orbix™

distributed object technology

The Orbix™ Architecture

IONA Technologies Ltd.

August 1993

Summary

An Object Request Broker (ORB) mediates between applications - including distributed ones. This document presents the design goals and philosophy that lead IONA Technologies to produce the object request broker, Orbix. A few simple programming examples are given, together with some performance measures.

The introduction discusses the needs for ORBs and the industry initiatives that arose from that need - culminating in the OMG CORBA specifications. This is significant because Orbix is a full and complete implementation of CORBA.

Table of Contents:

What is OMG's CORBA Specification?

The problem of high level application interworking and construction is well recognised by industry. Microsoft, in particular, has led the way with its OLE technology as a means for interworking applications at a higher level than byte streams. SunSoft's Tooltalk, Apple's Publish & Subscribe, and Hewlett-Packard's SoftBench are examples of similarly inspired approaches. However each of these mechanisms, including OLE, is proprietary. Recognising the urgent requirement for a standard in high level application interworking, including across multiple platforms and network architectures, industry has formed a consensus via the Object Management Group (OMG) and in particular the Common Object Request Broker Architecture (CORBA) specification.

The OMG is the primary industrial body for the promotion of the standardisation and adoption of object technology. The OMG was formed in April 1989 by American Airlines, Canon, Data General, Gold Hill, Hewlett Packard, Philips, Prime, Soft-Switch, Sun, Unisys and 3-COM. AT&T joined two months later, followed by Digital and NCR in March 1990. By January 1993, membership had reached over 290 companies, including for example IBM, Borland and Microsoft.

In October 1991, OMG announced its adoption of the CORBA specification. The formal announcement brought together six submitting companies - Digital, Hewlett Packard, HyperDesk, NCR , Object Design and SunSoft. CORBA specifies a messaging facility for a distributed object environment: a mechanism for a number of objects to have a standard way of invoking the services of each other. The specification comprises two chief parts - an Interface Definition Language, and a Dynamic Invocation Interface. Both are provided in the context of an Object Request Broker (ORB) - a fundamental service to enable messaging between objects in centralised or distributed systems.

The Interface Definition Language (IDL) structures objects so that, when combined with an API for accessing objects at run time, applications are constructed with prior knowledge of the kinds of objects with which they will interwork at runtime. For this reason, this approach is sometimes called the "static" approach.

The Dynamic Invocation Interface (DII), emerging from both HyperDesk and Digital, does not present a new language - such as IDL - but instead an API that can be called from C. In return for explicitly building messages and argument lists at run time, the dynamic API allows decisions to be made much later than the static approach. This is attractive for certain applications, such as browsers and resource managers, in which insufficient information about services is available at compile time. This disadvantage is that the API is more complex to use than the static approach.

Technical Goals for a CORBA Implementation

IONA Technologies Ltd. started trading in March 1991. The Company was formed primarily as a result of experience gained from participation in various ESPRIT projects during the period 1985-91, all of which had to a greater or lesser extent encompassed object technology. When the OMG published the CORBA specification in October 1991, IONA realised that the experimental and prototyping lessons from the ESPRIT participation could be used to build an implementation of CORBA.

When designing Orbix, we were very conscious of the original goals of the designers of UNIX: lightweight, elegant, and easy to use. In meeting these aims, we focused on building a “low cost” ORB which provided the essential distributed object paradigm. However, we were also keen to allow the object request mechanism to be open, so that Orbix application programmers could intercept a request at various stages of processing. In this way many ORB extensions and tailoring could be performed by the programmer. This combination of a simple and reusable ORB implementation would offer the best combination of flexibility and performance.

Performance was considered an essential aspect of being "lightweight". Similar to a database, an ORB is a generic engine upon which various applications are built. If users or programmers believe that the ORB is a performance bottleneck, then they will consider alternatives in coupling their applications together.

Flexibility and elegance are equally vital concerns. Consider the database analogy again: a database is a generic engine, and is purchased in binary form. Since its users do not have access to its source code, the database must provide sufficient "hooks" to allow customers to tailor and extend the system when appropriate: database trigger mechanisms are an example. An ORB faces a similar challenge in meeting the demands of a wide variety of application requirements.

The third theme was ease of use. CORBA itself helps: its emphasis on language technology and the object paradigm are keystones in making distribution simple to use and reuse. CORBA itself however is not that easy to understand. In building Orbix we wanted to remain faithful to CORBA, but also to make CORBA more palatable to programmers. Orbix makes it possible for programmers to build and integrate applications with relative ease.

We felt it was important to have a full implementation of the CORBA specification, including both the static and dynamic invocation interfaces. Although perhaps we would have been able to build a partial implementation more quickly, we fully expected our customers to require the flexibility of both of the invocation interfaces. We further wanted to ensure that all of the CORBA specification was faithfully implemented in Orbix: however this should not - and did not - restrict us from adding certain value-added services and extensions to make the product simple to use.

