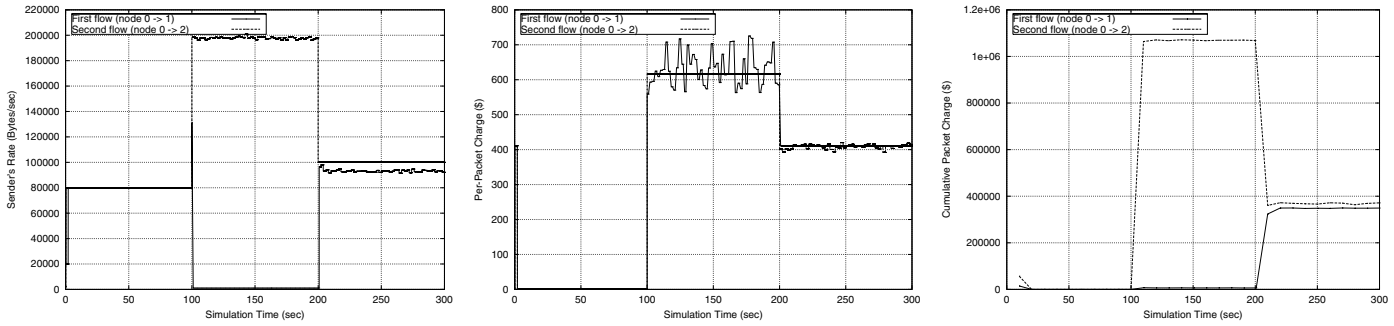

Figure 4: Single-hop dynamic scenario. (a. Sending rate; b. Per-packet charge; c. Cumulative charge.)

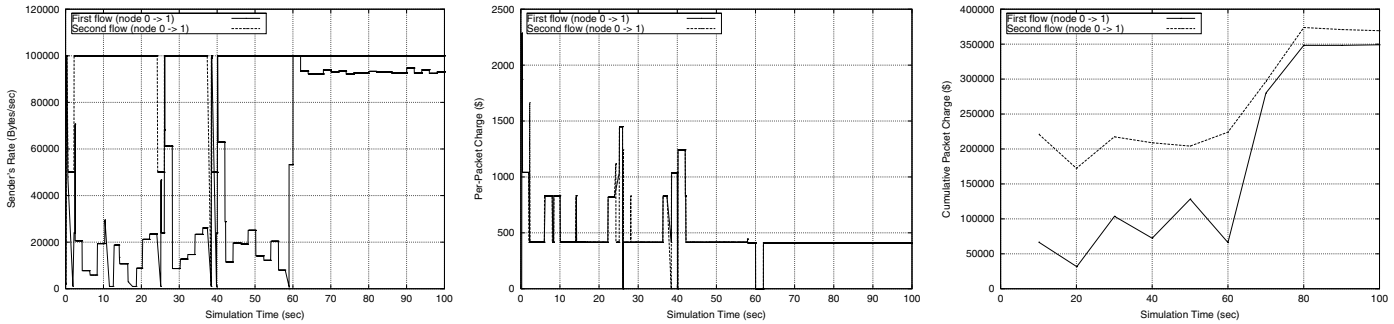

Figure 5: Multi-hop dynamic scenario. (a. Sending rate; b. Per-packet charge; c. Cumulative charge.)

[12] J. MacKie-Mason and H. Varian. Pricing the internet. In B. Kahin and J. Keller, editors, *Public Access to the Internet*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.

[13] S. Marti, T. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proc. 6th ACM/IEEE Annual Intl. Conf. on Mobile Computing and Networking (MobiCom 2000)*, Boston, Massachusetts, USA, Aug. 2000.

[14] P. Michiardi and R. Molva. Core: A collaborative reputation mechanism to enforce node cooperation in mobile ad hoc networks. In *Proc. 6th IFIP Conf. on Security Communications and Multimedia (CMS 2002)*, Portoroz, Slovenia, Sept. 2002.

[15] P. Michiardi and R. Molva. A game theoretical approach to evaluate cooperation enforcement mechanisms in mobile ad hoc networks. In *Proc. Workshop on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt'03)*, INRIA Sophia-Antipolis, France, Mar. 2003.

[16] D. Monderer and M. Tennenholtz. Optimal auctions revisited. *Artificial Intelligence*, 120(1):29–42, 2000.

[17] R. Myerson. Optimal auction design. *Mathematics of Operations Research*, 6:58–73, 1982.

