Fast and Low Cost Recovery Techniques for Distributed Shared Memory

Jerzy Brzeziński  Michał Szychowiak
Institute of Computing Science
Poznań University of Technology
Piotrowo 3a, 60-965 Poznań, POLAND
phone: +48 61 665 28 09, fax: +48 61 877 15 25

Abstract

The goal of this paper is to indicate how the mechanisms already available in standard Distributed Shared Memory (DSM) systems can be efficiently used to reduce the cost of fault-tolerance. It can be achieved by using DSM replication mechanism for recovery and integration of both recovery and memory coherence protocols. We analyze recently developed techniques of recovery for DSM systems which offer high availability of shared data with relatively low overhead and fast recovery of the memory space in case of failures of the system nodes. Two important memory consistency models are investigated: atomic consistency (the strongest consistency model) and release consistency (the most popular of relaxed consistency models). A new technique of boundary-restricted checkpoint replication is also proposed.

Keywords: fault tolerance, distributed shared memory, replication, rollback recovery, checkpointing, logging

1. Introduction

Distributed Shared Memory (DSM) systems offer a single address space shared by several nodes connected via a communication network, making development of parallel programs in distributed environment easier. From user’s point of view, DSM is similar to a traditional virtual memory: the whole communication and synchronization can be done via memory, using no message passing. As a result, DSM systems offer, at least potentially, important advantages: attractive for users shared memory programming model, scalability allowing practically unlimited and incremental growth, large size of shared memory that may consist of local memories available at all nodes, possibility to run programs written for shared memory multiprocessors.

Although DSM offers remarkable advantages, a practical use of them requires improved availability of shared data and reliability of the data access. This can be ensured by fast recovery of the system in case of failures that have caused a loss of some shared data. The DSM recovery problem consists in restoring the values of lost data in such a way, that the whole memory remains in a consistent state, according to the consistency model that has been assumed. A large range of recovery protocols for general distributed systems have been explored in literature. Two most important techniques used for ensuring recoverability of the system are message logging (event logging) and checkpointing. Several DSM systems developed until now, offer recovery capability, but mostly at a very high cost, resulting in low efficiency of the system. This high cost is mainly due to direct adaptation of recovery procedures from general message passing systems. Although this adaptation is logically correct, it is not well suited for most DSM consistency models, because both logging and checkpointing require frequent accesses to a stable secondary storage in order to save the log or a current state of the computation, and therefore they suffer from significant overhead when imposed on DSM systems. This motivates investigations for new recovery protocols dedicated for DSM.

This paper presents two different approaches to overhead reduction of DSM recovery protocols. The first is the use of message semantics enabling to limit necessary logging only to a subset of all messages (lower space overhead) and perform the logging only at carefully selected moments (lower time overhead of the access to stable storage). The second approach is to extend the coherence protocol in order to reuse the native replication mechanism for storing checkpoints (thus resigning from a stable storage at all).

This paper is organized as follows. Section 2 presents a DSM recovery problem formulation. Section 3 gives a background of the recovery mechanisms in DSM. Sections 4 and 5 describe some techniques intended to

---

1 A version of this paper has appeared in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA’02, Las Vegas, Nevada, June, 2002.
reduce the overhead of the recovery in logging and checkpointing approaches, respectively. Concluding remarks are given in Section 6.

2. Problem definition

DSM system is a distributed system composed of a finite set of sequential processes $P_1, P_2, ..., P_n$ that can access a finite set of shared objects. We distinguish two different general operations on a DSM object $x$ issued from process $P_i$; read access, denoted $r(x)$, and write access – $w(x)$. Any write access results in a new value of $x$. Typically, a replication mechanism is aimed to increase the efficiency of DSM object access by allowing each process to access locally a replica of the object. However, concurrent access to different replicas of the same shared object requires consistency management. Consistency protocol synchronizes each access to replicas, accordingly to the DSM consistency criterion. This protocol performs all communication necessary for the interprocess synchronization via message-passing.

