Object Caching in a CORBA Compliant System

R. Kordale and M. Ahamad, Georgia Tech
M. Devarakonda, IBM T.J. Watson Research

Object Caching in a CORBA Compliant System*

R. Kordale **M. Ahamad** **M. Devarakonda**

College of Computing,
Georgia Inst. of Technology
{kram, mustaq}@cc.gatech.edu

IBM T. J. Watson
Research Center
mdev@watson.ibm.com

Abstract

Distributed object systems provide the key to build 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, to our knowledge, the CORBA specification does not address object caching. The purpose of this paper is to serve as a data point in a discussion of issues related to caching in CORBA by describing the design and implementation of *Flex*, a scalable and flexible distributed object caching system. Flex is built on top of Fresco that 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.

Keywords: Object caching, flexible sharing, open systems, CORBA, UNIX.

*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 build large scale distributed applications that can execute on a range of platforms. Examples of systems which hope to address interoperability and heterogeneity issues include the various CORBA (Common Object Request Broker Architecture) [28] compliant object systems. Existing implementations of these systems experience communication latency inherent in distributed systems as well as the software overheads associated with messages that are exchanged when a client node invokes methods of an object that is implemented by a remote server. These overheads can be reduced significantly if the client node is allowed to cache the object state locally. In this case, future invocations can be executed locally which could lead to better performance.

The benefits of caching are well known and have been demonstrated in systems that range from shared memory systems to distributed file systems. This important technique and the implementation issues related to it have not been addressed in CORBA specifications. A goal of this paper is to contribute towards the discussion about adding caching as a common object service to CORBA. We describe the design and implementation of

Flex[1], a scalable and flexible distributed object caching system. We wanted to build Flex on top of a general purpose and a generally available platform. We chose Fresco [12] which uses the CORBA object model (and was readily available from the X11-R6 distribution). Fresco runs on UNIX and provides an object wrapper around Sun RPC. All invocations on a distributed object reference are translated to remote procedure calls to the object server. In Fresco, object interfaces are specified using OMG's Interface Definition Language (IDL). Flex enhances Fresco by adding caching to it. In addition to the implementation of Flex, issues and solutions discussed in this paper have also been influenced by our previous prototyping of caching in IBM's System Object Model (SOM/DSOM) [15].

Caching has been studied extensively in the realm of file systems [30, 14] and software based implementations [22, 19, 18, 5] of distributed shared memories (DSM) in which files, memory pages, etc. are cached. We will refer to the entities that are cached in these systems as flat objects to distinguish them from the more sophisticated objects (which we will just refer to as objects) which is the focus of this paper. Though object caching has many similarities to caching flat objects, there are a number of important differences.

1. Not all objects encapsulate just data. Many objects are really wrappers around control or hardware devices. Thus, it is not possible or desirable to cache all objects. In particular, it leads to the issue of control over what objects get cached.

2. Objects have references to other objects. Thus, when an object created at a server is cached by a client, it is not always appropriate to cache all objects referenced by the object; some control is needed regarding which of the referenced objects, if any, should be cached. Issues such as this and the previous one leads us to provide object caching as a **Common Object Service** [28] rather than make it a part of the Object Request Broker (ORB).

3. Objects have 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 flat objects, they support only two elementary types of accesses - namely, read and write. Security considerations resulting from these differences have implications in the design and implementation of object caching.

4. When methods are invoked on objects, it is harder to detect the type (read or write) of access. This is because objects are accessed using method invocations instead of elementary operations such as read and write.

5. Information about objects is accessed more often than the objects themselves. Object caching, therefore, should also be concerned about caching associated information that is stored outside the object.

While the above design issues are specific to object-oriented systems, some caching issues that are not specific to object-oriented systems can be addressed better by object technology. For instance, caching creates multiple copies of an object which introduces the problem of consistency among the copies. The performance and functionality of a distributed object system could depend very much on the level of consistency it provides. Instead of providing a single consistency level to applications, we take the approach of providing multiple consistency guarantees. In Flex, we take advantage of object technology by allowing class implementors to subclass from a spe-

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

cific consistency class in the *consistency framework* (described in section 4.1).

