Fundamentals of Distributed Object Systems: The CORBA Perspective
Zahir Tari, Omran Bukhres
Copyright © 2001 John Wiley & Sons, Inc.
Print ISBN 0-471-35198-9 Electronic ISBN 0-471-20064-6

PART II

ADVANCED CORBA

Fundamentals of Distributed Object Systems: The CORBA Perspective
Zahir Tari, Omran Bukhres
Copyright © 2001 John Wiley & Sons, Inc.
Print ISBN 0-471-35198-9 Electronic ISBN 0-471-20064-6

CHAPTER 4

Object Adaptors

OMG provided the specification of two adaptors, the *Basic Object Adaptor* (BOA) and the *Portable Object Adaptor* (POA). BOA was released in an earlier version of the OMG specification. Because of several problems related to this adaptor, such as portability and flexible activation policies, OMG recently released the POA specification.

Even though most of the existing CORBA systems are supporting POA or being updated to support it, we have decided to provide details of BOA (e.g., architecture and object/servant/server activation and deactivation). We believe that this will provide a clear idea about the BOA limitations and therefore make it easier to understand the OMG adoption of POA. This chapter also provides a comparison between BOA and POA with regard to the issues relating to object adaptors. The chapter starts with an overview of CORBA and its adaptors, and in Section 4.2, the architectures of BOA and POA are described. Section 4.3 provides an evaluation of both BOA and POA with regard to a set of criteria. Finally, in Section 4.4, we show how POA can be extended to construct database adaptors that enable making object references persistent.

4.1 OVERALL PICTURE

The rationale behind the inclusion of an object adaptor in the OMG architecture is interface flexibility and management. To communicate without an adaptor, both the ORB and the implementations must agree on one fixed set of interfaces or they must support multiple sets of interfaces. Providing a unique set of interfaces is not desirable because different sets of interfaces might be needed, depending on the application's aims. For example, an implementation might perform a radically new kind of service, include new functionalities, or better performance may be needed. The second alternative, that is, to support multiple sets of interfaces, is complex and could result in only a subset of the adaptor's interfaces being used. Confusion might also occur over which set of interfaces should be selected. It is conceded that foreseeing all the types of interface is impossible.

To be efficient, the interface of an adaptor would have to be tailored according to the interface of the ORB and the interfaces of the implementations. This does not mean that adaptors can be introduced sparingly; however, it happens only in special cases and OMG expects to have a small number of adaptors existing. The advantage of placing an object adaptor between an ORB and implementations is to make them independent of each other. In this way, an ORB relies on its adaptor's interface to communicate indirectly with the implementation objects and vice versa. An adaptor will be responsible for adapting the interfaces of its implementation objects for the ORB. It will also allow the implementation object to gain access to the ORB service via the adaptor's public interface.

The adaptor will be then dependent on the interface of its implementation objects and the interface of its ORB. Changes in the interfaces will affect the adaptor's interface.

In addition to the role of linking an ORB with object implementations, object adaptors have other functionalities, including generation and interpretation of object references, method invocation, security of interaction, activation and deactivation of object implementations, mapping object references to their implementation objects, and registration of implementation. In earlier versions of CORBA, BOA (*Basic Object Adaptor*) [70] was proposed as a basic adaptor that provides these functionalities. However, BOA's specification is plagued by problems in its portability, clarity and completeness. As a result, currently available BOAs are vendor specific and incompatible. An RFP (Request For Proposal) for ORB Portability was published to address these problems [71]. This document reveals that OMG had several choices: (i) improve the BOA specification to eliminate multiple interpretations; (ii) allow multiple versions of BOA to exist, each for a major operating environment (e.g., POSIX, Windows, Macintosh) and standardize each of those versions; (iii) publish a new "universal" object adaptor which is not necessarily derived from BOA; or (iv) abandon the BOA standardization effort. The first choice seems too hard, if not impossible. Because of too many incompatible versions of BOA, producing a specification that considers all of them seems like a futile attempt. The second will cause the proliferation of multiple versions of BOA. The next choice promises a clean break from the past but invalidates existing applications. The last one is a "hands-off" approach and should not be selected. Ultimately, OMG decided to start fresh by taking the third choice as their solution [74]. They agreed to let the vendors keep their BOAs, but no new BOA specification will be published. Applications that use BOAs will continue to function and be supported by the ORB vendors.

POA (*Portable Object Adaptor*) is the OMG's new adaptor. It is not an improved version of BOA and is specified in [78]. POA was designed from the ground-up with portability in mind. Server code benefits most from POA, since they can now operate across different CORBA products. POA interfaces are declared in their own PortableServer module. This gives more room and more flexibility for future expansions of POA. Like BOA, POA is designed to handle objects with conventional implementation.

4.2 ARCHITECTURES

When an invocation is made, the client-side ORB is responsible for interpreting the object's roles, for locating the server in which the object is implemented and for sending a request to that server. On the server side, the request is received by the ORB, where three steps of dispatching are necessary: (i) the ORB must find the object adaptor that the object is implemented in and pass the request on to that object adaptor; (2) the object adaptor must find the servant that implements the object; and (3) if the servant uses a static skeleton, the request is unpacked by the IDL-generated code and the desired method is invoked. Before any of this can happen, the object adaptor must first know about the servant. After registering the servant with the object adaptor, an implementation must be able to create and export object references that address the servant. So besides merely performing invocations, the object adaptor must provide an administrative interface as well. The object adaptor will be responsible for: (i) registration of implementations; (ii) mapping object references to the corresponding object implementations; (iii) activating and deactivating objects and implementations; (iv) generating and interpreting object references; (v) method invocation; and (vi) security of interactions.

Figure 4.1 shows the object adaptor in relation to the other server-side parts of an ORB. The object adaptor is trapped between the ORB core and the object implementation and provides three interfaces: one to the ORB, consisting of a single method to receive an incoming request; one to the user code, for forwarding requests, using either DSI or DII; and one interface through which an implementation can cause objects to be activated or deactivated and can influence the processing of requests.

The most simple object adaptor would be no more than a table that maps object keys (the server-side part of an object reference) to servants. Upon invocation, only

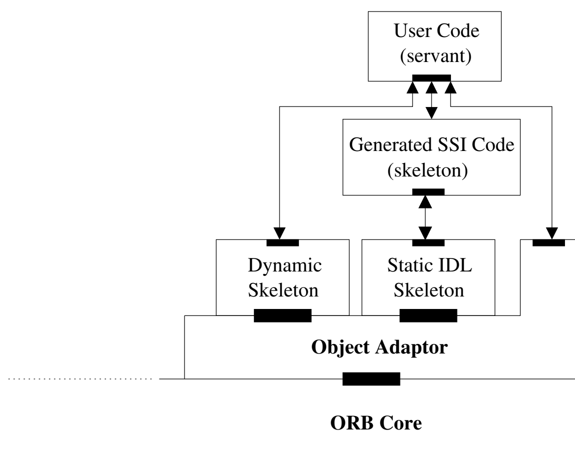

Figure 4.1 ORB server side.

a single table lookup would be necessary; activation and deactivation of servants would cause insertions and deletions into that table.

4.2.1 Basic Object Adaptor

The BOA's purpose was to be a simple and generic type of object adaptor that could be used, as the name suggests, for basic purposes. As such, its interface is intentionally minimalistic, providing several methods, conveniently specified in IDL as shown below:

```
module CORBA {
   // from Interface Repository interface
   interface interfaceDef;

   // from Implementation Repository
   interface ImplementationDef;

   interface Object;

   // an object for the authentication service
   pseudo interface Principal;

   typedef sequence<octet,1024> ReferenceData;

   ...

   pseudo interface BOA {

      // implementation activation and deactivation
      void impl_is_ready (in ImplementationDef impl);
      void deactivate_impl (in ImplementationDef impl);
      void obj_is_ready (in Object obj,
                         in ImplementationDef impl);
      void deactivate_obj (in Object obj);

      // generation and interpretation of object references
      Object create (in ReferenceData id,
                     in InterfaceDef intf,
                     in ImplementationDef impl);

      void dispose (in Object obj);
      ReferenceData get_id (in Object obj);
      void change_impelemtation (in Object obj,
                                 in ImplementationDef impl);

      // identification of the principal making a
      // request Principal
      get_principal (in Object obj, in environment ev);
```

```
    };
    ...
};
```

Figure 4.2 depicts a global picture of BOA. It shows the translation of a user's (IDL) operation to a set of BOA operations. The first step is to start a server process when the first request to its object arrives. The implementation registers as being ready to receive requests, by calling $BOA :: impl_is_ready()$. Next, BOA performs an up-call to the object activation routine. Later, BOA delivers the requests to the object using the appropriate skeleton method. The implementation might or might not use BOA's services, through its interface for operations such as object deactivation using $BOA :: deactivate_obj()$.

4.2.2 Portable Object Adaptor

POA is an object that is visible to the server. Object implementations are registered with the POA. As for BOA, the ORB, POA, and object implementations cooperate to determine on which servant the operation should be invoked, and to perform the invocation. As shown in Figure 4.3, there are several elements within the POA architecture that are involved in the process of activation and invocation: servant, object reference, POA, policy, POA manager, servant manager, and adaptor activator. In this section we will provide definitions of these elements of the POA architecture based on [79] and will explain how their roles fit within the overall picture of the POA functions.

- Basically, a servant class is a programming language object (e.g., C++ class) that implements operations on one or more objects (e.g., C++ objects). Servants generally exist with the context of a server process.
- A user has a reference to an object, called an object reference, and invokes an (IDL) operation on that object. The request is mediated by the ORB and transformed into invocations on a particular servant. An object reference in the POA model is the same as in the CORBA object model. This reference contains an

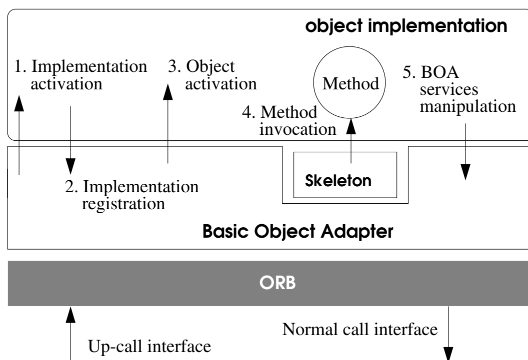

Figure 4.2 BOA operations in a shared server.

