Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Binding mechanisms in distributed systems

Rob van der Linden

Abstract

Several distributed systems infrastructures are becoming available, amongst them are OSF's DCE, various OMG Corba implementations (e.g. Iona's Orbix), and ANSAware. Allthough conceptually similar, these systems all employ different binding mechanisms. The cost of developing applications which are supported by more than one of these platforms remains therefore traditionally high.

This paper seeks to examine the differences in more details, with the aim of paving the way to interception technology which can be employed to bridge the boundaries between different distributed systems platforms.

The current state of the paper is that of an unfinished draft.

APM.1281.00.01 **Draft** 4th August 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

1 Binding across boundaries

1.1 The problem

Communications between software components is increasingly hidden from programmers by a procedure call, or method invocation abstraction. The relation between the name of an interface (in which operations can be requested) and the interface instance (with the methods which execute the operations) is often referred to as a *binding*. When an operation request occurs in a different system from method execution, there is a need for external communication between the components.

Before two parties can communicate, it is necessary for there to exist a path along which messages can flow. For remote interactions, this requires common sets of protocols and data encoding. The trading process helps distribute information about this to binding processes which then try and achieve compatibility before interaction takes place.

In different systems, these processes have been implemented in different ways. The information which is exchanged is not always the same either because different assumptions are made about what needs to be conveyed.

Interacting components increasingly find themselves in different systems. The demand for cross boundary trading and binding is therefore increasing.

1.2 This document

The aim of this document is to highlight different ways in which binding is implemented and to discuss ways in which different implementations can be made to interwork.

This document will be used as an example from which abstractions may be derived. It should also be used as a test-case for any generic interception mechanism we may propose.

1.3 Scope

Different distributed systems infrastructures have been examined [ANSA], [DCE], [COMMANDOS][1], [SOS][2]. Others are to be added: [Orbix].

1.4 Audience

This document is aimed at those working on interception and trading in ANSA Phase 3.

1. We choose the TCD Amadeus implementation of COMANDOS.

2. Developed at INRIA France.

2 ANSA binding

ANSA has three distinguished services relevant to the provision of a communications path between two parties that require to interact with one another. These services are the trading service, the binding service and the location service. The extent to which these services are visible to application programs (and their programmers) depends on whether binding is implicit or explicit. Traditionally, all binding in ANSA has been implicit. Explicit binding has been added more recently so as to give programmers the ability to control the quality of service of communications paths.

2.1 Implicit binding

Of the trading, binding and location services, only the trading service is visible to application programs (and their programmers). The binding and location services are provided as part of the distributed system infrastructure that supports distributed applications. None of the services are required of a host operating system. All a host operating system is expected to do is to deliver messages between parts of an application program as quickly and as often as is possible.

It is important to understand the difference between trading, binding, and location services on the one hand and name servers, databases (of what is where), and communications protocols on the other. Name servers and databases may be used in the provision of trading and binding information, but do not need to be. Communications protocols remain invisible to application programs.

- A *trader* matches a request from a service consumer with an offer of service provision made by a service provider. Such requests are not based on the location or address of the service, but on its type and property.
- The *binder* provides commonality of infrastructure and communications protocols for service consumer and provider.
- A *relocator* finds service providers that have moved, been recovered from failure, or that have been passivated (i.e. swapped from memory).

A service is provided at an interface, and the ability to use a service at an interface may be passed from one entity to another. From a computational perspective this is done by passing the interface as an argument or result. In the engineering model this is implemented by passing an *interface reference* of the service.

The mapping of the computational view of this process to engineering data structures generated by the compiler and used by the infrastructure to generate and pass around interface references is described in [DPL Engineering].

The receiver of an interface reference (the potential client) uses the information contained in the interface reference to set up its infrastructure

and communications protocols in a manner which conforms to the infrastructure and communications protocols of the service to which the interface reference belongs (the server). This is done by the local binder, invisible to the application and independent of what happens at the server end: neither client nor server applications have any knowledge whatsoever of their own or one another's address or protocol(s). A client may thus (transparently) use a service over a number of different protocols and networks.