The paper is organized as follows. We start with some details on the technique that we use to build multiple consistency levels in section 2. The issues involved in building an object caching system, possible solutions in the context of Fresco and UNIX that take advantage of object technology, and the ones we adopted are discussed in Section 3. Section 4 includes an overview of our implementation of Flex. We present preliminary performance results in Section 5 which demonstrate that caching could result in improved performance for applications that exhibit a reasonable amount of locality of access. Our experience with CORBA and UNIX are detailed in Section 6 and we conclude the paper in Section 7.

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 [23]. 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 the nodes, where objects are cached, may be required before an access can be completed when strong consistency is required. 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* technique that is used to implement multiple consistency levels. The idea behind the technique is to keep cached object copies on a node (machine) mutually consistent.

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* [8] that could have occured during the execution of the system. Note that if caching is not employed, only mutually consistent copies of the two objects can be accessed by applications. 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 technique 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 technique 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 since the idea is to keep the cached copies mutually consistent.

As we saw in the example above, two object copies are *mutual 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 corresponding to *discussion.new*, since it has been overwritten by *results.new*. Our implementation of the mutual consistency technique 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 is such that all causally preceding events in the system are reflected. 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 as *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 technique by prescribing rules to assign *creation* and *validation* times to object copies.

The reader is referred to [20] for more details on the mutual consistency technique 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 which need to be addressed in building an object caching system and the various possible solutions.

1. 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* [13]. 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 [13]. Object oriented programming languages can exploit the indirection [9] 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 [10] 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 that can be used to handle details of object faulting and access detection. In [37], 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 [3]. 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 [13, 37], the authors argue that software based schemes outperform VM based schemes. However, these arguments are made either in the context of an entry-consistent DSM system [37] or in the context of implementing persistent stores, and garbage collection [13]. On the other hand, a previous study [7] 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, there is no clear choice and we chose the VM based scheme.

Another 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), 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 can be inefficient. We take the approach of having the class implementor specify the access type associated with each method invocation. Obviously, such information will have to be placed outside the object in order to decide on the access mode in which to request a copy of the object during object faulting. Consequently, access to such information should be made very efficient possibly by caching relevant parts locally.

2. 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 behaviour, 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. Cache Organization: Several issues arise related to the memory pool which constitutes the cache. Firstly, clients may or may not be allowed to directly access the cache. For example, in Spring [27], 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. This design assumes more importance in object caching because it is customary to allow clients to have access to specific methods of an object instead of the entire object. 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 an user-level RPC (URPC) [4] like mechanism to reduce inter-process communication overhead at which time we will investigate the proxy server approach.

```
class myClass{
  long *ptrVar;
  void set(long *longVar) {ptrVar = longVar;}
  long * get() {return(ptrVar);}
}
```

```
main() {
  myClass *obj;
  ...
  obj = new(...);
  obj->set(new long(10));
  ...
}
        Client A
```

```
main() {
  myClass *obj;
  long *longVar;

  obj = new(...);
  longVar = obj->get();
  ...
}       Client B
```

Figure 1: Per-node caching and legacy applications

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, there are disadvantages associated with per-node caching. A notable one is the issue of enabling caching in legacy applications. Consider the class definition given for *myClass* in figure 1. Suppose that client applications A and B accessed *obj* (in figure 1) by doing remote invocations on a server and would now like to enable caching for objects of type *myClass*. One way to do it is to define a new class *myNewClass* that inherits from *myClass* and another class that enables caching (such as the *Cached* class shown in figure 2 in the next section). 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 *ptrVar* in *obj* to an area of memory that is in A's private address space. When B executes, it gets the right handle to *obj* but its attempt to dereference *longVar* results in a segmentation violation.