[18] N. Nisan and A. Ronen. Algorithmic mechanism design. In *Proc. 31st ACM Symp. on Theory of Computing (STOC 99)*, pages 129–140, Atlanta, Georgia, USA, May 1999.

[19] S. H. Shah, K. Chen, and K. Nahrstedt. Dynamic bandwidth management for single-hop ad hoc wireless networks. In *Proc. IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom 2003)*, Dallas-Fort Worth, Texas, USA, Mar. 2003.

[20] V. Srinivasan, P. Nuggehalli, C. F. Chiasserini, and R. R. Rao. Cooperation in wireless ad hoc networks. In *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, San Francisco, California, USA, March-April 2003.

[21] A. Urpi, M. Bonuccelli, and S. Giordano. Modeling cooperation in mobile ad hoc networks: a formal description of selfishness. In *Proc. Workshop on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt'03)*, INRIA Sophia-Antipolis, France, Mar. 2003.

[22] H. Varian. Economic mechanism design for computerized agents. In *Proc. USENIX Workshop on Electronic Commerce*, New York, NY, USA, July 1994.

[23] W. Vickrey. Counter-speculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, Mar. 1961.

[24] S. Zhong, J. Chen, and Y. Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad hoc networks. In *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, San Francisco, California, USA, March-April 2003.

A Distributed Implementation of Sequential Consistency with Multi-Object Operations

Michel RAYNAL* Krishnamurthy VIDYASANKAR†

* IRISA, Université de Rennes, Campus de Beaulieu, 35 042 Rennes cedex, France

† Computer Science Dept, Memorial University, St. John's, Newfoundland, Canada A1B 3X5

raynal@irisa.fr vidya@cs.mun.ca

Abstract

Sequential consistency is a consistency criterion for concurrent objects stating that the execution of a multiprocess program is correct if it could have been produced by executing the program on a mono-processor system, preserving the order of the operations of each individual process. Several protocols implementing sequential consistency on top of asynchronous distributed systems have been proposed. They assume that the processes access the shared objects through basic read and write operations. This paper considers the case where the processes can invoke multi-object operations which can read or write several objects in a single operation atomically. It proposes a particularly simple protocol that guarantees sequentially consistent executions in such a context. The previous sequential consistency protocols, in addition to considering only unary operations, assume either full replication or a central manager storing copies of all the objects. In contrast, the proposed protocol has the noteworthy feature that each object has a separate manager. Interestingly, this provides the protocol with a versatility dimension that allows deriving simple protocols providing sequential consistency or atomic consistency when each operation is on a single object.

Keywords*: Asynchronous Distributed System, Message Passing, Multi-Object Operation, Object Manager, Shared Objects Memory Abstraction, Sequential Consistency.*

1 Introduction

Context: consistency criteria A concurrent object is an object than can be accessed concurrently by several processes. So, the consistency of concurrent objects is a crucial issue for the correct execution of multiprocess programs. A consistency criterion defines the correct behavior of a concurrent object. In read/write objects, such a criterion defines which value has to be returned when a read operation on the object is invoked by a process. The strongest (i.e., most constraining) consistency criterion is *atomic consistency* [13] (also called *linearizability* [9]). It states that a read returns the value written by the last preceding write, "last" referring to real-time occurrence order (concurrent writes being ordered). *Causal consistency* [3, 5] is a weaker criterion stating that a read does not get an overwritten value. Causal consistency allows concurrent writes; consequently, it is possible that concurrent read operations on the same object get different values (this occurs when those values are produced by concurrent writes). Other consistency criteria (weaker than causal consistency) have been proposed [1, 18].

Sequential consistency [11] is a criterion that lies between atomic consistency and causal consistency. Informally, it states that a multiprocess program executes correctly if its results could have been produced by executing that program on a single processor system, that is, if we can totally order its operations in such a way that (1) the order of operations in each process is preserved, and (2) each read gets the last previously written value, "last" referring here to the total order. The difference between atomic consistency and sequential consistency lies in the meaning of the word "last". This word refers to real-time when we consider atomic consistency, while it refers to a logical time notion when we consider sequential consistency (namely the logical time defined by the total order). The main difference between sequential consistency and causal consistency lies in the fact that (as atomic consistency) sequential consistency orders all write operations, while causal consistency does not require to order concurrent writes.