The current version of the CORBA specification - v1.1 - is itself specified in terms of a C language binding. In our view, this binding is cumbersome and not particularly easy to use. We therefore made an early commitment to C++, the language of choice of object oriented programmers. Using C++ also gave us opportunities to make the dynamic invocation interface a little more user friendly. Although OMG had yet to provide a C++ mapping for CORBA, we believed that it would be reasonably obvious how such a binding could be derived, based on the already published C mapping. In the spring of 1993, the OMG formally invited proposals for a C++ mapping: Hewlett-Packard and SunSoft, NEC, and IONA independently proposed three almost identical mappings which were subsequently merged.

We were keen to recall our good and bad experiences from our ESPRIT participation in designing and implementing distributed object systems. One of the key lessons learnt was that it is essential to allow server programmers to maintain close control over how their services are represented to clients. One example might be to cache state from a server out to its clients, and so significantly improve performance by reducing the number of remote calls. Another key lesson was that it is highly beneficial if programmers can transparently insert application code into invocation paths - for example for debugging, auditing, or encryption purposes. Yet a further conclusion was that it is desirable to be able, at different times, to bind the same objects together into the same process address space, or alternatively across a network in different machines, without having to recode either clients or object implementations. A couple of negative lessons we recalled, was that both transparent distributed garbage collection and transparent persistence mechanisms are highly challenging to achieve.

Although inspired by the original themes for UNIX, and as reflected in the name Orbix itself, Orbix however is not intended to be an environment solely for UNIX. For example, Orbix will shortly be available on Microsoft's NT platform, and on Windows 3.1 towards the end of 1993.

Orbix Architecture

The CORBA specification is relatively free of architectural stipulations. An ORB mediates between clients and implementations (of application objects). The ORB must provide a specified interface to such clients, and another to such implementations. In CORBA, the interface specified for object implementations is not as rigorously specified as it is for clients.

What, in practice, is the ORB? CORBA does not specify whether it is a set of runtime libraries, a set of daemon processes, a server machine, or part of an operating system: it can in principle be any of these.

Orbix is fundamentally implemented as a pair of libraries - one for client applications, and for servers - and an activation daemon, `orbixd`. The client library provides a subset of the server library: while the server library can both issue and receive remote object operation requests, the client library can only initiate such requests. `Orbixd` need only be present at server nodes, and is responsible for (re-)launching server processes dynamically as required, in accordance with the various activation policies described in the CORBA specification.

Where is the Orbix ORB ? Because of its library implementation, the Orbix ORB is conceptually omnipresent: there is no distinct component which one can identify and state that this one component encapsulates the entire ORB. There is no central component through which all object requests must pass: instead object requests are passed directly from the client code to the invoked object implementation.

The role of `orbixd` is somewhat similar to the well-known UNIX `inetd` daemon. We unfortunately could not easily make use of inetd itself, since the CORBA specification identifies several different ways in which object implementations can be activated. Resolving which activation mode is appropriate on which occasion requires extra sophistication beyond that offered by inetd. `Orbixd` uses a simple database, the Implementation Repository, to obtain activation information for its object implementations: for each such implementation, the information includes the appropriate CORBA activation mode, the name of the associated executable image and any command line parameters.

In addition to the `orbixd`, client and server libraries, Orbix also consists of an IDL compiler, Interface Repository and various utilities. The IDL compiler is primarily a part of the development environment, and used to translate IDL descriptions into stub code to aid remote operations. However the IDL compiler can also be a runtime component, as is explained in the next section. The Interface Repository (IR) is an Orbix application which allows other applications to determine the interface properties of objects at runtime: the IR is specified in the CORBA document. The utilities are used to simplify the management of the Implementation Repository.

Finally, a third library is provided - currently called "`both`" - which can be used as an alternative to the client and server libraries. By relinking the client and server parts of an application with the `both` library, the entire application can be collocated and run as a single (e.g. UNIX) process. Such collocation can aid the development of an application by easing debugging, but also shows how CORBA facilities can be used within a non-distributed application to aid large scale application development.

The entire Orbix system is itself implemented in C++ (and has been compiled with several compilers including ATT cfront 2.1 and 3.0 compatible UNIX compilers, gnu G++ 2.3.3, and Microsoft C7).

The remainder of this document explains the components of figure 1.

