Binding, Migration, and Scalability in CORBA

Michi Henning
michi@dstc.edu.au
CRC for Distributed Systems Technology (DSTC)
University of Queensland, Qld 4072, Australia

Abstract

This article explains how CORBA binds requests to object implementations with the help of an implementation repository. The design of the implementation repository has profound influence on the flexibility, performance, scalability, and fault tolerance of an ORB, and we illustrate some of the trade-offs involved in various repository designs. Implementation repositories play an important role in building scalable object systems; we point out some issues that CORBA will need to address in the future to continue to scale and remain at the forefront of distributed object technology.

One Repository to register them all, One Repository to find them,
One Repository to start them all, and with IIOP bind them
In the land of CORBA where the Objects lie.

1 INTRODUCTION

CORBA's object model [4] relies to a large degree on the semantics of *object references*. An object reference uniquely identifies a local or remote object instance—clients can only invoke an operation on an object if they hold a reference to the object. Object references can be converted into a string and transmitted via arbitrary means (such as e-mail or even smoke signals). The receiver of a string can convert it back into an object reference and invoke operations on the object.

Apart from the ability to denote remote objects and to convert to and from strings, CORBA references have semantics that are very similar to those of C++ class pointers: references can dangle (point at a nonexistent or unreachable object) and they can be nil (contain a special value that indicates a reference pointing nowhere).

Object references are opaque. Application code is not allowed to look inside a reference or to make any assumptions about its internals. The details of how an object reference identifies an object and how requests are dispatched to their correct destination are carefully hidden away in the ORB's run-time support. The opacity of references provides important transparencies to applications, such as language, transport, and protocol independence. These transparencies in turn are the foundation of CORBA's interoperability in heterogeneous networks [11].

The semantics of object references have profound implications on the design of an ORB, in particular its ability to support object migration and to scale to very large numbers (millions) of objects. This article explains some of the mechanisms and design trade-offs involved, and speculates how CORBA should evolve to meet the need for ever larger distributed systems.

2 ANATOMY OF AN OBJECT REFERENCE

Conceptually, an *Interoperable Object Reference* (*IOR*) has the structure shown in Figure 1 (some details are omitted since they are not relevant to this discussion):

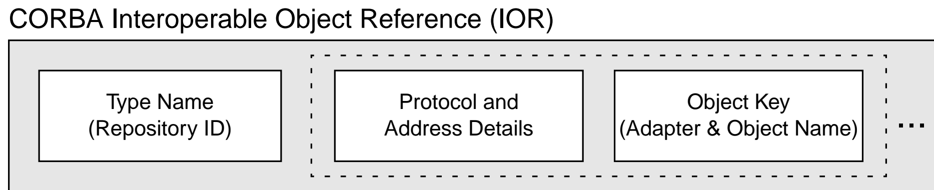

Figure 1: Major components of an Interoperable Object Reference (IOR).

An IOR contains three items of information:

- Type Name

  The type name, also known as a *repository ID*, identifies the most derived type of the object associated with the IOR at the time the IOR was created. The repository ID serves as an index into the *Interface Repository (IFR)*, which allows the IDL definition for an interface to be retrieved at run time. The repository ID can also be used by an ORB to implement run-time type checking.

- Protocol and Address Details

  This field specifies a protocol and the addressing information appropriate for that protocol. In the case of the *Interoperable Inter-ORB Protocol (IIOP)*, the addressing information consists of a host name and a TCP port number.

- Object Key

  This information is an opaque piece of binary data, proprietary to the ORB that created the IOR. It is usually referred to as the *object key* because it identifies which particular object is pointed to by the IOR. The object key itself consists of two components: the *object adapter name* and the *object name*. The object adapter name identifies the particular object adapter in a server that accepts requests for this object. The object name identifies which particular object within the adapter is pointed to by the IOR.

CORBA permits a single reference to contain several pairs of protocol and object key—that way, a single object reference can support multiple protocols or contain different addresses for the same object (for example, to provide fault tolerance).

3 REQUEST BINDING

When a client invokes a request via an IOR, the ORB run time is responsible for sending the request to the correct *servant*. A servant is whatever construct is used by a specific programming language to represent a CORBA object. (If the implementation language is object-oriented, servants typically are programming language objects.) The process of associating a request with its servant is known as *binding*.

