# Implementing a Caching Service for Distributed CORBA Objects*

Gregory V. Chockler[1], Danny Dolev[1], Roy Friedman[2], and Roman Vitenberg[2]

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

**Abstract.** This paper discusses the implementation of CASCADE, a distributed caching service for CORBA objects. Our caching service is fully CORBA compliant, and supports caching of active objects, which include both data and code. It is specifically designed to operate over the Internet by employing a dynamically built cache hierarchy. The service architecture is highly configurable with regard to a broad spectrum of application parameters. The main benefits of CASCADE are enhanced availability and service predictability, as well as easy dynamic code deployment and consistency maintenance.

## 1 Introduction

One of the main goals of modern middlewares, and in particular of the CORBA standard [45], is to facilitate the design of interoperable, extensible and portable distributed systems. This is done by standardizing a programming language independent IDL, a large set of useful services, the Generic InterORB Protocol (and its TCP/IP derivative IIOP), and bridges to other common middlewares. Thus, CORBA compliant middlewares combined with the global connectivity of the Internet, creates a potential for truly global services that are available for clients anywhere in the world.

However, the long and unpredictable latencies of the Internet as well as its unreliability, complicate the realization of this potential. We conducted a simple test using the UNIX ping program to measure the Internet delays. The results of this test demonstrate the variance of latencies incurred by the Internet: Pinging a local host in the same LAN takes less than one millisecond, pinging a host in the Hebrew University from the Technion takes about 11ms, while pinging a machine in the USA takes almost 600ms. Therefore, the difference in the response

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

time in accessing objects spread over the Internet might be dramatic, regardless of the Object Request Broker (ORB) being used.

This calls for caching solutions for improving availability and predictability of distributed services. In this paper we propose to enrich CORBA middleware systems with generic caching service. We developed Caching Service for CORBA Distributed objEcts (CASCADE) which offers a scalable and flexible framework for general CORBA objects. CASCADE facilitates scalable application design by building cache hierarchies for the objects it manages. The hierarchy construction is dynamically adaptive with respect to service demand. As different applications have different consistency, security, and persistence requirements, our architecture is highly configurable with regard to a broad spectrum of application parameters. CASCADE allows client applications to fully control many aspects of object caching, by specifying a variety of *policies* for cache management, consistency maintenance, persistence, security, etc.

Our work is based on a high abstraction level provided by standard, commercially available CORBA compliant ORBs, and is aimed at preserving native programming models wherever possible. Furthermore, as discussed in Section 3, CASCADE design strictly follows the standard CORBA services design principles outlined in [46].

CASCADE caches both object data and code. Code caching allows us to preserve the standard CORBA programming model: The application works with the cached copy through the same interface it would have worked with the original object. In addition, all object methods (including updates) can be invoked locally eliminating the need to contact the remote object.

In this paper we report on the implementation of CASCADE, its performance, and the lessons learned from the implementation experience.

## 2 Related Work

### 2.1 Object Caching in CORBA Compliant Systems

To the best of our knowledge, there is no any programming framework provided by commercially available ORBs that allows for caching general CORBA objects. A very limited solution is provided by the so called *smart stub* mechanism provided by some existing CORBA implementations (e.g., Orbix [26] and VisiBroker). This mechanism allows an application programmer to override automatically generated client stubs. Using smart stubs it is possible to cache results of method invocations so that the subsequent method invocations will return locally cached values without invoking the remote operation. This framework, however, is not general enough for implementing a generic caching mechanism that will allow for caching true CORBA objects and not just some method-specific information. Moreover, with smart stub based caching the burden of maintaining coherency of cached object copies lies entirely on the application programmer.

MinORB [37] is a research ORB that allows caching partial results of read method invocations at the client side. Since this system is conceptually similar

to a caching solution that can be built using smart stubs, it bears the limitations incurred by this approach (see above).

The ScaFDOCS system [32] is another research project concerned with the object caching service for CORBA compliant systems. ScaFDOCS provides multiple consistency levels for copies of cached objects. This system supports several protocols for various cache consistency semantics [31]. However, it does not support caching of active CORBA objects (both data and code). In addition, though cache consistency protocols used in this system scale relatively well, the system architecture is not hierarchical and is not aimed to operate in wide area networks.

### 2.2 The Service Approach to Caching and Migration

Java$^{TM}$ [11] enables writing mobile programs that run on different hardware platforms and operating systems. Recently, several distributed systems and architectures (e.g., Voyager [43], FarGo [1] etc.) that utilize these features to provide a transparent or application-controlled object migration have emerged. Yet, these systems dictate their own programming model to the application.

In contrast, our work is based on a higher abstraction level provided by standard, commercially available CORBA compliant ORBs, and is aimed at preserving native programming models wherever possible.

### 2.3 Consistency in Shared Memory and Distributed Systems

Our caching service is highly configurable with respect to a great variety of consistency disciplines that can be enforced on the cached object copies. Here we benefited from the vast amount of research that was dedicated to implementing shared memory systems with various consistency guarantees, including *sequential consistency* (sometimes referred to as *strong consistency*) [33], *weak consistency* [25], *release consistency* [19], *causal consistency* [3, 4], *lazy release consistency* [29], *entry consistency* [14], and *hybrid consistency* [22]. In contrast to our service, such systems are geared towards high-performance computing, and generally assume non-faulty environments and fast local communication. We refer the reader to [7, 8, 22, 53] for other applied and theoretical studies of consistency strategies.

The Globe system [49] 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.

The recent LOTEC protocol [24] maintains consistency for nested object transactions. This protocol spares the programmer the burden of explicitly specifying synchronization operations needed for transactional processing. Finally, the novel Millipage [27] technique enables efficient control over the granularity of a shared unit, thus enabling applications to achieve good performance while maintaining sequential consistency.

*Object-based* shared memory systems, in which consistency guarantees are given per object, have also been studied. Orca [10] supports object replication

and migration with strong consistency guarantees. However, all objects in this system must be written in a special Orca language.

