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

The ANSA Binding Model

Dave Otway

Abstract

The ANSA architecture implicitly binds clients to servers. This is simple to use and enables the use of a simple default resource allocation policy that optimises the use of resources, but it doesn't cater for the requirements of (multi-media) streams and predictable (time-critical) applications.

These applications require application program control over the timing and qualities of bindings.

The paper documents the implicit binding model plus its default resource allocation policy and develops an explicit binding model that:

provides precise control over the timing and duration of bindings,
generates domain specific Quality of Service specifications,
caters for multi-channel and multi-party bindings,
handles all binding management in applications,
preserves type safety,
and places no requirements on the communications technology.

The binding model is presented as facilities of the underlying mechanisms (in the engineering model) seen by the application programmer (via the computational model)

APM.1392.01	**Approved**	10th January 1995
	Architecture Report	

Distribution:
Supersedes:
Superseded by:

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

The ANSA Binding Model

The ANSA Binding Model

Dave Otway

APM.1392.01

10th January 1995

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 Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, 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	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

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

3	**1**	**Introduction**
3	1.1	Purpose
3	1.2	Audience
3	1.3	Relevance
4	1.4	Background
4	1.5	Scope
4	1.6	Notation
5	1.7	Previous definitions
7	**2**	**Requirements**
7	2.1	Implicit binding of operational interfaces
7	2.1.1	Resource allocation policies
8	2.1.2	Quality of service
8	2.1.3	Binding management
8	2.2	Explicit binding of operational and stream interfaces
9	2.2.1	Resource allocation policies
10	2.2.2	Quality of service
10	2.2.3	Binding management
13	**3**	**Architectural principles**
13	3.1	Objectives
13	3.1.1	Preserve the semantics of the ACM
13	3.1.2	Preserve type safety
13	3.1.3	Don’t invalidate the security architecture
13	3.1.4	Avoid placing requirements on the technology
14	3.1.5	Follow general architectural principles
14	3.2	Problems
14	3.2.1	Type safety
14	3.2.2	Security
14	3.2.3	Avoidance of technology requirements
15	3.3	Observations
15	3.3.1	All binding operations are local
15	3.3.2	A switch is not a binder
16	3.4	Choices
16	3.5	Summary of key principles
19	**4**	**Binding components**
19	4.1	Computational bindings
19	4.2	Transparency and indirection
20	4.3	Engineering bindings
21	4.3.1	Local engineering bindings
21	4.3.2	Remote engineering bindings
26	4.3.3	Technology addresses

29	**5**	**Implicit binding**
29	5.1	Computational view
29	5.2	Engineering view
29	5.2.1	Engineering objectives
30	5.2.2	Dormant bindings
31	5.2.3	Active bindings
31	5.2.4	Sessions
33	5.3	Technology view
33	5.4	Relocation
34	5.5	Garbage collection
35	**6**	**Explicit endpoint binders**
35	6.1	Explicit endpoints
35	6.2	Endpoint binders
36	6.3	Client endpoints
36	6.4	Client binders
37	6.5	Server endpoints
37	6.6	Server binders
38	6.7	Stream endpoints
38	6.8	Stream binders
39	6.9	Client control of implicit bindings
39	6.10	Binding failures
39	6.11	Customised binders
41	**7**	**Explicit binding management**
41	7.1	Server binding managers
42	7.2	Client and server binding managers
43	7.3	External binding managers
44	7.4	QoS specification
45	7.5	Monitoring and control
46	7.6	Multi-channel bindings
47	7.7	Multi-party bindings
47	7.8	Future work

1 Introduction

1.1 Purpose

The ANSA architecture implicitly binds clients to servers. This is simple to use and enables the use of a simple default resource allocation policy that optimises the use of resources, but it doesn't cater for the requirements of (multi-media) streams and predictable (time-critical) applications.

These applications require application program control over the timing and qualities of bindings.

The paper documents the implicit binding model plus its default resource allocation policy and develops an explicit binding model that:

- provides precise control over the timing and duration of bindings
- generates domain specific Quality of Service specifications
- caters for multi-channel and multi-party bindings
- handles all binding management in applications
- preserves type safety
- and places no requirements on the communications technology

The binding model is presented as facilities of the underlying mechanisms (in the engineering model) seen by the application programmer (via the computational model).

1.2 Audience

This document is aimed at a specialized technical audience. Readers should be familiar with the ANSA Computational Model (ACM) [APM.1001] or PART 3 of the ODP Reference Model [ODP-3].

The explicit binding model discussed in this document provides the framework for specifying, monitoring and controlling Quality of Service and configuration in large scale distributed systems.

1.3 Relevance

The explicit binding mechanisms are needed to cater for:

- time-critical applications
- multi-media applications

in large scale distributed systems.

Applications of the work in PHASE III are:

- input to the ODP Reference Model and TINA-C connection manager

- design of multi-media architectures and their mapping to C++, CORBA, etc.

1.4 Background

This document proposes some extensions to the ANSA programming model [APM.1014 & APM.1015], and run-time engineering [RC.273] to provide explicit binding for both operational interfaces [APM.1001] and stream interfaces [APM.1108].

The stream interface extensions to the ACM described in “Streams and Signals” [APM.1108] have no implicit binding semantics and rely exclusively on the explicit bindings mechanisms developed in this document.

The explicit binding model also provides a type safe framework for dealing with the domain specific Qualities of Service described in “A Model of Real-Time QoS” [APM.1151].

The ANSA explicit binding model has been developed in conjunction with the binding model of the ODP Reference Model [ODP-3] and the formalisation of the ANSA/ODP Computational Model being done by ENST & CNET [NAJM & STEFANI].

