A Causally Consistent Protocol for Distributed Shared Memory

Juan E. Navarro and Alvaro E. Campos
Departamento de Ciencia de la Computación
P. Universidad Católica de Chile

RT-PUC-DCC-97-02 December, 1997

Los Reportes Técnicos PUC-DCC pueden solicitarse a:
PUC-DCC Technical Reports can be requested from:

Secretaría de Investigación
Departamento de Ciencia de la Computación (143)
Pontificia Universidad Católica de Chile
Vicuña Mackenna 4860
Santiago 22, Chile

También disponible en la Internet en el URL:
Also available on the Internet under URL: `ftp://ftp.ing.puc.cl/pub/escuela/dcc/techReports/rt97-02.ps`

A Causally Consistent Protocol for Distributed Shared Memory

Juan E. Navarro and Alvaro E. Campos
Departamento de Ciencia de la Computación
P. Universidad Católica de Chile

December, 1997

Abstract

Distributed shared memory provides a virtual address space that is shared among processors in a distributed system. It allows application programmers to elude message passing and use the familiar shared-memory paradigm. To increase efficiency, DSM implementations replicate memory pages, introducing the problem of consistency. As fewer restrictions are imposed to the replicas, more efficient implementations are possible, but application programming becomes more difficult. Causal consistency is a model that offers a balance, by allowing efficient implementations without significantly increasing programming difficulty.

In this work, a causally consistent DSM protocol is proposed. The protocol requires that each time a node sends the contents of a page p, it piggybacks a list of all pages the receiver must invalidate. The list includes all pages written by operations unknown to the receiver, and that potentially causally precede the most recent write operation over p known by the sender. The protocol's main feature is less communication overhead than other causally consistent protocols in the literature. This behavior was confirmed running test applications on top of a simple simulator.

1 Introduction

In a distributed system, the only means of communication among processors is message passing. Since programming parallel applications in this kind of systems is considerably complex, some abstractions have been proposed to facilitate this task. They all aim to give the appearance that the programmer is working on a centralized environment, and can thus apply familiar programming paradigms. *Remote procedure calls* [7] hide communication complexity by allowing programs to call procedures that execute in remote machines, in a way that resembles calls to local procedures. Nevertheless, because there is no shared state among machines, the programmer must deal with problems such as passing pointers to complex data structures as parameters. Li [21] proposed a different approach, known as *distributed shared memory* (DSM), where a collection of processors that share no physical memory, do share a single, paged, virtual address space. Since user processes have the illusion that the memory is shared, they can communicate and synchronize through the memory.

Typical DSM implementations replicate memory pages to reduce memory latency, introducing the problem of consistency among copies. The first DSM system proposed *sequential consistency* (SC), a model to which programmers are used. Informally, SC requires that all processes agree in the same sequence of memory references [20]. Because SC is hard to implement efficiently and does not scale well, numerous alternative models have been proposed [24]. By imposing fewer restrictions to the replicas, more efficiency, scalability, and parallelism is achieved, but programming becomes a more difficult and less intuitive task.

Causal consistency [13] is a more relaxed model than SC. Causal consistency (CC) only requires that processes agree in the order of *potentially causally related* operations [19]. CC can be implemented much more efficiently than SC, and yet provides a good model for programming. In fact, many programs written for SC can run with no modifications under CC [2, 15].

This work proposes a causally consistent DSM protocol. Using a simple simulator and four test applications, the protocol is empirically compared against two other propositions in the literature. The purpose is to obtain a preliminary evaluation and gain insights that guide future work. Results are encouraging, since this protocol shows less communication costs than the others, though some more work is necessary, both in optimizing the protocol and in performing more elaborate comparisons.

Section 2 introduces the system model and notation, and formally defines causal consistency. The next section begins with a trivial protocol, which, through several refinements, leads to the proposed protocol. How to implement this proposition and how to extend it with explicit synchronization operations are the subjects of Sections 4 and 5. A comparison with previous works is done in Section 6, along with experimental results. The article concludes with a discussion of the results and a perspective of future work.

2 Basic concepts

This section defines the model and notation that are used throughout the paper. Part of the terminology is borrowed from [2] and [15].

2.1 System model

A distributed system consists of a set of nodes interconnected by a reliable communications network. Each node $\mathcal{P}$ is composed of private, local memory, a memory consistency manager P, and an application process (or, simply, a process) p, as shown in Figure 1. The processes in the system conform a distributed application. The manager's duty is to present a memory view to the local process as if there was a single memory (the DSM) shared by all processes in the system. The private memory is used, for this purpose, as a local cache of the virtual global memory.

Managers interact among them to keep the DSM consistent. A process interacts with the DSM (and, indirectly, with other processes) only through *read* and *write* operations over the local cache. Managers can catch any such operations and update the cache—if necessary to ensure consistency—before the operation is completed.

A write operation in which process p writes the value v into location x is denoted $w_p(x)v$. Similarly, $r_p(x)v$ denotes a read operation in which process p obtains v when reading x. The value and process can be dropped when they are irrelevant. For the sake of simplicity, and without loss of generality, it is assumed, as in [23], that for a given location no two operations write the same value into it: a location and a value uniquely identify a write operation.

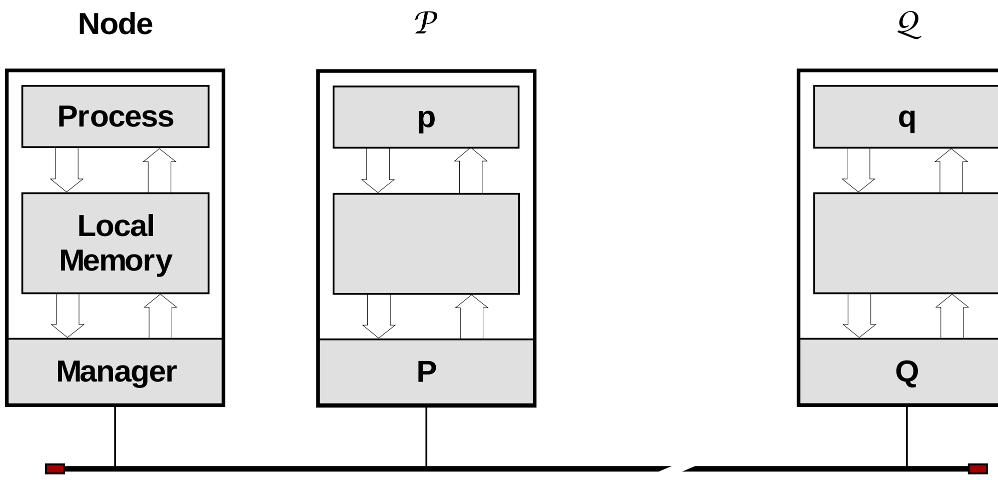

Figure 1: Distributed system model.

2.2 Potential Causality and Causal Consistency

Lamport [19] defines a partial order for events in a distributed system through a *happens before* relation. The sending and receipt of a message are used to relate events in different nodes: any event that happens before the sending of the message in the sender node also happens before any other event that happens after the receipt of the message in the receiver node. The same relation can be defined in terms of memory operations: if process p performs memory operation o_1 followed by $w_p(x)v$, and process q performs $r_q(x)v$ followed by another memory operation o_2, then it can be said that o_1 happened before o_2. Moreover, since o_2 could have happened as a consequence of o_1, this relation is also known as *potential causality*.

Formally, for any two memory operations o_1 and o_2:

- o_1 *precedes* o_2 *in program order* (written $o_1 \rightarrow o_2$), iff o_1 and o_2 take place in the same process, and o_1 occurs before o_2,
- o_1 *writes into* o_2 (denoted $o_1 \mapsto o_2$), iff $o_1 = w_p(x)v$ and $o_2 = r_q(x)v$, i.e., o_2 reads the value written by o_1, and
- o_1 *potentially causally precedes* o_2 (written $o_1 \rightsquigarrow o_2$), iff $o_1 \rightarrow o_2$, or $o_1 \mapsto o_2$, or there is some other operation o' such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$.

This is to say, $\rightsquigarrow$ is the transitive closure of the union of relations $\rightarrow$ and $\mapsto$, and defines a partial order for the operations, called *causal order*. Two operations o_1 and o_2 that are not potentially causally related (i.e., neither $o_1 \rightsquigarrow o_2$ nor $o_2 \rightsquigarrow o_1$) are said to be *concurrent*.

