Below is an article I posted to comp.object.corba on 25 Sep 1997.
It explains how CORBA clients bootstrap, and how a client manages to find
the correct implementation of a CORBA object from the information
contained in an object reference.

From michi@foxtail.dstc.edu.au Thu Sep 25 08:34:27 1997
Newsgroups: comp.object.corba
Date: Thu, 25 Sep 1997 08:34:24 +1000
From: Michi Henning <michi@foxtail.dstc.edu.au>
Reply-To: Michi Henning <michi@foxtail.dstc.edu.au>
Subject: Re: Orbix -> other Orb IMPOSSIBLE?
In-Reply-To: <342812E4.910D327D@slb.com>
Message-ID: <Pine.OSF.3.95.970924145741.31000I-100000@foxtail.dstc.edu.au>
References: <342651B7.1470C49E@router.mgm-edv.de> <342812E4.910D327D@slb.com>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
Status: RO
X-Status:

On Tue, 23 Sep 1997, Hrishikesh Dixit wrote:

> The other way (which also, apparently, is one of the 'politically
> correct' ways to do IIOP), is publishing IORs (Interoperable Object
> References) for server objects in well-known locations, from where the
> clients can access them. The two most common ways of doing this are :
> 1 Using a Naming Service (such as OrbixNames) (you can refer to the
>   OrbixNames documentation for this)
> 2 'stringifying' the object reference and saving the string to a
> public file which the client can access.

I would like to clarify a few things here. There are three distinct
issues involved in this thread.

1) How does a client get an IOR so it can start making calls?

2) How is the object reference bound to a physical server process
   at the correct location once the client has obtained the
   reference?

3) How is a server activated automatically when a client makes
   a call and the server is not running?

These are three completely distinct and unrelated issues. The confusion
usually arises because the proprietary _bind() call deals with parts
of all three.

Issue 1:

This issue deals the with chicken-and-egg problem of how to bootstrap.
A client can obtain references by making calls, but to make a call, it
needs a reference.

CORBA has several answers to this problem.

- string_to_object

  The client reads a stringified object reference from some input
  source (file, email, smoke signals, morse code) and turns it
  back into an active reference.

- resolve_initial_references

  This call delivers references to a small number of well-known
  services to the client. These services can be used to obtain
  more references. The services that are currently well-known
  are naming, trading, interface repository, and transactions
  (this is from memory - if a service is missing from this list,
  someone please let me know).

  The particular reference returned for each well-known service
  is an ORB configuration issue. For example, which particular
  IOR to return for the naming service is configured through

an administrative interface (command-line tool, or text editor, or registry editor). Quite often, resolve_initial_references simply reads a stringified IOR out of a well-known file for each service.

- Lookup in some object reference store house, such as a naming service or trading service. These services are no more than big stashes of object references. The references can be retrieved by supplying some key, such as a name or a query.

In all cases, the client simply gets an IOR through some defined API. The IORs returned from string_to_object are exactly identical to the IORs returned by resolve_initial_references or a trader lookup. In other words, there is nothing magical about them, and IIOP has nothing to do with how these references are obtained.

Issue 2:

Once a client has an IOR (from anywhere) it must be able to somehow connect to the correct server which implements the corresponding object. Given that object references can propagate freely (for example as strings inside email), it is clear that a stringified IOR must have some form of addressing information inside it. However, this information is in *all* IORs, whether they are stringified or not. In other words, a stringified IOR is just a printable version of information that is naturally part of *every* IOR.

What is the addressing information inside an IOR?

There are two kinds of IOR, transient ones and persistent ones.

Transient IORs:

A transient IOR is one that works only for as long as the corresponding server process stays up. If the server process shuts down, the IOR no longer works, and calls via it will simply fail (they won't re-activate the server). Moreover, if the server is started up again, the transient IOR must be assumed to still be broken. In other words, a transient IOR works only for as long as its server is running. If the server ever shuts down and starts up again, the IOR must be assumed to never work again thereafter. In practice, it may work again occasionally, but that is a side-effect of the implementation - a high-quality ORB will actively take steps to permanently disable transient references forever.

What is the addressing information inside a transient IOR? Simple. It is the address and port number at which the server was running when it created the IOR. In other words, when a transient server creates an IOR, it writes its own address and port number into the IOR. This explains why transient IORs usually stop working once the server shuts down - if the server is restarted, it usually ends up getting a different port number, so the previously created IOR now contains stale information and will not longer bind.

A high quality ORB will additionally write some pseudo-random id into a different part of the IOR. The random id is assigned once, when the transient server starts up, and thereafter is embedded in all transient references created by that server. This mechanism is used to make sure that a transient reference stays dead forever, even if the server is restarted and happens to get the same port number. The ORB won't bind the transient reference unless the id it carries matches the id of the server. This never happens, because every time the server starts up, it gets a brand-new unique id (typically a UUID is used for this).