1.5 Scope

Chapter 2 defines the requirements for implicit and explicit bindings along with the resource allocation policies, qualities of service and management policies that can be associated with them. Chapter 3 discusses the architectural principles that guided the design of the binding model.

Before delving into explicit binding, a full description is given of the semantics of implicit binding and of its default engineering. Chapter 4 gives a static view of the components of a binding in both the computational and engineering viewpoints, while chapter 5 describes how and when an implicit binding is established.

Chapters 6 and 7 define the programming language extensions for generating explicit binding endpoints for operational and stream interfaces respectively.

Chapter 8 discusses the many ways that these endpoints can be used to manage the establishment of explicit bindings. It then goes on to describe how to extend the basic mechanisms to provide domain specific QoS specification, monitoring and control, multi-channel and multi-party bindings while preserving type safety.

Although the explicit binding model provides the basic framework for providing domain specific QoS, it does not consider what these domains should be, what their semantics should be or what parameters they should have.

1.6 Notation

The examples are coded in DPL [APM.1014 & APM.1015] which is a language tailored to the semantics of the ACM. This enables the semantics of distributed computations to be expressed succinctly without the clutter of non distribution features found in most programming languages.

The explicit binding functionality described in this document can be added to other programming languages in a variety of ways, such as:

- a set of templates and coding rules
- an Application Programming Interface (API) implemented by a subroutine library
- a class library using inheritance mechanisms
- a preprocessor
- language extensions
- a combination of any of the above

Each implementor is free to choose how to map the explicit binding functionality onto a particular programming language independently of other implementors.

The new DPL constructs are defined in Backus Naur Form (BNF) with the extension that {} enclose zero or more occurrences of part of a production rule [APM.1015].

1.7 Previous definitions

This section brings together relevant binding related definitions from previous documents and indicates how they relate to this document.

Names and bindings are defined in the naming model [APM.1003] as:

a *name* is a linguistic entity, that singles out a particular entity from among a group of entities

a *binding* is the association of a name with a particular entity

an *invocation name* can be used to invoke some reaction from the entity with which it is associated

This document is concerned with the binding of invocation names.

Bindings are created in the ANSA Computational Model (ACM) [APM.1001] by:

a *binder expression* associating an invocation name with an interface instance

For example, the invocation name `x` is bound to an interface by:

```
x = interface ( do() ->() [ something ] )
```

and the operation `do` in interface `x` can then be invoked by:

```
x.do()
```

The ACM requires that an invocation name can be used to invoke operations on its associated interface instance anywhere in its scope. This document distinguishes between creating a binding (making the association) and establishing a binding (engineering the allocation of all the resources needed to make an invocation).

In the engineering model[1]:

an *(engineering) binding* is a local memory address pointing to the local invocation engineering for a particular interface instance (e.g. a local proxy for a remote object)

an *interface reference* contains all the engineering information necessary to establish a binding to a remote interface instance (e.g. network addresses or UUIDs)

To avoid confusion with the terminology developed in this document, these terms are renamed *invocation reference* [§4.3] and *remote interface reference* [§4.3.2].

1. In the absence of a full engineering model description, [RC.273] is the best reference to engineering bindings. The terminology is outdated and both the data structures and code templates are C specific, but the design and principles are still valid.

2 Requirements

This chapter discusses the requirements for binding models, resource allocation policies, quality of service and binding management in open distributed systems.

2.1 Implicit binding of operational interfaces

The original ANSA Computational Model (ACM) [APM.1001] only contains operational interfaces and the bindings to them are regarded as fully established as soon as they are created. This model has been adopted by most distributed computing architectures (CORBA, DCE, etc.).

A fully established binding is one on which operations can be invoked. The ACM has no notion of partially or non established bindings and makes no distinction between bindings to local and remote interfaces. Because the application programs play no part in establishing bindings, the bindings are regarded as being implicitly established.

2.1.1 Resource allocation policies

2.1.1.1 Point of establishment

Establishing a local binding uses no more (memory) resources than recording its creation, so there is no point in not establishing them when they are created. There is also no point in disestablishing them before they are garbage collected. This resource allocation policy is known as aggressive allocation and lazy release.

On the other hand, a remote binding can tie up significant (memory and communication) resources, so there are savings to be made if it is never used or while it is not being used. The need to conform to the (ACM) view that bindings always work, restricts the points at which the engineering can intervene to establish (or disestablish) bindings to:

1. when the binding is created (or garbage collected)
2. the first time an operation is invoked (or the last time one is terminated)
3. the first operation in a burst is invoked (or the last one in a burst is terminated)
4. every time an operation is invoked (or terminated)

Option 1 is the same as for local bindings and clearly conforms to the ACM, but offers no resource savings. Option 4 is a bit too aggressive in that the savings made are unlikely to be worth the extra time and delays of constantly establishing and disestablishing the binding.

Option 2 offers no advantages over option 3 and reliably detecting the last operation can only be done with application knowledge or hindsight. Therefore, option 3 offers the best compromise for general purpose use.

A binding is established on first use. A timeout is used to detect a dormant binding and trigger its disestablishment. The binding is re-established next time an operation is invoked on it. If a binding is never used, it is never established. This policy can be regarded as lazy allocation and moderately aggressive release.

2.1.1.2 Sharing

A sensible general purpose resource allocation policy is to optimise the use of resources. Therefore, all bindings between the same pair of capsules should be multiplexed on one communication channel.

2.1.2 Quality of service

The standard Remote Procedure Call (RPC) guarantees described below are required.

