*Object Caching in a CORBA Compliant System**

R. Kordale and M. Ahamad
Georgia Institute of Technology
M. Devarakonda
IBM T. J. Watson Research Center

ABSTRACT: Distributed object systems provide the key to building large scale applications that can execute on a range of platforms. The Common Object Request Broker Architecture (CORBA) specification from OMG attempts to address interoperability and heterogeneity issues that arise in such systems. Our goal is to investigate performance issues for distributed object systems. We claim that object caching is a must for improved performance and scalability in distributed object systems. However, this important technique and implementation issues related to it have not been widely studied in the context of distributed object systems and have not been addressed in CORBA specifications so far. In this paper, we discuss the design and implementation of *Flex*, a scalable and flexible distributed object caching system. Flex is built on top of Fresco, which uses the CORBA object model. Fresco runs on the UNIX operating system and our implementation of Flex exploits the features of object technology, Fresco, and UNIX. This system allows us to quantify the performance improvements for object invocations that are made possible by caching.

*This work was supported in part by NSF grant CDA-9501637 and ARPA contracts N00174-93-K-0105 and DABT-63-95-C-0125.

1. Introduction

Distributed object systems provide the key to building large scale distributed applications that run on a range of platforms. Examples of distributed object systems include various Common Object Request Broker Architecture (CORBA) [Object Management Group] compliant object systems. In distributed object systems, applications experience communication latency and overhead when invoking methods on remote objects. This performance penalty can be minimized using object caching. The benefits of caching are well known and have been demonstrated in distributed shared memory systems (DSM) and distributed file systems. However, this important technique and its implementation issues have not been widely studied in the context of distributed object systems and have not been addressed in CORBA specifications.

In this paper, we discuss the design issues in object caching and describe the implementation and measurements of Flex[1], a scalable and flexible object caching system. A goal of this paper is also to contribute towards the discussion about adding caching as a common object service to CORBA. We implemented Flex on top of Fresco [X Consortium 1994], which is a distributed object system that uses the CORBA object model. Fresco is readily available as a part of the X11-R6 distribution for UNIX systems. Fresco converts invocations on a remote object to remote procedure calls to the object server. Flex enhances Fresco by adding object caching. Measurements of our object caching implementation show the obvious advantage of caching repeatedly accessed objects.

Design issues in object caching fall into two categories: (1) Issues that are specific to object-oriented systems; (2) Issues that can be better addressed in object-oriented systems although they are not unique to objects. Issues in the first category identify important differences between object caching and caching of files and memory pages (as in distributed file systems [Nelson et al. 1987; IBM 1994] and distributed shared memory systems [Li & Hudak 1989; Kohli et al. 1995; Keleher et al. 1994]). These issues from the first category are listed below:

1. Our system is not related to the "fast scanner generator" program [31] which is also called Flex.

1. The programmer should be able to specify which objects can be cached because some objects may be only wrappers around some control logic or device drivers and such objects should not be cached.
2. An object may have references to other objects, and therefore the system requires flexibility in deciding which of the referenced objects, if any, should be cached.
3. Objects may have method level access control. While this is true even of files and memory pages, they support only two elementary types of accesses—namely, read and write. Performance considerations resulting from these differences have implications in the design and implementation of object caching.
4. When methods are invoked on an object, it is hard to detect the type of access (i.e. whether the state of the object will be changed or not), but the ability to detect the type of access is invaluable in reducing the cache consistency overhead.
5. Information about objects, such as the methods and superclass identification, is accessed more often than the objects themselves. Object caching, therefore, should be concerned about caching such information, which is typically stored outside the object itself.

While the design issues in the first category are specific to object-oriented systems, the problems of maintaining cache consistency are not specific to object-oriented systems. However, cache consistency issues can be addressed better by object technology. The performance and functionality of a caching based distributed object system depends heavily on the level of cache consistency it provides. Instead of providing a single consistency level to all applications, we take the approach of providing multiple consistency levels. Such an approach offers improved performance by allowing a consistency level that is customized for a given application. In Flex, we find that object technology makes it easy to provide multiple consistency levels and customization. Flex provides various consistency classes as a *consistency framework* and allows class implementors to subclass from a specific consistency class for customization.

The rest of the paper is organized as follows. Section 2 discusses building multiple consistency levels for cached objects. Section 3 describes the design issues that are specific to caching in distributed object systems and possible solutions in the context of Fresco. Section 4 provides an overview of Flex implementation, and Section 5 presents performance measurements. Section 6 describes our experiences with a CORBA-based object system running on UNIX. Section 7 concludes the paper.

2. Building Multiple Consistency Levels

Caching creates multiple copies of an object which introduces the problem of consistency among the copies. The consistency needs differ significantly across different application domains. For example, in a document sharing collaborative environment, changes to documents may not have to be made visible to all users immediately. On the other hand, in a multi-user interactive simulation, changes to the state of shared objects must be made visible to all users immediately and in the same order as the changes were made. Thus, we take the approach of allowing multiple consistency levels to coexist in an application. This can improve performance because of two reasons. Firstly, it allows the use of weaker consistency levels when applicable. Secondly, weaker consistency levels can be implemented more efficiently than stronger ones [Ladin et al. 1992]. In addition to improved performance, a system that provides multiple consistency levels for shared objects can facilitate increased functionality.

Strong consistency cannot be provided in systems where clients are temporarily disconnected, which can happen involuntarily or voluntarily in a mobile environment. This is because communication between nodes, where objects are cached, may be required before an access can be completed when strong consistency is needed. Thus, even applications requiring strong consistency can benefit from mechanisms that facilitate a graceful weakening of consistency requirements in order to be able to make progress. We allow applications to employ multiple levels of consistency, which can increase functionality and improve performance. In this section, we briefly explain the *mutual consistency* mechanism that is used to implement multiple consistency levels. The idea behind the mechanism is to keep cached object copies on a node (machine) mutually consistent.

