An Extended Home-Based Coherence Protocol for
Causally Consistent Replicated Read-Write Objects

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
Jerzy.Brzezinski@cs.put.poznan.pl  Michał.Szychowiak@cs.put.poznan.pl

Abstract

This paper considers the reliability of software Distributed Shared Memory systems where the unit of sharing is a persistent read-write object. We present an extended coherence protocol for causal consistency model, which integrates replication management with independent checkpointing. It uses a novel coordinated burst checkpoint operation in order to replicate consistent checkpoints of shared objects in local memory of distinct system nodes. No special reliable hardware devices are required. The protocol offers high availability of shared objects with limited overhead and ensures fast recovery in case of multiple node failures. In case of the network partitioning all the processes in a majority partition of the system can continuously access all the objects.

1. Introduction

Modern Distributed Shared Memory (DSM) systems definitely tend towards object-oriented processing, and the resulting products, like distributed persistent objects systems, receive growing attention. One of the most important issues in designing such systems is fault tolerance aimed at guaranteeing continuous availability of all objects even in case of failures of some DSM nodes. Each node failure may result in a loss of some objects stored in the local volatile memory of the failed node. The DSM recovery problem consists in restoring the values of lost

---

1 A version of this paper appeared in Proceedings of the International Workshop on Distributed Shared Memory on Clusters (IEEE International Symposium on Cluster Computing and the Grid – CCGrid 2003), May 12 - 15, 2003, Tokyo, Japan
data in such a way that the whole memory remains in a consistent state, according to the consistency model used.

One common solution for this problem is to use replication in conjunction with the write-update strategy, which ensures propagation of every modification of a given object to all its replicas. Thus, in case of any failure, the lost objects can be fetched from the surviving replicas. The exact protocols differ in the scope of the replication, using either full replication where each object is replicated on every system node (as in [1]), partial replication ([5]) or primary/backup replication ([2] and [4]), or in use of communication paradigms, such as broadcast ([1]), or group communication ([12]). However, the write-update strategy is very costly, especially in object-based systems, because of the typically low read-to-write ratio of object accesses. Moreover, the full replication requirement is impractical for most DSM applications.

Alternatively, invalidation-based coherence protocols incur lower overhead of the write operation, in our opinion, being better suited for object-based DSM systems. Unfortunately, they offer no failure resilience and the required level of system reliability must be ensured separately with checkpoint/recovery protocols. Checkpoints are backup copies of objects saved in order to restore the memory state on recovery from node failures. In contrast to the write-update protocols which perform a de facto checkpoint on each write access, the invalidation approach enables to limit the checkpoint frequency only to the moments when actual interprocess dependency occurs. In this approach, the most common solution consists in adapting checkpointing and logging algorithms, developed formerly for message-passing distributed systems, for a DSM environment (e.g. [8],[10],[14],[16]), extending the algorithms with special data structures, conforming to the shared memory unit, used to manage memory checkpoints. The main drawback of this approach is the requirement of an external stable storage able to survive failures (hard discs, in practice) for saving checkpoints, which causes significant overhead and limits the DSM efficiency. However, some recovery protocols for DSM ([3],[4],[9],[15]) make use of the DSM replication mechanism to store checkpoints in the local memory of distinct nodes in place of stable storage. We follow this direction in our research.

This paper presents a new solution for the DSM recovery problem – a home-based coherence protocol extended for low cost checkpointing, which ensures fast recovery and high data availability in spite of multiple node failures and network partitioning. It is an extension of formerly designed home-based coherence protocol [6] for causal consistency model [1]. In the home-based approach every object has been assigned a home node responsible for collecting all
the modifications of that object. Recently, some reliability extensions have been proposed for home-based DSM systems. Most of them concern lazy-release consistency LRC ([4],[11],[16]) which ensures the consistency of object accesses issued only within periods synchronized by pairs of acquire (lock) and release (unlock) operations. The dynamic data replication protocol [4], for instance, uses a primary/backup replication of all home nodes and can recover from a single node failure only. It maintains two distinct homes for each shared object, both updated on every release operation. On the other hand, the protocols [11] and [16] overcome the 1-failure reliability by adequate use of efficient logging and occasional checkpointing. However, they require stable storage to keep checkpoints.