The relative location of client and server is not relevant from the application point of view either. Services may be distributed in space and time. They may be

- static (in the same place for a long time)
- dynamic (created on demand),
- migrated (for load balancing or fault avoidance)
- recovered (from a previous checkpoint)
- passivated (saving over-used resources)
- replicated (for increased reliability)

All the attendant location problems are invisible to a client application, who should be able to interwork with a service as if it was always available in the same place. The desired effect is achieved by the use of the ANSA Interface Reference.

An interface reference consists of four components:

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

Each member record consists of a sequence of address records, one for each set of protocols that can be used to access a service. Each address record contains a tuple consisting of

- a protocol identifier
- a protocol address

Protocol identifiers and protocol addresses are names which have meaning within the specific communications network domain (context relative).

The ANSAware interface reference is defined in both IDL and C data structures in Chapter 7.

The relocator interface reference can be used to access a location service. It has all the components of an interface reference (potentially including a reference to yet another relocator interface).

The information contained in an interface reference is sufficient to set up a connection through one or a number of networks and protocol stacks to the service with which the interface reference is associated. When two services are located in two different networks, that is, within a different addressing context, then proxies are introduced at the gateway between the networks. These proxies ensure that end-to-end bindings can be effected, despite the different naming contexts for information elements in the interface reference.

Thus the strict requirement for context relative naming remains satisfied at all times.

If in the cause of interaction between a client and a server no responses are forthcoming, it is possible to rebind using a different address record. If the service has migrated, then none of the address records will yield a successful interaction. In that case a new interface reference may be obtained from one of the Relocators, whose references are included in the interface reference. All (re-)binding and access to Relocators is performed by the client's infrastructure, invisible from the application.

The nonce is used as an end to end check. It allows a server to check that a request for its service originates from a client that does in fact hold an interface reference that was generated by that server. When the server infrastructure receives a request it checks that the nonce is one that it issued. The response from a server to a client can be similarly checked.

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 the same binding, then the binding is broken. Thus the nonce (which is a 64 bit random number) is only used in the context of an interaction on a connection determined by the address records. The chances of erroneous connection are so reduced to acceptable levels, without having to resort to globally unique identifiers.

The structure and use of the interface reference allows applications using multiple protocols and in different networks to interwork. An interface reference is automatically assigned by the infrastructure when an interface is created and used only by the binder, invisible from applications. There is no need for a global database or unique identifiers anywhere. This is a substantial advantage when interconnecting systems which have been separately designed and implemented.

Interface references are visible only in the infrastructure and remain invisible to application programmers. Application programmers may assign any name to an interface, and in reality, such an application level name is associated with the interface reference in the infrastructure. This association is purely local to the context of the entity that holds the interface reference. As interface references may be freely copied, it is possible for two applications to hold a reference to the same interface, and for each application to have assigned a different name to the interface. This follows the rule for strict context relative naming that is required of a naming system suitable for a large distributed system with good scaling characteristics.

3 Binding in Orbix

Note: Editorial paragraph

Text Paragraph

4 DCE binding[1]

Binding in DCE is done on the basis of a binding handle. The binding handle is generally obtained from a name server upon submitting the UUID for a service. UUIDs may be freely passed throughout the system.

In DCE, distributed applications are written using the remote procedure call (RPC) model, and therefore are called RPC applications. DCE RPC offers the following basic components to build RPC applications:

1. The DCE Interface Definition Language (IDL) for declaring RPC interface definitions.
2. The IDL Compiler, which generates client and server stubs from RPC interface definitions.
3. The DCE UUID Generator, which creates UUIDs with a single command.
4. The RPC Runtime, which provides RPC applications with a variety of services such as communications for making and managing remote procedure calls.

An RPC application consists of an RPC server and one or more RPC clients that generally reside on separate systems and communicate over a network. For each RPC interface that the server offers, it must provide at least one set of remote procedures that implements the interface; each client contains a main, calling program that calls into the RPC interface.

A remote procedure call requires a relationship -or binding- to be established between the client and the server. The binding allows the call to find the right server and the right procedure. For a binding to exist:

- The client must find a server that offers the RPC interface of the called procedure.
- In an object-oriented application, the server must also offer any object that the client specifies in a call.
- The client and server systems must use a common set of communication protocols.