2.1. Mutual Consistency

Informally, copies of two objects are mutually consistent if they could exist together in an application's view. Consider the following collaboration example. Scientist I writes a chapter on results (*results.old*). In this application, each chapter is written as an object. Scientist II reads the copy of results and writes a discussion chapter (*discussion.old*) on the results. Scientist I rewrites the chapter on results (*results.new*) and also a discussion chapter (*discussion.new*) on the new set of results. Now, suppose that a third scientist on a different node tries to read the two chapters. Our definition of mutual consistency specifies that the combination $<$ *results.old, discussion.new* $>$ is not mutually consistent. It is based on the observation that the other three combinations correspond to possible *global system*

states [Chandy & Lamport 1985] that could have occurred during the execution of the system. Note that if caching is not employed, combinations of the object copies accessed by applications in the above example correspond to the possible global system states. Of course, the specific two copies that the scientist reads will depend upon the consistency requirements. Thus, a mutually consistent view of a set of objects should correspond to a global system state that is meaningful with respect to the desired consistency level.

An underlying theme of the mutual consistency mechanism is to ensure that caching overhead on a node is proportional to the amount of caching activity on the node. For example, consider the case in which multiple read-only copies of a strongly consistent object exist when a client wishes to update its object copy. In conventional protocols, this would have induced communication with the other clients that have the read-only copies in order to either invalidate or update them. However, our protocol for strong consistency, which is based on the mutual consistency mechanism, does not invalidate or update all the read-only copies. Instead, consistency is maintained by using a novel technique that invalidates some of the locally cached object copies at the time a new object copy is added at the node. In this way, the cached copies are kept mutually consistent.

As we saw in the example above, two object copies are *mutually consistent* if the copies and more specifically, their corresponding values, coexisted in a consistent global system state. The more specific question in a test for mutual consistency is whether the "older" object copy is still "valid" in the global system state (view) corresponding to the "newer" object copy. For example, in the above application, *results.old* is not "valid" in the view that includes *discussion.new*, since it has been overwritten by *results.new*. Our implementation of the mutual consistency mechanism uses the notion of a *lifetime* for a value of an object. The lifetime of a certain value of object f is the duration defined by two logical times: the time when this value was created (the *creation time*) and the time until which the system has been able to establish that this value of the object has not been overwritten or become invalid (the *validation time*).

Note that the creation and validation times are assigned differently for different consistency levels. For example, suppose that *causal consistency* is desired for an object. Causal consistency only makes use of causal orderings to determine if a cached copy of an object is current. In this case, the creation time assigned to a copy of the object can be read from a logical clock that respects causal orderings between events in the system. The creation time would be assigned differently if in addition to causal consistency, *coherency* is also desired where coherency requires that all writes to any given object are totally ordered. We call this consistency level *causal coherency*. In this case, the creation time would reflect causally preceding events as before; additionally, the creation time also needs to

be greater than creation times assigned to all previously created copies of the object. Different consistency levels can be implemented using the mutual consistency mechanism by prescribing rules to assign *creation* and *validation* times to object copies.

The reader is referred to [Kordale & Ahamad 1996] for more details of the mutual consistency mechanism and the protocols used to ensure causal consistency, causal coherency and strong consistency (SC). SC is related to serializability and sequential consistency which are used in databases and shared memory systems respectively. The current implementation of Flex supports causal and strong consistency.

3. *Issues in Object Caching*

This section deals with the problems that are independent of consistency levels that need to be addressed in building an object caching system and the various possible solutions.

3.1. *Control in Caching*

We alluded to the issue of control in caching in Section 1. Here, we recapitulate the issues that lead to the necessity of control. Not all objects encapsulate only data. Some objects are really wrappers around control or hardware devices. Thus, it is not desirable to cache all kinds of objects. Having decided to cache an object, the issue of the most appropriate consistency level for the object remains. Objects have references to other objects. When an object is faulted in, it is not always clear if all referenced objects should also be faulted in. Control over the above aspects of caching can be exercised by one or more of the following entities among others.

1. Class implementor: This is based on the assumption that the class implementor knows best how the object should be implemented.
2. Client application: Usage patterns of objects may vary depending upon the application. In such cases, client application initiated caching and consistency maintenance can be very useful.
3. System: Traditional considerations such as load balancing can be best handled by the underlying system.

Clearly, no single method of control is suitable for all purposes. For example, an implementor of a class may choose to provide caching for objects of that class

based on the fact that most applications could benefit from caching such objects. While this benefits applications that need this default behavior, some applications may have different requirements. For example, they may not choose to cache these objects. Similarly, the system may decide not to cache instances of the class if the machine on which the client application is executing does not have enough resources. Thus, the above scenario illustrates that while the class implementor may be able to decide for the general case, client applications and/or the system may also need control in this regard. In our system, we have implemented class implementor initiated caching.

The class implementor can choose to inherit (directly or indirectly) from one of the classes in the consistency framework depending upon the need for caching and the desired consistency level. Also, one of the classes in the consistency framework (the `Cached` class) provides two methods (`readFromString` and `writeToString`) as part of its interface that provide control to the class implementor regarding the amount of object state that needs to be exchanged between processes. We describe the consistency framework in Section 4.1.

3.2. Cache Organization

Several issues arise related to the memory pool that constitutes the cache. Firstly, clients may or may not be allowed to directly access the cache. For example, in Spring [Nelson et al. 1993], clients do not directly access the cache; instead, they communicate with a local *proxy server* which in turn has direct access to the cache; in other words, objects accessed by clients are actually cached by the proxy server. Such a proxy server provides better security since all client invocations can be intercepted by the proxy server. However, every client access to cached objects incurs inter-process communication overhead with this approach.

We chose to allow client processes to have direct access to shared objects in the present implementation because of the high overhead of inter-address space communication in UNIX. In the future, we intend to build a user-level RPC (URPC) [Bershad et al. 1991] mechanism to reduce inter-process communication overhead at which time we will investigate the proxy server approach.

Once we decided to allow clients direct access to cached objects, the second issue that arises is whether the memory pool used for caching should be shared across all the clients on a node. We call this scheme *per-node caching*. The alternative is for each client to have its own pool, which we call *per-process caching*.

The per-node caching scheme has the advantage that object faulting and consistency related actions only need to be done for a single copy at a node even when several clients access the cached object at the node. However, the mutual

consistency based approach to implementing multiple consistency levels (described in Section 2), requires that whenever an object copy is cached in, all locally cached copies are checked to see if they are mutually consistent with the incoming object copy. Per-node caching can result in such checks being done amongst objects that belong to unrelated applications. One simple way to avoid such checks is to adopt per-process caching. In Section 3.4.1, we will show another reason why per-process caching is better than per-node caching. In our implementation, we employ per-process caching.

