Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Information Processing Letters 86 (2003) 169–176

Information Processing Letters

www.elsevier.com/locate/ipl

On the composability of consistency conditions ☆

Roy Friedman [a,*], Roman Vitenberg [a], Gregory Chockler [b]

[a] *Computer Science Department, Technion – The Israel Institute of Technology, Haifa 32000, Israel*
[b] *Institute of Computer Science, The Hebrew University of Jerusalem, Givat Ram, Jerusalem 91904, Israel*

Received 4 March 2002; received in revised form 25 November 2002

Communicated by F.Y.L. Chin

Abstract

This paper presents a formal framework, which is based on the notion of a *serialization set*, that enables to compose a set of consistency conditions into a more restrictive one. To exemplify the utility of this framework, a list of very basic consistency conditions is identified, and it is shown that various compositions of the basic conditions yield some of the most commonly used consistency conditions, such as *sequential consistency*, *causal memory*, and Pipelined RAM. The paper also lists several applications that can benefit from even weaker semantics than Pipelined RAM that can be expressed as a composition of a small subset of the basic conditions.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Concurrency; Distributed computing; Distributed systems; Formal semantics

1. Introduction

Distributed shared objects is a common technique for information sharing in distributed systems. That is, processes communicate by updating and querying the same objects. For efficiency and availability purposes, most implementations of distributed shared objects, and other forms of distributed shared memory, employ replication and caching. However, once an object is replicated and/or cached, multiple copies of the same object may exist in the system at the same time. If care is not taken, these replicas might end up with uncorrelated data, rendering them useless as a mean for communication. Thus, it is common to restrict the ways the content of replicas might diverge. Such restrictions are known as *consistency conditions*.[1]

As been shown in several research works, there is an inherent tradeoff between the level of consistency of shared objects and the communication overhead and access latency [4,5]. Moreover, conditions that provide very week guarantees, are too weak to solve some fundamental problems in concurrent programming [3]. This has inspired the definition of a large number of

☆ This work was supported in part by the Israeli Ministry of Science grant number 1230-1-98.

* Corresponding author.
E-mail addresses: roy@cs.technion.ac.il (R. Friedman), romanv@cs.technion.ac.il (R. Vitenberg), grishac@cs.huji.ac.il (G. Chockler).

[1] If there is only a single copy of each object in the system and no caching is used, then this copy is always consistent with itself. Yet, there might be implementation optimizations that make it look inconsistent to applications that are using the object. This special case is covered by the general framework in this paper.

different consistency conditions and their corresponding implementations, each trying to tackle the tradeoff slightly differently. Of course, without this tradeoff it would have been possible to identify one consistency condition that is clearly better than all others.

In this work, we try to present a more generic approach to consistency conditions, by looking at the question of their composability. That is, we start by presenting a formal framework that enables composing a collection of consistency condition into a more restrictive condition. We then identify a list of six very basic consistency conditions, and prove that various compositions of these basic conditions yield several well-known consistency conditions, such as sequential consistency [10], causal memory [2], and Pipelined RAM (PRAM) [11]. Moreover, we list several applications that can benefit from weaker consistency semantics than PRAM that are easily expressed as a composition of a small set of the basic conditions.

We would like to emphasize that sequential consistency, causal memory, and Pipelined RAM were chosen since they are well known and widely used. They also represent three interesting points in the expressiveness vs. performance tradeoff. That is, sequential consistency provides a logical illusion of executing the operations on a truly non-replicated shared memory (or objects). Yet, its implementations require synchronization which hurts its performance and scalability. Pipelined RAM is the weakest consistency condition that was offered for distributed shared memory and is still claimed to be useful for a reasonably large set of applications. Yet, for many applications it is too weak. Causal ordering is somewhere in the middle with respect to both concerns.

From a theoretical point of view, looking at some basic consistency conditions and composing them into higher level ones presents an important insight into the differences between the high level conditions. Also, for a given set of n consistency conditions, there can be 2^n possible compositions. The composability framework gives us access to all of these combinations, without having to formally define each one of them independently.

From a practical point of view, this has two main advantages: First, it can simplify proving that a given implementation of a high level condition is correct. Specifically, the basic conditions we propose are very simple. Thus, it is likely to be easier to show that a given implementation obeys the relevant basic conditions rather than proving directly that it obeys the corresponding high-level condition. In case an implementation is incorrect, it may be possible to identify the errors when considering the simpler conditions in a more effective and accurate way than when looking directly at the composed condition. Second, we hope that our work can be used to devise flexible, composable, implementations of consistency conditions. In such an ideal system, each basic condition will be implemented as a layer of code. When a given application needs a specific high-level consistency conditions, it could simply pick the layers that implement the collection of basic conditions from which the high-level condition is composed of.