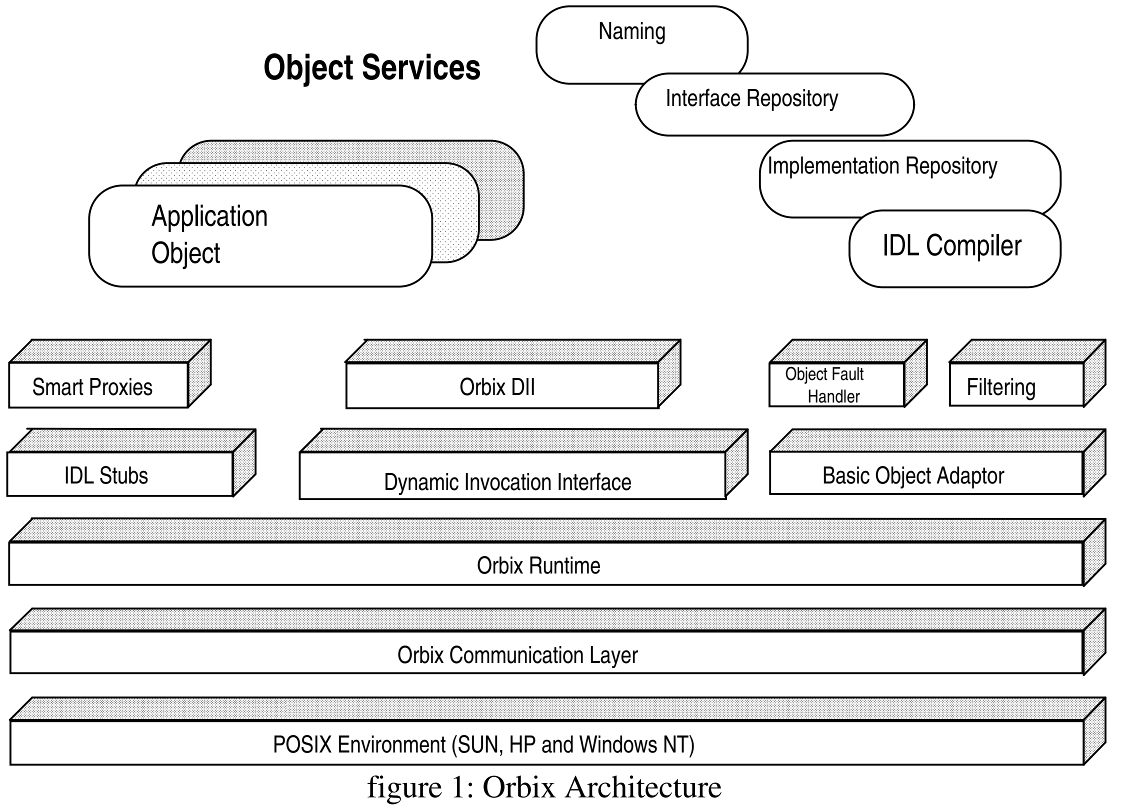

figure 1: Orbix Architecture

Orbix Architecture - The Communication Layer:

This layer provides the essential transport facilities. In the Orbix source code, the layer is implemented primarily by four classes: `MediaAccess`, `Network`, `RequestSender` and `RequestReceiver`.

`MediaAccess` and `Network` are two abstract classes. The default implementation of these classes uses TCP/IP and XDR encoding, together with a simple messaging protocol. Alternative implementations of these classes (for example to use a different transport stack) are possible.

`RequestSender` and `Receiver` implement messaging of object requests using `MediaAccess` and `Network`. Currently they are both concrete classes and rely on the

UNIX select system call (or equivalent functionality)[1]. Both are present in the Orbix server library, while the client library only has the Sender class.

Orbix Architecture - The Runtime

The Runtime implements the CORBA client APIs (e.g. the dynamic invocation interface, DII) and server APIs (e.g. the basic object adaptor BOA). It also implements the functionality required by the messaging stub code produced by the IDL compiler from IDL source descriptions.

The fundamental classes in the runtime are the `Request` and `Object` classes. The interfaces to both classes are available to Orbix application programmers via the CORBA module. Programmers who choose to use the CORBA static invocation interface need not however be particularly aware of these two classes, since the generated stub code largely insulates such programmers.

The `Request` class implements the `Request` interface defined in the CORBA specification. It is a part of the DII, and in Orbix is also used by generated stub code. Orbix extends the DII as specified by CORBA, with a stream based interface[2]. When presented with arguments to an invocation, the `Request` class internally checks whether these are being delivered as a part of the DII or from the statically generated stubs. DII arguments are passed into a Named-Value list (`NVList`) and their marshalling deferred until `Request::invoke` is called. Static arguments are marshalled directly.

The `Object` class implements the `Object` interface defined in the CORBA specification. In effect, an instance of the `Object` class has the fundamental information necessary to communicate with a remote object. For each IDL interface compiled with the IDL compiler, there is a corresponding generated (e.g. C++) class: we sometimes call this an IDL class. The `Object` class is the ultimate base[3] class of all IDL classes.