It has been shown that determining whether a given execution is sequentially consistent is an NP-complete problem [14, 19]. This has an important consequence as it rules out the possibility of designing efficient sequential consistency protocols (i.e., protocols that provide sequentially consistent executions and just these). This means that, in order to be able to design efficient sequential consistency protocols, additional constraints have to be imposed on exe-

Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)
1063-6927/04 $20.00 © 2004 **IEEE**

cutions [15][1]. One of these constraints, proposed in [15], called WW-constraint, requires that all write operations be ordered. Another constraint, called OO-constraint, requires that any pair of conflicting operations be ordered (two operations *conflict* if both are on the same object and one of them is a write). It has been shown that any WW-constrained (or OO-constrained) execution whose read operations are all legal (i.e., do not provide overwritten values) are sequentially consistent [15]. This suggests a general approach for designing sequential consistency protocols, namely, by combining two mechanisms, one providing the WW-constraint (or OO-constraint), and the other providing read legality. Actually, although they have not been designed with such a methodology in mind, the protocols proposed so far to implement sequential consistency can be divided up into two families, a big family including those based on the WW-constraint (e.g., [2, 4, 6, 15, 17]), and a small family including those based on the OO-constraint (e.g., [16]).

Motivation and content of the paper Atomic consistency and causal consistency have been defined assuming that an operation invoked by a process is a read or a write on a shared object. Atomic consistency has then been generalized, under the name *linearizability*, to objects of any type (e.g., stack, queue, etc.) [9].

Even if they can consider typed objects whose semantics is richer than the one provided by the basic read/write operations, linearizability and sequential consistency have been implicitly defined for concurrent object models where all operations are "unary", each operation being invoked on a single object (e.g., an *enqueue* operation involves a single object of the type queue). While this unary operation abstraction level is appropriate for low level object models such as the shared registers of a distributed shared memory, they are not suited for higher level applications that may require basic operations that encompass multiple objects (such as the merge of two queues). Such a higher abstraction level is characterized by the fact that an operation appears as being executed "atomically" whatever the number of the objects it involves. It is interesting to notice that the multi-object operation approach unifies and generalizes the previous consistency approaches. When we restrict the number of operations per process to one, a multi-object operation reduces to a transaction that "atomically" accesses (reads/writes) several data [7]. On another side, when each operation is reduced to a single read or write operation, the computation model reduces to the very classical shared memory model. This paper is on sequential consistency for concurrent objects whose type provides read/write operations that can span several objects.

A computation model where operations can span several objects, has been developed in [8] (where a new criterion, called *normality*, is also introduced). This model has been investigated in [14] where it is shown that the constraint-based approach developed in [15] for implementing sequentially consistent objects with unary operations can also be used when operations span several objects. To illustrate it, [14] presents a WW-constraint-based protocol providing sequential consistency in presence of multiple object operations.

This paper presents an OO-constraint-based protocol implementing sequentially consistent concurrent objects whose operations can possibly span several objects. It generalizes the OO-constraint-based protocol introduced in [16] for unary operations. It is important to note that the design of an OO-constraint-based protocol for multi-object operations is not immediate. Due to the very essence of the OO-constraint-based approach, a manager is associated with each object separately (there is no centralized manager), and consequently, an operation spanning several objects must involve their managers and coordinate them in an appropriate way. This makes the design of such an OO-constraint-based protocol a challenging issue.

Roadmap The paper is made up of five sections. The object model provided to upper layer applications is presented in Section 2. Then, Section 3 presents the OO-constraint-based protocol guaranteeing sequential consistency of the concurrent objects. This protocol considers that the underlying system is distributed and asynchronous. Section 4 discusses the versatility dimension of the protocol. Finally Section 5 concludes the paper.

2 Application Model

2.1 Processes, Objects, Actions and Operations

At the application level, a *concurrent system* (or *multi-process program*) consists of a finite set Π of *sequential processes* (denoted $p_1, p_2, ..., p_n$) that communicate through a finite set X of *shared objects* (or *concurrent objects*).

Each object $x \in X$ can be accessed by a read or a write action[2]. A write into an object defines a new value for the object; a read provides the value of the object. The execution of a write action that assigns the value v into object x is denoted $w(x)v$ (for simplicity, and without loss of generality, we assume all values written into an object are different). The execution of a read action of the object x that returns value v is denoted $r(x)v$.

[1] Considering additional constraints to design efficient consistency protocols has first been investigated in the database domain [10].