Binding is protocol-specific—the remainder of this article assumes that the Internet Inter-ORB Protocol (IIOP) is used. IIOP is a TCP/IP-specific instance of CORBA's *General Inter-ORB Protocol (GIOP)*. GIOP is not a complete protocol itself, but a specification for defining protocols such as IIOP. GIOP assumes that the underlying transport provides a reliable byte stream abstraction and is connection-oriented.

CORBA distinguishes between *transient* and *persistent* IORs:

- A transient IOR continues to work only for as long as its associated server process remains available. Once the server shuts down, a transient reference becomes permanently nonfunctional (it never works again, even if the server is restarted).
- A persistent IOR continues to denote the same CORBA object even if the server shuts down and is restarted, possibly on a different machine. Many ORBs can transparently start a server process when a client uses a persistent reference and shut the server down again after some period of idle time.

Whether an IOR is transient or persistent is determined at IOR creation time by the server application code. An ORB binds transient references differently from persistent references.

3.1 Binding of Transient References

Figure 2 illustrates binding of transient references.

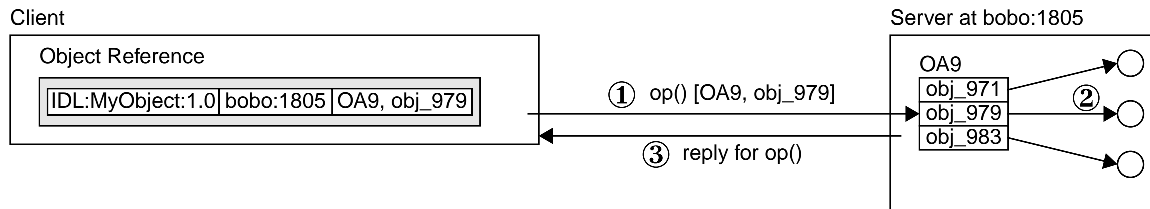

Figure 2: Binding of a transient reference. The server with ID OA9 runs on machine bobo at port 1805.

Binding of transient references relies on the server being available at the time a client invokes an operation:

1. The client-side ORB run time opens a connection to the host and port specified in the IOR and sends a *request* message to the server process (some ORBs send a *locate request* message instead—see Section 4). A request message contains:
   - the size of the message,
   - a unique *request identifier*,
   - the object key (which contains the adapter name and object name),
   - the name of the operation being invoked,
   - any **in** and **inout** parameters that need to be transmitted to the operation.

   The client-side run time then waits for a *reply* message to be returned on the connection on which it sent the request.
2. The server uses the adapter name and object name inside the object key to locate the correct servant for the request. The adapter name identifies a data structure known as the *active object map*. For each object adapter in a server, the active object map stores the object name and the memory address at which the corresponding servant can be found. The object name serves as an index into the active object map and therefore identifies the servant for a request.
3. The server sends a reply message back to the client. The reply message contains the request identifier sent in the corresponding request. This allows the client to associate a request with its reply so it can have several requests outstanding. A reply message also indicates success or failure. For success, the reply contains the results of the operation. For failure, it contains exception information that explains why the request failed.

Binding of transient references can encounter one of the following scenarios:

- The server is running at the host and port indicated by the reference.

  In this case, binding succeeds. The server uses the object key to determine which servant should handle the request and eventually returns a reply to the client.

- No process is listening at the specified address.

  No connection can be opened and the client-side run time raises an OBJECT_NOT_EXIST exception in the application code.

- The original server that created the reference has shut down and a different server is now listening on the same host and port.

  In this case, the client sends the request to the wrong server, so the request must fail with OBJECT_NOT_EXIST.

- The original server that created the reference has shut down, but was restarted again later and happened to get the same port number.

  In this case, the client sends the request to the correct server. However, because the reference is transient, the request must fail with OBJECT_NOT_EXIST.

The last two scenarios require cooperation from the receiving server. There are two relevant cases:

- The IOR contains an object key created by another vendor's ORB.
- The IOR contains an object key created by the same vendor's ORB (possibly by a previous instance of the same server process).