The basic consistency conditions that we list are inspired by the work on the Bayou project [13]. However, in Bayou the conditions were specified in an informal, operational and implementation dependent way, while our definitions are formal and implementation independent. Moreover, the work on Bayou did not address the issues of composability. Another difference between Bayou and our work is the context of each work: Bayou is mainly concerned with distributed databases while we are interested in distributed shared objects. In distributed databases often the requirement is to ensure ordering semantics between complete transactions, where each transaction can be composed of several operations and may even access multiple objects. In shared objects and distributed shared memory each operation only accesses a single object, and the ordering guarantees should be maintained between individual operations.

Our work was motivated from a CORBA caching service, called CASCADE, that we have implemented [6]. However, for the sake of generality, the model and claims in this paper refer to distributed shared memory in general. Also, in this paper we do not discuss any implementation. The implementation of some useful combinations of the basic consistency conditions are described in [7].

2. Definitions and conventions

2.1. Basic definitions

In this paper, we are only interested in the operations invoked by client processes that may access the distributed shared memory concurrently. The exact details of how it is implemented, whether it is replicated or not, and how the replication mechanism works are outside the scope of this work.

Formally, consider a system consisting of a collection of *client processes*, or simply *processes*, numbered $p_1, \dots, p_n$, communicating by invoking *operations* on a collection of *objects*. Each operation *op* is composed of an invocation event $inv(op)$ and a corresponding response event $resp(op)$, both occurring at the same process, denoted by $pr(op)$. Moreover, each operation *op* is restricted to a single object, denoted by $obj(op)$; we say that *op accesses obj*(op). The invocation event may take an input parameter $ival(op)$ and the response event may return an output parameter $oval(op)$, both defined over some allowed range of values $\mathcal{V}$. Here we only consider read and write operations; for read operations the input value is always a special value $\perp$, while for write operations the output value is always $\perp$. For a given read operation r, we say that *r returns v* (or *reads v*) if $oval(r) = v$. Similarly, we say that a write operation *w writes v* if $ival(w) = v$.

We define a *history* to be a sequence of invocation and response events. A *well formed history* is a history in which for each invocation event $inv(op)$ there is a corresponding response event $resp(op)$. For the rest of this paper, we will assume that all histories are well formed. A *sequential history* is a history in which each invocation event is immediately followed by the matching response event. For a given history H and process p_i, we denote by $H|p_i$ the restriction of H to events of process p_i, $H|w$ the restriction of H to write operations, and $H|p_i + w$ the restriction of H to events of process p_i and events of write operations by any process. A history that only includes events of a single process is called *local history*; clearly, for any history H, $H|p_i$ is a local history. In this work, we assume that for any history H and process p_i, $H|p_i$ is sequential. This corresponds to a sequential execution model in which a process is not allowed to issue a new operation before a previous one returns. A *serialization S* of a history H is a sequential history containing all the operations of H.

Legality is a central concept in consistency conditions. We define a sequential history H to be *legal* if the value returned by each read operation r in H is the same as the value written by the last write operation that accesses the same object before r in H.

Given a sequence of operations, or a sequential history, S, we denote $o_1 \xrightarrow{S} o_2$ when o_1 precedes o_2 in S. A history H induces a partial order, $\xrightarrow{H}$, on the operations that appear in H: $o_1 \xrightarrow{H} o_2$ if $o_1 \xrightarrow{H|p_i} o_2$ for some $H|p_i$. Moreover, we slightly abuse the notations and for a given sequential history H and a given serialization S we denote by $H = S$ the fact that H and S include the same set of operations, and all operations are ordered the same in H and in S.

2.2. Composability framework

We define a *consistency condition* (or simply, *consistency*) as a set of restrictions on allowed histories. We say that consistency A is *stronger* than consistency B if the set of allowed histories under A is contained in the set of histories allowed under B.

Note that in order for shared objects to be a meaningful tool for communication between processes, every write operation by any process must be seen by all other processes. This is captured by the following property:

Eventual Propagation. For every process p_i and history H, there exists a legal serialization S_{p_i} of $H|p_i + w$.