3.3. Cache Management

As we described before, clients on a node can freely cache in copies of objects to which they have access. To maintain consistency among the copies, mechanisms to invalidate or update object copies, and extract state from the object copies are required. Requests for these actions can come asynchronously with the execution of a client process. For example, when an object state is updated at a remote node, the client may receive a message asking it to invalidate its local copy of the object. Such requests can be handled in several ways. However, the initial version of Fresco, on which Flex was built, was layered on top of Sun RPC. Thus, the object transport interface was limited by the single-threaded transport runtime of Sun RPC. Here, we discuss possible solutions in this context.

Our approach is to create a cache manager process that shares the memory pool used for caching with the client and fields external (consistency) requests which may require invalidations or updates and sometimes extraction of the state from an object copy. In our implementation, two processes share memory by opening a common file and memory mapping the file to their respective address spaces. We considered two approaches.

In the first approach, one additional cacher process exists per node which pairwise shares memory with each of the clients and we call this process the *node cacher*. This approach has two problems. Firstly, since the node cacher shares memory with all clients on the node, it needs to open one file per client and may exceed the limit on the maximum number of file descriptors that can be opened by a process[2]. This problem can be solved by managing the open file descriptors independent of the number of client processes on the node. This can be done by closing files that are not currently required to be open and re-opening them on demand. While this performance overhead may be acceptable, this approach towards cache management does not scale well; the node cacher can become a bottleneck

2. Using shared memory segments to implement shared memory between two processes (instead of the mmap approach) has a similar problem.

since the transport is single threaded. Thus, we rejected this approach in favor of the approach described below.

In this approach, one additional cacher process exists per client process and we call such a process the *process cacher*. Each client process shares memory with its process cacher. While this approach scales well, this approach has two performance overheads that did not exist in the previous approach. Firstly, by doubling the number of processes on a node (one additional process cacher per client), general degradation in performance can be expected. Also, the cost of maintaining cache consistency can be higher in cases where more than one client process on a node, cache the same object. This can be illustrated by the following example. Let two client processes on a node (say A) have copies of some object (O). Suppose that a client on another node (say B) wants to access O and the consistency actions need communication with all the copies (e.g., their invalidations). In order to communicate with the copies on node A, two remote invocations from B to A are required instead of one remote invocation in the first approach.

3.4. Object Faulting and Access Detection

A method invocation on an object can be executed locally if the object's state currently resides in the cache. Thus, to execute a method invocation, it is necessary to determine if the object state is in the cache. If the object is non-resident, then it must be brought to the client and made available to the program in memory; this is called *object faulting* [Hosking & Moss 1993]. Also, concurrent sharing of cached objects leads to the issue of consistency among the copies. Protocols used for maintaining consistency of shared objects may require the system to *detect* updates to object state.

Many present day operating systems allow user-level programs to exploit virtual memory (VM) mechanisms (e.g., `mmap` and `mprotect` calls) to manipulate page protections. These mechanisms can be used for object faulting. VM mechanisms to detect access violations, coupled with user defined handlers, can be used to implement both object faulting and to detect accesses to shared objects.

Several software schemes also exist to facilitate object faulting and access detection that include tagging to distinguish between references to resident and non-resident objects [Hosking & Moss 1993]. Object oriented programming languages can exploit the indirection [Edelson 1992] implicit in the method invocation mechanism to fold residency checks into the overhead of method invocation. In IBM's SOM, one could use the BeforeAfter metaclass mechanism [Forman et al. 1994], which allows a *before* method and an *after* method to be called before and after (respectively) every method of a class. This way, the metaclass can gain control before and after an object invocation to handle details of object faulting

and access detection. In [Zekauskas et al. 1994], the authors present a method for write detection in a DSM system that relies on the compiler and runtime system.

The VM based scheme induces page faults that have considerable overhead. Also, large page sizes can create problems with false sharing [Bennett et al. 1990]. However, using VM mechanisms induces caching overhead only on object faults and some object accesses. Software schemes, on the other hand, require extra instructions on every fetch and/or store. In [Hosking & Moss 1993; Zekauskas et al. 1994], the authors argue that software based schemes out-perform VM based schemes. However, these arguments are made either in the context of an entry-consistent DSM system [Zekauskas et al. 1994] or in the context of implementing persistent stores, and garbage collection [Hosking & Moss 1993]. On the other hand, a previous study [Carey et al. 1994] on the issue of granularity choices for data transfer in client-server object-oriented data base management systems (OODBMS) favored page servers (mainly due to the consequent advantages of object clustering). Thus, previous studies, which dealt mainly with performance based comparisons, do not indicate a clear choice. In the next section, however, we will present an argument in favor of software based implementations of object faulting.

3.4.1. Advantages of Software Based Object Faulting

Our argument favoring software based object faulting falls into two categories—performance and functionality. Objects in CORBA based distributed applications have sophisticated method level access and it is customary to allow clients to have access only to specific methods of an object. While this is true even of files and memory pages, the important methods that they support are read and write. We will show below how this difference adversely affects performance of VM based schemes for object faulting. We will then show situations in which software based implementation of object faulting elegantly provides more functionality than a VM based scheme.

3.4.1.1 Performance

In DSM and file systems, per-method access control corresponds to clients having either read-only or read-write access to cached objects. Consequently, VM based faulting schemes are sufficient to efficiently detect access to objects. This is not true, however, in object-oriented systems. This is due to the mismatch in the protection provided by a VM based object faulting mechanism and the protection required by the application. We will illustrate this with an example.

Consider an object O which is an instance of class C. Suppose C supports methods m_1 and m_2 among other methods. Suppose that both these methods up-

date the state of the object. Suppose client A, which is allowed access to only m_1, caches O. If VM based mechanisms are used for object faulting, O should be cached in read-write mode. In this case, however, A can freely invoke method m_2 on O also. One way to prevent such an illegal access is to cache O in read-only mode. This way, each time A invokes either method, the fault handler can intercept and disallow illegal access. Clearly, this approach is inefficient since every invocation results in a page trap. The resulting overhead is much larger than the per-invocation overhead experienced in software based object faulting.

