USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA, May 3–7, 1999

Implementing Causal Logging Using OrbixWeb Interception

Chanathip Namprempre, Jeremy Sussman, and Keith Marzullo
University of California, San Diego

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Implementing Causal Logging using OrbixWeb Interception

Chanathip Namprempre Jeremy Sussman
Keith Marzullo
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA, 92093-0114
{cnamprem,jsussman,marzullo}@cs.ucsd.edu

Abstract

Some form of replicated data management is a basic service of nearly all distributed systems. Replicated data management maintains the consistency of replicated data. In wide-area distributed systems, *causal consistency* is often used, because it is strong enough to allow one to easily solve many problems while still keeping the cost low even with the large variance in latency that one finds in a wide-area network. *Causal logging* is a useful technique for implementing causal consistency because it greatly reduces the latency in reading causally consistent data by piggybacking updates on existing network traffic.

We have implemented a CORBA service, called COPE, that is implemented by using causal logging. COPE also shares features with some CORBA security services and is naturally implemented using the OrbixWeb interception facilities. In implementing COPE in OrbixWeb, we encountered several problems. We discuss COPE, its implementation in OrbixWeb, and the problems we encountered in this paper. We hope that this discussion will be of interest to both those who are implementing and who are planning on using CORBA interception facilities.

1 Introduction

In nearly all distributed systems, some data values are replicated across different processors. For example, one might have a distributed cache to allow reads to be performed more quickly. Or, one might have multiple copies of a critical data structure to ensure that should a processor crash or become isolated by a network failure, a copy of the data will still remain accessible. Replicated data management is therefore an essential service of distributed systems.

One issue arising in replicated data management is how *consistent* the replicas need to be. Considerable effort has gone into defining and analyzing different consistency models. A very strong requirement is for the replicas to reflect the real-time order in which they were written. For example, consider two values x and y that are replicated on processors a, b, c and d, and let the initial values of x and y be zero. If processor a sets x to 1 before processor b sets y to 2, then c and d will both see x set to 1 before seeing y set to 2 (the same, of course, holds for a and b). This kind of consistency, called *atomic consistency* [7], is in general expensive to implement, and thus used only when absolutely required. A weaker form of consistency is called *sequential consistency* [6] in which *some* total order on updates is imposed. With the above example, c and d will both see the same order of updates to x and y, but not necessarily the update of x before the update of y. Sequential consistency is less expensive to implement than atomic consistency and yet is often strong enough for many applications.

A still weaker consistency property is *causal consistency* [1]. Suppose that processor b did not update y until it read x and found a value of 1. In this case, we say that the value of y *causally depends* upon the value of x—that is, the act of a setting x to 1 in some sense caused b to set y to 2. Causal consistency ensures that the sequence of updates as read by other processors is consistent with causal dependency. In this case, both c and d would see the update of x before the update of y. On the other hand, if b did not read x before setting y, then the

updates are said to be *concurrent*. Concurrent updates are not ordered, and so c could see x updated before y while d could see y updated before x.

Preserving only causal dependencies among replicated data is sufficient for many applications [1]. We give an example of such an application later in this paper, namely optimistic execution in an object-oriented environment. And, causal consistency is cheaper to implement than sequential consistency in a wide-area setting. A problem with wide-area networks is the large variance in communication latency. With sequential consistency, a processor that updates a shared variable must ensure that its update is ordered with any other potentially concurrent updates, and so the latency of an update can be no smaller than the longest latency from the updating processor to a copy of the data [11]. Causal consistency does not require concurrent updates to be ordered and so a processor can simply update its local copy and continue. There can, however, be latency introduced when reading a shared variable [10].

Latency can be reduced by implementing causal consistency through a technique called *causal logging* [3]. With causal logging, a message that updates a shared variable piggybacks the updates upon which the variable causally depends. That is, causal logging trades off bandwidth for latency. Causal logging has been used in several applications besides implementing causally consistent replicated data, including distributed simulation [5] and techniques for low-cost failure recovery [2]. These applications all share the same general property: a process does not observe an action (such as the delivery of a message or the update of a shared variable) until it has observed all actions that the observed action causally depends upon.

