The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Work-programme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Work-programme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited
Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK (0223) 323010
INTERNATIONAL +44 223 323010
FAX +44 223 359779
ARPA Internet apm@ansa.co.uk

Copyright © 1993 Architecture Projects Management Limited

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.
## Contents

1 Introduction  
1.1 Background  
1.2 Contents  
1.3 Audience  
1.4 Scope  

2 Interoperability Requirements  
2.1 One ORB or many ORBs?  
2.1.1 Diverse technology  
2.1.2 Diverse non-functional requirements  
2.1.3 Consistency versus availability  
2.1.4 Security versus convenience  
2.1.5 Autonomy versus uniformity  
2.1.6 Abstraction versus specialization  
2.2 Selective transparency  
2.3 ORB Support for selective transparency  
2.4 Interoperability between ORBs  
2.5 Boundaries between ORBs  

3 Binding  
3.1 Introduction  
3.2 A model for interface implementation  
3.3 Discovering interfaces  
3.4 Type checking  
3.5 Establishing communications  
3.6 Garbage collection  
3.7 Object migration  
3.8 Interpretation of CORBA IDL  
3.9 Relationship with ANSA and ODP  

4 Interface references  
4.1 Introduction  
4.2 Interface references  
4.3 Information content of an interface reference  
4.4 Connection  
4.5 Relocation  
4.5.1 Solution 1: Use of interface creation time  
4.5.2 Solution 2: Use of the relocation context
## ORB Interoperability

### 5 Interception

- 5.1 Introduction
- 5.2 Interceptors
- 5.3 Interception
  - 5.3.1 *Fixed interceptors*
  - 5.3.2 *Nomadic interceptors*
- 5.4 Translation
  - 5.4.1 *Type descriptions*
- 5.5 Building interceptors
  - 5.5.1 *Garbage collection*
  - 5.5.2 *Engineering issues*

### 6 Implementation guidelines

- 6.1 Introduction
- 6.2 Endpoint identification
- 6.3 Sub-addressing
- 6.4 Relocators
- 6.5 Relocator database
- 6.6 Integrity checking
- 6.7 Replicated locations
- 6.8 Interface reference contents
- 6.9 Dynamic endpoints
- 6.10 Activation
- 6.11 Relocator dependability
- 6.12 Optimized implementation of interface references

**References**
Chapter 1

Introduction

1.1 Background

The responses to the Object Management Group Request for Information on Revision of the Common Object Request Broker Architecture unanimously state that interoperability between Object Request Brokers (ORBs) is a major requirement.

This report submitted jointly by Architecture Projects Management (APM) and Bell Northern Research Europe at the request of the ANSA sponsors explains the architectural issues that should be addressed in achieving interworking between ORBs. The report is based on the ANSA architecture and has been confirmed in practical implementations. The concepts used are compatible with the Basic Reference Model for Open Distributed Processing (RM-ODP) being developed within ISO (ISO 10746-3 Basic Reference Model for Open Distributed Processing: Part 3 Prescriptive Model).

1.2 Contents

The report is structured along the lines of the RM-ODP viewpoints working from enterprise aspects of interoperability, through computational requirements into engineering detail:

- chapter two reviews the requirements for and constraints upon interoperability between ORBs
- chapter three explores the restrictions that have to be imposed upon the CORBA computational model to permit interoperability and the practical impact of these restrictions on the CORBA Interface Definition Language (IDL) and Object Adaptor respectively
- chapter four describes the requirements upon object references for ORBs that provide support for transparent object migration and object replication
- chapter five outlines the interception mechanisms required to support interoperability between otherwise incompatible ORBs
- chapter six describes implementation issues for interface references and interface relocation.

Chapter five is the core of the architecture for interoperability between ORBs. The other chapters provide the context to understanding and implementing the architecture.
Within each chapter sections which consolidate recommendations for the CORBA revision process are shown in italic type.

1.3 Audience

The report is written for an audience familiar with the CORBA 1.1 specification and reasonably familiar with some of the implementation issues for distributed computing environments. Where possible OMG terminology is followed, the main exception being the use of the term "interface" which is used here with the ODP connotation of "an access point to a set of operations provided by an object", rather than the OMG connotation of a "type". Each interface has a type; there may be several interfaces of the same type in a typical system.

To facilitate discussion of interworking issues, the report is written in terms of a network of "ORB instances" interconnected by a network (i.e. an engineering view) whereas to the applications developer transparency and interoperability will give the appearance of a single distributed ORB. An "ORB instance" is the functionality found in each computer on a network that contributes towards the overall distributed ORB.

A number of examples are given using ANA DPL and ANSA IDL notation. In the next revision of this report they will be converted to using CORBA IDL.

1.4 Scope

Interoperability is related to four architectural issues:

- **naming** - how interfaces to objects are named and how such names are transferred as arguments and results of invocations
- **communication** - how interaction between ORBs is organized, using computer networks
- **interception** - how interaction between ORBs can cross organizational and technology boundaries
- **transparency** - how the engineering details of ORBs and interoperability between them can be masked from applications developers.

To enable interworking between ORBs, the following elements are critical:

- a defined subtype relationship between client interface requirement and server interface provision is a pre-requisite to binding (chapter 3)
- types should be independent from implementations (at least as far as the application developer is concerned) (chapter 3)
- the type system should neither imply nor exclude different kinds of distribution transparency (chapter 3)
- object references should contain sufficient information to enable a client to initiate interaction with server (chapter 4)
object references should not be invalidated if objects migrate or are replicated transparently. (chapter 4)

In summary, the basic requirement is to fully define the semantics of the object (interface) type system implied by the CORBA interface definition language (IDL) and add a binding model to the CORBA object adaptor specification.

With these elements it will always be possible to link different ORBs with an interceptor. An interceptor is responsible for copying object references between ORBs and, optionally, enabling objects to migrate from one ORB to another (chapter 5).
Chapter 2

Interoperability Requirements

2.1 One ORB or many ORBs?

The requirement for the OMG to define the means by which ORBs can interoperate comes from the inevitability of there being many different implementations of CORBA in the marketplace. This diversity is desirable, since it will enable applications developers to select the most appropriate ORB implementation to suit their need.

The diversity of ORBs will come from two factors:

- diversity of underlying technology
- diversity of non-functional requirements on ORB implementations.

2.1.1 Diverse technology

Vendors already plan to produce ORBs on different technology baselines. For example, HP is committed to using the Open Software Foundation Distributed Computing Environment (OSF/DCE) to support CORBA. Sun is committed to using Open Network Computing (ONC). The use of objects in system management may lead to implementations of CORBA based upon Open Systems Interconnection (OSI) standards. Vendors in the telecommunications and real-time control market place may implement high performance ORBs on bare (real-time) kernels.

2.1.2 Diverse non-functional requirements

The purpose of an ORB is to support access to distributed services, presented as interfaces to objects. In addition to its functionality, a service can be characterized by its non-functional attributes, such its security, reliability, availability and so forth. These non-functional attributes are guarantees on how the function will be supported by an ORB.

There is no choice of non-functional attributes that is sufficient to meet all user requirements. It is optimistic to imagine there could be a single generic ORB that can be tailored to provide all possible attributes. This is because many non-functional requirements are in conflict.

For example:

- consistency versus availability
- security versus convenience
- autonomy versus uniformity

7
Applications developers and ORB providers have to decide which of these trade-offs they are going to make. Different choices will lead to different implementations of CORBA.

### 2.1.3 Consistency versus availability

An object's function can be accessed faster by more users simultaneously if the object is distributed over several locations. Unless they are read-only, the copies will rapidly become inconsistent. Coordination protocols that guarantee consistency become a greater overhead as the number of copies increases. The optimum choice between availability and consistency depends upon both the application and the pattern of use.

For some objects it will be appropriate to reduce availability to ensure consistency. For other objects some degree of inconsistency may be tolerated in the interests of availability. A good example is airline flight reservation. There is a highly available but often inconsistent database of free seats for use by ticket agents. There is a less available, but more consistent, database of passenger rosters for check-in clerks.

### 2.1.4 Security versus convenience

Security complicates the means to gain access to a service, and careful control must be exercised over the location of objects. This reduces the freedom available to the application developer and puts constraints on the ORB provider. Security depends upon careful management and monitoring. This inevitably reduces convenience. It may also impact performance adversely.