The *process history*, H_p, is the sequence of all operations performed by process p. The *execution history*, H, is the set of all processes's histories. A *view* $H|p$ of process p over execution history H is a sequence of all write operations in H and all read operations in H_p such that

- the order in H_p is respected, that is, if o_1 precedes o_2 in H_p then o_1 precedes o_2 also in $H|p$, and

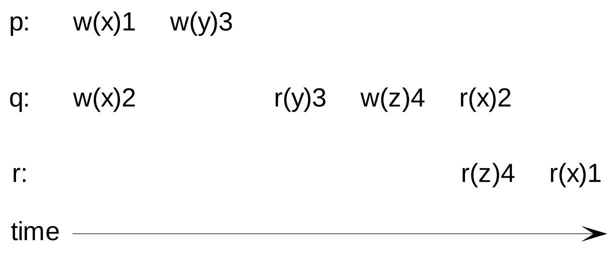

Figure 2: History of an execution.

- every read operation $o_1 = r_p(x)v$ reads the value written by the last write operation o_2 that writes x and precedes o_1 in $H|p$ (if such o_2 does not exist, then o_1 can read any value).

Note that, given an execution history, a process can have more than one view as shown in the example below.

An execution history H is *causally consistent* if there exists a *causal view* for each process. A view $H|p$ is causal if it respects causal ordering, i.e., if o_1 precedes o_2 in $H|p$ whenever $o_1 \rightsquigarrow o_2$. A system or protocol provides causal consistency, if it allows only causally consistent execution histories.

It is also said that two processes, p and q, *see two write operations o_1 and o_2 in the same order*, if it is possible to build respective views, $H|p$ and $H|q$, in which o_1 and o_2 appear in the same relative order. Thus, in a causally consistent execution, potentially causally related write operations are seen in the same order by all processes, but concurrent write operations can be seen in different order. In contrast, under sequential consistency, there must exist one view for each process such that all write operations appear in the same order in every view.

2.3 An example

Figure 2 represents the history, H, of an execution, where the following relations hold: $w_p(x)1 \rightarrow w_p(y)3$; $w_p(y)3 \rightsquigarrow w_q(z)4$ (since $w_p(y)3 \mapsto r_q(y)3 \rightarrow w_q(z)4$); $w_p(x)1 \rightsquigarrow w_q(z)4$ (by transitivity); $w_q(x)2 \rightarrow w_q(z)4$; $w_p(x)1$ and $w_q(x)2$ are concurrent; and $w_p(y)3$ and $w_q(x)2$ are concurrent.
Two views for r are

$$\begin{aligned} H|r &= w_p(y)3 \;\; w_q(z)4 \;\; r_r(z)4 \;\; w_p(x)1 \;\; r_r(x)1 \;\; w_q(x)2 \\ H|r' &= w_q(x)2 \;\; w_p(x)1 \;\; w_p(y)3 \;\; w_q(z)4 \;\; r_r(z)4 \;\; r_r(x)1 \end{aligned}$$

Only the latter is causal (and is the only causal view that can be built for r). Causal views for p and q are

$$\begin{aligned} H|q &= w_p(x)1 \;\; w_q(x)2 \;\; w_p(y)3 \;\; r_q(y)3 \;\; w_q(z)4 \;\; r_q(x)2 \\ H|p &= w_p(x)1 \;\; w_q(x)2 \;\; w_p(y)3 \;\; w_q(z)4 \end{aligned}$$

Other causal views for p and q are obtained by swapping the concurrent writes $w_q(x)2$ and $w_p(y)3$. Even though $w_p(x)1$ and $w_q(x)2$ are concurrent, it is not possible to swap them in $H|q$, because the operation $r(x)$ that q performs yields 2, which would not be the last value previously written over x according to the view.

Therefore, given that there are causal views for each process, H is causally consistent. The reader can easily check that H is not sequentially consistent.

3 Providing Causal Consistency

In this section a causally consistent protocol for DSM is developed starting from a simple solution, inspired in causal message broadcast [6], which is to messages what causal consistency is to memory operations. The simple solution is refined to obtain an efficient protocol based on virtual memory management techniques. Advantage is taken from the fact that messages have queue semantics, while memory operations have overwrite semantics. In other words, if a process accepts messages infinitely often, it will receive all messages sent to it, but if it reads a memory location infinitely often, it cannot be assured that it will read all values written to that location [2].

3.1 A simple protocol

Causal consistency requires that write operations that are potentially causally related are seen in the same order by all processes. This is the same requirement that causal broadcast (CB) imposes over the order in which processes see broadcast messages. A trivial protocol to achieve CB is to piggyback all previously received messages, in the order they were received, each time a new message is sent. The receiver first processes the piggybacked messages not yet received at the node, and then the new message.

If a write operation over a shared variable is thought of as a broadcast message, then a simple (and quite impractical) protocol for causally consistent DSM is directly obtained from the trivial CB protocol:

1. Write operations that have been carried out in a node $\mathcal{P}$ are recorded in a write list W_P, in the order they were carried out. To carry out a local operation $w_p(x)v$ means to append a tuple $(P\!:\!wsn, x, v)$ to the list W_P, in which wsn is a sequence number assigned by P to each local write (thus, $P\!:\!wsn$ uniquely identify a write operation). For the moment, consider that appending the tuple is atomically done with the occurrence of the operation. To carry out a remote operation $w_q(x)v$ means to record it at the end of the list, and to perform it in the local cache, (that is, to set the value of x to v), all in one atomic action.

2. Each time a local write operation $w_p(y)u$ occurs at node $\mathcal{P}$ and is consequently carried out by P, the manager broadcasts the operation, piggybacking all operations that have been previously carried out. More formally, P broadcasts the tuples that represent such operations, and more precisely, P sends the whole list W_P.

3. When another manager receives the message, it carries out, in order, every enclosed operation that has not yet been carried out at the node.

On one hand, the order of the operations in the list is the order in which they were performed in the local cache; hence, it is the order in which the local process sees the operations. On the other hand, the order is consistent with potential causality. Formally stated, this means that a causal view in which write operations appear in the order they have in W_P can be built for local process p [25]. Consequently, this simple protocol provides causal consistency.

3.2 Paging and coherence

To transform the previous protocol into a practical one, virtual memory management techniques are applied to catch memory accesses that could violate consistency. The address space is divided in pages, which become the basic unit of consistency. Nevertheless, since applications work at a different level of data granularity (i.e., variables, not pages), paging produces *false sharing* [8] when two unrelated variables share the same page. If simultaneous writes into such unrelated variables are allowed, then, sooner or later, any copy of the page should reflect both changes. The only way to manage a situation like that, is actually knowing exactly what values were written into exactly what positions, but conventional paging hardware only allows consistency managers to know that *some* position of a certain page is about to be written with *some* value. It is possible to find out by indirect means the necessary information to keep track of changes to each portion of a page, and to keep copies updated by merging the changes. As this approach complicates the protocol and adds significant overhead [16], it was left for future study.

The alternative chosen in this work is simply not to allow simultaneous changes to a page. In other words, write operations over the same page are totally ordered, which allows to say that each write produces a new *version* of the page. The resulting model is stricter than pure causal consistency; not only potentially causally related write operations, but also write operations over the same page, are seen in the same order by all processes. *Coherence* [11] (also called *cache consistency* [12]) requires that accesses to the same memory position are sequentially consistent. The resulting model is thus named *page-coherent causal consistency*, since it is simultaneously coherent, in a page basis, and causally consistent.

Page-coherence is achieved using page ownerships. At any given instant, each page is owned by exactly one manager, whose local process is the only one that can write into the page. Before a process writes into a non-owned page, the manager must obtain the ownership from the current owner, and the protocol must ensure that all previous operations over the page are carried out at the node; for the moment, it does not matter how. In this way, every write operation is performed over the most recent version of the page, which allows consistency managers to treat pages as big variables, that is, to consider a write operation $w_p(x)v$ as $w_p(\mathcal{X})\mathcal{V}$, where $\mathcal{X}$ is the page where x resides, and $\mathcal{V}$ is the contents of $\mathcal{X}$ right after performing the operation. Therefore, to record $w_p(x)v$, managers actually record the tuple $(P\!:\!wsn, \mathcal{X}, \mathcal{V})$ instead of $(P\!:\!wsn, x, v)$. The simple protocol remains correct (and impractical too, because it is either expensive or impossible to determine $\mathcal{V}$).