An obvious solution to this problem is to change the implementation of *myClass* such that the *set* method makes *ptrVar* point to some area in the shared pool of memory that constitutes the cache. One way to do this is to redefine *ptrVar* to be a pointer of type *newLong* and define the constructor of this type to allocate memory from shared memory instead of the process' private address space. However, changing (and recompiling) legacy code may not always be an acceptable option. This issue persists even in other approaches to enable caching such as a scheme that employs runtime inheritance [26].

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. Another approach would be to allow such pointers to private address spaces and handle segmentation faults as they occur. Both approaches introduce additional overhead during each method invocation. In our implementation, we employ *per-process* caching. The reason the scenario in figure 1 does not cause problems in per-process caching is briefly the following. 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.

4. 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, Fresco's object transport interface is limited by the interface provided by Sun RPC on which it is layered. Moreover, the transport runtime is not multi-threaded. Here, we discuss possible solutions in this context.

Our approach is to create a cache manager process which 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 *mmap*ing the file to their respective address spaces. We considered three approaches. Problems exist in each of the approaches and we discuss them below:

1. 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*. 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.

2. The second approach is to have one additional cacher process per client process and we call such processes *process cachers*. Each client process shares memory with its process cacher. This approach has two problems:

   (a) By doubling the number of processes on a node (one additional process-cacher per client), general degradation in performance can be expected.

   (b) Consider the case where multiple client processes on the same node (say A) have copies of some object. Suppose that a client on another node (say B) wants to access the object and the consistency actions need communication with all the copies (e.g., their invalidations). In order to communicate with the copies on node A, multiple remote invocations from B to A are required. This can be avoided if incoming requests to a node are channelled through some process (for example, the node cacher). This is the approach taken in next case.

3. This approach is the same as the previous one except that it additionally has a node cacher (which does not share memory with clients); all incoming requests are channelled through the node cacher which then contacts the individual process cachers. This approach has the following problems:

   (a) It has the same problem as in case (2a) above.

   (b) This approach incurs the overhead of an additional inter-process communication (IPC) between the node cacher and the process cachers.

The prototype presented in this paper adopts the last approach. However, based on our experiences, we would like to move

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

away from this approach in the next revision of the implementation of the caching system. In particular, we would like to adopt the second approach while moving the node cacher functionality to the object manager or a server.

5. Object Implementation: So far, we have focussed 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 [17], 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 that object already exists locally; if it does not, the kernel obtains a copy from another node using some location algorithm. In [34], 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 1.2) [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 adaptors in CORBA are responsible for the interpretation of object references and mapping object references to the corresponding object implementations amongst other duties. 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 adaptor, 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 use implementations of cached objects, possibly with the aid of a library object adaptor linked with clients that cache objects. This may necessitate that object implementations be installed at each of the sites of use.

This is an important issue 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 [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 2 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

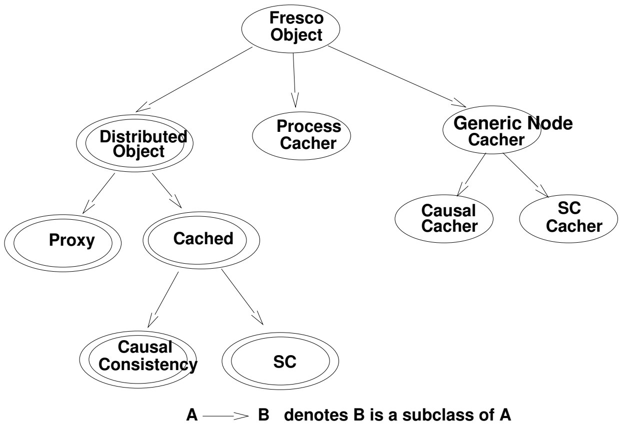

Figure 2: Caching Framework

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 various cacher processes discussed in the previous section. These classes define what we call the *implementation framework*.

4.1 Consistency Framework

At the root of the consistency framework is the *Distributed Object* 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 *Distributed Object* 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 "heavy-weight" 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 *Distributed Object* class because the implementation of Fresco, that we use, does not support these methods of the ORB interface.