### 2.1.5 Autonomy versus uniformity

Autonomy is about freedom to choose how things are done. Autonomous systems are flexible and can evolve: they can be tailored to meet individual needs. This has to be balanced against the chaos of complete anarchy. Effective management and migration between systems is easier when systems have much in common. Uniformity can be established through conventions adopted by application developers, or be enforced by an ORB imposing standards.

### 2.1.6 Abstraction versus specialization

Abstraction is about blurring detail and treating different things as if they were the same. Whilst this reduces complexity it reduces the scope for discrimination, optimization and fine grained control. An object model, such as that defined in the OMG Object Management Architecture, can be interpreted at many levels. For example OMG objects may be equated to programming language objects, database objects, operating system processes or even entire applications. ORBs oriented towards these different granularities of object will be qualitatively different because of issues of scale and performance. This is reflected in the small amount of prescription in the CORBA specification relating to Object Adaptors, so that these choices are left open. For interoperability, more detail needs to be given.
2.2 Selective transparency

Most users (including application programmers) are not expected to be familiar with all the detailed complexities of the underlying infrastructure of a distributed system. It is therefore necessary to hide various infrastructure mechanisms from such users. These hidden mechanisms are introduced into the system, as required, in support of various distribution requirements hinted at in the last section. The range of mechanisms possible includes, but is not restricted to, migration to balance loads or reduce interaction latency, recovery from checkpoint to mask failures, passivation (e.g. taking an object off-line until next required) and replication for enhanced availability.

These are requirements for distribution transparency and the mechanisms the provide them are called transparency mechanisms.

Architecturally, transparency mechanisms are object services used by a "non-distribution transparent" ORB to achieve distribution transparency.

To make the kinds of trade-off outlined in the previous section, transparency must be selective, on an object-by-object basis.

2.3 ORB Support for selective transparency

Non-functional requirements such as reliability require supporting mechanisms. There are two choices for the application developer - the mechanisms can be built into the application itself, or the developer can select a specific ORB (perhaps by customizing a generic ORB) to obtain the attributes transparently.

The latter is more attractive to the application developer, since less code has to be written, and the application can be moved unchanged between ORBs offering different transparency guarantees.

This requirement for selective transparency is reflected in the structure of the ANSA engineering model, a simplified form of which is shown in Figure 1.

Figure 1: Engineering Model
There is a basic layer of distribution functionality embodied in nucleus objects. The nucleus is a rudimentary CORBA object adaptor linked to a communications interface. The nucleus provides a basic infrastructure supporting objects that provide transparency mechanisms which in turn define the infrastructure seen by the applications developer.

The simplest example of a transparency mechanism is a stub which marshalls and unmarshalls arguments and results into and out of network buffers. More complex transparency mechanisms can be provided to add transactional capabilities and support for migration or replication. These mechanisms modify the behaviour of both the invocation path and the object adaptor. As an example, in the case of adding transactional capabilities, the invocation path could be modified to enforce a two-phase commit protocol and the object adaptor modified to record state changes in a log.

The application programmer sees the combination of the nucleus and the transparency mechanisms as a high level ORB that meets application requirements.

In ANSA, an application programmer describes a service in terms of its interface description using an IDL and a declarative statement of the guarantees (i.e. implied transparency requirements) upon operation invocation within that interface. The tools used to compile and link applications code build an ORB for the application that includes appropriate transparency mechanisms in both the invocation path (i.e. stubs) and the object adaptor.

In CORBA the internal structure is not made visible as it is in the ANSA engineering model. This is because CORBA is concerned only with portability. The CORBA 2.0 revision calls for information on how to add non-functional capabilities to the architecture. The ANSA engineering model provides a framework for exploring the possibilities. It emphasizes the scope for “plug in” transparency mechanisms that add capability to a basic ORB.

For the purpose of standardization it is sufficient to:

(i) define how an object adaptor permits transparency mechanisms to be loaded, and the types to which the mechanisms have to conform
(ii) define how an object adaptor enables objects to link interfaces to specific transparency mechanisms
(iii) include in an object reference information about transparency requirements of the target object and the protocols by which it is accessed.

2.4 Interoperability between ORBs

There are four levels at which interoperability issues arise between ORBs:

- **Communications - interconnection.** This is about addressing, protocol and encoding matching. Since interaction at this level is in terms of request and reply messages, protocol conversion is usually trivial, except when tight quality of service constraints have been applied. Conversion of data within messages is also straightforward, since the
IDL for the interface specifies what data are to be found in each possible message, enabling translators to be automatically generated.

- **Transparency mechanisms - interworking.** This is much harder since it is about matching non-functional guarantees. It is an area where standards for particular transparencies bring benefits. Note that these standards can be made independent of the communications that support them.

- **Invocation - interoperability.** This is a non-issue since CORBA specifies the structure of types. Assuming type-safe binding, invocations arriving at an object will be in the appropriate form and contain appropriate arguments. The invoked operation will return appropriate results.

- **Services - integration.** Since IDL only specifies the signature of operations and not their semantics, it is possible to bind to an object that supports operations of the right "shape" but which do not behave in the way expected. Programmers tend to believe in the functionality implied by the name of an operation, but in reality the name is arbitrary. To interconnect services with confidence, the meaning of names (or translations between them) must be agreed by both interconnected parties.

### 2.5 Boundaries between ORBs

It is inevitable as distributed systems grow in extent that they will cross organizational boundaries, and to some extent at least the boundaries must be visible in the enlarged system. As organizations do business with one another, it is equally inevitable that autonomous systems based on diverse technologies will need to interconnect. The result in both cases is a **federation** of domains.

A federation consists of a set of domains that have established a common contract for interactions between the members of the federation. An important property is that of **autonomy**: the members have freedom to join and leave the federation at any time, and to select which services from their own domain are available to the other domains, and which services from those domains they make available within themselves.

An **administrative domain** consists of a set of objects whose security, accounting, monitoring and other management functions are under a single administration.

Between administrative domains, either or both administrations may wish to impose their own access controls for such purposes as security, accounting and monitoring in addition to controls imposed by the objects themselves. Administrative boundaries are also the points where changes of management responsibility take place for such things as resource allocation and dependability guarantees.

**Technology domains** consist of a set of objects supported by ORBs which have common protocols to enable communication between them.
Within a technology domain, ORBs have identical representations and/or functionality of protocols, object naming and addressing. Between technology domains protocol conversion and name translation is required.

A technology domain is generally established by managers exercising a procurement policy and/or registration authorities controlling the use of a particular technology. A technology domain may include several diverse technologies, and ORBs within the domain support interaction and conversion between them. In a technology federation, neither party is prepared to (or has the ability to) introduce the other domain's technology into its own.

To meet the autonomy requirements of federation some form of interception is required between the domains. Interception consists of some or all of access control, monitoring, accounting and translation.

To extend the CORBA architecture to include interoperability, two aspects must be addressed

- enabling an ORB to use several technologies simultaneously within a single technology domain
- using interception to cross administrative and technology boundaries.
Chapter 3

Binding

3.1 Introduction

Binding is the process by which an activity in one object establishes the ability to invoke operations at an interface to some other object. Having a binding does not guarantee interaction is possible. Interaction may be disrupted by failures in the path between client and server object.

Binding is a complex issue involving trade-offs between the guarantees offered to the application programmer, management of state in the client and server object ORBs and use of communications resources. For ORBs to interoperate, the ORB object adaptor should provide an abstraction of the binding process that insulates applications from the details of the underlying mechanisms.

The simplest, practical interaction model on which to base binding is one in which:

- the client selects the server interface it wishes to use
- the client selects an operation in the server's interface by name to begin an invocation
- the client can include a set of arguments with the operation name
- the server selects an outcome (a "termination") to be signalled to the client on completing the operation, where the outcome is a name and a set of result parameters
- formal arguments and results are bound to object interfaces.

The model reflects the symmetry of communications in terms of request-reply structures.

The ability of the server to select one of several possible returns enables interfaces to be defined in which "values" are represented by the use of particular termination names. For example Boolean values can be represented by interfaces with signatures conforming to

\[
\text{Boolean} \rightarrow \text{type} \ (\text{Test} () \rightarrow \text{True} () \rightarrow \text{False} ())
\]

That is, any type which provides an operation called Test, which takes no argument and produces either the termination True, or the termination False.