To locate the owner of a page, each manager P keeps a hint of its probable owner: $\mathtt{PO}_P[\mathcal{X}]$ is the last owner of $\mathcal{X}$, as far as P knows. If P wants the ownership of $\mathcal{X}$, it sends a WRITE request to $\mathtt{PO}_P[\mathcal{X}]$, say Q. If Q is the current owner of $\mathcal{X}$, it grants the page ownership to P. Otherwise, Q forwards the request to $\mathtt{PO}_Q[\mathcal{X}]$, and so on, until, eventually, the request reaches the current owner, who directly replies to P. All managers in this chain update their hint of $\mathcal{X}$'s probable owner to P. In the worst case, if N is the number of nodes, $N-1$ messages are required to find the owner of a page. Nevertheless, the cost is amortized in several ownership requests: for K requests for the same page, the total number of messages is $O(N + K\log N)$ [22]. The cost is further reduced when the number of nodes that contend for the same page is small. In fact, it is not unusual that only two nodes contend for a page—as experiments in this work and in [22] have shown—in which case only one message is required to locate the owner.

3.3 Making the protocol lazy

There is no reason to broadcast immediately every local write operation; operations can be sent to other node only when needed at that node. Moreover, operations received at a node need not be immediately performed in the local cache; performing a remote operation can be deferred until the local process needs access to the page. These facts suggest the use of an invalidation policy (instead of an update one) to substantially reduce the amount of transmitted data: when an operation is received, instead of performing it locally, the page is invalidated. Hence, it is not necessary to send values in piggybacked operations. If the local process attempts to read an invalid page, a read fault is triggered, and only then the contents of the page are retrieved and stored in the local cache. In other words, a remote operation is carried out in two stages: first, it is added to the write list and, later, it is performed in the local cache. Nevertheless, as far as the process is concerned, both actions still appear to be atomic, since the process cannot see the intermediate state. It is said that manager P has *partially carried out* a remote write operation, if P has added the operation to W_P but not yet performed it in the cache, and that P has *totally carried out* a remote write operation if the operation has been both added to W_P and performed. Local write operations are said to be totally carried out since the instant they are performed by the local process.

Assume that the tuple that represents the operation $w_p(\mathcal{X})\mathcal{V}$ is piggybacked to Q, thus causing the invalidation of $\mathcal{X}$ at node $\mathcal{Q}$ (recall that $\mathcal{V}$ is not sent). Later, if process q tries to read the page, the value $\mathcal{V}$ should be retrieved if exactly the same results as with the simple protocol are desired. The inconvenience is that $\mathcal{V}$ must be stored in some node, and kept even if $\mathcal{X}$ is overwritten with a newer value in that node. However, it should be noted that, for q, reading $\mathcal{V}$ or any newer version of the page is consistent. Moreover, $\mathtt{PO}_Q[\mathcal{X}]$, the last owner of $\mathcal{X}$ known by Q, must have a version of $\mathcal{X}$ at least as new as the needed one. Consequently, Q asks $\mathtt{PO}_Q[\mathcal{X}]$, say R, for a copy of $\mathcal{X}$, by sending a READ request for the page. Since the idea is to avoid storing old copies of pages, R sends back to Q its current copy. Two cases must be considered, depending on the value held by R's copy.

1. The value is $\mathcal{V}$. In this case, nothing else should be done.

2. The value is $\mathcal{U} \neq \mathcal{V}$. Then, the copy holds a newer version of the page. Let $w_s(\mathcal{X})\mathcal{U}$ be the operation that wrote such value into $\mathcal{X}$. This case is handled by making as if in that very moment Q had received the broadcast message corresponding to $w_s(\mathcal{X})\mathcal{U}$, that is, a list that includes all write operations that potentially causally precede $w_s(\mathcal{X})\mathcal{U}$ and the operation itself. Thus, R sends to Q its write list (excluding values), up to $w_s(\mathcal{X})\mathcal{U}$, along with the value $\mathcal{U}$. Q sets its cache of $\mathcal{X}$ to the newly received value, appends to W_Q all operations in the message that had not been previously received, and invalidates all pages written by those operations (with the exception of $\mathcal{X}$, since its value has just become valid).

Note that it is always safe to take the actions of case 2, even if $\mathcal{U} = \mathcal{V}$. Also note that READ requests are not forwarded to the current owner, even if the requested manager does not own the page.

WRITE requests are treated as READ requests, except that they must be forwarded until they reach the current owner; servicing the request implicitly grants the ownership of the page. In this way, what was previously assumed in Section 3.2 is satisfied: when the requester becomes the new owner of the page, all preceding operations over the page have been carried out at the node. Moreover, the last operation is totally carried out, and, since the whole page is written, the effect is the same as if all previous operations had been totally carried out, even if they had been only partially carried out.

With these improvements, values are not needed in write lists either. For a write operation $w_p(\mathcal{X})\mathcal{V}$, only P, the write sequence number, and $\mathcal{X}$ are recorded. Furthermore, the protocol needs write-list tuples only to determine which pages managers have to invalidate when receiving a copy of a page.

3.4 Intervals

Assume that the first three operations that process p executes are $w_p(\mathcal{X})\mathcal{V}'$, $w_p(\mathcal{X})\mathcal{V}$, and $w_p(\mathcal{Y})\mathcal{U}$. According to the protocol so far, they should be recorded in W_P as $(P\!:\!1,\mathcal{X})$, $(P\!:\!2,\mathcal{X})$, and $(P\!:\!3,\mathcal{Y})$. If no remote operation is received in between, then the three tuples will remain contiguous in W_P. Suppose that, next, P receives a READ request for $\mathcal{X}$ from another manager, Q. Then, P should answer with $\mathcal{V}$, $(P\!:\!2,\mathcal{X})$, and $(P\!:\!1,\mathcal{X})$ (respectively, the current value of $\mathcal{X}$ the last operation on $\mathcal{X}$ that P has totally carried out, and all operations previously recorded in W_P). Would it hurt to send all three operations instead of only the first two? The only difference would be Q's invalidation of an additional page, $\mathcal{Y}$, action that might affect performance or parallelism, but by no means correctness. In that case, the three tuples can be grouped so that all managers always consider the three operations as one atomic block: they are transmitted together and (partially) carried out together. When the block is received by, say R, it is added to W_R, and pages $\mathcal{X}$ and $\mathcal{Y}$ are invalidated. Hence, it does not matter anymore the exact order of the operations within the block, nor how many of them there are. The only necessary information is the set of pages written. In the case of the example, the three tuples can be joined into one structure containing P, the interval's sequence number, and the set $\{\mathcal{X},\mathcal{Y}\}$. Such structure is called an *interval record* because it includes the operations that occurred in some interval of the process's execution.

Although space is saved with this improvement, that is not the intended objective, but to reduce the number of write faults that managers have to handle. In effect, to implement the protocol without grouping operations, managers have to know the exact order in which pages are written by the local process. To achieve that goal, all pages, except the most recently written, must be write-protected. When the process attempts to write another page, a write fault is triggered, which allows the manager to record the unprotected page in W_P; then, that page is reprotected, and the page to be written is unprotected. Under this approach almost every write produces a fault. The only exceptions are the second and subsequent consecutive writes over the same page. Using interval records instead, the process's execution is divided in intervals. At the beginning of an interval, all pages are protected. The first time a page is written in the interval, it is added to a set of dirty pages and unprotected. When the interval is closed and a new one begun, the set of dirty pages is recorded in the write list. Only then, dirty pages are reprotected and the dirty set is emptied.

Even though the expected number of unnecessary invalidations increases as intervals include more operations, the decision was to make intervals long, in the belief that it pays off to reduce the faults. The question is how far to go, or where the limit is. As there is no need to close the current interval while the node does not communicate with the external world, the only events that could force finishing the interval are communication events. More precisely, there are two cases to consider: servicing a request, and processing a reply.