In one of our research projects, we faced the problem of implementing causal logging when constructing a CORBA service. We call this service *COPE* and briefly describe it in Section 2. To implement this service in CORBA, we hoped to use an *interception* facility. Interception is a way to add functionality to CORBA services in a manner that is orthogonal and non-intrusive to the main computation. CORBA interception is implemented using *interceptors* which are code that can be invoked upon a message being sent, or upon a message being received (as well as other *trigger* points). We describe why this facility allows for a straightforward implementation of causal logging, as well as describing this implementation, in Section 3. We chose OrbixWeb as the platform for implementing COPE because it is a popular Java ORB that provides interception facilities. We detail these features in Section 4.

COPE is a somewhat complex CORBA service, and to the best of our knowledge no other group has considered a service with similar functionality. It shares some features with proposed CORBA security services, most notably *Application Access Policy* [9]. In implementing COPE, we encountered several problems with OrbixWeb, which we describe in Section 5. We believe that the problems we encountered will also be encountered by those implementing similar CORBA services. Our goal in writing this paper is to discuss the problems we encountered in hope that those building CORBA ORBs will be aware of them when building interception facilities.

2 COPE

We have implemented a new CORBA service called COPE that is based on causal logging. We give a brief overview of COPE to ground the discussion in Section 3 on what we require of a CORBA causal logging implementation.

One of the two abstractions that COPE implements is the class of *assumptions*. An assumption is a CORBA object that is eventually either asserted or refuted. An assumption keeps track of the objects that wish to be notified when it is resolved. Assumptions can also be subclassed to associate semantics with them, such as assumptions that depend on other assumptions. An example of such an assumption is a *proposition* which is expressed as a boolean formula over a symbol table. Each entry in the symbol table is itself an assumption. A proposition assumption becomes asserted or refuted when the value of its formula evaluates to *true* or *false* as determined by the assumptions that have been asserted and refuted in the symbol table.

The other abstraction that COPE implements is the class of *optimists*. An optimist is a CORBA object that takes on assumptions. Optimists can execute *optimistically* based on the assumptions that it has taken on. More specifically, an optimist can either checkpoint its state when it takes on an assumption or it can block, awaiting the eventual assertation or refutation of the assumption. If, in either case,

the assumption is asserted then the optimist is notified so that it can either discard the checkpoint or continue execution. If, on the other hand, the assumption is refuted, then the optimist is notified so that it can either roll it state back to the associated checkpoint or continue execution knowing that the assumption was refuted.

Assumptions are causally consistent with respect to CORBA communications. For example, consider optimist a invoking method $b.m$ on optimist b. If a has taken on an assumption x which is still unresolved by the time a invokes $b.m$, then b must take on x by the time it begins execution of $b.m$. Similarly, if $b.m$ constructs an optimist c, then c must also take on x by the time c completes initialization.

Put into terms of shared memory, each optimist has a list of replicas of unresolved assumption. These replicas are causally consistent, where "causally depends" is defined in terms both of optimists invoking methods on other optimists and of optimists creating new optimists. This list is maintained as follows:

1. When an optimist a makes a method invocation on an optimist b, it piggybacks on the method invocation a list of unresolved assumptions that a has taken on.

2. When an optimist b has a method invoked by an object a, b checks to see if a has a class that derives from *Optimist*. If so, then b strips off any assumptions piggybacked on the method invocation and decides whether to add them to its own list of unresolved assumptions or to block the invocation.

3. When an optimist a creates an optimist b, it makes a method call to an *optimist factory*. As when invoking a method on an optimist, a piggybacks on the method invocation a list of assumptions that a has taken on. The factory f checks to see if a has a class that derives from *Optimist*. If so, it then passes a's unresolved assumptions to b. Optimist b can choose either to accept a's assumptions, in which case the creation is successful, or to deny them, in which case b is not created.