Formally speaking, this is a simple "abstract data type system". Abstract data types are known to be a sufficient basis for defining any data types.
ORB Interoperability

The interworking requirements of abstract data types are very simple:

- translation of operation name formats
- translation of termination name formats
- translation of object reference bindings.

Most importantly, there is no requirement for any set of concrete data types to be built into the type system as a pre-requisite to interworking.

3.2 A model for interface implementation

The currency of the abstract type system is the interface. Interfaces are the elements that are named, assigned, invoked and communicated. To meet basic requirements for distribution transparency, application developers need to be able to name, assign, invoke and communicate both local and remote interfaces indistinguishably. However, the ORB implementor cannot avoid the distinction and has to provide the desired transparency.

The means of doing so are outlined in Figure 1.

An interface is, in basic terms, a data structure containing the object state associated with the interface and pointers to the procedures that implement the operations in the interface. (Referencing the procedures indirectly allows multiple instances of the same interface type to share them).

A local interface is therefore represented by its local address. This can be assigned to a name (e.g. an identifier in a program) and communicated as a result or argument in invocations of operations on local interfaces.

When a local invocation occurs, the addresses of the interfaces nominated as arguments are passed to the server. The client is effectively sharing the interface with the server, since knowledge of the interface’s address is passed on. Result parameters work the same way and the client learns about an interface known to the server. Both client and server can invoke operations in the interface and any updates made by one will be visible to the other.

Access transparency requires that the same behaviour occur in the case of interaction with a remote interface.

A remote interface is represented by the address of a proxy interface which embodies the information and procedures needed to communicate with the remote interface. The proxy interface is identical in structure to a local interface. It contains the information needed to establish communication with the server (called an interface reference) and a set of stub procedures, one for each operation in the remote object.

The function of the stub procedures is to convert local arguments and results (i.e. interface addresses) into interface references. Interface references are internal to the ORB and should not be seen directly by the applications programmer.

This structure means that from the point of view of a client, local and remote interfaces are represented identically. They can be assigned to names, passed
as parameters or results and be invoked identically. This is possible because in both cases, access to state and operations is indirect: one level of indirection in the local case, two or more levels in the remote case.

Locally, invocation is in terms of procedure calls with interface and/or proxy interface addresses as arguments and results. Remotely, invocation consists of requests and replies, where requests contain an operation and a number of interface references and replies contain a termination name and a number of interface references.

At the server side the ORB receives requests and turns them into invocations of the local interface. The function of the ORB is to decode the request, unmarshall parameters, make the local invocation, marshall results and reply.
Marshalling a local interface address consists of:

- creating an interface reference for the interface
- transmitting the interface reference.

One it has been marshalled for the first time, the interface reference can be cached in the interface state. This is not strictly necessary, but may help the local ORB conserve communication resources, such as network endpoints. The local object makes no use of the reference itself.

Marshalling a proxy interface consists of transmitting a copy of the interface reference in the proxy's state.

Unmarshalling an interface reference consists of:

- creating a local proxy with the interface reference as part of its state
- returning the address of the proxy.

The interface reference generated by the server must give the client sufficient information to enable it to send a request and receive a reply when a proxy is invoked.

Some of the information in the interface reference may need to be included in each request by the client so that the remote ORB can identify which local object to invoke, or it may be implicit in the network naming scheme of the particular protocol used.

The integrity of interface references is critical to correct functioning of a distributed system. Interface references should be invisible to applications programmers and only visible to ORBs. To the applications programmer, only bindings to local interfaces (either genuine local interfaces, or proxies for remote interfaces) should be visible.

Testing bindings to interfaces for equality is specifically excluded since the obvious interpretation of comparing of local addresses is meaningless in the remote case. Equality for instances of a type should be defined in terms of the semantics for the type since in general "substitutability" is sufficient.

3.3 Discovering interfaces

An ORB object adaptor must provide the means to instantiate objects and interfaces from templates.

Basic object and interface creation cannot be an object service, since this would beg the question of how the service creates the object or interface, however, ORBs can host factory services which provide greater control over object and interface creation than the object adaptor, and perhaps make the function accessible remotely.

Creating a reference when the interface is created is wasteful if a binding to the interface is never to be marshalled. Deferring creation of the reference until a local binding is marshalled adds a tiny increment to the cost of marshalling. Late conversion is the sensible default.
An object can discover an interface as the result of an operation invocation. This was discussed in the previous section. The semantics of argument and result parameters are "call-by-sharing".

Name and trading services provide a means for an object to pass knowledge about interfaces to other objects including those that are not available (possibly haven't even been created) at the time the reference is passed.

A name service provides bindings between names and interfaces. A trading service provides the means to identify interfaces by type and properties. These are extreme examples of services which store and retrieve interfaces but do not invoke them. It would be wasteful to pin down communications resources for such interfaces.

Communication resources should only be allocated to interfaces when operations in them are invoked.

Services which only store and retrieve bindings to interfaces can cast the interfaces to the type "Top" which has no operations. The local proxies for the interfaces will then contain no pointers to marshalling procedures and the representation of the remote interface will be simply details of its type and its interface reference, optimizing the storage resources required to remember the interface.

### 3.4 Type checking

Binding should be type-safe.

This ensures that trivial interaction errors arising from invoking the wrong kind of interface are excluded, and that distributed applications can be checked in advance to avoid "message not recognized" errors at runtime.

Type-safety makes trading more attractive than name serving. A trader retains type information and can do type checking, whereas a name server typically does not. An object reference obtained from a name server must either be trusted to be of the right type, or else the name service must be restricted to holding bindings for interfaces of a specific type.

A trader has to believe the type assertion about an interface made when the interface is exported. This is an authentication question, and the trader may require that the type assertion be signed by the compiler that generated the marshalling code.

To enable trading it must be possible to transmit signature information about all IDL types, both concrete and abstract. The type "Type Signature", more boldly, the type "TYPE", needs to be defined in IDL. As part of doing so, the range of types that may be wrapped in an IDL Any should include typed interface references.

A trader has to match a type required by an importer to a type offered by an exporter. Requiring exact identity of types is too restrictive, in particular it prevents types from being extended. The requirement is that the imported interface has a type which is a subtype of that requested.

The minimum requirement for sub-typing is that:
the offered interface includes at least the operations required
for each such operation, the operation returns a subset of or exactly the terminations required
for each such operation, the argument types required are super types of the argument types offered
for each such operation, the result types offered are super-types of the argument types required.

These rules can be summarized as “no surprises” to either client or server when invocations are made and results are returned. They allow a server to add operations to interfaces, or to “widen” argument types without requiring changes to existing clients. The rules can be checked by program; this is safer than relying (for example) on programmer assertion that one type is a revision of another.

Sub-typing rules should be defined as part of the CORBA revision, since the requirement to support use of sub-types has implications for the encoding of operation names and terminations. In particular, simple schemes based on indexing operations within an IDL specification are not sufficient.

3.5 Establishing communications

An interface reference is the information needed to establish communication - before an invocation can be made the information implied by the reference must be used to connect a communications path from client to server.

Setting up a communications path is a separate process from trading, or argument and result passing.

The least restrictive connection model is one in which communication resources are only tied down whilst the request and response are in transit. In other words the connection need not persist between interactions. This kind of connection is implicit and requires no action on the part of the client or server application.

Implicit connection should be provided as the default connection model by an ORB, for all interface references.

The implications of this simple model are that:

- every invocation, in addition to the terminations named in the signature of the invoked operation, can give rise to a termination indicating connection failure
- an interface reference must contain sufficient information to enable the creation of a new connection at the client’s initiative.

For some kinds of communication, such as datagrams, implicit connection is very efficient, since the connection information can be piggy-backed on the request itself.

For other kinds of communication, such as those based upon byte streams (e.g. OSI connection-oriented protocols and TCP), there is additional latency implied by setting up connections. The overhead of setting up and tearing
down on every invocation may be prohibitive. It may be better to cache the connection for use by future invocations, but this must be tempered by the resources tied down by each retained connection. These are quality of service issues and must be jointly agreed by client and server. They cannot be decided unilaterally.