2.1.2.1 Execution guarantees

If the client receives an application termination then it is guaranteed that the operation has been executed *exactly once*. If the client receives an engineering termination then it is guaranteed that the operation has been executed *at most once*. The exactly once guarantee conforms to the expectations of the ACM, while the at most once guarantee caters for the problems of network and partial failures.

2.1.2.2 Performance optimisations

If possible, the protocols should be tuned to optimise for low latency on short messages and high throughput on long ones.

2.1.3 Binding management

For implicit bindings, neither the client or server applications are involved in managing the bindings.

But, even with implicit bindings on operational interfaces, the client could be given limited control over some aspects of the binding without requiring cooperation from the server; provided that it was aware of the server's binding policies and did not conflict with them.

Clients of implicitly bound servers could enjoy the benefits of increased predictability if they could exercise control over the timing of binding establishment.

Given a suitable monitoring and control interface to a binding, a client could use application knowledge to optimise the resource allocation policies described in [§2.1.1].

2.2 Explicit binding of operational and stream interfaces

Implicit binding successfully hides all those fiddly details concerning TCP, DCE, OSI, Berkeley sockets, binding handles, UUIDs, IP addresses, port numbers, TSAPs, connections, datagrams, ATM adaption layers, etc. from the application programmers. Although it is very easy to use and is adequate for the majority of cases, it is inadequate for two specialised but very important application areas, namely:

- predictable applications
- multi-media applications
- cooperative peer-to-peer

These terms are used very generally: the term predictable is intended to cover all applications (especially time-critical ones) that provide non-functional guarantees and the term multi-media is intended to cover all applications that use the stream extensions to the ACM as defined in [APM.1108].

Fortunately, these application areas have much in common, in that they require resources to be allocated and scheduled to deliver application defined guarantees. In general, guarantees are required for processing, memory and communications resources; but this document primarily considers the issue of communication resources.

The required guarantees could be installed by default, specified declaratively or the relevant resources can be controlled imperatively by operation invocations or extra arguments. This document takes the default approach for implicit bindings and the imperative approach for explicit bindings, but tries not to preclude a more declarative approach in the future. The imperative approach is taken because the imperative mechanisms will need to be in place before a more declarative set of tools can drive them.

Explicit binding of operational interfaces is needed to enable the application programs to exercise control over communication resources, while for stream interfaces there are no generally applicable resource allocation or quality of service policies so the application must take control.

2.2.1 Resource allocation policies

2.2.1.1 Point of establishment

An application that demands predictability needs to calculate its resource requirements and make its resource reservations in advance of its dependence on them. It may also need to avoid jitter caused by resource allocation delays. These require application control over the timing and duration of bindings.

A distributed predictable application also needs to know that all its components are present and correctly configured before it makes any commitment to deliver a specific level of service. This also requires application control over the timing and duration of bindings.

2.2.1.2 Separation

A predictable application needs to protect the resources it depends on from competing demands by requesting strict resource separation. For example it may wish to preclude transport channel multiplexing. But this requirement can be considered to be part of the more general QoS requirement [§2.2.2].

2.2.1.3 Sharing

Strict resource separation wastes resources, so many applications with less stringent requirements may be forced into resource sharing for economic reasons. This requires a policy for resolving contention. Again, this can be considered to be part of the more general QoS requirement.

2.2.2 Quality of service

If the correct functioning of an application depends on the behaviour of the communication mechanisms then the application needs to be able to specify what type and level of service it requires; and to be assured that this will be provided. This requires a mechanism for communicating QoS requirements to binders and resource managers.

The QoS parameters are specific to a QoS resource domain [APM.1151] and will therefore differ in their types and numbers between different bindings. This precludes a generic QoS specification and requires a very flexible QoS mechanism.

2.2.3 Binding management

2.2.3.1 Explicit endpoints

With implicit bindings, the client receives a purely local binding which is the invocation reference [§4.3]. The client application has no direct access to a binding endpoint for the server and the server application has no access to a binding endpoint for the client. So, for instance, a server cannot identify a specific client.

A binding endpoint is defined as the argument given to a binder that enables it to construct a binding to the entity represent by the endpoint. In engineering terms, a remote interface reference is a server's binding endpoint; while in technology terms an endpoint is usually a form of address such as an IP address and a UDP port number, but feeding remote interface references or technology addresses to an application program would destroy most distribution transparencies. A computationally visible binding endpoint can be constructed by hiding an engineering endpoint behind an interface which contains a bind operation.

Such endpoint binder interfaces are required for explicit binding so that applications can identify a specific binding between them.

2.2.3.2 Multi-channel bindings

It is sometimes necessary to batch together a number of bindings because they require a coordinated Quality of Service such as mutual synchronisation.

It would also be convenient to pass around a single endpoint reference for a set of related bindings so as to minimise their management.

2.2.3.3 Multi-party bindings

Implicit bindings only have two[1] endpoints. But there are interesting communication scenarios that involve multiple parties, such a distance learning and video conferencing.

These multi-party applications could be constructed at the application level out of two party bindings, but these are difficult to program and will not provide the optimum solution [VAN JACOBSON]. A better job could be done in the communications engineering and some optimisations require the active cooperation of the network.

1. Bindings to and from groups [APM.1002] may be regarded as having multiple endpoints. But the replicated group members are really an engineering mechanism to provide failure transparency and are computationally identical.

Although it would be desirable if the multi-party binding mechanisms could also cope with groups [APM.1002], this is not regarded as a constraint.

2.2.3.4 *Coordination and negotiation*

Once the default resource allocation policy has been abandoned in favour of increased flexibility, the parties involved will need to coordinate their resource policies and possibly negotiate the most optimal configuration.