These three rules for maintaining the list of assumptions together implement causal logging of assumptions. Other features of COPE, such as assertion resolution and object notification, are not germane to the discussion. Interested readers can find further details on COPE in [8].

3 Implementing Causal Logging Using Interception

Consider implementing causal logging on top of CORBA. The following properties of an implementation state what we believe constitutes a well-engineered solution.

- **Transparency**. The piggybacking and stripping of piggybacked information should be implemented without explicit involvement of the objects using causal logging. To do otherwise would make it hard to ensure that the causal logging mechanism is correct.

- **Scheduling**. Causal logging implies that information is made available to an object at certain points in its execution. To do otherwise might violate the causal consistency condition. Hence, the causal logging mechanism notifies of the delivery of causal information ordered with respect to the invocation of methods and creation of new objects.

- **Context Sensitivity**. The information that an object a piggybacks to an object b may depend both on the state and class of a and on the class of b. Without knowledge about a, it is hard to piggyback *any information* because there is no way to obtain it, and without information about b an object would have to piggyback all possibly useful information on every method invocation. The latter would both be inefficient and would pose a possible security problem.

CORBA interception is ideally suited as a piggybacking mechanism that provides the properties of transparency and scheduling. Interceptors are orthogonal to the regular path of computation, and therefore provide transparency. Furthermore, since interception can be placed at many points in the method invocation sequencing, it can be used to provide scheduling as well.

A simple implementation of causal logging would be as follows. Consider an interception mechanism in

which every object has an interceptor that is specific to that object. The interceptor knows the identity of the object with which it is associated, and the interceptor is invoked upon both ends of a method invocation—that is, by the invoked object and by the invoking object.

When a method is invoked, the interceptor on the invoking object uses some mechanism to determine what type of information is to be piggybacked. This mechanism can base its determination on the class of the invoked object. The actual information can be determined from the current state and the class of the invoking object. Hence, context sensitivity is implemented. The interceptor adds this data to the outgoing method invocation.

When the invoked object receives the invocation, its interceptor removes the data that was added by the invoking object's interceptor. The invoked object's interceptor then implements scheduling by ordering the method invocations that deliver the causally logged information with respect to the incoming method invocation.

This scheme implements encapsulation of the method invocations within the causal logging mechanism. That is, the underlying method invocations are not altered, but rather are used as a conduit of causally logged information and are scheduled to maintain causal consistency. The CORBA interception facility is intended for exactly this kind of encapsulation. Our simple model of CORBA interception requires the following capabilities:

1. Interceptors should be invoked on all outgoing and incoming invocations.

2. Interceptors should be able to add information when a method invocation is initiated at the invoking side and remove information when the method invocation is initiated at the invoked side.

3. An interceptor on the invoking side should know the state and class of the object with which it is associated and the class of the object being invoked.

4. An interceptor on the invoked side should have the ability to make method calls on the object with which it is associated before it allows the initiating method to be invoked.

As discussed in Section 5, we had a few difficulties in creating such an architecture in OrbixWeb.

4 OrbixWeb

In COPE we implement interception by using OrbixWeb filters. OrbixWeb is an implementation of CORBA by IONA Technologies Inc. It is compliant with the Object Management Group's (OMG) CORBA specification Version 2.0 and is implemented in Java. It provides a filtering mechanism with support for piggybacking of additional information onto method invocations. There are two types of filters in OrbixWeb: *per-process* and *per-object* filters. We describe both types in turn.

4.1 Per-Process Filters

A per-process filter is code that is associated with a client or server process. The filter monitors all incoming and outgoing method invocations and attribute references that reference objects associated with another process. More than one process filter can be chained together to the same process. There are ten points where code in a filter can be associated with a process: inRequestPreMarshal, outRequestPreMarshal, inReplyPreMarshal, outReplyPreMarshal, inRequestPostMarshal, outRequestPostMarshal, inReplyPostMarshal, outReplyPostMarshal, inReplyFailure, and outReplyFailure. Figure 1 (taken from [4]) illustrates these filter points.