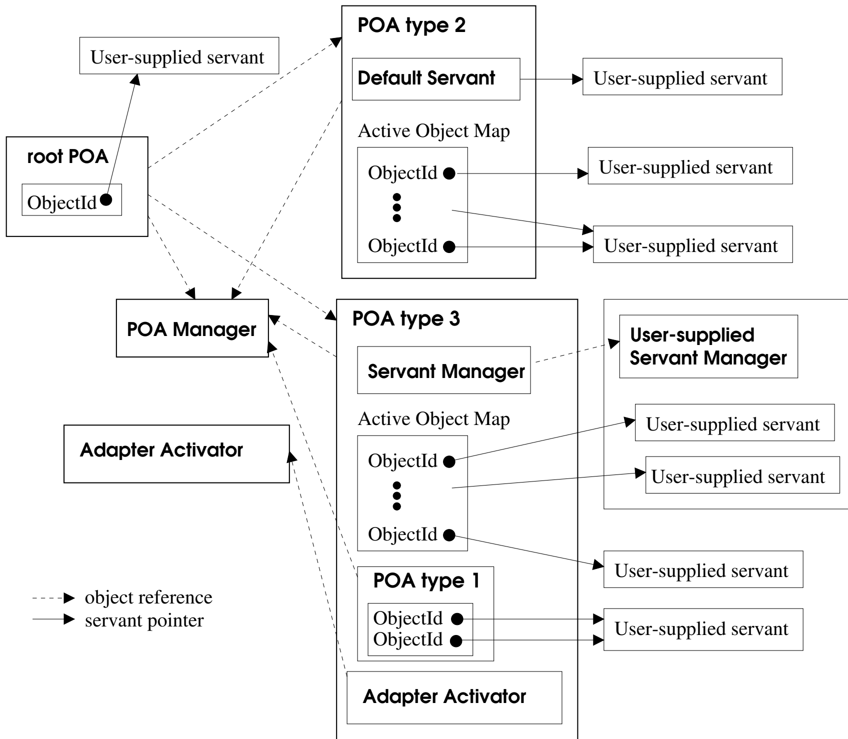

Figure 4.3 POA architecture.

object Id and a POA identity. In a few ORB implementations, object references contain additional information that helps increase efficiency of the binding process (e.g., location of the server).

- A POA is an identifiable entity within the context of a server. Each POA provides a name space for object ids and name space for other (nested or child) POAs. Policies, as shown in Figure 4.4, associated with a POA, describe the characteristics of the objects implemented within that POA. Nested POAs form a hierarchical name space for objects within a server.
- A Policy is an object, associated with a POA by an application, in order to specify a characteristic shared by the objects implemented under that POA. The specification defines policies controlling the POA's threading model as well as other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.
- The servant manager activates and deactivates servants when requested by the ORB (through specific operations). The servant manager is responsible for managing the association of an object (as characterized by its object id value) with

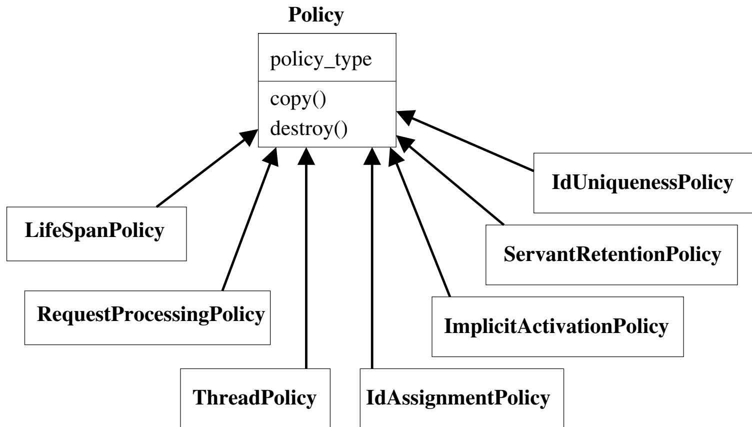

Figure 4.4 Portable server policies.

a particular servant, and for determining whether an object exists or not. There are two types of servant managers, namely *Servant Activator* and *Servant Locator*. The type used in a particular situation depends on policies in the POA.

- An adaptor activator is an object that the application developer can associate with a POA. The ORB invokes operations on an adaptor activator when a request is received for a child POA that does not currently exist. The adaptor activator can then create the required POA on demand.

Clients hold references upon which they can make requests. These references can be obtained by using CORBA services (e.g., Naming Service, Trading Service). When a request is invoked by using the information within the reference, as explained in Section 2.5 of Chapter 2, the first thing that the ORB does is to locate the appropriate POA object, which in turn has enough information to locate the responsible servant for the object for which the request is invoked. As we will see later on, POAs are identified by name within the namespace of their parent POA. The location POA starts from the root POA hierarchy (called `rootPOA`) and proceeds until the correct POA is found. The full path name is extracted from the reference to locate the position of the POA within the hierarchy.

There are several ways to locate a POA within a hierarchy, and obviously these are implementation considerations (and therefore OMG will not provide any detail). One option is that the request is delivered to `rootPOA`, which then scans the first part of the path name and then delegates the request to one of its child POAs. The request is handed down the line until the right POA is reached. This way follows a linear access to the POAs along the inheritance hierarchy, which is generally appropriate when the hierarchy is small. When there is a large number of adaptors, a better option is needed. For example, an access method (index or hash table) can be used to increase the performance of the lookup of the appropriate POA within the hierarchy.

If the lookup of the POA hierarchy fails, that is, the appropriate POA is not found, then the application (programmer) has the opportunity to create and register the required POA by using an adaptor activator. As shown in Figure 4.3, an adaptor activator is a user-implemented object that can be associated with an existing POA within the POA hierarchy. It is given an opportunity to create the required POA. If it cannot, the client receives an exception.

The creation of POA's children requires user intervention because programmers usually wish to assign a custom set of policies by writing the code that does this. An example of policy can be as follows: if a new POA is to be created but no adaptor activator has been registered with its parent, then the request will fail. Within the adaptor activator, the user cannot only create further child POAs, but also activate objects managed by these child POAs, possibly by reading state information from the disk.

Once an object's POA has been found, further processing depends on the POA's servant retention policy and its request processing policy; Figure 4.4 shows the different types of policies. The POA uses the oid of the object reference to locate the servant. Figure 4.3 shows the three possible ways to find the responsible servant, and these are based on the different policies: RETAIN, `use_default_servant`, and `use_servant_manager`. In the former, that is, POA has a servant retention policy of RETAIN, then POA performs a lookup in the *Active Object Map* to find if there is a servant associated with the oid value from the request. If found, the POA invokes the operation on the servant. This case relates to "POA type 1" in Figure 4.3. The lookup method can have a big impact on the POA's overall performance. As mentioned above, appropriate access methods are useful when large numbers of objects are registered.

If the servant retention policy is non-RETAIN, or has the RETAIN policy but no matching entry is found in the Active Object Map, the request processing policy is considered. If its value is a use default servant, the user can provide a single default servant that will be used regardless of the request's oid. The POA invokes the operation on the default servant associated to it. If a default servant is not found, then the POA raises an `OBJ_ADAPTOR` system exception. This case relates to "POA type 2" of Figure 4.3.

A default servant is useful when handling a group of usually identical objects through a single servant. A default servant can use the POA's context-sensitive introspection methods to query the oid of the current request and behave accordingly. Default servants provide scalability using the flyweight patterns: the server does not grow with the number of objects. Rather, the server can produce arbitrary numbers of object references while the number of active servants is constant. A database server is an example of the usage of a default servant. Each table would be represented by using a different POA and the key value is used as oid. In this way, all table rows are objects with their own object reference. Only a single default servant is needed per table; in an invocation, this default servant would query the request's oid and use it as a table index. By using the DSI, the default servant could even be identical for all database tables, examining the table structure to select its parameter's types.

The request processing policy can be set to use the servant manager, providing even more flexibility than in the previous policies. This case relates to "POA type 3" of Figure 4.3. Because the POA has a reference to a user-provided servant manager (see Figure 4.5), it delegates the search to this manager which will invoke specific operations (such as incarnate or preinvoke), depending on the type of the manager. There are two types of server managers, servant locator and servant activator, which can be used depending on the servant retention policy. If this policy's value is RETAIN, the servant activator will incarnate a new servant which will, after the invocation, be entered into the Active Object Map itself to reflect the new update. If the servant retention policy is non-RETAIN, the servant manager would have to be a servant locator; its task is to locate a servant suitable only for a single invocation. The servant locator supplements the default servant mechanism by providing a set of default servants. This is generally important for critical applications, when load balancing is required; the server locator selects the servants with the less load.

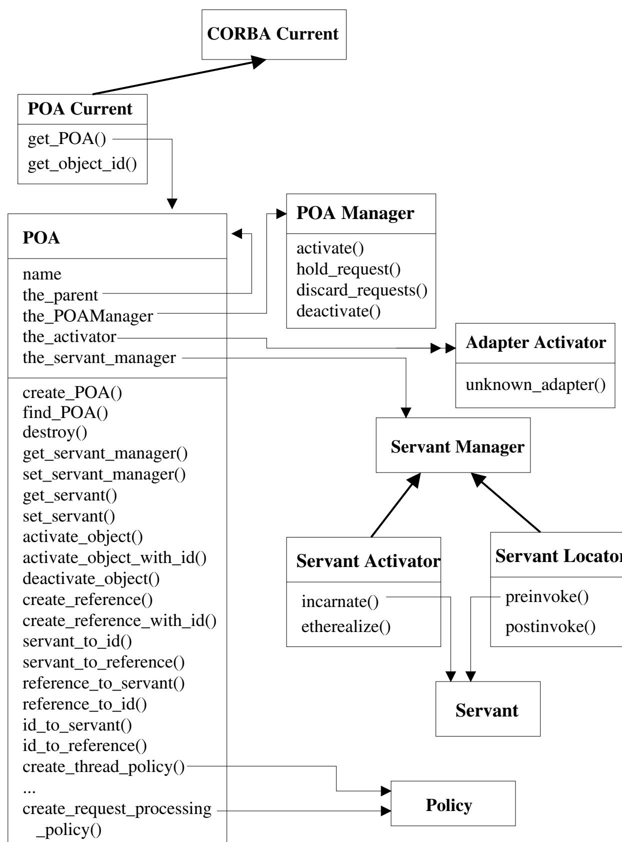

Figure 4.5 Different elements of a portable server.

