Using Value Types to Improve Access to CORBA Objects

Markus Aleksy and Stefan Kuhlins*

University of Mannheim
Department of Information Systems III
D–68131 Mannheim
Germany
Tel.: +49 621 181-1642
E-mail: {aleksy|kuhlins}@wifo3.uni-mannheim.de

Abstract

In this paper, we describe a new approach to enhance the performance of CORBA-based applications. For that purpose, we improve access to CORBA objects with the help of value types. In this way, only few changes are necessary to speed up existing CORBA applications. Moreover, we discuss several techniques for updating cache objects within the scope of the CORBA event service. Finally, we compare the performance of these techniques.

1. Introduction

At the present time, several techniques are available to develop distributed applications. In the world of object-oriented and distributed systems, the *Object Management Group*'s (OMG) *Common Object Request Broker Architecture* (CORBA) is widely used [8]. Implemented with CORBA, applications are independent of specific computer architectures, operating systems, programming languages, and *Object Request Brokers* (ORB). An ORB enables objects to transparently send and receive requests and responses in a distributed environment.

Since version 2.3 (December 1998), CORBA supports two kinds of objects [8]:

- CORBA objects
- Value type objects

CORBA objects are CORBA's regular objects. In the *Interface Definition Language* (IDL), these objects are defined by the keyword `interface`. Strictly speaking, a "CORBA object" is an instance of an interface type. An interface type specifies a set of operations that an instance of that type must support. For arguments and return values of operations CORBA objects are passed by reference [8]. That is, the client uses a reference to access the remote object. Since remote calls are very expensive compared to local calls, the use of CORBA objects in conventional algorithms and data structures is not efficient.

In order to solve this problem, the OMG introduced the concept of *value types* [8]. A value type specifies state as well as a set of operations that an instance of that type must support. So, value types share many of the characteristics of interfaces and `struct`s. In IDL, value types are defined by the keyword `valuetype`.

Value type objects (strictly speaking, instances of a value type) are passed by value when used as operation arguments or as return values. The recipient of a parameter passed by value is provided with a description of the object's state. Then, it creates a new instance with that state. It should be noted that it is a different object and not the same instance – which implies that it has a different identity. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances. That is, there is no explicit or implicit sharing of state. Implementations of value types are local to their creating context. Hence, their operations are called in a local context and will not require remote calls.

Across or within other instances, an instance of a value type can be shared. If, for example, the same instance of a value type is used for two parameters of an operation only one instance is created on the receiver's side.

Altogether, value types can be used in conventional algorithms and data structures efficiently.

The remainder of this paper is organized as follows: Section 2 shows that the use of `struct`s instead of value types is less suitable. In Section 3, we introduce a simple example, which is used in the following sections for demon-

*Accepted for SNPD'01 – 2^{nd} International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing, Nagoya Institute of Technology, Japan; August 20–22, 2001; http://www-ishii.ics.nitech.ac.jp/snpd01/

stration purposes. Section 4 describes how cache objects could be notified about state changes. Section 5 contains the main part of our work. Here, we present several designs for caching techniques. In Section 6, we evaluate the efficiency of the aforementioned techniques with some performance experiments. A summary of the results is given in Section 7. Section 8 discusses related work, and Section 9 concludes the paper with some final remarks on value types.

2. Before Value Types

The administration of lectures is a typical example for an application where clients read objects on the server frequently, but those objects rarely change. After creation, the attributes of lectures (e.g., title, date, time, location, and speaker) are seldom modified. If modifications are necessary the new object state has to be transferred to clients.

Before value types were part of the CORBA standard, the only way to improve performance of such applications was to define a `struct` that includes the relevant attributes. `struct`s are transferred by value, and thus cause less overhead. However, they do not have methods, and therefore access to attributes cannot be restricted. Hence, the object-oriented principle of information hiding is violated. Moreover, if changes to state of the underlying object happen, the `struct` has to be modified too.

Value types offer a new solution to the problem. Like `struct`s, they are passed by value, but unlike `struct`s, they can have methods to control attribute access.