2.2.3.5 *Monitoring and control*

Applications need the ability to monitor and control bindings in order to:

- perform end-to-end application specific optimisations
- employ application specific knowledge to recover from failures
- degrade gracefully during resource shortages
- exploit resource gluts

This requires that applications can acquire control interfaces to particular bindings and provide their own monitoring interfaces to bindings.

3 Architectural principles

This chapter discusses the architectural principles that guided the extension of the binding model to include explicit bindings.

3.1 Objectives

The specific requirements for explicit binding were given in the previous chapter, but these have to be delivered in the much wider context of an existing architecture for distributed computing. Therefore the explicit binding model has to meet the following more general objectives

3.1.1 Preserve the semantics of the ACM

The ANSA/ODP computational model successfully enables the construction of a large class of distributed applications. It is essential that any additional features do not alter its semantics. Those existing features which might be considered vulnerable include:

- implicit binding
- fixed and variable assignment
- scope rules
- invocation semantics
- argument passing by sharing
- location and access transparency

3.1.2 Preserve type safety

Constructing large scale, heterogeneous, federated systems is inherently complex and error prone. One of the most common kind of errors in such systems are configuration errors. Type checking catches a large proportion of configuration errors at very little cost. Relaxing type checking in the area least well understood by most application programmers would be a major folly. Note that where a compiler can infer the type of something it doesn't necessarily have to declared by the programmer; what matters is that the type is known to the compiler.

3.1.3 Don't invalidate the security architecture

The ANSA security architecture [APM.1007] is dependent on the integrity of the encapsulation mechanisms. A binding model which weakened encapsulation would also compromise security.

3.1.4 Avoid placing requirements on the technology

A binding model that required specific features to be present in the communications technology that it could bind (such as protocols, management

interfaces or address formats), would be too restrictive for use in heterogeneous federated systems.

3.1.5 Follow general architectural principles

The binding model is an integral part of the architecture and should follow the general principles described in [APM.1000] if at all possible.

The general principles relevant to explicit binding are:

- assume separation (no common environment or implied context)
- encapsulate state
- access state via Abstract Data Types (operational interfaces)
- leave in the indirection
- be prepared for (partial) failures
- separate policy from mechanism (declare what is required rather than how it is provided)
- check early
- use tools to optimize
- allow federation (cater for decentralised control)
- design for large scale systems
- use context relative names (no universal IDs)

3.2 Problems

The main problems stem directly from three of the general objectives described above, namely type safety, security and avoidance of technology requirements.

3.2.1 Type safety

A generic explicit binding service which could be invoked from an application program to explicitly bind to any type of interface could not be type specific to the interfaces being bound without a run-time type check. The absence of a run-time type check would weaken the type safety at the very point where it is most useful and the presence of a check would slow down binding.

3.2.2 Security

A binding service that could establish bindings in other capsules would break those capsules' encapsulation and invalidate their security.

3.2.3 Avoidance of technology requirements

This objective makes it impossible to negotiate, monitor and control bindings via special facilities in the communications technology.

Note that this objective does not ban the exploitation of relevant facilities that exist, but it requires alternative mechanisms if they don't.

What it is really saying is that if you are waiting for a universal binding (or network management) protocol to emerge, don't hold your breath. Everybody

that has gone down this path has ended up being restricted by an inappropriate technology; and they are all different.

3.3 Observations

Before considering the possible design choices, there area couple of relevant observations that may make things clearer.

3.3.1 All binding operations are local

The statement that “all binding operations are local” may be counter intuitive when considering the requirement to bind together distributed objects, but if you think about it a little it soon seems obvious.

The encapsulation principle of the architecture bans one object from changing the state of another and if the objects are on physically separate nodes it is impossible for one object to change the state of another. Therefore, the process of establishing a binding between two physically separate parties always involves a locally applied state change at each end and some form of communication between the parties to coordinate the state changes.

The communication may be by any means, including human coordination of both ends, via a third party or on an existing binding between the two parties. Each party generates a binding reference to itself which is specific to the binding being established and exchanges this for the other party’s binding reference.

The state changes may be software (allocating a port number) or hardware (plugging a cable into a socket) and may be done at widely different times, but there is always one at each end.

Binding is characterised by matching pairs of operations performed by each party (after they have generated and exchanged binding references) of the form:

```
this_party.bind(other_party)
```

Both bind operations only involve local state changes; which is why a distributed binding (noun) can be established using only local binding (verb) operations.

Thus an (ANSA) binder is defined as a local engineering object which establishes a mapping between a co-located application object and a communications channel and its takes the coordinated actions of (at least) two such binders to establish a binding.

After a binding has been established, the binders are no longer in the communication path.

3.3.2 A switch is not a binder

A switch[1] (such as a telephone exchange) makes an internal connection between two of its ports (or line cards) to which the end parties have already been bound. Any party, including one of the two parties being connected, can instruct a switch to make a connection.

1. Strictly, the switch fabric as opposed to the switch logic.

In ANSA terms, setting up a call between two telephones bound to the same exchange involves no extra binding operations beyond those that were done when the telephones were installed. For long distance calls, the exchanges need to have bindings established between them for the duration of the call.

A switch is characterised by a single operation performed by any object (including one of the parties) of the form:

```
switch.connect(first_port,second_port)
```

where both parties are already bound to their respective switch ports.

After a switch has established a connection, it forms an integral part of the communications path.