This requirement essentially expresses liveness of update propagation: For a given history and a given write operation in this history, if some process invokes an infinite number of queries, it will eventually see the result of this write. We therefore assume that Eventual Propagation holds for all histories considered in the rest of this paper. Fig. 1 gives a simple example of a history H with two processes p_1 and p_2 and two objects x and y. Each of the processes issues one read and one write operation. Fig. 1 also presents legal serializations $S1$ and $S2$ of $H|p_1 + w$ and $H|p_2 + w$, respectively.

Note that traditional definitions of strong consistency conditions, such as sequential consistency, typically require the existence of some special legal serialization of the history. This serialization represents the logical order by which all processes view the operations. On the other hand, definitions of weaker condi-

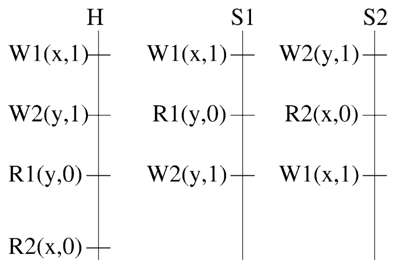

Fig. 1. Example of a history and its legal serializations.

tions, such as PRAM, only require the existence of one special serialization for each process. In other words, such definitions allow each process to view the operations in a different logical order. In order to relate these definitions in a single framework, we introduce the following definition:

Definition 1. A *serialization set of H* is a set of legal serializations of $H|p_i + w$ one for each process p_i.

Due to Eventual Propagation, at least one serialization set exists for a given history. In the example presented in Fig. 1, $\{S1, S2\}$ is a serialization set of H.

We then define that a history preserves a consistency condition by requiring the existence of a serialization set that obeys certain ordering restrictions. Moreover, as described in Section 3.3.1, some applications can do with very weak consistency conditions, in which the view of each process may be different in a very fundamental way. This leads us to define at least some of the consistency conditions by requiring a serialization set in which different serializations must obey different restrictions.

Formally, we define consistency conditions in the following way. For a given consistency condition X, a given history H, a serialization set S of H, and a serialization $S_{p_i} \in S$, we define what the conditions that S_{p_i} needs to obey in order to *preserve* X are. We then say that a serialization set $S = \{S_{p_j}\}$ *globally preserves* X if it preserves X for all the histories $H|p_i$. For a given history H, a serialization set $S = \{S_{p_j}\}$ globally preserves some set of consistency conditions if S globally preserves each condition in this set. Finally, we say that a history H is consistent with respect to a condition set (or a single condition) X if there exists a serialization set S of H such that S globally preserves X. We say that an implementation *A obeys a condition set* (or a single consistency condition) X is every history generated by A is consistent with respect to X.

3. Consistency conditions

3.1. Basic consistency conditions

We now present a set of elementary conditions that can be used as basic building blocks for constructing more composed consistency guarantees. While this set is in no sense complete and it can be extended in different ways, it proves a good example of the composability approach: By combining the basic conditions in this set we can express both widely known consistency conditions of varying strength and non-standard conditions that are shown to be useful for applications.

Read Your Writes. For a given history H and a process p_i, a serialization set $S = \{S_{p_j}\}$ preserves *Read Your Writes for the local history* $H|p_i$ if for every two operations o_1 and o_2 in $H|p_i$ such that $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, and $o_1 \xrightarrow{H|p_i} o_2$, implies $o_1 \xrightarrow{S_{p_i}} o_2$.

FIFO of Reads. For a given history H and a process p_i, a serialization set $S = \{S_{p_j}\}$ preserves *FIFO of Reads for the local history* $H|p_i$ if for every two operations o_1 and o_2 in $H|p_i$ such that $o_1 = \text{READ}$, $o_2 = \text{READ}$, and $o_1 \xrightarrow{H|p_i} o_2$, implies $o_1 \xrightarrow{S_{p_i}} o_2$.

FIFO of Writes. For a given history H and a process p_i, a serialization set $S = \{S_{p_j}\}$ preserves *FIFO of Writes for the local history* $H|p_i$ if for every two operations o_1 and o_2 in $H|p_i$ such that $o_1 = \text{WRITE}$, $o_2 = \text{WRITE}$, and $o_1 \xrightarrow{H|p_i} o_2$, holds $\forall j\ o_1 \xrightarrow{S_{p_j}} o_2$.

Reads Before Writes. For a given history H and a process p_i, a serialization set $S = \{S_{p_j}\}$ preserves *Reads Before Writes for the local history* $H|p_i$ if for every two operations o_1 and o_2 in $H|p_i$ such that $o_1 = \text{READ}$, $o_2 = \text{WRITE}$, and $o_1 \xrightarrow{H|p_i} o_2$, implies $o_1 \xrightarrow{S_{p_i}} o_2$.

Local Causality.[2] For a given history H and a process p_i, a serialization set $S = \{S_{p_j}\}$ preserves *Local Causality for the local history* $H|p_i$ if for every three operations o_1, o_2 and o_3 such that o_2 and o_3 are in $H|p_i$, $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, $o_3 = \text{WRITE}$, o_2 reads a result written by o_1 and $o_2 \xrightarrow{H|p_i} o_3$, implies $\forall j\ o_1 \xrightarrow{S_{p_j}} o_3$.

As noticed in [13], Read Your Writes, FIFO of Reads and Reads Before Writes only affect the local histories for which they are provided. On the other hand, Local Causality and FIFO of Writes contain guarantees with respect to the local histories of other processes.

Total Order. For a given history H, a serialization set $S = \{S_{p_j}\}$ globally preserves *Total Order* if for every two serializations S_{p_i} and S_{p_j} in S, $S_{p_i}|w = S_{p_j}|w$.

3.2. Examples of known consistency conditions

The following is a list of several important and well known consistency conditions.

Sequential Consistency (SC) [10]. A history H is sequentially consistent if there exists a legal serialization S of H such that for each process p_i, $o_1 \xrightarrow{H|p_i} o_2$ implies $o_1 \xrightarrow{S|p_i} o_2$.

PRAM Consistency [11]. A history H is PRAM consistent if there exists a serialization set S such that for every serialization $S_{p_i} \in S$ and operations o_1 and o_2 in $H|p_i + w$ for which $o_1 \xrightarrow{H} o_2$, implies $o_1 \xrightarrow{S_{p_i}} o_2$.

Causal Consistency [2]. For the definition of causal consistency we assume that no value is written more than once to the same variable. Given a history H, an operation o_1 *directly precedes* o_2 (denoted $o_1 \xrightarrow{H} o_2$) if either $o_1 \xrightarrow{H} o_2$ or $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, and o_2 reads a result written by o_1. Let $\xrightarrow{*}$ denote the transitive closure of $\xrightarrow{H}$.

A history H is causally consistent if there exists a serialization set S such that every serialization $S_{p_i} \in S$ respects $\xrightarrow{*}$, i.e., if o_1 and o_2 are two operations in $H|p_i + w$ and $o_1 \xrightarrow{*} o_2$, then $o_1 \xrightarrow{S_{p_i}} o_2$.

3.3. Examples of useful compositions

Any single condition that relates two events of the same type is trivial by itself. For example, if we only require FIFO of Reads, then naturally we can always find legal serializations in which all reads are ordered in FIFO order. This is because we have not placed any requirements on writes, and therefore we have the freedom to order the writes in the serialization so all the reads are legal. This applies similarly to FIFO of Writes, Local Causality and Total Order. Thus, these guarantees become meaningful only in combinations that contain several guarantees of different types. The only guarantees that are not trivial by themselves are Read Your Writes and Reads Before Writes.

We now present several theorems that show how some combinations of the basic consistency conditions relate to each other and to other known consistency conditions.

Theorem 1. *Any history that is consistent with respect to Total Order and Reads Before Writes is also consistent with respect to Local Causality.*

Proof. Assume, by way of contradiction, that there is a history H, which is consistent with respect to Total Order and Reads Before Writes, but is not consistent with respect to Local Causality. Thus, there are three operations $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, and $o_3 = \text{WRITE}$, such that o_1 was issued by p_i, o_2 and o_3 issued by p_j, and o_2 reads the result of o_1, and $o_2 \xrightarrow{H|p_j} o_3$, yet there is a serialization set S of H that globally preserves Total Order and Reads Before Writes in which $o_3 \xrightarrow{S_k} o_1$ for some process p_k. By assumption, for the serialization $S_{p_j} \in S$, $o_2 \xrightarrow{S_{p_j}} o_3$, and $o_1 \xrightarrow{S_{p_j}} o_2$. However, since S obeys Total Order, $\forall k\ o_1 \xrightarrow{S_{p_k}} o_3$. A contradiction. □

[2] This is similar to the condition called *Writes Follow Reads* in [13].

The order in which operations of some process p_i appear in $H|p_i$ is also known in the literature as *process order*. The conditions FIFO of Writes, FIFO of Reads, Read Your Writes, and Reads Before Writes can be seen as limitations of process order to the corresponding operations. For example, FIFO of reads only requires preserving process order for reads. Given this observation, the following theorem is not surprising.

Theorem 2. *Any history that is consistent with respect to FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes is also PRAM consistent.*

Proof. We need to show that there is a serialization set such that in each serialization S_{p_j}, $H|p_j = S_{p_j}|p_j$, and all write operations of any other process p_i are ordered in the same order as in $H|p_i$. Consider a given serialization set S that is consistent with respect to FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes, any serialization $S_{p_j} \in S$, and any process p_i. By FIFO of writes, every two write operations of p_i are ordered in S_{p_j} in the same order as in $H|p_i$. Thus, we only need to show that every pair of operations by p_j of which at least one is a read are ordered in S_{p_j} according to their order in $H|p_j$. However, this trivially holds for S_{p_j} due to FIFO of Reads, Read Your Writes and Reads Before Writes. $\square$

Theorem 3. *Any history that is PRAM consistent and is consistent with respect to Local Causality is also causally consistent.*

Proof. Let H be a history that is PRAM consistent and consistent with respect to Local Causality, and let S be the serialization set that obeys all the requirements of PRAM and Local Causality. We claim that S also obeys the requirements of the serialization set of causal consistency. Assume, by way of contradiction, that it does not. Thus, there must exist at least one serialization S_{p_i} in S for which there are two operations o_1 and o_2 such that $o_1 \xrightarrow{*} o_2$, but $o_2 \xrightarrow{S_{p_i}} o_1$. By the PRAM guarantees, o_1 and o_2 are operations of two different processes p_j and p_k, such that $k \neq j$, $k \neq i$, and $j \neq i$. By the definition of S_{p_i}, both o_1 and o_2 are write operations. Given that $o_1 \xrightarrow{*} o_2$, there is a sequence of operations $op_1, \ldots, op_l$ such that $o_1 = op_1$, $o_2 = op_l$, and $\forall q\ 0 \leqslant q \leqslant \frac{1}{2}l$, $op_{2\cdot q+1} = \text{WRITE}$, $op_{2\cdot q} = \text{READ}$, $op_{2\cdot q}$ reads the result of $op_{2\cdot q-1}$, and both $op_{2\cdot q}$ and $op_{2\cdot q+1}$ occur in the same process and in that order. Thus, for each couple of writes $op_{2\cdot q}$ and $op_{2\cdot q+2}$, Local Causality guarantees that they are ordered in this order on S_{p_i}. By transitivity, $o_1 \xrightarrow{S_{p_i}} o_2$. A contradiction. $\square$

Theorem 4. *Any history that is PRAM consistent and is consistent with respect to Total Order is also sequentially consistent.*

Proof. Let H be a history that is PRAM consistent and consistent with respect to Total Order. Thus, there exists a serialization set SS of H that obeys both the requirements of PRAM and Total Order. We we now show how to construct a legal serialization S of H such that for every process p_i, $H|p_i = S|p_i$. Due to Total Order, all the writes are ordered in the same order in all serializations in SS. Thus, we start by creating a serialization S of all writes in H ordered in the order they appear in all serializations. Next, we extend S by adding the read operations in the following manner, performed iteratively for all processes $p_i \in \{p_1, \ldots, p_n\}$: for each two write operations o_1 and o_2, we add all read operations by p_i (if any exist) that were ordered between o_1 and o_2 in S_0 and order them in S between o_1 and o_2. Also, if there are already some read operations by other processes between o_1 and o_2, we place the reads of p_i immediately after o_1. In a similar manner, we add to S all reads that are placed in any S_{p_i} before the first write, and place them before the first write in S, and add all reads that are placed in any S_{p_i} after the last write, and place them after the last write in S. Note that S now includes all operations of H, and is thus a serialization of H. Moreover, since all the operations of each process p_i are ordered in S in the same relative order as in S_{p_i}, they are also ordered in the same order as in $H|p_i$. Finally, since each read by any process p_i is placed in S between the same writes it was placed in S_{p_i}, and since S_{p_i} is legal, then S is also legal. Thus, S obeys all the requirements of sequential consistency, and hence, H is sequentially consistent. $\square$

3.3.1. Very weak conditions

When an applications obeys a known programming convention, it is often possible to run it on top of an implementation that provides a weak consistency condition, yet the result will be as if the application was run with a strong condition. The most prominent example of this is that executions of data-race-free programs on a release consistent distributed shared memory are in fact sequentially consistent [1]. Similarly, by exploiting the semantics of the application and specific operations, it may be possible to obtain meaningful correct behavior with weak ordering guarantees, as proposed in [9]. The benefit of this is that since the implementation only guarantees a weak condition, it can be implemented more efficiently. Below we give a couple of examples that demonstrate the usefulness of the basic conditions, even in combinations that involve only a few of them, and perhaps even provide different guarantees to each serialization.