As the causal consistency model offers replication transparency for the DSM applications, no explicit synchronization operations are required (although allowed) therefore it is not possible to limit the checkpointing frequency only to those synchronization moments. Therefore it is necessary to develop a new checkpointing technique that would enable to reduce the checkpoint frequency, still offering high DSM reliability. In [3] we propose a novel delayed burst checkpoint protocol which permits to process several object modifications without the checkpoint, but preserving the causal consistency. That protocol is an extended distributed directory coherence protocol and requires the reliability of directory services. The home-based approach does not require such services, but the delayed burst checkpointing incurs much higher checkpoint frequency than for the distributed directory. Thus, in this paper we propose a new protocol that better suits the home-based coherence. The protocol limits the number of checkpoint operations, but may require checkpoint coordination on several processes. To the best of our knowledge, no other recovery protocol has been proposed for home-based coherence in the causal consistency model. Compared to similar checkpoint replication protocols [9], [15], the one proposed here guarantees the availability of all objects in spite of multiple node failures and network partitioning (only in the majority partition).

This paper is organized as follows. In Section 2 we define the system model. Section 3 details a new coherence protocol extended with checkpointing. Some concluding remarks are given in Section 4.
2. Basic definitions and problem formulation

System model

The DSM system is composed of a finite set $P$ of sequential processes $P_1, P_2, ..., P_n$ that run on system nodes $h_1, h_2, ..., h_n$ and cooperate via a finite set $O$ of shared objects. The objects are persistent (long life) i.e. once created, the object continues to exist even after its creator process has terminated, until is explicitly destructed. Each persistent object consists of its current state (object value) and object methods which read and modify the object state. We consider only read-write objects, i.e. we distinguish two operations on the objects: read access and write access. The read access $r(x)$ to an object $x$ is issued when process $P_i$ invokes a read-only method of the object $x$. The write access $w_i(x)$ to an object $x$ is issued when process $P_i$ invokes any other method of $x$. Each write access results in a new object value of $x$. By $r(x)v$ we denote that the read operation returns value $v$ of $x$, and by $w_i(x)v$ that the write operation stores value $v$ to $x$.

To increase the efficiency of DSM access operations, objects are replicated on demand, allowing concurrent access to the same object at several processes. However, concurrent existence of different replicas of the same object requires consistency management, and this is job of the coherence protocol. This protocol performs all communication necessary for the consistency management via message-passing.

Causal memory

In this work, we investigate causal consistency [1]. This consistency model guarantees that all processes accessing a set of shared objects will perceive the same order of causally related operations on those objects. This model is sufficient for several classes of distributed algorithms but requires weaker synchronization than atomic or sequential consistency, thus allowing for more concurrency and efficiency.

We assume that causal consistency is maintained by a coherence protocol proposed in [6]. Here we will refer to this protocol as the basic protocol. It uses a write-invalidate schema which ensures that all local reads reflect the causal order of object modifications, by invalidating all potentially outdated replicas. If, at any time, process $P_i$ updates object $x$, it determines all locally stored replicas of objects that could have possibly been modified before $x$, and denies any further access to them (invalidates), thus preventing $P_i$ from reading inconsistent values. Any access request issued to an invalidated replica of $x$ results in fetching its up-to-date value from a master replica of $x$. In the home-based approach every object $x$ has been assigned a home node.
which holds the master replica and collects all the modifications of $x$. The process running on that node is called **owner** of $x$. Different objects can have different home nodes (owners). All replicas except master replicas will be called **ordinary**.

**Failure model**
We assume the **fail-stop** model of failures, i.e. the DSM nodes and communication links can fail at arbitrary moments by crashing. Any such failure must eventually be detected. Also, network partitioning must consequently result in detecting all unavailable processes as failed, for each system partition. We allow merging separate partitions of the system on provided that partitioning is detected before the merge.

**2.1 Basic coherence protocol**
The basic coherence protocol [6] introduces three different states of an object replica: writable (indicated by WR status of the replica), read-only (RO status), and invalid (INV status). Only the master replica can have the WR status, and this status enables the owner to perform instantaneously any write access to that replica. Meanwhile, every process is allowed to instantaneously read the value of any local RO replica. However, the write access to any RO replica requires additional coherence operations. We assume that there is a home-node for each object $x$, and the identity of the owner holding the master replica of $x$ is known. The outline of the basic protocol is presented below.