The raw switch interface may not be presented to the end parties, but packaged up in an interface (line card) that contains the identity of the calling party and has access to a name server which can resolve the name of the called party into a port number (on some switch). This functionality is migrating out of the switches into supporting computers and may eventually end up in the end systems.

3.4 Choices

The type safety problem can be solved at compile-time by generating type specific binders [§6.2] on demand; or at least creating the computational illusion that this is being done.

The “all binding operations are local” observation simplifies security and technology problems. Each object can take its own decisions[1] and instruct its communication engineering to do the appropriate things to the communications technology.

This design has the added benefit of doing all the negotiation, coordination, monitoring and control of bindings as applications with the full range of distribution transparencies, functionality and application services available to them.

3.5 Summary of key principles

- preserve the semantics of the ACM
- preserve type safety
- don’t invalidate the security architecture
- avoid placing requirements on the technology
- follow general architectural principles
- all binding operations are local
- a binder is not a switch
- generate type specific binders
- all binding management is done by applications

1. Or choose to delegate them to another object.

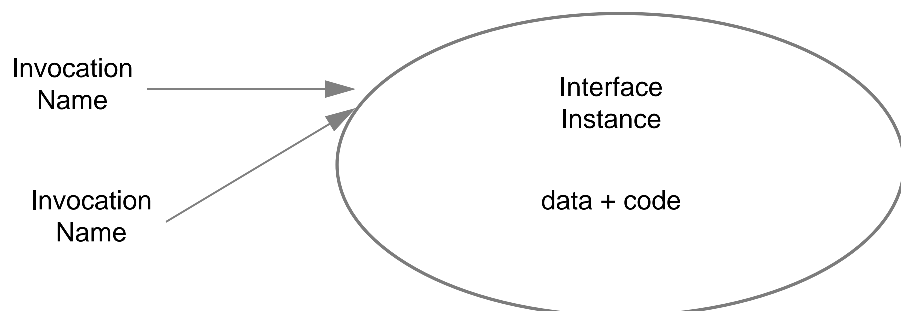
4 Binding components

This chapter defines what constitutes a binding and how bindings to (operational) interfaces are viewed and used in the current (asynchronous) computational and engineering models. The establishment of such a binding is discussed in the next chapter.

4.1 Computational bindings

In the computational model, an invocation name is bound to an instance of an interface as shown in figure 4.1.

Figure 4.1: Computational Binding



No distinction between local and remote interface instances can be made because the computational model requires access and location transparency.

The invocation name is not in scope until it is first bound, so an unbound invocation name can never exist and therefore cannot be invoked or copied.

Re-assignment is done by re-binding the invocation name to another interface instance. Copying is done by binding or re-binding another invocation name to the same interface instance.

The arguments (and results) of operations refer to interface instances. Since both the invoking (client) and invoked (server) object must share access to the same interface instance used as an argument or result, then the instance itself cannot be copied. Therefore, an argument (or result) must be passed by copying its binding.

4.2 Transparency and indirection

The computational model assumes location transparency and has nothing to say about where objects are located. To enable this assumption to be made, the semantics have been carefully constrained to prevent the direct manipulation

of data. All state is recorded by bindings to interface instances. An object records its own state as a set of bindings to interface instances. It manipulates its own state by rearranging its set of bindings and request other objects to change their state by invoking operations on the bindings it has to their interface instances.

When one object quotes one of its bindings as an argument (or result) of an operation, both client and server objects end up with a binding to the same interface instance referred to by the quoted binding. These bindings provide them with shared access to the argument (or result) interface instance.

Both location transparency and argument sharing force the engineering model to represent most bindings indirectly because the state representing an interface instance can't be in many places at once, so all but one of the bindings to it must be indirect.

Rather than make one binding first among equals, making all engineering bindings indirect will:

- hide the difference between local and remote bindings from the invocation mechanism; i.e. provide engineering access transparency
- make local copying of bindings very efficient

It is this indirection at the engineering level that is the key to engineering many other transparencies. This is because, once one level of indirection is placed between every client and server, neither of them can detect if multiple levels of indirection have been used. Therefore the compiler and binders are free to insert multiple levels of specialised stubs to implement particular transparencies, even for local bindings.

4.3 Engineering bindings

In the engineering model, bindings need to be access transparent at least to the client. All of the binding engineering can't be access transparent because it has to implement distribution. But the binding engineering should be as access transparent as possible so as to minimise the differences between local and remote bindings; thus enabling as much of the invocation mechanism as possible to be shared.

In the engineering model, the invocation name is bound at compile time to a (logical) memory cell[1] which can hold an invocation reference. The invocation reference is written into this cell at run time to complete the binding.

There is a mapping between each instance of an invocation name and the memory cell used to hold its invocation reference. Re-assignment of a binding is done by overwriting the old invocation reference with a new one. Copying of a binding is done by copying the invocation reference.

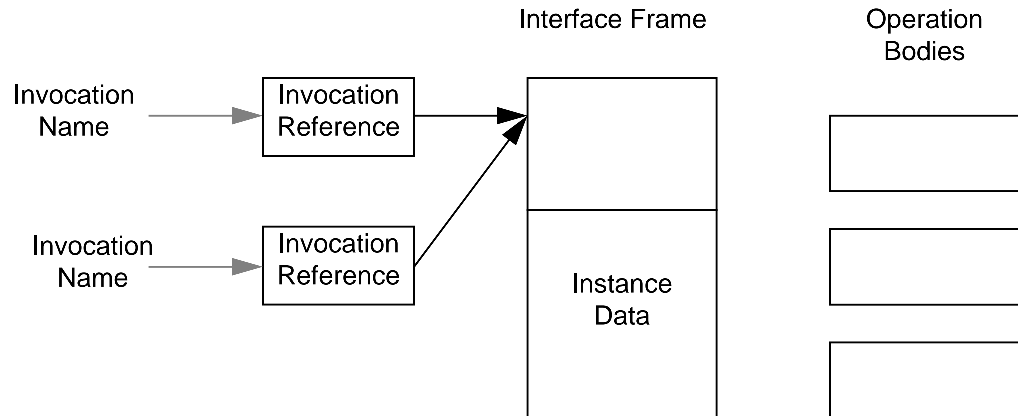
The engineering model relies on compilers not to generate code that attempts to invoke an incomplete binding (i.e. an invocation name bound to a cell which does not yet contain a valid invocation reference).

