# An Extended Coherence Protocol for Recoverable DSM Systems with Causal Consistency*

Jerzy Brzeziński and 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
{jbrzezinski,mszychowiak}@cs.put.poznan.pl

**Abstract.** This paper presents a new checkpoint recovery protocol for Distributed Shared Memory (DSM) systems with read-write objects. It is based on independent checkpointing integrated with a coherence protocol for causal consistency model. That integration results in high availability of shared objects and ensures fast restoration of consistent state of the DSM in spite of multiple node failures, introducing little overhead. Moreover, in case of network partitioning, the extended protocol ensures that all the processes in majority partition of the DSM system can continuously access all the objects.

## 1 Introduction

One of the most important issues in designing modern Distributed Shared Memory (DSM) systems is fault tolerance, namely recovery, aimed at guaranteeing continuous availability of shared data even in case of failures of some DSM nodes. The recovery techniques developed for general distributed systems suffer from significant overhead when imposed on DSM systems (e.g. [3]). This motivates investigations for new recovery protocols dedicated for the DSM. Our research aims at constructing a new solution for the DSM recovery problem which would tolerate concurrent failures of multiple nodes or network partitioning. In [2] we have proposed the concept of a coherence protocol for *causal consistency model* [1] extended for low cost checkpointing which ensures fast recovery. To the best of our knowledge it is the first checkpoint-recovery protocol for this consistency model. In this paper we present a formal description of the protocol as well as the proof of its correctness.

This paper is organized as follows. In section 1 we define the system model. Section 3 details a new coherence protocol extended with checkpointing in order to offer high availability and fast recovery of shared data. The protocol is proven correct in section 4. Concluding remarks are given in section 5.

* This work has been partially supported by the State Committee for Scientific Research grant no. 7T11C 036 21

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 475–482, 2004.

## 2 Basic Definitions and Problem Formulation

### 2.1 System Model

The DSM system is a distributed system composed of a finite set P of sequential processes P1, P2, ..., Pn that can access a finite set O of shared objects. Each shared object consists of its current state (object value) and object methods which read and modify the object state. We distinguish two operations on shared objects: read access and write access. The read access ri(x) to object x is issued when process Pi invokes a read-only method of object x. The write access wi(x) to object x is issued when process Pi invokes any other method of *x*. Each write access results in a new object value of *x*. By $r_i(x)v$ 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*.

The replication mechanism is used to increase the efficiency of the DSM object access, by allowing each process to locally access a replica of the object. However, concurrent access to different replicas of the same shared object requires consistency management. The coherence protocol synchronizes each access to replicas, accordingly to the appropriate consistency model. This protocol performs all communication necessary for the interprocess synchronization via message-passing.

### 2.2 Causal Memory

Let *local history* $H_i$ denote the set of all access operations issued by $P_i$, *history* $H$ – the set of all operations issued by the system and $HW$ – the set of all write operations.

**Definition 1.** *The **causal-order** relation in H, denoted by* $\rightarrow$, *is the transitive closure of the local order relation* $\rightarrow_i$ *and a write-before relation that holds between a write operation and a read operation returning the written value:*

(i) $\forall_{o^1,\, o^2 \in H} \left( \left( \exists_{i=1..n} o^1 \rightarrow_i o^2 \right) \Rightarrow o^1 \rightarrow o^2 \right)$,

(ii) $\forall_{x \in O} w(x)v \rightarrow r(x)v$ ,

(iii) $\forall_{o^1,\, o^2,\, o \in H} \left( (o^1 \rightarrow o \ \wedge \ o \rightarrow o^2) \Rightarrow o^1 \rightarrow o^2 \right)$.

As $P_i$ is sequential, it observes the operations on shared objects in a sequence which determines a local serialization $\mapsto_i$ of the set $H_i \cup HW$.

**Definition 2.** *Execution of access operations is **causally consistent** if serialization* $\mapsto_i$ *satisfies the following condition:*

$$\forall_{o^1,\, o^2 \in H_i \cup HW} \left( o^1 \rightarrow o^2 \Rightarrow o^1 \mapsto_i o^2 \right).$$

The causal consistency model guarantees that all processes accessing a set of shared objects will perceive the same order of causally related operations on those objects.