Spring [41] is a distributed operating system that provides a unified caching architecture that can be used for caching different types of remote objects. However, this system does not provide generic support for a variety of consistency and other requirements inherent for object caching. In order to address the requirements of a specific object type, the application developer must reimplement the object itself.

### 2.4 Object-Oriented Database Systems (OODS)

Some OODS have been designed to provide persistent storage for objects that include methods, e.g., O2 [21], GemStone [16] and Thor [35]. For example, the Thor system [35] supports highly-reliable and highly-available access to storage. For this purpose its object repositories (ORs) can be replicated, and objects can migrate from one OR to another. This system also supports client-side caching, providing a mix of consistency guarantees that can be determined by the user. It uses a variant of copying garbage collection [9] to manage the cache. In order to achieve *type-safe sharing*, all object implementations in Thor are required to be written in the Theta [36] programming language.

### 2.5 Web Caching

Caching web pages is nowadays a hot research topic (see, e.g., [2, 5, 12, 17, 38, 39, 44, 47, 51, 52]). Web caching is intended for use in wide area internets, similarly to our caching service, and scalability is also a major concern here. However, the web caching model is limited to data objects (HTML pages) only and does not deal with general objects that include executable code. Moreover, the contents of caches can change only as a result of a primary copy update.

The idea of *Active Web Caching* [18] can be considered a step towards active object caching. This work proposes to attach an applet to each web document. When a document (or its cached copy) is retrieved, the applet is executed.

Another point to be stressed about web caching is that the user has extremely limited control over caching decisions. The work of [42] describes several typical Enterprise network scenarios when incorrect caching decisions eliminate any improvements in response time gained from using web caching. In contrast, caching decisions in our service are application-controlled. Therefore, proper application choice can eliminate all of the above mentioned problems.

## 3 CASCADE and the Standard CORBA Services

The CORBA standard specifies a collection of object services, called *CORBAservices* that support basic functions for using and implementing objects [46]. These services are needed to support meaningful and productive communication at the application level and are useful for any CORBA applications regardless of their

specialization. The most prominent examples of the CORBA services implemented by almost any ORB vendor are the Naming Service, the Event Service and the Transaction Service.

In this respect our design goal was twofold: to preserve the programming framework provided by the standard CORBA services, and to allow co-existence and cooperation with these services. To achieve this, we strictly followed the standard CORBA services design principles outlined in [46] (see below). Thus, our caching service can be viewed as another useful object service aimed to improve responsiveness and availability of any CORBA-based application independent of application domain.

The following is a summary of the CASCADE features that place it in line with the standard CORBA services:

- The design of CASCADE uses and builds on CORBA concepts: separation of interface and implementation, object references are typed by interfaces, clients depend on interfaces, not implementations, etc.
- The service provided by CASCADE is generic with respect to cached object types.
- CASCADE allows local and remote implementations: The caching service is structured as a CORBA object with an OMG IDL interface and can be implemented as either an application library or a standalone server.
- CASCADE does not depend on any global identifier service or global id space in order to function. All the internal CASCADE components that require some kind of identification rely on ids generated internally by CASCADE. These ids are unique only within the caching service scope and are invisible for the client applications.
- Finding the caching service is at a higher level and orthogonal to using the service. Since the caching service is structured as an object, all that is needed for accessing the service is its interoperable object reference (IOR). The latter can be found using any general purpose service (e.g., the Naming Service).

## 4 CASCADE System Overview

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 (see Section 6.1 for further details). Once the local DCS has an object copy, all client requests

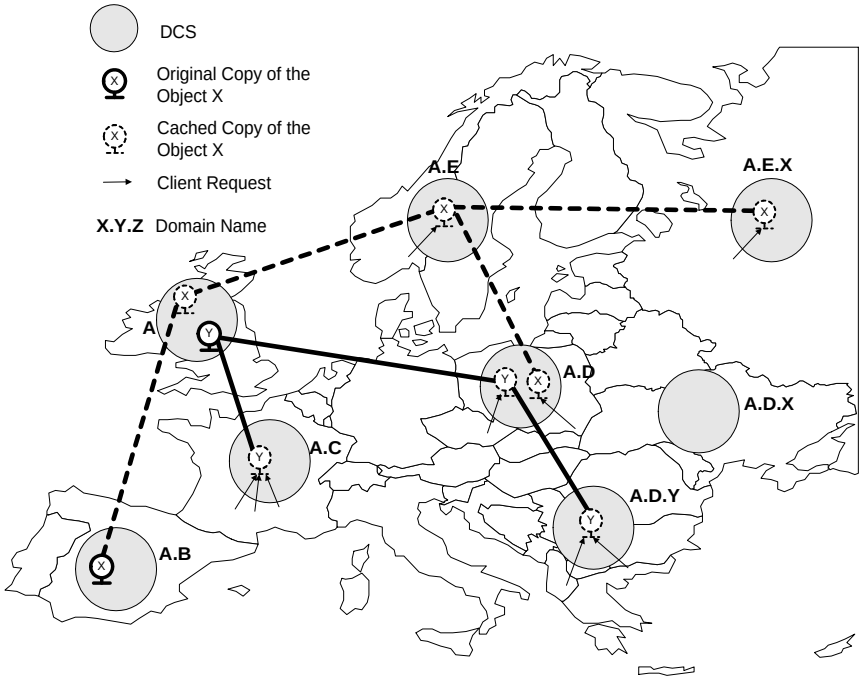



**Fig. 1.** The Caching Service Architecture

for object method invocation go to this DCS, so that the client does not have to communicate to a far server.

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 consistency 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.

Compared with other distributed architectures, using a hierarchy has the following advantages:

**Conserved WAN bandwidth consumption** : The bandwidth consumption of communication over a tree is low because each message is sent only $|V|-1$ times, where $V$ is the number of nodes in the tree.