1. The actual address of the cell is not known at compile time, but its relative position in the program data structures is known, so that code can be generated to access it.

4.3.1 Local engineering bindings

For a local interface instance, the invocation reference addresses an interface frame, as shown in figure 4.2.

Figure 4.2: Local Engineering Program Binding



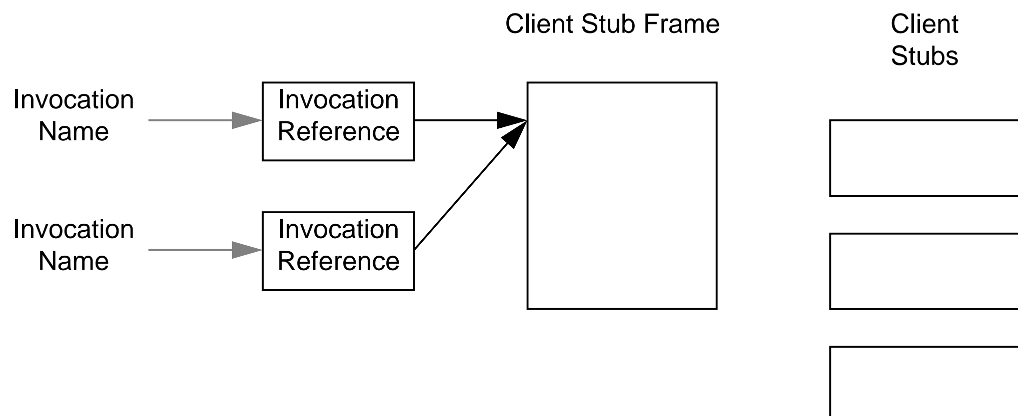
An interface frame represents an interface instance and contains (or refers to) all the instance specific data. It is associated with (language specific) data structures[1] which reference the operation bodies and which represent the "class" of the interface. When an operation is invoked on an invocation reference, the (language specific) invocation mechanism executes the corresponding operation body within the context of the interface frame.

When an interface instance is used as an argument (or result) of an operation in an interface instance located within the same capsule (address space) as the invoker, then the argument (or result) invocation reference can be copied.

4.3.2 Remote engineering bindings

For a remote interface instance, the invocation reference addresses a client stub frame, as shown in figure 4.3.

Figure 4.3: Remote Engineering Program Binding



1. These data structures are not shown in the diagrams, because they are dependent on the invocation mechanisms of the target language. For instance the C++ invocation data structures are either absent (at run-time) or hidden, but the C ones explicitly link the frame to the operation bodies via a dispatch table [RC.273].

The client stub frame acts as a local proxy for the remote interface instance and is constructed in such a way that the local invocation mechanism can use it in the same way as an interface frame.

The distinction between local and remote interface instances is transparent to both the invocation mechanism and the client code. This local access transparency is achieved by placing the body and stub pointers in the same place in the (language specific) class data structures associated with the interface or client stub frame.

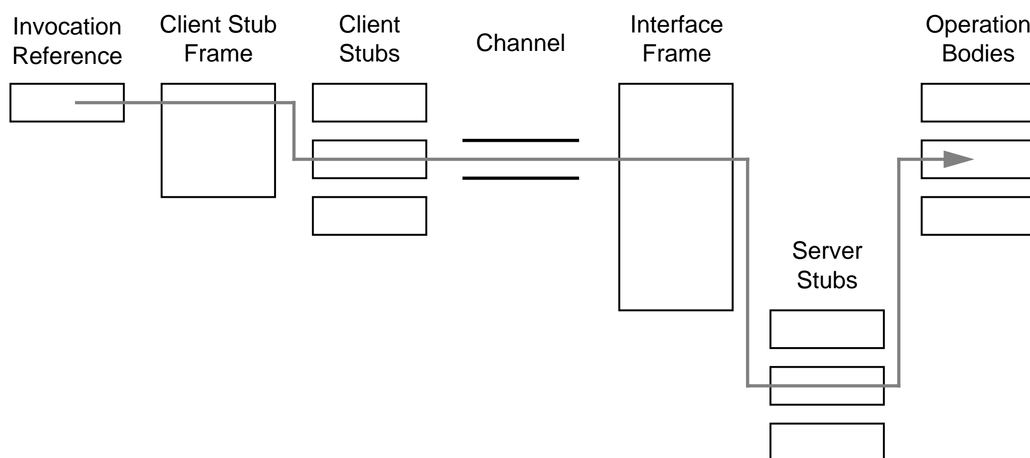
4.3.2.1 *Stubs*

The client code always invokes what it views as the server's operation body and passes arguments (and results) by copying invocation references to (and from) what it views as the body of a local operation.

The operation body is also transparent as to whether it has been invoked by a local or remote client; and therefore always views its arguments (and results) as copies of invocation references.