## 3 The Integrated Coherence-Checkpointing Protocol

We describe now the integrated coherence-checkpointing protocol, CAUSp, which is an extension of a basic coherence protocol originally proposed in [1]. The basic protocol 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 an object $x$, it determines all locally stored replicas of objects that could have possibly been modified before $x$, and invalidates them, preventing from reading inconsistent values. Any access request issued to an invalid replica of $x$ requires fetching the up-to-date value from a *master replica* of $x$. The process holding a master replica of $x$ is called $x$'s *owner*. We assume the existence of reliable *directory services* which can provide a process with the identity of the current owner of any required object.

### 3.1 The CAUSp Protocol

The CAUSp protocol distinguishes 3 ordinary states of an object replica: *writable* (indicated by the WR status of the replica), *read-only* (RO status), and *invalid* (INV status). Only the WR status enables to perform instantaneously any write access to the replica. However, every process is allowed to instantaneously read the value of a local replica in either RO or WR state. Meanwhile, the INV status indicates that the object is not available locally for any access. Thus, the read or write access to the INV replica, and the write access to the RO replica require the coherence protocol to fetch the value of the master replica of the accessed object.

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 $VT_i$. 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;
- $VT_i<VT_j$ – true iff $\forall k$: $VT_i[k]\leqslant VT_j[k]$ and $\exists k$: $VT_i[k]\neq VT_j[k]$

The replica of object $x$ stored at $P_i$ has been assigned a vector timestamp $VT_i^x$. The $local\_invalidate_i(VT)$ operation ensures the correctness of the protocol, by setting to INV the status of all RO replicas $x$ held by $P_i$, for which $VT_i^x<VT$ is true.

Checkpoints are stored in DSM as *checkpoint replicas* denoted C (checkpoint) and ROC (read-only checkpoint). The identities of DSM nodes holding checkpoint replicas are maintained in CCS (*checkpoint copyset*). CCS($x$) is initiated at the creation of $x$, then maintained by the object owner, but never includes the owner. The content of CCS($x$) can change accordingly to further access requests or failure pattern, or any load balancing mechanisms, but the number of checkpoint replicas should always en-

sure the desired failure resilience [2]. Checkpoint replica is updated on checkpoint operations and never becomes invalidated. Checkpoint operation $ckpt(x)v$ consists in atomically updating all checkpoint replicas held by processes included in CCS($x$) with value $v$, carried in CKPT($x$,$v$,$VT^x$) message, and setting their status to C. After that moment, any C replica can be switched to ROC on the next local read access to $x$. Until the next checkpoint, the ROC replica serves read accesses as RO replicas do (as the checkpoint replica holds a prefetched value of $x$).

Actions of the extended protocol are presented in Fig. 1.

### 3.2 Delayed Checkpointing and Burst Checkpointing

The checkpointing is performed independently for each process. Moreover, several modifications of objects can be performed without the need for taking checkpoints. Indeed, when $P_i$ issues several subsequent writes to $x$ (which is generally a typical behavior resulting from program locality), the checkpoint is not necessary as far as this sequence is not interrupted by any read access from another process. However, it is necessary to remark, that at the moment of checkpointing $x$, $P_i$ can also own some other object $y$, which has been modified before the last modification of $x$. Then, if $P_i$ fails after checkpointing $x$ but before checkpointing $y$, the consistency of the memory could be violated on recovery, since the formerly checkpointed value of $y$ will be inconsistent with the checkpointed value of $x$. Therefore, on each checkpoint of a dirty object, $P_i$ is required to atomically checkpoint all dirty replicas (*burst checkpointing*).

### 3.3 Recovery

Before any failure occurs there are at least $nr_{min}+1$ replicas of $x$, thus in case of a failure of $f \leqslant nr_{min}$ processes (at most $f$ processes crashes or become separated from the majority partition) there will be at least one non-faulty replica of $x$ available. If the directory manager discovers the unavailability of $x$'s owner, it sequentially contacts processes included in CCS($x$) and elects the first available as new owner. In case of the network partitioning, there is at least one such a process in the primary partition.

In order to protect the causal consistency of further access operations, the local invalidation operation must be processed on recovering an object from its checkpoint. The elected owner will perform $local\_invalidate_i(x.VT)$ for each $x$ recovered from a C state replica. This operation is presented in Fig. 2.