**Improved scalability** : The stateful communication over a hierarchical architecture is known to be scalable because of the low number of simultaneously opened node to node connections and because of the small communication state kept at each node.

**Reduced initial response time** : Since an object copy is obtained from the nearest member of the object hierarchy, it takes less time on average to bring this copy to a local DCS than to obtain the copy from the original object holder. Thus, the client waits less for the result of its first request addressed to this object.

**Easy management** : It is easier to add or to remove a server in a tree than in other distributed architectures. In addition, a tree can be relatively easily reconfigured (a root of the tree can be moved etc.).

**Easy consistency maintenance** : If strong consistency among cached copies is required, a universally known root of the hierarchy can impose a total order on object updates.

The main disadvantage of a tree is its vulnerability to node failures. A single node failure can disconnect the whole branch of the tree. This problem can be solved, for example, by local replication of each DCS using a primary backup approach (see, e.g., [15]).

Note that the communication between DCSs is also implemented via CORBA, so that each DCS provides a service for other DCSs and implements a well defined IDL interface for internal requests. Thus, the inter-DCS communication benefits from all CORBA advantages: interoperability, portability, reliability guarantees for communication, and a wide spectrum of services which can be used, e.g., for secure communication and for name resolution. It is also important to emphasize that there is only one central object at each DCS implementing this interface for internal requests and not one object per each cached object. This reduces the number of stubs used for DCS to DCS communication to one, rendering the system more scalable.

In the framework of the caching service we introduce two policy classes: *per-object* policies and *per-request* policies. The per-object policies do not change over the object life time, whereas the per-request policies specify a particular request semantics. In order to provide an adequate interface for configuring the system, we introduce the notion of a *policy object*, which is associated with either an object or a request. These policies are used to specify the required behavior of the caching service in terms of consistency guarantees, security, persistence, etc.

# 5 CASCADE System Modules

## 5.1 The Client Structure

The client structure is depicted in Figure 2. It consists of the following elements:

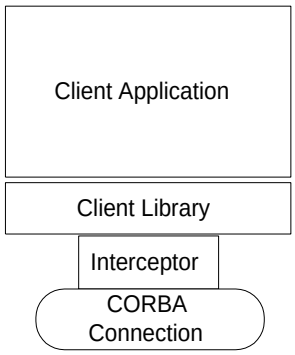



**Fig. 2.** The Client Structure

**Interceptor** [1]: This module is responsible for interception of client invocations of cached object methods and for altering the content of the request transparently for the client application. This module is used for passing implicit request parameters and invocation semantics (see Section 6.2). It is also used for encryption, decryption and authentication of client requests (see Section 6.5).

**Client Library** : The client library is aimed at facilitating the interaction between the client application and the caching service. In particular, the library hides the interceptor from the application. Thanks to the client library, the application interacts with the cached object in the same way as it would do with an ordinary CORBA object.

### 5.2 The DCS Structure

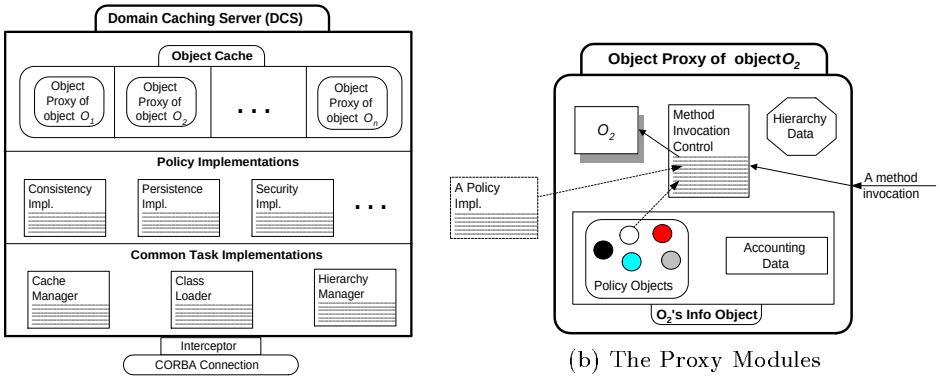
Figure 3(a) shows the DCS breakdown into modules. The DCS object consists of three parts:

**Object cache** : This is where objects cached in the local DCS actually reside. Each cached object is wrapped into a *proxy object* whose structure is depicted in Figure 3(b) and discussed in Section 5.3 below.

**Policy implementations** : This is a collection of implementations of various per-object policies. Each policy implementation is shared among all cached objects. However, for each cached object a policy implementation is parameterized by the corresponding *policy object*. Policy objects are located within the object proxy (see Figure 3(b) and Section 5.3).

**Common task implementations** : This part includes the implementations of the cache manager, the class loader and the hierarchy manager. The cache

[1] Interceptors are a common way of gaining access to CORBA's communication protocol; they are a part of the CORBA 2.3 standard [40]. While they are to undergo a technical revision in CORBA 3, no conceptual changes are anticipated.




(a) The DCS Modules

(b) The Proxy Modules

**Fig. 3.** The CASCADE System Modules

manager controls insertion/deletion of objects to/from the object cache. Its implementation is based on a particular cache replacement policy implemented by the DCS (see Section 6.4). The class loader is responsible for loading the cached objects' code into the Java virtual machine. Finally, the hierarchy manager controls the hierarchy construction for all objects cached within the DCS (see Section 6.1).

In addition, there is an interceptor object underneath the DCS that intercepts client requests to cached objects. This is used to extract the implicit request parameters added by client side interceptors (see Section 5.1).

### 5.3 The Proxy Object Structure

Figure 3(b) shows the internal structure of the proxy of a cached object $O_2$. It consists of the following elements:

**The cached copy of $O_2$** : This includes the copy of $O_2$'s state and code.

**$O_2$'s info object** : This object consists of two parts: $O_2$'s policy objects and $O_2$'s accounting data. There is one policy object for each policy defined for $O_2$. The policies are configured when $O_2$ is first registered with the CASCADE system. $O_2$'s accounting data includes various run-time statistics that are collected during its life-time.