3. Example

In the scope of this paper we are restricted to simple examples. Therefore, we present our caching strategies by means of a straightforward `Counter` interface. A `Counter` object has only one attribute `value` (the value of the counter) and two operations for incrementing and decrementing this value (`inc()` and `dec()`, respectively). Figure 1 contains the IDL definition of the `Counter` interface.

Since `Counter` is a CORBA object, accesses to the `value` attribute as well as calls of the operations `inc()` and `dec()` are remote calls making these calls relatively expensive.

In comparison to `Counter`, `CachedCounter` is a value type. Therefore, copies of `CachedCounter` objects are transferred to clients.

A `CachedCounter` supports the same interface as `Counter`. This is expressed by the IDL keyword `supports`. The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance. Value types may be derived from other

```
module CounterApplication
{
  interface Counter
  {
    readonly attribute long value;
    void inc();
    void dec();
  };

  valuetype CachedCounter
    supports Counter
  {
    public Counter realCounter;
  };

  interface CacheFactory
  {
    CachedCounter create();
  };
};
```

Figure 1. IDL definitions

value types and can support an interface and any number of abstract interfaces [8]. From the client's point of view, access to a `CachedCounter` value type is the same as access to a `Counter` object, except that performance is improved.

Clients could use the `CacheFactory` interface to create `CachedCounter` value types. In that case, the `CachedCounter` value type is created and initialized on the server. After that, the `CachedCounter` value type (bearing the same state as the `Counter` object) is transferred to the client and ready to use.

The first access to a `CachedCounter` value type should ask for the actual state of the `Counter` object on the server, because the state of the `Counter` object that belongs to the `CachedCounter` could have changed since its transmission.

4. Change Notification

Through calls to the operations `inc()` and `dec()` clients can cause `Counter` objects on the server to change their state. All clients must be notified of such state changes so that they always work with an up-to-date copy of the underlying `Counter` object. One way to solve this problem is to use the CORBA event service [7].

The CORBA event service does not belong to the CORBA core model; however, it was integrated into the OMG

