Object Interconnections

C++ Servant Managers for the Portable Object Adapter
(Column 14)

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 September 1998 issue of the SIGS C++ Report magazine.

1 Introduction

Welcome to the final installment of our series covering the new CORBA Portable Object Adapter (POA) specification. The POA allows servers to (1) generate and interpret object references, (2) activate and deactivate servants, (3) demultiplex requests to map object references onto their corresponding servants, and (4) collaborate with the automatically-generated IDL skeletons to invoke operations on servants. The POA is a component of CORBA visible solely to a server, *i.e.*, clients are not directly aware of the POA's existence or structure.

The first column in this series [1] discussed the concepts and terminology used in the POA specification. It described how the lifetime of a CORBA object can be decoupled from the lifetime of any C++ servant(s) that implement it. The second column [2] expanded this discussion by exploring the two lifespans that are possible for CORBA objects: *transient* and *persistent*. These lifespans differ entirely by how long a CORBA object "lives." Lifespan issues often arise when discussing object adapters since that's where the worlds of CORBA objects and programming language servants intersect. Our third column [3] of the series described how to create C++ servants and register them in order to incarnate CORBA objects.

For some applications, explicitly registering a servant for each CORBA object is prohibitively expensive; for others, it is virtually impossible. For instance, some applications contain many thousands of objects, with the state of each object kept separately in persistent storage. In such cases, it can be costly, in terms of memory resources, persistent storage access, and execution time, to create and register a servant for each CORBA object. There are also other types of applications that serve as gateways to other distributed systems, such as DCOM. When a new object is added to the DCOM system, the gateway needs to provide access to it dynamically, without needing to be shut down, recompiled, and restarted.

Both types of applications described above must activate objects *on demand* as requests are actually made on them, rather than activating them all before the ORB starts listening for requests. The POA specification provides several mechanisms, *i.e.*, servant managers and default servants, that allow applications with many objects to scale gracefully.

Servant managers are responsible for managing the association of an object (as characterized by its Object Id value) with a particular servant, and for determining whether an object exists or not. Default servants can process requests for an object if no other servant is available for it. In this column, we first describe servant managers and then we explain default servants. As always, we provide C++ examples that show the details of how each is used.

2 Overview of the POA Policy for Request Processing

The characteristics of each POA other than the Root POA are controlled at POA creation time using different *policies*.[1] One policy a POA can be created with is the `RequestProcessingPolicy`. This policy allows the application to control how a POA associates a servant with the target object for each request. As with all POA policy interfaces, the `RequestProcessingPolicy` is defined in the `PortableServer` module. For each POA, it has one of the following three possible values:

USE_ACTIVE_OBJECT_MAP_ONLY: A POA created with this policy is also required to have the `RETAIN` value for the `ServantRetentionPolicy`, which instructs it to maintain a table of associations between servants and the CORBA objects they implement. The table, called the *Active Object Map*, is indexed by the object identifier of type `ObjectId` (a `sequence` of `octet`) that either the application or the POA supplies when it creates an object reference. In addition, each POA has another policy, the `IdAssignmentPolicy`, that controls whether it creates `ObjectIds` or whether it expects the application to supply them.

[1] The policies of the Root POA are defined by the CORBA specification and cannot be changed by applications.

When a request is sent, the object key of the target object, of which the ObjectId is part, is sent with it to identify the target object. The receiving ORB uses the object key to identify the target POA. The POA then uses the ObjectId in the object key to index into the Active Object Map, obtain the servant for the target object, and dispatch the request to that servant. If a POA created with the USE_ACTIVE_OBJECT_MAP_ONLY policy value cannot locate the requested servant in its Active Object Map, it raises the standard CORBA::OBJECT_NOT_EXIST system exception. This exception indicates to the client that the target object no longer exists.

USE_SERVANT_MANAGER: A POA configured with the USE_SERVANT_MANAGER policy value relies on an application-supplied ServantManager object to supply object/servant associations. This ServantManager is used if (1) the POA does not find the appropriate servant in its Active Object Map or (2) if the NON_RETAIN value is present for the ServantRetentionPolicy.