Both servant activator and servant locator can also throw a special forward exception instead of returning a servant [79]. This exception contains a new object reference to forward the request to, possibly to an object realized in a different server on another system, employing the GIOP location forwarding mechanism. Forwarding allows, for example, the implementation of load balancing or redundant services: the servant manager would check its replicated servers and forward the request to an available one.

Figure 4.6 summarizes all the steps described above. A overall picture of the different elements related to the OMG's POA is given in Figure 4.5. Details of their IDL interfaces can be found in [79].

4.3 TECHNICAL ISSUES

This section provides a detailed description of BOA and POA, concentrating on different issues involved in processing object requests.

4.3.1 Overview

It is important to put both adaptors side by side to give a better understanding of their strengths and weaknesses. In order to do this, there are several aspects of an adaptor that need to be evaluated: server, object, servant, object grouping, request redirection, and multithreading and concurrency control.

Server A server must be registered before the adaptor can activate it. Activation happens when the first request arrives for the implementation. During its up-time, a server allocates and deallocates memory used by its active objects. When no longer needed, the server and its objects will be deactivated. Although this scenario is typical of a server lifetime, there are a few issues to be considered. One of them is related to the way servers are registered and how they manage their memories. Another issue is about the way servers are gracefully deactivated and the way objects and their servants participate in this process.

Object Several issues are related to objects. These include object identity (oid), lifespan, activation, deactivation, binding and lifetime relationship with their servants. Here we describe these concepts.

Each object has an oid, differentiating itself from others. The way an oid is defined and the information it supports (operations, etc.) are to be considered. An object's lifespan is the time period during which the object exists. Classified by their lifespan, there are two kinds of objects: persistent and transient. A persistent object is an object that can survive beyond the process that created it. Persistent objects continue to exist until they are explicitly deleted, while a transient object's lifespan is constrained by its creating process. Whether or not the object lifespan is contemplated in an adaptor should be considered. Object activation prepares an object to accept and process its requests, whereas object deactivation shutdowns an active ob-

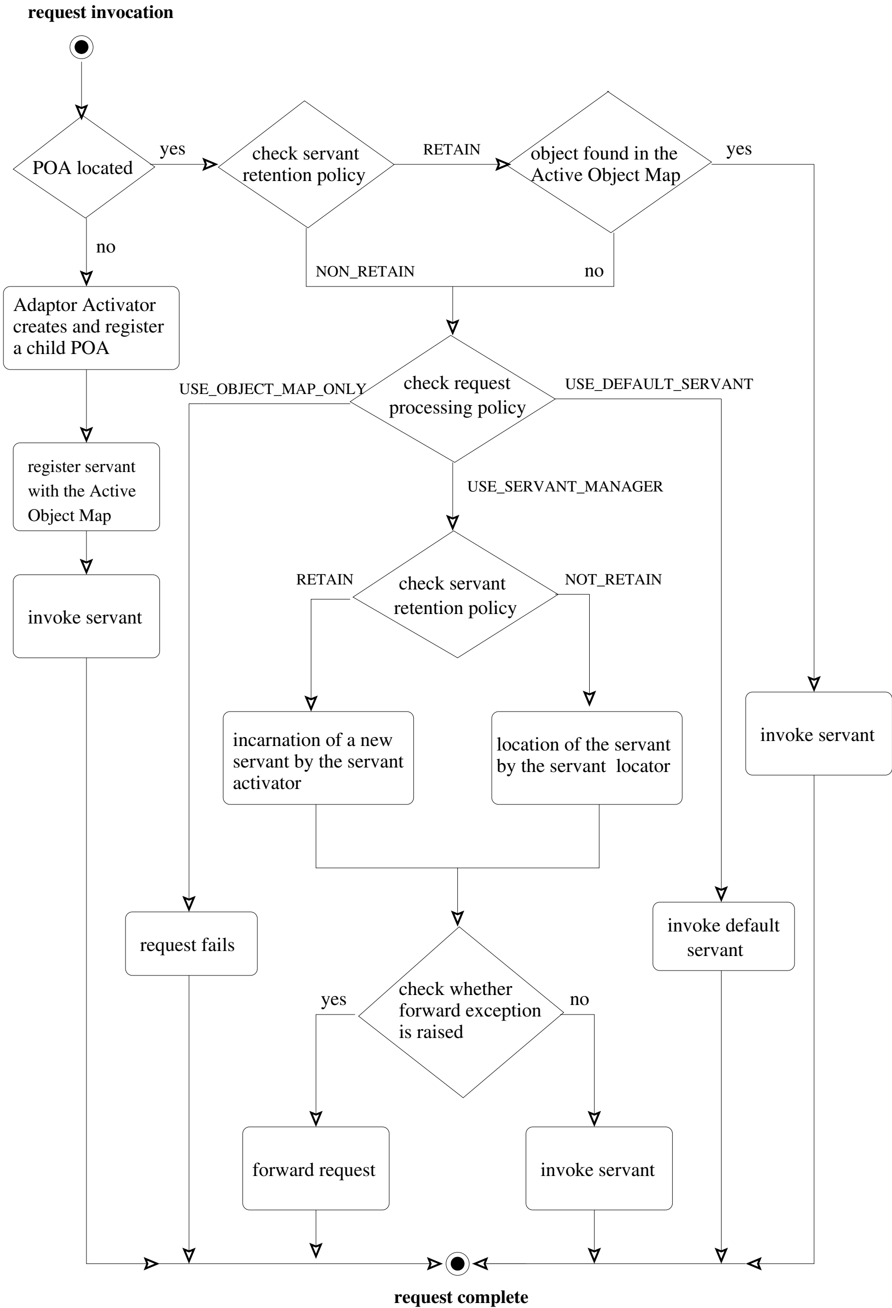

Figure 4.6 POA-based request processing.

ject. Issues about object activation and deactivation relate to the way objects of an adaptor are activated and deactivated, the styles of activating objects, and whether or not objects can be activated on demand (activated when the first request arrives) and if so. Finally, regarding the relationship with servants, the issues to be considered are the types of relationships that exist between objects and their servant lifetime and the way this affects object implementations.

Servant An important aspect of servants is the registration process. A servant is registered to record the association with its implemented object. There are two ways of registering a servant: explicit and implicit registration. An example of implicit registration is when a servant class's instantiation triggers its constructor to performs registration. In contrast, an explicit registration is usually obtained by explicitly calling a method. The types of registrations supported and how a servant is registered are important considerations for an adaptor.

Another important aspect of servants is the relationship between servant, skeleton and stub. One feature of this relationship that deserves special attention is collocation, an optimization technique which avoids request/reply overheads (marshaling, unmarshaling, etc.) when the target object is in the same process as the requesting object. This is achieved by having a separate inheritance hierarchy for the client-side and the server-side classes, allowing a request to be passed to the appropriate local servant when its target object is local or to the stub if the object is remote. The level of control which an adaptor has on the servant-skeleton-stub relationship is crucial for the server code's portability and performance.

Object Grouping Objects with the same characteristics are grouped together to simplify their manipulation. Object grouping enables programmers to apply the same set of operations to the members of a group, which is important when objects in the group need to be processed in a uniform manner. The issues to be considered are whether or not the adaptor supports object grouping and how a group is structured.

Assuming object grouping exists, object group activation and deactivation are also important issues. Object group activation activates an entire group of related objects when any single object in this group is accessed, similarly for object group deactivation. The purpose is to avoid overheads that might on a per-object invocation. What sort of strategy is devised by the adaptor should be carefuly weighed.

Request Redirection Request redirection is the act of forwarding requests to another object. It is applied in cases where the destination object is unable to or refuses to process the request.

Multithreading and Concurrency Control Single threaded servers are not scalable and maybe unable to handle a large number of requests, because requests are processed sequentially rather than concurrently. Employing multithreading increases a server's scalability, but raises the issues of thread allocation, critical section, and multithreaded requests.

Thread allocation is an act of assigning a request to a thread for its concurrent processing. How requests are distributed to their threads is reflected in the selection of thread policy. There are several thread allocation policies [34] to choose from: thread pool, thread per request, thread per client, thread per connection, thread per servant and thread per object. Servers with a thread pool pre-create their threads during startup; each incoming request is assigned to a thread. In the rest of the policies, operations are assigned a new thread. This thread is spawned for a request, client, client's connection, servant, and object, respectively. The issues related to thread allocation are the time when thread assignment take place, who should perform this and how it is done.

Critical sections are parts of a program where only one active thread should exist at any given time; if this not enforced, wrong results can occur. Ensuring this requires mechanisms such as locking and thread synchronization. Some means should be provided to specify and protect an object's critical sections.

A multithreaded request is a request which is handled by multiple threads. The request is broken down to smaller tasks and each of them is assigned to a thread. These tasks should be computationally complex enough to justify their thread creation. The approach taken by an adaptor to accomplish efficient request multithreading needs to be investigated.