The name of a filter method indicates where in the method invocation sequence it is invoked. The modifier *request* or *reply* indicates whether the filter is associated with the invocation of the method or the reply from the method. The modifier *in* or *out* indicates the direction the method invocation or method reply is going with respect to the process with which the filter is associated. In particular, *out* indicates the invoking object for method invocations and the invoked object for method reply, and *in* indicates the invoked object for method invocations and the invoking object for method reply. The stem indicates exactly where in the processing of the method invocation the filter is associated: *PreMarshal* is before parameter marshalling, *PostMarshal* is after parameter marshalling. *Failure* is for exceptions. Specifi-

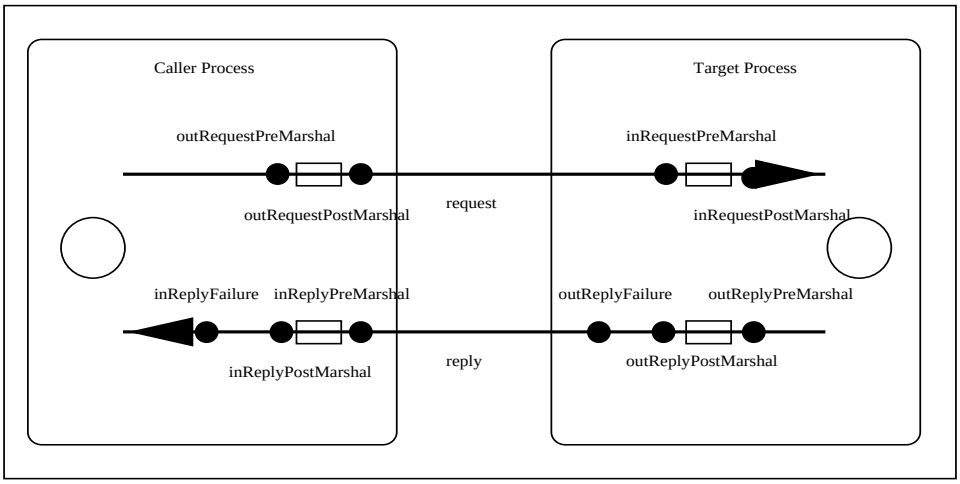

Figure 1: Per-process filter monitor points

cally, code that is associated with failure filter points is executed under two conditions: (1) when an exception condition is raised by the target of the invocation or (2) when there are return values from any preceding filter points indicating that the call is not to be processed any further.

The OrbixWeb abstract class `IE.Iona.OrbixWeb.Features.Filter` implements per-process filters. A user-defined filter is implemented by defining a class that inherits from the `Filter` class. When a process creates an object of this class, the newly-created filter is associated with the creating object's process. Successive filter creation results in these filters being chained in the order of their creation.

The following demonstrates the construction of a per-process filter [4]:

```
1  public class ProcessFilter extends Filter {
2  public boolean outReplyPreMarshal(Request r)
   {
3   String s, o;
4   long l = 27;

5   try {
6    s = ORB.init().object_to_string(
         (r.target()));
7    o = r.operation ();

8    OutputStream outs =
9    _OrbixWeb.Request(r).create_output_stream();
10   outs.write_long (l);
11  } catch (SystemException se) {
12    System.out.println("Caught exception "+se);
    }

13  System.out.println ("Request to "+ s);
14  System.out.println ("with operation "+ o);
15  return true; // continue the call
    }
   }
```

Information about the invocation such as the target, operation name, and arguments can be accessed through the parameter `r`, which is of type `IE.Iona.OrbixWeb.CORBA.Request`. For example, the call `r.target()` in Line 6 of the code above returns the target of the invocation while the call `r.operation()` in Line 7 returns the name of the operation being invoked.