To establish a binding, a client needs information about a suitable server that meets these prerequisites (that is, binding information). The specific components of binding information vary. Potentially, binding information can encompass the following components:

- An interface UUID (Universal Unique IDentifier) that identifies the RPC interface of a called remote procedure.
- An interface version, which identifies a specific generation of the interface.

1. We adopt DCE terminology in this section. Some of this section was created by BULL in the context of the Harness project. It is reproduced under the provisions of the ESPRIT Harness contract.

Version numbers allow multiple versions of an RPC interface to coexist. Strict rules govern valid changes to an interface and determine whether different versions of an interface are compatible.

Together, the interface UUID and version number uniquely identify a given instance of an RPC interface across systems and through time.

- An object UUID, which identifies a particular object upon which an object-oriented remote procedure operates.
- A protocol sequence, which identifies a specific combination of communication protocols.
- Network addressing information, which includes the network address and the transport endpoint of a server.

Clients reference binding information using a binding handle. A binding handle is a reference to binding information that defines one possible binding for a given server.

This binding information includes an object UUID, a string identifying an RPC protocol sequence, a network address, and an endpoint. The object UUID is the universal unique identifier of an object (e.g.: a resource such as a file) on which the remote procedure operates. (It can be uuid_nil if none object is specially concerned). The network address identifies a specific host by a string whose format is specific to the network protocol identified in the protocol sequence. An endpoint is a transport-layer address that is specific to the transport protocol identified in the protocol sequence. It gives the server process addressing information. (There exist dynamic and well-known endpoints. See DCE documentation for more details about this).

A string representation of binding handles can take the following format:

```
obj-uuid@rpc-protocol-seq:network-addr[endpoint]
```

For example:

b07122e2-83df-11c9-be29-08002b1110fa@ncacn_ip_tcp:16.20.15.25[2001]

Binding handles enable clients to recognize and find servers that offer a given RPC interface (and object). To establish a binding, a client must know the location of a server that has at least one compatible binding handle, which contains an RPC protocol sequence that correspond to an RPC protocol sequence that is available to the client.

A client can obtain a compatible binding handle from either:

- A directory service.

  Normally, servers place their binding information for each interface they offer into a Cell Directory Service (CDS) database (known as a name space) through a process called exporting. To call that interface, a client can opt to obtain binding information from the name space, through a process called importing, which begins with a designated CDS entry name. When the client stub is created, a developer can assign it responsibility for automatically obtaining binding information from a directory service, relieving the client application code from any involvement in binding management.

- A string representation of a binding handle.

  Alternatively, a client can receive binding information for a remote procedure in the form of a string that represents a binding handle. The

client must pass the string to the RPC runtime for conversion to a binding handle, which the client then use in the remote procedure call.

5 Other binding schemes

5.1 Binding in Amadeus

The Amadeus system accesses the name service with a name. The name service turns the name into a descriptor, which includes a stub. The stub is used to build the piece of infrastructure that will take care of the remote invocations when they occur. It includes marshalling code and has details of the protocols that should be used.

The name server is built in the UNIX file system, the name refers to the file that contains the stub.

Binding is done on the basis of the information in the stub. The building of the proxy is the result of the binding process.

5.2 Binding in SOS

Binding in SOS requires access to the low level binding functionality. This appears necessary to give the programmer control over distribution. In SOS, a client obtains information that allows the Bind primitive to be called.

The bind primitive calls a remote object with the GiveProxy operation. This will create a proxy in the remote end and returns to the local end with binding information that allows the binding to be established with the remote proxy. Invocations of the remote service will then take place through the remote proxy.

This scheme allows resource management policies to be implemented as services can be started only when a binding is established. Note however that the creation of a binding is a perogative of the client and does not guarantee that the server will in fact be called. Also, this scheme requires extra messages to the server infrastructure each time a reference to the service is passed and bound, thus unnecessarily creating traffic and causing resources at the server end to be consumed.

6 Building bridges

Note: This chapter is largely incomplete.

6.1 DCE and ANSAware united

When interconnecting an ANSA and a DCE based system, we are faced with the essential differences between the respective approaches to binding and addressing. There are two ways in which interworking can be achieved: by the creation of proxies in either system, and by the integration of UUIDs in the interface reference.

