Despite its original flexibility and applicability to various environments, CORBA evolves to remain viable as a standard for distributed object-oriented applications.

New Features for CORBA 3.0

~Steve Vinoski~

Fundamentally, CORBA is an application integration technology. Since its inception in 1991, CORBA has provided abstractions for distributed object-oriented programming that have allowed developers to seamlessly integrate diverse applications into heterogeneous distributed systems [2, 6]. The core of any CORBA-based system, the Object Request Broker (ORB), hides the low-level details of platform-specific networking interfaces, allowing developers to focus on solving the problems specific to their application domains rather than having to build their own distributed computing infrastructures.

CORBA, like any technology or standard, has had to continually evolve since its original publication in order to remain viable as a basis for distributed applications. For example, the initial version of CORBA did not specify a standard protocol that applications could use to communicate with each other. For early CORBA users, the fact that each ORB used its own proprietary communication protocols did not present a problem. This was because most CORBA applications using version 1.0 were small enough that each could rely on the services of only a single ORB. However, as applications became larger and more distributed, they eventually needed to interact with other applications built using a different ORB. As a result, the standard General Inter-ORB Protocol (GIOP) and its specification for implementation over TCP/IP, the Internet Inter-ORB Protocol (IIOP), were added to version 2.0 of the CORBA specification, which was published in 1995. Such enhancements and modifications of the CORBA specification over the years have increased its usefulness and applicability to an ever-growing variety of distributed computing problems.

As part of its continuing evolution, several signif-

icant new features are being added to CORBA as it approaches version 3.0: the Portable Object Adapter; CORBA Messaging; and Objects By Value. An overview of each of these new CORBA features is provided here.

Portable Object Adapter

In CORBA, object adapters mediate between the world of CORBA objects and the world of programming language implementations, which are called servants. Object adapters provide the following services:

- Creation of CORBA objects and their object references.
- Demultiplexing of requests made on each target CORBA object.
- Dispatching requests to the appropriate servant that incarnates, or provides an implementation for, the target CORBA object.
- Activation and deactivation of CORBA objects.

CORBA object adapters are real-world instances of the Adapter design pattern [1] because they adapt the interfaces of servants to the interfaces offered by CORBA objects.

From the first version of CORBA through CORBA 2.1, the only standard object adapter defined by the OMG was called the Basic Object Adapter (BOA). As its name implied, it provided basic services to allow a wide variety of CORBA objects to be created and implemented. Unfortunately, as ORB vendors built their BOA implementations and developers used them to build applications, many missing features and much ambiguity were discovered in the BOA specification. Vendors then compensated for these problems by developing their own proprietary extensions, resulting in poor application portability between ORB implementations.

The new standard object adapter defined in the CORBA 2.2 specification is called the Portable Object Adapter (POA). It provides features that allow applications and their programming language servants to be portable between ORBs supplied by different vendors. The POA was developed jointly by a collection of ORB vendors and users based on their experiences with the BOA and with other proprietary object adapters.

The following sections describe the basics of the POA and each of its major features.

POA basics. A POA essentially mediates between the ORB and the server application. Figure 1 shows a request flowing from a client into a server application. First, the client invokes the request using an object reference that refers to the target object. The request is then received by the server ORB. Part of the request contains an identifier kept in the object reference called an object key, which uniquely identifies the target object within the server application. Because an application can have multiple POAs, the object key helps the ORB dispatch the request to the POA that hosts the target object. The POA then uses a portion of the object key called the object ID to determine an association between the target object and a programming language servant. It may either store these associations in a map, or it may invoke the application to ask it to provide a servant for the target object ID, or it might use a default servant nominated by the application. In any case, the POA dispatches the request to the servant, which then carries out the request and delivers the results back to the POA, to the ORB, and finally to the client.

As this description implies, the POA deals mainly with three entities. Two of them, the object reference

and the object ID, are used for identification purposes, while the third, the servant, actually implements CORBA objects.