First, assume that P receives a request, either READ or WRITE, for page $\mathcal{X}$. P must reply with the current value of $\mathcal{X}$, and all interval records in W_P up to the one that contains the last operation over $\mathcal{X}$ totally carried out (how the concepts of partially and totally carried out operations change when using interval records is explained below). If the current interval, say $P\!:\!i$, does not contain any operation on $\mathcal{X}$, there is no reason to close it, since the reply does not depend on it. But if $\mathcal{X}$ was written during $P\!:\!i$, then the reply will contain that interval, and hence it should be closed. Were it not closed, then there is no guarantee that all operations in the interval are treated as an atomic block.

Second, assume that P receives a reply to a request. Let $P\!:\!i$ be the current interval at $\mathcal{P}$, and L be the list of interval records that must be added to W_P, that is, the list of those intervals in the reply that are not yet in P's write list. If L is added *to the end* of W_P, as the protocol so far dictates, then $P\!:\!i$ should be closed. The reason is that the next write operation executed by p cannot be included in $P\!:\!i$, since it cannot be considered to be consecutive with the operations in that interval: all operations represented by L interpose. Nevertheless, since interval $P\!:\!i$ has been known by no other manager (otherwise, it would have been closed), it is known that no operation in $P\!:\!i$ potentially causally precedes any operation in L's intervals. Thus, informally, P can legally behave as if it saw the latter operations before the former. In terms of the protocol, that means that P inserts L right before the current interval, which allows P to keep $P\!:\!i$ open.

In short, the current interval needs to be closed only when a request for a dirty page is serviced; records from remote intervals are always inserted before the current interval. The implementation shown in Section 4 attains the same effect by appending remote records at the end of the write list, but delaying the recording of the current interval until it is closed.

It can be thought that write lists only keep track of the last write operation over each page written during an interval. Taking advantage of the overwrite semantics of memory operations, all other operations are not recorded, since the value they write is overwritten before being made visible to remote processes. Consequently, the concepts of partially and totally carried out operations still apply, but only to the recorded operations, i.e., the last one to each page in each interval.

3.5 Avoiding unnecessary transmissions of interval records

When a manager P receives a reply to a request, it receives a list of interval records. For each interval, it has to be determined whether it is already present in W_P. If it is not, the record is added to W_P; otherwise, the record is discarded. Sending records that will be discarded by the receiver can be avoided if P reports, in the request message, what intervals it has already recorded. Such reports can be made quite compact, since it is enough to inform about the last interval from every manager recorded in W_P. If i is the sequence number of the last interval created by Q and stored in W_P, then W_P contains all intervals created by Q, from 1 to i [25].

3.6 Why the protocol is correct

Navarro [25] formally proves that the protocol is correct, i.e., that it provides causal consistency. The proof is based on building a causal view for each process as follows. Once the application finishes (or at any instant), let every manager complete its write list with the remote intervals not yet received. It can be thought as if every manager receives from each other manager the whole remote write list, and handles it as usual. Let $I(o)$ be the interval where write operation o occurred. For write operations o_1 and o_2, it is said that $o_1 <_P o_2$ if either $I(o_1)$ is recorded before $I(o_2)$ in W_P, or $I(o_1) \equiv I(o_2)$ and $o_1 \rightarrow o_2$. Then, $<_P$ defines a total order for all write operations. By inserting local read operations in the appropriate places (according to their relative order with respect to local write operations), a view for p is obtained. Finally, it is shown that the view is causal by proving that $\forall P : o_1 \rightsquigarrow o_2 \Rightarrow o_1 <_P o_2$.

4 Implementation

In summary, the history of a process p is partitioned into intervals. Each interval is uniquely identified by a pair $P\!:\!i$, where i is the interval's sequence number. Let S be the set of pages

```
Initialize
  Dirty_P := φ
  W_P := φ
  for each Manager Q
    MAX_P[Q] := 0
  for each Page 𝒳
    PO_P[𝒳] := InitialOwner(𝒳)
    κ_P[𝒳] := NIL
    λ_P[𝒳] := NIL
    if P = InitialOwner(𝒳)
      State[𝒳] := OWNED
      SetReadOnly(𝒳)
    else
      State[𝒳] := INVALID
      SetNoAccess(𝒳)
```

Figure 3: Initialization at node $\mathcal{P}$.

written during the interval $P\!:\!i$. When the interval is closed, an interval record $(P\!:\!i, S)$ is added to P's write list, W_P, and interval $P\!:\!i+1$ is begun.

Each manager P keeps a vector MAX_P such that $\text{MAX}_P[Q]$ is the maximum sequence number of the interval records created by Q that are stored in W_P. P also keeps track of the state of every page $\mathcal{X}$ at the node, in $\texttt{State}_P[\mathcal{X}]$. The state can be one of three: OWNED (the page is owned at the node), FOREIGN (the page is not owned, but its cache contains a valid copy), or INVALID (the page's cache is not valid for the local process). The manager uses the page-protection mechanism so that a read fault is triggered when the local process reads an INVALID page, and a write fault is triggered when the process writes a FOREIGN page or an OWNED page not previously written during the current interval.

Additionally, for each page $\mathcal{X}$, P maintains two pointers to interval records in W_P, $\kappa_P[\mathcal{X}]$ and $\lambda_P[\mathcal{X}]$. $\kappa_P[\mathcal{X}]$ points to the interval in W_P with the last operation over $\mathcal{X}$ totally carried out, whereas $\lambda_P[\mathcal{X}]$ points to the last interval record in W_P that contains page $\mathcal{X}$, i.e., to the last operation over $\mathcal{X}$ either partially or totally carried out. The local process can read a page only if the last operation over the page in W_P is totally carried out; while it is only partially carried out, the value the page should have is not known. In other words, $\mathcal{X}$ is invalid at $\mathcal{P}$ iff $\kappa_P[\mathcal{X}] \not\equiv \lambda_P[\mathcal{X}]$.

The initialization pseudocode, run by manager P, is shown in Figure 3. $\texttt{Dirty}_P$ is a set used to keep track of pages written during the current interval. `InitialOwner` is a globally known function that defines the initial owner of each page. `SetReadOnly`, `SetNoAccess` and `SetReadWrite` (used later) are system calls for setting the access-protection of a page.

4.1 Handling of read faults

Figure 4 shows the actions to handle a read fault. Chronologically, the events are the following. P sends a READ request to the probable owner of the page, in this case Q, enclosing the MAX_P vector and the page identifier, $\mathcal{X}$. Upon receiving the request, Q closes its current interval if $\mathcal{X}$ was written during that interval. Closing the interval means appending a new record to W_Q, where the set of pages written during the interval is stored, and reprotecting all dirty pages. Then, Q must reply with the contents of $\mathcal{X}$ in its cache, denoted by $\widehat{\mathcal{X}_Q}$, and a list of interval

```
on READ-FAULT(𝒳)
  send ⟨READ, MAX_P, 𝒳⟩ to PO_P[𝒳]
  receive ⟨D, 𝒱⟩
  Process(D)
  𝒳̂_P := 𝒱
  κ_P[𝒳] := λ_P[𝒳]
  State_P[𝒳] := FOREIGN
  SetReadOnly(𝒳)

Process(D)
  for each Record ϱ = (R:i, S) in D
    W_P := W_P + ϱ
    MAX_P[R] := i
    for each Page 𝒴 in S
      State[𝒴] := INVALID
      SetNoAccess(𝒴)
      λ_P[𝒴] := ϱ
      if R more recent than PO_P[𝒴]
        PO_P[𝒴] := R
```

(a)

```
on RECEPTION of ⟨READ, MAX_P, 𝒳⟩ from P
  if 𝒳 in Dirty_Q then CloseInterval
  send ⟨D_Q(𝒳), 𝒳̂_Q⟩ to P

CloseInterval
  increment MAX_Q[Q]
  W_Q := W_Q + (Q:MAX_Q[Q], Dirty_Q)
  for each Page 𝒴 in Dirty_Q
    SetReadOnly(𝒴)
  Dirty_Q := φ
```

(b)

Figure 4: Read fault at $\mathcal{P}$. (a) Actions of manager P. (b) Actions of other manager, Q.

records, $D_Q(\mathcal{X})$, called a *disclosure* for $\mathcal{X}$.

Note that $\widehat{\mathcal{X}_Q}$ is the value written by the last operation over $\mathcal{X}$ totally carried out by Q. $\kappa_Q[\mathcal{X}]$ points to the interval where it occurred. Therefore, $D_Q(\mathcal{X})$ is the prefix of W_Q that ends in $\kappa_Q[\mathcal{X}]$, filtered so that only the intervals that are not yet in W_P are sent. The filtering is easily done, thanks to having received MAX_P: only those intervals $R{:}i$ in the prefix such that $i > \text{MAX}_P[R]$ are sent; the disclosure can be empty.

P appends the disclosure to W_P and updates λ_P and MAX_P accordingly. Also, each page in the disclosure is invalidated, and its probable owner updated if the previous hint is known to be obsolete (the next section explains how it can be determined whether the hint is obsolete). Finally, P updates $\mathcal{X}$ and its attributes.

It should be noted that Q replies P's READ request even if it is not the owner of $\mathcal{X}$, and even if the page is invalid at the node. If the page is obsolete for process q, it does not mean that it is also obsolete for p. If P is requesting Q, it is because it is not yet aware of the write operations that invalidated the page at $\mathcal{Q}$. If it were aware, then P would not have requested Q.

4.2 Handling of write faults

When a write fault over non-owned page $\mathcal{X}$ is trapped by manager P, it sends a WRITE request to the probable owner of the page, say Q, as shown in Figure 5(a). If Q is not the current owner of $\mathcal{X}$, then it forwards the request to $\texttt{PO}_Q[\mathcal{X}]$, and sets the probable owner to P, as shown in Figure 5(b). Otherwise, it follows the same actions as in the case of a READ request to build the reply, but also, since it is granting the ownership of $\mathcal{X}$ to P, it downgrades the page to FOREIGN and sets $\texttt{PO}_Q[\mathcal{X}]$ to P. P processes the reply in the same way as before. In addition,

```
on WRITE-FAULT(𝒳)
  if State_P[𝒳] ≠ OWNED
    send ⟨WRITE, MAX_P, 𝒳, P⟩ to PO_P[𝒳]
    receive ⟨D, 𝒱⟩
    Process(D)
    𝒳̂_P := 𝒱
    κ_P[𝒳] := λ_P[𝒳]
    State_P(𝒳) := OWNED
    PO_P[𝒳] := P
  SetReadWrite(𝒳)
  Dirty_P := Dirty_P ∪ 𝒳
```

(a)

```
on RECEPTION of ⟨WRITE, MAX_P, 𝒳, P⟩
  if State_Q[𝒳] = OWNED
    if 𝒳 in Dirty_Q then CloseInterval
    send ⟨D_Q(𝒳), 𝒳̂_Q⟩ to P
    State_Q[𝒳] := FOREIGN
  else
    send ⟨WRITE, MAX_P, 𝒳, P⟩ to PO_Q[𝒳]
  PO_Q[𝒳] := P
```

(b)

Figure 5: Write fault at $\mathcal{P}$. (a) Actions of manager P. (b) Actions of other manager, Q.

the page is upgraded to OWNED, added to the $\texttt{Dirty}_P$ set, and unprotected. If the write fault is over an OWNED page, then no communication is necessary, because the page was protected with the sole purpose of knowing whether it was going to be written during the current interval.

There are two sources of information that give hints to a manager P about the owner of a page $\mathcal{X}$. One is the reception of a WRITE request for the page from, say, Q; in that case, given that Q is going to be the owner of the page, P points the probable owner to Q. If, later, the local process p wants to write into the page, the request is sent to Q. For this approach to work, Q must enqueue any requests it receives for $\mathcal{X}$ while it is waiting the ownership of the page.

The second source is the reception of a disclosure. Let $\texttt{PO}_P[\mathcal{X}] = Q$ and let $(R{:}i, \{\mathcal{X}\})$, with $R \neq Q$, be the last record in a disclosure. Then, after processing the disclosure, P will know that R had the ownership of $\mathcal{X}$ during the interval $R{:}i$. Should P update $\texttt{PO}_P[\mathcal{X}]$ with R? In other words, if the current value in $\texttt{PO}_P[\mathcal{X}]$ had just been set to Q due to the reception of a WRITE request from Q, then, who owned $\mathcal{X}$ more recently, Q or R? In this case the answer is Q; $\texttt{PO}_P[\mathcal{X}]$ was set to Q while Q was requesting the page, that is, even before Q actually got the ownership. Assume that Q's current interval was $Q{:}j$ at the moment it sent the request for $\mathcal{X}$. Then, any interval $R{:}i$ including $\mathcal{X}$ that P receives is known to be older than $Q{:}j$, until P receives $Q{:}j$ Therefore, to change $\texttt{PO}_P[\mathcal{X}]$, P does not have to take into account the intervals where $\mathcal{X}$ was written until it receives an interval record created by Q and containing $\mathcal{X}$. From then on, and until P replies or forwards another WRITE request for $\mathcal{X}$, the information that can be extracted from disclosures about $\mathcal{X}$'s ownership must be used to refresh $\texttt{PO}_P[\mathcal{X}]$. That is what the last two lines in Figure 4(a) do.

5 Synchronization

Because synchronization is fundamental to parallel programs, it is convenient to provide explicit synchronization operations. Such operations are used to control the dependencies among parallel tasks and, therefore, they contribute to the causal ordering of memory operations. For instance, each memory operation that occurred before a barrier potentially causally precedes every other memory operation that occurred after the barrier. By augmenting the relation $\mapsto$, such additional ordering restrictions can be modeled. Regarding locks and (non-reusable) barriers, it can be said that $o_1 \mapsto o_2$ iff

- o_1 is a write operation ($w_p(x)v$) and o_2 is a read operation ($r_q(x)v$) over the same location that reads the value written by o_1, or
- o_1 is a release operation over a lock, and o_2 is the next acquire operation over the lock that succeeds, or
- o_1 is an arrive operation to a barrier, and o_2 is a departure from the barrier.

The relation $\rightsquigarrow$ is, as before, the transitive closure of the union of $\rightarrow$ and $\mapsto$. Reusable barriers and other objects such as semaphores and condition variables can be modeled as well.

Since locks and barriers suffice to solve many synchronization problems in a natural way, the following discussion on how to implement explicit synchronization operations is centered around them. In general, the problem remains the same: when a read operation o takes place at a node $\mathcal{P}$, the protocol must ensure that all intervals with operations that potentially causally precede o are recorded in W_P and carried out before the operation completes. The only difference is that the relation $\rightsquigarrow$ is augmented with synchronization operations.

5.1 Locks

The same ownership mechanism used to totally order write operations over a page can be used to obtain a distributed implementation of locks, similar to the one in Munin [10]. For each lock l, there is always exactly one owner, which is the manager that last acquired the lock on behalf of its local process. Each manager keeps a hint of l's probable owner, to whom requests to acquire the lock are sent. Requests received by P are forwarded to the probable owner, except when P is the owner or has requested the lock. From the time that manager P requests the lock, and until it is released by the local process, all requests for l received by P are enqueued. If the queue is not empty when the lock is released, then P grants the lock and transfers the rest of the queue to the first manager in the queue. Upon receiving the lock, the manager inserts the queue transferred in front of its local queue for l. Otherwise, P remains as the owner of l, until it is requested by another manager.

When a lock is acquired by process p, a relation $o_1 \mapsto o_2$ is established, where o_1 is the most recent release operation over the lock, and o_2 is p's acquisition of the lock. As a consequence, each time a lock is granted, a disclosure for the lock is attached, which includes all intervals in the sender's write list up to the one where o_1 belongs, and excluding—as before—those already present in W_P. The receiver processes the disclosure as always. In this sense, granting a lock is like sending a page whose contents are of no interest.

This approach avoids bottlenecks that are inherent to centralized solutions, betting on locality of reference: the fewer the number of processes that contend for the same lock, the less the communication costs. An extreme but not unusual situation occurs when the last process that acquired the lock reacquires it. In such a case, no messages are sent [10].

5.2 Barriers

A barrier establishes a synchronization point for a set of processes. Informally, no process in the set can go beyond the barrier until it is aware of what every other process has done before getting to the barrier. Therefore, after a barrier, every manager must end up with a record in its write list for every interval generated at the other nodes before the barrier. With that purpose, one of the managers, say P, is designated as the server of the barrier. When a process arrives at the barrier, the local manager closes the current interval and sends to P all interval records in its list, except those that it knows P already has. When p itself arrives at the barrier, the procedure is conceptually the same. Once P has received records from every participant, it merges them into W_P, and sends back the whole W_P to each manager (excluding the records that each manager already has). Such a reply is processed by the receivers as any disclosure, that is, new records are added to the write list, and the pages in those records are invalidated.