When a server creates an interface, the server must state the range of quality of service options it is prepared to honour. This information must be carried along with the interface reference as a profile of the protocol families and quality of service options allowed within those families.

When a client binds to an interface it must select one of the quality of service options. When binding is implicit, this decision is deferred to the ORB.

In an ORB using stub technology, the stub generator can take as input both the signature of an interface and information about the quality of service options to be offered (accepted) in the case of a server (client) for the particular interface - enabling different options to be selected for different local interfaces.

In addition to a default of implicit connection, connection can be made explicit, by providing an object adaptor function to do so. This is useful when the application programmer wants to exercise greater control over the use of communications resources, establish quality of service guarantees in advance or remove the latency of the connection process.

The explicit connection function is parameterized by the binding to be resourced, and the quality of service option to be applied.

Alongside explicit connection, explicit disconnection is required to free up resources once they are no longer required.

Explicit connection does not necessarily rule out the risk of a connection failure when the reference is used. It depend upon the particular quality of service demanded. All that has been done is to separate the connection process from the invocation process.

In both implicit and explicit connection, interworking is achieved between objects using one of the protocol profiles they have in common, selected by the client, without an explicit negotiation with the server. Any negotiation that does occur is consequential to the client making the choice of a protocol that needs end-to-end confirmation. This is desirable in a highly heterogeneous system as a negotiation to initiate connection would require a common protocol, and which one should be used?

Only when the connection has been established end-to-end can interaction between client and server take place.

If there are no common protocol profiles, then it is impossible to communicate with the server. A binding error is the result if the client attempts to communicate with the server.

The choice of quality of service requirements by the client as part of connection may establish a single quality of service that is applied to all interactions using the binding, or there may be a range of options left open. The client must specify an exact requirement for each invocation, or accept a default.
3.6 Garbage collection

The simplest persistence model is one in which objects remain accessible whilst some other object has a binding to one of its interfaces. If the binding is local, local garbage collection can detect when interfaces are no longer referenced and delete objects as appropriate. If an interface has been marshalled, there may be remote objects holding references to it, and the object cannot be deleted while these reference remain. Tracking interface references reliably in a large system is extremely difficult and a topic of current research.

An object adaptor should provide the means for an object to cancel its interfaces. Attempts to invoke operations in a cancelled interface produce a connection failure. The resources associated with a cancelled interface can be reclaimed.

Distributed garbage collection is not impossible. To work effectively and overcome failures it must be possible for one ORB to enquire of another which of its references it believes are still valid. Both local and distributed garbage collection can be positioned as an object service, provided that information about external interface references is available.

An object adaptor should provide a function to reveal which local interface references have been marshalled as arguments or results, and which object references have been received as arguments and results from other ORBs.

3.7 Object migration

The simple interaction model, if implemented naively, would suffer from delays due to network latency, since passing references only conveys knowledge about objects rather than the objects themselves. The interaction model is one of “object sharing” rather than “communication”.

Latency can be overcome by moving objects between ORBs. Since the abstract type system can represent values, there is no need to extend the interaction model to add migration. It is sufficient to:

- agree a conventional management interface for objects, containing operations such as `Object.migrateTo (Location)`
- agree a convention that all interfaces provide an operation to obtain a management interface for their encapsulating object
- define “Locations” as services which accept some abstract representation of an object and reconstitute the object at the ORB hosting the “Location”
- provide object adaptor functions to enable objects to unlink and relink themselves from an ORB
- provide relocation services to help clients and servers track migrating objects.

With the exception of the object adaptor support for disconnecting and reconnecting objects, migration is a matter of providing appropriate Object Services.
Means for making the migration transparent to clients are discussed in Chapter 4.

The object adaptor support for linking and unlinking objects from an object adaptor require:

- **linking** - the ORB externalizes the interface references associated with bindings in the object
- **unlinking** - the ORB internalizes an object template and sets up local bindings for all interface references in the template.

Having an interaction model based on sharing and object migration is sufficient for distributed computing.

Sharing preserves the encapsulation of the object involved. It need not leave its parent ORB and therefore sharing does not change the security, reliability or availability of the object. The object's operations can contain code to defend it against misuse or damage - for example by imposing authentication checks.

Migration exposes the object to greater risks. It may cross a security boundary. The destination may be less reliable. By moving nearer to one client the object may become less available to other clients. Migration must be managed carefully. It is appropriate that it be made available through a management operation on the object involved so that the object can decide if and how to migrate. It is appropriate that the supporting functions be pitched as Object Services so that all of the mechanisms of security and dependability can be applied to them (see [TR.44] Specification of ANSA Storage Model).

When a object is migrated, its local interfaces at the source will become proxy interfaces; if there was a proxy at the destination it should become the (migrated) local interface.

An extension of the use of migration to move an object from one ORB to another is to use replication. The mechanisms will be essentially the same, with the additional requirement that the original object and its replicas converse amongst themselves to ensure consistency. Replication can increase both reliability (since there are more copies) and availability (access can be made to the nearest copy). Replication can be made transparent (see A Model for Interface Groups [AR.002]).

If an object is immutable (e.g. its state is read-only), copying is sufficient for migration or replication since the copy cannot diverge from the original. This can be a significant optimization. Copying is also sufficient for sharing, but only if the security constraints on the object permit the target to host the copy.

Copying is an attractive optimization since, for small objects particularly, it is cheap and reduces latency considerably. The optimization can be completely transparent to the applications programmer. The optimization can be implemented in a number of ways including:

- (dynamically) interfaces being upcalled by the ORB to marshall themselves
- (statically) stub generators being told which types to marshall by copy rather than by reference.
Both client and server must be consistent in the use of this optimization - in effect a presentation syntax agreement must form part of the connection between the two.

3.8 Interpretation of CORBA IDL

CORBA IDL can be easily re-interpreted in the guise of the simple abstract datatype model set out above:

- object references are abstract datatypes with "call-by-sharing" semantics; the engineering of sharing (reference passing versus migration versus replication) is not fixed
- the values defined in CORBA IDL (integer, boolean, etc.) are a built in set of optimized immutable interface types for which the engineering is fixed to be "call-by-copy".

The distinction between passing an interface by "sharing", and by "copy" is simply illustrated. Suppose a String object containing the text "Object Management Group" is passed by reference. The receiver could use an operation such as String.Replace ("Group", "Team") to effect a change to the value of the string that is visible to all objects having a binding to it. If, on the other hand the string had been passed by "copy" (i.e. the string type is defined to be immutable), the change would only be visible to the receiver.

CORBA IDL language bindings should make best efforts to equate the CORBA data types to immutable data types in the specific language, if it has them, or if not warn the applications programmer of the difference to object references.

The CORBA description should note that passing "network" pointers is not the only way of achieving sharing.

Since encoding is a link by link issue, there is not a strong requirement to agree an extensive range of pre-defined types. If there are changes to the type system, they should be to increase its expressive power, for example, by introducing generic functions so that types such as List[TYPE] can be programmer defined.

It is undesirable to add a concrete "pointer" data type to IDL since it is unclear to what depth pointers should be traced in recursive data structures. It is better to make complex data structures abstract and rely on each object in the structure to take its own choice of how best it should be "copied".

3.9 Relationship with ANSA and ODP

The requirements on CORBA IDL and object adaptors set out in the previous sections are essentially amount to presenting a programming model for distributed applications using the concepts of the Computational Language in the Basic Reference Model for Open Distributed Processing and the ANSA Computational Model [AR.001]. (The ODP Computational Language is less detailed that the ANSA model, but does include a treatment of explicit binding and handling references to "stream interfaces", which are not covered
in AR.001. Some aspects of stream binding are presented in [TR.12] Support for Multi-Media Operations.)
4.1 Introduction

An interface reference is the engineering information needed to establish a connection between a client object and a server object on different ORB instances. This chapter describes the kinds of information that must be present in an interface reference and how it is used in establishing a connection.

An interface reference must contain information that:

- names, in the client ORB's context, the endpoint to which requests should be sent
- ensures the integrity of the end-to-end binding against errors in transparency and network management mechanisms
- supports transparent replication of the service to which it belongs
- supports transparent migration of the service to which it belongs.