Some of the system components may fail at arbitrary moments, resulting in process failures during the execution. We assume here the fail-stop model of failures [17], i.e. $P_i$ fails only by crashing, and after eventual recovery $P_i$ is reexecuted. When process $P_i$ fails, the whole content of the volatile memory used by $P_i$ is lost, resulting in temporal unavailability of lost object replicas. If any of them was at that moment a unique replica of a given object, the DSM system will not be able to serve any further read nor write access issued to this object, until it will be restored. However, restoring some predefined value of a lost object may violate the consistency of the whole memory space. To avoid potential inconsistency, the restored object value should reflect the history of operations performed on the object by the coherence protocol. The restoration of consistent values of all lost objects is the role of a DSM recovery protocol.

One solution for the recovery problem is to maintain a list of all operations performed on DSM objects (a log) at each DSM node, in order to be able to reexecute them if the node fails and restarts after the repair. Instead of remembering the entire history of all operations, a recovery protocol can ensure, that consistent state of the memory is stored in a safe place in the form of backup copies of all existing DSM objects. These copies, called checkpoints, must be saved in a stable (nonvolatile) storage able to survive failures. So, on the recovery, consistent values of all objects can be quickly fetched from their checkpoints.

The cost of recovery

The role of the recovery, is to provide the ability of the DSM system to survive a failure (transient or permanent) of a DSM node, possibly by rolling back the memory state to a checkpointed state. There are three important factors that must be dealt with. The first one is the cost of recovery protocol operations in failure-free execution, expressed for instance in the number of DSM nodes involved in checkpointing required for a given degree of the failure resilience. Alternatively, this cost can be recognized as the communication overhead imposed on the system by the checkpointing, measured in the number of messages exchanged during a single checkpoint operation, a time overhead of the access to a stable secondary storage, or a space overhead resulting from logging information required to keep track of the recoverable operations in log based protocols. The second factor is the scope of a failure, i.e. the rollback propagation, expressed as the number of nodes required to recover in case of a single failure. The last factor is the time overhead of the recovery, i.e. the time the system takes to recover from a single failure.

3. Background

Three main approaches to DSM recovery were proposed in the literature.

Chronologically the first consists in directly adapting checkpointing techniques, developed formerly for message-passing distributed systems, for a DSM environment ([5],[10],[11],[17],[22]). In message-passing systems, checkpointing is intended to store a state of the computation (local states of processes) in order to enable state restoration in case of a further rollback due to some processes failure. In DSM, as each write operation performed by a process on a DSM object, changes the memory state (and process state), the process is required to save a checkpoint of the modified object accordingly to a DSM recovery protocol. The most often used solution, enforces a checkpoint before allowing other processes to read the modified value (dirty page flush). This approach extends message-passing checkpointing algorithms with special data structures, conforming to DSM memory unit, that are used to store checkpoints of DSM objects (e.g. twin pages [22]), and with supplementary operations performed by DSM controllers (i.e. owners and managers of shared object) ensuring correct maintenance of those structures in spite of failures. In this approach, the most often assumed consistency model of the DSM is sequential or release consistency.

The second approach uses message logging, also originally developed for message-passing. Since in DSM systems the coherence protocol maintains the memory
consistency by implicitly exchanging messages between nodes, the memory state of each process depends on the communication history pattern. In communication-induced recovery approach, coherency protocol messages are logged in order to track dependencies between replicas’ values ([7], [16]). In this case, the dependency tracking algorithms for message-passing can be directly used, however, they are not optimal for DSM, because many messages of a coherency protocol do not change the memory state (e.g. read requests issued to the owner, owner localization requests and answers processed by the manager, etc.). This motivates the adaptation of dependency tracking for a particular coherency protocol [12]. As in the first approach, the assumed consistency model of the DSM is usually sequential or release consistency.