The causal relationship of the memory accesses is reflected in the vector timestamps associated with each shared object. Each process $P_i$ manages a vector clock $VT_i$. A vector clock is a well-known mechanism used to track causal dependency of events in distributed systems [13]. In DSM, it is intended to track the causal precedence of only read and write operations. The value of $i$-th component of the $VT_i$ counts writes performed by $P_i$. More precisely, only intervals of write operations not interlaced with communication with other processes are counted, as it is sufficient to track the causal dependency between operations issued by distinct processes.

There are three operations performed on $VT_i$:
- $inc(VT_i)$ – increments a $i$-th component of the $VT_i$; this operation is performed on write-faults and read requests from other processes;
- $update(VT_i, VT_j)$ – returns the component wise maximum of the two vectors; this operation is performed on updating a local replica with some value received from another process;
Each replica of shared object \(x\) stored at \(P_i\) has been assigned a \(VT_i^x\). On any local modification \(w_i(x)\) \(VT_i^x\) becomes equal to \(VT_i\), and on the update \(UPD(x,v,VT^x)\) from the master replica, \(VT_i^x\) becomes equal to \(VT^x\). A \(local\_invalidate_i\(VT)\) operation ensures the correctness of the basic protocol, by setting to INV the status of all locally held replicas \(x\) not owned by \(P_i\), for which \(compare(VT_i^x<VT)\) is true. The reason for this operation is to invalidate all replicas that could have potentially been overwritten since \(VT_i^x\) till \(VT\).

Actions of the basic protocol are described in the subsequent points:

1) If process \(P_i\) wants to access a locally available (i.e. in any other state than INV) object \(x\) for reading, the read operation is performed instantaneously.

2) If \(P_i\) wants to access the master replica of \(x\) for writing, the protocol checks the status of the master replica of \(x\). If it is RO, the \(inc(VT_i)\) is performed first and the master replica switches into the WR state. The write operation is performed and the value of \(VT_i\) is stored with the replica as \(VT_i^x\).

3) If \(P_i\) wishes to gain a read access to object \(x\) unavailable locally (i.e. INV), the protocol issues a read request \(r_i(x)\) to the owner of \(x\), say \(P_k\). The owner sends back \(UPD(x,v,VT_k^x)\) message containing a RO replica of \(x\) along with the \(VT_k^x\). If the master replica of \(x\) was currently in state WR, it switches to RO, ensuring to increment \(VT_k^x\) on further local write operations. On receipt of the \(UPD(x,v,VT_k^x)\) message, first the \(local\_invalidate_i\(VT_k)\) is processes, and then \(r_i(x)v\) is performed.

4) If \(P_i\) wishes to perform a write access to ordinary replica of \(x\), it performs \(inc(VT_i)\) and issues a write request \(w_i(x)v\) timestamped with \(VT_i\) to the object owner – \(P_k\) (note that the value \(v\) being currently written is also sent to the owner). On this request, the owner performs \(local\_invalidate_i\(VT_k)\), then synchronizes the clocks with \(update(VT_i,VT_k)\), updates \(x\) and sends back to \(P_i\) \(UPD(x,–,VT_k)\). When arrived at \(P_i\), both \(update(VT_i,VT_k)\) and \(local\_invalidate_i\(VT_k)\) are executed, and finally \(x\) is stored with value \(v\) along with \(VT_i^x=VT_k\).

3. Checkpoint replication protocol for causal consistency

Here we propose extension of the basic consistency protocol [6]. The extension aims to offer low-cost checkpointing of shared objects and high availability of checkpoints in spite of multiple node failures and network partitioning. It ensures fast recovery and enables to continue operations on all the objects in the majority partition of the system. Objects are checkpointed by
erations on all the objects in the majority partition of the system. Objects are checkpointed by their owners, and only when a new interprocess causal dependency arises between object accesses. Several modifications may be performed on a given object without the need to checkpoint its value.