End-to-end access transparency is achieved by inserting a client stub, communication channel [§4.3.2.3] and server stub in the invocation path between client and server, as shown in figure 4.4. Both stubs are fully aware that they are dealing with a remote invocation.

Figure 4.4: Remote invocation path via client and server stubs



For remote invocations, the invocation reference points to a client stub frame which causes the invocation mechanism to invoke the client stub instead of the operation body. The client stub marshals [§4.3.2.2] the request, consisting of operation name and arguments, into a buffer and transmits this to the server stub using a suitable Remote Procedure Call (RPC) protocol. It then waits for the reply, unmarshals the termination name and results, and returns them to the invoker.

When the server's RPC protocol receives an incoming message, it calls the server stub corresponding to the operation being invoked on the (now local) interface instance. The server stub unmarshals the request, invokes the operation body, marshals the termination and results, then returns the reply to the client stub.

The server stubs are associated with an interface frame alongside the operation bodies and are only invoked for incoming remote requests. To avoid

confusion between the bodies and server stubs, the stubs must either be invoked by a different invocation mechanism or must have their names mapped (e.g. by adding a standard prefix to the corresponding operation name).

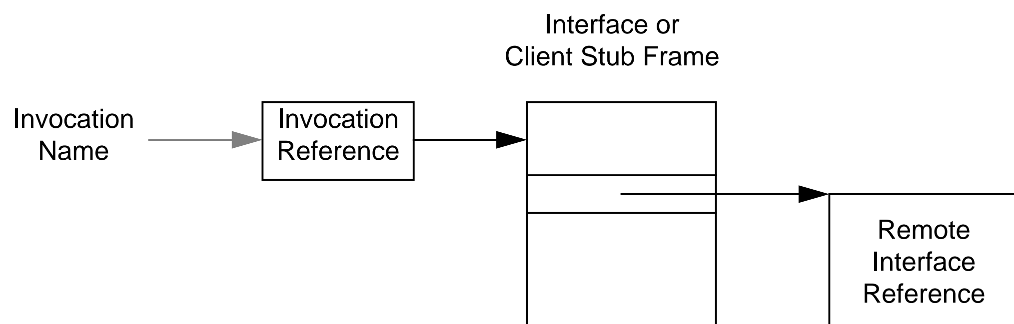
4.3.2.2 *Marshalling*

An invocation reference is a local memory address and has no meaning in another capsule, so if an interface instance is used as an argument (or result) of a remote invocation then a remote interface reference[1] must be constructed and passed instead.

A remote interface reference must contain the information needed for a remote capsule to establish a communications channel [§4.3.2.3] to the capsule in which the interface instance is located and to identify the particular interface instance within the capsule.

When an invocation reference is first used as an argument or result in a remote operation, a remote interface reference must be constructed. It can then be stored with the interface frame for future use, as shown in figure 4.5.

Figure 4.5: Remote Interface Reference



Whenever an invocation reference is subsequently used as an argument (or result) in a remote operation, its remote interface reference is marshalled into the buffer by the sending client (or server) stub.

Subsequently the receiving client (or server) stub will unmarshal the remote interface reference and create a new client stub frame to hold it. The address of this client stub frame is the invocation reference that is passed as the local argument or result.

A possible optimization would be to check, whenever a remote interface reference is unmarshalled[2], if there is an existing client stub or local interface frame for the interface instance it refers to. [Such an optimization would have to be carefully coordinated with garbage collection, migration, replication, passivation and relocation mechanisms.]

1. Unfortunately the ANSAware manuals and many other documents refer to both an invocation reference and a remote interface reference as an interface reference. This is a hangover from the time before the computational and engineering models had been fully prised apart. It has been the cause of such confusion that it is no longer safe to use the term interface reference on its own. The term invocation reference is preferred to the term local interface reference in order to avoid any confusion being caused by interpreting the latter as being restricted to *only* a reference to a *local* interface.

An end-to-end communications channel is not established when a remote interface reference is unmarshalled; see [§5.2.3] for when this is done.

The remote interface reference must be kept with the client stub frame [figure 4.5] in case:

- the invocation reference is ever used as the argument or result of a remote invocation

  [Because it is not for a local interface, the remote interface reference cannot be reconstructed.]

- a communications channel needs to be established.

When a remote interface reference is marshalled into a buffer, it must have a linearised (i.e. flat) structure. It must also be encoded according to the syntax of the presentation protocol being used on the communications channel.

A remote interface reference stored in its marshalled form will optimize the marshalling for bindings using the same presentation syntax. Alternatively, a remote interface reference stored as a tree which mimics its internal structure will provide fast access for establishing communication channels and faster marshalling for multiple presentation protocols. This is a trade-off which must be made for each system design.

Marshalling can be done either in-line (encoded as part of the stub code), or out-of-line (coded as functions called from the stubs). In-line marshalling may be slightly faster, but takes up considerably more space than out-of-line marshalling. If multiple presentation protocols are used, then the space penalty for in-line marshalling is increased.

4.3.2.3 *Communication channels*

For binding to a remote interface instance, an engineering program binding only represents the potential to communicate and must be augmented by a *communication* binding before an invocation can be made.

In order for a remote invocation to take place, a logical communication *channel* must be established between the client and interface stub frames, as shown in figure 4.6. [The interface frame doubles as a server stub frame.]