Both of the two approaches involve an external stable storage for saving checkpoints or logs. The main drawback of it is that the access to the stable storage implies significant overhead and as a result limits the DSM efficiency.

The third approach makes use of object replication. The replication can also help to restore the state of a DSM node on recovery. Indeed, the recovery protocol can fetch the values of all necessary objects from available replicas on other nodes (preserving consistency of the whole DSM memory). In this approach, checkpoints are stored not on a stable storage but in the memory itself, and replicated accordingly to the required level of the failure resilience. On checkpointing, the recovery protocol backups a checkpoint at distinct nodes [13]. In most of the implementations of this approach, for each checkpointed object only one backup is stored, which offers only 1-resilience. However, in case of f-node failures, at least f+1 replicas are necessary to enable recovery (f-resilience).

In the best possible solution, the recovery demands no additional replication as the replicas maintained by the coherency protocols are sufficient to ensure recovery. This can be illustrated with a full-replication system using update-all protocol, in which every DSM node holds a replica of each shared object, atomically updated at every modification. Although this solution is very simple and is n-resilient, it suffers from a very large update cost in the coherence protocol. Therefore, most coherence protocols use invalidate schema which in general, unfortunately, does not guarantee availability of all shared object in case of even a single node or communication failure (notably network partitioning). Hence, the recovery protocol has to keep some additional backup replicas (we will call this checkpoint replication). The goal of the solution to the recovery problem in this approach is to minimize the cost of the checkpoint replication.

In the next two Sections we show how to efficiently use operation (and message) semantics to reduce the cost of access logging (here, as an example we decided to use a relaxed consistency, because of its clear message semantics) and how to integrate the coherence and recovery protocols to minimize the cost of checkpointing and recovery operations.

4. Logging recovery in relaxed consistency models

Lazy Release Consistency (LRC) is currently the most popular relaxed consistency model, mainly because of its simplicity and popularity of DSM systems like TreadMarks [1] in scientific research centers. LRC model enforces ordering of memory accesses issued from different DSM nodes within periods synchronized by pairs of acquire (lock) and release (unlock) operations performed on synchronization variables. It allows accesses issued between synchronizations to be seen in any order. Synchronization guarantees that all accesses performed by one process before releasing a lock are ordered before any access made by another process after it calls a subsequent lock acquire. As long as a programmer ensures no data race in a program, LRC offers sequentially consistent ordering of all DSM accesses. LRC attempts to reduce the interprocess communication of coherence protocols, delaying propagation of modifications made to data protected by a synchronization variable until this variable is acquired on a different node. At that moment, the last releaser sends a set of write notices (invalidations, INV) to the acquirer. When the acquirer accesses an invalidated page, a coherency protocol fetches an up-to-date copy of that page (or a diff record) from the process which has modified it.

Suri, Janssens, and Fuchs introduced in [19] an optimistic logging protocol for LRC. Their approach called Shared-Access Tracking (SAT), is based on the observation, that actual dependencies are due only to accesses to synchronization variables. Therefore, it is sufficient to log INV messages received during an acquire operation, and update (UPD) messages received on access misses. Those data are stored in volatile log which is then flushed to stable storage whenever an UPD message is sent to another process. On recovery, when the log is replayed, the correct invalidations are applied at each acquire, and the access misses bring the correct update from the log, ensuring deterministic reexecution until the state preceding the failure. Additionally, this protocol periodically invokes independent checkpointing, in order to limit log size and speed up the recovery.

An accessed memory page can be logged either at a process which access it (the reader) or at a process that
has produced it (the writer). If the reader logs the page, the log must be stable since it has to survive reader’s node failure. On the other hand, in writer-based logging, a volatile log can be used, since after the reader’s failure the missing page can be reclaimed from the writer, and in case of the writer’s own failure, it can regenerate the same contents of the page when correctly re-executing on recovery. Moreover, since data written by a process are usually read by many processes, it is more efficient to store it once on the writer’s side, instead of logging it at each reader separately.