Both cases are dealt with by ensuring that object keys are unique for all transient IORs:

- IORs can carry a *vendor-specific tag*. The server examines the tag to see if a request is made via another ORB's IOR. If so, the IOR does not identify an object in this server. Vendor-specific tags are part of what is properly known as *multi-component profiles* [2]. The OMG maintains a registry of tags reserved for use by each vendor.
- When a server creates a transient reference, the ORB run time adds the server's *identity* to the object key by assigning a pseudo-random identifier to the adapter name (in Figure 2, the server uses the identifier OA9). The server identity can be any unique identifier, such as a *Universally Unique Identifier* (*UUID*) [7]. The ORB run time assigns a new identity to a server whenever the server starts up, so if the server receives a request that contains another server's identity, the request does not denote an object in this server (or denotes an object in a previous incarnation of the same server).

In either of the preceding cases, the server returns a reply containing an OBJECT_NOT_EXIST exception to the client and binding fails as it should.

3.2 The Implementation Repository

A *persistent* IOR survives server shut-down, that is, it continues to denote the same persistent CORBA object across server instantiations. A persistent object has a life cycle quite independent of that of its servant. The servant is created and destroyed whenever the server starts and shuts down, respectively, whereas the CORBA object denoted by a reference continues to exist until it is explicitly deleted.

Persistent IORs need to work even if the server starts on different hosts at different times, or if the server's port number is dynamically assigned. This means that, for persistent IORs, embedding the host name and port number of the server into the IOR is not an option—the IOR would stop working as soon as the server moved to a different port or host.

To bind persistent IORs, ORBs provide an *implementation repository.*

3.2.1 Standards Conformance of the Implementation Repository

We need to note here that the CORBA specification [2] does not standardize the implementation repository and only suggests a few functions that vendors may choose to implement. The reasons are:

- The implementation repository is intimately related to its underlying platform. For example, it needs to deal with details such as process creation and termination, threads, and signals. This makes implementation repositories nonportable.
- The CORBA specification permits ORB implementations for environments ranging from embedded systems to global enterprise systems. Implementation repositories need to provide functionality that must be tailored for each environment, and it is not feasible to write a specification that covers all possible environments.
- Object migration, scalability, performance, and fault tolerance all depend on the implementation repository. The repository therefore provides a major point at which vendors can add value to their ORB.

Despite the lack of standardization, interoperability between different vendors' ORBs is still guaranteed. CORBA strictly specifies how an implementation repository interacts with *clients*, so a client using vendor A's ORB can interoperate with an implementation repository from vendor B. Proprietary mechanisms only exist between *servers* and their respective implementation repositories. This means that a server written for vendor A's ORB requires an implementation repository from the same vendor. However, the interactions between servers and their repositories are not visible to other clients and servers and so do not compromise interoperability.

The explanations that follow describe the implementation repository and binding process as currently implemented (with minor variations) by a number of commercial ORBs.

3.2.2 Implementation Repository and Server Interactions

The implementation repository has several responsibilities:

- It maintains a registry of known servers.
- It records which server is currently running at what host and port number.
- It starts servers on demand if they are registered for automatic activation.

The implementation repository is typically implemented as a process that runs at a fixed address (host name and port number). If a server creates persistent IORs, the server's host must be configured with the address of the implementation repository. Hosts that are configured to use the same implementation repository are said to be in a single *location domain* (this implies that the implementation repository need not run on the same host as a CORBA server).

The implementation repository maintains a data structure such as the following table:

Adapter Name	Start-up Command	Address
Fred	rsh bobo "/usr/local/bin/fred -x"	bobo:1799
Joe	/usr/local/bin/joe	
Mike		foxtail:3333

The adapter name and command columns of the table are populated by an administrative tool when a server is installed.

The adapter name column identifies the object adapter in the server that processes requests. Servers still using the (now deprecated) *Basic Object Adapter (BOA)* have exactly one BOA. Servers using the *Portable Object Adapter (POA)* [2][8][9][10], may have several POAs (see Section 6.1).

The start-up command records how the server can be started on demand when a client attempts to contact an object provided by the corresponding object adapter. Note that the preceding example relies on the remote shell to start a server remotely. This is for illustration purposes only—instead of **rsh**, some other,

more secure mechanism can be used equally well. If no start-up command is recorded for a server, this indicates that the server is expected to be started manually (and will probably run continuously).