In addition to these two fundamental classes, class `TypeCode` and `any` implement their corresponding CORBA specifications. Although `any` and `TypeCode` only receive short descriptions in the CORBA specifications, their implementation is complex and comprises a major part of the runtime source code. A value of `any` is fundamentally mapped - in C or C++, to a `void*` - that is, essentially any value whatsoever, including arbitrarily complex structured data. In order to identify the specific kind of value an `any` actually has at runtime, each `any` value is also tagged with type information. The type information in turn is interpreted by `TypeCode` (and can be generated from the IDL compiler). Marshalling an `any` value can involve deep recursion, based on runtime interpretation of the type code tag.

Apart from these four classes, and others, which implement interfaces laid down in the CORBA specifications, the runtime includes classes specific to the Orbix implementation. During execution, the runtime builds a "proxy", or "surrogate" for each remote object used

[1] The set of file descriptors used in this select are available to Orbix application programmers and can, for example, be merged with other descriptors used in other sub-systems - for example X windows and its main event loop.
[2] It also implements the standard DII interface.
[3] In C++, the virtual base class.

by the local process. Each such proxy is an instance of an IDL class. If the IDL class is unavailable to the local process - because the IDL interface was unknown at the time the local process was built, and thus there is no IDL stub code available - then the proxy is instead made an instance of class `Object`.

The runtime maintains a table of all proxies, and of all implementations of IDL interfaces (these for example occur in servers), which in the Orbix source code is called the Object Table (OT). There is one such table per process context (e.g. per UNIX process).

Orbix Architecture - Smart Proxies

The default action of the generated stubs of each IDL class is to marshall the request and to forward it on to the remote object. These stubs are the methods of the proxy objects: recall that each proxy is an instance of an IDL class.

An Orbix programmer can however change this default. Using inheritance, the generated stub code can be overridden in a new derived class: furthermore, the original (generated) code which implements the remote operations is available by calling up the inheritance hierarchy. Orbix also provides a mechanism such that when a new proxy must be constructed for a particular IDL interface at runtime, the proxy can instead be made from a specified derived class of the corresponding IDL class.

In fact, for a specific IDL class, there can be several alternative derived classes. When any particular new proxy must be constructed, these classes can collaborate to agree which of them is responsible for this particular construction, based on the identity of the specific remote object for which the proxy is about to be built.

Smart proxy support for a particular IDL interface is typically provided by a server programmer, who wishes to control the behaviour of her server which is presented to its clients in their process contexts. Smart proxy support is typically transparent to an Orbix client programmer.

The most common use for smart proxies is when a server programmer wishes to allow her clients to cache state from the server, so as to improve performance and reduce the number of remote calls. It should be noted that while the Orbix client library cannot receive unsolicited incoming requests (which distinguishes it from the Orbix server library) it can nevertheless receive incoming requests - "call-backs" - from a server with which it has earlier corresponded. Server call-backs can be used to notify a smart proxy cache of a change of state at its associated server.

Smart proxies can have several other uses, for example:

- Server rebinding, where the proxy can be rebound to an alternative remote server when the original server fails

- Breakpoints, when debugging and trace code can be executed.
- Type conversion of IDL types to non-IDL types, for example converting IDL sequences into conventional linked lists, when migrating legacy applications onto CORBA.

Orbix Architecture - Filtering

Orbix does not directly implement various functions sometimes required in distributed environments: for example, it does not implement authentication, encryption, auditing, a threads environment, nor atomic transactions or two-phase commitment. One motivation for this was to keep the implementation flexible and light-weight: we felt it important not to insist that every Orbix installation must be configured in the same way. For example, some sites may require authentication and some may not. Furthermore, some may be content to use the Kerberos package as the basis for authentication, whilst others may insist on even stronger requirements.

In a sense, Orbix is an ORB and nothing more. We felt it important to keep its functionality orthogonal and complementary to other software infrastructure packages. This also provides third party developers with an opportunity to add value-added services to the basic substrate.

Smart proxies allow the behaviour of remote representatives of objects to be extended and modified. Smart proxies could conceivably be used as a way of introducing functionality such as Kerberos into Orbix: a proxy class would have to be provided for every IDL class, and each and every smart proxy class would have to be given such support. Clearly it would be preferable to have a mechanism that was independent of all the IDL and proxy classes. Furthermore, proxy classes are normally associated with clients and not servers: a package such as Kerberos requires support on both sides of a communication channel.

The Orbix runtime allows Orbix programmers to supply filtering code in both clients and servers. Filters are instances of filter classes, which in turn are derived classes of the abstract class `Filter` provided by Orbix. Filters are formed in a linked list: that is, an arbitrary number of filters may be installed.