A writer-based logging was proposed by Costa et al. in [6]. Their protocol logs less data than SAT and does not require communication-induced flushing of the volatile log. The protocol works in two levels. First level consists in volatile logging integrated with RLC protocol, and periodic independent checkpointing. Whenever a process sends a relevant information (as memory page update) to another, it logs a vector clock of this event. Since the LRC protocol already keeps most of those relevant information in its data structures (as diff list), the recovery protocol does not need to log it additionally. Instead, it logs only the time at which this information is transferred. It significantly reduces space overhead. Unfortunately, this approach can handle at most one failure at a time (no simultaneous failures of both reader and writer processes can be handled). In the second level, the protocol offers a coordinated checkpointing integrated with global garbage collection of LRC. Garbage collection is performed occasionally, in order to reduce the size of diff lists maintained by the coherence protocol on each DSM node. Memory consistency in the garbage collection is preserved with the use of barriers which synchronize all processes. Since coordinated checkpoints permit recovery from multiple node failures, the described two level approach supports more probable single failures with low overhead and less probable multiple failures with higher but less frequent coordination overhead (this is an adaptation of the approach proposed by Vaidya in [21]).

Another logging protocol with the possibility to recover from multiple failures was proposed by Park and Yeom in [16]. Here, Reduced Stable Logging (RSL) makes a volatile log of access timestamps only at synchronization points, since the vector clock is incremented on release only, and on acquire it is updated accordingly to modifications performed by former releasers. RSL overcomes the problem of 1-failure reliability by additionally tracking changes of the memory state, limited (in comparison with SAT) to logging only INV messages at each acquire. This is a consequence of the fact, that each memory update is actually dependent on the contents of the list of dirty pages previously received with INV message. Hence, it is adequate to log only the dirty-page lists, to enable deterministic reexecution on recovery. Stable logging is performed when a new dependency relation is established (i.e. another process requests a page modified since the last stable logging). Also diffs discarded by the garbage collector are stably logged.

In comparison to RSL, Lazy Logging (LL) technique, proposed by Kongmunvattana and Tzeng in [14], attempts to limit logging of dirty-page list received on acquire to only those pages that were read later, since information about modified pages not accessed by other processes will not serve for any possible recovery. Obviously, in the worst case, the space complexity of both LL and RSL protocols is the same. LL technique permits also for further reduction of stable logging frequency, by allowing to flush the volatile log not at each synchronization point. Instead, optimistic flushing is performed every \( f \) synchronizations (\( f \) is called flushing distance). Upon a failure, a volatile log of every failed process will be lost, however, unlike in optimistic logging in message passing, no surviving process will need to rollback. It is a consequence of the fact, that there are only three types of coherency-related messages whose arrivals cause nondeterministic events in LL: lock grant messages, UPD messages and UPD requests. None of them, in fact, create a rollback dependency between the sender and the receiver. The two first are sent only in response to explicit requests, and therefore they will not be recreated during recovery. On the other hand, receptions of UPD requests do not change the memory state, thus it is unnecessary to rollback from such reception events.

5. Checkpoint replication in strict consistency models

Now we focus our attention on a model of strict consistency – atomic consistency. This model ensures that every read access to a shared object will operate on the most recent (in real time) value of this object. For formal definition of the consistency model, the reader is referred to [9] or [15]. We believe this model is well suited for workgroup platforms that use object replication and it has been commonly adapted for such platforms. In this model, we investigate the coherence protocol proposed by Li and Hudak [15]. This protocol uses a data-invalidate schema. The protocol guarantees atomic consistency by ensuring simultaneous existence of several read-only replicas (i.e. in read-only state – RO) of a given object when no write access to this object is being performed, or only one read-write replica (i.e. in writable state – WR) with no RO replicas during a write operation. Operations of the protocol are described in four subsequent points.
1) If process $P_i$ wishes to gain a read access to object $x$ unavailable locally, the protocol issues a read request $r_i(x)$ to the current owner of $x$, say $P_o$, holding the master replica and the copyset CS of $x$. If the master replica is in state WR, then CS contains only $P_i$ and there is no other replica of $x$ in the system. In that case, on request from $P_o$, the WR replica becomes RO, preventing from further write operations. The owner adds the identity of $P_i$ to CS and sends back a new RO replica of $x$ to $P_i$.