## 4 Correctness of the Protocol

Due to space limitation we will here provide the proof of the safety property only. The safety property asserts that the CAUSp protocol correctly maintains the coherency of shared data, accordingly to the causal consistency model, besides any allowable failures. For the sake of simplicity of the presentation, the recovery operation $recover_i(x)v$ is considered as a read access operation $r_i(x)v$ from the safety viewpoint.

```
1.  on r_i(x)v :        if x.state = C
2.                          local_invalidate_i(x.VT)
3.                          VT_i := update(VT_i,x.VT)
4.                          x.state := ROC
5.                      if x.state = INV
6.                          send R_REQ(x.id) to x.owner
7.                          wait for UPD(x,VT_k)
8.                          local_invalidate_i(VT_k)
9.                          VT_i := update(VT_i,VT_k)
10.                         x.VT := VT_k
11.                         x.state := RO
12.                     v := x.value

13. on w_i(x)v :        if x.state ≠ WR
14.                         send W_REQ(x.id) to x.owner
15.                         wait for ACK(x.id)
16.                         inc(VT_i)
17.                         x.VT := VT_i
18.                         x.state := WR
19.                         x.owner := P_i
20.                     x.value := v
21.                     x.dirty := true

22. on R_REQ(id) from P_j:  inc(VT_i)
23.                         find x :: x.id = id
24.                             ckpt_i(x)
25.                             send UPD(x, VT_i) to P_j

26. on W_REQ(id) from P_j:  find x :: x.id = id
27.                             ckpt_i(x)
28.                             x.state := RO
29.                         send ACK(x.id) to P_j

30. on CKPT(x,VT) from P_j: find y :: y.id = x.id
31.                             y.value := x.value
32.                             y.state := C
33.                             y.VT := VT
34.                         send ACK(x.id) to P_j

35. on ckpt_i(x) :      if x.dirty=true
36.                       for each y :: y.dirty=true
37.                         atomically update checkpoint replicas of y with
                            sending CKPT(y,VT_i) to all P_j∈CCS(y) and
                            collecting ACK(y.id) from all correct of them
38.                       y.dirty := false
```

**Fig. 1.** Actions of the CAUSp protocol for process $P_i$

### 4.1 Proof of the Safety Property

First, let us introduce another vector clock comparison operation:

- $VT_i \leqslant VT$ – which is true iff $\forall k$: $VT_i[k] \leqslant VT[k]$

```
1  on recover_i(x)v :  if x.state = C
2                          local_invalidate_i(x.VT)
3                          VT_i := update(VT_i,x.VT)
4                      x.state := WR
5                      x.dirty := false
6                      x.owner := P_i
```

**Fig. 2.** Recovery procedure performed by the new owner of $x$

**Lemma 1.** *For any $P_i$, vector clock $VT_i$ is monotonically increasing.*

*Proof.* There are only 2 possible modification operations performed by the protocol on the vector clock:
1) $inc(VT_i)$ – incrementing the $i$-th position $VT_i[i]$ of the vector clock;
2) $VT_i := update(VT_i,VT)$ – which performes $VT_i[k] := max(VT_i[k],VT[k])$ for each $k$-th position of the vector clock. Therefore, the vector clock is never decremented and for any two subsequent vector values $VT_i^1$ and $VT_i^2$; $VT_i^1 \leqslant VT_i^2$ is true. □

**Lemma 2.** *For any operations $w_i(x)v$, $w_i(x)u$, such that $w_i(x)v \rightarrow_i w_i(x)u$ holds, and $r_j(x)v$ $(i \neq j)$, there is $VT^1 \leqslant VT^2$, where $VT^1$ is the value of $x.VT$ when operation $r_j(x)v$ ends and $VT^2$ is the value of $VT_i$ when the second write operation $w_i(x)u$ ends.*

*Proof.* First, note that value $v$ of $x$ has been made visible to $P_j$ only if, after $w_i(x)v$ $(i=x.owner)$, $P_i$ had received R_REQ sent by $P_j$ on $r_j(x)$. The first action performed by $P_i$ on reception of that R_REQ was $inc(VT_i)$. Let $VT^u$ be the incremented value of $VT_i$. Then, update message UPD$(x,VT^u)$ was sent from $P_i$ to $P_j$, making $x.VT$ equal to $VT^u$. Now, from Lemma 1, $VT^u \leqslant VT^2$ must be true, so we get $x.VT \leqslant VT^2$. □