Checkpoints are stored in DSM as special-purpose replicas (called checkpoint replicas). The identities of DSM nodes holding checkpoint replicas are stored in CCS (checkpoint copyset) maintained by object owners. CCS(x) is initiated at the creation of x and does not include the owner. The content of CCS(x) can change accordingly to further access requests or failure pattern, or any load balancing mechanisms. In order to limit the cost of a checkpointing operation, the number \( nr \) of checkpoint replicas should always be kept between boundaries \( nr_{\text{min}} \) and \( nr_{\text{max}} \). The number \( nr_{\text{min}} \) represents the minimum number of checkpoint replicas necessary to reach the 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. Here we let

\[
nr_{\text{min}} = \left\lceil \frac{n}{2} \right\rceil, \quad \text{where } n \text{ is the total number of system nodes.}
\]

This guarantees that, in case of network partitioning, there will always be a valid checkpoint in a majority partition. The problem of creating a majority partition has been addressed in several works (see [7], among others) and we do not consider it in this paper.

In addition to RO, WR and INV states, the proposed extended protocol introduces two new replica states: C (checkpoint) and ROC (read-only checkpoint). Checkpoint replica C is updated on checkpoint operations and never become invalidated. A single checkpoint operation \( \text{ckpt}(x,v) \) is performed by the owner of x, \( P_k \), and consists in atomically updating all C replicas of x held by processes \( P_j \) included in CCS(x) with the value \( v \), carried in CKPT(x,v,VT_{T_k}) message. After that moment, the C replica held by \( P_j \) will be duplicated to a new ROC replica on the first local access to x. In other words, a checkpoint replica is a prefetched current value of x and can therefore serve further accesses. The \( \text{local invalidate}(VT_j) \) must be executed when copying a checkpoint replica from C to ROC. However, during the next single checkpoint operation \( \text{ckpt}(x,v') \), all existing ROC replica of x are destroyed.

We denote by RO/dirty and ROC/dirty an additional information about the write accesses invoked locally on RO or ROC replica. Such a replica is in RO/dirty or ROC/dirty state when it was modified by the holding process but possibly not yet checkpointed by its owner.
The \textit{coordinated burst checkpoint} operation consists in atomically performing two operations: (1) taking a single checkpoint of all object $x$ in WR state and (2) sending checkpoint requests CKPT$_{REQ}(y)$ to owners of all other objects $y$ in RO/dirty and ROC/dirty states. After that, the RO/dirty and ROC/dirty objects are switched to RO and ROC states, respectively.

Actions of the extended protocol are described by the following rules:

Rule 1) If process $P_i$ wants to access the master replica of object $x$ for reading, the read operation is performed instantaneously.

Rule 2) If $P_i$ wants to access the master replica of $x$ for writing, the protocol checks the status of the master replica of $x$. If it is RO, the $inc(VT_i)$ is performed first and the master replica switches into the WR state. The write operation is processed and the value of $VT_i$ is stored along with the replica as $VT_i^x$.

Rule 3) If $P_i$ invokes a read access to a checkpoint replica of $x$ in C state, and local ROC replica of $x$ does not exist, first the $local\_invalidate(VT_i^x)$ is executed, then the replica value is copied to a new ROC replica and, finally, the read operation is performed.

Rule 4) If $P_i$ wants to access a replica $x$ in RO or ROC state for reading, the read operation is performed instantaneously.

Rule 5) If $P_i$ wishes to gain a read access to object $x$ unavailable locally (i.e. INV), the protocol issues a read request $r_i(x)$ to the owner of $x$, $P_k$. On receipt of the UPD$(x,v,VT_k^i)$ message from the owner, first the $local\_invalidate(VT_k^i)$ is processed, and then $r_i(x)v$ is performed.

Rule 6) If $P_i$ wishes to perform a write access to an ordinary replica of $x$, it performs $inc(VT_i)$ and issues a write request $w_j(x)v$ timestamped with $VT_i$ to the object owner – $P_k$ (note that the value $v$ being currently written is also sent to the owner). When the returned UPD$(x,v,VT_k^i)$ arrives $P_i$, synchronizes the clocks with update$(VT_i,VT_k)$ and performs $local\_invalidate(VT_k)$, and finally $x$ is stored with value $v$ along with $VT_i^x=VT_k^x$.