4.3.2 Basic Object Adaptor

This section provides details about BOA, elaborating on issues described in Section 4.3. As mentioned earlier in this chapter, BOA has problems with its specifications. Some of these problems, such as object activation and oid assignment, are caused by their operations having an `ImplementationDef` interface as its parameter. This interface contains information that describes an implementation object. Since OMG saw this information as system specific, it decided not to standardize `ImplementationDef`. As a result, operations that have this interface as their parameters are not portable. Other sources of problems are: BOA's basic nature (for problems in multithreading and concurrency control, relationship between object and servant lifetime, object grouping, object group activation and deactivation), considered as being environment specific (for the problems in server's memory management and registration) and inadequate specification of other problems.

Server Information related to the server registration process is left undefined because its characteristics are environment specific. Regarding the server activation issue, BOA specifies four policies to activate its servers: shared server, unshared server, server per-method and persistent server. These policies are illustrated in Figure 4.7. Under the shared server policy, multiple active objects of a given implementation share the same server. Figure 4.2 depicts a shared server policy. For the unshared server policy, there is only one object of a given implementation active. Object activation is not necessary in this policy. Servers based on this policy are launched for each method invocation, whereas servers in the persistent server policy are activated by means external to BOA. Note that concept of "persistency" used in this context

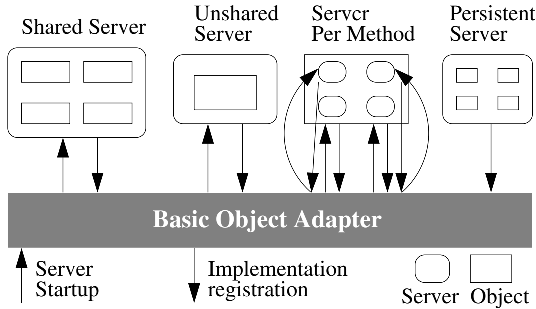

Figure 4.7 Steps in activating server in all policies.

is different from the same concept used in database systems. A more appropriate meaning is “externally activated server” [87] .

When a server with an unshared activation policy is ready to accept a request, it will notify BOA by calling $BOA :: obj_is_ready()$. This also activates its sole object, eliminating any need to perform a separate object activation. Servers with other activation policies should use $BOA :: impl_is_ready()$ instead and carry out their object activations in another step.

During the server’s up-time, its implementation might receive a request that returns a value. When this happens a memory space will be allocated to hold this value. Similarly, a memory space is also allocated for the implementation when its object is activated. When the server needs to be shut down, an appropriate deactivation’s operation will be called. Depending on the server’s activation policy, a server can be deactivated by calling $BOA :: deactivate_impl()$ (if the server is a shared or persistent server) or $BOA :: deactivate_obj()$ (if the server is an unshared server), or automatically after processing a request (if the server is a per-method server).

Although the above process seems clear, there are several details missing. First, there is no mention on how BOA and implementation objects are synchronized after their server is activated. Without synchronization, BOA might start to deliver the request even when the target object’s implementation is not ready with its event handling process. In some ORB products (e.g., Orbix [52, 53]), their server calls $BOA :: impl_is_ready()$ and blocks. Others make multiple $BOA :: impl_is_ready()$ calls, with another operation is used for synchronization. This causes $BOA :: impl_is_ready()$ to behave differently from one ORB product to another. No specification regarding server memory management is given. Some rules for each BOA’s operating environment (e.g., POSIX) should have been defined, but they were considered to be system dependent [71]. Also, how the objects and their servants can partake in the server’s deactivation process is not mentioned.

Object An oid in BOA is usually embedded in the object reference and unique within the scope of a server. It is of the type `ReferenceData` and mapped to a sequence of 1024 octets. An oid’s value needs to be converted to `ReferenceData` before it can be used to assign an object. Oid assignment can be done by calling

$BOA :: create()$ which creates an object reference as a side effect. Other arguments of $BOA :: create()$ are: `ImplementationDef` and `InterfaceDef`. The first argument was explained earlier, while the function of the second one is to define the object's interface. No ambiguities appeared from $BOA :: create()$ itself; however, the creation of an `ImplementationDef` has never been completely specificatied. For this reason, assigning an oid in BOA cannot be performed in a standardized manner. No other support on oids is available.

All BOA objects are persistent; however, BOA does not provide any specifications on transient objects. Proprietary mechanisms are used instead to carry out their creations. When the first request for an object arrives, BOA will perform an up-call to its object activation routine. The object's Id is made available for activating the appropriate object. After the object is activated, it will indicate its readiness to BOA by calling $BOA :: obj_is_ready()$.

Previously we showed all that BOA specifications give concerning object activation. No additional details on the routine's characteristics are given; this affects the availability of activation on demand. There are several points not addressed in the specification, including: (i) the parameters of the routine, (ii) the way BOA starts the activation, (iii) the way appropriate code is loaded and passed to BOA, (iv) the initialization of the state of the target object's state, and (v) the assignment of objects to the server. The point (v) is determined by the object implementor's defined policy. How this policy is defined in the `ImplementationDef` is again, unspecified. Vendors attempt to resolve these questions with their own solutions (e.g., Orbix with its `Loader` class [52, 53]), but these are proprietary. Furthermore, BOA does not explicitly mention the object activation styles available. Although, in practice it is clear that most, if not all, ORB products use the servant per-object activation style: one servant is instantiated and registered for every activated object.

Deactivating BOA objects is the job of their implementations and is accomplished by calling $BOA :: deactivate_obj()$. When there are too many objects active in the server, some might have to be deactivated to free up the server's memory. However, there is no call-back function that can be used by BOA to deactivate them. In some CORBA products, an object's deactivation is triggered by its servant's destructor, while others use a third call-back function.

Figure 4.8 shows the relationship between object and servant lifecycles. As depicted, the BOA specification does not attempt to isolate an object's lifetime from

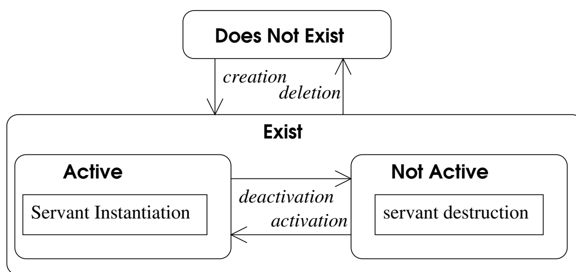

Figure 4.8 Object and servant lifecycles in BOA.

its servant's. Thus BOA's object activation is analogous to its servant instantiation. Similarly, an object deactivation in BOA will cause its servant to be destroyed. This coupling prevents servants from being reused to implement more than one object. Nevertheless, BOA offers a simple, but reasonably solid concept of object and servant lifetimes.

Servant There is no mention in the BOA specification about any mechanism for registering servants; instead, vendor-specific mechanisms are used, and the mechanisms are based on implicit registration using the servant's constructor. The specification does not provide details on skeletons and their relationship with servants. Servant methods and their signatures are described, but not the names of base classes that the servants have to inherit from. Eventually, servants code are not portable.

Other than allowing a separate inheritance hierarchy, little or no information related to collocation can be found in BOA specification because collocation is an implementation-specific issue rather than an independent one. Support for collocation varies from one ORB product to another. Most products have their ORBs checking the location of a target object. If it is local, the ORB will use the target object's stub to contact its remote implementation. Otherwise, ORB will call BOA to return the implementing servant. Location transparency is preserved since the client has no indication of where the target object actually resides. Clients cannot distinguish the target object's stub from its servant, and this because both have the same interface name and method signatures. However, if an interceptor [78] is used, (i.e., an additional component of an ORB which can provide additional processing steps before a request/reply continues with its normal invocation path,) it might not be able to detect local invocations [34]. Therefore, not all requests and replies will go through the interceptor. This could cause problems, especially if the interceptor is used to provide security and access control.

Object Grouping Details on object grouping are unavailable in BOA. Usually, programmers have to develop their own solution. No strategy has also been put in place to address object group activation and deactivation.

Request Redirection CORBA 2.0 's GIOP specification states that the use of a reply message with LOCATION_FORWARD status diverts requests to another object. This message contains a reference to an object where the request should be re-sent. Clients will transparently send the current and subsequent requests to the other object. However, BOA does not specify how to exploit this feature in a standardized manner. Even if it did, no operations for manipulating this message are declared in the GIOP specification. That message is used only by the ORB internally.

Multithreading and Concurrency Control No details on multithreading and concurrency control can be extracted from the BOA specification. In Orbix for example, the ThreadFilter class is used to deal with thread allocation. Some levels of support for other issues are also available, but these are proprietary.

4.3.3 Portable Object Adaptor

This section shows how POA addresses the issues presented in Section 4.3. Even though some of these issues, such as server management, were briefly described in Section 4.2.2 we prefer to re-discuss them in a more detailed way to make sure that the concepts related to POA are well understood.

Unlike BOA, multiple instances of POA can exist in a server; each is identified by its name. As depicted in Figure 4.9, these instances are organized in a hierarchical structure with `rootPOA` as its root. A root POA is managed by the ORB and its references can be obtained by calling $ORB :: resolve_initial_references()$. POA provides a namespace for its object's oid and another namespace for other POAs. Each POA manages a group of objects that share the same characteristics. An object is considered to be managed by a POA if its creation happened in that POA. Once an object has been created in one POA, its management responsibility cannot be transferred to another POA. Characteristics of the managed object are reflected in its policies. The value of a policy is set during the POA's creation or the policy's default value is used. A POA's policy is not inherited from its parent, nor it can be changed.

Server The POA specification did not have enough details regarding server registration, memory management rules, and process activation. Even so, the details of the synchronization process between the POA and its implementations are provided. Each POA is synchronized by its POAManager. This can be achieved because the POA manager controls the processing state of its POAs. There are four processing states of the POA manager: Active, Holding, Discarding, and Inactive. Its state transition diagram is shown in Figure 4.10. When a POA manager is in the active state, its POAs receive and process incoming requests. When a POA manager is in holding state, all incoming requests are queued. If a POA manager is in the discarding state, all incoming requests will not be queued or delivered. POAs whose POA manager are in the inactive state are ready to be destroyed; once a a POA manager enters this state it cannot proceed to any other state. Its POAs will stop working and all of their requests will be rejected.