4.2 Interface references

The following features of interface references are determined by the requirements set out in the previous chapters:

- the naming scheme employed in interface references should be completely context relative
  This gives an evolutionary approach to systems integration and makes it possible to combine new applications with existing applications (legacy systems) and to enable interworking over a range of distributed systems platforms. The interface reference may not reflect any one naming policy of any one of the systems it is to operate over. This can be achieved by adhering to context relative naming throughout.
- an interface reference should be able to include addressing information from more than one distributed system platform
  This is necessary to ensure that the validity of the interface reference is preserved when it is passed from one platform to another platform.
- an interface reference should contain the information that reflects what protocols and quality of service options are supported by the server that created it.
  For a client to be able to use a remote interface its ORB must use the
same protocols as the server (or an intervening interceptor). The client ORB must therefore have knowledge of what protocols to use before actually interacting with a particular server. Where a server supports more than one protocol stack, information about each stack may be included, so that the client ORB is offered a choice.

- **an interface reference contains the information that reflects the context relative network address of the server endpoint.**
  
  For the communications protocol to be able to route messages to the server, it must know the network address of the server network endpoint, relative to the client's position in the network. The interface reference holds the context relative address of the server. If an interface reference is passed from one network to another, then the addressing information is updated by the interceptor at the network boundary. Global addressing cannot be assumed.

- **an interface reference contains a nonce that can be used to perform an end-to-end check**
  
  When a request arrives at a server, it is necessary for the server to check that the request arose from the use of an interface reference that was generated by the server, and not by the interface reference of another server. This is necessary because where servers are mobile, the location of a server offers no guarantee about the kind of service or the service instance that is provided. The end-to-end check is performed in the context of the connection that has been established with the help of the address and protocol information.

- **each interface reference can contain information which can be used to find services which have migrated**
  
  When a server migrates to another location, all interface references previously issued become suspect since the server's address has changed. Since the server has no control over interface references once they have been given out to others, this can lead to considerable disruption, unless the interface reference contains information that can be used for both detecting the change in location and relocating the interface.

- **the structure of the interface reference should accommodate the possibility of a service being offered by a group**
  
  Where a service is actually (and transparently) provided by a group of servers (replication to increase dependability for instance), the interface reference should appear as if a single server is involved from the client application point of view, but contain all information to allow the orderly progression of group execution protocols (possibly using multicast messaging) by the client ORB.

- **an interface reference is not a capability**
  
  An entity that obtains an interface reference may use it in order to try and invoke operations on the service that generated it. The service may however decline to respond (for security reasons for instance).
4.3 **Information content of an interface reference**

An interface reference consists of three components:

- group data
- a nonce
- for each member in the group a member record
- a sequence of relocator interface references.

Group data must give information about the current membership of the group.

The nonce gives an end-to-end check of the integrity of the binding and transparency mechanisms between ORBs. It is a random number chosen by the server ORB when the interface reference is created. The server will refuse to establish a connection with a client that cannot quote the nonce for the interface being connected. These failures should be reported to system management since they are indicative of a configuration error in the system. Since nonces are random and generated locally by each ORB they provide an overall statistical integrity check, not an absolute guarantee.

Each member record consists of:

- a sequence of address records.

There is an address record for each set of protocols that can be used to access a service. The address record is interpreted in the context of the distributed systems infrastructure which supports the server.

Each address record contains a stack of tuples, each consisting of:

- a protocol identifier
- a protocol endpoint.

Protocol identifiers and protocol endpoints are context relative *names* which have meaning within the context of a specific subnetwork and distributed systems infrastructure.

The nonce is *not* used as a global identifier. The nonce is *not* used in any name resolution process either. If the client and server hold different nonces on a set of end-to-end bindings, then the bindings are broken. Matching nonces provide no guarantee that things are right however! Thus the nonce is only used in the context of an interaction on an end-to-end path determined by the address records.

4.4 **Connection**

The requirements on the connection arising from the requirements in previous chapters can be summarized as follows.

A server object's interface is prepared for connection when

(i) the server's ORB has been set up with the necessary communications endpoints, such that incoming connections can be received and results returned,
ORB Interoperability

(ii) an interface reference has been created, which contains the addresses of these endpoints.

A client is prepared for connection to an interface when

(i) an interface reference has been received which specifies the endpoints to connect to,

(ii) a proxy interface the client's application name space has been locally bound to the interface reference,

(iii) and either the client ORB has been set up with the necessary endpoints, such that calls can be made and results received or such endpoints can be set up immediately before interaction actually takes place.

4.5 Relocation

To allow flexible configuration and resource management policies, servers should be allowed to migrate freely from one location to another. As servers relocate clients may experience disruptions in service provision. Transparency mechanisms are put in place to hide these disruptions from application code. In designing these mechanisms, care should be taken not to break the binding rules.

A client's ORB, upon detecting the absence of a response from a server, may attempt to reconnect using a different address record. If the service has migrated, then none of the address records will yield a successful interaction. In that case the client ORB uses information contained in the interface reference to obtain a new interface reference, to reconnect and then continue interaction. All (re-)connection and access to location services is performed by the client's ORB, invisible from the application, and completely within the context provided by the local ORB. Client application code should experience no more than a delay in response.

The server nominates a relocator and includes the interface reference of this server in its own interface reference. There is no reliance on a "well known" address, such as that of a name server, as this is a solution which does not scale to very large systems. Moreover it forces all objects to adopt the same relocation system.

Relocators are objects. They may migrate as well and their interface references can contain interface references for "relocator-relocators" which can in turn be consulted.

When migration is used liberally, the likelihood that a particular location is visited by two or more interfaces who have each nominated the same relocator, increases. In that case neither the location, nor the relocator, nor the combination of the two provides sufficient context to disambiguate the entries for the interfaces in the relocator database. Two solutions to this

---

1. The resulting migration transparency may be relaxed and applications may be allowed access to relocators.
problem can be provided: either the time at which the interface was bound to 
the location or the relocator itself can be used as a disambiguating context.

4.5.1 Solution 1: Use of interface creation time

As no two interfaces are ever bound to the same location at the same time, 
each endpoint\(^1\) is associated with a monotonically increasing counter, 
advanced each time an interface reference is bound to it. The location of 
creation, together with the value of the counter are sufficient to disambiguate 
relocator database entries. Such a counter can be derived from a local stable 
clock.

4.5.2 Solution 2: Use of the relocation context

An alternative solution is to have the relocator issue keys for retention in 
interface references:

- when the server infrastructure constructs the interface reference for the 
  first time, it asks the relocator, whose interface reference it is about to 
  include, for a key;
- it includes the key in its interface reference, together with the relocator 
  interface reference;
- the key obtained from the relocator is strictly valid in the context of that 
  relocator only;
- the relocator assigns keys once only, thus ensuring they are 
  unambiguous;
- any clients who contact the relocator pass the “old” interface reference, 
  which contains the key (as before).

The advantage of this scheme is that servers can only insert the interface 
reference of a relocator if that relocator has agreed to perform the relocation 
function. It effectively agrees by handing out keys. Thus there is a measure 
of control on the amount of business a relocator is willing to get involved in, 
which is missing in solution 1.

A disadvantage is that the relocator could become a bottleneck for key 
allocation if the number of interfaces it looks after is very large. However, it 
can be expected that system designers would guard against this and 
distribute the relocation service instead of concentrating everything in 
place\(^2\). Furthermore there is a possible optimization by allowing a series of 
keys to be allocated at once, thus requiring only one relocator access per set 
of interface creations.

---

1. All the endpoints accessible to a single ORB can share the same counter.
2. The distribution policy for relocators may result in a relocator for all objects (1) on 
a particular computer, (2) using a particular ORB (instance), (3) belonging to a par-
ticular principal (owner), (4) all interfaces of a particular type, or some other classi-
fication.
Chapter 5

Interception

5.1 Introduction

This chapter explores the use of interceptors to achieve interoperability between ORBs that do not support overlapping protocol profiles for interworking. Interceptors are objects that impose controls and perform translations when boundaries between administrative and/or technology domains are crossed. Interception is possible if the CORBA specification gives precise semantics to the implied IDL type system and the binding model assumed by object adaptors.

