Object Interconnections

Using the Portable Object Adapter for Transient and Persistent CORBA Objects (Column 12)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
60 Aberdeen Ave., Cambridge, MA 02138

This column will appear in the April, 1998 issue of the SIGS C++ Report magazine.

1 Introduction

In this column, we continue our presentation of the new OMG *Portable Object Adapter* (POA) [1]. The POA was adopted by the OMG in mid-1997 as a replacement for the *Basic Object Adapter* (BOA), which was the original CORBA object adapter. As we described in our last column, the BOA was a major source of portability problems for CORBA applications due to its imprecise specification. Serious users of CORBA can attest to the frustrations of porting applications across multiple ORBs.

Rather than trying to repair the BOA, which would have required all CORBA applications written to the original specification to change, the OMG opted to create a whole new object adapter. This *Portable Object Adapter* (POA) specification is much more precise in its use of terminology than the old BOA specification. Thus, our last column was devoted almost entirely to explaining not only object adapters in general, but the POA terminology as well.

Our next several columns will examine detailed examples of how the POA can be used by real-world CORBA C++ applications. The POA specification is rather daunting, so we can't cover all the features and use-cases in a single column. Therefore, we'll start by explaining the usage of several POA policies described in our previous column [2]. Our primary focus in this column will be on the POA features that support *transient* and *persistent* CORBA objects. Subsequent columns will explore other POA features and discuss the circumstances under which various features are best used.

2 Developing a Simple Server with the POA

Our last column briefly described a very simple POA-based C++ server program. The following is a slightly revised version of this program. Each important step is highlighted in comments and described below.

```
int main (int argc, char **argv)
{
  // 1. Initialize the ORB.
  CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv);

  // 2. Obtain an object reference for
  // the Root POA.
  CORBA::Object_var obj =
    orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa =
    PortableServer::POA::_narrow (obj);

  // 3. Create a servant.  This C++ object
  // ultimately handles client requests.
  Null_Servant_Impl servant;

  // 4. Create a CORBA object and implicitly
  // register the servant with the RootPOA.
  Null_var null_impl = servant._this ();

  // 5. Export the new object reference, i.e.,
  // make the <null_impl> object available to
  // potential clients (not shown).

  // 6. Activate the POA, i.e., allow it to
  // listen for requests.
  PortableServer::POAManager_var poa_mgr =
    poa->the_POAManager ();
  poa_mgr->activate ();

  // 7. Run the ORB's event loop.
  orb->run ();

  // 8. The ORB is shutdown.
}
```

Let's examine this program again and study each of its steps in more detail:

1. Initialize the ORB: First, we must call the standard `ORB_init` method. This is a static member function that plays the role of an "ORB factory." Portable CORBA applications must make this call to initialize the ORB in a process.

An optional third argument to `ORB_init` allows a multi-ORB application to initialize a specific ORB. Eliding this argument results in the "default" ORB being initialized. This behavior is perfect for our application since only special applications that link multiple ORBs together in a single process need to supply the third argument. The return value of a successful call to `ORB_init` is an object reference for the initialized ORB.

2. Obtain an object reference for the Root POA: Once the ORB is initialized, the ORB object reference is used to obtain a reference to the *Root POA* in this ORB process. As we explained in our last column, a single server process may contain a hierarchy of nested POA instances, as shown in Figure 1. However, all servers contain at least one distinguished POA called the Root POA.