- A server application first asks the POA to create a new CORBA object, and the POA returns an object reference as a result. The object reference, which uniquely identifies its object across all server applications, allows other applications to invoke requests on the new object.
- When creating a CORBA object, either the application or the POA provides an object ID to uniquely identify that object within the scope of the POA. This is described in more detail later in this article.
- A servant is said to incarnate, or (literally) provide a body for, a CORBA object. Ultimately, a request made on a CORBA object is carried out by its servant. In both C++ and Java, servants are instances of programming language classes.

Essentially, the POA translates between object references, object IDs, and servants in order to dispatch requests from the world of CORBA objects to the world of programming language servants.

Persistent and transient objects. One feature of CORBA that originally helped set it apart from other distributed application development platforms is its provision for the transparent and automatic activation of objects. If a client application issues a request to a target object that is currently not running or activated, CORBA requires ORB implementations to activate a server process for the object if necessary, and then activate the object itself. Any activation of server processes and target objects must be transparent to the requesting client. CORBA objects that can live beyond any particular process in which they are created or activated are called *persistent* objects. These objects are so named because they persist across the lifetimes of multiple server processes.

Despite the utility of persistent objects, CORBA applications written prior to the adoption of the POA also required another type of object with a shorter lifetime. It is often valuable to create objects whose lifetimes are bounded by that of the process or even of the object adapter in which they are created. For example, one application might send a reference to one of its objects to another application with the intent of having the second application eventually call it back. However, if the first application exits, it may no longer want the callback information. In that case, it does not want the callback to be delivered, so it does not want the ORB to reactivate either it or the callback object.

The POA supports two types of CORBA objects: the persistent object originally specified by CORBA, and a new shorter-lived object called a *transient* object. The lifetime of a transient object is bounded by the lifetime of the POA in which it is created. Thus, transient objects are useful in situations requiring temporary objects, such as the callback scenario described previously.

One additional benefit of transient objects is that they require less bookkeeping by the ORB. Once its creating POA has been destroyed, a transient CORBA object cannot be reactivated, which means that the ORB does not need to keep track of how to locate the object if it is not active when a request is made on it, nor how to activate its POA within a new server process. This in turn typically means less overhead in administering the CORBA application itself.

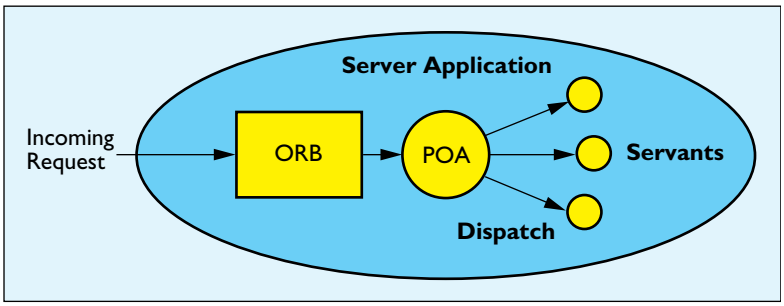

Figure 1. POA-based request dispatching

Explicit and on-demand activation. An application with just a few CORBA objects may want to create servants for those objects and register them with the POA before it starts listening for requests. For example, a CORBA application embedded in a sensor might contain only a single CORBA object representing the sensor itself. Such an application will most likely explicitly register a servant for the sensor object, thus explicitly activating that object. The POA provides operations that support this style of explicit object activation and servant registration.

At the other end of the spectrum, applications with many CORBA objects may only want to activate those objects that actually receive requests. For applications with many thousands of objects, this approach greatly helps minimize resource usage, because only those objects that are actually the targets of requests are activated. For example, an appli-

cation providing access to a large database may create a CORBA object for each database entry. Rather than creating a servant for each database entry every time it starts up, the application may instead use a *servant manager* to create servants on demand.

Servant managers are local CORBA objects that are upcalled by a POA if it receives an invocation on an object that has no associated servant. The servant manager can either provide the POA with a newly-created servant, or it can reuse an existing one. Either way, it returns the servant as the result of the upcall, and the POA uses that servant to complete the object invocation. Once the invocation completes, the POA either retains the association of the servant and the object ID of the target CORBA object in its *Active Object Map*, or it loses the association, meaning that the next invocation on the object will again require the services of the servant manager.