**$O_2$'s Hierarchy Data** : The knowledge of this proxy node about its location in the object hierarchy: its father node, its children nodes etc.

**The method invocation control** : This module processes incoming method invocations based on the policies defined for the object: Whenever a method is invoked on the cached object, it is forwarded to the object proxy. The invocation is then processed by the method invocation control module that passes control to the appropriate policy implementations and supplies the requested invocation semantics (per-method policy) and the per-object policies as parameters.

# 6 CASCADE Implementation in Detail

## 6.1 Hierarchy Construction

A hierarchy construction is started when a client calls *register_object* in order to create a new hierarchy for the object (or for the group of objects). This call registers the object (or the group of objects) with the local client's DCS which becomes the root of the new hierarchy. In the following description we call it a *root DCS* for this hierarchy. It keeps the knowledge of the whole hierarchy and it is responsible for the hierarchy construction. The root DCS also plays a special role in achieving consistency among the cached copies of the object, as we explain in Section 6.2. In addition, the root DCS registers itself with the naming service as a caching service provider for its cached object (or the group of objects).

Figure 4 shows the sequence of operations executed when a new DCS joins the hierarchy. When a client wishes to start working with a cached object, it calls *copy_object* on its local DCS. Unless the local DCS has this object already cached, it finds the root DCS for this object with the aid of the naming service. Then it contacts the root DCS with a request to join the hierarchy for this object. This request also contains the domain of the new DCS.

When the root DCS receives such a join request from a DCS of domain $D$, it finds a domain $D'$ in the existing hierarchy such that $D'$ is the closest to $D$. Then, the root DCS sends a reply specifying the location of the $D'$'s DCS to the local client's DCS. Upon receiving this reply, the latter sends a request to its designated father node in order to register as its son in the hierarchy and in order to obtain a cached copy of the object. The father node registers the new son and sends its cached copy, thus, completing the join protocol.

The notion of a distance between different domains is still an open question. Currently, we use symbolic domain names to determine the distance between domains, e.g., we assume that domain *a.b.c* is closer to *a.b* than to *x.y.z*. This approach works well for some Internet domain names, e.g., domain *technion.ac.il* is indeed closer to *huji.ac.il* than to *whitehouse.gov*. On the other hand, this scheme cannot differentiate between the enormous number of *.com* domains, and thus, we are also going to investigate other approaches.

The hierarchy construction protocol guarantees that for each client there is a local DCS that has a cached copy of the object; the local DCS handles the client's requests. It is this fact and the hierarchical architecture of the system that allow to significantly reduce the response time, to distribute the load on DCSs and to render the caching service scalable.

The disconnection from the hierarchy is currently supported only for leaves. If some object is no longer used, and some intermediate DCS node wants to leave the hierarchy for this object, this DCS should wait for its sons to disconnect first.

When a leaf DCS node $D$ wishes to disconnect from the object hierarchy, it first informs the root about this, so that the root updates its knowledge about the hierarchy. Then $D$ sends its father a request to detach it from the hierarchy. Only when $D$ receives a reply, it can safely disconnect.

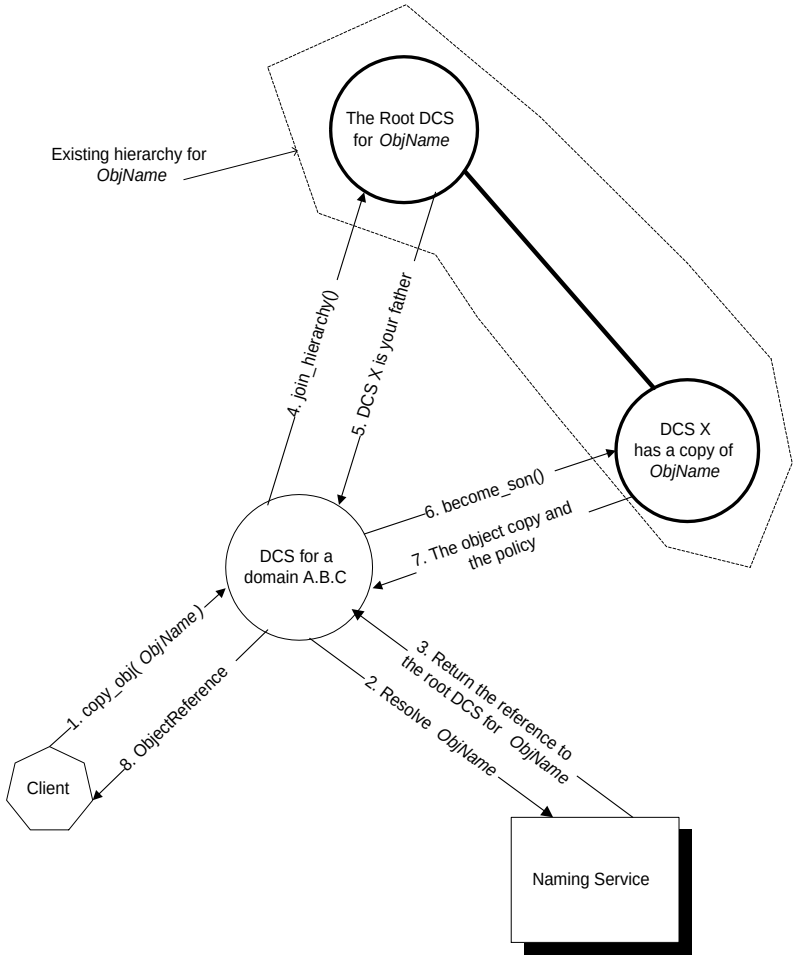



**Fig. 4.** Joining the hierarchy

It should be emphasized that CASCADE takes care of all synchronization problems that arise during joining and leaving the hierarchy.