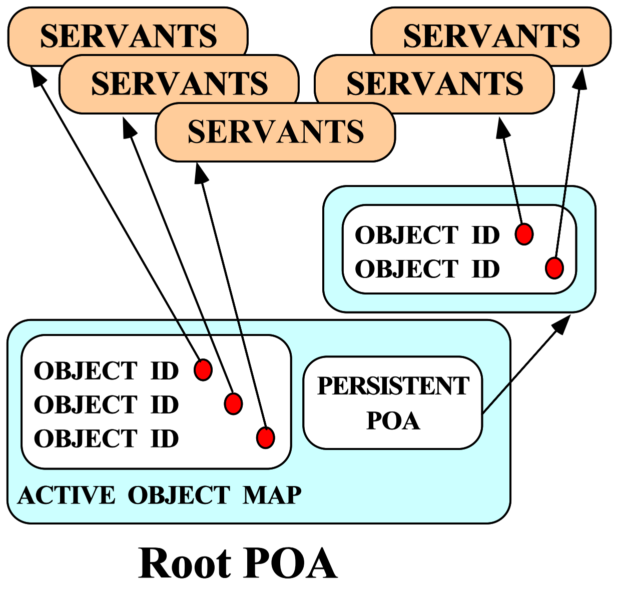

Figure 1: Hierarchical Nesting of POAs

An object reference for the Root POA is obtained by invoking `resolve_initial_references` on the ORB. This operation provides a miniature Naming service that bootstraps a CORBA application with essential object references. Object references obtained from `resolve_initial_references` can be references to per-ORB Singleton [3] services, such as the Root POA or Interface Repository. Applications can also obtain references to services from which other object references can be obtained, such as the Naming Service or the Trader Service [4]. Since the return value of `resolve_initial_references` is typed as `CORBA::Object`, the result must be narrowed to the desired type, which in this case is `PortableServer::POA`.

Note that the object references for both the ORB and the Root POA have slightly different semantics than those of normal CORBA objects. This difference stems from the fact that the ORB and all POAs can only have their methods invoked from within the process, *i.e.*, their methods cannot be invoked remotely. The OMG Portability Enhancement Specification describes objects with this property as being "locality constrained." In general, the locality constraint property is shared by all CORBA *pseudo-objects*, which are objects that comprise the local ORB implementation.

3. Create a servant: A *servant* is created by defining an instance of our C++ implementation class, which is called `Null_Servant_Impl` (not shown). Our previous column explained in detail that a servant is a programming language object capable of *incarnating*, *i.e.*, providing the implementation for, a CORBA object. Upon creation, the servant is just another C++ object. Until it has been registered with a POA, the servant lives only in the (local) C++ world, has no association with any CORBA object, and therefore cannot be accessed by a remote client.

4. Create a CORBA object: A CORBA object is *created* by invoking the `_this` function of the newly-created servant, which we've called `null_impl` for simplicity. Due to the policies supported by Root POA, especially its `IMPLICIT_ACTIVATION` policy, invoking the servant's `_this` function outside the context of a CORBA request invocation results in the creation of a *transient* CORBA object. Section 3.1 describes transient CORBA objects in detail and contrasts them with *persistent* CORBA objects.

In addition to creating a transient CORBA object, the `_this` operation also implicitly registers the servant with the *active object map* in the servant's default POA, which is the `RootPOA` in this example. Ultimately, the POA uses the active object map to demultiplex incoming CORBA requests to the appropriate servant [5], so that the servant's designated C++ method can be dispatched automatically.

The type of the object reference returned by `_this` is determined by the most-derived IDL interface supported by the servant. Since this example is not intended to focus on the IDL aspects of our CORBA object, our IDL interface is extremely simple, *i.e.*:

```
interface Null { void op (); };
```

The return value of `_this` is allocated dynamically. Therefore, it must be released by the caller. To simplify memory management, we assign the return value to a `Null_var`. The OMG IDL compiler generates this C++ smart pointer for `Null` object references. In general, the use of smart pointers, *i.e.*, instances of `_var` types, ensures that resources allocated locally for an object reference are automatically reclaimed once the `_var` goes out of scope.

5. Export the new object reference: Once the CORBA object has been created, we export it to interested clients. Typically, this involves the use of a location service, such as the Naming service or Trader service. Again, since our focus is on the POA we've omitted this step here. Future columns will examine CORBA location services in more depth.

6. Activate the POA: After the CORBA object is created, we can allow our POA to process incoming requests. The request processing states of a POA are controlled by its associated `POAManager` object, which can be accessed using the `POA::the_POAManager` operation as shown in the example.