Applications can also supply a *default servant* to a POA. A default servant incarnates all CORBA objects for a POA, avoiding the need to create a separate servant for each object as well as the invocation overhead associated with servant manager upcalls. Default servants can be useful when all CORBA objects in a given POA support the same IDL interface type.

Separation of servant and CORBA object life cycles. The POA specification allows a single servant to incarnate one or more CORBA objects during its lifetime. For example, a default servant normally incarnates all the CORBA objects for a given POA. Within the context of each CORBA request, the default servant incarnates the object that is the target of that request.

The POA specification also allows a single CORBA object to be incarnated by one or more servants during its lifetime. For example, if a POA does not maintain its associations between servants and objects in its Active Object Map, it may rely on a servant manager to supply servants for each CORBA request. If the servant manager creates a new servant every time it is invoked, each invocation on a given object will be carried out by a different servant.

The separation of servant and CORBA object life cycles is necessary for scalability. If a servant could only incarnate a single object, server applications hosting many thousands of objects would be difficult to deploy due to the memory resources such applications would require. Furthermore, if a CORBA object lived only as long as the servant that incarnated it, it would be impossible to support persistent objects, which outlive any single server process.

Many CORBA application developers are confused by the fact that CORBA objects and servants are separate entities with their own distinct lifetimes. Part of this confusion is due to the way many contemporary ORBs implement object references for objects that are located in the same address space as a client. It is common practice with BOA implementations to use direct programming language pointers or references to servants as object references for collocated objects. Such an implementation of collocated object references allows local invocations to cost the same as native programming language function calls. This approach is useful for certain types of applications that make heavy use of collocated objects, but for most applications, it has the downside of making local invocations act differently from those made on remote objects.

Different policies for multithreading. A major drawback to the BOA specification was that it did not address any issues related to multithreaded applications. It is common for CORBA server applications to use multiple threads in order to easily service multiple concurrent requests [5]. An application might wish to service each new request in a separate newly-created thread, handle all requests for a given object in a separate thread, or employ a fixed-size pool of threads to handle all requests, queuing requests if all threads in the pool are busy. The appropriate thread usage strategy for an application depends on a number of factors, including the number of objects hosted by the application, the expected request rate, and the multithreading support provided by the underlying operating system.

An application can create a POA with one of two different threading models. The *ORB-controlled model* allows the underlying ORB implementation to choose an appropriate multithreading model, while the *single-thread model* guarantees that all requests for all objects in that POA will be serviced on a single thread.

The ORB-controlled model allows multiple requests to be processed concurrently by multiple threads. Applications using POAs created for this model must be implemented to properly handle reentrant invocations and concurrency, since servants registered with such a POA may be required to handle multiple CORBA requests simultaneously.

Applications using single-thread model POAs, however, need not be thread-aware. In fact, using a single-threaded POA means that all requests for objects in that POA will be processed sequentially. This feature can be used to advantage when integrating existing code not designed for use in a multithreaded environment.

While these POA multithreading models are a vast improvement over the lack of multithreading specified for the BOA, they could be made even more flexible. For example, rather than merely supplying

the ORB-controlled model, the POA could provide for finer-grained control over multithreading policies by allowing applications to specify precise models such as a thread pool model, a thread-per-request model, or a thread-per-object model. Future standard extensions to the POA specification may indeed supply a variety of applications with this much-needed flexibility.

Application control over object existence. In order to be useful for the widest possible variety of applications, POAs maintain no persistent state. If a POA were required to keep track of its objects between different executions of a server application, it would require persistent storage. This requirement would greatly hamper deployment of POA-based applications in several ways. For example, it might require ORB vendors to either supply or require certain databases for use with their ORB products, and those databases might not integrate well with other databases already employed by the end user. Alternatively, the database chosen for use by the ORB vendor may not scale appropriately for the needs of certain applications—for example, it is highly impractical to deploy a large-scale relational database on an embedded industrial control sensor.