[2] Those read/write actions are usually called read/write operations. We deliberately use the term "action" to differentiate them from the "operations" invoked by the processes. An operation may involve read and write actions on several objects.

A process p_i executes operations sequentially. An *operation* is an "atomic package" made up of one or more read and write actions on shared objects. If the action $a(x)v$ (where a stands for w or r) appears in the operation op, we say "a belongs to op" (denoted $a \in op$). $R(op)$ and $W(op)$ denote the set of objects read and written, respectively, by the operation op. It is important to notice that, from a process point of view, there are only operations: the abstraction level provided to the processes is defined by the operations they can invoke.

2.2 History and Legality

The *local history* (or local computation) $\widehat{h}_i$ of p_i is the sequence of operations issued by p_i. Let h_i be the set of operations executed by p_i; the local history $\widehat{h}_i$ is the total order $(h_i, \rightarrow_i)$.

Definition 1 *An* execution history *(or history, or computation) $\widehat{H}$ of a concurrent system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that $H = \bigcup_i h_i$, and $op_1 \rightarrow_H op_2$ if:*
i) $\exists\ p_i\ :\ op_1 \rightarrow_i op_2$ *(in that case, $\rightarrow_H$ is called the* process-order *relation),*
or ii) $\exists (x, v) : w(x)v \in op_1$ *and* $r(x)v \in op_2$ *(in that case $\rightarrow_H$ is called the* read-from *relation),*
or iii) $\exists op_3\ :\ op_1 \rightarrow_H op_3$ *and* $op_3 \rightarrow_H op_2$.

As we can see, an execution history is defined at the operation abstraction level. Read and write actions induce precedence on operations but do not appear explicitly in a history. Two operations op_1 and op_2 are *concurrent* in $\widehat{H}$ if $\neg(op_1 \rightarrow_H op_2)$ and $\neg(op_2 \rightarrow_H op_1)$.

The legality concept is a key notion stating that no operation can obtain overwritten values. For the sake of simplicity, we assume in the following that there are initial fictitious operations that provide objects with their initial values.

Definition 2 *An operation op is* legal *if $\forall\ r(x)v \in op : \exists op_1$ such that:*
i) $op_1 \rightarrow_H op$ *(op_1 precedes op),*
ii) $w(x)v \in op_1$ *(op_1 is the operation that wrote v into x),*
iii) $\forall\ op_2$ *such that* $op_1 \rightarrow_H op_2 \rightarrow_H op : w(x) \notin op_2$ *(there is no overwriting operation).*

Definition 3 *An execution history $\widehat{H} = (H, \rightarrow_H)$ is* legal *if all its operations are legal.*

2.3 Sequential Consistency

Sequential consistency has been proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [11]. Formally, we define it in the following way. Let us remind that a *linear extension* $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{H} = (H, \rightarrow_H)$ is a topological sort of this partial order. (This means that: (i) $S = H$, (ii) $op_1 \rightarrow_H op_2 \Rightarrow op_1 \rightarrow_S op_2$ -$\widehat{S}$ maintains the order of all ordered pairs of $\widehat{H}$- and (iii) $\rightarrow_S$ is a total order.)

Definition 4 *A history $\widehat{H} = (H, \rightarrow_H)$ is* sequentially consistent *if it has a legal linear extension.*

This legal linear extension actually represents an "equivalent" execution produced by a fictitious mono-processor system.

2.4 The Constraint-Based Approach

As already indicated in the Introduction, the following two constraints have been introduced in [15]. An operation with at least one write action is called an *update* operation. Two operations *conflict* if both have actions involving the same object x and one of them writes x.

Definition 5 WW*-constraint. A history $\widehat{H}$ satisfies the* WW*-constraint if any pair of update operations are ordered under $\widehat{H}$.*

Definition 6 OO*-constraint. A history $\widehat{H}$ satisfies the* OO*-constraint if any pair of conflicting operations are ordered under $\widehat{H}$.*

The following theorems are stated and proved in [15] for the simple case where each operation is a read or write action. They have been extended in [14] to histories of multi-object operations.