**Lemma 3.** *For any operations $w_i(x)v$ and $r_j(x)v$, $i \neq j$, there is $VT^l < VT^2$, where $VT^l$ is the value of $VT_i$ at the moment when operation $w_i(x)v$ ends and $VT^2$ is the timestamp value received in* UPD$(x,VT^2)$ *message during operation $r_j(x)v$.*

*Proof.* Again, note that value $v$ of $x$ has been made visible to $P_j$ if, after $w_i(x)v$ time-stamped with value $VT^l$, $P_i$ had received R_REQ sent by $P_j$ on $r_j(x)$. So, $P_i$ has performed $inc(VT_i)$ on reception of that R_REQ. Then, UPD$(x,VT^2)$ message was sent with the incremented value of $VT_i$, which is $VT^2$. Thus we have $VT^l < VT^2$. □

**Lemma 4.** *For any operations $o_i^1$ and $o_j^2$, such that $i \neq j$ and $o_i^1 \rightarrow o_j^2$ holds, there is $VT^1 < VT^2$, where $VT^1$ is the value of $VT_i$ when operation $o_i^1$ ends and $VT^2$ is the value of $VT_j$ when operation $o_j^2$ ends.*

*Proof.* The case $o_i^1 = w_i(x)v$, $o_j^2 = r_j(x)v$ is subject of Lemma 3, thus it has been proven already. In case of $o_i^1 = w_i(x)v$ such that $w_i(x)v \rightarrow r_j(x)v \rightarrow_j o_j^2$, let $VT^r$ be the value of $x.VT$ when operation $r_j(x)v$ ends. From Lemma 3 $VT^1 < VT^r$ and from Lemma 1 $VT^r \leqslant VT^2$. Thus, $VT^1 < VT^2$ is true.

For a general case: $o_i^1 \rightarrow w_k(x)v \rightarrow r_j(x)v \rightarrow_j o_j^2$, we apply the following induction:

step 1) if $o_i^1 \rightarrow_i w_i(x)v \rightarrow r_j(x)v \rightarrow_j o_j^2$, let $VT^w$ be the value of $VT_i$ when operation $w_i(x)v$ ends and $VT^r$ be the value of $x.VT$ when operation $r_j(x)v$ ends. From Lemma 1 we have both $VT^1 \leqslant VT^w$ and $VT^r \leqslant VT^2$. On the other hand, from Lemma 3, $VT^w < VT^r$ is true. Thus, $VT^1 < VT^2$ is also true.

step 2) if $o_i^1 \rightarrow w_k(x)v \rightarrow r_j(x)v \rightarrow_j o_j^2$, where $k \neq i$, we can follow the induction recursively assuming $o^2 = w_k(x)v$, and apply step 1 or step 2 for $o_i^1 \rightarrow o^2$. □

In the following analysis we shall consider a value returned by a read access performed by the protocol, as the causal consistency violation may result only from reading an inconsistent value (there is no notion of "causally inconsistent writing").

**Definition 3.** *Read operation r(x)v performed by the CAUSp protocol is* ***legal*** *iff value v has not been overwritten by any causally preceding write operation, i.e.:*

$$\nexists_{w(x)u \in HW} \; u \neq v \;\wedge\; w(x)v \rightarrow w(x)u \;\wedge\; w(x)u \rightarrow r(x)v$$

The value that can be returned by a legal read is consistent in terms of Definition 2. A value which is not consistent will hereafter be called *outdated*.

Now let us note that as long as $P_i$ does not interact with other processes, all local access operations are performed on consistent values. This is because the local order preserves causal dependency. Value $v$ of $x$ held by $P_i$ may actually become outdated only if $P_i$ receives an UPD($y,VT''$) message from another process with some value $y.value=a$ that causally depends on $w(x)u$ which, in turn, overwrites $v$ with $u$, i.e.: $w(x)v \rightarrow w(x)u \wedge w(x)u \rightarrow w(y)a \wedge r_i(y)a \rightarrow_i r_i(x)$. However, this chain of dependency will be reflected in value $VT''$ of the vector clock received with this UPD($y,VT''$) message. This vector clock value is used in local invalidation which succeeds the reception of the UPD message.