### 6.2 Implementation of Cached Copies Consistency

When the system concurrently maintains several copies of the same object, it should also guarantee mutual consistency of these copies. Different levels of consistency are widely known, e.g., [13, 14, 23, 28, 33]. In general, the stronger the consistency level, the higher latency its implementation incurs [8].

In order to make our service as flexible as possible, we support several *consistency policies*. The set of supported policies is motivated by the guarantees

presented in [48] and [30]. Following [48] we present 6 consistency guarantees that can be combined together to form a consistency model.

**Update Propagation** This requirement, called also update dissemination, ensures that each update is eventually received by each DCS.

**Read Your Writes** This condition ensures that the effects of every update made by an application are visible to all subsequent queries of this application.

**Monotonic Reads** This condition requires that the effects of every update seen by an application query are visible to all subsequent queries of this application (unless overwritten by later updates). The Monotonic Reads guarantee implies that the observed object is increasingly up-to-date over time.

**Monotonic Writes (FIFO ordering)** This requirement guarantees that two updates initiated by the same application are applied in order of their issuance.

**Writes Follow Reads** This condition entails that the updates whose effects are seen by an application query are applied before all subsequent updates issued by this application. The Writes Follow Reads requirement along with Monotonic Writes ensure *causal ordering of updates* [34].

**Total Ordering** This requirement guarantees that all updates are applied in the same order by all GCSs. In other words, it implies that there is a global sequence of updates. Total Ordering entails Writes Follow Reads (see [50] for a discussion about the relation of different ordering properties).

CASCADE always guarantees eventual Update Propagation. Other requirements are fulfilled only if requested by the application.

The use of Total Ordering is determined by the object policy. It affects the way method invocation requests are applied and/or propagated through the hierarchy: While queries are always locally executed at the DCS of a client, updates might need to be propagated through the hierarchy before being applied at the DCS which initiates them. If total ordering is required, 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. At this point there are two possibilities (the choice is policy configurable): the root DCS can either propagate the request itself or the resulted version of the object, whichever is shorter. Of course, if the request has a side effect of updating another, non-cached object through the network (e.g., updating a non-distributed database), then the root can only propagate the new object version.

If total ordering is not deployed, updates are first applied locally and then propagate through the hierarchy. This shortens the latency of updates, and reduces the load imposed on the root DCS. However, different updates may be applied in different order at different DCSs.

Other consistency requirements (i.e., Read Your Writes, Monotonic Reads, Monotonic Writes and Writes Follow Reads) are part of a per-request policy. This is more flexible than specifying the required set of these guarantees once

for the whole client session as implemented in Bayou. [20] contains a detailed description of how these consistency requirements are implemented. It should be noticed that in some cases (in particular when total ordering is not used) their implementation implies a cooperation of the client. Along with a method invocation request, a client should provide consistency information such as the last update the client has seen. This can be done either explicitly through a special purpose interface or transparently for the client application by using interceptor library (see [20] for a detailed description of the client interface).

As noted in [48], these four consistency conditions (required by all the clients) in conjunction with total ordering guarantee the classical *strong* or *sequential* consistency [33].

**Object Group-Based Consistency** The above mentioned consistency policies apply to an individual object. However, some applications might wish to impose sequential consistency across several objects. To address this, we introduce *object group-based* consistency policy. With this policy, a group of objects, each one having its own hierarchy, are to be maintained in a strongly consistent manner. We impose a restriction for this policy that these hierarchies must have a common root. Without such a limitation the algorithms for achieving group consistency become prohibitively expensive.

This policy is implemented in the following way: The common root introduces a total order on the updates of all the objects in the group. Then, updates of each object are propagated through its own hierarchy. However, if some DCS has cached copies of more than one object in the group, this DCS does not apply updates to any object before it applies all other preceding updates, including updates of other objects.

### 6.3 Support for Atomic Operations and Locking

Sometimes several requests are to be executed *atomically*, without being interrupted by the execution of other requests. CASCADE provides the client a possibility to specify several update operations in one request. Each DCS applying this request atomically processes the specified sequence of update operations. When an object group-based consistency is used, and the client's local DCS has cached copies of two objects $X$ and $Y$ belonging to the same group, then the client can invoke *update_object* on $X$, and include an update of $Y$ into the atomic operation request.

Locking can be a very useful functionality for applications that deploy a caching service with strong consistency semantics. When the local DCS possesses a lock for an object, its clients can be sure that they work with the most updated object copy. In addition, locking can be used for implementing *distributed transactions*, even for objects with different root DCSs.

CASCADE supports three types of object locks defined in the specifications of CORBA Concurrency Control Service [46]: *Write* lock conflicts with any other lock, *read* lock conflicts only with a write lock, and *upgrade* lock conflicts with a write lock and with other upgrade locks.

A lock is assigned to the client application following its request. Normally, a lock is released also upon a client request. However, since CASCADE cannot rely on the reliability of the client, special care should be taken so that the object would not remain locked forever. Therefore, the DCS that obtained the locks releases them automatically after a given timeout unless it receives another lock request for the same object. Thus, if the client intends to continue updating the object, it should issue another lock request before the timeout elapses.

### 6.4 Overview of Cache Management in CASCADE

Since the size of the cache is limited, a DCS can hold only a limited number of cached copies. When a cached copy is to be brought to a DCS but there is no more space in the cache, some other object is to be evacuated from the cache. However, not all cached objects are suitable for evacuation. In particular, objects locked by this DCS and/or objects whose methods are currently being executed are never replaced. In addition, since each object registered with CASCADE should remain available until it is explicitly unregistered, objects for which this DCS is the root of the hierarchy are also never evacuated from the cache.

In order to prevent the uncontrolled growth of the total size of registered objects, CASCADE imposes two limits: (1) on the maximal number of objects registered at each DCS, and (2) on the maximal size of each registered object. If the total number of registered objects reaches the limit, new register requests are rejected. In addition, whenever the size of some registered object grows beyond the limit, it is unregistered and its hierarchy is destroyed.