It is easy to see that such a centralized solution is optimal in the number of messages. Likewise, the lack of locality of reference in barriers (since the most usual situation is that all processes participate) discourages a distributed solution [10].

5.3 Semaphores and others

Other synchronization objects, such as condition variables and semaphores, can be built on top of the previous ones. Nevertheless, an *ad-hoc* implementation is usually more efficient. For semaphores in particular, a mechanism that never blocks on a **V** operation is discussed in [25], in contrast to the proposition in [2], where **V** operations over the same semaphore are totally ordered and are, thus, blocking.

6 Preliminary evaluation

The performance of any DSM protocol depends on the memory access and synchronization patterns shown by the applications that run on top of the protocol. Since such patterns greatly vary from one application to another, evaluating a protocol is not simple. In this work, a small set of simple applications was chosen to benchmark the protocol of Section 4 and compare it against two protocols proposed by John and Ahamad [15], which appear as the most practical propositions for causally consistent DSM.

The comparison was accomplished through the use of a simulator, which allowed to simulate as many as 64 nodes—far more than the number of homogeneous workstations available to conduct the experiments using the alternative of an actual implementation.

6.1 Related work

Boyer [9] presented a causally consistent DSM system, in which, before writing a page, the writer sends to every holder of a copy an order for the page to become *old*, and thus read-only (under sequential consistency, the order should be for the page to become invalid). When receiving an up-to-date version of a previously old page, all other old pages at the site are invalidated. This simple method compares well to sequential consistency, since invalidations are deferred, and parallelism is consequently increased. Nevertheless, message overhead is still high, and more pages than necessary are invalidated. To relieve the latter problem, an optimization which involves keeping a record of complex dependencies among processes is possible, but it is effective only in the particular situation where different processes in the same node map different sets of pages from the shared address space. Boyer says that "performance of this second solution will have to be compared with the simplest one... [keeping dependencies] may be too expensive, leading to adopt the previous solution".

John and Ahamad [14] proposed a protocol based on vector clocks [27]. At each node, there is a vector timestamp for each page, and a vector timestamp for the node. When a write fault occurs, the page ownership is requested to the owner, say Q, indirectly via a fixed page server. The owner replies with the node's timestamp, say TS_Q, and the page contents. The requester updates its node's timestamp with TS_Q by computing the component-wise maximum and incrementing its own component; sets the timestamp of the page to TS_Q; invalidates all non-owned pages whose timestamp is less then TS_Q;[1] and unprotects the page. The comparisons needed to find out which pages must be invalidated generate high processing overhead. Moreover, the method may invalidate more pages than necessary. For instance, pages written only during initialization are always invalidated, even though they never change. A read fault is handled in a similar way, except that the timestamp at the requesting node is not incremented, and the page is set to read-only. That is to say—unlike the proposition in this article—read requests must be serviced by the page owner.

When applications are known to be data-race-free [1], managers are free to bother with data consistency only when synchronization operations complete at the node, as John and Ahamad proposed in [15]. In comparison to [14], processing overhead and the number of unnecessary invalidations are substantially reduced. In the same paper, another mechanism for data-race-free programs that completely eliminates the unnecessary invalidations was devised. Under this approach, each page has a version number, which is increased with each write fault. A version vector—which consists of the most recently known version number of each page—is attached to synchronization messages. The receiver invalidates all pages that have a version number smaller than the corresponding number in the attached vector. The version vector's size is about 1KB per MB of address space, so the benefits of this method with big shared data spaces are not clear.

Boyer's work represents only a small step from sequential consistency; it does not exploit the full potential of causal consistency. In [25] it is shown that the costs of unnecessary invalidations of the proposition in [14] are extremely high in most cases. For those reasons, the timestamp protocol and the version array protocol in the latter work of John and Ahamad (hereinafter the TS and VA protocols, respectively) were chosen to be empirically compared against the protocol in this work (hereinafter the write lists or WL protocol). TS and VA are not as general as WL since they do not guarantee causal consistency for non data-race-free programs. Nevertheless, since "most shared-memory parallel programs are free of data races even when written assuming a sequentially consistent memory" [10], the comparison is not so unfair to WL.

6.2 Simulations

The simulator was written in C++ to take advantage of operator overloading. It intercepts each operation over a shared variable, and resolves whether a page fault should be triggered or not. Each process is a thread that calls the manager routines when a fault occurs. The CPU is preempted from a process after a fault or after a random number of non-faulting accesses to the shared memory. The sending of a request is simulated with a function call that returns the reply. Excluding the code for the applications and for the protocols themselves, the simulator hardly has 250 lines of code. More details can be found in [26].

The problem with such a simple simulator is that it is not possible to measure total execution time. Instead, it gives other important measures that directly contribute to execution time, such as number of messages, amount of information transmitted, number of write faults, and processing overhead (i.e., the CPU time spent by managers).

It was also considered important to compare the number of times that requests are forwarded. To locate the owner of a page, VA and TS use a *fixed manager* approach [22], which

[1] $TS_1 < TS_2$ if every component of TS_1 is less than or equal to the corresponding component of TS_2, and at least one is not equal.

requires 2 or 3 messages for each read or write request. In contrast, WL uses a *dynamic distributed manager* strategy [22], thus requiring between 2 and n messages per request, where n is the number of nodes. Nevertheless, the amortized cost is smaller than n, and it is further reduced when the application shows some degree of locality of reference. Moreover, in WL read requests always cost 2 messages, since the current owner of the page need not be located.

The page size was set to 4KB as few machines support smaller sizes, and greater sizes increase false sharing. The simulation was realized in a Digital DECServer 3000/700 under OSF1 V3.0., equipped with an Alpha A21064A processor at 225 MHz, and 256 MB of RAM.

6.3 Results

Four small benchmark parallel programs were built to run on the simulator. Each program was run using from 2 to 64 simulated nodes for each protocol. In order to represent realistic situations, false sharing was not avoided.

Table 1 summarizes the results. In general, messages are longer in WL than in VA and TS, but the difference in the average size of message is not enough to make this point relevant Also, except for one application (see below), the number of write faults is quite similar for all protocols. Figure 6 shows the factors that appear as more influential. Left graphs depict the total number of messages that each protocol sends, using from two to 64 nodes. Right graphs depict the processing overhead at each node, also for two to 64 nodes.

6.3.1 Matrix multiplication

In this program the product of two 450×450-element matrices is computed in parallel by assigning to each process a number of contiguous elements of the result matrix. One of the processes collects the results after a barrier.

WL sends fewer messages than VA and TS, but the difference is entirely due to forwarding requests: the number of requests is basically the same for all protocols, but VA and TS forward more of them. That is why the difference increases up to almost 50% with the number of processes. Results regarding processing overhead are less clear; though differences are small, WL is the best with few processors, but it gets worse as the number of nodes increases.

6.3.2 Integer sort

This program sorts 3 million integer keys in the range 1 to 30000, using a three-stage parallel version of *counting sort* [18]. In the first stage each process inspects a portion of the input array to determine, for each integer i in the range, how many numbers less than or equal to i there are. In the second stage, each process adds its results to a shared array, using a lock to ensure exclusive access. In the last stage, after a barrier, the processes use the shared array to produce the sorted output.

Results are similar to the previous program: up to almost 50% more messages for VA and TS, all due to forwarding requests. Nevertheless, WL incurs in higher processing costs.

6.3.3 Finite differences

This program solves Laplace's equation $\nabla^2(\Phi) = 0$ using a finite differences iterative method [4]. Each process is assigned a set of rows from two 500×500-element matrices. In each iteration a new value for each cell is computed as the average of the neighbor cells and stored in the other matrix. Processes wait at a barrier and swap the role of matrices before the next iteration.

Table 1: Results of simulations.

