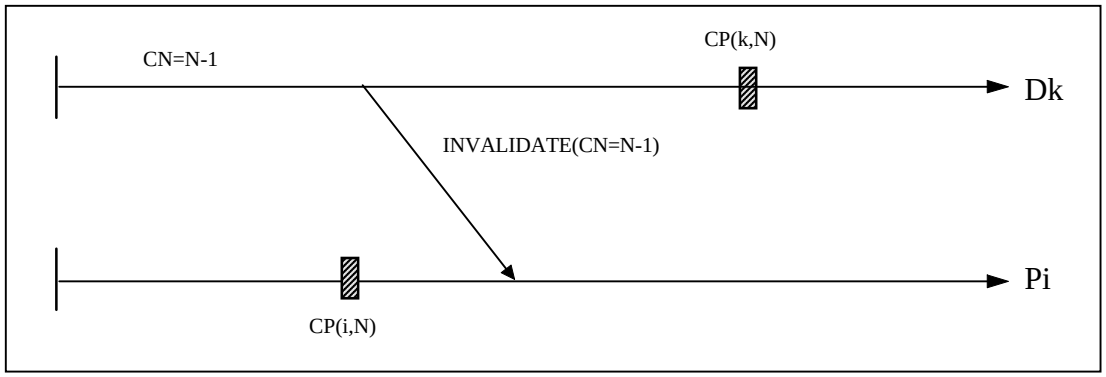

Figure 5: Example of a *missing* message.

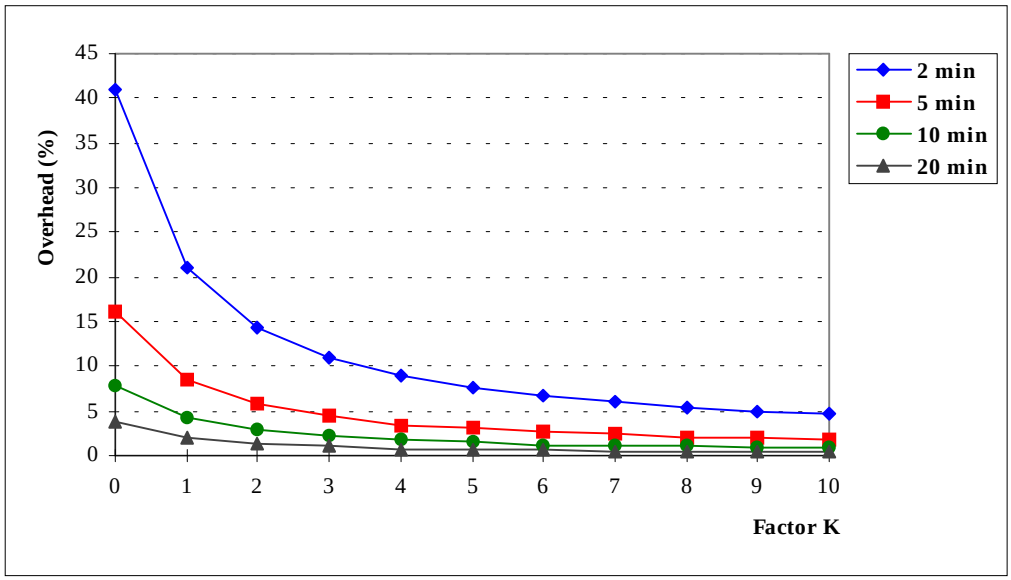

Figure 6: Two-level stable storage (SOR 512).

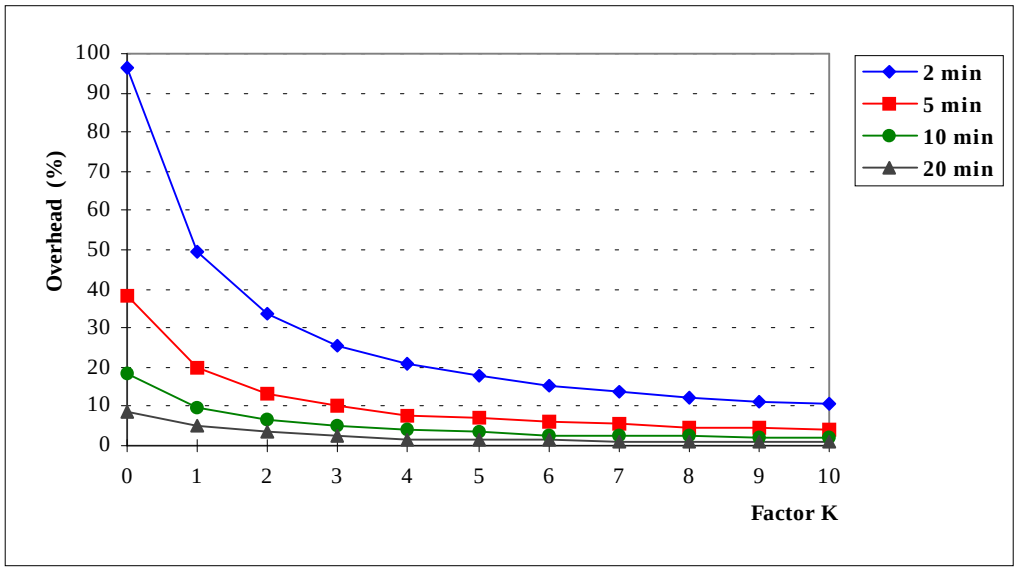

Figure 7: Two-level stable storage (SOR 1024).