ServantActivator and ServantLocator are the two standard interfaces derived from the ServantManager interface. One or the other is used depending upon the POA's value for the ServantRetentionPolicy. If the POA has the RETAIN value for servant retention, the POA expects its servant manager to supply the ServantActivator interface. Otherwise, the POA has the NON_RETAIN value, and it expects the ServantLocator interface. For either interface, the POA passes the ObjectId of the target object to the servant manager object, expecting it to either return a servant to incarnate the target object or raise an exception.

We'll explain the details of the ServantActivator and ServantLocator interfaces in Section 4.

USE_DEFAULT_SERVANT: With this policy value, if the servant is not found in the Active Object Map, or if the POA has the NON_RETAIN policy value, the POA invokes a single servant for all requests regardless of the ObjectId of the target object. This feature allows applications to supply servants for use with the Dynamic Skeleton Interface (DSI). DSI servants essentially provide a single generic invoke function that can be used to dispatch any request. A default servant based on static skeletons can also be used for a POA whose objects all support the same interface.

Now that we've introduced the features supported by the POA request processing mechanism, we'll apply these to our running stock quoter example in the next section.

3 Alternative Stock Quoter System

Our quoter system returns stock prices based on stock names via its get_quote operation:

```
module Stock
{
  interface Quoter
  {
    // Return the current value of <stock_name>.
    long get_quote (in string stock_name);
  };

  // ...
}
```

An alternative to this interface is to allow stocks to be manipulated via their own interface, rather than representing them only as string names. The following interface lets us manipulate stocks directly:

```
module StockTrading
{
  interface Stock
  {
    // Return the name of the stock.
    string name();

    // Return the current value of the stock.
    long value();
  };
};
```

Note that we've introduced a new module called StockTrading. This change allows us to use Stock as the name of our new interface, and to use our new solution alongside our previous approach without invalidating existing code. By eliminating the need for the Quoter interface, we can examine the state of a stock directly using the Stock interface, rather than asking a Quoter for information about a stock.

This approach may seem to raise new problems related to object discovery, however. Without something like a Quoter object, how can we obtain object references for the stocks we're interested in? Fortunately, we already have a solution that is far more flexible than the Quoter interface: the OMG Trading Service [4].

A Trader allows objects to advertise themselves using many more characteristics than just their names. For instance, a stock object could advertise itself via price, number of shares bought and sold, rating, or any other characteristic. In addition, clients can use Traders to perform very efficient lookups of groups of objects that all share similar characteristics.

Building this level of flexibility into a Quoter is hard and would just duplicate the functionality available from a Trader. Therefore, we assume for the rest of our examples that Trader functionality is available in our distributed system. Due to space limitations we don't show its usage, however.

One way for a server application to implement our Stock interface solution is to create a separate servant for each stock we supply information about. Here is our concrete servant class for the Stock interface:

```
class MyStock :
  public virtual POA_StockTrading::Stock
{
public:
  MyStock (const char *stock_name)
    : stock_name_ (stock_name) {}

  char *name (void)
    throw (CORBA::SystemException)
```

```
  {
    return CORBA::string_dup
      (stock_name_.c_str ());
  }

  CORBA::Long value (void)
    throw (CORBA::SystemException)
  {
    return stock_database_lookup (stock_name_);
  }

private:
  std::string stock_name_;
};
```

Our MyStock constructor takes the name of the stock and stores it into an ANSI C++ string. This name determines the stock for which the servant instance will supply information to clients. The name and value operations override the pure virtual methods inherited from the POA_StockTrading::Stock skeleton class. The name method returns the name of the stock that the servant was constructed with. The value operation performs a lookup in an external database to find the current value of the stock and return it.

Our server main uses this class to create and register a servant for each stock we expose to clients. Our server is fairly typical; it initializes the ORB, sets up the POAs that it needs, creates and registers its servants, and then starts listening for requests, as shown below:

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

  // Get our Root POA, create our persistent
  // POA as a child of the Root POA, and
  // activate the POA managers (not shown).