2) If $P_i$ wishes to perform a write access to object $x$ unavailable locally, the protocol issues a write request $w_i(x)$ to the object owner. On this request, the owner invalidates all the replicas in CS (invalidated replicas are marked INV and become unavailable locally for processes), clears CS and then sends back a new replica of $x$ along with the ownership of $x$ to $P_i$. When arrived at $P_i$, the replica is set to WR state and CS includes only $P_i$.

3) If $P_i$ wants to access locally available object $x$ for reading, the read operation is performed instantaneously regardless of a current state of the replica.

4) Before accessing locally available object $x$ for writing, $P_i$ has to become the owner of $x$, i.e. when it holds a RO replica of $x$, the protocol invalidates all other replicas and then transfers the ownership to $P_i$ (as in 2); when $P_i$ holds a WR replica, $P_i$ is actually the owner, so the write operation is performed instantaneously.

We show now how to extend this protocol to offer a low cost checkpointing of shared objects and a high availability of checkpoints in spite of a failure of multiple nodes. Checkpointing will be integrated with normal work of the coherence protocol, so the amount of additional synchronization is minimized.

Indeed, it is possible to store checkpoints as special-purpose replicas, called *checkpoint replicas*, and maintain additionally a copyset of checkpoint replicas CSR. The content of CSR can change accordingly to the further access requests or failure pattern, or any load balancing mechanisms. Obviously, the size of CSR influences not only the degree of checkpoint availability (i.e. failure resilience of the system) but also the cost of checkpointing (since an update operation is much more costly than invalidate one). A possible solution is to always keep the number $nr$ of checkpoint replicas between boundaries $nr_{\text{min}}$ and $nr_{\text{max}}$ (i.e. at any time $nr_{\text{min}} \leq nr \leq nr_{\text{max}}$ holds). The number $nr_{\text{min}}$ represents the minimum number of checkpoint replicas necessary to reach desired failure resilience. On the other hand, the number $nr_{\text{max}}$ represents the maximum number of checkpoint replicas allowed to limit the protocol overhead. As a result, this will ensure the required high level of availability of checkpoints at minimal cost. Furthermore, boundaries can be allowed to change dynamically at run-time [20].

Another important advantage of the proposed scheme is the possibility to reuse the checkpoint replicas as normal RO replicas until next invalidation of the checkpointed object (i.e. when the value of the object $x$ remains unchanged). In this moment, the value of a checkpoint replica reflects the state of $x$ at the moment of the last checkpoint. Therefore, on any read access to $x$, a checkpoint replica is used as if it was in RO state.

In addition to the above described protocol specification, we introduce two complementary rules.

First, we define a correlation between CSR and CS of a given object $x$. When the cardinality of CS is not less than $nr_{\text{min}}$, we require CSR $\subseteq$ CS. This way, after each checkpoint operation, we offer a possibility of instantaneous read access for $x$ at all processes in CSR holding a RO replica of $x$ till now, with no additional cost. As they are now holding ROC replicas updated on checkpointing, no supplementary operation will be required on read accesses issued from these processes. For the same reason, when the cardinality of CS is less than $nr_{\text{min}}$ we require CSR to be a superset of CS (i.e. $CS \subseteq CSR$) – this permits to reduce the cost of CSR update if any subsequent read accesses will be issued from any of processes formerly holding an RO replica.