Fundamentally, filters are applied when an operation request or reply is about to be transmitted from a process context, and when such a request or reply is received. The default action (inherited from class `Filter`), in each case, is to simply pass the event on to the next filter in the chain. Having a processed an event, a filter can choose to suppress the event from the remaining filters in the chain. The chief parameter to each filter event is the current request, from which the target object and operation name can be determined. Further parameters can still be marshalled into (or from, as appropriate) the current request by the filter, using the stream-based DII: for example, an authentication token might be marshalled into the request at the time it is about to leave the process context.

Coupling with a threads package is a special case of the filter mechanism. Orbix is not delivered integrated with a threads package, because we envisaged that some applications may not wish to use threads at all, and for those that do there may be several possibilities including SunOs lwp, Solaris 2.2 threads and Microsoft NT threads. A filter can however be written to catch all incoming requests into a process context and dispatch each on a new thread. The creating thread returns from the filter, with the request event apparently suppressed: the new thread continues with the request, applying the remaining filters (if any) in the filter chain, and then calling the target object.

As described above, filtering is a per-process level mechanism, and applies (transparently) to all requests and replies leaving and entering a process context. We have also seen how smart proxies can transparently mediate client requests. Orbix provides a second form of filtering which complements smart proxies and operates within servers.

Per-object filtering is a mechanism for providing a filter chain attached to a specific object instance (within a server). The chain can operate independently of other server objects. A per-object filter chain is applied after the per-process filter chain, in the case of an incoming request, and before the per-process chain when the reply (if any) to that request is formed.

A further distinction from per-process filtering is that at the time a per-object filter chain is applied, the actual parameters (if any) to the specific operation have been unmarshalled and are available to the filter code. The filter code in effect is thus another implementation of the IDL interface associated with the target object: the filter code has methods for each of the IDL operations for the target object. Once again, a filter can choose to suppress the event, and so a per-object filter might choose not to pass the request through to its target object and instead, perhaps, generate an exception.

Per-object filtering can have similar uses to smart proxies, including assistance in debugging, auditing, and legacy applications. However a further use is to be able to transparently propagate a server event across a collection of objects, where such a collection can transparently change. A "move" operation on a graphical object described in IDL could, for example, be notified to a set of attached graphical objects, so that the entire aggregate is moved in unison.

Orbix Architecture - The Object Fault Handler

Orbix does not have any direct support for handling persistent objects - that is, objects whose state can be saved and restored from non-volatile storage. Some of the IONA team in fact have had very considerable experience in building distributed and persistent object systems, based on our earlier work in various ESPRIT projects and other research. However, when building Orbix and in keeping with our themes of simplicity, flexibility and elegance, we were convinced that it would be unwise to ship Orbix with a tightly coupled persistent store, which could adversely affect performance and cost. In addition, no single approach or persistent store would be likely to suit all applications.

Coupling a persistent store - whether it be flat file based, an rdbms or an oodbms - is clearly nevertheless an important requirement for many applications. The fundamental support for this is the abstract class `LoaderClass` in the Orbix runtime. Instances of `LoaderClass` - loaders - are formed in a linked list. Whenever a new object (described in IDL) is built and registered with Orbix, the loaders are notified. A server programmer can also choose to name a new object (using a character string) and this name - if any - is also passed to the loaders together with the identity of the object's IDL interface. The name given to the object is called its marker name. The loaders must be coded so that they agree which loader is responsible for which object. Typically there is a single loader, or one loader for a particular set of IDL interfaces. The loader responsible for the new object can choose to adopt the proposed object name (if any) or generate a name for the object. A generated name might be, for example, a relational key which will be later used as a basis for storing the object.

Alternatively, a specific loader can be nominated when an object is registered with Orbix: for example, a class may be written so that all of constructors ensure that the same loader is used for all of its instances.

The loaders are notified when the server process exits, and can choose to carefully store the state of the objects for which they are responsible. A loader can also choose to unilaterally save the state of an object prior to process exit.

When an operation request is received into a server, the loaders are notified if the Orbix runtime cannot locate the target object - i.e. the target is not yet registered in the OT. The target object's name is also passed to the loaders, so that the responsible loader can be identified and so that it may attempt to restore the object's state from persistent storage. If the target object is successfully retrieved, the OT is updated and the operation request (transparently) resumed - the "object fault" has been successfully handled. If the target object cannot be retrieved, or no loader recognises the object's name, then an exception is returned back to the client.

The actual translation of the volatile state of a specific object into and from its persistent state is not handled by Orbix. We felt that this was more properly a concern for tools associated with a particular storage manager, rather than the ORB itself. We certainly did not wish to constrain the way in which this is done, by forcing the usage of a specific store.