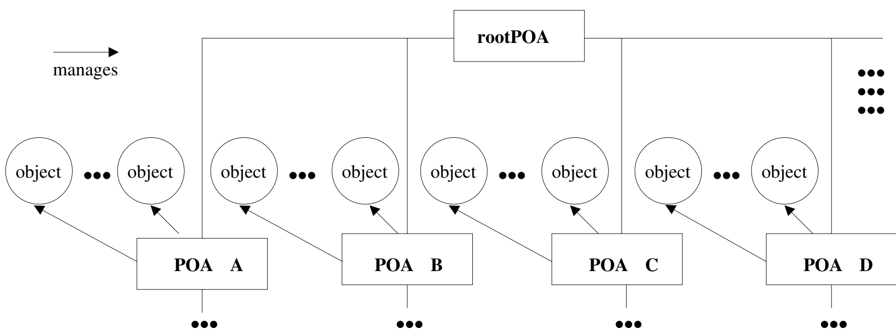

Figure 4.9 POA hierarchical structure.

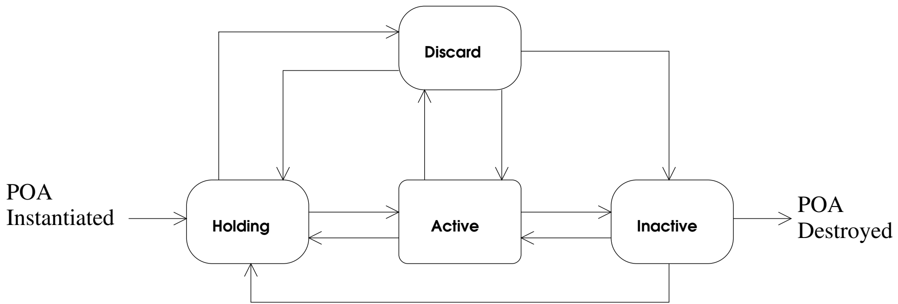

Figure 4.10 POA manager states.

A POA server is shut down by calling $ORB :: shutdown()$. This operation includes the option of immediately shutting the server down or waiting until all ORB processing tasks (object deactivation and requests processing) have been completed. When a server is being shutdown, all of its adaptors are deactivated and each of them triggers deactivation of their objects. This object deactivation allows objects and their servants to participate in the overall server deactivation process.

Object An oid is unique within the scope of a POA instance. It is opaque, has the type of `ObjectId` and is mapped to a standardized sequence of octets. An object's oid is usually embedded inside object references along with the object's POA name. An oid's value must be translated to octets before being assigned to an object. Two POA policies related to object identity are: id assignment and object id uniqueness policies. The id assignment policy provides two oid assignment methods: assigned by the server (has the policy value of USER_ID) or assigned by the POA (with the policy value of SYSTEM_ID). If no policy value is given at the POA creation, the default value SYSTEM_ID is used. The operation $POA :: create_reference()$ is used for the first method, while the latter should call $POA :: create_reference_with_id()$. Both operations return a new reference, pointing to an object of the assigned oid. Their usage is CORBA compliant.

Before describing the object id uniqueness policy, we first introduce the concept of object lifetime. Different from the BOA approach, the lifetime of an object and its servants in POA are separated. An object activation does not always correspond to a servant instantiation, similarly for object deactivation. Thus, the concepts of incarnation and etherealization are introduced to emphasize this point. Incarnation is a process of giving bodily form (i.e., servant) to a virtual CORBA object. Incarnation establishes an association between a servant and an object to serve the object's requests; in other words, it creates an implementation relationship between an object and its implementing servant. On the other hand, an etherealization is a process of taking away an object's bodily form/servant from it. This breaks the association between a servant and its implemented object and causes the object to be "unimplemented." As pictured in Figure 4.11, incarnations always occur when objects are activated. However, as we will see later for object deactivation, etherealization might not even be performed in an object lifecycle. Figure 4.11 also shows that a servant's

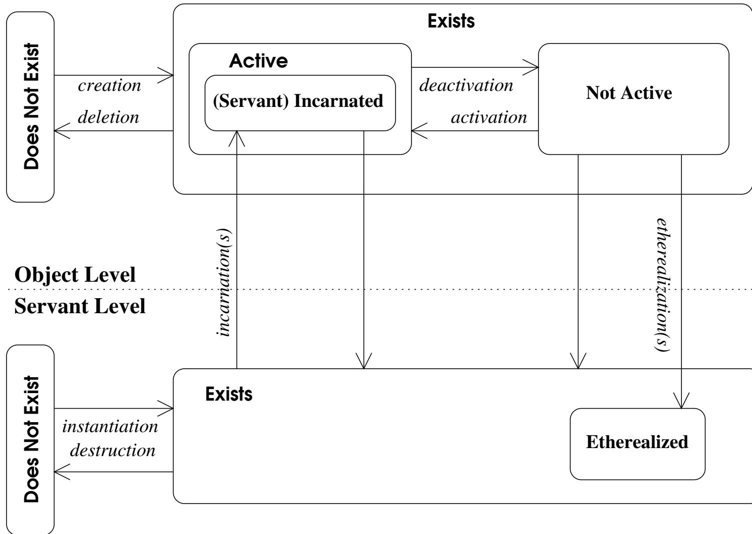

Figure 4.11 Objects and servants lifecycles in POA.

instantiation can happen before its object incarnation. The servant's destruction can occur later, possibly long after its object etherealization. This raises the possibility of one servant implementing multiple objects. The advantage of this separation is better memory and performance optimization techniques, although this could make the actual programming work confusing for people who are already used to BOA's lifetime concept.

As explained in Section 4.2.2, POA has several activation styles that can be chosen from. A choice can be made between various combinations of request processing and servant retention policy values. The request processing policy specifies how requests are processed by the POA, whereas the servant retention policy determines if a POA should RETAIN servants of active objects in its Active Object Map. This is owned by the POA and contains entries indexed by oids, each recording the association between an active object's oid and its implementing servant. When a request arrives for an object, this table will be referred to, for retrieiving the implementing servant.

When the ORB receives a request for an object, it will look for the target object's POA. Depending on the ORB product being used, the ORB might use the name of the POA, embedded in the target object's reference for this purpose. If this POA does not exist, its parent's `AdaptorActivator` will perform its creation process. An adaptor activator is developed by the programmer, but not needed for pre-created POAs. In order to create a particular POA in the POA hierarchy, all POAs that lead to it will also have to be created. Each of them will be created by their parent's `AdaptorActivator`. If we consider the example of POA A5 of Figure 4.12 when it is about to be created, if POA A1, A2, A3, A4 do not exist, then A4 will cause all of them to be created. POA A1 and A2 will be created by

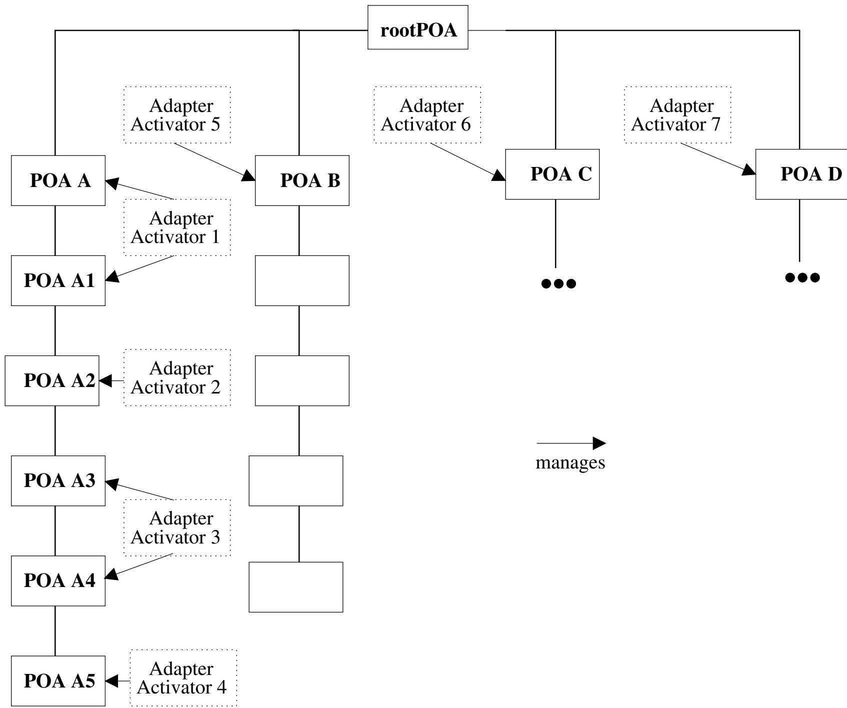

Figure 4.12 POA hierarchy.

`AdaptorActivator 1`, POA A3 will be created by `AdaptorActivator 2`, POA A4 and A5 by `AdaptorActivator 3`. After the target POA is created, the request will be delivered to this POA.

When the value of the request processing policy is USE_ACTIVE_OBJECT_MAP_ONLY, the POA must also have its servant retention policy's value set to RETAIN. This combination tells the POA to perform a look-up of its Active Object Map using the target object's oid. If a matching entry is found, a reference to the servant can be acquired, and the request will be passed to it. The POA's map has to be populated by explicitly pre-activating objects that will be used. $POA :: activate_object()$ can be used for that purpose if its id assignment policy value is SYSTEM_ID. If the value is USER_ID, $POA :: activate_object_with_id()$ is used instead.

When the combination is RETAIN and USE_DEFAULT_SERVANT, the POA should perform a look-up of the Active Object Map when a request arrives. When no entry matches, the request will have to be passed to a default servant for its processing. If the POA has no default servant registered with it, an exception is produced. Like the previous combination, the Active Object Map needs to be populated with objects that will be used. There are two types of default servants: per-interface default servant and for-all default servant [77]. In the former, the default servant is typically used for a POA instance which manages objects of a given interface. All requests are processed

by the default servant, which handles only those for objects of a given interface. The for-all default servant is capable of processing all requests regardless the interfaces the target objects have.