Theorem 1 [15] *Let $\widehat{H} = (H, \rightarrow_H)$ be a history that satisfies the* WW*-constraint. $\widehat{H}$ is sequentially consistent if and only if it is legal.*

Theorem 2 [15] *Let $\widehat{H} = (H, \rightarrow_H)$ be a history that satisfies the* OO*-constraint. $\widehat{H}$ is sequentially consistent if and only if it is legal.*

3 An OO-Based Protocol for Multi-Object Operations

3.1 Underlying System Model

As indicated in Section 2.1, at the application level, the multiprocess programs that we consider are made up of n processes $p_1, \ldots, p_n$ sharing m concurrent objects (denoted $x, y, \ldots$). Differently, at the underlying level, the system is made up of $n + m$ sites, divided into n process-sites and m object-sites (hence, without ambiguity, p_i denotes both a process and the associated site; M_x denotes the site hosting and managing x). The sites communicate through reliable channels by sending and receiving messages. There are assumptions neither on the processing speed of the sites, nor

on message transfer delays. Hence, the underlying distributed system is asynchronous.

To be as modular as possible, and in the spirit of clients/servers architecture, the proposed solution uses communication only between a process and an object, and not between two processes or two objects. This feature makes the addition of new processes or objects into the system easier.

3.2 Underlying Principles and Local Variables

An object manager M_x is associated with every object x. In that sense there is no (statically defined) centralized locus of control[3]. Object values can be cached by processes and (as in other protocols) the last writer of an object is its *owner* until the value it wrote is overwritten or read by another process. The combination of value caching and object ownership allows processes to read cached values and update the objects they own for free (i.e., without communicating with the object managers). Basically, the synchronization required to satisfy the OO-constraint is ensured by the object managers, while the operation legality is ensured by a procedure called *reset_legality* that is executed by process p_i each time it gets new values in its cache.

The protocol is based on cached copies and an invalidation strategy. The basic principles that underlie its design can be formally captured with invariants stating the relations linking the different local variables managed by object managers and processes.

Local variables of an object manager Each site M_x manages the following local variables:

- C_x contains the last value of x (known to M_x),
- $owner_x$ contains a process identity or $\perp$. If $owner_x = i$, then p_i has the last value of x and it is the only process that can modify it. Differently, if $owner_x = \perp$, M_x has the last copy of x,
- $latest_x[1..n]$ is a boolean array, such that $latest_x[i]$ is true iff p_i has the last value of x.

These local variables are related by the following invariant:

$$(owner_x = i) \Rightarrow \big(latest_x[i] \bigwedge (\wedge_{j \neq i} \neg latest_x[j]) \big).$$

Local variables of a process Each process p_i manages the following local variables:

- $C_i[x]$ is a cache containing a copy of x,
- $legal_i[x]$ is a boolean variable that is true when p_i can legally read its cached copy of x,
- $latest_i[x]$ is a boolean variable that is true when p_i has the last value of x,
- $owner_i[x]$ is a boolean variable that is true when p_i can read and write x.

These local variables are related by the following invariant:

$$(owner_i[x] \Rightarrow latest_i[x]) \ \wedge \ (latest_i[x] \Rightarrow legal_i[x]).$$

The subtle part of a sequential consistency protocol is to allow a process to be able to read its cache $C_i[x]$ even when $latest_i[x]$ is false, i.e., to keep $legal_i[x]$ true as long as possible. If we always had $latest_i[x] = legal_i[x]$, the protocol would implement linearizability (see Section 4).

From a global point of view, the local variables of the object managers and the processes are linked by the following invariants:
$owner_i[x] \Rightarrow \big((\forall j \neq i : \neg owner_j[x]) \wedge owner_x = i \big)$,
and
$(latest_i[x] \wedge \neg owner_i[x]) \Rightarrow$
$\big((\forall j : \neg owner_j[x]) \wedge owner_x = \perp \big)$.

Let us notice that it is possible to have two processes p_i and p_j such $owner_i[x] \wedge legal_j[x]$. Then, if p_j locally reads $C_j[x]$, it gets a value that does not violate sequential consistency.

System initial state Initially, only M_x knows the initial value of x which is kept in C_x. So, we have the following initial values: $\forall p_i$: $\forall x$: $owner_i[x] = false$, $latest_i[x] = false$, $legal_i[x] = false$, $C_i[x]$ is undefined, $owner_x = \perp$, $latest_x[i] = false$.

3.3 The Protocol: Object Manager Side

The behavior of the manager M_x associated with the object x is depicted in Figure 1. It is made up of two statements describing what M_x does when it receives a message. The *write_req* and *read_req* messages are processed sequentially[4].