The default implementation of `LoaderClass` - the default `Loader` - names its objects using simple increasing numeric values. It does not attempt to save objects to store, and ignores object faults.

Orbix Architecture - Object Naming and the Location Service

In Orbix, an object is named by concatenating the host name of the node at which it was created, with the name of the server which created it, and the name which the object has within that server - its marker name, as previously explained.

The name of a server is by default the same name as that of an IDL interface which it implements. For example, if we have a server implementation of an interface called `bank`, then the server by default will also be called `bank`. However it is quite common for the same server to implement several IDL interfaces: our bank server might also contain code to implement IDL interfaces for bank account objects, bank statements, the bank manager, and so on. In this case, it may be possible to choose the server name by identifying a master interface which abstracts the functionality which the entire server provides, and via which all objects managed by the server are obtained. For a banking application, the interface to the bank itself may be such a master interface.

Finally, a server name can be chosen which is independent of any particular IDL interface. For example, we could choose `financialRepository` as the name of our bank server, even though there is no IDL interface with that name in the application. Server names can also be hierarchically structured, similar to UNIX file names.

The name chosen for a server is significant because it is registered in the Implementation Repository, and used by `orbixd` to identify the executable file which should be used to activate the server. The same executable file can be registered under several server names; i.e. different servers can use the same executable image.

When a client program wishes to use a particular named service at runtime, it must instruct Orbix to bind the client to a suitable server. One way to do this is for the client to provide Orbix with a character string which represents a full Orbix object reference: this mechanism is prescribed by the CORBA specification. The source of such a character string is not prescribed: it could be found in a file, or indeed transcribed from an electronic mail or fax message!

Alternatively, in Orbix, a client can bind to a specific server name at a particular host. The server name must be one of the server names registered in the Implementation Repository at that host - for example, `bank` or `financialRepository` as above. The client can go further, and attempt to bind to a specific named object at that server, as given by the object's marker name. If the client does not specify a target marker name, then Orbix binds the client to any object within the server which provides an interface compatible with that expected by the server.

The most general form of a client bind is when the client identifies a service, but not a specific host which can provide that service. The service is named by a server name: zero or more hosts[4] may recognise that server name, based on the information in their respective Implementation Repositories. In this case Orbix must "search" the network, looking for suitable hosts.

Such a search is managed by the location mechanism, implemented in the Orbix runtime by the abstract class `locatorClass`. This class is called with a service name (as a character string) and is expected to return a list of host names at which the service appears to be present. The `locatorClass` is usually used transparently to application code.

The default implementation of `locatorClass` uses a configuration file at each host: this registers knowledge about which hosts, and groups of hosts, can provide specific services. Each configuration file can contain a pointer to another host to which queries can be

[4] In fact, the orbixd daemons running at those hosts.

forwarded if the information required is not found in the current file. The number of hops used to consult these configuration files is bounded.

The default implementation can of course be overruled by providing a derived class of `locatorClass` and registering an instance of this new locator with Orbix. An alternative implementation might for example use a directory service in which the mapping from service names to groups of hosts had been registered.

If the location service identifies that several hosts can provide the target service, Orbix selects any one of these hosts at random. Randomising the selection helps to spread server loading when multiple clients are using the same service.

Orbix Architecture - The IDL Compiler and Interface Repository

The IDL compiler is internally structured so as to maintain a clear separation between the front-end parsing and analysis, and the back-end interpretation of the parsed IDL source files. Three different back-ends have so far been written: simple regeneration of the original IDL source from the parse tree; stub code generation for C++; and generation of information for the Interface Repository.

The clean separation between front-end and back-end has possible commercial as well as technical benefits. We thought it possible that other parties may wish to have access to an IDL compiler in order to do their own interpretation of IDL source code. In the standard Orbix product the header files defining the internal compiler interfaces are not documented and shipped. However, in principle they could be given to interested customers, together with the source code for the simple IDL regeneration back-end.

At about the time that we had the first version of the IDL compiler completed, SunSoft placed a version of their own IDL compiler effectively into the public domain. At that stage, we were committed to our own development, and saw no particular reason to change. Naturally however we are keen to ensure that our compiler remains aligned to the specification of IDL and with SunSoft's compiler.

The IDL compiler is shipped in Orbix as a full executable for the stub generation. However during our design of Orbix, we were also concerned about the complexity of the CORBA Interface Repository specification and were keen to simplify its implementation as much as possible. Clearly, parsing of IDL source files should directly or indirectly have the side effect of providing the information required by the IR.

We considered how the IDL compiler could be integrated into the IR using the object fault handler mechanism described earlier. That is, the IR should be built as a normal Orbix object server. It should use the object fault handler to detect an attempted invocations on (IR) objects as yet unknown to it. To resolve these object faults, it could use the IDL compiler *at runtime* to parse the appropriate IDL source files and generate the objects used internally by the IR.