  // Create and register all of our servants.
  MyStock stock1 ("IONAY");

  PortableServer::ObjectId_var id1 =
    PortableServer::string_to_ObjectId ("IONAY");
  persistent_poa->activate_object_with_id
    (&stock1, id1);

  MyStock stock2 ("ACME");

  PortableServer::ObjectId_var id2 =
    PortableServer::string_to_ObjectId ("ACME");
  persistent_poa->activate_object_with_id
    (&stock2, id2);

  // Continue creating servants and activating
  // objects for each and every stock object. We
  // presume there are many thousands of stock
  // objects, so all the required code is omitted!

  // Export the object references for stock1,
  // stock2, etc. to a Naming Service or Trading
  // Service, etc.

  // Let the ORB listen for requests.
  orb->run ();

  return 0;
}
```

Note how our server main creates a servant for each of the stocks, IONAY and ACME. It then creates an object identifier for each servant and explicitly activates the objects using the POA::activate_object_with_id operation. Though we only show the creation, registration, and activation step for two objects due to space limitations, the intent is that many more stock objects need to be set up in the same fashion.

The statically configured approach shown above works reasonably well for a small number of servants whose implementations change infrequently. In reality, however, our stock server probably supplies information about many thousands of different stocks. Moreover, these implementations may change over time as new requirements or optimizations are incorporated. Thus, the statically configured approach shown above scales poorly as the number of stocks and number of servant implementations increases, for the following reasons:

Unnecessary code duplication: Our servant creation and object activation code fragment will be repeated for each stock. This type of duplication is tedious to write and error-prone to maintain manually.

Inefficient initialization overhead: The code required to create and activate the servants could literally take minutes to execute, meaning that the server could take quite awhile before it reaches the point where it can start listening for requests. With all those lines of servant initialization code, the executable program itself could be extremely large, as well.

Inefficient space utilization: A large amount of memory resources may be required to store all the statically configured servants. Moreover, we might use excessive machine resources for no reason since it is unlikely that all of our clients combined are actually accessing all the objects in our stock server.

Inflexible behavior: Changing the implementation of our servants in a statically configured CORBA server is hard. It requires modifying, recompiling, and relinking the existing server software, as well as terminating and restarting any running server processes.

To some extent, the code duplication and space utilization issues mentioned above could be addressed using loops and subroutines, of course, but the other issues remain. In order to alleviate these drawbacks in our stock server, we'll employ POA servant managers, as described in the next section.

4 Servant Managers

4.1 Overview

The POA specification allows server applications to register servant manager objects that activate servants on demand. This allows a server to avoid creating all of its servants before listening for requests. When combined with patterns like Service Configurator [5] and OS features like explicit dynamic linking [6], the POA servant managers make it possible to dynamically configure servants into CORBA servers.

Existing ORBs support dynamic configuration of servants. For instance, Orbix *Loaders* essentially allow objects to be loaded into a server on demand when requests arrive for them. In fact, Orbix loaders were one of the primary influences that led to servant managers being supported as part of the POA specification.

The lifecycle of a request is shown in Figure 1. As shown

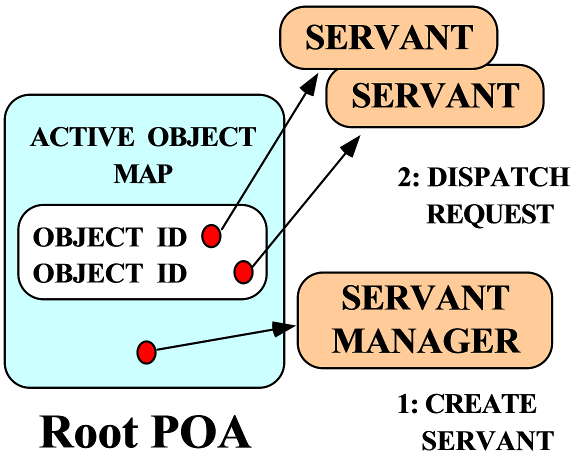

Figure 1: Lifecycle of a CORBA Request Through a POA

in this figure, when a request is sent to a CORBA object, the POA containing the target object invokes the servant manager to obtain a servant, then dispatches the request to that servant.

The servant manager object is supplied by the application. Therefore, the application can determine the strategy for creating and registering servants just for those objects that are actually the targets of requests. Moreover, servant managers are objects, which means that they themselves are also incarnated by servants. This is a result of servant managers being specified in IDL – they're handled and implemented just like regular CORBA objects, with the single restriction that they are *locality constrained*.

According to the CORBA specification, locality constrained objects must throw `CORBA::MARSHAL` exceptions if an application attempts to pass their object references out of the context of the ORB in which they were created. Thus, they cannot be passed as arguments to remote operations or be converted into strings via `ORB::object_to_string`.

The remainder of this section describes the roles of the two key IDL interfaces related to servant managers: *servant activators* and *servant locators*.

4.2 Servant Activators

If our POA was created with the `RETAIN` value for the `ServantRetentionPolicy`, our servant manager must fulfill the `ServantActivator` interface. A `ServantActivator` for our stock application can be defined as follows:

```
class MyStockActivator : public virtual
  POA_PortableServer::ServantActivator
{
public:
  PortableServer::Servant incarnate
    (const PortableServer::ObjectId &id,
     PortableServer::POA_ptr poa)
      throw(CORBA::SystemException,
            PortableServer::ForwardRequest);

  void etherealize
    (const PortableServer::ObjectId &id,
     PortableServer::POA_ptr poa,
     PortableServer::Servant the_servant,
     CORBA::Boolean cleanup_in_progress,
     CORBA::Boolean remaining_activations)
    throw(CORBA::SystemException);
};
```

The `incarnate` and `etherealize` methods are inherited from the `ServantActivator` skeleton class in the `POA_PortableServer` namespace. When the POA needs a servant to incarnate the target object, it invokes the `incarnate` operation. Likewise, when it needs to deactivate an object or destroy a servant, it invokes `etherealize`. These methods are shown below.

4.2.1 Implementing the incarnate() Method

Our implementation of `incarnate` supplies a `MyStock` servant:

```
PortableServer::Servant
MyStockActivator::incarnate
  (const PortableServer::ObjectId &id,
   PortableServer::POA_ptr poa)
    throw (CORBA::SystemException,
           PortableServer::ForwardRequest)
{
  CORBA::String_var str =
    PortableServer::ObjectId_to_string (id);

  CORBA::Long val =
    stock_database_lookup (str);

  if (val == -1)
    throw CORBA::OBJECT_NOT_EXIST ();

  return new MyStock (str);
}
```

Our implementation of `incarnate` is straightforward. It first uses the `ObjectId_to_string` helper method supplied in the `PortableServer` module to convert the `ObjectId` of the target object to a string. It then uses this string to look up the current stock price in its external database. If no such stock is found, the stock lookup returns -1 and the `incarnate` method throws a standard `CORBA::OBJECT_NOT_EXIST` exception to indicate that the target object does not exist. Otherwise, `incarnate` allocates a `MyStock` servant on the heap, passing the name of the stock it is incarnating to its constructor, and returns it to the POA.

Note that the `incarnate` method provides an ideal hook where the implementation of a `MyStock` instance could be brought into the address space of the server via dynamic linking. Also note that one of the exceptions that the incarnate method can raise is the ForwardRequest exception, which is described in the sidebar.

Sidebar on ForwardRequest: The ForwardRequest exception allows the servant manager to inform the client ORB to redirect this request, and any future requests for this object, to another object. The definition of ForwardRequest is shown below.

```
module PortableServer
{
  exception ForwardRequest
  {
    Object forward_reference;
  };
  // ...
};
```

A servant manager can assign a different target object to the forward_reference member of ForwardRequest, and raise the exception back to the POA. If the ORB uses IIOP as its underlying transport, it will turn the ForwardRequest exception into a LOCATE FORWARD response. This response causes the client ORB to transparently attempt to bind to the forward_reference object and reissue the request to it. This scheme is designed to work even if the new object is in a different server than the original target object.

Though our examples do not make use of the ForwardRequest feature, it is very useful for applications that want to perform their own server process activation or load balancing.

4.2.2 Implementing the etherealize() Method

The etherealize method is invoked when the object is deactivated or when the entire POA is deactivated or destroyed. Object deactivation typically occurs when an object is destroyed via one of its operations. For example, if the object's interface is derived from the standard OMG LifeCycleObject interface supplied by the Lifecycle Object Service, it inherits a remove operation that can be invoked to destroy the target object.

For our MyStockActivator implementation, the etherealize method is very simple:

```
void
MyStockActivator::etherealize
  (const PortableServer::ObjectId &id,
   PortableServer::POA_ptr poa,
   PortableServer::Servant servant,
   CORBA::Boolean cleanup_in_progress,
   CORBA::Boolean remaining_activations)
  throw (CORBA::SystemException)
{
  if (remaining_activations == 0)
    delete servant;
}
```

This method just ensures that the servant is not still in use for other request invocations by checking the value of the remaining_activations flag,[2] and if false, we delete the servant. If the cleanup_in_progress parameter (which we do not use here) is true, it indicates that the etherealize call was initiated by an invocation of the POA's deactivate or destroy operation. Applications might perform different etherealization actions for POA deactivation or destruction than they would for normal servant etherealization, such as releasing additional resources used by all servants in that POA.

4.2.3 Associating a Servant Manager With a POA

The POA supplies an operation that allows you to associate a servant manager with it:

```
module PortableServer
{
  exception WrongPolicy {};

  interface ServantManager {};

  interface POA
  {
    void set_servant_manager
      (in ServantManager mgr)
        raises (WrongPolicy);
    // ...
  };
};
```

The ServantActivator interface is derived from the ServantManager interface, which allows instances of it to be passed to the POA's set_servant_manager operation. To obtain an object reference for our ServantActivator, we need to create a servant for it and register it with a POA, just like any other object. The easiest way to do this is to implicitly create the object using the Root POA. Using the Root POA means that our stock activator object is a transient object, which is just fine because it is locality-constrained anyway.

```
MyStockActivator servant;

// Obtain an object reference for
// our stock servant activator.
PortableServer::ServantActivator_var
  activator = servant._this ();

// Associate the activator with this POA.
poa->set_servant_manager (activator);
```

This code uses the _this method to implicitly create a new CORBA object under the Root POA, register the servant for it, and obtain the new object reference. This object reference is then registered as the servant manager for the persistent POA containing the Stock objects. Note that even though the servant activator object itself resides in the Root POA, our stock servants are still registered with the persistent POA, which is a child of the Root POA.

4.3 Servant Locators

Because our POA has the RETAIN policy value, each servant returned by our ServantActivator is associated with its target object in the POA's Active Object Map. Thus, if the

[2] A servant might still be in use if it is registered multiple times under different object identifiers in a POA created with the MULTIPLE_ID value for the IdUniquenessPolicy.

server runs for a long time and many of our stock objects are the targets of invocations, our resource consumption could eventually become higher than we'd like. Specifically, the POA's Active Object Map could grow quite large, and all of our servants would be allocated on the heap.

To prevent the memory consumption problem described above, we can create the POA with the `NON_RETAIN` policy value for servant retention. In this case, the POA will not maintain object/servant associations in the Active Object Map. This type of POA expects our servant manager to inherit from `ServantLocator`, rather than `ServantActivator`.

`ServantLocators` have different semantics than `ServantActivators` since they are invoked for every request on an object. Like a `ServantActivator`, the `ServantLocator` returns the servant and the POA dispatches the request to it. Once the request completes, however, the POA returns the servant to the `ServantLocator`. Depending on the design of the application, the `ServantLocator` can then destroy it immediately or store it into an application-defined pool of servants to be reused for another request.

A `ServantLocator` for our `MyStock` servants might be defined as follows:

```
class MyStockLocator : public virtual
  POA_PortableServer::ServantLocator
{
public:
  PortableServer::Servant preinvoke
    (const PortableServer::ObjectId &id,
     PortableServer::POA_ptr poa,
     const char *operation,
     PortableServer::Cookie &cookie)
       throw (CORBA::SystemException,
              PortableServer::ForwardRequest);