Software based implementations of object faulting can handle this case efficiently. Every client (actually, Flex's library which is linked with client code) maintains the per-class meta-data that specifies what methods the client is allowed to invoke. This meta-data can be retrieved from the server when the client caches the first instance of the class. Since every method invocation is intercepted by the object caching system library, illegal accesses can be detected and disallowed.

3.4.1.2 Functionality

In this section, we will discuss three issues and show that software based implementations of object faulting provide more functionality than a VM based scheme in an elegant fashion.

- Access Type Detection: An issue that arises in object faulting is the difficulty in detecting the type of access—namely, read-only or read-write—while faulting on an object. When a simple memory object is faulted (as in DSM systems) using the VM based scheme, the type of access is readily known since operations are elementary ones (like read or write). However, objects are accessed using method invocations. When a method is invoked on an object, the type of access is not obvious. Faulting an object copy in read-only mode only to realize later during the course of the method invocation that a read-write copy was needed is inefficient. Software based implementations of object faulting can handle this case elegantly. For example, in our implementation, cacheable objects support a method (`setAccessType`) to specify the access type of a method. The constructor code in a user defined class can invoke this method to specify the access type of each of the methods in the user defined class. Similarly, a method (`getAccessType`) to ascertain the access type given the object and the method name is supported. On every invocation of a user defined class' method, the access type is ascertained accurately using the `getAccessType` method.

  Such methods cannot be used in VM based schemes because the handle to the object that is obtained as a result of a segmentation fault is a pointer

to an object of type `char`. Since the pointer is not appropriately typed, the above specified methods are not accessible.

- Object State Transfer: Object caching requires transferring of object state between nodes. In our implementation, each cacheable object supports two methods `readFromString` and `writeToString`. We will explain these methods in detail in the context of the consistency framework described in Section 4.1. For now, it suffices to know that the `writeToString` method stringifies the object's state and `readFromString` reconstitutes the object using the stringified state. In particular, when a client A faults on a cacheable object, it requests some other entity such as a server for the object state. The server returns the stringified object state, possibly after communicating with one or more clients and servers. At this point, A invokes the `readFromString` method of the object to reconstitute the object from the stringified object state.

  Software based implementations of object faulting integrate well with such state transfer mechanisms provided by the object caching system. At every method invocation, it is checked to see if the required object state is locally resident. If not, the client receives the object state and reconsistitutes the object as discussed. However, this cannot be handled as elegantly in systems that employ the VM based scheme due to the following reason.

  In a VM based scheme, control is transferred to the specified segmentation fault handler on an object fault. The consistency framework (see Section 4.1) contains the implementation for the segmentation fault handler which returns with the new state of the object. At this point, the local object copy needs to be created or updated using the received object state. However, the handle to the object that is obtained as a result of a segmentation fault is a pointer to an object of type `char`. Since the reference is not appropriately typed, the `readFromString` method cannot be invoked.

  One way to solve this problem is to require that each client process maintains a *function table* that contains the address of the `readFromString` method of every type of object for which the client has a reference. When a client creates an object reference, the constructor of the corresponding class invokes the constructor of the `Causal` class with one of the arguments being the address of the `readFromString` method. This address is stored in the function table. Later, when object state is received as a result of an object fault, the stored address of the corresponding `readFromString` method can be used to update the local object copy. Clearly, this solution departs from object technology. This can be avoided if software based mechanisms were used to implement object faulting.

Interface Definition of MyClass:

```
interface MyClass {
  void set(in string strPtr);
  string get();
};
```

C++ mapping of MyClass interface:

```
class MyClass {
public:
  //Constructors and other required methods

  virtual void set(const char* strPtr);
  virtual char* get();
};
```

Implementation of MyClass class:

```
class MyClassImpl: public MyClass {
public:
  //Constructors and other required methods

  void set(const char* strPtr)
     { myPtr = new char(strlen[strPtr] + 1);
       strcpy(myPtr, strPtr);}

  char* get() { return(myPtr);}

protected:
  char *myPtr;
};
```

Figure 1. Definition of user-defined class.

- Per-node Caching and Legacy Applications: Another issue in which hardware based implementations encounter problems is while enabling caching in legacy applications. This problem arises if the object caching system employs per-node caching (see Section 3.2). Consider Figures 1 and 2. The interface definition for `MyClass`, its C++ mapping, and the definition of the corresponding implementation class `MyClassImpl` are given. We will focus here on two client applications A and B. Relevant portions of code for clients A and B are shown in Figure 2. Suppose that client applications A and B originally accessed instances of class `MyClassImpl` using remote invocations (i.e. function shipping). In this case, the

Client A code:

```
main() {
  MyClass *obj;
  char *someStr;
  ...
  obj = new MyClassImpl(...);
  obj → set(someStr);
  //someStr has been allocated
  //and assigned
  ...
}
```

Client B code:

```
main() {
  MyClass *obj;
  char *someStr;
  ...
  obj = new MyClassImpl(...);
  someStr = obj → get();
  ...
}
```

Figure 2. Per-node Caching and Legacy Applications.

example in Figure 2 would run as expected. A creates `obj` and sets `myPtr`. B then accesses the same object and gets a copy of `obj`'s `myPtr`.

Now, suppose that caching is enabled for instances of class `MyClassImpl`. One way to do this is to define a new class `myNewClass` that inherits from `MyClassImpl` and another class that enables caching (such as the `Cached` class shown in Figure 3 in the next section)[3]. The client application then creates instances of `myNewClass`. Now, A creates `obj` and caches it. However, when it invokes the `set` method on `obj`, it assigns `myPtr` in `obj` to an area of memory that is in A's private address space. When B executes, we assume that it gets the handle to the object (`obj`) created by A. However,

3. A similar argument can be made if one were to use a scheme that employs runtime inheritance [Mohindra et al. 1995] instead.