**Lemma 5.** *All outdated values of locally available replicas are invalidated during the local invalidation operation.*

*Proof.* By contradiction, let us assume that $P_i$ holds a replica of $x$ with outdated value $v$ timestamped $x.VT$, thus $r_i(x)v$ is not legal, i.e.:

$$\exists_{w(x)u \in HW} \; u \neq v \;\wedge\; w(x)v \rightarrow w(x)u \;\wedge\; w(x)u \rightarrow r_i(x)v$$

The following relations are possible between $w(x)v$ and $w(x)u$:

1) both operations $w(x)v$, $w(x)u$ have been issued by the same process, say $P_j$, i.e.: $w_j(x)v \rightarrow_j w_j(x)u$. The causal dependency $w_j(x)u \rightarrow r_i(x)$ is only due to some operation $r_i(y)a$ performed by $P_i$ before $r_i(x)$. Let us assign value $VT^1$ of $VT_j$ to $w_j(x)u$, $VT^2$ to $w_j(y)a$, $VT'$ to UPD($x,VT'$) message sent by $P_j$ after $w_j(x)v$, and, finally, $VT''$ to UPD($y,VT''$) message sent after $w_j(y)a$. In this case $x.VT=VT'$ holds for replica of $x$ at $P_i$ and the following predicates are fulfilled: $VT' \leqslant VT^1$ from Lemma 1, $VT^1 \leqslant VT^2$ from Lemma 1 and $VT^2 < VT''$ from Lemma 3. Provided the above, $x.VT<VT''$ is true, and it makes $x$ invalidated during *local_in-validation*$_i(VT'')$ performed on reception of the second update message UPD($y,VT''$). The value returned then by subsequent $r_i(x)$ cannot be $v$ anymore.

2) operations $w(x)v$, $w(x)u$ have been issued by distinct processes, i.e.: $w_k(x)v \rightarrow w_j(x)u$. Let us assign value $VT^0$ of $VT_k$ to operation $w_k(x)v$ and then values $VT^1$, $VT^2$,

$VT'$, $VT''$ to $w_j(x)u$, $w_j(y)a$, UPD$(x,VT')$ and UPD$(y,VT'')$, respectively, as in point 1 of this proof. In this case, still $x.VT=VT'$ holds for replica of $x$ at $P_i$, $VT^1 \leqslant VT^2$ and $VT^2<VT''$ also hold as above and, moreover, $VT'<VT^1$ is true from Lemma 4. Therefore, $x.VT<VT''$ is also true in this case, and $x$ will be invalidated during *local_invalidation*$_i(VT'')$ on reception of UPD$(y,VT'')$. Again, the value returned by subsequent $r_i(x)$ cannot be $v$ anymore. □

**Theorem 1. (safety)** *Every read access to object x performed by any process is legal.*

*Proof.* Let us consider $r_i(x)v$ issued to the replica of $x$ in one of the following states:

1) WR – this case is trivial as this state designates the master replica of $x$ which is never outdated since only the owner can modify $x$;

2) INV – also trivial as no access is ever possible to a replica in the INV state;

3) RO – no replica in that state can contain an outdated value $v$ because any modification causally following the write of $v$ can be made visible to $P_i$ only after reception of some UPD message which triggers local invalidation. From Lemma 5 all outdated values have been invalidated then;

4) C – similar to INV, except that no update is necessary; only the local invalidation is processed on the first read access issued to a C replica and the replica state is switched into ROC before value $v$ is read;

5) ROC – the local invalidation was performed before the replica state has been switched to ROC, so all outdated values were invalidated (from Lemma 5). □

## 5 Conclusions

The coherence-recovery protocol proposed in this paper uses a novel burst checkpointing approach. Checkpointing is performed independently for each process, thus does not incur high interprocess synchronization overhead. The protocol enables fast recovery on the assumption of the reliability of the directory services, i.e. directory information for each object must be correctly provided to the requested processes at any moment in the majority partition. Fortunately, this assumption can be fulfilled with directory checkpointing and replication for each object, similarly to application data objects.

## References

1. Ahamad M., Hutto P.W. and John R.: Implementing and Programming Causal Distributed Shared Memory. 11th Int.Conf. on Distributed Computing (1991) 274-281
2. Brzeziński J. and Szychowiak M.: An Extended Coherence Protocol for Recoverable DSM Systems with Causal Consistency. Int. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA'03, Las Vegas (2003)
3. Kongmunvattana A., Tanchatchawal S. and Tzeng N.-F.: Coherence-Based Coordinated Checkpointing for Software Distributed Shared Memory Systems. 20th Conf. on Distributed Computing Systems (2000) 556-563