The address field records the host and port number at which a server is currently running (an empty address field indicates that the server is stopped).

A server that wants to create persistent references must be registered with the implementation repository. Sometime during start-up (typically just before the event loop is entered), the server sends its adapter name and its host and port number to the implementation repository. (The server knows how to find the implementation repository from its local machine configuration.) If the server's object adapter is not registered, the implementation repository returns an exception that is propagated back to the client application code (usually as a BAD_PARAM or OBJ_ADAPTER exception).

With this mechanism, the implementation repository always knows the latest address details of each running server, and only registered servers can create persistent references (servers that only generate transient references need not be registered with the repository). Most implementation repositories also include mechanisms to recover from failures, such as detecting a crashed server. However, the details are not relevant to this discussion.

The communication between the server and its repository can use any proprietary protocol it likes. In practice, however, servers contact the repository on a normal IDL interface via IIOP.

3.3 Binding of Persistent References

Persistent references created by a server contain the following information:

- the repository ID of the most derived interface (as for transient IORs),
- the host name and port number of the implementation repository,
- the object adapter name (embedded in the object key),
- the object name (also embedded in the object key).

For persistent references, the adapter name, instead of being a random identifier, is controlled by the server application code. The server code uses a fixed adapter name for each of its objects. The object name is also supplied by the server application code and associates the IOR with a particular servant within a particular adapter. The server application code registers each servant with the same adapter and object name it used for previous instantiations of that servant. The object name is a unique piece of state such as a database row identifier, social security number, or whatever else is suitable to provide object identity. The object name links a particular servant to the CORBA object represented by that servant and therefore supplies object identity. Note that persistent references contain an address that points at the implementation repository instead of the actual server.

To bind a persistent IOR, the client behaves exactly as for a transient one. It opens a connection to the address in the IOR and sends the request (or a locate request—see Section 4) via that connection. Of course, for persistent references, the connection leads to the implementation repository, which unpacks the object adapter name from the object key and uses it as an index into its server table:

- If the server is not registered, the repository returns an OBJECT_NOT_EXIST exception (which is propagated back to the client application code).
- If the server is registered for manual start-up but is not running, the repository returns a TRANSIENT exception.
- If the server is registered for start-up but is not running, the repository starts the server and waits for a message from the server that provides the server's address details.

- If the server is running (possibly after being started first), the repository returns a *location-forward* reply to the client.

A location-forward reply indicates to the client-side run time that it sent the request to the wrong process and should try again elsewhere. The forwarding location is returned with the reply in form of another IOR. The implementation repository constructs that IOR by extracting the address details of the server from its table.

The client-side run time (transparently to the application code) sends the request a second time via the IOR returned by the repository, and the request ends up at the correct server.

Once the request arrives in the server, binding is identical for transient and persistent references:

- The server verifies that the object adapter name in the IOR matches its own adapter name. If not, the server sends an OBJECT_NOT_EXIST reply.
- The server uses the object name as an index into its active object map. If the server can find a servant in the map, the request succeeds. If the server cannot find a servant, it replies with OBJECT_NOT_EXIST.

Figure 3 shows the sequence of interactions for the successful binding of a persistent reference with automatic server start-up:

1. The client invokes operation op, sending the request to the implementation repository.
2. The implementation repository starts the server.
3. The server informs the implementation repository of its current address details.
4. The repository returns the server address to the client in a location-forward reply.
5. The client re-issues the request to the new location.
6. The server uses the object key sent with the request to locate the servant.
7. The server returns the results of op to the client.