Nodes	Messages			Info (MBytes)			Overhead (msec)			Write faults			WriteFwd/WriteReq			ReadFwd/ReadReq		
	WL	VA	TS	WL	VA	TS	WL	VA	TS	WL	VA	TS	WL	VA	TS	WL	VA	TS
Matrix multiplication																		
2	1986	1986	1986	3.9	3.9	3.9	55	55	72	1188	1188	1188	0/200	0/200	0/200	0/792	0/792	0/792
4	5358	6692	6692	10.5	10.5	10.5	105	155	155	1190	1190	1190	2/301	150/301	150/301	0/2374	1186/2374	1186/2374
8	11804	16211	16211	23.3	23.1	23.2	305	266	283	1194	1194	1194	6/354	261/354	261/354	0/5538	4152/5538	4152/5538
16	24550	35251	35251	48.7	48.1	48.7	622	583	605	1202	1202	1202	14/387	340/387	340/387	0/11866	10375/11866	10375/11866
32	49963	73303	73303	100.7	97.9	100.6	1105	1088	1060	1217	1217	1217	29/414	388/414	388/414	0/24522	22981/24522	22981/24522
64	100761	149375	149375	209.3	197.5	209.0	2466	2288	2194	1250	1250	1250	61/452	437/452	437/452	0/49835	48238/49835	48238/49835
Integer sort																		
2	8449	8450	8452	16.6	16.6	16.5	560	527	499	5920	4456	4456	0/1371	0/1371	0/1371	0/2851	0/2851	0/2852
4	12884	16098	16098	25.3	25.4	25.2	711	532	699	5982	5250	5250	2/2105	1049/2105	1049/2105	0/4327	2163/4327	2163/4327
8	15498	21287	21287	30.6	30.7	30.4	911	772	810	6106	5740	5740	6/2564	1922/2564	1922/2564	0/5160	3867/5160	3867/5160
16	17924	25711	25711	35.7	35.9	35.4	1088	1016	944	6354	6171	6171	14/3059	2673/3059	2673/3059	0/5848	5118/5848	5118/5848
32	20918	30607	30607	42.4	42.6	41.7	1294	1099	1138	6850	6759	6759	30/3724	3490/3724	3490/3724	0/6620	6211/6620	6211/6620
64	25620	37777	37777	53.9	53.5	52.4	2016	1505	1499	7842	7797	7797	62/4800	4654/4800	4654/4800	0/7775	7531/7775	7531/7775
Finite differences																		
2	2298	2298	2298	4.5	4.6	4.4	344	138	99	11034	1585	1585	0/558	0/558	0/558	0/570	0/570	0/570
4	3991	4947	4947	7.6	8.0	7.5	483	205	177	11246	2027	2027	39/931	490/931	490/931	0/982	505/982	505/982
8	5975	7750	7750	11.1	11.8	10.8	661	294	272	11489	2536	2536	217/1302	921/1302	921/1302	0/1430	1071/1430	1071/1430
16	9218	12218	12218	16.7	18.0	15.9	1044	527	388	11890	3380	3380	580/1851	1657/1851	1657/1851	0/2153	1923/2153	1923/2153
32	15508	20104	20104	27.7	29.8	25.9	2316	977	777	12602	4985	4985	1456/2858	2718/2858	2718/2858	0/3517	3334/3517	3334/3517
64	27860	35396	35396	51.4	53.0	46.4	6099	1805	1449	14101	8139	8139	3190/4824	4692/4824	4692/4824	0/6188	6034/6188	6034/6188
Linear equations																		
2	3660	3660	10292	7.2	7.2	20.1	100	105	244	1822	1822	1828	0/987	0/987	0/993	0/838	0/838	0/4148
4	11196	12253	37113	20.7	20.5	59.6	366	222	883	3546	3547	3562	706/2714	518/2714	526/2728	0/2516	1251/2513	6213/12444
8	21011	24952	88650	36.1	35.7	127.0	583	321	1838	4022	4022	4017	2817/3189	2374/3189	2372/3185	0/5873	4402/5864	21778/29030
16	37127	45585	188367	64.2	62.4	260.0	1155	749	3721	4079	4076	4079	5107/3248	2839/3244	2842/3247	0/12687	10982/12563	54471/62205
32	66239	86124	387579	120.6	115.4	531.7	2055	1133	8499	4080	4080	4080	7179/3249	3043/3250	3043/3250	0/26126	24345/25963	120622/128552
64	121359	166930	786082	238.1	221.3	1099.4	3688	2261	21098	4079	4080	4080	8383/3247	3147/3248	3148/3249	0/52926	51123/52767	253314/261246

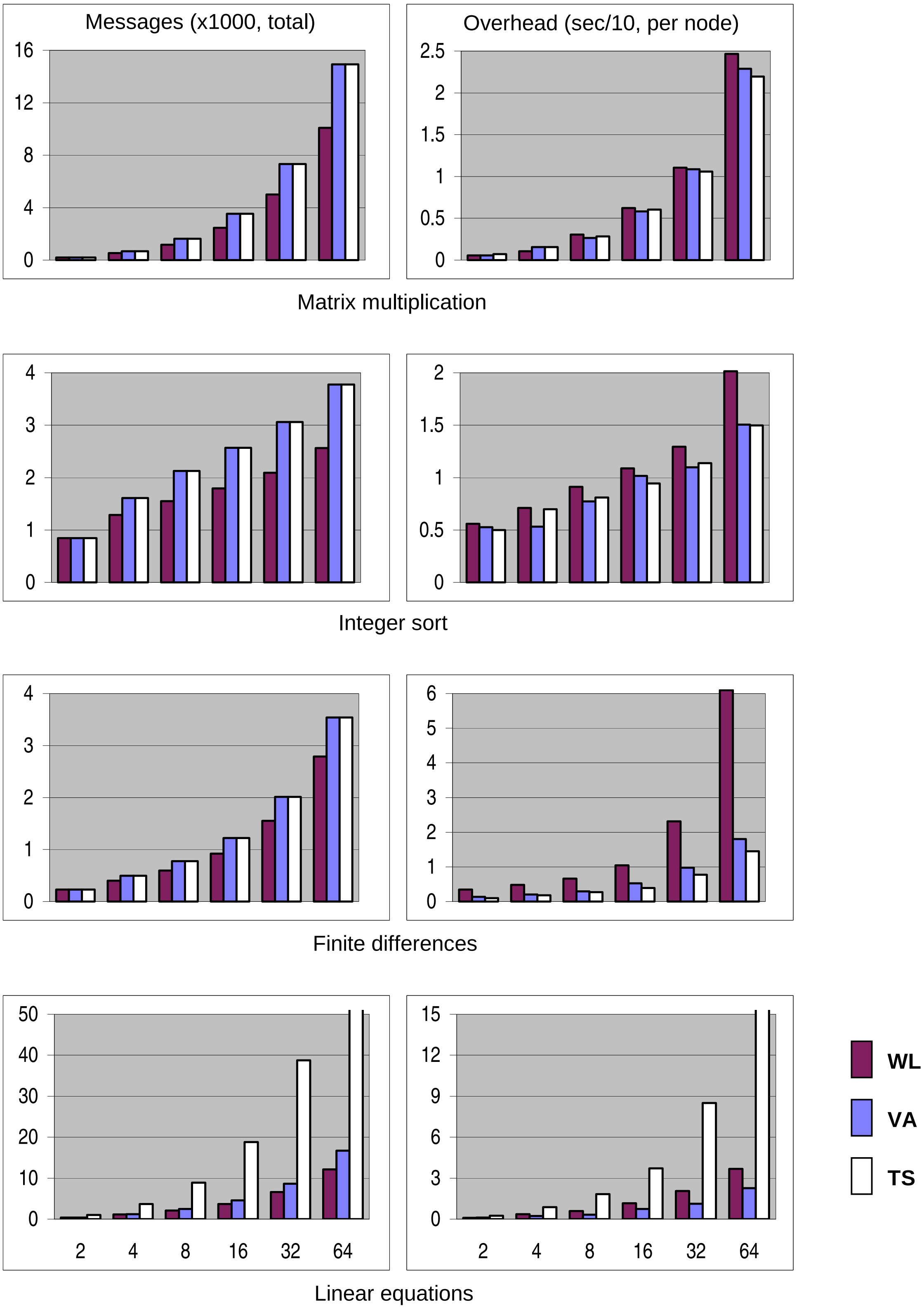

Figure 6: Number of messages and processing overhead.