As a result the IR does not use a persistent object store for its information: rather IDL source code files are in effect the persistent store for the IR. An immediate benefit was the use of standard utilities - in UNIX, such as emacs and RCS - to update and provide version control on the IR information.

A consequence of the design however was that the IDL compiler (front-end, and at least one back-end) would have to be runtime callable code. The major issue here was to ensure that when the compiler was called at runtime, no dynamically allocated memory was leaked as a result of parsing. The compiler (and its cpp-like pre-processor) has been engineered accordingly, making extensive use of the Purify tool to ensure leak-free operation - as indeed has the remainder of the Orbix system.

One concern regarding runtime parsing of IDL source files might be the delay in handling an object fault while the IDL compiler was called. The compiler is dynamically linked into the IR application and is not called as a separate process, so at least there is little overhead on its activation. However, the IR can also be instructed to pre-parse specified IDL source files, and so avoid object faults on IDL interfaces which are known in advance to be commonly used.

A second concern regarding runtime parsing has been the virtual memory requirements of the IR. For long-running systems, in which many IDL interfaces need to be known to the IR, there is a possibility that memory usage by the IR may grow. We have yet to experience that so far by the IR, but may need to introduce a mechanism into the IR code so that memory usage can be monitored and so that less recently used interfaces registered (i.e. parsed) with the IR are forgotten: if they are needed again, they will have to be reparsed.

Writing an Orbix Application

Given an IDL object description, it is possible to generate all the necessary code to marshall and unmarshall all the parameters which must be passed with an object invocation. The IDL stubs are automatically generated by the IDL compiler.

Consider the following IDL:

```
// IDL
interface foo {
      string op1(in long x, inout float y, out string z);
};
```

The IDL compiler will generate the appropriate code to marshall the parameters on the client side, transmit the request to the server, dispatch it to the correct object and send the results of the invocation back to the client.

To the client programmer this is very natural to invoke from C++:

```
// C++
char *result;
long x;
float y;
char *z;
fooRef myFoo;

// bind myFoo (see below)
myFoo = foo::_bind ();

// use myFoo
result = myFoo->op1(x, y, z);
```

In this case, `myFoo` was bound to any `foo` server running somewhere on the network (using the location service).

More elaborate binding can be done, and exception handling can be included:

```
// C++
TRY {
      // bind to object 16716 at the Fred server on host qwerty
      myFoo = foo::_bind ("16716:Fred","qwerty", IT_X);

      // use myFoo
      result = myFoo->op1(x, y, z, IT_X);
}
CATCHANY {
      cout << "Unexpected exception " << IT_X << endl;
}
ENDTRY
```

The `TRY`, `CATCHANY` and `ENDTRY` macros are provided to assist exception management pending the widespread availability of C++ exception handling in C++ compilers. The variable `IT_X` is defined by the `TRY` macro and it includes exception information (which can be displayed on a `ostream` using `operator<<`).

Writing an Orbix Application - The Server

For a server programmer, providing an implementation of the example interface is again very natural:

```
// C++
class foo_impl : public fooBOAImpl {
public:
      virtual char* op1(long x, float& y, char&* z,
                        CORBA::Environment &env
                        =CORBA::default_environment);
};
```

The class `fooBOAImpl` is produced by the IDL compiler for interface `foo`, and is responsible for arranging for the unmarshalling of the `bar` operation. Note the extra (defaulted) trailing argument which corresponds to the optional exception argument for the client.

A server programmer can create an instance of `foo` and provide the `foo` service by telling Orbix that the server is now ready to receive bar requests. The simplest mainline is thus:

```
// C++
main () {
      foo_impl obj;
      CORBA::Orbix.impl_is_ready ();
}
```

Writing an Orbix Application - The Dynamic Invocation Interface

Use of the IDL compiler, as above, requires a client to have available (possibly by dynamic link loading) the marshalling stub code used to access any of the remote objects which its uses.

For some applications, this constraint may be too restrictive. CORBA specifies an alternative, API-based, DII. Orbix provides the CORBA DII API. However, this API is, in our view, quite difficult to use and so we have provided an alternative stream based interface. Using the Orbix stream based DII the same client code as above can be written:

```
char *result;
long x;
float y;
char *z;


TRY {
      // Initialise an object reference and a request
      CORBA::ObjectRef target =
            CORBA::Object::_bind ("16716:Fred","qwerty", IT_X);
      CORBA::Request r(target, "op1");

      // stream in the arguments,  and make call
      r << x << CORBA::inOutMode << y
                  << CORBA::outMode << z;
      r.invoke (IT_X);

      // obtain result if no exception pending
      if (!IT_X)
            r >> result;
}
CATCHANY {
      cout << "Unexpected exception " << IT_X << endl;
}
ENDTRY
```

Measurements

The following performance measured were obtained on a pair of lightly loaded Sun SPARCstation ELCs, configured with 8Mbytes memory, and under SunOS 4.1.3, and using the gnu g++ compiler (v2.3.3), using TCP/IP and XDR encoding. A null-bodied (non-inlined) null-argument C++ function call on these machines takes 0.3µsecs. A null-bodied null-argument virtual member function call takes 0.4µsecs.