Particularly in the case of interception at technology boundaries it is possible to arrange that interception is fully dynamic and transparent. Dynamic in the sense that having configured the interceptor between to different ORBs, the interceptor can cope with any type of interface that might be accessed through it, and transparent in the sense that the applications programmer is unaware of its presence.

Interceptors can be positioned either as ordinary objects, in which case they can draw on other transparencies and object services and be managed identically to other objects. Alternatively they can be more closely integrated with ORBs and network protocols, for reasons of optimization.

Interworking between ORBs with overlapping protocol profiles can be viewed as an optimized case of interception. Rules for interception are the architectural foundation of interoperability.

5.2 Interceptors

An interceptor can be either fixed or nomadic. Fixed interceptors provide immediate access across administrative and technology boundaries while nomadic interceptors enable an object to be disconnected from one domain and then subsequently reconnected in another.

Fixed interceptors correspond to the notions of "gateway", "agent" and "monitor": objects which stand between two domains and enable or permit interactions on the basis of a contract between the administrations specifying the basis for their federation.

Nomadic interceptors are required to track objects which are moved from one location in a system to another by media exchange (e.g. on a floppy disc sent through the post, or stored in a mobile computer which has moved between
domains). A nomadic interceptor enables the tracking by taking part in the process of reconnecting the exchanged medium back into the system.

If objects in different domains require a transparency mechanism in order to interact, and that transparency mechanism is absent from one of the domains, then such interactions cannot cross the domain boundary. Interactions which only require functions supported by both domains are unaffected by the presence of a boundary for transparency functions they are not using.

*If transparency mechanisms are themselves implemented as object services, interception can be used as a basis for interoperability between alternative transparency mechanisms.*

### 5.3 Interception

Interceptors mask boundaries between domains and are access transparent. Interceptors perform the checks, renaming and transformations required to make boundaries invisible to applications.

Administrative boundaries require controls to be imposed and records to be kept. Technology boundaries require transformations to be made.

#### 5.3.1 Fixed interceptors

Fixed interceptors provide transparent access to interfaces and object migration across boundaries.

They must be capable of intercepting invocations and the transformation of arguments and results.

Fixed interceptors are bi-directional and symmetrical. They must be part of the communications path (connection) between the interacting objects.

The interception may be at the source, at the destination or at an intermediate location.

*If only an administrative boundary is being crossed the interceptor applies administration specific access controls and re-assigns responsibilities for security and dependability.*

*Where the interceptor is to be trusted to fulfil protection responsibilities for an administrative domain, for instance in the translation of security information, it must reside entirely within the domain.*

*Where there is administrative interception on both sides of a boundary, there may be interaction between the ends to exchange and agree relevant information such as cryptographic keys independent of the interception of particular interactions. These exchanges carried out by administrative interceptors may be applied to several successive interactions between objects without further negotiation. In other words, administrative negotiation can be out-of-line.*

*Technology interceptors carry out protocol and data translation over technical boundaries; they are involved in all object interactions over the boundary.*
These are also known as "gateways" or "protocol convertors". For efficiency only one in-line interceptor would be used for each technology boundary.

5.3.2 Nomadic interceptors

Nomadic interceptors provide a batched store and forward migration service via removable media or arising from the disconnection and subsequent reconnection elsewhere of mobile computers.

Removable media may be used to store objects. Stored objects may contain references to objects on other media, or still active at an ORB. Similarly objects off the media may have references to objects on the media. There may be internal references between objects on the media.

When the media are removed from the location in a system and reconnected at another, the references must still be valid. This introduces three requirements:

(i) to ensure that objects on the media can still be located by other objects off the media holding references to them

(ii) to ensure that objects on the media, when activated, can locate the objects both on and off the media to which they hold references

(iii) to ensure that the references retain the same meaning at the destination as they had at the source.

Furthermore, the representation of stored objects may be different at the two points. There is a potential for conflict if the source and the destination are in different name spaces, since identifiers within the object may have different interpretations in each name space. These conflicts can be avoided by ensuring all names carry their context with them. The term "repair" is used to denote whatever transformations of interface references are required to meet the location and disambiguation requirements.

Object representations may have to be transformed when the medium is mounted in a sub-system with a different technology from the one in which the objects were created.

Objects which require access controls should only be written to media which are subject to the same security policy as their origin.

References within the media must be repaired when the media are mounted at a different location from the one at which the references were created.

References off the media must be repaired when the media are mounted in a sub-system with a different technology from the one in which the references were created.

References onto the media must be blocked while the media are off-line and repaired when they comes back on-line, with the help of relocation.

A nomadic interceptor has the option of doing a batch repair of all objects and references on removable media when it is mounted or of intercepting objects as they are being activated or transported from the media.
5.4 Translation

Interface types are abstract and can thus be implemented in any concrete protocol with the ability to represent the information content of an interface reference (sequences of bytes are sufficient). The transparency requirement is that transformations can be made between any pair of protocols which are both capable of supporting the guarantees requested by the applications.

Objects in the source system which cannot be migrated to the destination system must be instantiated in the interceptor as an object encapsulating the object and an interface reference transported onwards.

Objects that can be represented as data in the destination system but not in the source system can have their interface reference transported in both technologies.

When an object migrates across a boundary then the representation of the object must be intercepted. For administrative boundaries the functions required to be performed on a intercepted object are equivalent to those for interface references. For technology boundaries an intercepted object must be transformed into the correct representation for the destination technology. This could be done by translating into a common intermediate form derived from the Open Software Foundation’s Architecture Neutral Distribution Format (ANDF).

Interface references can be transformed by editing them to provide redirection via a protocol-to-protocol converter or by capturing them in a proxy interface and transmitting a reference to the proxy (i.e. an application level gateway).

Redirection can be done by prefixing the original endpoint address in the interface reference with the address of a generic invocation forwarding service built into the interceptor. When this is subsequently invoked with the transformed interface reference it uses the remainder of the new address (equal to the original address) to invoke the original service.

A proxy interface is simply an invocation forwarding service made specifically for a particular interface by including the original interface reference.

Interceptors can sometimes bridge the functionality mismatch between different technologies by providing the missing functionality themselves.

5.4.1 Type descriptions

In order to perform the required interceptions, interceptors need access to the relevant type descriptions of the interfaces to be accessible across the boundary.

The type description of an argument or result parameter is contained in the type description of the interface containing the operation passing the argument or result (i.e. they are part of the same type description).

If objects are bound together from an initial static configuration, their types will be known in advance and appropriate interceptors can be included in the configuration. In an open configuration, objects get to discover one another via trading (see [AR.005] The ANSA Model of Trading and Federation),
therefore in general the roots of type descriptions are the type descriptions of traded interfaces.

Type descriptions can be encoded as concrete data types, or as (interface) references to type description objects held in an interface type repository (which could itself be a distributed object).

In-line interceptors must intercept invocations of trading import operations which cross their boundaries and record the type description of the required type.

The possibility of encoding an interface's type description as part of its interface reference as an alternative to treating interception of trading specially is rejected because it would significantly enlarge all interface references.

The required type is recorded by the interceptor rather than the provided type because this is all that is needed to define the interactions and it is the one that is exchanged as an argument of the import.

This means that:

(i) once a type has been traded then all the types in its signature can be intercepted

(ii) no type can be intercepted until it has been traded itself or is included in the type description of a traded type.

This then leaves the problem of who trades the trader.

If a trader is accessible through an interceptor then that can be used to trade subsequent traders, but the interceptor must provide special initialisation functions to trade at least an initial trader across itself in both directions.

This initialisation can be done by having the interceptor federate traders on either side of its boundary, by having the interceptor become a proxy trader or by building a trader into the interceptor itself. Initialising an interceptor is analogous to mounting a nomadic interceptor.

5.5 Building interceptors

Interceptors may be constructed either as transparency functions or as "applications".

Interceptors constructed as transparency functions must be type transparent and capable of intercepting interfaces of any type.

They will read the type description information exchanged during trading to set up appropriate translations when invocations of the traded interface references occur. This may be done by looking up in a table of stubs generated when the need for interception was set up, or by encoding the required transformation as a "format string" to be decoded by a generic translator. The conversion itself may be done either at the protocol level, setting up a proxy address on the interceptor, or at the application level by setting up a proxy interface.
Nomadic interceptors will generally be implemented within operating systems as part of the machinery for mounting new file systems or databases onto the local storage system. The mounting process will require the user to give information about the context from which the mounted object(s) originated so that references can be repaired correctly.