In this case, VA and TS also send more messages than WL because of the fixed manager strategy. However, the relative difference is smaller than in the previous two cases since the large number of barriers imposes a high fixed cost in messages, which is the same for every protocol.

The processing overhead of WL compared to the overhead of the other protocols stands out in this experiment. The problem is due to the fact that in each iteration half the shared space is written before a barrier. As a consequence, barriers in WL are quite expensive, since they carry big interval records that are processed at every node. Navarro suggests alternative implementations of barriers [25]. One of them would make barrier processing costs virtually null in a case like finite differences, but might indirectly increase other costs in other programs, since it might produce more invalidations. Further study of this issue is necessary.

Other cost that is much higher for WL is the number of write faults. In this application, especially with few nodes, most pages are actually not shared (read and written by only one process), but such pages are reprotected at each barrier in WL, when intervals are closed. In contrast, VA and TS never reprotect those pages because they are never requested by other processes. The problem can be relieved if a page is considered private while it is invalid at all other nodes, and all owner hints point to the actual owner. It is easy to check (or enforce) this condition during barriers. Private pages need not be considered for consistency, and, hence, need not be reprotected. A private page remains private until requested. At that time it must be reprotected, and included in the current interval. Again, other applications may be subject to more invalidations with this variant.

6.3.4 Linear equations system

This program uses the Gauss-Jacobi method to solve the equations system $Ax = b$, where x and b are 650-element vectors and A is a 650×650-element matrix. Each process is assigned a piece of x. In successive iterations separated by barriers, a new value for each component of x is computed, using values from the previous iteration.

The results of this test differ from the former ones in the following sense. First, TS behaves substantially worse than the rest because of the problem of unnecessary invalidations (which does not appear in the previous cases). Second, WL forwards more write requests than VA and TS for eight or more nodes. This phenomena is due to the fact that in each iteration all processes write into x, which fits in just two pages. Hence, as all processes contend for at least one of those two pages, ownership hints become old quickly. And third, WL sends more read requests than VA because it makes some invalidations between barriers that VA does not make, since only synchronization operations can produce invalidations in VA. All in all, WL still sends less messages than VA because it does not forward any read request. But, as usual, processing costs are higher.

6.4 Fixed or dynamic manager?

So, if the dynamic distributed manager strategy is so much better, could not VA or TS use it to improve their performance? The answer is, VA could, but TS could not. When a page is invalidated, WL knows the writer that produced the invalidation and whether that writer is newer or older than the current hint; TS does not. Worse yet, in TS a page may be invalidated even if there was no write operation that justifies the invalidation. In conclusion, ownership hints cannot be as good as in WL. Moreover, TS is compelled to deliver read requests to owners since they may not be serviced by the mere probable owner.

VA, in contrast, could be augmented to keep ownership hints if, along with the version of each page, the last writer of the page is sent. The obvious consequence is that syncronization messages would increase their size. A dynamic manager version of VA is expected to send exactly the same number of messages than WL, but more bytes in total.

7 Conclusions and future work

The experimental results show that the proposed protocol is promising, since in every test with more than two nodes it sends fewer messages than the other causally consistent protocols used in the comparison. Nevertheless, more work is necessary, especially to reduce processing costs and the number of write faults in some cases.

The authors believe that it is possible to optimize the protocol to achieve such reductions. Some mechanisms have been discussed in this article and others in [25]. For instance, it was observed that barriers contribute to a great part of the total overhead; methods to reduce the cost of barriers are possible. It is also possible for the sender to detect easily that some interval records are subsumed by subsequent ones in the same disclosure; in that case they will never produce any effect, and can be discarded. Although the detection should slightly increase processing costs at the sender, in some situations the savings both in bandwidth and in receiver's processing time can be substantial. For instance, in the second stage of integer sort, disclosure sizes would decrease from quadratic to linear in the number of nodes [25].

WL could also take advantage of data-race-free applications: by restraining consistency-related actions only to synchronization operations, scale economies would reduce the total amount of transmitted information (although it would make network traffic burstier). In addition, those additional invalidation observed in Section 6.3.4 would disappear. Another optimization involves *double faults* (a read fault closely followed by a write fault over the same page [17]), which force two page transfers. If the page contents are identical in both transfers, the second is useless; it can be avoided easily by detecting such situation. Indeed, TS and VA can also benefit from detecting double faults.

An actual implementation of the protocol, including these and other optimizations, and garbage collection (see below) is under way. Objectives include comparing it with other protocols, using more complex applications and bigger problem sizes. Thanks to the implementation, the comparison can now be based on total execution time, arguably, one of the most important factors. It might be interesting to compare an optimized version of WL, tailored for data-race-free applications, against a dynamic manager version of VA. The authors believe that WL can also compete with protocols that offer more relaxed consistency models.

To bound the size of write lists, garbage collection is, rather than an optimization, a necessity. Once an interval record is present in every write list, it becomes garbage, since it will not be used any more. Whether a record is garbage or not is a global property; to determine it, some extra information must be exchanged among nodes. Furthermore, it is conceivable that a write list reaches its maximum possible size without containing any garbage. In that case, the manager is forced to broadcast some of its records, causing invalidations at the other nodes that would not occur otherwise. There is probably no best approach to garbage collection, since there are many trade-offs. In any case, a barrier where all processes participate is a convenient point to do it. Garbage collection was not implemented for the experiments of Section 6; some extra costs are expected due to this feature.

A multiple-writer variant has not been explored. To realize it, it is necessary to detect what portion of a page a writer modifies, in order to be able to merge concurrent changes to the same page. This can be achieved *à la* TreadMarks [3]: before a page is written, a copy is made. To determine the actual modifications, the page is later compared with the copy. Such approach might offer more parallelism, but at the cost of significantly increasing processing and memory overhead [16]. It is worth noting that the single-writer variant has the advantage of offering a stricter model, since it is page-coherent. For instance, it has been established elsewhere [5] that under pure causal consistency it is not possible to obtain mutual exclusion without cooperation; nevertheless, any non-cooperative mutual exclusion protocol (such as Peterson's), is correct under page-coherent causal consistency if all its variables are in the same page. Also note that a multiple-writer version of VA is not conceivable.

Acknowledgments

The authors gratefully acknowledge that this work has been partially funded by FONDECYT grant Nº 196-0381, and DIPUC grant 96/08E. Comments of Yadran Eterovic and Luis A. Rojas helped improve this article.

References

[1] S. Adve and M. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation, and programming. Technical Report 93/55, College of Computing, Georgia Institute of Technology, Sept. 1993. (Revised 1994).

[3] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[4] G. Andrews. *Concurrent Programming.* Benjamin/Cummings, Redwood City, California, 1991.

[5] H. Attiya and R. Friedman. A correctness condition for high performance multiprocessors. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 679–690. ACM Press, May 1992.

[6] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1987.

[7] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[8] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the Usenix Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–72, 1993.

[9] F. Boyer. A causal distributed shared memory based on external pagers. In *Proceedings of the 2nd Usenix Mach Symposium*, pages 41–57, Nov. 1991.

[10] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory. *ACM Trans. Comput. Syst.*, 13(3):205–243, Aug. 1995.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[12] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, Mar. 1989.

[13] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–311, May 1990.

[14] R. John and M. Ahamad. Causal memory: Implementation, programming support and experiences. Technical Report 93/10, College of Computing, Georgia Institute of Technology, 1993.

[15] R. John and M. Ahamad. Evaluation of causal distributed shared memory for data-race-free programs. Technical Report 94/34, College of Computing, Georgia Institute of Technology, July 1994.

[16] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[17] R. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 498–507, 1989.

[18] D. E. Knuth. *Sorting and Searching*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[20] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[21] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, Sept. 1986.

[22] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[23] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Prog. Lang. Syst.*, 8(1):142–153, Jan. 1986.

[24] D. Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, Jan. 1993.

[25] J. E. Navarro. Un protocolo de consistencia causal para memoria compartida distribuida. Master's thesis, Departamento de Ciencia de la Computación, P. Universidad Católica de Chile, 1996.

[26] J. E. Navarro. Simulación bajo memoria compartida de un sistema distribuido que simula memoria compartida. In *Actas del III Congreso Argentino de Ciencia de la Computación*, Sept. 1997.

[27] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report SFB124-15/92, Department of Computer Science, University of Kaiserlautern, Germany, 1992.