As mentioned previously, OrbixWeb also allows extra information to be piggybacked onto the method invocation. The call `_OrbixWeb.Request(r).create_output_stream()` in Line 9 above creates a stream to which extra information can be written (`outs.write_long(l)` in Line 12). This information can later be read by the corresponding filter point on the other side of the invocation (in this example, an `inReply` filter on the client side).

4.2 Per-Object Filter

Filters can be associated with a given object as well. To define a per-object filter, one defines a Java object that implements the Java interface for the CORBA object that was generated by the IDL compiler. This new Java object implements the desired filters. For example, assume the CORBA object a provides a method m. The IDL compiler generated Java interface includes the declaration of method m. Any per-object filter that can be associated with the class a must implement this interface, including the method m. The association is done by having the implementation of a create an instance of the per-object filter object, and then specify where in the method invocation path the filter is to be invoked. The following demonstrates the association of two per-object filters `filter1` and `filter2` with an object of class `Foo`:

```
1   Foo foo;
2   (( _FooSkeleton) foo)._preObject
       = filter1;
3   (( _FooSkeleton) foo)._postObject
       = filter2;
```

This mechanism for implementing a per-object filter is simple and elegant. However, as will be discussed in Section 5.2, it is too restrictive for our purposes.

5 Problems

In building COPE, we encountered difficulties in using OrbixWeb. This section describes two of these difficulties:

1. Representing the CORBA inheritance of an object in the underlying Java implementation.

2. The lack of support for a generic per-object filter.

We also give our corresponding workarounds and evaluate their effectiveness.

5.1 CORBA Inheritance Issues

This problem arose because COPE piggybacks references to assumption objects, and it is common to derive specific kinds of assumptions from the assumption class. It is always somewhat complex figuring out how to implement a CORBA-based program using a specific ORB, but figuring out how to structure COPE was especially hard. It took approximately a month of mail exchange with Iona before the problem was understood well enough to resolve.

Most ORBs are implemented using an object-oriented language, such as Java or C++. A CORBA-based application is defined, in IDL, as a set of classes that may be related via single inheritance. The IDL compiler translates these inheritance relations in some manner into a set of class definitions in the implementation language. Consider a CORBA class `Bar` that inherits from a CORBA class `Foo`. The implementation object (say, `BarImpl`) should also inherit from the implementation object `FooImpl`. In addition, with most IDL compilers both implementation objects are instances of a base CORBA object class.

The problem discussed here is concerned with the translation chosen by the OrbixWeb IDL compiler.[1] Before discussing the problem further, we provide some background concerning how objects are implemented in OrbixWeb. Suppose we have an IDL interface as follows:

```
interface Foo {
   void foo_method();
};

interface Bar : Foo {
   void bar_method(Foo f);
};
```

The OrbixWeb IDL compiler generates eight files for each CORBA interface. For example, the CORBA interface `Foo` is compiled into `FooHolder`, `FooHelper`, `_FooSkeleton`, `_FooStub`, `_FooImplBase`, `_tie_Foo`, `_FooOperations`, and `Foo`. The first two are helper classes that support marshalling, narrowing, and other CORBA support

[1] Recall that Java supports only single inheritance, and so a translation based on multiple inheritance is only possible through interfaces, not classes.