5.5.1 Garbage collection

Interceptors must cooperate with interface reference garbage collection because they hold extra (non-computationally visible) references to intercepted interfaces.

In-line interceptors can be of great help in limiting the scope of garbage collection sweeps by keeping records of which interface references have crossed which boundaries (and by implication, which haven’t).

Nomadic interceptors must ensure that interfaces which are referenced from objects stored on removable media are not garbage collected while the media are off-line.

5.5.2 Engineering issues

Implementing an N-way boundary in a single interceptor is an engineering optimization that can only be performed if it preserves the semantics of the 2-way boundaries.

In general, boundaries are N-way because more that two sub-systems may meet at the same interceptor. But logically, an N-way boundary can always be modelled as N 2-way boundaries, so architecturally, only this case needs to be considered.

Administrative and technology boundaries will frequently coincide and implementing them both in a single interceptor is an engineering optimization that can only be performed if it preserves their distinct semantics.

Responsibility for the management of migrated objects lies with the destination administration because its resources are being consumed; it is divorced from the administration of the source object, because both objects are distinct.
Chapter 6

Implementation guidelines

6.1 Introduction

This chapter offers guidelines on the implementation of interface references and relocators. It adds detail to the outline given in chapter four.

6.2 Endpoint identification

The creation of an endpoint to support a server interface is a function of an ORB.

Many operating systems allocate endpoints internally and provide no user control over the naming of the endpoint. Since an object may migrate, an interface may be associated with several different endpoints throughout its life.

A node may be connected to more than one network, and within each such network there may be several protocols available. Therefore an interface reference must have the potential to indicate several alternative endpoints by which the interface can be accessed and which protocols are associated with which alternate (i.e. the address record outlined in section 4.3).

Protocols are identified by protocol names. A set of interacting ORBs must agree the binding of protocol names to protocol implementations. Interaction between ORBs where such agreement does not exist is a federation problem and must be overcome by an interceptor.

Many network architectures provide the means to register protocol names. Where possible ORB protocol naming contexts should be aligned to such network naming contexts.

Where an ORB supports more than one protocol, it is common to find a layered approach to naming so that higher level protocols (e.g. transport protocols) can share common lower layer protocols (e.g. a network layer protocol). This often leads to addresses being structured as sequences. For example there may be a network layer address that selects a network interface and a transport address that selects a particular connection through that interface. The endpoint name of an interface is then a set of alternative composite endpoint names, where each composite endpoint name consists of a sequence of protocol name, endpoint pairs.

Quality of service negotiation involves all the layers in a layered protocol environment (because higher layers cannot always recover attributes 'thrown away' in lower layers). Quality of service information can be captured in an
interface reference as sets of **quality of service ranges** associated with each layer of each composite endpoint name for an interface. A quality of service range can be represented as a quality name and a quality value (typically boolean, numeric or textual).

As with protocol names, the interpretation of quality names must be consistent across a set of interacting nodes. Interception is required when binding involves different quality of service name contexts.

When encoding an endpoint it is often possible to compact the information content. For example if all application protocols have a common transport and network layer this need not be repeated in each alternate endpoint description in the reference.

### 6.3 Sub-addressing

Network architectures differ in the granularity of endpoints they support. The minimum requirement is to identify the ORB instance to which a message is being sent (or a particular network interface for a node with multiple connections to the network). Further levels of sub-addressing may be defined to identify interfaces relative to that ORB instance.

Where the network provides sufficient granularity to do so, each interface can be equated with a network endpoint and network addressing be relied upon to deliver messages completely.

If the network addressing scheme is too limited, it can be extended by using the local names within the ORB as sub-addresses, and adding an additional layer to the composite endpoint names to hold the local name as a sub-address. Examples of schemes for generating suitable sub-addresses include node-wide mailbox names and a process name/socket number within process pair (assuming ORB instances are equated to processes).

Network endpoint name spaces are often limited (because of a desire for compact encoding). This opens the possibility that when an interface is destroyed, its endpoint will be recycled and allocated to some other interface. Thus endpoint names are potentially ambiguous, and this impacts the design of the relocator database.

### 6.4 Relocators

Object migration and object passivation can lead to an interface being disassociated from an endpoint. In this situation the client will seek the help of a **relocator** to rebuild an end-to-end binding to the interface.

The client detects that its interface reference has become out-of-date either as communications timeout, or as an explicit end-to-end binding failure signalled by the addressed ORB.

The purpose of a relocator is to give a client a revised interface reference for one which is no longer valid.

*When an ORB provides location transparency, the relocation process is entirely internal to the ORB.*
There are many possible relocation policies, using directories, searching or forwarding. The policies differ in their cost, scaling and resilience characteristics.

Forwarding in particular has several problems and is the least attractive solution:

- it requires the ORB supporting the old location to remain in service for as long as the target interface persists
- long chains of redirection can build up which are difficult to prune and precarious to maintain.

Forwarding should be used only as an optimization to save the latency of an interaction with the relocator.

Relocators need only retain information about interfaces that have moved so that:

(i) the amount that has to be stored in a relocator's database is reduced
(ii) relocators need not be provided in systems where objects are immobile
(iii) the latency cost in accessing a relocator is only incurred when binding to objects that have moved.

The disadvantages of this approach are that:

(i) it may take some time to detect that a connection has been invalidated by an object moving
(ii) interface references have to include both interface and endpoint information.

The speed of detection can be increased if the network provides a negative acknowledgement of messages sent to disconnected endpoints.

In some networks there may already be a network name service which performs identifier to endpoint address translation. This name service is functioning as a relocator.

The principal operation of a relocator is \texttt{registerLocation} which takes an old and a new interface reference as argument.

\textit{Since there is no single ideal relocation policy it must be possible for different interfaces to have different relocators.}

An interface reference must in consequence contain addressing information about the relocator for the interface in addition to the addressing information for the interface itself. The addressing information for the relocator has exactly the same structure as that of the interface. A relocator is a service like any other. A relocator may itself be mobile and have a relocator. This requires that an interface reference be a recursive data structure starting with information about the address of the interface at the time the reference was made and terminating with the address of an immobile or ubiquitous relocator.

If a client receives a binding error when interacting with a server it contacts the relocator with the failing interface reference as argument. The relocator
ORB Interoperability

uses the whole interface identifier as a key to determine the current addressing information for the interface.

6.5 Relocator database

If the interface identifier contains an explicit key, the relocator database can be as simple as a lookup table using the key as index.

If interface references are disambiguated by including the timestamp at which the interface was associated with the endpoint(s), the relocator database must record all interface references that have been associated with the interface, since clients can retain references indefinitely.

When an object migrates between a small number of locations, the references can be safely re-used when the reference returns to a former site, reducing the burden on the relocator.

When an object migrates widely, the relocator can age references and move old ones to archival storage, retaining only the most recent in active memory.

The relocator can cooperate with interface reference garbage collection, so that database entries corresponding to interface references that are no longer extant are purged.

6.6 Integrity checking

Interface addressing depends upon many mechanisms functioning correctly: network management, communication, relocation and interception. The potential for failure is high. The overall integrity of location transparent addressing can only be achieved by end-to-end checks between client and server. A server should include in each interface reference it generates a nonce (i.e. a large random number) and require that all messages to the server quote the nonce. If a message is received containing a different nonce an integrity failure should be reported.

Nonce checking is not absolute: the same nonce may be allocated to different interfaces and therefore any error in the addressing of these interfaces may not be detected. However any systematic failure of network management or transparency provision will lead to a significant number of failure reports. Thus nonces contribute towards checking of the location independent addressing scheme as a whole, not the integrity of individual bindings. Because of this, less care is required about the seeding and size of random numbers used as nonces than if random numbers are used as interface identifiers.

There is redundancy between the interface identifier and the nonce components of an interface reference. In a practical encoding they may be merged into a single composite field. The key trade-off to consider when this is done is:

- the need for an unambiguous key so that a relocator can identify the interface in question
• the need to ensure addressing is not compromised by failures in network administration and distribution transparency mechanisms.