One of the subclasses of the *Distributed Object* 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 [33]. 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 *Distributed Object* class is the *Cached* class. As we shall see later, 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 talked above in the context of the *Distributed Object* 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. A similar interface is part of CORBA's **Externalization** service proposal. 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 amongst 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, the default mechanism is for a preprocessor to generate these methods automatically.

The subclasses of the *Cached* class implement the specifics related to providing different consistency guarantees. The caching framework shown in figure 2, shows only two subclasses of the *Cached* class that provide strong and causal consistency guarantees. Ultimately, we intend to provide a bigger suite of consistency classes somewhat similar in flavor to the various session guarantees provided in Bayou[35]. The discussion on the relevance of the different consistency guarantees and their details can be found in [20].

4.2 Implementation Framework

The implementation framework can be explained by relating it to the cacher processes that are employed at a node to enable caching (see figure 3). In Section 3, we motivated node and process cachers. While we discussed one node cacher per node, we show three cachers

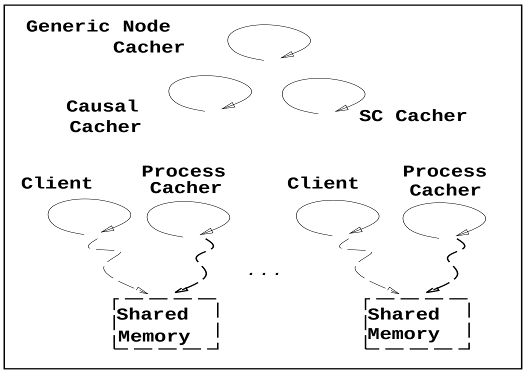

Figure 3: Cacher Processes at a node

at the node level in figure 3 - namely, the *generic node cacher*, the *causal cacher* and the *SC cacher*. We call the causal and the SC cachers as the *consistency cachers*. We chose to use separate cacher processes for different levels of consistency instead of having a single process (the node cacher) handle all the responsibilities of caching at the node level since this design allows new consistency levels to be added more easily. Though we describe a design that allows consistency level of objects to be decided statically, the goal of this prototype is to allow consistency levels of objects to be changed dynamically. This will be facilitated by the generic node cacher which is designed to implement the type-independent aspects of caching. Thus, the generic node cacher class implements functionality that is both type-independent and common to all the consistency cacher classes.

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, as mentioned in Section 4.1, 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.

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 2.

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 a host name, a process id, and a process-unique identifier. For example, if the object-id of an object is $< hname, pid, nid >$, then this object was created by process with *pid* as its process-id on a host named *hname* and *nid* uniquely identifies the object within the process. A client creates a distributed object on a particular node by specifying the hostname and the process-id while specifying the third component to be zero. When the object is created, its process-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.

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 two 60MHz Sun SPARCstation 20s running SunOS 4.1.3 and Fresco version 0.7. 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[3] is started on a node. Clients then specify which object server they need to talk to in a semi-transparent manner and obtain a reference to the object; this is really a reference to the object server. The client then makes a null invocation (an equivalent of a null RPC call) on the server. Unlike in the Sun RPC case, the null 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 clients executing on the two SPARCstations create objects and access each other's objects. Clearly, invocations on a cached object are very fast - a

[3]An object server provides an address space for the distributed object.

Null Sun RPC	2.35
Null Proxy Invocation	2.7

Table 1: Null RPC and Proxy invocation timings (in milliseconds)

Creating a cacheable object	3.8
Caching an object	8.0
Method invocation on a cached object	0.02
Validating a cached object	6.2

Table 2: Times (in milliseconds) for operations on cached objects

simple invocation on a cached object takes 20 microseconds whereas the cost of a null proxy invocation is 2.7 milliseconds. This might be expected and we now proceed to examine the overheads of caching.