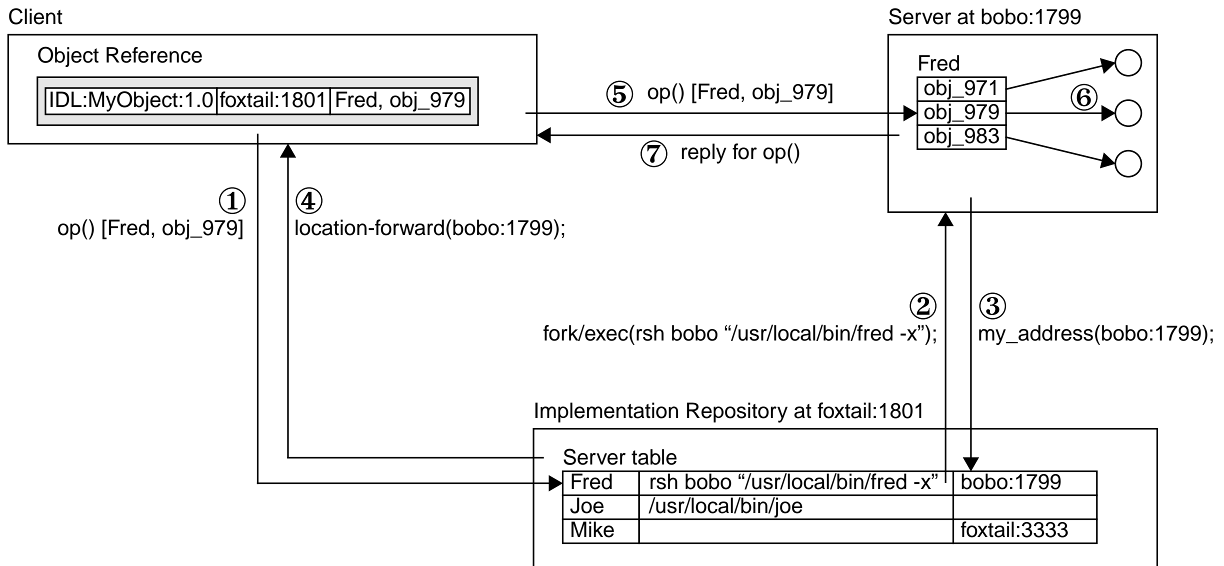

Figure 3: Binding of a persistent reference with automatic server start-up. The implementation repository runs on machine foxtail at port 1801, and server Fred runs on machine bobo at port 1799.

All messages to and from the client are normal IIOP requests. Message (3), in which the server informs the implementation repository of its location, can be sent using either IIOP or a proprietary protocol without compromising interoperability.

Note that the client knows nothing about the implementation repository or automatic server start-up. Instead, it simply waits for a reply to its request. The reply either returns the results of the operation, indicates an exception, or contains an IOR via which to retransmit the request. Of course, a retransmitted request may be answered with yet another location-forward reply. (GIOP places no limit on the number of times a request may be forwarded.) If a server needs to be started first, the client's request is transparently delayed until the server is ready.

Binding returns OBJECT_NOT_EXIST only either if the server is completely unknown to the repository, or if the server itself has determined that no implementation matches the object name. In other words, OBJECT_NOT_EXIST is an authoritative reply indicating that the object does not exist. In contrast, if the server cannot be reached, the repository replies with a TRANSIENT exception. TRANSIENT simply indicates a failure to reach the server and does not imply anything about object existence.

With the preceding design, the implementation repository knows about servers, and servers know about their own objects. The implementation repository does not accumulate information about each individual object, which makes it unlikely that state piles up in any one place and causes scalability problems.

The POA specification also permits a server application to arrange for a callback if the server-side run time cannot locate a servant in the active object map. This mechanism allows a server to bring objects into memory on demand instead of permanently having all servants in memory.

4 SCALABILITY AND PERFORMANCE

Clients need to contact the implementation repository to get the first request for an object to the correct process (once an IOR is bound, further requests do not involve the repository because the client already knows how to contact the object). The problem with this is two-fold:

- The additional overhead incurred by retransmission of requests following a location-forward reply can be noticeable, particularly if requests carry large amounts of data (all of the data needs to be retransmitted).
- There is an upper limit of the number of servers that can use the same repository. As the number of servers and objects grows, more client requests need to be forwarded.

CORBA provides some features to mitigate these problems:

- Instead of an actual operation invocation, the client-side run time can send a *locate request* to the repository. A locate request does not contain any parameter values, so the client can avoid redundant transmission of parameters. Of course, this optimization is worthwhile only for large requests.
- If the client-side run time encounters an IOR it has not yet bound, it can look at the vendor tag to determine whether the IOR was created by the same ORB as used by the client. If so, the client knows how the object key is encoded and can extract the object adapter name from the IOR. If the client has previously bound an IOR to the same server, it already knows how to contact that server and need not involve the implementation repository.