Second rule states that checkpointing is performed on ownership migration only. When the extended recovery protocol needs to transfer the ownership of $x$ from $P_i$ to $P_j$, first it updates all checkpoint replicas of $x$ (actually, this operation takes a new checkpoint for $x$). In order to guarantee the atomicity of the checkpointing operation, a 2PC (Two Phase Commitment [8]) or nonblocking 3PC (Three Phase Commitment [18]) schema of the update can be used, with the current owner as a coordinator and all other nodes in CSR as cohorts. After all checkpoint replicas have been updated (i.e. checkpoint has been taken successfully), the ownership of $x$ is transferred to $P_j$, along with the object itself, if necessary (i.e. when $P_j$ is not in CS of $x$). Another approach could simply update all replicas in CS of $x$ after each write access to $x$, instead of invalidating replicas of $x$ and updating those of CSR. That approach guarantees the atomic consistency, but has two considerable drawbacks. First, it does not guarantee required availability of $x$, if a current cardinality of CS is less than $nr_{\text{min}}$. Second, it suffers from update overhead on each write access. The overhead can significantly be reduced in our approach. Indeed, when $P_i$ issues several subsequent writes to $x$ (which is generally a typical behavior resulting from program locality), the checkpointing is not necessary as far as this sequence is
not interrupted by any read access from another process. The reason for this is that even if \( P_i \) fails after having modified \( x \), these modifications were not yet visible for any other processes (since none of them issued a read request to \( x \)), and therefore restoring the value of \( x \) from the last checkpoint (taken before the modifications of failed \( P_i \)) will preserve the consistency of DSM memory.

In the original Li-Hudak protocol, an ownership of \( x \) migrates following external (i.e. from a process other than the current owner) write requests issued to a WR replica of \( x \). In the extended protocol proposed here, the ownership migration is triggered by any external access to a WR replica, either read or write access. Thus, every external access request to a WR replica of \( x \) enforces a checkpoint of the current value of \( x \). Moreover, this strategy is substantiated additionally by the fact that, in most applications, objects are read directly before writing. In our approach, when a process \( P_i \) issues two subsequent read and write access to the same object \( x \) owned by \( P_j \), only one checkpoint will be performed, since only a single ownership movement will take place.

However, it is necessary to remark, that at the moment of checkpointing \( x \), \( P_i \) can also own some other object \( y \). Then, if \( P_i \) would fail after checkpointing \( x \) but before checkpointing \( y \), the atomic consistency of the memory could be violated on recovery, since the formerly checkpointed value of \( y \) will be inconsistent with checkpointed value of \( x \) (if the recovered state of the memory reflects modifications performed by \( P_i \) on \( x \), it has to reflect as well those done on \( y \)). Therefore, at any object’s ownership transfer, \( P_i \) is required to checkpoint all owned and not yet checkpointed objects (a simple timestamp mechanism can be used to avoid unnecessary checkpointing of objects not modified since the previous checkpoint).

6. Conclusions

In this paper we discussed how to minimize the overhead of the recovery techniques in DSM systems. We showed, that using the knowledge of the message semantics can significantly reduce the cost of logging operations in relaxed consistency models. On the other hand, using replication mechanism for keeping checkpoints seems to be very promising approach to reach high degree of DSM reliability. Moreover, integration of coherency and recovery protocols allows fast recovery at low cost for strict consistency models.

The extended atomic coherence protocol described in this paper, brings up two important contributions: it offers an efficient recovery technique for atomic consistency of shared objects and it tolerates network partitioning as far as a majority partition exists. The protocol has been implemented in PORSHE (Process and Object Replication System for Heterogeneous Environments [4]) – a reliable distributed processing system offering passive and active process replication, based on group communication library Cobra [2], as well as data objects replication, based on Jash DSM platform [3]. Further work should include an investigation of an appropriate dynamic management of the boundary restriction for checkpoint replicas.

References