- M_x receives *write_req* (v) from p_i.
  In that case, p_i has issued an operation that involves a write action $w(x)v$. If x is currently owned by some process, M_x first downgrades this previous owner (lines 101-102). If x has no owner but some processes have its last value, the copies of x at these processes are downgraded from the "last value" point of view (lines 103-106). Then, M_x sets C_x and $owner_x$ to their new values (line 108), updates the boolean vector $latest_x$ accordingly (line 109), and sends back to p_i an ack message indicating it has taken the write action into account (line 110).
- M_x receives *read_req* from p_i.
  In that case, p_i has issued an operation involving an

[3] As it is the case when a global manager keeps and maintains a copy of each object (e.g., [2, 15]).

[4] Assuming no message is indefinitely delayed, it is possible for M_x to reorder waiting messages if there is a need to favor some processes or some operations.

```
upon reception of write_req (v) from p_i:
(101) if (owner_x ≠ ⊥) then send downgrade_owner_req (w, x) to p_owner_x ;
(102)                       wait downgrade_owner_ack (−) from p_owner_x
(103)                  else for each j ≠ i such that latest_x[j] do
(104)                         send downgrade_latest_req (x) to p_j ;
(105)                         wait downgrade_latest_ack from p_j
(106)                       end_do
(107) end_if;
(108) C_x ← v; owner_x ← i;
(109) latest_x[i] ← true; for each j ≠ i do latest_x[j] ← false end_do;
(110) send write_ack to p_i

upon reception of read_req from p_i:
(111) if (owner_x ≠ ⊥) then send downgrade_owner_req (r, x) to p_owner_x ;
(112)                       wait downgrade_owner_ack (v) from p_owner_x ;
(113)                       C_x ← v; owner_x ← ⊥
(114) end_if;
(115) latest_x[i] ← true;
(116) send read_ack (C_x) to p_i
```

Figure 1. Behavior of an Object Manager M_x

$r(x)$ action, and its cache does not include a legal copy of x. If x is currently owned by some process, M_x downgrades this previous owner and gets the last value of x (lines 111-114). Then, it updates $latest_x[i]$ accordingly (line 115), and sends the last value of x to p_i (line 116).

3.4 The Protocol: Process Side

The behavior of a process p_i is described in Figure 2. It is made of four statements that are executed atomically. We consider that an operation is either a simple read action ($r(x)$ at lines 216-220) or a multi-object operation *op* made up of the write actions $w(x_1)v_1, w(x_2)v_2, \ldots$ and the read actions $r(y_1), r(y_2), \ldots$ (lines 201-215). As indicated in Section 2, $W(op)$ and $R(op)$ denote the set of objects that *op* writes and reads, respectively. (If $x \in W(op) \cap R(op)$, the protocol needs only to consider $x \in W(op)$. Let us notice that, for the sake of simplicity, an operation that contains a single write action is considered as a multi-object operation.)

- p_i invokes a multi-object operation.
  In that case p_i launches in parallel two sets of statements, one for each object it wants to write, and one for each object it wants to read.
  - For each object x it wants to write, and for which it is not the current owner, p_i communicates with M_x to become the new owner of x (lines 201-207).
  - For each object y it wants to read, and for which it does not have the last value, p_i communicates with M_x to get its last value (lines 208-213).
  Then, p_i executes the procedure *reset_legality*.

- p_i invokes $r(x)$.
  If the local copy of x does not guarantee a legal read action (i.e., $legal_i[x]$ is false), p_i gets the last value of x from M_x (line 216), updates accordingly its context (line 217) and executes the *reset_legality* procedure (see next item).

- The *reset_legality* procedure.
  Although simple, this procedure is at the core of the protocol: it guarantees that all the values defined as legal in p_i's local cache will actually provide legal operations. To attain its goal, *reset_legality* cancels the legality of each object y for which p_i does not have the last value.

- p_i receives $downgrade\_owner\_req$ (type, x).
  In that case, p_i is the current owner of x. It relinquishes its ownership on x (line 222) and sends back the last value of x to M_x (line 224). If the ownership downgrading is due to the fact that another process becomes the new owner (in that case type=w), p_i has no longer the last value of x and $latest_i[x]$ is updated accordingly (line 223). In all cases, the local value of x at p_i remains legal (hence, $legal_i[x]$ is not modified).

- p_i receives $downgrade\_latest\_req$ (x).
  In that case, p_i is required to downgrade $latest_i[x]$ to false. It does it (line 225) and sends back an acknowledgment (line 226).