All `POAManagers` are created in a *holding* state. In this state, all requests sent to the POA are held in a queue. To allow requests to be dispatched, the `POAManager` associated with the Root POA must be switched from the *holding* state to the *active* state. This transition is accomplished by invoking the `POAManager::activate` operation.

7. Run the ORB's event loop: Ultimately, the ORB itself must be told to start handling incoming requests. This may seem redundant since we've already activated request handling for the Root POA, but it's actually quite handy. For instance, the application may want to share the only thread in a single-threaded server between the POA and some other event loop, such as a GUI event handling loop.

Putting a blocking `run` operation on the POA would mean it could take over the main thread completely. Not only would this prevent the GUI handlers from receiving their events, but any other POAs in the process would be prevented from receiving and dispatching CORBA requests. In fact, in a server program composed of independently-developed component libraries, the program `main` may not have access to all the POAs that are created in the process. Therefore, it could not share the main thread with all of them. For these reasons, the `ORB::run` call serves as a centralized starting point to allow all active POAs to start receiving requests.

8. The ORB is shutdown: Eventually, the server process either exits cleanly or exits abortively. A clean exit may occur because an operation upcall or another thread in the program invokes `ORB::shutdown` on an `orb` pointer, which causes its `ORB::run` to return. An abortive exit can occur due to an overall system shutdown or by a system administrator killing the process.

This completes our detailed analysis of our small, yet functional, POA-based server program. As you can see, the POA allows simple servers to be written simply, mainly due to the sensible defaults for the policies of the Root POA. However, as we'll see in the following section and in subsequent columns, the entire range of POA policies provides considerable flexibility and support for CORBA server applications.

3 CORBA Object Lifetime

The focus of the remainder of this article is how the POA supports different policies for managing the lifetime of CORBA objects.

3.1 Transient vs. Persistent Objects

Once the server process described in Section 2 exits, the CORBA object created by the program no longer exists. This is because it was created as a *transient* object. Transient objects have a lifetime bounded by the lifetime of the process in which they are created.

The key to understanding transient objects is to recognize that if our original server program was restarted, it would create a new transient CORBA object with a completely different object reference. Therefore, any clients still holding references to the previous CORBA object would receive a `CORBA::OBJECT_NOT_EXIST` exception if they used those old object references to attempt request invocations. Note that compliant ORBs must not allow clients to use old object references to successfully reach the new transient CORBA object. This property must hold even if the server program were restarted on the same host using the same communications port to receive requests.

Transient CORBA objects can be useful for certain things like callback objects, such as the event notification service described in [6]. In this case, once the process goes away, receiving a callback or event notification is probably no longer a necessity.

However, many CORBA applications must be able to exit and then later start back up. For these applications, it is essential that clients still be able to invoke requests on their original, *persistent* CORBA objects. A persistent CORBA object is one that outlives the process in which it's created. As we explained in our last column, the use of the overloaded term "persistent" in this context does not imply that the CORBA objects are stored in databases when inactive. Rather, it refers to the fact that the lifetimes of such CORBA objects "persist" across server process activation and deactivation cycles.

If a client invokes a request on a currently-inactive persistent object, the ORB must ensure that a process to hold the object is created transparently and that the request reaches the object. Any ORB that doesn't do this is simply not a CORBA-compliant ORB, though it may still be useful for many common use-cases.

An example of an application that may require persistent objects is the root `NamingContext` object in a Naming server for a workgroup. Typically, this `NamingContext` serves as a bootstrapping point through which client applications discover other object references. These object references can subsequently be used to access other departmental resources.

Using a transient object for the root `NamingContext` would result in clients having to continually update their root `NamingContext` references. However, this may not be possible because administrator privileges might be required for update. Thus, the root `NamingContext` object is an ideal candidate for being a persistent CORBA object.