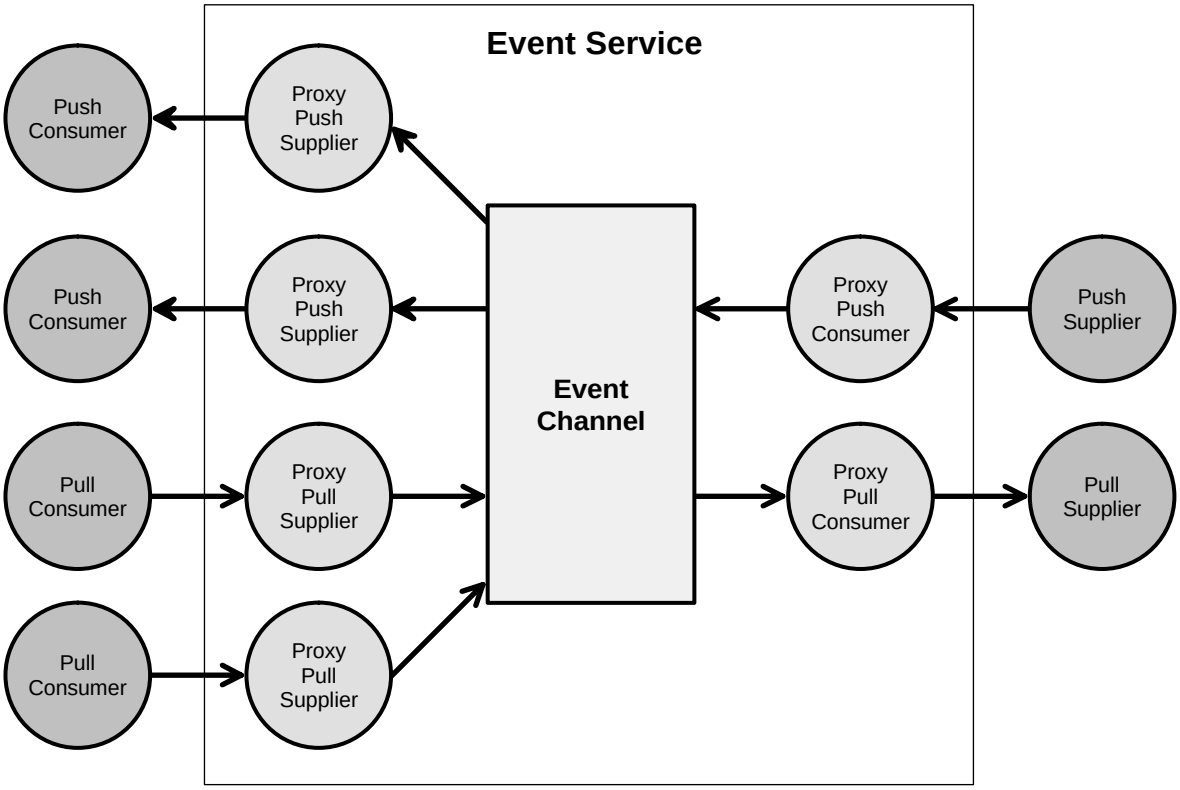

Figure 2. Different models of the event service

standard along with CORBA services [6]. The event service is based on the ”publish/subscribe”-pattern [2]. That is, users of the event service are divided into *event suppliers* and *event consumers*. Suppliers produce event data, and consumers process event data. Event data is communicated between suppliers and consumers by issuing standard CORBA requests.

The core of the event service is the event channel. An event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is itself both a consumer and a supplier of event data. Consumers can either request events or be notified of events; whichever is more appropriate for application design and performance. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests. [7]

The event service provides the following models (see Figure 2):

- Pure pull model
- Pure push model
- Mixed pull/push model
- Mixed push/pull model

In the pull model the event consumer (”pull consumer”) plays the active role, i.e., it has to determine whether events are available. In the push model the event consumer (”push consumer”) behaves passively. It is informed automatically when an event occurs. Both models are also valid for event suppliers, but in this case the roles are reversed. That is, the pull supplier operates passively whereas the push supplier has to deliver events to the event channel actively.

A proxy supplier is similar to a normal supplier (in fact, it inherits the interface of a supplier), but includes an additional method for connecting a consumer to the proxy supplier. In the same way, a proxy consumer acts as a normal consumer.

The use of the event service offers great flexibility as well as a standard way of data exchange. Any data type or data structure can be used to describe an event. This information is of data type `any` – it can be as simple or as complex as necessary.

Another advantage of the event service is that it decouples event suppliers and event consumers. In case of failure of an event supplier, it can be replaced with a replica without event consumers realizing this.

As shown in [9], an event service from a specific vendor can be used together with servers and clients that rely on an ORB from a different vendor without any problems. This

could be another reason to use a standard service instead of a self-written solution.

5. Design Considerations

In the case of CORBA objects, communication between client and server is based on the usual access to a remote CORBA object (see Figure 3).

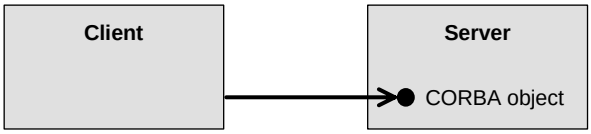

Figure 3. Usual access to a remote CORBA object

Bringing value types into play, the simplest approach is a direct communication between client and server via pull (see Figure 4). Here, the value type object interrogates the state of the underlying CORBA object periodically.

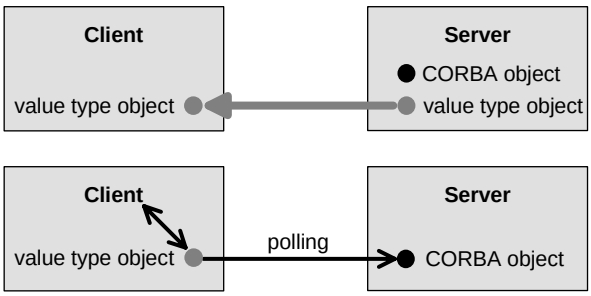

Figure 4. Simple pull model

The upper part of Figure 4 shows creation and transmission of a value type object. This is always the first step when value types are used.