Because the POA maintains no persistent state, it is the responsibility of the application to determine whether a given CORBA object still exists or not. For example, if a servant manager receives an upcall to incarnate a servant for a given object, and the servant manager determines that the object in question no longer exists, it raises a CORBA standard `OBJECT_NOT_EXIST` exception.

When a request is dispatched to a servant, or a servant manager is invoked to incarnate a servant, the application can determine the identity of the target object by its object ID. An object ID is a sequence of bytes that is associated with a CORBA object when it is created, and it can either be supplied by the application or provided by the POA. For persistent objects, the application usually supplies an object ID that somehow indicates the location of the persistent storage for the object. For example, if each entry in a database is represented as a separate CORBA object, an application might use database keys for object IDs. When a request is made on such an object, the application can use its object ID—its database key—to see if the associated entry in the database still exists. If using the key to access the database fails because the entry is no longer present, the application can authoritatively report that the target object no longer exists.

This aspect of the POA design acknowledges the fact that, in one way or another, the application must ultimately be responsible for tracking object identity and existence. Solutions based on the desire to relieve the application of this responsibility and instead give it to the object adapter are typically brittle and inflexible because the object adapter simply does not have enough knowledge about the application context to make object existence decisions.

> THE PORTABLE OBJECT ADAPTER, CORBA MESSAGING, AND OBJECTS BY VALUE ALLOW CORBA TO BE EMPLOYED FOR APPLICATIONS FOR WHICH IT WAS NEVER BEFORE SUITED.

Static and dynamic skeleton interfaces. Many server applications are compiled with complete knowledge of the interfaces of the objects they support. The IDL interface descriptions for each object type are compiled to produce programming language *skeleton* code that is then compiled and linked into the server application. Because each skeleton contains specific knowledge about all the operations supported by the interface it represents, including the number and types of arguments, they serve to assist the object adapter in dispatching each request to the proper servant. This approach, where statically-typed skeletons are compiled into the application, is used for most CORBA server applications.

There are some server applications, however, that must be able to determine object interface types at run time. For example, consider a gateway application that makes objects from another distributed system appear as CORBA objects. Such an application might allow CORBA clients to access objects implemented using Microsoft COM. When objects with a new COM interface type are added to the system, the gateway should immediately allow access to them without needing to be shut down and recompiled with new static skeletons.

The POA supports the *Dynamic Skeleton Interface* (DSI), which allows servants to determine at run time the details of the interface and operation

invoked on the target object. A servant implementing the DSI often makes use of the Interface Repository, a service that allows run-time access to IDL definitions, in order to determine the types of the arguments and the return value for the invoked operation.

Multiple POAs per application. All of the degrees of freedom discussed in the previous sections are controlled by applying different *policies* to a POA when it is created. For example, the value of a POA's `LifespanPolicy` determines whether the objects it creates are transient or persistent. The value of a POA's `ServantRetentionPolicy` controls whether it retains associations between object IDs and servants in an Active Object Map, or whether it requires the application to supply a servant manager to provide such associations. Other policies control whether the application or the POA supplies object IDs for new objects, whether a single servant can incarnate multiple CORBA objects or not, and whether the POA uses the ORB-controlled threading model or the single-thread model. Once a POA has been created, its policies cannot be changed.

To enable a single application to utilize different sets of POA policies, it is possible for an application to create and use multiple POAs. For example, an application wishing to support persistent CORBA objects must create a POA with the `PERSISTENT` policy. New POAs are created by invoking an operation on an already-existing POA. The ORB supplies a default POA called the *Root POA*, which applications can use to create additional POAs. For portability, the set of policies supported by the Root POA are standardized.

POA summary. The main goal of the POA is to provide portability for CORBA server applications, something that the BOA specification was sorely lacking. It achieves this goal while providing a great deal of flexibility for server implementations. The variety of servant implementation types, servant managers, and object/servant associations made possible by the POA specification can be used to fulfill the requirements of many different types of CORBA server applications.