6.1.1 Integrating UUIDs and Interface references

To integrate the UUID and interface references, the UUID is simply added as one of the addresses in the address data structure. The interface reference can include the standard ANSA binding information (socket id, protocol information), the UUID, or the DCE binding handle. UUIDs are converted to binding handles by access to the DCE name service.

Note: Decide on similarities between ANSA interface references and DCE binding handles: can interface refs be implemented above DCE?

6.1.2 Proxies

A proxy in this context is an entity in one (type of) system that acts as an agent for an entity in another (type of) system. The use of proxies to achieve interworking is considered inferior to a proper integration of Interface references and UUIDs/binding handles. Nevertheless, it is easier to implement and can be used as the basis for the quick prototyping of interworking.

Any ANSA service that is to be made visible in DCE needs to have its DCE proxy in the DCE environment. The proxy has a UUID and an associated binding handle can be obtained from the DCE name service. Invocations are DCE invocations on a local DCE interface. Any such invocations are passed on to the ANSA service, represented by an interface reference in the ANSA environment.

Any DCE service that needs to be made visible in the ANSA environment will require an ANSA proxy to which a binding can be established using an interface reference. Invocations on the proxy will need to be forwarded to the DCE service bound through the binding handle, obtained from the name service with a UUID as argument.

The static case appears to work well. However, when interface references and UUIDs are being passed around from one environment to another, there is a need to build new proxies all the time. This may lead to performance degradation. To decide when to clean up proxies that are no longer needed is

not straightforward. One solution is to implement a single object that can act as proxy for many objects (of different type).

The proxy mechanism supports most transparencies. The migration of an ANSA service in the ANSA system will remain invisible to the DCE client as the proxy does not migrate. The same is true for migrating DCE services: it would not be visible from the ANSA client. Conversely, when a proxy migrates, this will be visible to the client object, the server object remains unaware of this.

Migration across a boundary between ANSA and DCE involves many changes, due to the differences in both infrastructures. This kind of migration or portability is therefore unlikely to be practical.

6.2 ANSAware and SOS compared

ANSAware has a similar scheme to SOS for resource management. It operates by placing a proxy offer in the trading service. The proxy offer is an offer for a service that has not been instantiated. When a proxy offer is being imported, the trading service faults to the node manager, who sets the required service provider running and returns the true interface reference. This is the reference that the importing client sees in the result obtained from the trading service.

Although this scheme does not create as many new messages, this scheme suffers from a similar problem as the SOS scheme in that the action of importing a service is no guarantee for that service actually being used. The action of importing is however more deliberate than being passed a reference to a remote service in the result of some call.