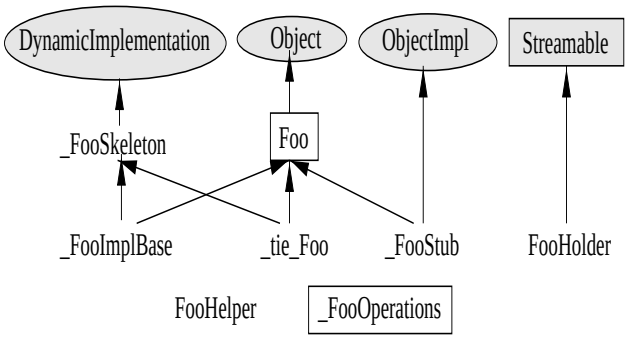

Figure 2: IDL-generated files for `Foo`

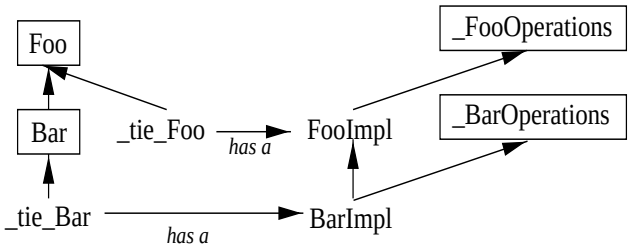

Figure 3: Inheritance Diagram

operations. The next two are classes that are placeholders for the stubs. The last four, two classes and two interfaces respectively, are described in more detail below.

The complete inheritance hierarchy for `Foo` is depicted in Figure 2. Rectangles represent interfaces. Shaded ovals and rectangles represent classes and interfaces provided in standard packages such as `org.omg.CORBA`. These packages are part of OrbixWeb core classes.

There are two approaches with which one can implement an OrbixWeb object: the "ImplBase" approach and the "tie" approach [4]. Suppose you wish to implement the CORBA class `Foo` with the Java class `FooImpl`. In the ImplBase approach, `FooImpl` has the following signature:

```
public class FooImpl extends _FooImplBase
```

That is, `FooImpl` is a subclass of the class `_FooImplBase`. And, since `_FooImplBase` implements the Java interface `Foo` (See Figure 2), `FooImpl` also implements `Foo`. Therefore, a CORBA `Foo` object can be instantiated as follows:

```
Foo foo = new FooImpl();
```

The tie approach, in contrast, is a delegation model. With this approach the `FooImpl` class implements the `_FooOperations` interface. However, since `_FooOperations` and `Foo` are both interfaces, to instantiate a CORBA `Foo` object one first instantiates a `FooImpl` instance and then instantiates a `_tie_Foo` instance with the `FooImpl` instance as a parameter:

```
Foo foo = new _tie_Foo( new FooImpl() );
```

Since `_tie_Foo` implements `Foo`, the variable `foo` has type `Foo` as desired.

Now consider implementing the class `Bar`. One would naturally wish the implementation `BarImpl` to inherit from `FooImpl`. But, `BarImpl` also implements the methods declared in the CORBA `Bar` class. As was done in implementing `Foo`, we can use either the ImplBase approach or the tie approach. However, the ImplBase approach requires `BarImpl` inheriting from `_BarImplBase`. This implies multiple inheritance of classes, which Java does not support. Thus, one is constrained to use the tie approach, viz.:

```
public class BarImpl extends FooImpl
   implements _BarOperations
```

Since we are using the tie approach, an instance of `Bar` is created by wrapping an instance of `BarImpl` in an instance of `_tie_Bar`:

```
Bar bar = new _tie_Bar( new BarImpl() );
```

Figure 3 illustrates the resulting inheritance diagram of a `FooImpl` object and a `BarImpl` object.

The problem with the tie approach, however, is that the implementation objects (`FooImpl` and `BarImpl` in this example), are not instances of the Java interface that represents the CORBA object (`Foo` and `Bar` in this example). This poses a problem when using CORBA operations.

For example, suppose that there is a CORBA `Bar` object `B1` on processor 1 and a CORBA `Bar` object `B2` on processor 2. `B2` has a reference `r` to `B1`, and invokes the method `r.bar_method(this)`.

Further suppose that the implementation of `bar_method` in `BarImpl` tests the parameter `f` to see if it is an instance of `Bar`:

```
1  public bar_method (Foo f) {
2    if (f instanceof Bar)
3       ...
4    else
5       ...
     }
```

One might expect that the code in Line 3 would be executed, but it is not due to an implementation decision by Iona. While marshalling `this` on `B2`, CORBA determines the class of the value it is marshalling through a method on it named `type`, e.g., it invokes `this.type()`. If `this` were to implement the Java interface `Bar`, then `this.type()` would return a value indicating the CORBA class `Bar`. But, since `this` implements `_BarOperations`, `this.type()` returns the value `null`. The marshalling code therefore declares the parameter passed in the message to `B1` to be of type reference to `Foo`.