Any POA with NON_RETAIN does not have an Active Object Map and requires its request processing policy value to be either USER_DEFAULT_SERVANT or USE_SERVANT_MANAGER. If NON_RETAIN is combined with USE_DEFAULT_SERVANT, all requests will always be delivered to a default servant since no Active Object Map exists. As before, an exception will be produced if there is no default servant registered.

If the policy combination is NON_RETAIN and USE_SERVANT_MANAGER, the POA will rely on its registered servant manager to incarnate the requested object. A servant manager is a custom-made class written by programmers. Its incarnating operation is called by the POA to return an implementing servant of the wanted object. There are two kinds of servant managers available: servant activator and servant locator. A servant locator is used in this combination, with preinvoke as its incarnating operation. An Active Object Map does not exist and servant locator is always called for every incarnation.

If the combination is RETAIN and USE_SERVANT_MANAGER, the POA will first attempt to locate the appropriate entry in its active object map. If not found, the servant activator's incarnate operation will be called to incarnate the target object. After incarnation is completed, the POA will receive the servant from its servant activator and pass the request to this servant. A new entry for this servant will be added to the Active Object Map for its usage in the next invocations.

POA's object activations can happen in three ways: explicitly (by calling $POA :: activate_object()$) or $POA :: activate_object_with_id()$), on demand (using servant managers), or implicitly (by setting implicit activation policy's value to IMPLICIT). The first two have been explained previously. Implicit activation occurs when operations that logically require an oid to be assigned to an inactive object's servant are executed. Examples of these kinds of operations are: $POA :: servant_to_reference()$ and $POA :: servant_to_id()$. Intuitively, two values can be chosen for this policy: IMPLICIT_ACTIVATION and NO_IMPLICIT_ACTIVATION. The default value is NO_IMPLICIT_ACTIVATION. IMPLICIT_ACTIVATION also requires the POA to have SYSTEM_ID and RETAIN.

An object's deactivation occurs when the object is deactivated explicitly (by calling $POA :: deactivate_object()$); its POA is deactivated (by calling $POA\ Manager :: deactivate()$); its POA is destroyed (by calling $POA :: destroy()$); or deactivated by ORB/POA. The last deactivation is triggered internally by ORB/POA and it is unstandardized. When the POA has no servant manager registered, the object will be deactivated immediately without any etherealization process. POA will search for this object's entry and remove it from the Active Object Map. Etherealization also does not happen when function POAManager::deactivate and $POA :: destroy()$ are called, if their `etherealize_objects` parameter is set to false. Care should be taken when a POA Manager is set to the inactive state, just before its server's shutdown, as this would also deactivate its POAs's servant managers and stop etherealizations from happening.

Both persistent and transient objects are supported in a POA. A POA's definition of transient object has been revised in [78]. It states that a transient object is an object that cannot outlive the POA instance in which it was first created. Lifespan policy deals with this by providing two policy values: TRANSIENT and PERSISTENT. If no value is specified at POA creation, TRANSIENT will be used. A POA with policy values of PERSISTENT and UNIQUE_ID assigns oids which are unique across all instantiations of the same POA.

Servant POA supports both explicit and implicit servant registrations. The operation $POA :: activate_object()$ or $POA :: activate_object_with_id()$ can be used an explicit registration. These operations will activate an object by using a specified servant and register a new entry with the Active Object Map. Objects are implicitly registered if the POA's servant retention and request processing policies have the value of RETAIN and USE_SERVANT_MANAGER. A new entry is inserted into the Active Object Map after the incarnation process finishes. If the servant retention policy is NON_RETAIN instead, implicit registrations can occur when the servant locator is implemented with a map similar to the POA's Active Object Map.

The names of base classes and the operation signatures of POA servants are specified. Also, inheritance and delegation based approaches are better described and the servant's methods and its inheritance hierarchy are more clearly defined. At the end, the code's portability is significantly improved.

Collocation in POA is not specified because of its implementation specific nature. Beside location transparency, there are several other important issues that must also be considered in collocation, with interceptor and multithreading being some that are relevant to database adaptors. Like BOA, levels of collocation supports vary from one CORBA implementation to another.

Object Grouping An object group in POA consists of the objects that are managed by a POA instance. Their grouping is based on characteristics shared by its member objects. An object is a member of a group if it is managed by the group's POA. Note that since the structure of nested POAs is hierarchical, the overall structure of their object groupings is also hierarchical. However, because an object's management responsibility is determined during its creation, this grouping is static. Despite this limitation, the same operations can now be performed on all members of a group. For example, consider a servant manager being used for request processing. Its POA will use the same incarnate and etherealize the operation to apply the same algorithms to all its objects. Unfortunately, no strategy has been specified for object grouping activation and deactivation. Proprietary supports from ORB vendors, if any, are also limited. Programmers usually have to produce their own "home made" strategy to devise object group activation and deactivation.

Request Redirection POA facilitates request redirection with its $ForwardRequest$ exception. This exception contains the reference to an object to which a request should be forwarded. Assuming a GIOP based protocol is used, this exception will be returned to the client in a reply message with LOCATION_FORWARD reply

status. In the POA specification, for example, *ForwardRequest* can be raised during object incarnation by its servant manager's incarnate (for servant activator) or preinvoke (for servant locator) operation.

Multithreading and Concurrency Control POA's thread allocation is controlled by its thread policy. The value SINGLE_THREAD_MODEL does not use any of the thread allocation policies mentioned earlier. Instead, all requests will be processed sequentially in a single thread. The ORB_CONTROL_MODEL value implies that ORB is responsible for the thread allocations. Although the POA has a thread policy in its specifications, there are still some details missing. First, no description is given on how exactly a POA with an ORB_CONTROL_MODEL policy value can allocate requests to threads. The ORB might use one of the thread allocation policies previously mentioned or it might still use a single threaded model. The critical section problem is left unsolved, while the specifications on multithreaded servants are not provided. A POA's code portability cannot be guaranteed for multithread server; future POA revisions on this topic are expected.

4.4 DATABASE ADAPTORS

This section discusses issues related to POA-based database adaptors, but we will not describe solutions for these issues. The aim of this section is to provide appropriate details about the integration of CORBA and database technologies. Some solutions are provided in [95], which is a POA-based database adaptor, and [87], which is BOA-based adaptor; however, these will be not discussed in this book. Nonetheless, these issues are valid regardless of the basis adaptor adopted. For database adaptors which are not based on BOA or POA, this section can be used as a reference. Issues which are specific to an adaptor will be explicitly noted.

Persistency of objects is critical for many applications, such as banking and telecommunication applications. A POA does not deal with such a persistency; however, it can provide a nice framework for extensions for building database adaptors. There is no CORBA standard regarding database adaptors; however, because we believe that persistency is a very important issue, we would like to explain how the basics of POA can be extended to deal with persistency. As the reader may notice, persistency can also be achieved by using the OMG's Persistent Service. We have not chosen to explain this service because we believe that persistency should be transparent to the users, and therefore it needs to be addressed by the ORB (through POA) without any use of the services.

This section explains issues pertinent to development of a database adaptor. It starts off with the concepts and design, and includes the issues of a reference model, the selection of a basis adaptor, and architectural considerations. We continue with a discussion of the deficiencies of the present object model, followed by issues of the persistent object's identity, state, lifecycle, reference and its servant lifecycle. This section ends with performance, scalability and portability issues.

Concepts and Design Furnishing persistency to CORBA applications demands an integration with database systems. An integration approach must be devised for this purpose. First, a set of concepts is needed to construct the nature of the approach taken. These concepts are obtained from the approach's reference model. A reference model provides an overall view of the problems. It also provides knowledge of the approach's entities and their elements, thus enabling an understanding of things which are involved in the approach and to proceed with the conceptualization task. Once this is done, the rest of the issues should be resolved in line with those concepts.

At the start of the design process, developers have to consider whether or not to build the database adaptor on top of an existing conventional adaptor. If they decide to do so, two adaptors can be chosen from: BOA and POA. Each has its own strengths and weaknesses. Starting from BOA promotes reusability of an ubiquitous adaptor. However, its specification is problematic, not portable and will be phased out. Consequently, choosing BOA would commit a database adaptor to be a proprietary and short-term solution. POA, on the other hand, is not yet widely available in the current systems. Major CORBA vendors (e.g., IONA and Inprise) have not released their POA supported ORBs. Selecting a POA will advantageously position a database adaptor as a long term and portable solution. One could also avoid BOA and POA altogether and build a database adaptor from scratch. This gives maximum freedom in the development at the expense of portability and standard compliance. The last design issue is the database adaptor's architecture. For complex and sophisticated software like a database adaptor, having an architecture is compulsory. An architecture organizes the comprising elements of the software into a structural form. A good architecture provides clarity, extensibility, and maintainability to the software.

Object Model The CORBA object model defines concepts related to the object semantics and implementation objects. It does not address persistency since this feature is optional to CORBA objects and outside of its scope. Thus, this object model was not designed to model persistent objects, and consequently its semantic concepts such as types, interfaces, operations and attributes do not have persistent counterparts. Due to their absence, persistency syntax and semantics are undeclarable in IDL. As the result, the following issues are unsolvable: (i) how and when a CORBA object can be determined as persistent, (ii) what indicates its persistency, (iii) how and when a servant can be denoted as being capable of implementing a persistent object, and (iv) how non-persistent objects are treated. The last issue is particularly important in cases where persistent objects refer to transient objects.

One may extend the CORBA model to take persistent properties into account by adding new object semantics. An example of this approach is found in Secant's Persistent Object Manager [84]. While this solution is sensible, the introduction of these persistent semantics creates new IDL syntax and semantics which are foreign to existing applications. Unless their IDLs and implementation are changed, interoperability will be compromised. This creates an unwanted dilemma: extending the current object model could jeopardize interoperability, but without the IDL syntax and semantics persistency is left unfacilitated. Even if this problem is resolved, de-

velopers still have to be careful not to pollute the IDL with syntax or semantics derived from the database schema. Such design is prone to changes that occur from the schema updates.