Invocation Tests

# arguments	*Request/ Reply*	*Request/Reply using collocation*	*Oneway calls*
0	8msecs	2.5µsecs	3msecs
1	8msecs	2.5µsecs	3msecs
10	8msecs	2.7µsecs	3msecs
100	10msecs	2.7µsecs	5msecs
1000	25msecs	2.7µsecs	20msecs
10000	191msecs	2.7µsecs	140msecs

Each argument was an `in unsigned long`, with multiple arguments being passed as `in` arrays of `unsigned longs`. The bodies of the invoked operations were empty. The flat behaviour of the results using the collocation library - i.e. the "both" library - is explained by the fact that array arguments in IDL are passed by pointer value in the corresponding C++, and are therefore independent of the actual size of the array.

We experimented with the cost of using smart proxies and filters to the invocation path. We added a smart proxy to the tests above, which simply invoked its base class (i.e. the generated IDL class). The cost was insignificant for remote calls, being of the order of a further 1µsec. Likewise the cost of filtering code (assuming null filter functions) is about 1.5µsec per filter.

The creation of an object at a server requires registering the object with the Orbix runtime (including its insertion in the COT). This code is not manually written, but inherited from a generated base class constructor. It takes 650µsecs.

Remotely activating a server, when the locator service is not used (i.e. the target host is explicitly provided) takes about 4 seconds. This perhaps could be reduced: the Orbix client library currently makes up to 10 consecutive attempts to activate a remote server via the associated remote `orbixd`, at 2 second intervals. The `orbixd` must launch a new process during activation (in UNIX, a `fork` and `exec` call), and the new server must initialise itself and call `impl_is_ready` before the client can succeed in passing request through to it.

In the case where the server has been previously activated, binding a client to that server takes about 960msecs.

Conclusions

The UNIX operating system has been one of the most unexpected success stories of computing. One motivation for its design was a reaction against an alternative and complex operating system, Multics. One of the major reasons for the early interest in UNIX was that it was relatively lightweight, requiring only a moderate amount of memory and CPU power to run. It had a reasonable performance for interactive usage, in part as a consequence of its size. It became available on a wide range of platforms, again in part as a consequence of its size and ease of comprehension. It also had relatively open interfaces, and new applications can be composed simply from others using the shell facilities such as pipes and i/o redirection.

Much of the design of Orbix has been inspired the original goals of the UNIX design: lightweight and easy to use. However, Orbix follows an agreed industry standard - the OMG CORBA specification - unlike the early versions of UNIX, which perhaps as a consequence of the lack of a standard, were slow to be accepted by industry. Furthermore, much experience has been learnt from the difficulties in extending the early monolithic versions of complex software systems such as UNIX: Orbix follows the trend towards modular design, based on object orientation, and towards open interfaces which can be tailored to particular operating requirements.

Orbix is being made available on both UNIX and non-UNIX platforms.

UNIX originally introduced elegant mechanisms for developing new applications, and - what was at the time - a revolutionary mechanism for coupling applications together using byte streams. Much of this elegance has been lost in today's computing environments, since heterogeneous hardware and operating systems are being used, and the information being exchanged frequently has a higher level of structure than simple raw, byte-oriented streams. The OMG CORBA specification attempts to restore some order to the problem of application interworking, and Orbix is a faithful, lightweight and easy to use implementation of this standard.

Glossary

API	Applications Programming Interface
BOA	CORBA Basic Object Adaptor
CORBA	Common Object Request Broker Architecture
COT	Orbix Context Object Table
DII	CORBA Dynamic Invocation Interface
ESPRIT	European Strategic Programme for Research in Information Technology
IDL	CORBA Interface Definition Language
IR	CORBA Interface Repository
NVList	CORBA Named-Value List
OLE	Microsoft's Object Linking and Embedding mechanism
OMG	Object Management Group
ORB	Object Request Broker

References

Common Object Request Broker: Architecture and Specification. Published by the Object Management Group and X/Open, Reference OMG.91.12.1

Contact:

Mr. Colin Newman
IONA Technologies Ltd.
8-34 Percy Place
Dublin 4
IRELAND

Tel: +353-1-6686522
Fax: +353-1-6686573
email: info@iona.ie
ftp: ftp.iona.ie