3.2 Programming a Persistent CORBA Object with the POA

The following two changes must be made to make the server in Section 2 contain a persistent object rather than a transient object:

1. The object reference must not be created using the Root POA: The lifespan policy of the Root POA is TRANSIENT, not PERSISTENT. Therefore, another POA with the PERSISTENT lifespan policy must be created and used.

2. The CORBA object must not be implicitly activated: Our invocation of `_this` on our servant causes the POA to generate the `ObjectId` of our object for us. We need to control the `ObjectId` explicitly so that we can ensure that the same id is used for each activation of the server process.

The example below includes the code required to effect these changes:

```
int main (int argc, char **argv)
{
  // Initialize the ORB.
  CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv);

  // Obtain an object reference for
  // the Root POA.
  CORBA::Object_var obj =
    orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa =
    POA::_narrow (obj);

  // (1) Create the desired POA policies.
  CORBA::PolicyList policies;
  policies.length (2);
  policies[0] =
    poa->create_lifespan_policy
      (PortableServer::PERSISTENT);
  policies[1] =
    poa->create_id_assignment_policy
      (PortableServer::USER_ID);

  // (2) Create a POA for persistent objects.
  PortableServer::POAManager_var poa_mgr =
    poa->the_POAManager ();
  poa = poa->create_POA ("persistent",
                         poa_mgr,
                         policies);

  // (3) Create an ObjectId.
  PortableServer::ObjectId_var oid =
    string_to_ObjectId ("my_object");

  // Create a servant to service
  // client requests.
  Null_Servant_Impl servant;

  // (4) Register the servant with
  // the POA explicitly.
  poa->activate_object_with_id (oid, &servant);

  // Allow the POA to listen for requests.
  poa_mgr->activate ();

  // Run the ORB's event loop.
  orb->run ();

  // ...
}
```

The four primary modifications to the code needed to support a persistent CORBA object, rather than a transient one, are marked by numbered comments, which are described below.

1. Create the desired POA policies: In the code marked with comment (1) a list of POA `Policy` objects is created. The two policies required are:

• **Persistent Lifespan:** Make the POA support persistent objects rather than the default transient objects. This policy is selected by calling the POA policy factory operation `create_lifespan_policy` with the `PERSISTENT` enumeral.

• **User-supplied ObjectId:** Rather than allowing the POA to supply an object identifier to identify the object within that POA, we choose to supply our own `ObjectId`. This policy is also selected by calling the POA policy factory operation `create_id_assignment_policy` with the `USER_ID` enumeral.

2. Create a POA for persistent objects: The second change required to support persistent objects is to create a new POA with the policies created in the previous step. All POAs except the Root POA are created as children of other POAs, and they are logically nested under their parent POAs, as shown in Figure 1.

Our new POA is created by invoking `create_POA` on the Root POA. This method is passed a name for the new POA (*i.e.*, "persistent"), a `POAManager` (*i.e.*, the one from the Root POA), and a `PolicyList`. Once the new POA is created, this program no longer needs its reference to the Root POA, so we overwrite it with the reference to the new POA. The `resolve_initial_references` method on the ORB can always be used to access the Root POA again if necessary.

3. Create an ObjectId: After the comment marked (3) is our third code change: which causes the creation of a `PortableServer::ObjectId`[1] for our persistent CORBA object. The `ObjectId` uniquely identifies the object within the scope of the POA with which it's registered. It is not a globally unique identifier; it's only intended to disambiguate objects within the scope of one POA. As has always been the case in CORBA, objects are ultimately identified by their object references, and the POA does not change that.

An `ObjectId` is defined as an unbounded `sequence` of `octet`. However, it's usually easiest just to use a string for the object identifier and convert it to an `ObjectId`. The OMG C++ mapping for the POA contains special conversion functions, *e.g.*, `string_to_ObjectId`, for just this purpose.

As usual for `sequence` return values, the `sequence` is returned by pointer and the caller is responsible for deallocating it. Therefore, we assign our `ObjectId` pointer to an `ObjectId_var` to ensure automatic cleanup.

4. Register the servant with the POA explicitly: Finally, the fourth change is to explicitly activate the CORBA object. This is done by calling `activate_object_with_id` on the POA and passing it our `ObjectId` and a pointer to our servant. Internally, the POA stores the `ObjectId` and the servant pointer in its active object map, which is used for subsequent request demultiplexing and dispatch.

3.3 Creating a Persistent Object Reference

The revised program is missing one important step: the creation of the object reference for the persistent CORBA object. It is important to note that the `activate_object_with_id` operation does *not* create

[1] There is a different ObjectId type in the CORBA module–be careful not to confuse them.

an object reference; it only registers the servant into the POA and reactivates the CORBA object. Thus, in its original form, our program only provides a place in which the already-existing CORBA object can be reactivated. This leads to the question of exactly how this persistent CORBA object was created in the first place.

To create the object reference, let's change the code near our call to `activate_object_with_id` near comment (4) as follows:

```
PortableServer::ObjectId_var oid =
  string_to_ObjectId ("my_object");

// Check command-line arguments to see if
// we're creating the object or reactivating it.
if (argc == 2 && strcmp (argv[1], "create") == 0)
  {
    // Create an object reference.
    CORBA::Object_var obj =
    poa->create_reference_with_id (oid,
                                   "IDL:Null:1.0");

    // Stringify it and write it to stdout.
    CORBA::String_var s =
      orb->object_to_string (obj);
    cout << s.in () << endl;
  }
else
  {
    // Create a servant to service
    // client requests.
    Null_Servant_Impl servant;

    // Same reactivation code as before.
    poa->activate_object_with_id (oid, &servant);

    // Allow the POA to listen for requests.
    poa_mgr->activate ();

    // Run the ORB's event loop.
    orb->run ();
  }
```

This modification allows our server program to play the role of a *factory* program or a server depending on how it's invoked. If we invoke it with the command-line argument "create" it creates our CORBA object using the `POA::create_reference_with_id` operation. This operation takes two parameters: (1) the *object ID* for the new CORBA object and (2) a *repository ID* string that identifies the most-derived interface supported by the object.

Our repository ID `string`, `"IDL:Null:1.0"`, uses the standard OMG repository ID format to indicate that our CORBA object supports the `Null` interface, version 1.0[2], as its most-derived interface. The code then stringifies the new object reference, writes it to the standard output (perhaps to allow us feed it to another tool that enters it into the Naming service or a Trader service), and exits. Otherwise, it assumes the CORBA object was already created in the past, reactivates it using the `my_object` object id, and starts listening for requests.

With this approach, a separate administrative factory program to set up the object in the first place is not needed. This approach reduces the maintenance overhead of having to have our `ObjectId` specified in multiple source files, which would allow them to get out of sync. More importantly, having a single combined factory/server program eliminates the administrative burden of having to keep track of which factory program should be used with which server. We'll discuss these and other server administration issues in future columns.

[2]Note that the version number component of repository IDs is currently unused.

4 Concluding Remarks

This column described in detail two simple, yet functional, POA-based server programs. In the process, we've highlighted the differences between transient and persistent CORBA objects. Our example programs showed how POAs can be created with different policies to support transient and persistent CORBA object types. They also show the different forms of servant registration and activation required to support both kinds of CORBA objects.

Our next column will introduce how servant classes can be declared and implemented. This important aspect of our server wasn't presented here. We will also continue our presentation of POA policies by investigating the use of *servant managers* to dynamically load servants as requests arrive for them.

As always, if you have any questions about the material we covered in this column or in any previous ones, please email us at `object_connect@cs.wustl.edu`.

References

[1] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 ed., June 1997.

[2] D. C. Schmidt and S. Vinoski, "Object Interconnections: Object Adapters: Concepts and Terminology," *C++ Report*, vol. 11, November/December 1997.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[4] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 97-12-02 ed., Nov. 1997.

[5] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[6] D. Schmidt and S. Vinoski, "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, vol. 8, October 1996.