  void postinvoke
    (const PortableServer::ObjectId &id,
     PortableServer::POA_ptr poa,
     const char *operation,
     PortableServer::Cookie cookie,
     PortableServer::Servant servant)
       throw (CORBA::SystemException);
};
```

The `preinvoke` and `postinvoke` methods are inherited from the `POA_PortableServer::ServantLocator` skeleton class. The POA invokes `preinvoke` before dispatching a request to obtain a servant, and invokes `postinvoke` after the servant completes the handling of the request.

Notice that the signatures for the `preinvoke` and `postinvoke` methods differ slightly from those for `incarnate` and `etherealize`. This is because the POA knows that the servant returned by a `ServantLocator` is used to process only a single request at a time, so it can pass additional information to it. This extra information consists of the following parameters:

Operation name: The `preinvoke` and `postinvoke` methods are given the name of the operation that is being invoked on the target CORBA object. This allows the `ServantLocator` to return a different servant depending on which operation is being invoked.

Cookie: An implementation of the `preinvoke` method is allowed to return a value to the POA, which will subsequently pass it back to the `postinvoke` method. This allows the `ServantLocator` to attach any kind of state it would like to each pair of `preinvoke`/`postinvoke` invocations. The `PortableServer::Cookie` type is simply a `void *`, allowing the application to use whatever it would like for a cookie.[3] The POA accepts the cookie from `preinvoke` and passes it unchanged to the matching `postinvoke`, *i.e.*, it doesn't try to access or otherwise interpret the cookie value.

4.3.1 Implementing the preinvoke() Method

Our implementation of `preinvoke` returns a `MyStock` servant:

```
PortableServer::Servant
MyStockActivator::preinvoke
  (const PortableServer::ObjectId &id,
   PortableServer::POA_ptr poa,
   const char *operation,
   PortableServer::Cookie &cookie)
     throw (CORBA::SystemException,
            PortableServer::ForwardRequest)
{
  CORBA::String_var str =
    PortableServer::ObjectId_to_string (id);

  CORBA::Long val =
    stock_database_lookup (str);

  if (val == -1)
    throw CORBA::OBJECT_NOT_EXIST ();

  return new MyStock (str);
}
```

This method is identical to the implementation of the `MyStockActivator::incarnate` method we showed in Section 4.3. In this simple example, we don't use the `poa`, `operation`, or `cookie` parameters.

4.3.2 Implementing the postinvoke() Method

Our implementation of `postinvoke` is also almost exactly like its `ServantActivator::etherealize` counterpart:

```
void MyStockActivator::postinvoke
  (const PortableServer::ObjectId &id,
   PortableServer::POA_ptr poa,
   const char *operation,
   PortableServer::Cookie cookie,
   PortableServer::Servant servant)
     throw (CORBA::SystemException)
{
  delete servant;
}
```

[3]This cookie is an example of the Asynchronous Completion Token pattern [7].

Just like preinvoke, we don't make use of the poa, operation, or cookie parameters. We simply delete the servant and return.

Finally, we create our ServantLocator object just like we created our ServantActivator:

```
MyStockLocator servant;

PortableServer::ServantLocator_var
  locator = servant._this ();
poa->set_servant_manager (locator);
```

The fact that ServantLocators allow developers to create servants on a per-request basis is useful for certain types of transactional applications. For instance, the preinvoke and postinvoke hooks allow persistent state to be loaded before an operation call on the servant and written back to the persistent store after the operation completes [8].

5 Default Servants

5.1 Overview

When all objects in a POA support the same interface, it is sometimes possible to support them all using only a single servant. This situation typically arises when the application uses the Dynamic Skeleton Interface (DSI). The DSI enables servants to supply a generic invoke function that allows any operation on any object to be invoked, regardless of its IDL interface. Using a default servant is also possible if static skeletons are used, as well, as long as all the objects incarnated by the default servant have the same interface.

5.2 Applying Default Servants to the Stock Quoter

All of our stock objects in our quote server support the Stock interface. Therefore, we can reduce memory consumption by using a single servant to support them all if we change our MyStock servant class slightly. Specifically, our constructor takes a parameter indicating the name of the stock serviced by the servant; the name operation simply returns that name. This implementation does not work for a default servant because it incarnates multiple objects simultaneously. Thus, for each request, the default servant must use the ObjectId of the target object to determine which object it is servicing the request for.

The PortableServer::Current interface enables servants to determine information concerning the identity of the target object. This interface is defined as follows:

```
module PortableServer
{
  interface Current : CORBA::Current
  {
    exception NoContext {};
    POA get_POA () raises (NoContext);
    ObjectId get_object_id () raises (NoContext);
  };
  // ...
};
```

Within the context of a request invocation, the get_POA operation returns a reference to the POA that is dispatching the request. Likewise, the get_object_id operation returns the ObjectId of the target object. Invoking these operations outside the context of a request invocation raises the NoContext exception.

The ORB's resolve_initial_references "bootstrapping" factory operation can be used by applications to obtain a reference to the PortableServer::Current. Thus, we can reimplement our MyStock servant as follows:

```
class MyStock :
  public virtual POA_StockTrading::Stock
{
public:
  MyStock (CORBA::ORB_ptr orb)
    : orb_ (CORBA::ORB::_duplicate (orb)) {}

  char *name (void)
    throw (CORBA::SystemException)
  {
    CORBA::String_var nm = get_target_name ();
    return nm._retn ();
  }

  CORBA::Long value (void)
    throw (CORBA::SystemException)
  {
    CORBA::String_var nm = get_target_name ();
    CORBA::Long val = stock_database_lookup (nm);
    if (val == -1)
      throw CORBA::OBJECT_NOT_EXIST ();
    return val;
  }

private:
  char *get_target_name (void)
  {
    CORBA::Object_var obj =
      orb_->resolve_initial_references
        ("POACurrent");
    PortableServer::Current_var cur =
      PortableServer::Current::_narrow (obj);

    PortableServer::ObjectId_var id =
      cur->get_object_id ();
    return PortableServer::ObjectId_to_string (id);
  }

  CORBA::ORB_var orb_;
};
```

We've changed our constructor to take a reference to the ORB as a parameter. We use this ORB reference in our private get_target_name method. This method returns the name of the stock that is the target of the current request.

Note that our get_target_name helper function assumes that the name of the stock is being used as the ObjectId for each object. Therefore, it can get the name of the target stock by getting a reference to the PortableServer::Current from the ORB, invoking its get_object_id method to get the ObjectId of the target object, and then using the ObjectId_to_string helper conversion method to convert the ID back into a string. The resulting string name is returned to the caller.

Our name and value methods use the get_target_name function to get the name of the target stock. The name method merely returns this name by invoking the String_var::_retn method to "steal" the string away from the `String_var` and return it to the caller.[4] The `value` operation uses the target name as the argument to the external `stock_database_lookup` method. If it returns -1, our servant throws the `CORBA::OBJECT_NOT_EXIST` exception to indicate that the stock no longer exists. Otherwise, it returns the value read from the external database.

To set up the default servant, we create a `MyStock` servant and pass it to the `POA::set_servant` operation, as follows:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

// Create our default servant.
MyStock servant (orb);

// Set the default servant on our POA.
poa->set_servant (&servant);
```

The POA will dispatch all incoming requests for our stock objects to our single default servant. This means that regardless of how many different stocks our application supports, all of them are implemented using only a single servant. An application using a default servant thus trades off the time required to look up the target object identity in the `PortableServer::Current` object against the space required for using multiple servants.

5.3 Object Creation Without Servants

Applications that use servant managers and default servants typically create their objects without actually incarnating them with servants. We can accomplish this using the `POA::create_reference_with_id` operation:

```
module PortableServer
{
  interface POA
  {
    Object create_reference_with_id
      (in ObjectId id, in CORBA::RepositoryId intf)
        raises (WrongPolicy);
    // ...
  };
};
```

The arguments to `create_reference_with_id` are the `ObjectId` for the new object, and the *repository ID* of the most-derived interface that our object will support. For POAs created with the `SYSTEM_ID` value for the `IdAssignmentPolicy`, you can alternatively invoke the `create_reference` operation to create an object reference with a POA-supplied `ObjectId`.

We can use `create_reference_with_id` to create all of our stock objects as follows:

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

  // Get our Root POA and create our persistent
  // POA as a child of the Root POA (not shown).
  // Activate the POA managers.