Liveness of multi-object operations As the protocol is written, it is possible that processes invoking multi-object operations on intersecting sets of objects enter a deadlock or a livelock configuration. To prevent such a scenario to

```
operation op (w(x_1)v_1, w(x_2)v_2, ...r(y_1), r(y_2), ... ):
        % The lines 201-207 and the lines 208-213 can be executed in parallel %
(201)   for each x_a ∈ W(op) do if (¬owner_i[x_a]) then send write_req (v_a) to M_{x_a};
(202)                                                   wait write_ack from M_{x_a};
(203)                                                   owner_i[x_a] ← true; latest_i[x_a] ← true;
(204)                                                   legal_i[x_a] ← true; reset ← true;
(205)                           end_if;
(206)                           C_i[x_a] ← v_a
(207)   end_do;
(208)   for each x ∈ R(op) do if (¬latest_i[x]) then send read_req to M_x;
(209)                                                   wait read_ack (u) from M_x;
(210)                                                   latest_i[x] ← true; legal_i[x] ← true;
(211)                                                   C_i[x] ← u; reset ← true;
(212)                   end_if
(213)   end_do;
(214)   if reset then reset_legality; reset ← false;
(215)   return {C_i[y] : y ∈ R(op)};

operation r(x):
(216)   if (¬legal_i[x]) then send read_req to M_x; wait read_ack (v) from M_x;
(217)                   latest_i[x] ← true; legal_i[x] ← true; C_i[x] ← v;
(218)                   reset_legality
(219)   end_if;
(220)   return (C_i[x])

procedure reset_legality:
(221)   for each y such that ¬latest_i[y] do legal_i[y] ← false end_do

upon reception of downgrade_owner_req (type, x):
(222)   owner_i[x] ← false;
(223)   if (type=w) then latest_i[x] ← false end_if;
(224)   send downgrade_owner_ack (C_i[x]) to M_x;

upon reception of downgrade_latest_req (x):
(225)   latest_i[x] ← false;
(226)   send downgrade_latest_ack to M_x
```

Figure 2. Behavior of a Process p_i

occur, a brute force solution consists in using a single token that is acquired just before line 201 and released after line 213. It is important to notice that the aim of such a token is not directly related to sequential consistency, but only to deadlock/livelock prevention. Less "brute force" solutions can be used. As an example, instead of a single token, it is possible to have a token per object: a process has then to request the tokens associated with the objects it wants to access, according to a predefined total order on the tokens. Interestingly, this "one token per object" approach favors parallelism.

3.5 Proof of the Protocol

Theorem 3 *The protocol described in Figures 1 and 2 implements sequential consistency.*

Proof Let us consider the execution of a multiprocess program on top of the protocol described in Figures 1 and 2. This execution generates a relation $\widehat{H} = (H, \rightarrow_H)$ on the set H of operations issued by the processes. Let us first note that, due to its definition, $\widehat{H}$ satisfies the *process-order* and *read-from* relations defined in Section 2. The proof is as follows. We claim that there is a relation $\widehat{H'} = (H, \rightarrow_{H'})$ such that:

- (i) $\rightarrow_H \subseteq \rightarrow_{H'}$,
- (ii) $\widehat{H'}$ is a partial order (i.e., $\widehat{H'}$ is an execution history),
- (iii) $\widehat{H'}$ satisfies the OO-constraint,
- (iv) $\widehat{H'}$ is legal.

If follows from the items (ii)-(iv) of the claim and Theorem 2 that $\widehat{H'}$ is sequentially consistent. Then, as $\widehat{H'}$ is sequentially consistent and includes the process-order and read-from relations of $\widehat{H}$ (item (i) of the claim), it follows that $\widehat{H}$ is sequentially consistent.

Proof of the claim.
For each object x let us define two relations denoted $\rightarrow_{ww(x)}$ and $\rightarrow_{rw(x)}$, respectively. ($\rightarrow_{ww(x)}$ captures write-write conflicts on x while $\rightarrow_{rw(x)}$ captures read-write conflicts; the write-read conflicts on x are captured by the read-from relation and the transitive closure with $\rightarrow_{ww(x)}$.)