Rule 7) When $P_i$ receives a $r_j(x)$ request from $P_j$, first the state of the master replica of $x$ is consulted. If the master replica is in state WR or RO/dirty, the \textit{coordinated burst checkpoint} operation is executed. Then, the owner sends back to $P_j$ a UPD$(x,v,VT_i^x)$ message containing a RO replica of $x$ along with the $VT_i^x$.

Rule 8) When $P_i$ receives a $w_j(x)v$ request from $P_j$, both $local\_invalidate(VT_i)$ and update$(VT_i,VT)$, are executed, and then $P_i$ updates the master replica of $x$ with $v$. If the master replica is in WR state or it is RO/dirty and $x.writer$ is not $P_j$, the \textit{coordinated
**burst checkpoint** operation is executed (the new value $v$ of $x$ is checkpointed) and $x.writer$ becomes $P_j$. However, if $x.writer$ is already $P_j$ and the master replica is in RO or RO/dirty state, no checkpoint operation is necessary, only $x$ switches into RO/dirty state. In the end, $\text{UPD}(x, -, VT_i)$ is sent back to $P_j$ (note that the current status of $x$ is sent with $x$).

**Rule 9)** When $P_i$ receives a $\text{CKPT_REQ}(x)$ message, and the master replica of $x$ is in WR state, the **coordinated burst checkpoint** operation is executed and all master replicas held by $P_i$ are switched to RO.

A sample execution of the extended protocol is presented on . Process $P_i$ owns object $x$ and $P_j$ owns $y$. Both $x$ and $y$ are single-value objects initialized to 0. The initial value of both $VT_i$ and $VT_j$ is $[0,0]$. Let $x\{S,v,VT^s\}$ mean that at a given point of time $x$ is in state $S \in \{\text{WR, RO, RO/dirty, ROC, ROC/dirty}\}$, and its value $v$ is timestamped with $VT^s$. When $P_i$ has modified $x$, and receives a write request from $P_j$, the new value 2 of $x$ is checkpointed (in order to keep the drawing small, no checkpoint replicas of $x$ are shown on the figure). In case of $P_i$ failure, the checkpointed value $x=2$ can be restored as it remains consistent with any subsequently modified object by $P_j$ (e.g. $y=4$). Since that moment several modifications of $x$ are possible, provided that no third process access any modified value causally related with some modification of $x$. In this example, when $P_i$ issues a read request to $y$, the current value 9 of $y$ is checkpointed, forcing also $P_i$ to checkpoint $x$ (as the formerly checkpointed value 2 of $x$ is not causally consistent with $y=9$).
3.1 Recovery
At any time before any failure occurs, there are at least \( nr_{\text{min}} + 1 \) replicas of \( x \) (the master replica plus \( nr > nr_{\text{min}} \) C replicas), thus in case of a failure of \( f < nr_{\text{min}} \) processes failure (at most \( f \) processes crash or become separated from the majority partition) there will be at least one non-faulty replica of \( x \) in the primary partition, and can serve further access requests to \( x \). As long as the current owner is non-faulty in the majority partition, the extended coherence protocol assures processing all requests to \( x \) issued in the majority partition. However, if the current owner becomes unavailable, the recovery procedure must elect a new owner from among all available processes in the partition – the one holding the most recent replica of \( x \) (timestamp comparison is necessary for RO/ROC replicas). If there are no RO/ROC replicas of \( x \) any available C replica may be used to restore the \( x \) value. This is sufficient to continue the distributed processing in the majority partition. Alternatively, all shared object may be synchronously rolled back to their checkpoint values, and all RO/dirty and ROC/dirty replicas discarded.

4. Conclusions
The coherence-recovery protocol for causally consistent replicated objects proposed in this paper uses a novel coordinated burst checkpointing technique, which reduces the frequency of checkpointing in home-based coherence approach. The presented protocol brings up important
contributions: it offers a checkpoint replication resulting in high availability of all DSM objects in spite of multiple node failures, potentially improves read access performance by object pre-fetch and tolerates network partitioning.

The proposed checkpoint replication technique can be directly used in other coherence protocols for the causal consistency model. For example one of the coherence protocols proposed in [6] differs from the basic protocol [2] in using object-version vectors instead of vector clocks to timestamp update messages. Such a protocol does not incur unnecessary invalidation. However, in fine-grained object-based memory, version vectors can grow very large and the overhead of using them in the coherence messages will probably make the implementation impractical.

References:


