# Consistency Conditions for a CORBA Caching Service*

Gregory Chockler[1], Roy Friedman[2], and Roman Vitenberg[2]

[1] Institute of Computer Science, The Hebrew University, Givat Ram, Jerusalem 91904, Israel,
grishac@cs.huji.ac.il,
WWW home page: http://www.cs.huji.ac.il/~grishac
[2] Computer Science Department, Technion – The Israel Institute of Technology, Haifa 32000, Israel,
{roy,romanv}@cs.technion.ac.il,
WWW home page: http://www.cs.technion.ac.il/{~roy,~romanv}

**Abstract.** Distributed object caching is essential for building and deploying Internet wide services based on middlewares such as CORBA. By caching objects, it is possible to mask much of the latency associated with accessing remote objects, to provide more predictable quality of service to clients, and to improve the scalability of the service. This paper combines a theoretical and practical view on specifying and implementing consistency conditions for such a service. First, a formal definition a set of basic consistency conditions is given in a vary abstract, implementation independent, manner. It is then shown that common consistency conditions such as sequential consistency, causal consistency, and PRAM can be formally specified as a combination of these more basic conditions. Finally, this paper describes the implementation of the proposed basic consistency conditions in CASCADE, a distributed CORBA object caching service.

The novelty of this work is three fold: The resulting caching service is very flexible in the consistency conditions it can provide to its clients, i.e., the programmer can pick a combination of basic conditions. Yet, since these basic constraints are simple, so is the implementation and its proof of correctness. Finally, the programmer is provided with a formal specification of the consistency conditions the service provides, which is important, e.g., for reasoning about the correctness of applications.

## 1 Introduction

Object caching is a promising approach for improving the scalability, performance, and predictability of Internet oriented services that are based on object-oriented middlewares such as CORBA [20]. Accessing a local or nearby cache incurs a much lower latency than accessing a far away object, and the access time

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

and availability of a cached copy is much more predictable than when accessing a remote object. Also, object caching greatly enhances the scalability of services because most client requests can be satisfied from a local cache and the service provider is relieved from the burden of servicing a large number of concurrent clients.

An inevitable side-effect of caching is the need to maintain copies of the same cached object consistent, at least to some degree. In this paper, we explore, both theoretically and practically, a flexible approach to consistency of such services. That is, we start by formally defining a basic set of consistency conditions, in an abstract, implementation independent, way, and show how other consistency conditions can be implemented as a combination of these basic conditions. We then describe how we have implemented these conditions within CASCADE [9], our CORBA object caching service [1]. The paper also describes some interesting optimizations we employed in this implementation.

Our specifications and resulting implementation have the following distinct features:

**Rigor**: We provide a formal definition of basic consistency conditions, given from the application point of view, as requirements on possible ordering of clients' local histories. As discussed in [5], this implementation independent approach yields more rigorous definitions, and it is easier to prove program correctness with such definitions than with operational definitions.

**Modularity**: Our conditions can be combined in various ways to yield guarantees with different levels of strength and complexity. This approach allows the known tradeoff between the strength of the consistency semantics and the overhead it imposes (cf. [5]) to be taken into consideration when configuring the set of consistency guarantees for a particular application. For users of our service, this means that they have more freedom in choosing the exact consistency semantics they need. From the implementation standpoint, this yields a more modular implementation. Since the implementation can be divided into basic conditions, each of which is easier to implement than, say sequential consistency [15], the entire implementation is simpler, and therefore more robust. Similarly, the implementation correctness proof is easier, since we can prove the correctness of the implementation of each basic condition separately; the formal proof about the combination of these conditions then immediately implies that our service correctly implements the corresponding more elaborate consistency conditions, e.g., sequential consistency.

**Comprehensiveness and usefulness for applications**: The presented specifications cover a wide range of consistency requirements for distributed applications. This is shown by proving that many existing consistency conditions such

[1] In [9], we described CASCADE, the motivation behind it, its general implementation and a performance analysis. The current paper is the first place where we formally specify the basic consistency conditions, and elaborate on their exact implementation within CASCADE.

as sequential consistency [15], PRAM [16], and causal consistency [2, 3] can be specified as certain combinations of our basic conditions. We also discuss usefulness of other combinations and analyze the inter-dependencies within the set of guarantees. In the full version of this paper we present examples of several applications, each of which requires some of our guarantees or a combination of them. Moreover, we show there that all of our conditions are indeed useful, i.e., that there are applications that require each of them.

Although, our implementation of consistency conditions is based on the widely known notion of version number and vectors, it is nevertheless unique in exploiting the peculiarities of hierarchical cache architecture such as the one used in CASCADE. We envision that similar techniques can be applied to other systems that employ hierarchical caches.

Finally, our implementation preserves consistency guarantees even when clients access cached copies at different servers during the execution. Furthermore, we have designed novel optimizations that reduce the amount of information transferred between roaming clients and static servers. We believe that this latter contribution can be applied to other systems where it is required to maintain consistency for mobile clients. As Internet mobile clients become more common, we expect that our techniques will be useful for a wider range of applications.

### 1.1 Related Work

Many consistency conditions have been defined and investigated, mostly in the context of distributed shared memory, e.g., [3, 5, 11, 15, 16, 19, 21] and data-bases [17, 24]. Vast amount of research was dedicated to implementing shared memory systems with various consistency guarantees, including *sequential consistency* (sometimes referred to as *strong consistency*) [15], *weak consistency* [12], *release consistency* [8], *causal consistency* [2, 3], *lazy release consistency* [14], *entry consistency* [6], and *hybrid consistency* [10]. In contrast to our service, such systems are geared towards high-performance computing, and generally assume non-faulty environments and fast local communication.

Much less attention, however, was devoted to exploring consistency guarantees suitable for object-oriented middlewares, especially for middlewares in which a client is not bound to a particular server and can switch the servers all the time.

Our work is motivated by Bayou project [24], which introduced a set of basic consistency conditions for sessions of mobile clients and discussed version vectors as a possible way of their implementation. This work also brought numerous examples illustrating that these conditions are indeed useful for applications. However, these definitions are introduced in [24] as constraints on an implementation and are defined in a framework of a particular database model.

The Globe system [25] follows an approach similar to CASCADE by providing a flexible framework for associating various replication coherence models with distributed objects. Among the coherence models supported by Globe are the PRAM coherence, the causal coherence, the eventual coherence, etc.

## 2 Definitions and Conventions

We generally adopt the model and definitions as provided in [4] and [1], but slightly adjust them to our needs. We assume a world consisting of *clients* and *servers*. Clients invoke *methods* on objects as specified in a *program*. These methods are then transformed into messages sent to one or more servers. The servers can exchange messages among themselves and eventually send a reply to the client. We assume that all messages are sent on a reliable network and that processes do not fail, but the network might delay messages for an arbitrarily long time and neither clients nor servers have access to real-time clocks. This failure model is discussed in Section 4.1.

We assume that each method operates on one object, but each object might have several read/write variables. Our definitions below are given from the client point of view and thus, for the rest of this section, we will no longer discuss servers; servers will be important in discussing the implementation (Section 4). Also, our definitions and discussions assume one object [2]. Note that since each object has multiple variables, each object can be thought of as a single distributed shared memory.

We assume that methods can be classified as either queries or updates, depending on whether they simply return the value of variables they access, or change them. To make the definitions comparable to the ones used in distributed shared memory research, we will refer to updates as WRITEs and to queries as READs. Each method can either read or write several variables atomically. In particular, a single READ operation might return values written by several WRITE operations.

A *local execution* of a client process $p_i$, denoted $\sigma_i$, is a sequence of READ and WRITE operations, denoted $o_1$, $o_2$, ..., that are performed by $p_i$. We assume that client's operations are always ordered in its local history in the order specified in the program. For the sake of simplicity, we will omit the variables accessed by an operation whenever possible. In what follows, we sometimes refer to local execution as session, and use these terms interchangeably. A *global execution*, or just *execution* $\sigma$, is a collection of local executions for a given system run, one for each client of the system.

Given a sequence $S$ of operations, we denote $o_1 \xrightarrow{S} o_2$ when $o_1$ precedes $o_2$ in the sequence. An execution $\sigma$ induces a partial order, $\xrightarrow{\sigma}$, on the operations that appear in $\sigma$: $o_1 \xrightarrow{\sigma} o_2$ if $o_1 \xrightarrow{\sigma_i} o_2$ for some $p_i$.

For a given execution $\sigma$ and a process $p_i$, denote by $\sigma|i$ the restriction of $\sigma$ to events of $p_i$; denote by $\sigma|i+w$ the partial execution consisting of all the operations of $p_i$ and all the WRITE operations of other processes. Similarly, for a given sequence $S$ of operations, denote by $S|i$ the restriction of $S$ to operations invoked by $p_i$ and denote by $S|w$ the restriction of $S$ to WRITE operations.

[2] This is sufficient for CASCADE in which consistency conditions are indeed provided per each object since each object has a separate hierarchy. However, in the future it would be interesting to extend the definitions to multiple objects.

A *serialization* $S$ of the execution $\sigma$ is a linear sequence containing all the operations of $\sigma$. A serialization is legal if each READ operation reading from several variables returns the result of the latest WRITE operations in the serialization to the corresponding variables.

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

# 3 Consistency Conditions

## 3.1 Basic Consistency Conditions

**Eventual Propagation:** For every process $p_i$ there exists a legal serialization $S_i$ of $\sigma|i+w$.

This requirement essentially expresses liveness of update propagation: For a given execution and a given update in this execution, if some process invokes an infinite number of queries, it will eventually see the result of this update. An implementation in which updates are not propagated does not guarantee any level of consistency. Henceforward, we assume that this condition always holds.

Let us define a *serialization set of* $\sigma$ as a set of legal serializations of $\sigma|i+w$, one for each $p_i$. Due to Eventual Propagation, at least one serialization set exists for a given execution.

We now present five session guarantees. Each guarantee is defined as a predicate that takes a serialization or a serialization set and verifies whether this set satisfies the condition w.r.t. a session.

**Read Your Writes:** For a given execution $\sigma$ and a process $p_i$, a valid serialization $S_i$ of $\sigma|i+w$ preserves *Read Your Writes for the session* $\sigma_i$ if for every two operations $o_1$ and $o_2$ in $\sigma_i$ such that $o_1 =$ WRITE, $o_2 =$ READ, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $o_1 \xrightarrow{S_i} o_2$.

**FIFO of Reads:** For a given execution $\sigma$ and a process $p_i$, a valid serialization $S_i$ of $\sigma|i+w$ preserves *FIFO of Reads for the session* $\sigma_i$ if for every two operations $o_1$ and $o_2$ in $\sigma_i$ such that $o_1 =$ READ, $o_2 =$ READ, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $o_1 \xrightarrow{S_i} o_2$.

**FIFO of Writes:** For a given execution $\sigma$ and a process $p_i$, a serialization set $S = \{S_j\}$ preserves *FIFO of Writes for the session* $\sigma_i$ if for every two operations $o_1$ and $o_2$ in $\sigma_i$ such that $o_1 =$ WRITE, $o_2 =$ WRITE, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $\forall p_j\ o_1 \xrightarrow{S_j} o_2$.

**Reads Before Writes:** For a given execution $\sigma$ and a process $p_i$, a valid serialization $S_i$ of $\sigma|i+w$ preserves *Reads Before Writes for the session* $\sigma_i$ if for every two operations $o_1$ and $o_2$ in $\sigma_i$ such that $o_1 =$ READ, $o_2 =$ WRITE, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $o_1 \xrightarrow{S_i} o_2$.

**Session Causality:**[3] For this definition we assume that no value is written more than once to the same variable. For a given execution $\sigma$ and a process $p_i$, a serialization set $S = \{S_j\}$ preserves *Session Causality for the session* $\sigma_i$ if for every three operations $o_1$, $o_2$ and $o_3$ such that $o_2$ and $o_3$ are in $\sigma_i$, $o_1 =$ WRITE, $o_2 =$ READ, $o_3 =$ WRITE, $o_2$ read a result written by $o_1$ and $o_2 \xrightarrow{\sigma_i} o_3$, holds $\forall p_j\ o_1 \xrightarrow{S_j} o_3$.

As noticed in [24], while Read Your Writes, FIFO of Reads and Reads Before Writes only affect the sessions for which they are provided, Session Causality and FIFO of Writes contain guarantees w.r.t. the executions of other processes. Accordingly, we define the former conditions for a single serialization and the latter conditions for a serialization set. However, the following definitions require the same form for all the conditions. Therefore, we assume below that Read Your Writes, FIFO of Reads and Reads Before Writes are defined for a serialization set in which only a single serialization is used in the definition (the definitions in this latter form can be found in the full version of this paper).

For any condition $X$ of these five session properties and a given execution $\sigma$, we say that a serialization set $S = \{S_j\}$ *globally preserves* $X$ if it preserves $X$ for all the sessions $\sigma_i \in \sigma$.

We now introduce a definition of the *Total Order* condition:

**Total Order:** For a given execution $\sigma$, a serialization set $S = \{S_j\}$ globally preserves *Total Order* if for every two serializations $S_i$ and $S_j$ in $S$, $S_i|w = S_j|w$.

For a given execution $\sigma$, a serialization set $S = \{S_j\}$ globally preserves some set of the conditions defined above if $S$ globally preserves each condition in this set. Finally, we say that an execution $\sigma$ is consistent with respect to a condition set (or a single condition) $X$ if there exists a serialization set $S$ of $\sigma$ such that $S$ globally preserves $X$. We say that an implementation $A$ *obeys a condition set* (or a single consistency condition) $X$ if every execution generated by $A$ is consistent with respect to $X$.

### 3.2 Examples of Known Consistency Conditions

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

**Sequential Consistency (SC) [15]:** An execution $\sigma$ is sequentially consistent if there exists a legal serialization $S$ of $\sigma$ such that for each process $p_i$, $\sigma|i = S|i$.

**PRAM Consistency [16]:** An execution $\sigma$ is PRAM consistent if for every process $p_i$ there exists a legal serialization $S_i$ of $\sigma|i + w$ such that if $o_1$ and $o_2$ are two operations in $\sigma|i + w$ and $o_1 \xrightarrow{\sigma} o_2$, then $o_1 \xrightarrow{S_i} o_2$.

[3] Called *Writes Follow Reads* in [24]

Note that instead of requiring a legal serialization $S_i$ for every process $p_i$ this definition can be rephrased to require an existence of a serialization set. We will use this latter form in order to define conjunction of PRAM consistency with other consistency conditions, e.g., in the theorems below. This latter form also appears in the full version of this paper.

**Causal Consistency [3]:** For the definition of causal consistency we assume that no value is written more than once to the same variable. Given an execution $\sigma$, an operation $o_1$ *directly precedes* $o_2$ (denoted $o_1 \stackrel{\sigma}{\longrightarrow} o_2$) if either $o_1 \stackrel{\sigma}{\longrightarrow} o_2$ or $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, and $o_2$ read a result written by $o_1$. Let $\stackrel{*}{\longrightarrow}$ denote the transitive closure of $\stackrel{\sigma}{\longrightarrow}$.
An execution $\sigma$ is causally consistent if for every process $p_i$ there exists a legal serialization $S_i$ of $\sigma|i + w$ such that $S_i$ respects $\stackrel{*}{\longrightarrow}$, i.e., if $o_1$ and $o_2$ are two operations in $\sigma|i + w$ and $o_1 \stackrel{*}{\longrightarrow} o_2$, then $o_1 \stackrel{S_i}{\longrightarrow} o_2$.

### 3.3 Discussion

Most consistency implementations preserve Reads Before Writes mainly because in most implementation a READ operation is blocking and execution is resumed only after a result is returned. We bring this condition here, however, for completeness and because it plays an important role in dependencies between consistency conditions. In the future, we intend to investigate the implications for the systems in which this condition does not hold.

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 thus we have the freedom to order the writes in the serialization so all the reads are legal. This applies similarly also to FIFO of Writes, Session Causality and Total Order. Thus, these guarantees become meaningful only in combinations that contain several guarantees of different types. The only guarantee that is not trivial by itself is Read Your 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. The proofs of these theorems can be found in the full version of this paper.

**Theorem 1.** *Any execution that is consistent w.r.t. Total Order and Reads Before Writes is also consistent w.r.t. Session Causality* [4].

**Conclusion:** Since Reads Before Writes holds in almost all implementations, the practical meaning of this theorem is that Total Order implies Session Causality.

**Theorem 2.** *Any execution that is consistent w.r.t. FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes is also PRAM consistent.*

[4] Note that being consistent w.r.t. a set of properties is a stronger property than just being consistent w.r.t. each property in the set.

*Vice versa, any PRAM consistent execution is also consistent w.r.t. FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes.*

**Theorem 3.** *Any execution that is PRAM consistent and is consistent w.r.t. Session Causality is also causally consistent. Vice versa, any causally consistent execution is also consistent w.r.t. Session Causality and PRAM.*

**Theorem 4.** *Any execution that is PRAM consistent and is consistent w.r.t. Total Order is also sequentially consistent* [5]*. Vice versa, any sequentially consistent execution is also consistent w.r.t. Total Order and PRAM.*

# 4 Implementation of Consistency Conditions in CASCADE

The general implementation of CASCADE has been presented in [9], but without specific details about the support for the basic consistency conditions that were presented in Section 3. We start this section by discussing the failure model of CASCADE. Then we proceed by covering general elements of CASCADE architecture that are needed to provide the right context for describing consistency implementation. Finally, we explain in detail how each individual consistency condition is implemented in CASCADE.

However, due to lack of space, we do not present here any pseudo code or proofs that our implementation obeys a given combination of basic consistency conditions; these appear in the full version of the paper.Furthermore, we do not discuss Reads Before Writes in this section: As explained in Section 3.3, Reads Before Writes trivially holds in any natural implementation.

## 4.1 Failure Model of CASCADE

As alluded to in Section 2, we assume that processes do not crash during the execution. While this assumption could be considered strong, it is widely acceptable in shared memory systems. Since the main difference between the Internet environment and the LAN environment of shared memory systems is in the quality of links rather than computers, this assumption is equally reasonable for CASCADE. At the same time, we are investigating different ways of rendering CASCADE less vulnerable to server failures.

While we account for communication link failures, we assume that all the links are eventually operational. CASCADE is built atop the underlying communication layer that employs widely known techniques to overcome temporary link failures and guarantee eventual reliable message delivery and FIFO order.

Thus, the main peculiarity of the environment in which CASCADE operates is arbitrarily long network delays and delivery latencies. This makes optimizations aimed at reducing the amount of transferred information particularly important.

[5] This claim can also be derived from the results of [21] whose focus, however, is different from ours.

### 4.2 Hierarchical Caching in CASCADE

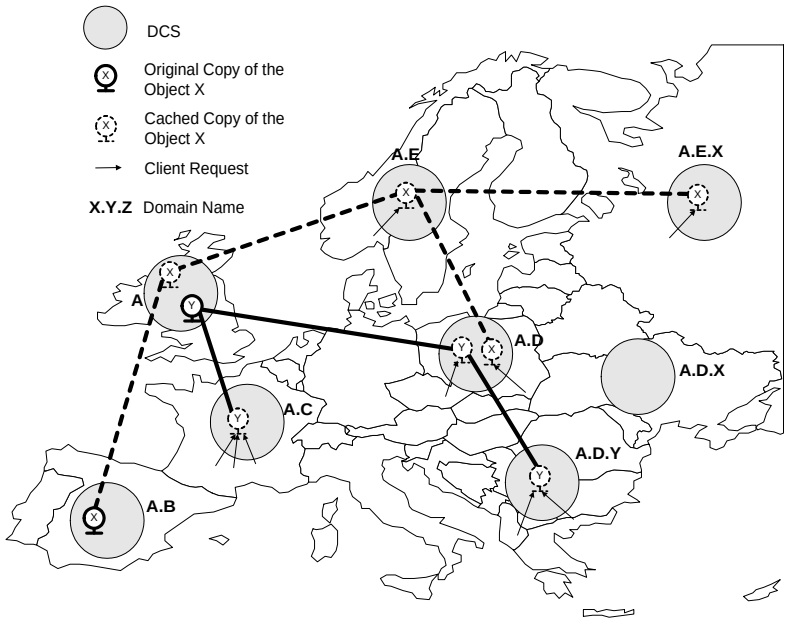


**Fig. 1.** The Caching Service Architecture

Our caching service is designed along the following lines (see Figure 1): The service is provided by a number of servers each of which is responsible for a specific *logical* domain. In practice, these domains can correspond to geographical areas. We call these servers *Domain Caching Servers (DCSs)*. Cached copies of each object are organized into a hierarchy. A separate hierarchy is dynamically constructed for each object. The hierarchy construction is driven by client requests. The construction mechanism ensures that for each client, client's local DCS (i.e., the DCS responsible for the client's domain) obtains a copy of the object. In addition, this mechanism attempts to guarantee that the object copy is obtained from the nearest DCS having a copy of this object. Once the local DCS has an object copy, client requests for object method invocation normally go to this DCS, so that the client does not have to communicate to a far server. Only if the local DCS becomes overloaded or unavailable, the client can decide to switch to another DCS. While we count for such a possibility in CASCADE, we consider it an unlikely event. Therefore, our implementation is optimized for the case when the client communicates with a small number of DCSs during its execution (see Section 4.5).

The DCS that holds an original object becomes the *root* for this object cache hierarchy. It plays a special role in building the hierarchy and in ensuring consis-

tency of the cached copies, as described below. Hierarchies corresponding to each object are superimposed on the DCS infrastructure: Different object hierarchies may overlap or be completely disjoint. Also overlapping object hierarchies do not necessarily have the same root. For example, in Figure 1 the original copy of the object $X$ is located in the DCS of domain $A.B$. This DCS is the root of the $X$'s hierarchy. The cached copies of $X$ are located in the DCSs of domains $A$, $A.E$, $A.D$ and $A.E.X$. Note that, in addition to being the holder of the cached copy of $X$, the DCS of domain $A$ also serves as the root of the object $Y$ hierarchy. Further, the $A.D$'s DCS contains only cached object copies and the $A.D.X$'s DCS does not contain objects at all.

In [9] we discuss the main pros and cons of this hierarchical structure compared with other distributed architectures. Briefly, using a hierarchy allows CASCADE to conserve bandwidth, be highly scalable, provide short initial response time, be easy to manage, and simplifies the consistency maintenance. On the downside, a hierarchy is vulnerability to node failures, and our future works tries to overcome this shortcoming by replication and other known forms of fault recovery [7].

### 4.3 Implementation of Eventual Propagation and Total Order

CASCADE always guarantees Eventual Update Propagation while the use of other conditions can be controlled by the application. To guarantee Eventual Update Propagation, queries are always locally executed at the DCS a client communicates to and updates are propagated through the hierarchy. However, the way updates propagate and the order in which they are being applied depend on whether Total Order is required.

If Total Order is not required by the application, Eventual Propagation is implemented as follows: A DCS that receives an update request from a client applies it locally and sends it to all its neighbors in the hierarchy in parallel. A DCS that receives an update request from a neighbor DCS $X$ applies the update and performs flooding, i.e., sends the request to all its neighbors but $X$. Note that this propagation protocol preserves per-DCS FIFO of updates because all the links are FIFO (as specified in Section 4.2) and because there is only one path in the hierarchy between any pair of nodes. Furthermore, per-DCS Session Causality also holds: If a DCS receives and applies an update, and then some client queries the object state and issues another update at this DCS, then the second update will be broadcast to the neighbors of this DCS after the first one. We will show later in this section how these facts can be exploited in order to provide an efficient implementation of session guarantees.

The Totally Ordered Eventual Propagation (i.e., the Total Order + Eventual Propagation conditions) is implemented as follows: Updates first ascend through the hierarchy towards the root. The root of the hierarchy orders the updates in a sequence, applies them and propagates ordered updates through the hierarchy downwards towards the leaves[6].

[6] This is somewhat similar to the sequencer protocol of Amoeba [13], but adjusted to the hierarchy structure.

Note that this implementation of Total Order is not affected by presence or absence of application demand for other consistency conditions. Moreover, this implementation is entirely based on the DCS algorithm and inter-DCS protocol, and does not require any client involvement.

Since our goal is to address Internet applications, where extremely long delays are common, we have made the design choice that update requests can return before the update has traversed the entire object hierarchy. The result, however, is that the implementation of session conditions requires client cooperation in most cases.

Also, the implementation of the session conditions adapts itself to the set of consistency requirements chosen by the application. In particular, their implementation is significantly affected by presence or absence of Total Order. Therefore, we discuss their implementation with and without Total Order separately.

### 4.4 Implementing Session Guarantees in Presence of Total Order

The implementation of the session guarantees is greatly simplified by the presence of the Total Order implementation. First, Session Causality is achieved for free, as Theorem 1 implies. Second, the root of the hierarchy can assign each update a global *update identifier* that serves as a version number of the object. Hence, an object version can be identified by a single number. As a result, the implementation of session guarantees becomes simpler, less information needs to be stored at both clients and DCSs, and most important, less consistency related data needs to be transferred between a client and a DCS per method invocation.

Specifically, with each query result, a DCS returns to the client the number of the object version this query sees. It would be more complicated to handle updates in a similar way because updates have to be propagated first to the root DCS which assigns them an update identifier. In principle, a DCS that received an update request from a client can block the client until the update identifier is received from the root DCS and then pass this version number to the client. This way, the only consistency information to be transferred between a client and a DCS would be a single global version number. However, since CASCADE is intended to operate in a WAN environment and the propagation latency between a client DCS and a root DCS may be quite significant, this solution may block the client for prohibitively long.

Therefore, CASCADE adopts an alternative identifier scheme for updates: Each DCS maintains a counter of updates originated at this DCS and each update is assigned a local update identifier consisting of the DCS identifier and a counter value. In contrast to the global version numbers, two local update identifiers assigned by different DCSs are incomparable. When a client invokes an update request on a DCS, the DCS immediately produces a new local update identifier and returns it to the client.

For implementation of some session guarantees we need to maintain a version vector for an object with one entry per DCS in the hierarchy; each entry in this

vector corresponds to the last local update identifier received from the corresponding DCS. Version vectors are maintained in the following way: When an update ascends through the hierarchy towards the root, the local update identifier is piggybacked on the update message. When this update is propagated from the root towards the leaves, its global version number and local update identifier are both piggybacked. Upon receiving and applying this update, a DCS updates its current object version number and version vector.

We now describe the individual implementation of the three session guarantees that require a non-trivial implementation:

**FIFO of Reads:** As previously explained, with each query result, a DCS returns to the client the object's version number that this query sees. The client passes this number to a (possibly different) DCS upon its next query. This DCS does not apply the query and blocks the client until it receives and applies the update referred to by the version number (in other words, the DCS *synchronizes* the query with the version number).

**FIFO of Writes:** For implementing FIFO of Writes, the root DCS should maintain a version vector which contains the last local update identifier received from each DCS. Keeping only the last update identifier is sufficient because Total Order preserves per-DCS FIFO of updates: Two updates issued at the same DCS reach the root where they are ordered in order of their issuance. As previously explained, when a client invokes an update request on a DCS, the DCS transfers a local update identifier back to the client. The client only remembers the last local identifier it received from some DCS and forgets all previous local identifiers. This is sufficient because FIFO of Writes is a transitive relation and it is enough to remember only the last predecessor.

The client passes the last known local identifier to a (possibly different) DCS upon the invocation of its next update request. The DCS piggybacks this identifier on the update message that traverses the hierarchy towards the root. The root DCS compares this identifier against the version vector and blocks the message until the update referred to by the identifier is received and applied.

**Read Your Writes:** For implementing Read Your Writes, each DCS maintains a version vector. When a client invokes a query request on a DCS, it passes the local update identifier(s) of the last update(s) it initiated. The DCS synchronizes the query with these identifiers based on the information stored in its version vector.

If Read Your Writes is provided along with FIFO of Writes, one last local update identifier is sufficient to be synchronized with because FIFO is a transitive relation. Otherwise, for each DCS the client sent an update request to, it should remember the last local update identifier received from this DCS. In this case, the query must be synchronized with the entire set of identifiers. However, since we assume in the model that a client only communicates with a small subset of all existing DCSs in the object hierarchy, the set of identifiers is also small and its transfer between a client and a DCS is not an expensive operation.

If Read Your Writes is provided along with FIFO of Reads, the amount of information to transfer and store at a client can be optimized in a different way: The client should only remember the local update identifiers it received since the last query. If the client first issues several updates and then two queries, the first query will be synchronized with the updates and the second query will be synchronized with the first one. Therefore, no explicit synchronization of the second query with the updates is necessary in this case.

Even if Read Your Writes is provided without FIFO of Reads and FIFO of Writes, other optimizations are possible based only on the Total Order. Due to space limitations, these optimizations are only described in the full version of this paper.

In summary, if Total Order is provided, the implementation of the session guarantees introduces an insignificant extra overhead: The amount of consistency information that needs to be stored at clients and transferred between clients and DCSs is small and does not depend on the number of clients and DCSs in the system.

### 4.5 Implementing Session Guarantees Without Total Order

When the Total Order implementation is not employed, an object does not have a single version number. In this case its state can only be characterized by the version vector that has to be maintained by each DCS. While this does not affect the implementation of Read Your Writes and the implementation of FIFO of Writes remains almost as simple as in the case of Total Order, the implementation of FIFO of Reads becomes more complicated and expensive. In addition, an implementation of Session Causality should now be provided. We elaborate on the changes in the implementations below:

**FIFO of Writes:** As with Total Order, a DCS returns a local update identifier to the client that initiates the update, a client remembers only the last local identifier and forgets the previous one, and this local identifier is transferred to a DCS upon the next update request. The only change is that now the DCS blocks the client and does not assign the update request a local identifier until it receives the referred update. If the DCS immediately produced a local update identifier, released the client and left the update request in a pending state, then all later (unrelated) update requests with higher local update identifiers would have to wait until this update would be applied. This is a shortcoming of the version vector method which assumes that updates originated at the same DCS are applied in the order of their local identifiers. An appealing alternative to blocking the client is to use a version vector of sliding windows instead of just a vector of update identifiers. In this solution updates can sometimes be applied in an order different from that of their local identifiers. However, a DCS has to remember the identifiers of the updates applied out of order. Therefore, while eliminating unnecessary delays,

this solution requires more space and more complicated version management. Moreover, this solution makes the implementation of FIFO of Reads complicated and inefficient.

**FIFO of Reads:** Without Total Order, the simplest implementation of this condition is that a DCS transfers the entire version vector to a client along with the results of a query. The client remembers the version vector it received the last time and forgets the previous vector. This vector is passed to a (possibly different) DCS upon the next client query, and the DCS synchronizes the query with each local identifier in the vector.
This implementation is inefficient because the entire version vector whose length is the number of DCSs in the object hierarchy is sent twice per each query. In Section 4.5 we introduce an optimization that allows us to reduce the average amount of transferred information.

**Session Causality:** Again, the simplest implementation is that a DCS transfers the entire version vector to a client along with the query results. However, unless FIFO of Reads is also provided, it is not sufficient that a client remembers only the version vector it received in the previous interaction with the DCS. Actually, the client must merge all the vectors it received during the execution by computing their maximum. This merged vector is passed to a DCS upon the next client update. Furthermore, since every DCS has to synchronize this update with this vector, the DCS piggybacks the entire vector on the update message sent to other DCSs. In the future, we intend to investigate the possibility of using the *causal separators* technique [22] in order to reduce the amount of piggybacked information. This technique appears especially appealing due to the hierarchical architecture employed by CASCADE in which each intermediate node can act as a causal separator.

As we see, when Total Order is not employed, the straightforward implementations of FIFO of Reads and Session Causality are quite expensive in terms of the amount of information to be transferred over the network. Fortunately, the implementation of FIFO of Reads can be significantly improved by using the optimization that is explained below.

**Efficient FIFO of Reads Implementation** First, rather than sending the entire version vector to a client as part of the response to queries, a DCS can send the difference between its current version vector and the vector received from the client for the purpose of synchronization [7]. This difference is usually much shorter than the entire version vector and contains only few elements. For example, the difference of $\langle\langle A,1\rangle,\langle B,3\rangle\rangle$ and $\langle\langle A,1\rangle,\langle B,1\rangle\rangle$ is $\langle\langle B,3\rangle\rangle$. Upon receiving such a vector difference the client can add it to the vector it sent and restore the entire version vector of the DCS in its local memory. However, the client should still send its entire version vector for synchronization.

Another optimization is based on the following observation: If a client does not switch DCSs (in other words, it invokes all updates and queries on the same

[7] This optimization is similar to Singhal-Kshemkalyani technique [23] for implementing vector clocks.

DCS), then FIFO of Reads always holds in a trivial way and does not need to be implemented at all. Furthermore, FIFO of Writes and Session Causality also trivially hold due to per-DCS FIFO of Writes and per-DCS Session Causality, respectively. This situation is summarized in Table 1 that clearly shows the cost of client mobility.

**Table 1.** The implementation cost of session guarantees

| Session Guarantee | with TO | | w/o TO | |
|---|---|---|---|---|
| | Mobile Clients | Not | Mobile Clients | Not |
| FIFO of Reads | × | √ | ×× | √ |
| Session Causality | √ | √ | ×× | √ |
| FIFO of Writes | × | √ | × | √ |
| Read Your Writes | × | × | × | × |

√ – trivially holds
× – adds extra cost
×× – requires costly communication

Unfortunately, the consistency implementation unaware that the client continues to work with the same DCS always transfers the same amount of information as in the case of switching DCSs. This observation calls for optimizing the implementation for the most usual and frequent case when a client communicates with a single DCS. In the simplest case, a client can just verify that it invokes a current request on the same DCS as the previous one. If this is true, the client does not need to send any information for synchronization.

However, the general situation with FIFO of Reads is more complicated: a DCS still has to return its version vector along with the query results in order to account for the possibility that a client invokes the next query on another DCS. Furthermore, if a client sends no synchronization information, we can no longer use the differential optimization described above because a DCS has no reference point to compute the difference of vectors.

Thus, there is a need for synchronization information shorter than just an entire version vector. To this end, we introduce a notion of *local DCS history* which is a numbered sequence of update identifiers of all the updates applied at the DCS during the execution. A *local history pointer* is just an index to local DCS history. A DCS can return this pointer to a client, and a client can transfer it back to the DCS for synchronization at some later point. As a result, only local history pointers and vector differences are transferred over the network instead of entire version vectors.

As part of this optimization, a DCS should be able to compute the difference between its current version vector and a pointer to some past point of its local history. An important question is how this can be done efficiently without keeping the whole local history. The full version of this paper provides a detailed explanation of the algorithm used in CASCADE that satisfies these requirements.

This optimization can be generalized for the case when client communicates with several DCSs: For each DCS a client invoked a query on, the client remembers the last local history pointer it received from this DCS and the version

vector corresponding to this pointer. (As we have already seen, this vector can be restored in the client memory without being transferred over the network). In other words, a client keeps a knowledge about how advanced each DCS is. Note that since FIFO of Reads is a transitive relation, the vector corresponding to the last query is always more advanced than all other vectors. More precisely, this vector represents the knowledge of the client itself about the last object version. When a client invokes a query on some DCS, the DCS transfers the last known local history pointer of this DCS along with the difference between the last version vector known to the client and the last version vector known for this DCS. When the client receives a new local history pointer and a vector difference piggybacked on query results, it updates its local knowledge about the DCS.

While this optimization requires that a client keeps a version vector for each DCS it communicates to, it is still very efficient because even a mobile client usually works with a small number of DCSs as noted in Section 4.2.

## 5 Future Work

It would be interesting 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 this conditions is necessary for implementing at least one consistency condition. In our opinion, this should be made at the application point of view, like our definitions and the works of [2, 3, 5], since such definitions are more rigorous, easier to understand, and can be used more easily by programmers to prove correctness of their applications.

As for the implementation, it is possible to implement each of the basic consistency conditions separately, and then trigger the required ones based on the application's need. We have decided not to follow this path, and to optimize the implementation of various conditions based on the other conditions being provided, since an independent implementation of each condition was too wasteful and slow. Perhaps the right way to tackle this issue is by providing an independent implementation for each condition, and then use a high-level compiler to optimize combinations of conditions, similar to the work on automatically optimizing and proving group communication protocol stacks in Ensemble [18].

## References


1. M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *Proc. of the 5th ACM Symposium On Parallel Algorithms and Architectures*, pages 251–260, June/July 1993.
2. M. Ahamad, P. Hutto, and R. John. Implementing and Programming Causal Distributed Shared Memory. Technical Report TR GIT-CC-90-49, Georgia Institute of Technology, December 1990.
3. M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1), 93.

4. H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. *SIAM Journal of Computing*, 27(1), February 1998.
5. H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.
6. B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Intl. Computer Conf. (COMPCON)*, pages 528–537, February 1993. Available at http://www-cgi.cs.cmu.edu/afs/cs/project/midway/WWW/CompCon93.ps.
7. K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, December 1996.
8. J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Rel ease Consistency.* PhD thesis, Department of Computer Science, Rice University, September 1993.
9. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proceedings of Middleware '00*, pages 1–23, April 2000. The Best Conference Paper award.
10. R. Friedman. *Consistency Conditions for Distributed Shared Memories.* PhD thesis, Department of Computer Science, The Technion, 1994.
11. M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
12. P. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. Technical Report TR GIT-ICS-89/39, Georgia Institute of Technology, October 1989.
13. M. F. Kaashoek and A. S. Tanenbaum. An Evaluation of the Amoeba Group Communication System. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 436–447, May 1996.
14. P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, December 1994.
15. L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.
16. R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Computer Science Department, Princeton University, September 1988.
17. B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD International Symposium on Management of Data*, pages 318–329, June 1996.
18. X. Liu, C. Keitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *Proceedings of the 17th Symposium on Operating Systems Principles*, December 1999.
19. M. Mizuno, M. Raynal, and J. Zhou. Sequential Consistency in Distributed Systems. In *Proceedings of the Intl Workshop "Theory and Practice in Distributed Systems"*, pages 224–241, September 1994.
20. OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, 1995.
21. M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. In *Proc. of the 15th Int. Conf. on Foundations of*

*Software Technology and Theoretical Computer Science*, pages 180–194, December 1995.

22. L. Rodrigues and P. Verissimo. Causal Separators for Large-Scale Multicast Communication. In *Proceedings of the 15th IEEE Intl. Conference on Distributed Computing Systems*, pages 83–91, June 1995.
23. M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47–52, August 1992.
24. D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welsh. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, Austin, TX, September 1994.
25. M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurency*, 7(1):70–78, January-March 1999.