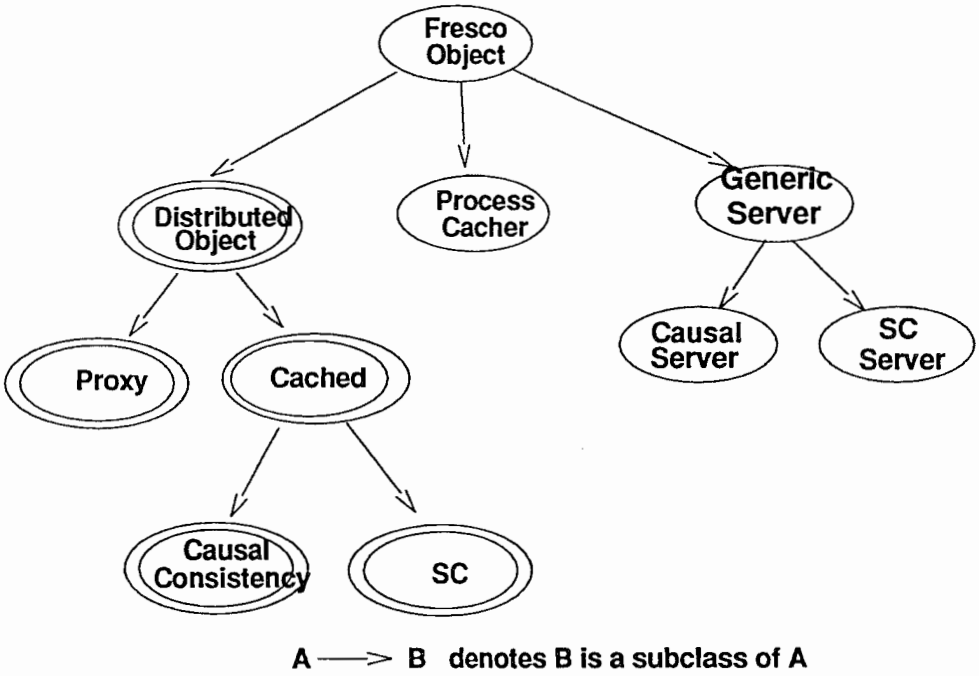

Figure 3. Caching Framework.

its attempt to dereference `someStr` results in a segmentation violation because `someStr` points to a memory location in A's private address.

An obvious solution to this problem is to change the implementation of `MyClassImpl` such that the `set` method makes `myPtr` point to some area in the shared pool of memory that constitutes the cache. One way to do this is to redefine `myPtr` to be a pointer of type `newString` such that the allocator for this type allocates memory from a shared pool instead of the process' private address space. However, changing (and recompiling) legacy code may not always be a feasible option. However, this problem is an artifact of using hardware based mechanisms for object faulting. Solutions exist if software based mechanisms were used for object faulting. One way to get around this problem is to route method invocations on shared objects through stub routines that do the necessary marshalling/unmarshalling of arguments.

As we mentioned in Section 3.2, our system employs per-process caching. The scenario in Figure 2 does not cause problems in per-process caching. When B tries to access the shared object (`obj`), it cannot get a pointer to A's copy of `obj`. Instead, the entire state of the object is obtained

and stored in B's per-process cache, thus precluding the problem of accessing memory belonging to another process' address space.

3.5. Object Implementation

So far, we have focused on the issue of managing object state when objects are cached. A related issue that is important in object caching is that of installing, finding, and using code that implements an object's methods; we will refer to the code as the object's implementation. A number of choices exist for dealing with each of the above activities. For example in Emerald [Jul et al. 1988], object implementations are stored in *concrete type* objects. When a kernel receives an object's state, it determines whether a copy of the concrete type object implementing the received object already exists locally; if it does not, the kernel obtains a copy from another node using a location algorithm. In [Steensgaard & Jul 1995], code mobility in Emerald is achieved on heterogeneous computers at the native code level; migrated code runs at native code speed before and after migration.

In the Common ORB Architecture document (CORBA 2.0) [Object Management Group .28], on the other hand, it is stated that the object implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery. Object adapters in CORBA are responsible for the interpretation of object references and mapping object references to the corresponding object implementations. These functions are performed with the cooperation of the ORB Core and the implementation skeletons. When a client invokes a method on an object, the ORB Core, object adapter, and the implementation skeleton at the server end arrange that a call is made to the appropriate method of the implementation. A similar mechanism can be used at the client end to find and install implementations of cached objects, possibly with the aid of a library object adapter linked with clients that cache objects. This may necessitate that object implementations be installed at each of the sites of use.

This issue concerning object implementation is particularly in the context of CORBA compliant systems, since an important goal of CORBA is to provide interoperability between applications on different machines in heterogeneous distributed environments and to seamlessly interconnect multiple object systems [Object Management Group .29]. In our implementation, however, we make the assumption that the implementation for the cached object is available; in fact, the application code is either compiled or dynamically linked with the code that implements the shared objects.

4. Implementation

The universe of objects in Flex can be divided into a set of inherently private and shared objects. An object may contain any number of references to other objects. The shared objects can be either passive or active. Active objects are associated with a dedicated process usually known as an object server. Active objects are usually large grained and are not cached in our system. Passive objects do not have a process dedicated to them and can be cached.

The system architecture of Flex is defined by the *caching framework* shown in Figure 3, which consists of the important classes that enable caching. As shown in the figure, the framework consists of three subtrees. The left-most subtree shows the various styles of accessing distributed objects and this is the framework that is visible to the users. These classes are distinguished with double lines in the figure. Class implementors subclass from one of these classes directly or indirectly while invoking distributed objects. These classes constitute what we call the *consistency framework*. The remaining branches define the classes which are transparent to the application programmers. These define the code that is executed by the implementation of the object caching system. These classes define what we call the *implementation framework* and is described in Section 4.2.

4.1. Consistency Framework

At the root of the consistency framework is the *DistributedObject* class. Because an object reference is opaque, it is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. The *DistributedObject* class interface provides methods such as `stringify` and `objectify` to solve the above problem. Actually, the object request broker (ORB) interface in the CORBA specification provides more "heavyweight" methods such as the `object_to_string` and `string_to_object` to solve a similar problem; these methods in the ORB interface additionally help to convert an ORB dependent distributed object reference to an ORB independent object reference and vice-versa. We provide these methods in the *DistributedObject* class because the implementation of Fresco, that we use, does not support these methods of the ORB interface.

One of the subclasses of the *DistributedObject* class is the *Proxy* class. This reflects the style of access that is specified by CORBA and is generally available in existing CORBA compliant systems. Basically, when a client tries to access an object server, it receives a handle to a proxy object [Shapiro 1986]. Any invocation on the proxy object is translated into an invocation at the remote object

server. Thus, objects that are not intended to be cached (such as wrappers around hardware devices) are designed to be instances of classes that are descendants of the `Proxy` class.