A better approach uses the event service to decouple client and server. In this case, communication between a remote value type object and the underlying CORBA object on the server can be realized via pull (see Figure 5). This approach is characterized by a relatively complex implementation of the value type object, because it contains the whole polling logic.

In order to reduce the complexity of the value type object's implementation, an auxiliary CORBA object on the client can be used (see Figure 6). The auxiliary object interrogates the state of the CORBA object on the server periodically via pull model. In case of state changes, the auxiliary object informs the value type object. This way, the value type object behaves like a cache. Its implementation is simple, therefore its size is small and transfer time is short. One

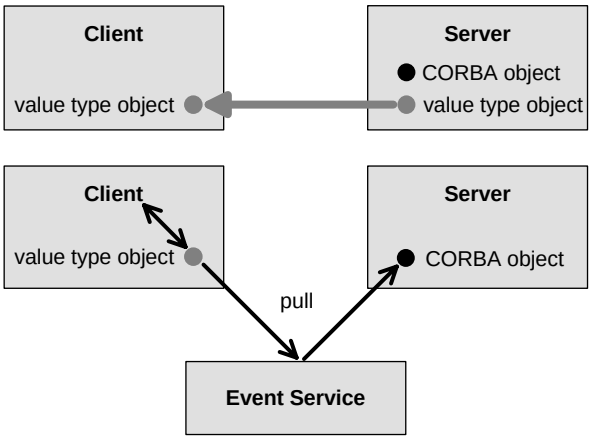

Figure 5. Pull model

drawback of this approach is the need for an extra object on the client side.

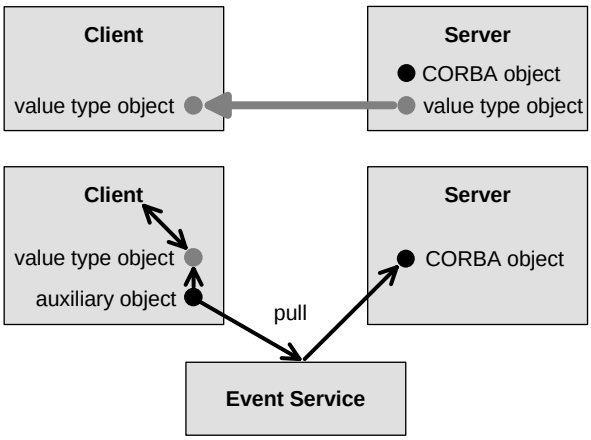

Figure 6. Pull model with an auxiliary object

All approaches mentioned so far suffer from the same problem. If the polling time is too low clients realize changes too late. If it is too high net resources are wasted. To overcome this problem, the CORBA object on the server should inform its clients when state changes. For that purpose, the last approach uses the push model of the event service (see Figure 7).

As in the previous approach, an auxiliary CORBA object on the client is needed. The reason for this is that CORBA objects and value type objects are strictly separated. The *Portable Object Adaptor* (POA) manages CORBA objects. They are identified unambiguously by the *Interoperable Object Reference* (IOR). On the other side, value type objects are not managed by the POA. They exist only in a local context. As shown in Figures 4 and 5, a value type object can reference a CORBA object. In the reverse case, a CORBA object cannot reference a value type object. Hence,

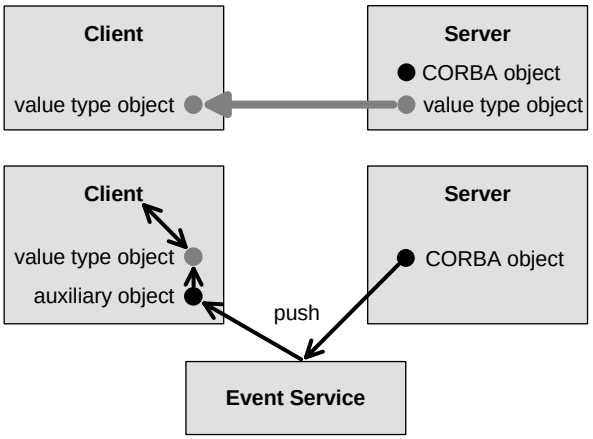

Figure 7. Push model with an auxiliary object

it is impossible that a CORBA object notifies a value type object of state changes directly. That is left to the auxiliary CORBA object. It receives state changes of the original CORBA object via push model and forwards these state changes to the value type object. This results in a complex implementation on the client side. The advantage is that no unnecessary state requests take place.