CORBA Messaging

For years, the area of asynchronous messaging has been deemed by many as an obvious hole in the CORBA specification. Asynchronous messaging is widely viewed as a highly scalable solution allowing for the deployment of practical large-scale distributed systems. This is mainly because asynchronous messaging assumes that all systems that need to communicate with one another will not be reliably connected to each other at all times. Disconnections may occur because of normal occasionally connected and mobile systems such as laptops, palmtop computers, and other hand-held devices, or they may occur because of problems such as network partitioning and system failure. These disconnections are seen as being inevitable in real-world computer networks, especially as the size of the distributed system grows.

Because of its heavy reliance on synchronous invocation techniques, which tend to tightly couple clients and servers, CORBA has traditionally been criticized as being unable to properly cope with large distributed systems.

The CORBA Messaging Specification [3] is one of the most solid specifications ever considered for adoption by the OMG. It adds asynchronous messaging, time-independent invocation, and facilities for specifying messaging Quality of Service (QoS) to CORBA in a way that meshes well with existing CORBA features.

Asynchronous messaging. The first versions of CORBA provided three different models for request invocation:

- Synchronous—The client invokes an operation, then blocks waiting for the response.
- Deferred Synchronous—The client invokes an operation, then continues processing. It can later go back and either poll or block waiting for the response.
- Oneway—The client invokes an operation, and the ORB provides only a best-effort guarantee that the request will actually be delivered. There is no response.

The Deferred Synchronous request model was only available for clients using the Dynamic Invocation Interface (DII) rather than static invocation techniques. The Oneway model, which is often erroneously thought of as an asynchronous request model, was originally intended to allow CORBA requests to be delivered using unreliable transports such as the Unix Datagram Protocol (UDP).

The CORBA Messaging Specification preserves these three request models and adds two asynchronous request models:

- Callback—The client supplies an additional object reference parameter with each request invocation, and when the response arrives the ORB uses that object reference to deliver the response back to the application.
- Polling—The client invokes an operation, which

immediately returns a valuetype that can be used to separately poll or wait for the response.

Unlike the traditional deferred synchronous request model, the Callback and Polling asynchronous models are available for clients using statically typed stubs generated from IDL interfaces. This is a significant advantage for most programming languages because static invocations provide a much more natural programming model than the DII.

Time-independent invocation. To allow invocations on objects that are not active or are currently disconnected, the CORBA Messaging Specification augments the standard GIOP and IIOP ORB interoperability protocols with the ability to deliver requests to intermediate routing agents. These agents typically provide store-and-forward capabilities that can keep messages alive when the target is not currently available or active. Replies may also be handled by routing agents if the requesting client becomes unavailable or disconnected. The overall lifetime of the request can be controlled by specifying the appropriate QoS policies for the request.

Quality of service. CORBA traditionally has hidden certain issues from client applications, specifically those related to locating objects, establishing network connections, and delivering requests. Originally, this was done to preserve maximal implementation freedom for ORB vendors and to hide low-level networking details from clients. Unfortunately, hiding these details makes it difficult for CORBA applications to control the qualities of the message delivery services provided by the underlying ORB.

The CORBA Messaging Specification corrects this deficiency by allowing clients to specify the QoS they require for message delivery, message queuing, and message priorities. QoS policies can be specified at the ORB level to affect all requests, at the thread level to affect only those requests made from that thread, and at the object reference level to affect only those requests made on the referred-to object.

Objects by Value

As with asynchronous messaging, the lack of support for passing objects by value has long been seen as a serious hole in the CORBA specification. CORBA has always allowed data types such as long integers, structs, and arrays to be passed as arguments to operations. To allow objects to be able to invoke operations on other objects, CORBA also allows references to objects to be passed as arguments. Passing a reference to a remote application means that the object stays put, and thus any invocations of the object performed by the remote application cause the requests to move over the network to the object, rather than the object itself moving to the site of each request.