One important aspect when considering persistency in CORBA environments is transparent persistency. Transparent persistency is the property in which persistency details are hidden at two levels: object level and implementation level. At the object level, clients should not be aware of the persistency of the objects. This is because CORBA allows interface-related information to be disclosed to the clients, but not its implementation-related information. Thus, persistence which is an implementation issue [80], should also be excluded from the client's view. Transparent persistency at the implementation level dictates that persistency details should not be revealed to the servants. The new object model is obliged to accomplish persistency in accordance to this transparent persistency property.

Object Identity The oid of a persistent object is used for several purposes, such as identification, storage and restoration of states, and processing requests. In the first usage, the oid needs to be in the form of an uninterpreted sequence of octets. It must have information on how to differentiate one object from others. In the next case, the oid must be in a form which is adherent to its database product and paradigm. It has to contain information for storing and restoring the right state. In the last case, the oid ought to be in an application-dependent form. The oid is expected to contain sufficient information to have requests to their objects processed by the right application service. In each of the above cases, its information should be unique. After these facts are analyzed, questions are raised on what sort of information, form and structure a persistent oid will actually have and how the above heterogeneities are accommodated. These issues are crucial since they affect how persistent objects are bound, identified, and manipulated.

A more complex issue is found in the oid generation process. A new persistent oid must correspond to a stored state and a set of services. This is not difficult for objects that represent new data and/or new service. However, it is more complicated when legacy databases and applications come into the equation. A database adaptor is obliged to resolve this issue appropriately.

oid uniqueness is also of concern because of the information it must contain. Each item of information might be used to uniquely identify the right object, state and service. However, its combined values might not be unique for an oid. Additionally, the database adaptor must now deal with the possibility of having multiple sources of the information. Each might have its own generation methods and issues.

Object State Issues in this section stem from the lack of details concerning the persistent object's state and its determination process. Persistent state determination selects which part of an object's state will be made persistent. Its method of selection and timing are the particular concern. How a persistent object's state can be accessed, especially if access to its values is restricted, should also be considered. Manipulating databases of different products and paradigms, to store and restore the state, is also problematic. Each database requires a different way of loading and sav-

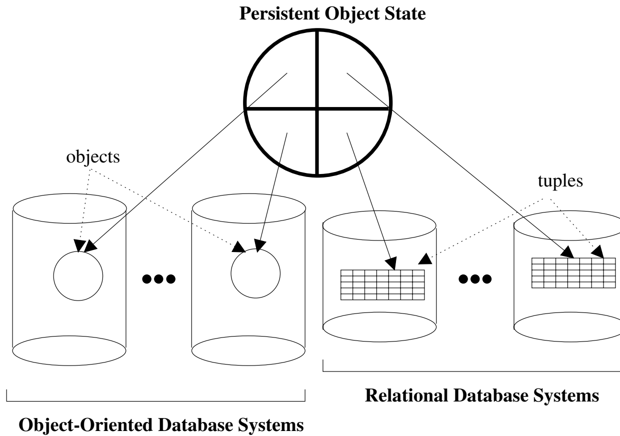

Figure 4.13 Persistent object's state with multiple databases and paradigms.

ing the object's state. Most importantly, the database adaptor must know how to use RDBMS's tuple or ODBMS's object for storing and restoring the state. This is further complicated when there are multiple possible databases in which the state can be stored. It is the database adaptor's obligation to select which of the databases should keep the state. Locating the stored state is also harder, since the database adaptor must find which database that stores the state of a particular object.

Moreover, the adaptor must also determine which attributes of state should be stored. Whether or not it is capable of dealing with state fragmentation [83] is a matter of consideration. Objects which have their state fragmented and stored across multiple databases must be reconstructed before they can be accessed. This increases the difficulties in finding the right databases, storing and restoring objects, and manipulating their state.

Object Lifecycle Events There are several issues which are common to object lifecycle events. Central to these is the issue of their integration with the storage steps of persistent objects. These steps are loading, inserting, querying, updating, and deleting the state of the objects. Exploitation of the basis adaptor, to manage a persistent object's lifecycle events, is significant in resolving the above issues. With respect to BOA, such integration must be accomplished in compliance with the CORBA standard. Last, but not least is the issue of the storage steps execution transparency of events. That is, how in programming terms can transparent persistency be imposed on the executions of these storage steps. There is a trade-off between the database information hiding and the performance which ought to be balanced to gain a suitable level of transparency and performance.

- *Event creation:* A persistent object comes into existence when its reference is created for the first time. During this process, the referenced object can either be activated at the same time (early binding) or later when the first request

arrives (late binding) [34]. In both cases, the reference will be associated with the servant of the activated object. The selection of binding is important since it influences the length of time required to create a persistent object. Which binding(s) are available, how they are supported, and which one to choose are the points that need to be considered.

Besides this, one also has to realize that the creation of a persistent object is not always matched with a similar process being performed for the state and service. A database adaptor might have to act as a front end to the existing data and/or service. Therefore, the conventional meaning of "new object" needs to be redefined for persistent objects. Further, consider when new data does need to be inserted into the database. In this case, the timing of the insertion is the influencing factor to the server's response time. When this insertion is to be done is to be considered.

- *Event activation:* How persistent objects are activated is significant. As explained in the previous sections, BOA based objects suffer from lack of specification on this issue. Employing proprietary supports is usually the only way to fill this vacuum. It is highly desirable for BOA-based database adaptors to be able to navigate through this problem. In contrast, POA specifies several styles in activating its objects. Hence, the selection of the activation styles to be deployed and whether some/all of the styles can be used to activate persistent objects is to be considered.
- *Event querying:* Designing query facilities in a database adaptor requires a trade-off between performance and the hiding of database internal details. As database internals are better hidden, the performance degrades. A database adaptor should allow query facilities to be used without violating transparent persistency or causing its query performance to suffer. The location of query strings is equally important in relation to the previous issue. Query strings must exclusively exist in the database code fragments. What databases (RDBMS/ODBMS/both), types of queries (pre- compiled/dynamic/both), and query manipulation language (OQL/SQL/both) are supported influences the performance and applicability of the database adaptor. Furthermore, relevant standards should be followed while catering for proprietary extensions. How queries that return a collection of objects are handled needs to be considered as well.
- *Event update:* The first step in an update process is to determine the updatables. Pertinent to this step is the issue of granularity. A fine-grained granularity is selective and more efficient, whereas the coarser one is simpler but less efficient. This, along with the update timing of the stored data, are both the determining factors in an update's response time. Moreover, the cause of an update is significant in as much as it decides the controls over the update.
- *Event deactivation:* An active object will eventually be deactivated to reclaim the memory it occupies. Deactivations are governed by the memory management rules of its server. As mentioned earlier, neither BOA nor POA have these rules. One of the reasons is to avoid distributed memory management

becoming part of the CORBA standard. This is due to its error prone and costly approaches. Thus, unless programmers provide their own object deactivation scheme, a server will crash because of memory exhaustion. The scheme devised might activate and deactivate objects several times, but it must observe the concepts of deactivation control, victim selection, deactivation safety and deactivation transparency.

Loosening the deactivation control amounts to less programming work since it lets the database adaptor take charge instead. However, it increases the reliance on the accuracy of its victim selection algorithm. The less accurate it is, the greater the chances of deactivating a still-in-use object. This is in conflict with the scheme's transparency and safety properties. A safe deactivation scheme guarantees the continuity and integrity of objects it deactivates, while a transparent deactivation scheme stops clients from being aware that such deactivation takes place. There are two kinds of deactivation safety: safety by state and safety by processing. A deallocation scheme is considered as safe by state if the state of objects it deactivates is preserved across their activations. Safety by processing protects a currently working object from errors and inconsistencies that might arise from their sudden deactivations. In particular, these properties must prevent cases where errors from the deactivations, which are triggered by clients/database adaptors, from being exposed and handled by the clients. This is because such actions increase the code complexity of the database adaptor's clients.

An object's deactivation process is always carried out in its lifecycle. This process is an ideal place to undertake any of the object's storage steps before it is finally deactivated. Deciding which storage step(s) should be included in an object deactivation is to be carefully contemplated.

- *Event deletion:* When a deletion occurs, it may or may not cause the deletion of its stored data and service as well. If the deletion triggers the removal of its data, the database will be located and the data will be removed. Crucial to the deletion's response time is the timing of the removal of stored data from the database. How a database adaptor determines when the deletion of data and/or service should take place and how the data can be removed needs to be answered.

Object Reference A relationship between two objects is represented by an IOR. However, relationships in DBMS are represented differently. Relationships in RDBMS are symbolized by keys, while in ODBMS they are presented by database references. Such mismatches must be bridged before an IOR can be used to access its persistent object.

Generating IORs is the responsibility of an adaptor. This usually requires the information pertinent to the referenced object to be encapsulated inside the IOR. Furthermore, in order to preserve the relationships that a persistent object has, all the IOR's of the relationships must persist as well. Their referential integrity must be maintained so that they are still usable when they are restored. Referential integrity

is conserved if references are not dangled, that is, pointing to a non-existent object. Also important is how to acquire references to remote CORBA objects when the references are not available locally. How oids can be made available during an IOR creation and how IORs of stored persistent objects are provided to the application are questions to be answered. How the mechanism providing IORs can fit into the application's code also needs to be considered. The above issues are to be solved in accordance with the IOR's transparent storability [87]. Transparent storability is a feature where converting an IOR to its usable form, that is, to string before it is stored or to a CORBA reference when it is retrieved, is not required. In other words, code that provides the actual application service are not aware that such IOR is actually stored in the database. How a database adaptor conserves this property is yet to be solved. In essence, IORs of persistent objects must be managed in such a way that they behave and perform similarly to a normal object.