On server side the CORBA object must act as event supplier (`PushSupplier` or `PullSupplier`), so that client applications can be notified of state changes via the event service. Here, a developer has different possibilities. In the following, we describe a solution that is based on the `Counter` example (see Section 3). We derive an `EventCounter` interface from `Counter` and `PushSupplier` (see Figure 8). In this way, `EventCounter` acts as application object and event supplier at the same time.

The `CachedCounter` value type has a reference to a `ProxyPullSupplier`. Consequently, the client is able to ask the event supplier via pull model of the event service if state has changed.

6. Performance Measurements

For our performance measurements we used *VisiBroker Event Service* version 3.2 and *JavaORB* version 2.2.6. The *VisiBroker Event Service* is implemented in C++ [1]. *JavaORB* is a CORBA 2.3 compliant open source implementation in Java provided by the *Distributed Object Group* (D.O.G.) [3]. It is free for both commercial and non-commercial projects. Our test environment consisted of three computers with a 10 Mbit/sec Ethernet connection to the Internet:

- Workstation SunUltra-1 with Solaris 2.6 and Java Development Kit (JDK) 1.2.2 (the server application ran here)
- PC with Intel Pentium III processor, Linux 2.2.10, and JDK 1.1.8 (the different clients were tested here)
- Sun SPARCstation 10 with Solaris 2.5 (the event service was hosted here)

```
#include "CosEventComm.idl"
#include "CosEventChannelAdmin.idl"

module Caching
{
  interface Counter
  {
    readonly attribute long value;
    void inc();
    void dec();
  };

  interface EventCounter : Counter,
    CosEventComm::PushSupplier
  { };

  valuetype CachedCounter
    supports Counter
  {
    public CosEventChannelAdmin::
      ProxyPullSupplier realCounter;
  };

  interface CacheFactory
  {
    CachedCounter create();
  };
};
```

Figure 8. Event service based IDL definitions

Due to this heterogeneous arrangement with different computer architectures, operating systems, programming languages, and ORB products, CORBA's ability to interoperate was also checked successfully.

The following four scenarios were tested:

1. Usual access to a remote CORBA object (see Figure 3)
2. Simple pull model (see Figure 4)
3. Pull model with an auxiliary object (see Figure 6)
4. Push model with an auxiliary object (see Figure 7)

In each test run, the client sent 100,000 requests to the server (one every 50 milliseconds). The state of the CORBA object on the server was automatically changed by a thread every 100 milliseconds. Table 1 presents the test results.

Scenario	Average access time
1.	7,32114 ms
2.	0,00053 ms
3.	0,00070 ms
4.	0,00092 ms

Table 1. Test results

In the scenarios 3 and 4 the event service is used for communication between the value type object and the original CORBA object on the server. On the other hand, in scenario 2 this communication is direct. It is not surprising that the average access time in scenario 2 is lower than in scenarios 3 and 4. One reason is that in scenarios 3 and 4 an additional computer is needed for the event service. Another reason is that both approaches use different data types. In case of the direct communication (scenario 2) the value type object "knows" the original CORBA object. Therefore, it can use its type directly. In contrast, the event service uses `any`. Because marshalling and unmarshalling of type `any` takes more time, access times are worse. Despite this, the event service is very flexible and useful for a wide range of applications.

7. Summary

Each of the approaches mentioned above is associated with certain advantages and disadvantages. The usual way of direct access to a remote CORBA object is easy to implement but has the worst performance.