6.7 Replicated locations

A server may be transparently replicated (as described in the Model for Interface Groups [AR.002].) Transparent replication requires that an interface of a replicated server has a single interface reference.

If the network supports multi-cast addressing it is possible to assign a single, multi-cast endpoint. Otherwise replication must depend upon point-to-point interactions. In an environment where some parts of the network support multi-cast and others do not it is desirable to multi-cast where possible to reduce network traffic.

Replication therefore requires that a replicated server be described by a replica endpoint set (i.e. the member record of section 4.3) consisting of one or more composite endpoints. In the simple multi-cast case the set will be a singleton, in the point-to-point case the set will have as many elements as the group has members.

Since the members of the group are replicas it is appropriate that the nonce, relocator database key and relocator components of an interface identifier are the same for each member.

If some members are accessible via a multi-cast endpoint it is important to note they must have identical protocols, since they will each receive the same message.

If a client invokes an interface group it may receive answers from several members which have to be collated. The interface reference must contain a collation field which gives the information needed to perform collation. The exact content of the collation field will depend upon the replication protocol being used: it may be an indication of the size of the group, or the minimum number of messages required for successful collation.

The membership of a group can be dynamic. It is therefore necessary that a group can detect if a client is using an out-of-date membership view. An interface reference contains an incarnation count. A group increments this incarnation every time its membership changes. Messages sent to and from a group include this count. If the recipient retains an older count, it must rebind to the group.

The collation and incarnation fields comprise the group data of section 4.3.

---

1. This rebinding must be carefully synchronized: it is described in AR.002. If a group has the property that some of the members of the first incarnation of the group will remain in the group forever, the rebinding process can be internal to the group protocol. Otherwise the group must have a relocator. Where groups are being used for reliability, it is necessary that the relocator also be reliable.
6.8 Interface reference contents

The requirements upon interface references presented in the preceding sections are captured in the following schema for interface references:

- a key or timestamp to disambiguate interface to endpoint bindings
- a nonce for integrity checking
- an incarnation number to validate group membership if the interface is part of a group
- a collation field to guide the collation process if the interface is part of a group
- a location transparent address for the interface's relocator if the interface is mobile
- a set of endpoint addresses (up to one for each member if the interface is part of a group).

Each replica can have several alternative composite endpoint addresses. Each composite endpoint description is a sequence of protocol name, network endpoint address and quality of service range triples.

In some implementations the composite endpoint will include a so called unique identifier which can function as a relocator database key.

In ANSA IDL this schema could be encoded as:

```idl
InterfaceRef :TYPE = SEQUENCE OF InterfaceRecord;
InterfaceRecord :TYPE = RECORD
  nonce :CARDINAL
  key :CARDINAL
  incarnation :CARDINAL -- increments on membership change
  cardinality :CARDINAL -- collation field
  replicas: SEQUENCE OF Alternates -- replica endpoints
);
Alternate :TYPE = SEQUENCE OF Location; -- alternate endpoints for a
  -- group member
Location :TYPE = RECORD [ -- an alternate endpoint
  layers :ProtocolLayers, -- protocol layer names
  accesspoint: AddressLayers, -- endpoint addresses for each layer
  qos: QoSLayers, -- QoS for each layer
  operations :SEQUENCE OF String -- which operations can be
    -- supported by the protocol
 ];
ProtocolLayers :TYPE = OctetSequence; -- binary codes protocol
  -- "numbers"
AddressLayers :TYPE = SEQUENCE OF Address;
Address :TYPE = OctetSequence; -- an endpoint address
QoSLayers :TYPE = SEQUENCE OF QoSOffers;
QoSOffers :TYPE = SEQUENCE OF String; -- QoS parameter name /
  -- value pairs
```
6.9 Dynamic endpoints

In the foregoing an endpoint was associated with every object accessible at an ORB.

Sometimes this is too limiting a constraint: for example if it desired to interact with a database of many millions of accessible objects the resource demands of retaining an endpoint for each interface may be untenable. In this case the endpoint address space must function as a cache of currently active interfaces.

The ORB will not be able access an interface directly. Instead it must use the endpoint in a request message as a key to locate the interface on disc, and bring it into the active address space. This extra step is not visible to the invoker (except perhaps as additional latency).

6.10 Activation

In addition to objects which migrate, the relocator has to deal with transparent reactivation of objects. Activation is with reference to a passive version of the object (typically on some kind of storage service such a file system).

The object may have been passivity to save resources (i.e. liveliness or resource transparency), or it may be a checkpoint of an object that is being recovered after failure of its active copy (failure transparency).

In both cases the relocator must be told of the location of the passive object holding the interface being reactivated and on which ORB the object should be reactivated.

The act of storing a passive object generates an interface reference to a service consisting of operations to read and delete the passive object from the storage service.

Different objects may have different activation policies, including:

- reactivation in an already existing capsule
- activation in a new capsule.

Therefore in addition to the registerMigration operation a relocator provides a registerPassivation operation, which has as arguments

- the interface reference of the passive object
- the interface reference of an activation service.

An activation service provides a single operation activate, which accepts a passive object as its argument and instantiates the object on its host ORB. The activation service determines the revised interface references for all the current interfaces of the newly activated object, so that they can be notified to the relocator.

(In ANSA, the simplest activation service is a capsule manager (see the Storage Model AR.006). More complex activation services may attempt to load balance objects across several processors, for example in a processor farm).
When an interface is deleted, knowledge of it can be removed from its relocator, using a delete operation. This operation is typically called by the local object management regime within the ORB supporting the interface, or by a distributed garbage collector that has determined there are no remaining references to the interface.

In summary the operations supported by a relocator are:

- `registerMigration (oldInterfaceReference, newInterfaceReference) -> () -> Fail (Why :String)`
- `registerPassivation (oldInterfaceReference, passiveCluster, activator) -> () -> Fail (Why: String)`
- `delete (interfaceReference) -> () -> Fail (Why: String)`
- `createKey () --> (New :Key)`

(The last operation need not be provided if the relocator does not use keys to disambiguate references).

### 6.11 Relocator dependability

A relocator is potentially a single point of failure and a critical component in the successful operation of a system. It can be made reliable in several ways:

- (using failure transparency) frequent checkpoints can be made of the relocator database and an alternate relocator be specified which takes over if the primary relocator fails. Care must be taken to ensure that the checkpoint storage service itself is sufficiently reliable (e.g. it is replicated)

- (using replication transparency) the relocator can be replicated at several sites (with independent failure modes). The remaining replicas can provide service if one of them fails. The resilience of the relocator can be increased by adding further replicas. It can be expected that updates to the relocator are less common than reads, therefore replication should not introduce excessive latency.

- in a network supporting multi-cast the relocator can be replicated at every node, and the replication database partitioned such that each node retains information about the interfaces located at that node. A client relocating an interface will multi-cast to every other node and if there is an active node hosting the interface it will reply with its location.

The last scheme, whilst simple, as the disadvantage that it only works for multi-cast networks, and that a relocation request consumes processing resources at every node in the network.

Most current networks provide name serving capabilities which fulfill much of the functionality of a relocator. In these networks, each node can provide a local proxy which maps from the relocator interface onto the functions of the network name service.
6.12 Optimized implementation of interface references

If encoded directly the information content of an interface reference will be large, particularly in a multi-protocol technology domain. However, all the information they contain is useful and cannot be discarded without losing functionality.

There are various ways in which interface references can effectively be abbreviated, at some cost in performance.

- Interface references could be stored in name servers and short names marshalled in requests and replies. When a connection is required (or has to be remade) the client will have to consult the name server to acquire the full reference. This increases the latency of connection. The name server could also provide the relocation functions so that clients are always given up-to-date references which may help reduce latency when accessing interfaces that have migrated.

- The first time a reference is transferred between ORBs, the full reference is copied, and both ORBs count how many references they have exchanged. If the reference is transferred again, its index is transferred rather than the whole reference. This effectively amounts to the ORBs building up a shared short name space for references. It imposes a storage burden on both of them.

- Often different interfaces on the same ORB will have identical fields for much of the addressing information, this need only be transferred once between ORBs and thereafter only the part of reference that varies need be moved.
References

[ISO 10746-3]

[AR.001]

[AR.002]

[AR.003]

[AR.005]

[TR.28]

[TR.44]