7 ANSAware interface reference definition

7.1 IDL definition of the ANSAware Interface Reference

```
BaseTypes: INTERFACE =

BEGIN

 -- Identifier used for capsules by both the Factory service
 -- and the underlying Capsule library
 ansa_CapsuleId: TYPE = CARDINAL;

 -- Interface Reference structure

 -- Protocol Identifiers
 ansa_ProtocolLayers: TYPE = SEQUENCE OF OCTET;

 -- Protocol Addresses
 -- All octet sequences for addresses must be in
 -- big endian, network, byte order
 ansa_Address: TYPE = SEQUENCE OF OCTET;
 ansa_AddressLayers: TYPE = SEQUENCE OF ansa_Address;

 -- QoS offers
 ansa_QoSOffers: TYPE = SEQUENCE OF STRING;
 ansa_QoSLayers: TYPE = SEQUENCE OF ansa_QoSOffers;

 -- Operations to which this AddressRecord applies
 ansa_Operations: TYPE = SEQUENCE OF STRING;

 ansa_AddressRecord: TYPE = RECORD [
     -- Encode all fields in parrallel to conserve space when
     -- any of these fields are not required.
     -- layers     - list of named layers provides an index
     --                 into the other fields
     -- addresses  - address for each layer
     -- qos        - list of QoS offers for each layer
     -- operations - list of operations to which this
     --                 AddressRecord applies

     layers: ansa_ProtocolLayers,
     addresses: ansa_AddressLayers,
     qos: ansa_QoSLayers,
     operations: ansa_Operations
 ];

 ansa_GroupMember: TYPE = SEQUENCE OF ansa_AddressRecord;
 -- A record for each protocol set used by the server (group
 -- member). If any record contains a multicast network address
 -- then all must.
```

```
ansa_MemberList: TYPE = SEQUENCE OF ansa_GroupMember;
 -- Contains the set of addresses to be used for issueing
 -- requests to a group.

ansa_Nonce: TYPE = ARRAY 20 OF OCTET;
 -- The Nonce must be unique for a given interface at a given
 -- location over a long enough period of time to ensure
 -- that messages cannot be mis-delivered to that interface.
 -- Mis-delivery includes:
 -- i) a message for another interface being delivered to
 --  the current i/f
 -- ii) a message for a previous incarnation of the current i/f

ansa_InterfaceRecord: TYPE = RECORD [
 -- Need one AddressHint for each distinctly addressed sub-group.
 -- Because multicast addresses may reach multiple members then
 -- the length of addressList may be less than the cardinality.
 -- The cardinality is the number of members in the current
 -- incarnation of a group and therefore defines the number of
 -- expected replies.
 -- Whenever the cardinality changes the incarnation is
 -- incremented.

 -- singleton interfaces have incarnation= 0, cardinality 1
 -- empty groups have incarnation > 0, cardinality = 0
 -- single member groups have incarnation > 0, cardinality = 1

nonce: ansa_Nonce,
incarnation: CARDINAL,
cardinality: CARDINAL,
members: ansa_MemberList
];

 ansa_InterfaceRef: TYPE = SEQUENCE OF ansa_InterfaceRecord;
 -- The first record is for the application interface; subsequent
 -- records are for interfaces to increasingly robust locators.
 -- If invocation of the application interface using the first
 -- record fails to reach the server then the locators are invoked
 -- in order requesting a complete new InterfaceRef.

END.
```

7.2 C structure - ANSAware Interface Reference

```
/*
 * Generated by `stubc $Revision: 1.18 $'
 *   from `../.././include/idl/BTypes.idl'
 *   on `Fri Jan 7 10:14:49 1994'
 */

#ifndef _BaseTypes_types_
#define _BaseTypes_types_
#include "machine.h"

typedef ansa_Cardinal ansa_CapsuleId;

typedef struct ansa_ProtocolLayers {
     ansa_Cardinal length;
     ansa_Octet *data;
} ansa_ProtocolLayers;

typedef struct ansa_Address {
     ansa_Cardinal length;
     ansa_Octet *data;
} ansa_Address;

typedef struct ansa_AddressLayers {
     ansa_Cardinal length;
     ansa_Address *data;
} ansa_AddressLayers;

typedef struct ansa_QoSOffers {
     ansa_Cardinal length;
     ansa_String *data;
} ansa_QoSOffers;

typedef struct ansa_QoSLayers {
     ansa_Cardinal length;
     ansa_QoSOffers *data;
} ansa_QoSLayers;

typedef struct ansa_Operations {
     ansa_Cardinal length;
     ansa_String *data;
} ansa_Operations;

typedef struct ansa_AddressRecord {
     ansa_ProtocolLayers layers;
     ansa_AddressLayers addresses;
     ansa_QoSLayers qos;
     ansa_Operations operations;
} ansa_AddressRecord;

typedef struct ansa_GroupMember {
     ansa_Cardinal length;
     ansa_AddressRecord *data;
} ansa_GroupMember;

typedef struct ansa_MemberList {
ansa_Cardinal length;
ansa_GroupMember *data;
} ansa_MemberList;
```

```
typedef ansa_Octet ansa_Nonce[20];


typedef struct ansa_InterfaceRecord {
     ansa_Nonce nonce;
     ansa_Cardinal incarnation;
     ansa_Cardinal cardinality;
     ansa_MemberList members;
} ansa_InterfaceRecord;

typedef struct ansa_InterfaceRef {
     ansa_Cardinal length;
     ansa_InterfaceRecord *data;
} ansa_InterfaceRef;

#endif /* _BaseTypes_types_ */
```

References

[ANSA 91]

ANSA: A Systems Designer's Introduction to the Architecture, APM Ltd., Cambridge U.K., April 1991.