The use of a value type object without an event service offers the best performance. But the main drawback of this approach is the direct connection to the original CORBA object on the server. In case the server restarts or the CORBA object migrates to another computer, the connection is lost because the IOR becomes invalid. With the help of the event service this problem can be solved.

There are two event models (push and pull) that can be used together with the event service. From the programmer's point of view the pull model has the advantage that the polling mechanism can be contained completely within the value type object, there is no need for an auxiliary CORBA object. Thus, changes of the client application can be minimized. A drawback of this approach is that the value type object becomes relative complex. Therefore, transfer time of the object increases. Introducing an additional CORBA object that controls the polling mechanism can solve this problem. Conversely, the maintenance costs for the client application grow. In general, a difficulty of pull models is the determination of the optimal polling frequency. If it is too low clients realize changes too late. If it is too high net resources are wasted.

In push models clients are notified of changes at the right time automatically. Here, the main problem is that a CORBA object on the server cannot communicate with a value type object on the client directly. For that reason an auxiliary CORBA object is needed. This auxiliary object receives changes via push model and forwards them to the value type object. Because of that, the implementation of this approach is also relatively complex.

8. Related Work

Despite its importance, there is little related work on value types in general, and caching techniques implemented on basis of value types in particular. Mowbray's approach [5] includes an object caching technique that intercepts any remote invocation within the client if the object is locally available. Linnhoff-Popien [4] analyses caching and polling techniques in case of asynchronous communication. Wagner and Tari [10] develop a caching protocol. Their work addresses the problem of including object based caching aspects into CORBA applications with minimal impact to existing sources and permits inter-transaction caching. But all of those works do not consider value types. This paper fills this gap.

9. Conclusions

Besides better performance, value type objects offer several advantages.

- A value type object is able to support the same interface as the underlying CORBA object. This is expressed by the IDL keyword `supports`. If the interface of the underlying CORBA object is changed the value type object changes as well.
- After creation, value type objects are transferred including state. Thus, they are ready to use immediately and need no further initialization.
- Value type objects are a standardized concept that works well with all CORBA 2.3 compliant ORBs.
- Implemented with value type objects and based on the event service, a caching mechanism is very portable.

Value type objects have the minor drawback that one has to define factory objects for them [8]. If ORBs with different language mappings are used a factory object is needed for each language.

In summary, value types are very useful, especially to improve performance of CORBA applications with caching techniques.

References

[1] Borland. Visibroker. http://www.borland.com/visibroker/.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns – Pattern-Oriented Software Architecture*. John Wiley & Sons, Chichester, 1996.

[3] Distributed Object Group (D.O.G.). JavaORB. http://www.multimania.com/dogweb/index_javaorb.html.

[4] C. Linnhoff-Popien. *CORBA – Kommunikation und Management*. Springer-Verlag, Berlin, 1998.

[5] T. Mowbray and R. Malveau. *CORBA Design Patterns*. John Wiley & Sons, Chichester, 1997.

[6] OMG. CORBAservices: Common Object Services Specification, Nov. 1997. OMG Technical Document Number 98-07-05, ftp://ftp.omg.org/pub/docs/formal/98-07-05.pdf.

[7] OMG. Event Service Specification, 2000. OMG Technical Document Number 01-03-01, ftp://ftp.omg.org/pub/docs/formal/01-03-01.pdf.

[8] OMG. The Common Object Request Broker: Architecture and specification, editorial revision, CORBA 2.4.2, Feb. 2001. OMG Technical Document Number 01-02-01, ftp://ftp.omg.org/pub/docs/formal/01-02-01.pdf.

[9] M. Schader, M. Aleksy, and C. Tapper. Interoperabilität verschiedener Object Request Broker nach CORBA 2.0-Standard. *OBJEKTspektrum*, 3:72–77, 1998. http://www.wifo.uni-mannheim.de/IIOP/.

[10] S. Wagner and Z. Tari. A caching protocol to improve CORBA performance. In *Proceedings of the Australian Database Conference (ADC'00)* Canberra, Jan. 2000. http://goanna.cs.rmit.edu.au/~zahirt/Papers/adc00.pdf.