Valuetype basics. To address this shortcoming, support for passing objects by value was recently added to CORBA [4]. The adopted approach involves the addition of a new construct to OMG IDL called the *valuetype*. A valuetype is essentially a cross between a struct and an interface because it supports both data members and operations, much the same as C++ and Java class definitions. Like interfaces, valuetypes can derive from other valuetypes (single inheritance only), and they can also support interface types in order to inherit their operations.

```
//IDL
valuetype Node {
  init(in Object ref, in Node next);     // constructor
  Node next();                           // access next Node
  void set_next(in Node next);           // set next Node

  public Object ref_data;   // object reference data member
  private Node next_node;   // pointer to next node in list
};
```

Figure 2. Valuetype definition for a singly-linked list of object references

When a valuetype is passed as an argument to a remote operation, it is created as a copy in the receiving address space. The identity of the valuetype copy is totally separate from the original, and operations on one have no effect on the other. Because of the copy semantics, all operations invoked on valuetypes are always local to the process in which the valuetype exists—that is, unlike operation invocations on CORBA objects, valuetype invocations never involve the transmission of requests and replies over the network.

A valuetype definition for a singly-linked list of object references might appear as shown in Figure 2. The IDL specification shown in the figure defines a new valuetype named `Node` that contains an initialization function, two operations, and two data members. As its name implies, the initialization function, introduced by the `init` keyword, provides a hook for the application to construct instances of the val-

uetype. Initializers are essentially equivalent to constructors in both C++ and Java. The `next` operation takes no arguments and returns the next `Node` in the linked list. The `set_next` operation allows the next `Node` to be set. The first data member, `ref_data`, contains the object reference held by each linked list node, while the `next_node` member refers to the next node in the linked list. The `ref_data` member is declared `public` and so can be accessed directly, while the `next_node` member is declared `private` to prevent all but the operations on `Node` from accessing it.

For C++ applications, an IDL compiler translates the `Node` type to the C++ class shown in Figure 3. The application developer is expected to derive from the generated C++ class shown in Figure 3 in order to support concrete instances of the `Node` valuetype. Note that this class is abstract due to the pure virtual `next` and `set_next` member functions. Because the IDL compiler cannot possibly know the semantics their implementations should have, the `next` and `set_next` member functions are expected to be overridden and implemented by derived classes. Also note that the IDL data members `ref_data` and `next_node` are respectively mapped to public and protected virtual non-pure pairs of accessor and modifier member functions. The `next_node` function is protected to allow derived classes to access it without making it public, and accessor/modifier pairs are used for the mapping instead of actual data members to allow derived classes the choice of how to best represent the data members.

Of particular importance is the fact that all uses of the `Node` type as argument types and return types in C++ map to C++ pointers. Pointers are used for two reasons:

- The address of a C++ valuetype instance is effectively its identity, which allows acyclic graphs of valuetype instances to be properly marshaled and unmarshaled. If two nodes in a graph point to the same node, the shared node is marshaled and unmarshaled only once, allowing the graph to be correctly replicated in the receiving process.
- It must be possible for a reference to a valuetype instance to be null, that is, to refer to no instance. Since C++ references are not allowed to be null, but C++ pointers are, pointers must be used as the mapping for valuetype references.

State and behavior. Passing the state (the data members) of valuetypes as operation arguments to remote applications is much like passing other data types, but most valuetypes also possess behavior in the form of their own operations. Unlike state, behavior is not so easily passed over the network. Given the heterogeneous nature of most networks in which CORBA is used, a client may very well be written in Java or Smalltalk while a server may be written in C++, and the client and server may execute on different computer architectures as well. In this case, passing the implementation of the valuetype operations from client to server is simply not possible.

```
//C++
class Node: public virtual CORBA::Valuebase {
public:
   virtual Node* next() = 0;
   virtual void set_next(Node* next) = 0;

   virtual CORBA::Object_ptr ref_data() const;
   virtual void ref_data(CORBA::Object_ptr ref);

protected:
   Node();
   Node(CORBA::Object_ptr ref, Node* next);
   virtual ~Node();

   virtual Node* next_node() const;
   virtual void next_node(Node* n);
};
```

Figure 3. A C++ class generated from IDL Node valuetype definition

For Java and Smalltalk applications, enough information can be marshaled along with each valuetype to allow the receiving application to download the bytecodes for the valuetype operation implementations. For languages like C++ and C, the receiving application has to either already have compile-time knowledge of the valuetype, or be able to somehow acquire it, perhaps in the form of a dynamically loadable library (DLL). In theory this may seem somewhat restrictive, but in practice it actually works very well. This is because within a security domain where clients and servers are developed, maintained, and administered together, compiled languages like C++ and C can be used for implementations, and the necessary valuetype implementations can be supplied as described previously. Where clients and servers exist in different administrative domains, such as when the client exists in a Web browser and accesses servers over the Internet, Java can be used for the

client and server implementations so that downloading valuetype implementations can be performed with relative ease and safety.