The cost to create a cacheable object[4] is 3.8 milliseconds. This time is dominated by the time for an inter-process communication (IPC) between the client and the corresponding process cacher. This is required for 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 6.2 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 consistency cacher on the node on which the object was created. The consistency cacher communicates with the process cacher corresponding to the client process that created the object (and therefore has a copy of the object) and receives the object state. The consistency cacher then returns the object state (and the associated timestamps) to the requesting client. 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 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 (one additional IPC). 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. We would like to point out that invalidations are local and take an insignificant amount of time.

[4]A cacheable object is an instance of a class that is a descendant of the Cached class.

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 vector timestamps that we associate with each object copy. Various efficient implementations that reduce the space overhead considerably are outlined in [32, 36]. Below, we comment on our experiences with the present CORBA specification and UNIX.

Passing object reference 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 straight-forward 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. For example, an application might want to pass "an object reference" 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 reference argument by "value" as in [6, 16, 25, 17]. 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.

Transport interface: CORBA specifications include synchronous, asynchronous, and one-way remote object invocations. 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 on which 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 ORB API does not support it. In [24], the authors argue the need for a multicast interface in the context of object groups.

UNIX and object-oriented technology: We mentioned in section 3 that we use virtual memory (VM) mechanisms for object faulting and access detection. This does lead to some compromise on the object oriented aspect of system development. For example, consider the following scenario. A client faults on a causally consistent object and experiences a segmentation fault. The object fault handler code is defined in the *Causal* class of the consistency framework. The handler returns with the new object state. At this point, the local object copy needs to be created or updated using the received object state. The *readFromString* method in the *Cached* class (or overriding method in the object's class) defines the code that needs to be used to update the object copy's 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. Our solution, which is outlined below, departs from object technology. Each 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 *readFrom-*

String method is used to update the local object copy. This could be used as a reason to adopt software based solutions instead of VM based schemes for object faulting and access detection.

Caching as an object service: The Object Management Architecture guide [29] includes a reference model which 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 differences in object caching which manifested in the design of the consistency framework can serve as the interface of the object service itself. We discussed an extension to the IDL while arguing the need to pass objects by value. We also pointed out a desirable enhancement to the interface that CORBA provides to its applications when we argued about the need for multicasting. We also broached upon the issue of providing library object adaptors which are linked with the client applications and which help find implementations of objects that get cached. Clearly more issues remain to be explored, particularly in the realm of identifying interactions with other object services. Our intention in this paper is only 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. In particular, our experiments show that the overhead of caching an object can be amortized over less than four method invocations.

Acknowledgements: We would like to thank the referees of this paper 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, G. Neiger., J. E. Burns,

P. W. Hutto, and P. Kohli. Causal memory: Definitions, Implementations and Programming. *Dist. Computing*, Vol. 9, 1995.

[2] 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.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *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. In *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] Fresco Sample Implementation Reference Manual. X Consortium Working Group Draft. Version 0.7. April 1994.

[13] 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.'93.*

[14] 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).

[15] IBM, 11400 Burnet Road, Austin, TX. *SOMObjects Developer Toolkit Users Guide,* Oct. 1994.

[16] B. Janssen, M. Spreitzer, D. Severson. Inter-Language Unification. Xerox PARC. URL: *http://www.parc.xerox.com/Projects.html*

[17] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, Feb 1988.

[18] P. Keleher, A. L. Cox, S. Dwarkadas, W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Winter USENIX Conf.*, 1994.

[19] 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.

[20] 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.

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

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

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

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

[25] 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.

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

[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] OMG Home page. *http://www.omg.org.*

[29] Object Management Architecture Guide, Revision 1.0, OMG TC Document 90.9.1. *http://www.omg.org.*

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

[31] Flex - A fast scanner generator. Information available at *http://www.ns.array.ca/ipe-1.2/doc/gnu/flex/flex_toc.html.*

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

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

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

[35] 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. In *IEEE Symp. on Parallel and Dist. Information Sys.*, 1994.

[36] F. J. Torres-Rojas. Efficient Time Representation in Distributed Systems. Masters Thesis. Georgia Inst. of Technology.

[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.