If some previously evacuated object is required later by a client, it will be acquired again from the father node transparently for this client. In turn, if the father node also does not have a copy of the requested object, it will try to acquire it from its father node. This way the request ascends all the way up along the hierarchy until the DCS that has a copy of the object is reached. (In the worst case, the chain reaches the root of the hierarchy). The object copy then descends along the hierarchy back to the request originator.

When an object copy is evacuated from the cache, the DCS continues to keep a small record needed for information dissemination along the hierarchy. This record is finally removed only when (1) no client has issued a request for the object during a pre-defined timeout, and (2) this DCS becomes a leaf of the object hierarchy. When this occurs, the DCS undertakes the steps detailed in Section 6.1 in order to disconnect from the hierarchy.

In the current version of CASCADE, the decision about which objects to evacuate from the cache is made by the LRU algorithm driven by client requests to objects. That is, whenever a new or previously evacuated object $O$ is to be brought into the cache, currently cached objects are evacuated according to the LRU criterion until there is enough space to accommodate $O$.

We intend to study the applicability of other cache replacement algorithms (see [20] for a discussion of potential alternatives).

### 6.5 Support for Security

CASCADE offers support for securing cached object access and communication between clients and cached objects. This is done by encrypting and signing communication and deploying object access control. Which of these measures are undertaken is determined by means of per-object *security policy* that can be specified by the object creator when the object is registered with CASCADE.

We are currently working on DCS protection from malicious objects and malicious clients. [20] provides a more detailed description of these and other issues related to security in CASCADE.

## 7 Applications

In this section we describe three typical applications that can benefit from our caching service.

### 7.1 Yellow Pages Service

An example of a typical application that requires only Update Propagation and Monotonic Updates consistency guarantees is a yellow pages service. The object to be cached here is a catalogue of named information items. Clients can register their names along with the information they would like to publish about themselves. They can update information about themselves and they can also query information about some particular name or search the catalogue for some particular information.

Under the realistic assumption that no two clients will try to register under the same name, updates from different clients are not interrelated. While synchronous catalogue update provides better fairness, it will usually be permissible for two clients to temporarily see different views of the catalogue. After all, even the phone company does not deliver a new yellow pages book at the same time to all houses. In addition, if the DCS hierarchy reflects the geographical location of the servers, then, according to the principle of locality, if some client registers information at some DCS, it is more probable that queries about this information will be issued to the same DCS. Thus, additional consistency guarantees are not required for this type of application.

### 7.2 Tickets Reservation Service

In contrast to the Yellow Pages Service, this service requires the classical strong consistency [33]. The object to be cached here is a ticket reservations database. Clients can book a ticket and they can also query information about ticket availability. The service should not allow two clients to book the same ticket. This implies that all reservation requests should be totally ordered.

This service can deploy a distributed transaction for booking a group of tickets. In addition, this service can make use of a dirty query mechanism explained

below. Consider the following situation: Some client reserves a ticket at some DCS, and then another client attempts to reserve the same ticket at the same DCS before the previous update has been propagated. Then a dirty query can already show that this ticket is not available before the actual update propagation. Note that the probability that the second client contacts the same DCS is relatively high because of the above mentioned principle of locality.

Note also that our caching service can be used for both airline ticket reservations and theater reservations. Here, again, the fact that we cache active objects is instrumental as the rules for performing reservations in both cases are different. For example, airlines allow for over-booking, but over-booking might not be acceptable in theaters.

It should be noticed that queries in a ticket reservation service can benefit from *dirty copy consistency* [6]. Loosely speaking, a dirty copy consistency query returns a value resulted from all locally known updates, even if some of these updates were not ordered in the global sequence yet. Suppose, for example, that a number of clients connect to their regional DCS and request to book a seat in the same flight. Suppose also that as a result of all these booking requests the flight gets overbooked. Now another client connects to the same DCS in order to book a seat. It first wants to figure out the situation so it issues a dirty copy query request and immediately learns that the flight is overbooked and therefore, the chance for it to get a seat is small. The client can then go and try to book a ticket for another flight to the same destination.

### 7.3 Distributed Bulletin Board

In a distributed bulletin board, users can post events, or poll the board for recently posted events. Also, it is possible to define topics, in which case a user may look for postings in a specific category, or get a listing of all postings. An additional requirement from a bulletin board is to preserve causality [34] of event postings so that a follow-up posting is always seen after the original posting it is referring to. Thus, this application requires Update Propagation, Monotonic Writes, Writes Follow Reads and Monotonic Reads consistency guarantees.

## 8 CASCADE Performance

In order to assess CASCADE performance we conducted a simple test involving two DCSs: one running on a machine connected to the Hebrew University of Jerusalem CS department LAN and another running on a machine connected to the Haifa Technion CS department LAN. Both CS departments are connected to the Israeli Academic and Research network and separated by 7 hops. Both hosts are Intel architecture machines (Pentium II 300 Mhz with 128 MB RAM) running Windows NT 4.0 operating systems. The ORB used in testing environment was Visibroker 3.2 for Java.

Our goal was to evaluate the effect of using CASCADE on method invocation time of a CORBA object. To achieve this, we measured the method invocation

time twice: once for a standard CORBA remote invocation and another time for an invocation when the object was cached with CASCADE. The test was conducted with a 10 KB CORBA object whose interface consisted of both updates and queries. Method argument data types were of small size and therefore their marshaling/demarshaling had a little effect on overall invocation times. The object consistency policy included only the Update Propagation guarantee (see 6.2).