Objects by value summary. The addition of the valuetype construct to OMG IDL provides support for passing objects by value between CORBA applications. Valuetypes, which can be defined in IDL with both data members and operations, are best used to define lightweight encapsulated data types such as linked lists, graphs, and dates.

Unfortunately, the Objects By Value (OBV) Specification is not without drawbacks. It is overly complicated and at the same time lacks features. Due to the nature of this article, the complicated aspects of the specification have not been described here, but they include various notions of valuetype and interface inheritance, special valuetypes called *boxed valuetypes*, which are essentially valuetypes with only a single data member and no operations, and a new special form of interface called an *abstract interface*. (The latter is an especially strange construct, given that normal OMG IDL interfaces have always been abstract, ever since the very first version of CORBA.) The combination of all of these new features with each other and with existing CORBA features was unfortunately not thought out very well. Therefore, OBV features are more difficult to use than they need to be. Furthermore, there is a high likelihood that several serious unknown problems still exist in the OBV Specification, having not yet been stumbled upon by ORB developers or users.

One obvious feature missing from the OBV Specification is the ability to pass true CORBA objects by value. The OBV Specification allows valuetypes to be derived from OMG IDL interfaces, but just as for normal CORBA objects, only their object references can be passed as arguments if they are used in a context requiring an object. Valuetypes are therefore not really CORBA objects, even if they are derived from interfaces, but are instead a new OMG IDL type. Passing true CORBA objects by value is a difficult problem due to required interactions with object adapters, issues surrounding locating objects in motion, issues regarding whether objects are copied or moved, issues related to object identity and its preservation, and issues concerning the movement of objects between ORBs from different vendors. It is therefore not surprising that the OBV Specification submitters did not attempt to tackle this problem, but it effectively means that CORBA still lacks a real OBV solution.

Advances in CORBA have traditionally been conservative; they have been largely based on well-known and proven technologies and techniques. For example, the first version of CORBA was based largely on existing Remote Procedure Call (RPC) technology that had already been deployed in production distributed systems. Unfortunately, because several unproven and unimplemented techniques serve as its foundation, the OBV Specification has broken with this tradition. The OMG is currently reviewing its technology adoption procedures with the goal of preventing similar hasty and ill-advised additions to CORBA in the future.

Summary

Since its inception in 1991, CORBA has proven itself as a solid basis for heterogeneous object-oriented distributed systems. Like all technologies, however, CORBA must evolve in order to remain viable. The addition of three significant new CORBA features—the Portable Object Adapter, CORBA Messaging, and Objects By Value—allow CORBA to be employed for applications for which it was never before suited. These new features thus serve to further increase the already wide variety of applications and technologies that can be practically integrated under the CORBA umbrella. C

References

1. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.
2. Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, Framingham, MA, 1998.
3. Object Management Group. *CORBA Messaging Joint Revised Submission*. Object Management Group, Framingham, MA 1998, document orbos/98-05-05.
4. Object Management Group. *Objects By Value.* Object Management Group, Framingham, MA, 1998, document orbos/98-01-18.
5. Schmidt, D. and Vinoski, S. Comparing alternative programming techniques for multithreaded servers. *C++ Report 8*, 2 (Feb. 1996).
6. Vinoski, S. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications 14*, 2 (Feb. 1997).

Steve Vinoski (vinoski@iona.com) is senior architect for IONA Technologies in Cambridge, Mass., makers of the industry-leading Orbix and OrbixWeb object request brokers.