  This optimization works if client and server use the same ORB, but does not apply if the client holds an IOR created by a different ORB. This is because the client-side run time cannot decode the object key of another ORB's IOR.

- Some ORBs use the object key to cache the address at which the server originally created a reference (known as the *server birth address*). If the client holding an IOR uses the same ORB as the server, it can extract the server birth address from an IOR and attempt to connect to that address first. There is a chance that the server may still be running at the same host and port—if so, the client avoids contacting the implementation repository.
- Several implementation repositories can be used. This reduces the size of each location domain, and therefore the number of requests sent to the repository.

5 FAULT TOLERANCE

The binding algorithm shown in Section 3.3 relies on contacting the implementation repository. This creates a single-point-of-failure scenario: if the repository is down, none of the objects in the repository's location domain can be bound. The GIOP protocol offers a mechanism that can be used to mitigate the problem: instead of using a single address, an ORB can embed multiple addresses in an IOR, each of which denotes a redundant instance of the same repository. This can be used to provide fault tolerance as well as load-sharing to improve performance. However, commercial ORB implementations currently do not use the feature, and the specification does not make it clear how a client should treat multiple addresses during binding (the OMG has made initial moves to standardize fault tolerance [5]).

Another approach to making applications more robust against repository failure is to reduce the size of location domains. Instead of sharing a repository among a number of hosts, each host can run its own repository. If a host crashes, only objects implemented on that host become inaccessible. This approach also improves performance because each repository does less work.

6 OBJECT MIGRATION AND SCALABILITY

Reducing the size of location domains improves reliability and performance, but it also limits *object migration*. Object migration refers to the ability to move an object from one address space to another (change its physical location) without breaking references to that object currently held by clients.

Consider the binding scenario presented in Section 3.3. If several hosts share a repository, a server can be moved to any host within the repository's location domain. This is possible because all IORs produced by servers in that domain share the same repository address. However, there are two limitations:

- All objects implemented in a server need to move at once.

  This limitation arises because the implementation repository uses the object adapter name as a key into the location table. If an object moves, all objects in the same server need to move with it because all references to objects in the server contain the same object adapter name.
- A server cannot move into a different location domain.

  Moving across a location domain boundary implies that a server now depends on two repositories, one to bind references created while the server was in the old domain, and one to bind references created in the new domain.

6.1 Granularity of Object Migration

Ideally, an ORB would permit free movement of objects, including movement of a single object from one server to another. This is possible, but only at a cost: the implementation repository needs to know about individual objects instead of servers. This in turn compromises scalability, because more state needs to be externalized (stored in the repository). A server may implement millions of individual objects, and it is difficult to maintain performance of a repository if it stores millions of object locations. In addition, it is hard to maintain consistency between the servers' view and the repository's view of object locations.

Alternatively, instead of storing a registration for each individual object, the repository can communicate with servers during binding to dynamically acquire the addressing information for objects. This avoids the problem of storing large numbers of object locations, but requires more messages and so reduces performance.

One compromise approach uses multiple POAs. A physical server can contain a number of named POAs, each of which acts as a scope for object names. As before, IORs still contain a single POA name as part of the object key. However, by using multiple POAs, the server can maintain different naming scopes for its object names. The implementation repository contains a separate registration for each POA and therefore can locate objects at the grain of a POA instead of a whole server process. This permits some objects to be moved out of a server to a different location while other objects can remain at the original location. However, all objects using the same POA must migrate together. This approach scales well and provides more flexibility, but in practice only works well if the number of POAs is small because registrations need to be maintained manually.

Commercial implementations use different approaches to object migration. Some ORBs have implementation repositories that can locate an individual object instance, whereas others can only locate objects at the granularity of a server process or a POA. The various approaches are subject to the performance and scalability trade-offs mentioned previously.

6.2 Migration Across Location Domains

It is possible to migrate servers across location domain boundaries, but this again reduces performance and scalability. There are two approaches:

- When a server migrates to a new domain, it registers itself with its new implementation repository as well as all repositories it has used in the past.