Tables 1 and 2 summarize our results (all the times are given in *ms*).

| Operation | Avg. time | Std. deviation |
|---|---|---|
| Processing in interceptors | $< 1$ | $< 1$ |
| Request processing by proxy | 0.83 | 2.87 |
| Proper invocation time | 2.4 | 6.59 |
| Full invocation time | 11.1 | 25 |

**Table 1.** Method invocation profiling on a DCS

| Operation | w/o CASCADE | | using CASCADE | |
|---|---|---|---|---|
| | Avg. time | Std. deviation | Avg. time | Std. deviation |
| Marshaling/demarshaling | 1.32 | 4.1 | 1.49 | 3.91 |
| Client interceptors | N/R | N/R | $< 1$ | $< 1$ |
| Request invocation | 93 | 8 | 46.78 | 15.7 |

**Table 2.** Method invocation profiling on a client

We measured the following times at a DCS:

**Processing in interceptors** - time spent on request processing by server interceptors. It was always close to 0 because of the weak consistency policy we chose. However, even when we measured this time for strong and group consistency policy, it was never more than few milliseconds.

**Proper invocation time** - time spent in the *invoke* function of the cached object. It was only few milliseconds for our lightweight object.

**Request processing by proxy** - time taken by an object proxy to process the request in order to satisfy consistency and other policy requirements.

**Full invocation time** - this is the sum of the above times and the time of the request processing by ORB. The latter includes the times of marshaling/demarshaling.

We measured the following times at a client application:

**Marshaling/demarshaling** - time spent in the stubs on marshaling the *in* parameters and demarshaling *out* parameters. It was few milliseconds because of the small size of parameter types.

**Client interceptors** - time spent on request processing by client interceptors. All explained about time spent in server interceptors applies here as well.

**Request invocation** - time taken by *request.invoke()*. It consists of the time spent in client interceptors, the full invocation time at the DCS, additional request processing by ORB, and the network latency.

The results shown in the tables clearly indicate that the network latency is the most influential factor in the overall invocation time. Since use of CASCADE significantly reduces the network latency, it leads to a 2 times speedup for our settings. Furthermore, taking into consideration that a ping from Israel to the US or to Europe takes about 600ms, we expect the magnitude order of a speedup to be 20-30 times for far objects.

In future we intend to conduct tests for a larger number of DCSs and cached objects, for objects of different sizes and for more data transmitted during request invocation. We expect the speedup for bigger requests to be even greater because the time of their transmission over the network will overweigh the difference in marshaling time.

## 9 Lessons Learned from CORBA Experience

**Object by Value** CORBA has the advantage over many commercial RPC systems in that it allows object references to be manipulated as regular data type values in a straightforward manner. In particular, an object reference can be passed as an argument of a method invocation on another object. However, other parameter passing semantics are not supported by CORBA. Specifically in CASCADE, there is a need to pass objects by value when the object is cached at some DCS. Currently, this is implemented with the aid of Java serialization. The shortcomings of this approach are that (a) we are limited in the choice of programming language, and (b) the object state is passed as a sequence of bytes. This does not stay in line with object-oriented approach.

While CORBA 2.3 introduced the object-by-value standard to tackle this problem, this standard is not mature enough and its implementations are rare. Furthermore, CORBA lacks the ability to forward an incoming request to another server, i.e., to pass it by value. This ability is important for CASCADE that executes the same update request at multiple locations and totally orders update requests in order to achieve consistency. Below we explain how we circumvent the lack of this ability with the aid of interceptors.

**Interceptors** Use of interceptors is vital for CASCADE implementation. Interceptors allow CASCADE to perform a set of operations transparently to the client applications: to encrypt and sign requests and replies, to maintain consistency and to pass implicit per-request policy parameters.

Unfortunately, the interceptor standard of CORBA is to undergo a major revision, and there are no implementations of the current standard. Therefore, we used proprietary Visibroker 3.2 interceptors which conceptually correspond to the standard of message-level interceptors. Per-request policies

and consistency parameters were passed through the service context field of a standard GIOP request message header. Unfortunately, Visibroker (and even the CORBA standard) defines no means for encapsulating typed data into service context, so we had to implement marshaling/demarshaling by ourselves.

An additional use of interceptors in CASCADE is for forwarding an incoming request to another DCS. This was implemented in the following way: when DCS $A$ receives a request to be propagated to its parent DCS $B$, $A$ calls an internal DCS interface method on $B$ and passes the GIOP request message body as a parameter (see [20] for a description of the internal DCS interface). If $B$ is to apply this request, it creates an instance of CASCADE class that implements *CORBA::ServerRequest* and that is initialized with a GIOP request message body.

However, in some ORBs, this solution could lead to little endian - big endian incompatibilities: If $A$ and $B$ have the same endian order, the standard CORBA input/output streams are used for marshaling/demarshaling: The request message body is written to the *CORBA::portable::OutputStream* by calling *write_octet_array*, *CORBA::portable::InputStream* is created out of the OutputStream, and the typed data is read from this InputStream. However, if $A$ and $B$ have different endian order, we had no choice but to use input and output streams implemented in CASCADE.

An alternative way of redirecting a request would be to use request-level interceptors, to demarshal request parameters into *any*s, and to call an appropriate method on $B$, passing the list of these *any*s. However, in this solution the types of *any*s would be also passed on the wire, while these types would be absolutely unnecessary.

## 10 Future Work

CASCADE is a rapidly evolving system that is under permanent development. As part of our future work we intend to evaluate applicability of several existing protocols for reliable multicast and dynamic resource discovery in the Internet for improved hierarchy construction and update propagation. Other research directions include evaluating cache replacement policies, working out the security model, adding fault-tolerance, conducting thorough performance tests etc.

Another interesting problem to be addressed is whether the hierarchical cache architecture employed by CASCADE is most adequate for wide area distributed settings. To answer this question we intend to examine other possibilities for replicating and caching CORBA objects. For example, an interesting option to examine would be a hybrid architecture that combines replication of CORBA objects at limited number of powerful servers with light-weight cache servers at the end-user sites.

Also, it would be interesting to identify features provided by various CASCADE components that are generic enough in order to be incorporated independently into the CORBA standard. For example, consistency policy implementa-

tions can be beneficial for a wide range of distributed applications that involve object replication for improved availability and fault-tolerance.