When a client uses a transient IOR, it blindly connects to the address and port number inside the IOR:

- Either the server is still running at that place, in which case everything is fine.

- Or the server has shut down, in which case nothing is listening at that port, and the call fails.

- Or another server is now running at the same port number as the original server. In that case, the client ends up sending

the request to the wrong server. Again, the server id is used to catch this - if a server gets a request that contains the wrong server id, it rejects it.

- Or the original server was stopped and restarted, and happened to get the same original port number. In this case, a quality ORB will have assigned a new id to the server on startup. In other words, the second incarnation of the server process is considered a different server from the first incarnation, and the request fails to bind.

Persistent IORs:

Having only transient IORs is very limiting, so we'd like to have persistent ones as well. A persistent IOR continues to identify the same object while that object conceptually exists as a CORBA object. The IOR continues to identify that same object regardless of how many times the server is shut down and restarted.

A persistent IOR stops working only when someone decides to delete the conceptual *CORBA* object. Deleting the *CORBA* object is quite different from deleting its implementation - the implementation object will be created and destroyed many times (every time the server starts up and shuts down again). The CORBA object has a life cycle quite independent from that. As an example, a person object will be destroyed every time the server shuts down, but that does not mean that the person has died.

A persistent IOR does *not* imply that the server will be started automatically when a request is sent (automatic server activation is an orthogonal concept, covered in issue 3).

A persistent IOR does *not* imply that the server must always start up at the same port number, or even the same machine.

So, how can an IOR be persistent if the server keeps starting up at different port numbers or on different machines?

The answer is that to create a persistent reference, several things must happen:

1) Each server must use a name to uniquely identify itself. That name is written into the (proprietary) part of each IOR. The name is known as the server name or server marker.

2) When creating a reference, the server must supply a name that uniquely (within that server) identifies the object belonging to the reference. That name is written into the (proprietary) part of the IOR, and known as the object name or object key.

3) The machine on which the server runs must be configured with the address of an implementation repository. "Implementation Repository" is OMG-speak - in Orbix, it is implemented by orbixd, in ORB Plus, it is implemented by obj_locator. Every time a server creates a persistent reference it writes the address of the *implementation repository* into a public part of the reference (instead of its own addressing information).

We now have three key pieces of information embedded in the IOR:

1) The address and port number of the implementation repository.

2) The name of the server.

3) The name of the object.

The implementation repository need not run on the same machine as the server, it can be anywhere.

Now, consider what happens when a server that wants to create persistent references starts up.

- The server knows its own server name.

- Every time the server starts up, it may start on a different machine and/or port number from last time.

- Sometime during initialization, the server-side run-time contacts the implementation repository. It does that to announce the presence of the server to the implementation repository. In effect, the server contacts the implementation repository and says "Hi, my name is fred, my machine is thismachine.acme.com, and my port number is 1239."

  The server picks up the address and port number of the implementation repository from the local ORB configuration, so it always knows where to reach the implementation repository.

- The implementation repository stores the server name, machine name, and port number in some data structure.

If the server shuts down and starts up a second time, it again contacts the implementation repository and announces its latest address details. The point is that the implementation repository knows:

- which servers are running

- the name of each server

- the machine name of each server

- the current port number of each server

There are various mechanisms to recover from nasty scenarios, such as crashed servers or loss of connectivity - the details don't matter for this discussion.

Now suppose a client has a persistent IOR (possibly destringified from a piece of email). How does the client bind on the first request?

- The client run-time looks at the address inside the IOR. That address is the machine name and port number of the implementation repository.

- The client run-time blindly sends the request to that address.

- Of course, what is running there is not the actual server, but the implementation repository, which can't handle the request.

Next, the implementation repository receives the client's request. Because the implementation repository is provided by the same ORB as the *server-side* run-time, it understands how the private part of the IOR is encoded (because the IOR was created by the server). The sequence of events now is:

- The implementation repository breaks open the private part of the IOR and looks up the server name.

- It uses the server name to look in its internal data structures to see if the server is running.

- If the lookup fails, the server is down, and the implementation repository returns a negative reply to the client, which propagates up to the client application code as a TRANSIENT exception (the request couldn't be bound).

- If the lookup succeeds, the implementation repository picks up the current machine name and port number from its internal data structure, and returns those details to the client in a LOCATION_FORWARD message.

Now the client receives the LOCATION_FORWARD. This in effect tells the client

"You sent the request to the wrong place so I can't help you, but I suggest you try again at the following machine name and port number."

Now the client-side run-time starts the game again, trying at the forwarding address next. With any luck, the correct server hasn't crashed and is running, so the client sends the request to the correct place.

Notice what happened here - the server name (a logical name independent of machine name or port number) was used by the implementation repository to tell the client how to find the server. In other words, the implementation repository knows how to relate the server name to the correct process at the correct port on the correct machine. The implementation repository knows *nothing* about which objects are implemented by the server, or whether objects exist or not.