Let us consider the sequence of the successive non-$\perp$ values of the variable $owner_x$ (line 108, Figure 1). Moreover, if while owning x a process p_i writes it several times (e.g., twice), let p_i appear the same number of times (e.g., twice) consecutively in this sequence. This sequence defines a total order $\rightarrow_{ww(x)}$ on the operations that write x, namely, $\forall \ell : W_\ell(x) \rightarrow_{ww(x)} W_{\ell+1}(x)$, where $W_\ell(x)$ denotes the ℓth write on x. Let us now consider an operation op with read action $r(x)v$. Due to the read-from relation, we have $w(x)v \rightarrow_H r(x)v$, that is, $W_\ell(x) \rightarrow_H op$. Let us define the relation $\rightarrow_{rw(x)}$ as follows $op \rightarrow_{rw(x)} W_{\ell+1}(x)$. Let $\widehat{H'} = (H, \rightarrow_{H'})$ where

$$\rightarrow_{H'} \;=\; \rightarrow_H \bigcup_x (\rightarrow_{rw(x)} \cup \rightarrow_{ww(x)}).$$

$\widehat{H'}$ satisfies the OO-constraint. This follows directly from the definition of $\rightarrow_{H'}$. For each object x, the write operations are totally ordered ($\rightarrow_{ww(x)}$ relation), and each read is inserted between the write $W_\ell(x)$ it is reading from and the write $W_{\ell+1}(x)$ that follows it in the $\rightarrow_{ww(x)}$ relation.

$\widehat{H'}$ is legal. We need to show only that all read actions are legal. The read actions in multi-object operations use the latest values and hence are legal. Let us consider $r_i(x)v$. There are two cases.
- p_i gets the value v from M_x. In that case, it executes the lines 216-220 of Figure 2 and M_x executes the lines 111-116 of Figure 1. If follows that p_i gets the last value v of x known by M_x. Consequently, the read does not get an overwritten value and is legal.
- p_i gets the value v directly from its cache $C_i[x]$ (in that case, $legal_i[x]$ is true). As initially p_i caches no value, it has previously executed a read or write on x (that set $legal_i[x]$ to true). Let $op_i(x)$ be this action. Moreover, let $op_i(y)$ be the last action issued by p_i before $r_i(x)$ that involved communication with an object manager. Between (and including) the actions $op_i(x)$ and $op_i(y)$ no execution of the *reset_legality* procedure by p_i has modified the value of $legal_i[x]$. It follows that the value of $C_i[x]$ is the last value of x known by M_x at the time $op_i(y)$ was invoked. Consequently, p_i does not read an overwritten value, and the read action is legal.

$\widehat{H'}$ has no cycle. First, we consider the history consisting of the actions and show it has no cycle. We observe that (1) the process-order relation is acyclic; (2) the relation on the read and write actions on each object defined by the OO-constraint is also acyclic. So, for the history to have a cycle, it is necessary for a read action to get a value from the "future" (as in the cycle defined by the following process-order edges $r_i(x)v \rightarrow_H w_i(y)u$ and $r_j(y)u \rightarrow_H w_j(x)v$ and the read-from relation). But, as we have seen in the legality proof, the *reset_legality* procedure forces a process p_i to falsify the legality of the cached values that have been updated, but does not provide it with new values. It follows that p_i cannot get "future" values.

Now consider any operation *op* of $\widehat{H'}$. All the "incoming" edges (read-from and $\rightarrow_{ww}$ edges) for the actions of an *op* can be considered as occurring at the beginning of *op*, and all "outgoing" edges as occurring at the end of *op*, since neither the ownership nor the latest value status is downgraded until the end of *op*. Then, any cycle involving *op* would yield a cycle involving actions, which is guaranteed not to exist, as shown above. *End of the proof of the claim.*

$\square_{Theorem\ 3}$

4 Protocol Versatility

This section briefly shows how the protocol can be simplified to implement linearizability, and for unary operations.

Linearizability Let us first consider the following modifications:

- Each pair of boolean variables $latest_i[x]$ and $legal_i[x]$ is merged in a single boolean $ll_i[x]$,
- The *reset_legality* procedure and all its invocations are suppressed.

Then, we get an invalidation-based protocol providing linearizability. When $owner_i[x]$ is true, p_i is the only process to have read/write access to x, and no other process can access x. When $owner_i[x]$ is false and $ll_i[x]$ is true, p_i has read access to x and no process can write x.

Unary operations Let us now modify the protocol as follows:

- No deadlock/livelock prevention mechanism (such as the tokens) is used,
- The operation $w(x)v$ is implemented by the lines 201-207, plus the line 214
- If a *downgrade_latest_req* (x_a) arrives at p_i while it is waiting for *write_ack* from M_{x_a} at line 202, then p_i executes that request rightaway.