We dealt with this problem in the manner recommended by Iona. We save in the Java implementation of every CORBA object a reference to the tie object. For example, let the member variable referring to the tie object of an instance of `Bar` be `tieObject`. The declaration of `tieObject` and the constructor in the definition of the class `BarImpl` can be as follows:[2]

```
public class BarImpl
  implements _BarOperations {
  protected Bar tieObject; // declaration

  BarImpl() {
     ...
     tieObject = new _tie_Bar( this );
     }
  }
```

[2] Note that the setting of `tieObject` must be the last member variable initialization. If not, then the member variables of the tie object may be initialized to incorrect values.

Then, `B2` invokes `r.bar_method(tieObject)` instead of `r.bar_method(this)`. Since `tieObjects` implements `Bar`, `tieObject.type()` returns a value indicating the CORBA class `Bar`.

This problem occurs in other situations. It is in general a good OrbixWeb design practice to have objects like `BarImpl` implement a method that returns a reference to the tie object. This reference should be used in all places where a reference to the implementation is passed using CORBA.

This additional complexity in structure can be avoided by enforcing a file naming structure on the user's code. For example, some IDL compilers generate a file that the user edits to include the Java implementation of the CORBA object. Unlike OrbixWeb, the IDL compiler knows the name of the implementation class when it generates the files. Hence, the IDL compiler can generate files that explictly inherit from this class as needed. In our example, if the IDL compiler names the implementation class for `Foo` as `FooObj`, then the implementation class for `Bar` might be generated as

```
public class BarObj
   extends FooObj
   implements _BarOperations
```

5.2 Per-Process Filters

The filters that implement COPE's causal logging perform the same operations for each method of an optimist object: they add assumptions to a method invocation on the invoking object's side and remove the assumptions on the invoked object's side. Ideally, one would like to be able to associate the same filter with each method of any class that derives from optimist, and this association should be done in a general way. Unfortunately, this cannot be done in OrbixWeb. As discussed in Section 4.2, an object that implements per-object filters is required to implement all methods defined in the IDL definition for the class a of objects with which it is associated. Classes that inherit from a require their own filters to be explicitly implemented.

Since a general purpose filter cannot be constructed as a per-object filter and since we are unwilling to change the OrbixWeb IDL compiler, a per-process filter is our only option. A per-process filter is invoked for all method invocations leaving and enter-

ing the process, and so it can be used to implement a generic filter. However, using per-process filters raises other problems. We have found workarounds for these problems, but we do not believe that the workarounds are acceptable in terms of meeting our engineering requirements.

5.2.1 Performance

There are performance reasons leading one to implement several objects in the same process. It is relatively inexpensive for these objects to invoke each other's methods, since such an invocation does not require a context switch. Furthermore, there are serious problems in resource utilization with running multiple Java virtual machines on the same processor. Hence, one often tries to structure an OrbixWeb application with as many objects as possible in the same process.

However, per-process filters are not invoked for method invocations between objects in the same process. Hence, per-process filters can not be used to implement causal logging among objects in the same process. This implies that each COPE optimist must run in its own process. This solution carries an enormous performance penalty.

5.2.2 Lack of Required Information

An OrbixWeb filter can obtain various information about the invocation that it is intercepting. This information includes the reference of the object whose method is being invoked, the name of the method being invoked, the parameters of the method being invoked, and the name of the user running the program that resulted in the invocation. Unfortunately, the filter cannot determine the reference of the object making the invocation.

Without this information, it is impossible to provide the context sensitivity property defined in Section 3. The reason is that the piggybacked data depends on the state of the invoking object. Thus, the filter cannot obtain the information to be piggybacked from the invoking object.

A per-object filter *is* aware of the object with which it is associated, and so does not suffer from this problem. However, as we saw in Section 5.2, per-object filters cannot be used. Hence, we needed to find a workaround.

Because of the constraint of having only one object per process, we work around this problem by allocating a static variable that contains a reference to the tie object. This static variable is referenced by the filters when they need to make a invocation on the invoking process. This is a simple workaround, but it is artificially simple because of the constraint of one object per process. If the per-process filter could be imposed for communications between objects in the same process, then this static variable would need to be updated before every method invocation. Doing so would violate transparency of Section 3.

6 Discussion

In implementing causal logging to achieve causal consistency for our CORBA service, COPE, we use the interception facilities provided by OrbixWeb filters. OrbixWeb filters allow us to add functionality to legacy software in a manner that is orthogonal and non-intrusive to the main computation. Using filters, invocations in the system can be captured and processed before continuing with the normal flow of the program. However, despite our best endeavors, we were faced with a few difficulties. One problem was in understanding how to structure an OrbixWeb application that passes references to CORBA objects that can be subclassed. With the help of Iona, we were able to find a practical solution. The two remaining problems were more serious:

1. One cannot impose a per-object filter that is generic—that is, that need not conform to the interface implemented by the object. The level of interception implementable using OrbixWeb filters therefore poses a fundamental problem. As we discovered in our case, getting around the problem incurs a very high performance penalty and is thus not a practical solution.

2. One has no means to access the calling object in a filter. This violation of context sensitivity property leads to the static variable solution which, except for the problem above, would violate the transparency property.

It is not clear why such a limit to disallow access to the calling object was imposed in the first place. But with this limitation, the piggybacking mechanism in place cannot be utilized to its full potential.

We believe that any communications middleware platform for distributed computing should be powerful enough to build the engineering solution described in Section 3. Such a solution might even be considered a benchmark for the utility of such platforms.

Although we have not attempted to implement COPE using other CORBA ORBs, we have looked into using the Legion system [12]. And, Legion appears to be powerful enough to efficiently implement COPE, but it would be interesting to actually do so to see what problems might arise. Of course, it would also be interesting to see how well other CORBA ORBs could support COPE.

7 Acknowledgments

We would like to thank Treasa O'Shaughnessy, Anne O'Shea and John O'Shea of IONA who helped us work through the technical problems we have encountered with OrbixWeb. We were unable to resolve the problem described in Section 5.1 without their help. We benefited from discussions with Ang Nguyen-Toung at the University of Virginia of how COPE might be implemented on Legion. The COPE architecture was developed by the authors and Ramanathan Krishnamurthy and Aleta Ricciardi, both of the University of Texas at Austin.

This work was supported by the Defense Advanced Research Projects Agency (DoD) under contract number F30602-96-1-0313. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

8 Availability

The source of COPE is available from the COPE home page at the following URL:

`http://www.cs.ucsd.edu/users/marzullo/COPE.html`

References

[1] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal memory. In *Distributed Algorithms, Fifth International Workshop WDAG '91*, pages 9–30, October 1991.

[2] Lorenzo Alvisi and Keith Marzullo. Message logging: pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[3] K. P. Birman and T. Joseph. Reliable communications in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] IONA Technologies Inc., Dublin, Ireland. *OrbixWeb Programmer's Guide*, 1997.

[5] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[6] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] Keith Marzullo, Chanathip Namprempre, Jeremy Sussman, Ramanathan Krishnamurthy, and Aleta Ricciardi. Combining optimism and intrusion detection. Technical Report TR CS98-605, University of California, San Diego, Department of Computer Science and Engineering, October 1998.

[9] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: Common Object Request Broker Architecture*. Prentice Hall Press, October 1995.

[10] Michel Raynal, Andre Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[11] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[12] C. Viles, M. Lewis, A. Ferrari, A. Nguyen-Toung, and A. S. Grimshaw. Enabling flexibility in the Legion run-time library. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97)*, pages 265–274, June 1997.