We are also working on development of various wide area CORBA based distributed applications that will utilize CASCADE for improved availability and service time predictability. Among the applications currently under development are a distributed news facility and a shared whiteboard. These applications will help us to assess the CASCADE impact on the service quality and to identify the features that should be added to CASCADE in order to better meet the application needs.

## References


1. FarGo home page. http://www.dsg.technion.ac.il/fargo/.
2. Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: limitations and potentials. In *Proceedings of the 4th International WWW Conference*, December 1995. Available at http://www.w3.org/pub/Conferences/WWW4/Papers/155/.
3. 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.
4. M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 93.
5. Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, December 1996. Available at http://cs-www.bu.edu/faculty/best/res/papers/pdis96.ps.
6. Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1995.
7. H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors. In *Proc. of the 24th ACM Symp. on the Theory Of Computing*, pages 679–690, May 1992. Revised version: Technical Report #767, Department of Computer Science, The Technion. Submitted for publication.
8. H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.
9. Henry Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.
10. H. Bal, F. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transaction on Software Engineering*, 18(3):190–205, March 1992.
11. H. J. Bekker. The Java Platform, A White Paper. JavaSoft, Sun Microsystems. Available at http://www.javasoft.com/docs/white.
12. H. J. Bekker. Survey on caching requirements and specifications for prototype. DESIRE European project deliverable D4.1. Available at http://www.ruu.nl/~henny/desire/survey.html.
13. J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the 2nd ACM Symp. on Principles and Practice of Parallel Processing*, pages 168–176, 1990.

14. 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.
15. K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Company and Prentice Hall, December 1996.
16. P. Butterworth, A. Otis, and J. Stein. The GemStone database management system. *Communications of the ACM*, 34(10), October 1991.
17. Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, December 1997. Available at http://www.cs.wisc.edu/ cao/papers/gd-size.ps.Z.
18. Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the web. In *IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.
19. J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Rel ease Consistency.* PhD thesis, Department of Computer Science, Rice University, September 1993.
20. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Cascade: Caching service for corba distributed objects. Technical report, Department of Computer Science, The Technion, October 1999. In preparation.
21. O. Deux et al. The story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.
22. R. Friedman. *Consistency Conditions for Distributed Shared Memories.* PhD thesis, Department of Computer Science, The Technion, 1994.
23. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.
24. Peter Graham and Yahong Sui. LOTEC: A Simple DSM Consistency Protocol for Nested Object transactions. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1999. To appear.
25. 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.
26. IONA. *Orbix Programming Guide.* IONA Technologies Ltd., November 1995.
27. Ayal Itzkovitz and Assaf Schuster. MultiView and Millipage — Fine-Grain Sharing in Page-Based DSMs. In *Proc. of the 3rd Symp. on Operating Systems Design and Implemen tation (OSDI'99)*, New Orleans, February 1999. To appear.
28. K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *15th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 213–228, December 1995.
29. P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, December 1994.
30. A. M. Kermarrec, I. Kuz, M. van Steen, and A. S. Tanenbaum. A Framework for Consistent, Replicated Web Objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998.
31. R. Kordale and M. Ahamad. A scalable technique for implementing multiple consistency levels for distributed objects. In *16th International Conference on Distributed Computing*, 1996.

32. R. Kordale, M. Ahamad, and M. Devarakonda. Object caching in a corba compliant system. *USENIX Computing Systems Journal*, 9(4), 1996.
33. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.
34. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.
35. 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.
36. Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta reference manual*. Programming Methodology Group Memo 88, MIT Lab. for Computer Science, feb 1994. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/.
37. P. Martin, V. Callaghan, and A. Clark. High Performance Distributed Objects using Caching Proxies for Large Scale Applications. In *Proceedings of the IEEE International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, September 1999.
38. Ingrid Melve. Web caching architecture. DESIRE European project. Available at http://www.uninett.no/prosjekt/desire/arneberg/.
39. Ingrid Melve, Lars Slettjord, Henny Bekker, and Ton Verschuren. Building a Web caching system - architectural considerations. In *Proceedings of the 1997 NLANR Web Cache Workshop*, June 1997. Available at http://ircache.nlanr.net/Cache/Workshop97/Papers/Bekker/bekker.ps.
40. P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, July 1999.
41. M. N. Nelson, G. Hamilton, and Y. A. Khalidi. A framework for caching in an object-oriented system. SMLI TR 93-19, Sun Microsystems Laboratories, Inc., October 1993.
42. Thomas Nolle. To cache or not to cache. http://www.nwfusion.com/columnists/1109nolle.html (registration required).
43. ObjectSpace. Voyager home page. http://www.objectspace.com.
44. Katia Obraczka, Peter Danzig, Solos Arthachinda, and Muhammad Yousuf. Scalable, highly available Web caching. Technical Report 97-662, USC Computer Science Department, 1997. Available at ftp://usc.edu/pub/csinfo/techreports/papers/97-662.ps.Z.
45. OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, 1995.
46. OMG. *CORBA services: Common Object Services Specification*. OMG, 1995.
47. Guillaume Pierre and Mesaac Makpangou. Saperlipopette!: a distributed Web caching systems evaluation tool. In *Proceedings of the 1998 Middleware conference*, September 1998.
Available at http://www-sor.inria.fr/publi/SDWCSET_middleware98.html.
48. 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.
49. M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurency*, 7(1):70–78, January-March 1999.

50. R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1999. Also: Technical Report CS #0964, Department of Computer Science, The Technion.
51. Philip S. Yu and Edward A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. In *Proceedings of the 7th International WWW Conference*, April 1998. Available at http://www7.conf.au/programme/fullpapers/1829/com1829.htm.
52. Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive web caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, June 1997. Available at http://ircache.nlanr.net/Cache/Workshop97/Papers/Floyd/floyd.ps.
53. R. N. Zucker and J.-L. Baer. A performance study of memory consistency models. In *Proc. of the 19th International Symposium on Computer Architecture*, pages 2–12, May 1992.