  // Set up our Stock interface repository ID.
  const char *rep_id = "IDL:StockTrading/Stock:1.0";

  // Create all of our stock objects.
  PortableServer::ObjectId_var obj_id =
    PortableServer::string_to_ObjectId ("IONAY");
  CORBA::Object_var obj =
    persistent_poa->create_reference_with_id
      (obj_id, rep_id);

  // Export the new object reference to a Naming
  // Service or Trading Service, etc.

  obj_id =
    PortableServer::string_to_ObjectId ("ACME");
  CORBA::Object_var obj =
    persistent_poa->create_reference_with_id
      (obj_id, rep_id);

  // Export the new object reference to a Naming
  // Service or Trading Service, etc.

  // Continue creating stock objects. We
  // presume there are many thousands of stock
  // objects.

  // Let the ORB listen for requests.
  orb->run ();

  return 0;
}
```

Our application `main` first creates all of our stock objects in our persistent POA without incarnating them, exports each to the Naming or Trading Service (not shown), and then listens for requests. As requests arrive, they are either processed by our servant manager or by our default servant, depending on which approach we're actually using.

Our example here is a bit misleading in that object creation would typically be a one-time event, rather than occuring each time we run this server. It might be best to use the approach of creating a server `main` that can either serve as a factory to initially create our objects or as a regular server, as we showed in [2]. We leave the creation of such a server `main` as an exercise for the reader.[5]

6 Conclusion

This column addressed yet another aspect of the creation and management of C++ servants using the POA. We introduced the POA concept of servant managers, which are objects that are used to activate objects and create and destroy servants on demand. In addition, we discussed default servants, which can process requests for an object if no other servant is available for it. An implementation of the POA specification that contains the types of servant managers described in this paper is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

Choosing whether to use a servant manager or a default servant for your application depends mainly on the number of objects your POA manages. The following are some general guidelines we recommend:

[4] This is a new feature of the CORBA 2.2 C++ mapping, so your ORB vendor may not yet support it.

[5] In other words, we're running over our page limits!

- If few objects are needed, use the `activate_object` or `activate_object_with_id` methods to explicitly activate them all. Though we did not discuss it here, it's even possible to explicitly activate a number of objects all with the same servant if the POA has a `IdUniquenessPolicy` value of `MULTIPLE_ID`, thus gaining some of the same benefits provided by default servants. This policy is yet another example of the POA's flexibility.

- If you have many objects, but expect few of them to be invoked in any given execution of the server process, consider using a `ServantActivator` with a `RETAIN` POA. This policy will maximize dispatching efficiency for each second and subsequent invocation on each object and minimize servant manager overhead. Alternatively, `ServantLocators` are useful when (1) the operation name can be used as a key to select a servant to dispatch to, (2) when the `Cookie` parameter is needed for post-request cleanup, or (3) when an application would rather manage its own pool of servants instead of using the POA Active Object Map.

- Default servants are the best way to use the DSI. Moreover, as our `Stock` application explained here shows, they minimize servant creation overhead and memory usage if all the objects contained by a POA implement the same IDL interface, even if you're using static skeletons.

This column concludes our series presenting the new Portable Object Adapter. We covered the basics of writing servants, and creating and activating CORBA objects. In addition, we described various ways for applications to supply object/servant associations, such as explicit registration and using servant managers.

The POA supplies a very rich set of features. Therefore, we couldn't cover them all. For instance, we hardly described the `POAManager` interface, which lets applications control the flow of requests into individual POAs or groups of POAs. Likewise, we didn't cover adapter activators, which let applications activate POAs themselves on demand. There are also details related to multi-threaded applications and POA/application interactions that any serious POA user needs to understand. You may want to get the POA specification for yourself from `www.omg.org` to read about these and other features we did not have the opportunity to explain.

In our next column, we'll start explaining some of the new *asynchronous messaging* features recently added to CORBA. 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`.

Thanks to Ron Witham and Irfan Pyarali for comments on this column.

References

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

[2] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 12, April 1998.

[3] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, vol. 12, June 1998.

[4] Object Management Group, *Trading ObjectService Specification*, 1.0 ed., Mar. 1997.

[5] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[6] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[7] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[8] S. Vinoski and M. Henning, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1998.