Servant Lifecycle Events Issues in the servant lifecycle events are focused on providing actual programming language logic to the objects. As CORBA has several programming language bindings, solving these issues clearly rests upon the programming language's capability. This section discusses all of these issues below; they are applicable regardless of what programming language is used.

- *Event instantiation:* Objects with multiple servant classes have problems with their servant instantiations. Having multiple choices of the servant class is relevant when an object is to provide the same service in different ways. The reasons might be because of better performance, support for previous service, and so on. However, each of these servants might be implemented in a different approach—POA/TIE/DSI. Parameters of their constructors could be distinct and require them to be passed with different values. Some of the constructors have no parameters at all (zero/null constructor), while others have multiple parameters. When dealing with the existing code, these servants may already have factories to carry out their instantiations. A database adaptor is compelled to cope with these variables before it is able to instantiate these servant classes. Although not crucial, adding and removing the selectable servants can cause minor irritations. Usually, the relevant code has to be manually changed and recompiled. A more convenient way of handling these tasks is needed to speed up the entire editing process.
- *Event incarnation:* A persistent object's incarnation restores the object's state from its database and prepares the object with the state as loaded. There are two choices in relation to the servant that is used to incarnate. The first option is to use an already instantiated servant, while the second option instantiates a new one. Choosing which option should be taken is the problem in incarnating. When the second option is chosen, the selection of an existing servant to use is another issue. A database adaptor might have to be careful not to cause a servant to over-incarnate. This is to avoid a servant being overloaded with the requests of its objects. It is especially true in cases where threads are used in the servant. Likewise, finding the database that stores the persistent object's state is hard

particularly when there are multiple databases at the back-end. Furthermore, a database adaptor must load the object's state, set it with this state, and finally activate the object.

Two kinds of initialization steps can be identified in an incarnation process: steps related to restoring the object's state and those pertinent to the business logic. The former is more likely to be similar for all applications than the latter. Therefore, the same set of code could be used to handle the former's steps. This raises the possibility of automating the task of restoring the object state for servants in all applications. If there are multiple servants to choose from, each of them might have unique initialization steps. The adaptor should be able to execute the right initialization steps for a particular servant. A mechanism to automate both steps is required, whilst allowing their customizations.

Incarnation is a potentially dangerous step for the concept of separation of concern. A naive programmer would write the incarnation code without separating code fragments related to the database from others. However, since they are very closely related, separating them is not an easy task. The design of a database adaptor should be aimed at simplifying this task.

- *Event etherealization:* Similar to incarnation, two kinds of steps can happen in an etherealization process; the object's storage and business logic processing steps. The etherealization's problems resemble those of the incarnation as well. It is compulsory for the database adaptor to be able to prepare the object before it is etherealized. This might involve one or several storage steps. The storage steps are similar from application to application, compared to the business related steps. Hence, automation is a possibility. The existence of multiple servants might also coincide with the presence of their dissimilar etherealization steps. The adaptor should select the right steps for an appropriate servant. The customization and the automation of both steps are to be facilitated in their mechanism. The issue of enforcing the separation of concern also needs to be tackled. Recall from the previous chapter that etherealization may or may not happen. This could cause problems to objects that rely on the etherealization to perform their storage step(s). The adaptor should be able to cope with this problem, while maintaining the state's consistency of those objects.
- *Event destruction:* A servant destruction in some BOA based products could be used to provoke its object deactivation. However, caution must be exercised when destroying a servant of the POA or other BOA-based products. An accidental destruction could cause the objects implemented by the servant to be forcefully unimplemented without their proper deactivations. The servant's destruction event is ready to be used for the execution of its object's storage step(s). The selection of what storage step(s) should be performed in the destructor requires some consideration.

Performance, Scalability and Code Portability Increasing scalability and the performance of a database adaptor is more likely to stretch CORBA standard to its limit. More improvement can only be gained from employing proprietary extensions. However, such use works against the portability of the server's code. Unless

standardized mechanisms are used, this problem remains as the stumbling block in realizing a scalable, performance minded and portable database adaptor. The issues below describe problems encountered when developing database adaptors with those goals. Some of the issues described below are not directly related to the database adaptor, but rather to the overall server. They are there because a database adaptor can play a pivotal and influential role in attaining those goals for the server.

Clients measure a server's performance according to its response time and availability. A response time is the time elapsed from the invocation of a remote object until its reply is received. Availability is the server's probability of failing or becoming unreachable [22]. Clearly, a low response time and higher availability are very desirable. Speeding up a server's response calls for the reduction of time consumed by all steps in an object invocation. These optimizable steps are illustrated in Figure 4.14. A server's availability is enhanced if the server failures can be minimized.

The first optimizable step is marshaling and unmarshaling steps on the client side. There are two factors that influence the time taken to complete those steps: message size and the IDL data types it contains. More data to be transferred means a larger message, which at the end prolongs the time taken [34]. Increasingly complex and structured data types also play an adverse role on the response time. The delivery rate/call latency is dependent on the ORB products used. The rate is usually uncontrollable except if modifications are made to the source code of the ORB product being used. Lowering the response time of a database adaptor is much harder than lowering it in a conventional adaptor counterpart. The processing time of a database adaptor is longer due to the database operations that have to be performed. For example, in an object activation, the database is accessed to load the object's state. For a remote method invocation, the database might be queried/updated. If an object's state is cached, the database operation will perform faster than its remote method invocation. Hence, there is a significant time difference existing between these two; as a result, access to fine-grained data via a remote method invocation would incur

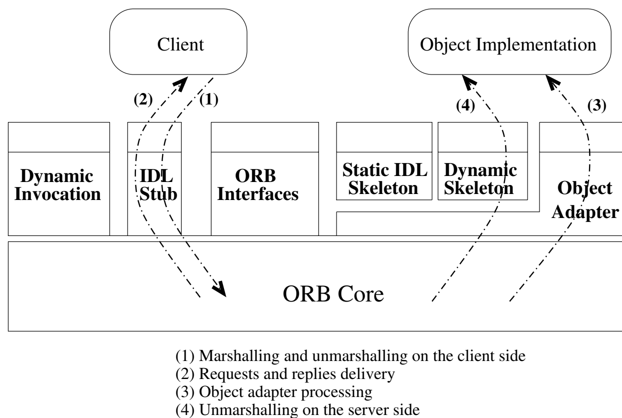

Figure 4.14 Optimizable steps of the flow of a request and its reply.

unacceptable overhead [87, 99]. If data is uncached, database operations could slow down the entire invocation process. Instantiation of the classes involved (e.g., servant class) is also effectual, especially in interpreted programming languages like Java. A massive number of instantiations decreases the response time of a database adaptor. The last optimizable step is the processing time of an implementation object. A heavily loaded implementation object has a sluggish response time, which slows the server down. Thus, managing the request load of an implementation object is also essential. The number of requests and replies contributes to the overall server performance. A decline in their numbers automatically improves the server's response. This is because it eliminates the need to perform the steps for their requests and replies.

Availability of a server with a database adaptor should be resolved at two levels: server level and database level. Failures at the first level are classified as omission failure, response failure, and crash failure [22]. An omission failure is a failure in which a server fails to respond to a request. Response failure is divided into two classes: value failure and state transition failure. The former happens when the server returns a wrong value, while the latter has the wrong effect on resources (e.g., sets the wrong values to data items). Crash failure is repeated omission failure and includes all kinds of server crashes. At the database level, failures range from transaction failures to physical and catastrophic failures. At the first glance, there seems to be no connection between these failures and the database adaptor. However, due to its closeness to the implementation objects, it is logical for a database adaptor to at least accommodate any fault tolerance mechanisms that are being used by its server.

Scalability is the characteristic where the system and application software do not need to change when the scale of the system increases [22]. In servers with database adaptors, scalability concentrates on their capacity to bring about reasonable performance as the number of persistent objects, databases, and requests grows. Thus, some optimization techniques need to be devised to enable the database adaptor to cope with an increase of these variables.

In order to avoid a database adaptor becoming a proprietary solution, it should be built on existing standards. However, unstandardized parts of a database are the potential problems when moving to different environments (different DBMS and/or ORB products). These parts might come from the DBMS or CORBA side. In the DBMS, they can be found in queries, database-generated oids, and so on. From CORBA these parts can be found in optimization techniques such as threading.

4.5 SUMMARY

In this chapter we have presented one of the main components of the CORBA architecture, the object adaptor, and explained the importance of including such an adaptor within CORBA. We have also explained the main architectural differences between the BOA and POA adaptors. Based on technical issues related to object adaptors, we provided an analysis of both BOA and POA and showed their strengths

and weaknesses. In the last section of this chapter we provided details of the issues that need to be taken into account in order to build database adaptors.

4.6 REVIEW QUESTIONS

- What is the advantage gained from placing an object adaptor between an ORB and implementations?
- What are the three steps in dispatching a request on the server side, assuming SSI is used?
- Compare the strengths and the weaknesses of using a default servant with those of using a servant manager.
- What is the difference between servant activator and servant locator?
- What is a persistent and a transient object?
- Why do vendors provide proprietary solutions (e.g., loader classes in IONA's Orbix) for loading inactive objects in BOA?
- What is the difference in the relationship between servant and object lifecycle in BOA, compared to that in POA?
- Which has a better support in multithreading: BOA or POA? Justify your answer. What is the problem with POA's multithreading support?
- Explain the concepts of incarnation and etherealization.
- Does the incarnation in an object activation corresponds to an etherealization in its object deactivation? Justify your answer.

4.7 EXERCISES

- What are the possible problems that could occur when a request bypasses interceptors which are used to provide security and access control.
- What is the consideration behind the use of USER_DEFAULT_SERVANT request processing policy?
- CORBA applications often need to store the states of their objects persistently in databases. The simplest approach would be to include in the servant classes of those objects all the necessary codes (SQL query strings, etc.) that access these databases, in order to manipulate (load, delete, edit, etc.) their object states. What is the main problem with this approach?
- What are the problems that occur if a database adapter is not built based on any of the CORBA adapters?