Consider an application in which there is only a single writer. In this case, it is enough to require FIFO of Writes, and the result will be as if Total Ordering was used. In particular, when there is a single writer, PRAM is equivalent to sequential consistency. However, the equivalence of FIFO of Writes to Total Ordering might be useful even on weaker combinations than PRAM.

Another interesting application that can benefit from a condition that is weaker than PRAM is a bulletin board. Here, a client is only interested in other client's postings, and thus does not need Read Your Writes.

As a final example, consider an application in which there are several simple clients and a few supervisor clients. Each simple client reads and writes to different objects than the other simple clients. Yet, supervisors can read all objects. In this case, the serializations of simple clients should obey FIFO of Writes and Read Your Writes, while supervisors need FIFO of Reads.

4. Discussion

In our work on the CASCADE project, we have implemented several interesting combinations of the basic consistency conditions specified in this work, as detailed in [6]. However, for the sake of performance, we have created a single monolithic implementation for each chosen combination, rather than having a truly modular implementation. The main obstacle in providing such a modular implementation is that some of the basic conditions can be implemented much more efficiently when it is known that other conditions are also provided. The challenge is to generate automatic optimizations for a given composition of conditions, based on a set of implementations, one for each condition. Such optimization can either be done in compile time, or ideally, on-the-fly.

Another open problem is to generalize the framework to more generic operation types, and to be able to capture other consistency conditions such as release consistency [8], entry consistency [12], and hybrid consistency [4]. Also, an interesting question is whether there exists a basic condition, which is weaker than linearizability, which can be combined with sequential consistency to yield linearizability. An even grander challenge is to arrive at a complete set of basic consistency conditions. That is, be able to show that any consistency condition can be provided as a combination of a subset of these conditions, and that each of these conditions is necessary for implementing at least one of the currently known consistency conditions.

Acknowledgements

We would like to thank Danny Dolev, Michel Raynal, and the anonymous referees for interesting discussions and helpful comments.

References

[1] S. Adve and M. Hill, Sufficient conditions for implementing the data-race-free-1 memory model, Technical Report 1107, Computer Science Department, University of Wisconsin, Wisconsin-Madison, September 1992.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, P. Hutto, Causal memory: Definitions, implementation, and programming, Distributed Comput. 9 (1) (1993) 37–49.

[3] H. Attiya, R. Friedman, Limitations of fast consistency conditions for distributed shared memories, Inform. Process. Lett. 57 (5) (1995) 243–248.

[4] H. Attiya, R. Friedman, A correctness condition for high-performance multiprocessors, SIAM J. Comput. 27 (2) (1998) 1637–1670.

[5] H. Attiya, J. Welch, Sequential consistency versus linearizability, ACM Trans. Comput. Systems 12 (2) (1994) 91–122.

[6] G. Chockler, D. Dolev, R. Friedman, R. Vitenberg, Implementing a caching service for distributed CORBA objects, in: Proc. Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms, April 2000, pp. 1–23.

[7] G. Chockler, R. Friedman, R. Vitenberg, Consistency conditions for a CORBA caching service, in: Proc. 14th International Conference on Distributed Computing 2000, October 2000, pp. 374–388.

[8] P. Keleher, Lazy release consistency for distributed shared memory, PhD thesis, Department of Computer Science, Rice University, December 1994.

[9] R. Ladin, B. Liskov, L. Shrira, S. Ghemawat, Lazy replication: Exploiting the semantics of distributed services, in: 9th Ann. Symp. Principles of Distributed Computing, August 1990, pp. 43–58.

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

[11] R. Lipton, J. Sandberg, PRAM: A scalable shared memory, Technical Report CS-TR-180-88, Computer Science Department, Princeton University, September 1988.

[12] N. Neves, M. Castro, P. Guedes, A checkpoint protocol for an entry consistent shared memory system, in: ACM Proc. of Distributed Systems, 1994, pp. 121–129.

[13] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, B.B. Welsh, Session guarantees for weakly consistent replicated data, in: Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS), Austin, TX, September 1994, pp. 140–149.