Another subclass of the *DistributedObject* class is the `Cached` class. One of the enabling mechanisms we use to implement caching is to stringify object state, pass it over the network and reconstitute the object at the other end. The `writeToString` and the `readFromString` methods accomplish this. In `writeToString`, the relevant object state is linearized and written out as a string. This string can now be sent out on the wire and the `readFromString` method on the other end can read from the string and update the object's state. Note that this is not the same as the methods we discussed in the context of the *DistributedObject* class. In that class, only the means to access the remote object can be stringified; in contrast, the methods described here stringify the object state and make it possible to transfer object state between processes.

An interface similar to that of the `Cached` class is part of CORBA's `Externalization` service proposal. While similar mechanisms can be used to implement both interfaces, the two interfaces clearly have different functionality and the difference is in the amount of relevant object state that is stringified. The relevant object state stringified in the Externalization service corresponds to the state that is required to activate an object from secondary storage. On the contrary, the relevant object state in the Cached interface corresponds to just the amount of information that needs to be transferred between object copies to maintain consistency.

Our intention is for the class implementors (who wish that instances be cached and thus inherit from one of the descendants of the `Cached` class) to have the option to override the `readFromString` and the `writeToString` methods defined by default in the `Cached` class. The advantage of such a provision is that the class implementor knows a great deal about the class and therefore can provide optimizations. For example, if instances of a cacheable object have a large state and it is known that the relevant state information that needs to be passed among object copies is only a small part of the entire state (e.g., only the modifiable parts of the state), then it is more efficient to stringify only that part of the state instead of the entire object state. Also, as we pointed out in Section 3, objects have references to other objects. These two methods allow the class implementor to decide which of the referenced objects need to be faulted in and which of them will just be sent as object references. Of course, if the class implementor does not override these methods, a preprocessor can generate default methods for state transfer automatically.

The subclasses of the *Cached* class implement the specifics related to providing different consistency guarantees. The caching framework shown in Figure 3

shows only two subclasses of the *Cached* class that provide strong and causal consistency guarantees. Ultimately, we intend to provide a richer suite of consistency classes somewhat similar in flavor to the various session guarantees provided in Bayou [Terry et al. 1994]. The discussion on the relevance of the different consistency guarantees and their details can be found in [Kordale & Ahamad 1996].

4.2. Implementation Framework

Clients on a node can freely cache in copies of objects to which they have access. To maintain consistency among the copies, mechanisms to invalidate or update object copies and extract state from the object copies are required. Requests for these actions can come asynchronously with the execution of a client process. For example, when an object state is updated at a remote node, the client may receive a message asking it to invalidate its local copy of the object. In Section 3.3, we discussed the need for a cacher process per client process which we called the process cacher.

The process cacher class implements the code that is executed by process cachers. A process cacher handles the type-independent aspects of caching that include actions such as maintaining a table of objects that are cached and their corresponding locations in the cache, invalidating the object copy, and invoking methods on the object reference to extract and restore state. The type-specific aspects of object access are implemented partially in the consistency framework and by the specific class that the object is an instance of.

For example, if an object fault is experienced while trying to access a causally consistent object, the fault handler that is run is part of the `Causal` class which comes from the consistency framework. Also, the state of the object is extracted and restored using the default `writeToString` and the `readFromString` methods of the *Cached* class or the overridden definitions of these methods defined in a class outside the consistency framework.

While the process cacher implements functionality required at the client end, the *Generic Server*, the *Causal Server* and the *Strong Server* classes implement functionality at the server end. As the name suggests, the Generic Server class implements the generic and type-independent functionality such as creating objects, maintaining the logical clock, assigning unique object identifiers (in the context of the server), etc.

The type-specific functionality at the server end is implemented in the Causal and Strong Server classes. We will illustrate this by the following example scenario. Consider that a client requests a server for an object copy. Depending upon the consistency requirements of the object, different actions need to be taken. For example, information about just the creation and validation timestamps of the

copies of the requested object suffices in the case of a causally consistent object. However, in the case of a strongly consistent object, more information such as the access types (read/write) of the other copies of the object and the access type of the requesting client is required.

In Flex, client processes can create objects locally and at servers. When clients create cacheable objects locally however, the associated process cacher needs to implement the functionality of all the four classes in the implementation framework. This is because, other clients may be interested in accessing these objects and thus the associated process cacher also needs to implement server functionality. Thus, the active entities in Flex are pure clients which are client processes that create objects only at remote servers, pure servers, and *hybrid clients* which create cacheable objects both locally and at remote servers.

4.3. System API

Object caching does result in some changes that need to be exposed to the application level. These changes affect the way objects are programmed, created and used. First, we allow class implementors to specify the desired level of consistency by having the user defined class explicitly inherit (directly or indirectly) from the appropriate class in the consistency framework shown as part of the caching framework in Figure 3.

The second aspect of the API that affects the application program is the following. Since memory pointers do not make sense across address spaces, there needs to be an address-space independent way of accessing a shared object. We call this an *object-id* and it consists of the name of the server on which the object was created and an identifier that uniquely identifies the object in the context of the server. For example, if the object-id of an object is < *sname, nid* >, then this object was created at a server named *sname* and *nid* uniquely identifies the object within the server process. A client creates a distributed object on a particular server by specifying the server name and making the second component zero. When the object is created, its server-unique id is assigned by Flex. Thus, the constructor for the object's class takes an additional argument—the object-id. For example, a client obtains a reference to an existing object by executing `new className (oid, <arguments>)` instead of `new className (<arguments>)`, where `oid` is the object-id of the shared object that the client wishes to access.

Null Sun RPC	1.22
Null Proxy Invocation	1.85

Table 1. Null RPC and Proxy Invocation Timings (In Milliseconds).

Creating a cacheable object	1.61
Caching an object	8.3
Method invocation on a cached object	0.48
Validating a cached object	5.2

Table 2. Times (in Milliseconds) for Operations on Cached Objects.