  The idea is that all repositories a server has ever used know the current location of the server and therefore can continue to bind requests arriving via an IOR generated at a previous location. This works, but has the drawback that, over time, more and more server registrations accumulate in repositories. This in turn compromises scalability, because it increases the amount of externalized state and raises consistency issues.

- When a server migrates to a new domain, an administrator changes the registration in the old repository to create location-forward replies to the repository in the new domain.

  The idea here is to leave a "footprint" in the old repository that forwards binding requests to the new repository, which knows about the actual location of the server. Again, the problem is that the forwarding footprints accumulate over time. In addition, a bind from a client via an IOR created at a previous location must be forwarded from repository to repository until it finally arrives at the server. The binding chain gets longer with every migration, so this approach does not scale if servers migrate more than a few times.

Hybrids of the these two basic ideas are possible. For example, repositories can be arranged into domain hierarchies to reduce the length of forwarding chains, and repositories can be combined into redundant groups to gain performance and fault tolerance. However, all approaches are subject to the basic trade-offs between flexibility, scalability, performance, and fault tolerance. No commercial implementations currently offer inter-domain migration.

7 GARBAGE COLLECTION

Object migration is complicated by the need to externalize location information in implementation repositories. It would be nice if one could "fade" such information once it is no longer required by clients. Unfortunately, it is generally impossible to know when that time has arrived.

7.1 The Pacific Ocean Problem

Consider the following scenario: You are stranded on an island in the Pacific Ocean, with a CORBA server as your only link to the rest of the world (you can reply to CORBA messages, but you cannot send them). Being desperate to get home, you decide to create a persistent SOS-object in your CORBA server (the server is properly registered with its implementation repository). You write the stringified IOR for your object on a piece of paper, put it into a bottle, and, having carefully inserted the cork, you toss the bottle into the ocean.

The bottle floats around for a few months and eventually washes ashore in Australia, where it is found by someone strolling along the beach. Luckily, the finder of your bottle knows all about CORBA, destringifies the object reference, contacts your object to learn about your predicament, and comes to the rescue.

Contrived as this example is, it illustrates an important point: because CORBA permits persistent references to propagate by uncontrollable means, there is no way of knowing whether or not an IOR is still of interest to some client. In the preceding scenario, the IOR continues to be of interest while it is floating in the Pacific Ocean, and the finder of the bottle has every right to expect a CORBA invocation via the IOR to reach your SOS-object.

These semantics make it impossible to "fade" or *garbage collect* a forwarding registration in an implementation repository without running the risk of leaving a client with a dangling IOR.

7.2 Garbage Collection of CORBA Objects

Garbage collection issues arise not only for implementation repositories, but also apply to CORBA in general. For example, CORBA relies extensively on the *factory pattern* [1][3]. A factory is an object that creates other objects. To create a new object, a client invokes an operation on the factory. The factory operation creates a new object and returns its IOR to the client. To delete the object, the client invokes an operation on the object that instructs it to destroy itself.

GIOP has no session concept and does not permit servers to distinguish between orderly and disorderly client shut-down. If a client crashes or loses connectivity before it gets around to destroying an object, the server has no means to detect this, and the object simply hangs around forever (or, if it is a transient object, continues to exist until its server shuts down). If too many persistent objects are left behind over time, the object garbage can accumulate to the point where performance is severely compromised.

CORBA currently does not offer garbage collection as a platform feature and forces applications to deal with the problem. Garbage collection can be quite easy to implement for a specific application scenario; the most common approach is to destroy objects if they have not been accessed for some time. However, application-level garbage collection requires additional development effort.

7.3 Referential Integrity

The flip-side of losing an IOR is losing an object. This happens if a client decides to delete an object while other clients still hold references to that object. Once the object is gone, the references held by other clients dangle. The problem is similar to that of broken links on Web pages.

Dangling references fall under the broader topic of *referential integrity*. A system of CORBA objects and their IORs has referential integrity if there are no dangling references (references without objects) and there are no orphaned objects (objects that cannot be contacted via a reference). As an analogy, the Web would exhibit referential integrity if there were no broken links, and every page could be reached from some starting point by traversing some sequence of links. Clearly, it is very difficult to maintain referential integrity in a heterogeneous distributed system that spans enterprise and administrative boundaries; random failures that compromise referential integrity are unavoidable.