Now the request arrives in the server. With the request, the client-side run-time sends the private part of the IOR. The server now breaks open that private part (it knows how to do this because it created that IOR) and looks for the object name. This name is now used as an index into a server-side data structure known as the running object table. This data structure contains a mapping from object names to virtual memory addresses of the corresponding C++ proxy objects.

The server side run-time uses this table to figure out which object in the server the request is for. Of course, this means that when you use persistent objects, you must make sure that each C++ instance uses the same object name to register itself with as it did last time. Typically, the object name is therefore some unique piece of state in the actual object - something like a database row identifier or social security number, or whatever is appropriate.

Notice that the server name is used by the implementation repository to tell the client where to find the server, and the object name is used to figure out which particular object inside the server the request is for.

Now finally, the request from the client ends up in the right object. If the object isn't in memory at the time the request arrives, you can arrange for a callback from the server-side run-time back into your code (this is what is called a Loader in Orbix, and an Activator in ORB Plus). This gives you a chance to load objects on demand, instead of having them in memory all the time.

If the object isn't in the server's memory (and can't be activated on demand), the server returns an OBJECT_NOT_EXIST exception to the client. Because the server was involved, this is an authoritative answer - the object simply does not exist. Contrast this with failure of the implementation repository to find a running server, which raises TRANSIENT. TRANSIENT means that the server wasn't running and could not be started. However, if the client tries again in five minutes, someone may have started the server in the mean time, and the request may work again. In other words, TRANSIENT is not an authoritative answer that an object doesn't exist. Instead, it simply indicates a failure to reach the server.

Notice how state is distributed here. The implementation repository only knows about server processes, and knows nothing about the individual objects. Each server, on the other hand, only knows about its own objects and its own implementation repository. This is deliberate - state is distributed such that it is unlikely to pile up in any one place and cause scalability problems.

If you read the OMG docs, there is an unfortunate naming clash in them. The specs talk about "persistent servers". Unfortunately, a "persistent server" does not mean that it deals with persistent objects. Instead, it refers to a server that is started by hand (instead of being started by the implementation repository). This means that persistent servers have nothing to do with persistent references or persistent objects. In fact, a persistent server may exclusively create transient references.

Issue 3:

How are servers activated on demand?

Given the answers to issue 2, this isn't hard to figure out. Again, the implementation repository is involved. You can register a server for automatic activation with the implementation repository (this corresponds to the Orbix putit command, where you specify the server startup command line). In effect, the implementation repository maintains

a table which contains the server name as the key, and the command line
required to start the server if it isn't running already.

Now just replay the scenario for binding a persistent reference. Again,
the client blindly sends the request to the implementation repository.
The repository looks at the server name, finds that the server isn't
running, and then forks/execs using the command line that was registered
with putit (or obj_loc_admin for ORB Plus). The implementation repository
then waits for the server to send it a message saying

> "Hey, my name is fred, my machine is thismachine.acme.com,
> and my port number is 1888. And by the way, I'm ready to
> accept requests now."

If you have used the -port option, the implementation repository arranges
for the server to connect to the nominated port. However, all this works
even if the server comes up on a random port, because the server tells
the implementation repository its own port number when it is ready to
accept requests.

At that point, the implementation repository returns the address of
the server to the client in a LOCATION_FORWARD as before. Note that
the client request is transparently delayed until the server is ready,
which is handy.

The point is that IIOP does not know anything about automatic server
activation at all. All that happens is that the client contacts the
implementation repository, and the server is started transparently.
The same LOCATION_FORWARD message is used after server startup is complete
to inform the client of the current address.

If you read all of the above carefully, you will see that each issue
deals with a separate and orthogonal piece of functionality. There are
also some very interesting consequences of the above. For example,
with the scenario I outlined, a server can hop around from machine to
machine, and can change port number every time it starts up *provided
that all the machines are configured to use the same implementation
repository*.

I hope the above explanations will help to clarify this a bit.

Disclaimer:

The above is grossly simplified to keep it short. There are
many many variations to the above overall scheme. All sorts of clever
optimizations and caching techniques can be used to make things more
efficient (mainly to avoid the implementation repository becoming a
bottleneck).

Also, there are many ways to skin the persistent object cat. For example,
an implementation repository can choose to know about individual objects,
or groups of objects. This provides greater flexibility for object
migration, but reduces scalability.

IIOP has some message types (LOCATE_REQUEST and LOCATE_REPLY) to make
binding more efficient in certain circumstances, and there are literally
dozens of other things involved that I simply didn't touch on.

The above is meant to be a keep-it-simple explanation of the basic
ideas, no more. If you tell me that there is lots of stuff that I forgot,
simplified, brushed under the table, or that different ORBs do this
differently - you are right, and I know ;-)

Cheers,

Michi.



--
Michi Henning  +61 7 33654310
DSTC Pty Ltd  +61 7 33654311 (fax)
University of Qld 4072  michi@dstc.edu.au
AUSTRALIA  http://www.dstc.edu.au/BDU/staff/michi-henning.html