5. *Performance*

This section presents measurements of execution times of simple invocations on the basic RPC system and also on top of Flex. The experiments were done on a 167MHz Ultra 1 and a 85MHz SPARCstation 5 running Solaris 2.5. The machines reside on two different 10Mbps ethernet based subnets connected by a router. The execution times for a single null RPC and a null invocation on a proxy are shown in Table 1. The set up in the Sun RPC case is straightforward. A server is started on a node and a client is started on a different node. The client then makes null RPC calls. The set up in the proxy case is also very similar. An object server[4] is started on a node. Clients obtain a reference to the object which is really a reference to the object server. The client then makes a "simple" method invocation on the server. The simple method neither takes any argument nor returns any result and nothing is done in the method's body. Unlike in the Sun RPC case, the simple method is invoked on the object reference which adds a small overhead.

Table 2 shows the timings for the various actions related to caching causally consistent objects. In this case, two hybrid clients executing on the two machines create objects and access each other's objects. Clearly, invocations on a cached object are very fast—an invocation on a cached object takes 0.48 milliseconds

4. An object server provides an address space for the distributed object.

whereas the cost of a "simple" proxy invocation is 1.85 milliseconds. This is to be expected and we now proceed to examine the overheads of caching.

The cost to create a cacheable object[5] is 1.61 milliseconds. This time is dominated by the time for an inter-process communication (IPC) between the client and the corresponding process cacher. This permits the process cacher to create meta-data on the object copy that the process cacher can use while servicing requests for the object. Validating an object copy takes 5.2 milliseconds. Validating an object copy includes acquiring the object's new state from a valid copy elsewhere in the system. For example, in the experiments conducted, when a client accesses an invalid copy and determines that validation is necessary, it communicates with the server [6] on the remote node on which the object was created. When a valid copy is added to the cache, cached object copies that are found to be mutually inconsistent with the incoming copy are invalidated and the local clock is updated.

Table 2 shows that caching an object copy takes 8.3 milliseconds. The actions required for caching an object copy are similar to the ones required for validation except that the client also informs the process cacher about it. Thus, the difference in the two times is because of the additional IPC and the additional context switches required to inform the process cacher. Once again, this is because the process cacher needs to maintain meta-data regarding the object that has been cached in order to service later requests for the object. Note that invalidations are local and take an insignificant amount of time.

6. Discussion

We saw in the previous section that caching considerably reduces latency in accessing shared distributed objects when applications exhibit a reasonable amount of locality of reference. One of the goals of our system is scalability. There are many aspects of scalability. We believe that by allowing applications to use weaker consistency models whenever applicable, the overall scalability of the system is improved. Towards this end, we provide flexible notions of state consistency in our system. Another aspect of scalability that can be of concern is the size of the logical timestamps that we associate with each object copy. In our implementation, logical timestamps are implemented using vector timestamps. Simple implementations of vector timestamps contain a component per process

5. A cacheable object is an instance of a class that is a descendant of the `Cached` class.
6. Hybrid clients act as both servers and clients.

that is allowed to update objects. Various efficient implementations that reduce the space overhead considerably are outlined in [Raynal & Singhal; Torres-Rojas].

6.1. Caching as an Object Service

The Object Management Architecture guide [Object Management Group .29] includes a reference model that identifies and characterizes the components, interfaces, and protocols that compose the OMA. The reference model consists of four major parts: the object request broker (ORB), object services, common facilities, and application objects. The ORB enables objects to transparently make and receive requests and submissions in a distributed environment. Object Services is a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Common Facilities is a collection of services that provide general purpose capabilities useful in many applications while application objects are objects specific to particular end-user applications.

As we pointed out earlier, it is neither desirable nor possible to cache all objects. Furthermore, different consistency guarantees may be required of cached copies. Thus, caching should be provided as a Common Object Service and not as part of the ORB. The consistency framework can serve as the interface of the object service itself. We discuss below the extensions to CORBA IDL and the CORBA object model that can be useful in implementing object caching.

6.1.1. CORBA IDL

The suggested extensions to IDL result from the need to detect the access type (read or write) of a user defined class's methods and the need to pass objects by value. We describe each of these below.

- Detecting access type: As we mentioned earlier in Section 3.4, detecting whether the access type is a read or write is important to service an object fault efficiently. For example, acquiring a read-only copy of an object on an object fault only to realize later that a read-write object copy was required is inefficient. As we mentioned before, detecting access type is harder in object caching as opposed to the case in caching flat objects such as files and memory pages in which case every operation on the object carries information about the access type. For example, the operations are either elementary read or write operations or operations such as opening a file that specifies the mode in which the file should be opened.

  In Section 3.4.1, we described how we specify access types of user defined class's methods. A more elegant way to solve this problem is to

specify the access type along with every method declaration in the IDL definition for the user-defined interface.

- Passing objects by value: CORBA compliant systems have the advantage over many commercial RPC systems in that they allow object references to be manipulated as first-class values in a straightforward manner. In particular, an object reference can be sent as an argument of an invocation on an object server. Thus, CORBA compliant systems provide for network-wide references, and support distributed object reference semantics. However, this does not always provide the required semantics. An application might want to pass an object by value just like pointer structures are passed by value.

  For example in Flex, many method invocations in the caching framework take a vector timestamp as one of the arguments. We have defined a VectorTimeStamp IDL interface and we will call the instances of this class as vector timestamp objects. Now, suppose that we want to pass the vector timestamp arguments as references to objects of type `VectorTimeStamp` in invocations on the object server. Firstly, we need to activate an object server to service the vector timestamp object sent as an argument. Moreover, every invocation on such an object reference (argument) would be an inter-address space invocation, which we clearly want to avoid. Ideally, we would like to have the choice to pass an object argument by "value" as in [Birrell et al. 1993; Janssen et al.; Mitchell et al. 1994; Jul et al. 1988]. The object structure should be linearized, sent over the wire, reconstructed at the other end and finally, we should be able to make an object invocation locally at that end.

6.1.2. CORBA Object Model

From our experience, we found that *multicast* is an important mechanism that is useful in consistency maintenance in distributed object systems. For example in Flex, each time an object copy is validated, the node cacher at the owner node can potentially contact node cachers on all nodes where clients contain copies of the object. This is a perfect candidate for exploiting multicast communication. Though a system level multicast may be available, it cannot be used because the CORBA object model does not support it. In [Landis & Maffeis 1995], the authors argue the need for a multicast interface in the context of object groups.

Clearly more issues remain to be explored, particularly in the realm of identifying interactions with other object services. Our intention in this paper is to provide a data point in a discussion of issues related to caching in CORBA.

7. Concluding Remarks

In this paper, we described the design and implementation of Flex, a scalable and flexible distributed object caching system, on top of a CORBA compliant system running on the UNIX operating system. An important feature of Flex is that it supports flexible notions of object state consistency thus improving system performance through the use of weaker consistency levels when applicable. We presented issues that arise in an object caching system. We also argued how some of these issues led us to believe that object caching should be provided as a Common Object Service and not as part of the ORB itself. Our implementation experience indicates that object caching can considerably reduce latency in accessing shared distributed objects when applications exhibit a reasonable amount of locality.

Acknowledgments

We would like to thank the referees of an earlier version of this paper (which appeared in the *Second USENIX Conf. on OO Technologies and Systems*) and Doug Schmidt for their many valuable comments. We would also like to thank Ajay Mohindra who helped clarify many implementation issues in DSOM (IBM's CORBA compliant system) to one of the authors during the author's summer internship at IBM T. J. Watson Research Center.

References

1. M. Ahamad, F. J. Torres-Rojas, R. Kordale, J. Singh, S. Smith, Detecting Mutual Consistency of Shared Objects, *Proc. of International Workshop on Mobile Systems and Applications*, December 1994.
2. M. Ahamad, G. Neiger, J. E. Burns, P. W. Hutto, and P. Kohli, Causal memory: Definitions, Implementations and Programming, *Dist. Computing*, Vol. 9, 1995.
3. J. K. Bennett, J. B. Carter, and W. Zwaenepoel, Adaptive software cache management for distributed shared memory architectures, *Proc. of the 17th ISCA*, 1990.
4. B. N. Bershad, T. E. Anderson, E. D. Lazowska, H. M. Levy, User-level interprocess communication for shared memory multiprocessors, *ACM Trans. on Comp. Sys.*, May 1991.
5. B. N. Bershad, M. J. Zekauskas, W. A. Sawdon, The Midway distributed shared memory system, *COMPCON*, Spring 1993.
6. A. Birrell, G. Nelson, S. Owicki, and E. Wobber, Network Objects, *ACM Symp. on Oper. Sys. Principles, 1993*.
7. M. J. Carey, M. J. Franklin, and M. Zaharioudakis, Fine-grained sharing in a page server OODBMS, *ACM SIGMOD*, 1994.
8. K. M. Chandy, and L. Lamport. Distributed snapshots: Determining global states of distributed systems, *ACM Trans. on Comp. Sys,* Feb 1985.
9. D. R. Edelson, They're Smart, but They're Not Pointers, *USENIX C++ Conference*, 1992.
10. I. R. Forman, S. Danforth, and H. Madduri, Composition of before/after metaclass in SOM, *Object Oriented Programming Systems, Languages, and Applications*, 1994.
11. C. Fidge, Timestamps in message-passing systems that preserve the partial ordering, *Australian Computer Science Conf.*, 1988.
12. A. L. Hosking and J. E. B. Moss, Protection traps and alternatives for memory management of an object oriented language, *ACM Symp. on Oper. Sys. Principles 1993*.
13. J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham, Scale and performance in a distributed files system. *ACM Trans. on Comp. Sys.*, Vol. 6 (Feb. 1988).
14. IBM, 11400 Burnet Road, Austin, TX, *SOMObjects Developer Toolkit Users Guide,* Oct. 1994.
15. B. Janssen, M. Spreitzer, and D. Severson, Inter-Language Unification, Xerox PARC, URL: `http://www.parc.xerox.com/Projects.html`
16. E. Jul, H. Levy, N. Hutchinson, and A. Black, Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, Feb 1988.
17. P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems, *Winter USENIX Conf.*, 1994.

18. P. Kohli, M. Ahamad, and K. Schwan, Indigo: User Level Support for Building Distributed Shared Abstractions, *Fourth Symp. on High Performance Dist. Computing*, August 1995.

19. R. Kordale and M. Ahamad, A scalable technique for implementing multiple consistency levels for distributed objects, *Proc. of the Intl. Conf. on Distributed Computing Systems (ICDCS)*, May 1996.

20. L. Lamport, Time, clocks and the ordering of events, *Comm. of the ACM*, July 1978.

21. K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. on Comp. Sys.*, Nov 1989.

22. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, Providing high availability using lazy replication, *ACM Transactions on Computer Systems*, November 1992.

23. S. Landis and S. Maffeis, Building reliable distributed systems with CORBA, *Proc. of USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1995.

24. J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. Powell, and S. R. Radia, An Overview of the Spring System, *Proceedings of COMPCON*, Fall 1994.

25. A. Mohindra, G. Copeland, and M. Devarakonda, Dynamic insertion of object services, *Proc. of USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1995.

26. M. N. Nelson, B. B. Welch, and J. K. Ousterhout, Caching in the Sprite network file system, *ACM Symposium on Operating Systems Principles*, 1987.

27. M. N. Nelson, G. Hamilton, and Y. A. Khalidi, Caching in an Object Oriented System, *Intl. Workshop on Object Orientation in Operating Systems (IWOOOS)*, 1993.

28. Object Management Group home page: `http://www.omg.org`.

29. Object Management Group Object Management Architecture Guide, Revision 1.0, OMG TC Document 90.9.1: `http://www.omg.org`.

30. G. T. Nicol, Flex—A fast scanner generator, information available at `http://www.ns.array.ca/ipe-1.2/doc/gnu/flex/flex_toc.html`.

31. M. Raynal and M. Singhal, Logical time: A way to capture causality in distributed systems.

32. M. Shapiro, Structure and Encapsulation in Distributed Systems: The Proxy Principle, *Intl. Conf. on Dist. Comp. Sys*, 1986.

33. B. Steensgaard and E. Jul, Object and native code thread mobility among heterogeneous computers, *ACM Symp. on Oper. Sys. Principles*, 1995.

34. D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, Session Guarantees for Weakly Consistent Replicated Data, *IEEE Symp. on Parallel and Dist. Information Sys.*, 1994.

35. F. J. Torres-Rojas, Efficient Time Representation in Distributed Systems, Masters Thesis, Georgia Inst. of Technology.

36. X Consortium, Fresco Sample Implementation Reference Manual, X Consortium Working Group Draft. Version 0.7. April 1994.
37. M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, Software Write Detection for a Distributed Shared Memory, *Oper. Sys. Design and Implementation*, 1994.