Garbage collection is one way to address referential integrity. It can be used both to prevent deletion of objects that are still of interest and to guarantee deletion of objects that are no longer wanted. Another way

to guarantee integrity is to use *transactions*. CORBA already includes a transaction specification [3], and the OMG is in the process of addressing garbage collection [6].

7.4 Dealing with Lack of Referential Integrity

One way to deal with lack of referential integrity is to live without it. In real life, we cope with lack of referential integrity all the time. For example, when people dial a telephone number and get a “no such number” message (the equivalent of a dangling reference), they do not throw up their hands in despair. Instead, they have a number of fall-back behaviors to recover from the problem (such as using the phone book or calling directory assistance).

In CORBA, the equivalent fall-back behavior is not to rely on references to work at all times, but to dynamically re-acquire them when they fail. Implementation repositories provide fall-back behavior through their location-forward mechanism. However, fall-back behaviors are useful not only for repository registrations but also for application objects. For application objects, fall-back behavior is substantially easier to implement if transactions and garbage collection are available as platform features.

8 SUMMARY

This article has explained how the implementation repository binds persistent IORs. The design of the repository has profound influence on the performance, scalability, flexibility, and fault tolerance of an ORB. CORBA intentionally leaves vendors with considerable freedom in repository design. Even though implementation repositories are not typically seen as a central point of interest, their capabilities determine at least in part how well an application will perform, scale, and evolve over time. This makes the features provided by an implementation repository an important consideration when choosing an ORB.

Garbage collection and referential integrity are areas where CORBA offers only partial solutions. Forthcoming revisions of the specification need to address these issues for CORBA to remain at the forefront of distributed object technology.

9 ACKNOWLEDGEMENTS

I am indebted to Keith Duddy, David Jackson, Michael Neville, Jocelyn Thompson, and Steve Vinoski, who commented on drafts of this article.

Portions of this article have been excerpted from the forthcoming book *Advanced CORBA Programming with C++* by Michi Henning and Steve Vinoski. Copyright 1999, Addison Wesley Longman, Inc.

The work reported in this article has been funded in part by the Cooperative Research Centre Program through the Department of Industry, Science, and Tourism.

10 APOLOGIES

Apologies to J.R.R. Tolkien.

11 ABOUT THE AUTHOR

Michi Henning is a Senior Research Scientist at DSTC, where he spends much of his time providing CORBA consulting and training to international customers. He has contributed to a number of OMG specifications, is involved in ongoing CORBA-related research, and is a member of the OMG’s C++ Revision Task Force. Currently, he is working with Steve Vinoski on a book about CORBA programming with C++. When he is not doing computer-related things, he desperately tries to improve his golf handicap—hope springs eternal…

12 REFERENCES

[1] Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Framingham, Mass., 1998.

[3] Object Management Group, *CORBAservices: Common Object Services Specification*. Object Management Group, Framingham, Mass., 1997.

[4] Object Management Group, *A Discussion of the Object Management Architecture*. Object Management Group, Framingham, Mass., 1997.

[5] Object Management Group, *Fault Tolerant CORBA*. Draft RFP, ftp://ftp.omg.org/pub/docs/orbos/98-03-05.pdf, Object Management Group, Framingham, Mass., 1998.

[6] Object Management Group, *Garbage Collection of CORBA Objects*. Draft RFP, ftp://ftp.omg.org/pub/docs/orbos/97-08-08.pdf, Object Management Group, Framingham, Mass., 1997.

[7] The Open Group, *DCE 1.1: Remote Procedure Call*. CAE Specification C706, The Open Group, Cambridge, Mass., 1997.

[8] Schmidt D. and Vinoski S. Object Adapters: Concepts and Terminology. *C++ Report* 9,11 (Nov. 1997).

[9] Schmidt D. and Vinoski S. Using the Portable Object Adapter for Transient and Persistent CORBA Objects. *C++ Report* 10,4 (April 1998).

[10] Schmidt D. and Vinoski S. Developing C++ Servant Classes Using the Portable Object Adapter. *C++ Report* 10,6 (June 1998).

[11] Vinoski S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications* 14,2 (Feb. 1997).