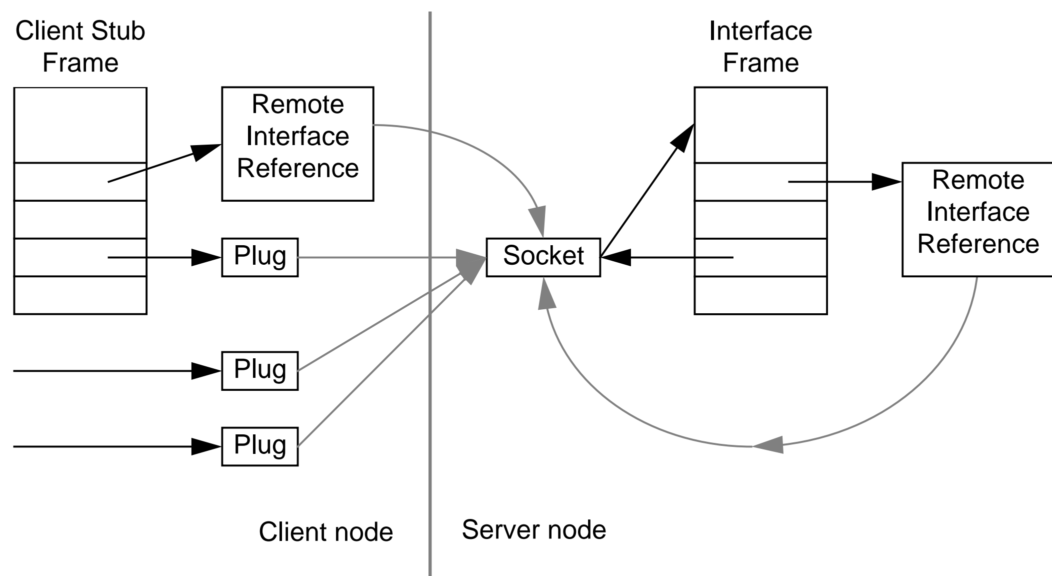
The client end of a channel is known as a *plug* and the server end is known as a *socket*. There is a many to one concurrent mapping between plugs and a socket which reflects the multiple clients each interface instance may have.

Each socket represents a particular interface instance and references the interface frame (i.e. contains a copy of the invocation reference). The socket is created before the corresponding remote interface reference because the remote interface reference contains the socket address within the capsule as well as the capsule's transport address.

Each client stub frame which is used for invoking operations (as well as storing a remote interface reference) must be associated with a plug. The plug cannot be created until the remote interface reference has been received and unmarshalled.

2. Another optimization for large scale databases (e.g. traders) which store bindings that they never intended to invoke themselves, is to discard or avoid creating the client stub frame. But this would break access transparency and allow the application program to manipulate the engineering data structures in a way that can't be type checked.

Figure 4.6: Communication binding



The job of the binding engineering is complete when the logical channel between a plug and a socket has been established. It is the job of the underlying communications engineering (i.e the RPC, session and transport protocols) to record the transient state of an invocation and to make and break connections when and if required.

4.3.2.4 *Remote interface references*

In figure 4.6, the pair of remote interface references associated with a client stub frame and an interface frame are logically identical; that is they refer to the same[1] interface instance but may be formatted differently [§4.3.2.2]. Both the client and server object can use their copy of the binding as an argument or result, so both ends need to keep a copy of the remote interface reference in order to be able to marshal their copy of the binding. In addition, the client stub frame needs the information in the remote interface reference to create a plug [§5.2.3].

A remote interface reference is completely unidirectional. The information it contains is specific to the server providing the service. It contains no information about any actual or potential clients.

A remote interface reference must contain enough addressing information to identify a particular socket in a particular capsule over a common protocol stack. These addresses are represented as a sequence of optional protocol stacks with each stack consisting of a sequence of layers and each layer consisting of a protocol name and a protocol specific address. Each protocol specific address is represented as a sequence of bytes with no internal structure.

A remote interface reference may also contain other information relating to Quality of Service, groups, relocation etc. See [RC.268] for the definition of (remote) interface references in ANSAware 4 and [APM.1021] for a full discussion of their functionality.

1. The remote interface reference in the client stub frame may be out of date, but the relocation mechanisms [§5.4] will fix this when an invocation is attempted.

Remote interface references can get rather large, but if a record is kept of which references have been transmitted and received on each channel, then subsequent transmissions can be optimized by marshalling the reference as a short integer indexing the order of its first transmission.

4.3.2.5 *Nonces*

Large distributed systems are created and maintained by a constantly evolving, heterogeneous, federated design and implementation process. It may still be possible to test a modified application before it goes live. But after its initial deployment, the underlying system can never again be tested off-line. There is an ever present danger that the engineering components of large system may fail or diverge as they constantly and independently evolve.

This danger is countered by constantly performing end-to-end checks using a nonce. This is a random number generated by a server whenever it creates a remote interface reference, inserted into the remote interface reference and checked whenever a client establishes a session connection [§5.2.4] to the service.

A random number is used because it is not possible to generate large numbers of unique identifiers efficiently in a large scale federated system.

There is a very small probability that an invalid remote interface reference could contain a valid nonce but this is not detected directly. It is the absence of nonce failures that provides a high degree of confidence that the binding and relocation mechanisms are working correctly.

Because of their randomness, nonces can't function as unique interface identifiers; anything that needs a unique identifier will have to pay the price of generating them without burdening every object in the system. Also if a nonce is used as a key to lookup remote interface references, then provision must be made for duplicate entries, as with hash tables.

4.3.3 Technology addresses

It is the function of the engineering binding and communication mechanisms to adapt to the locally available technology while preserving the portability and interoperability of the applications. Enabling the engineering to do this, requires a consistent approach to all technology addresses such as IP addresses, TCP port numbers, TSAPs etc. Therefore, the majority of the engineering software adheres to the principles that:

1. all technology addresses are represented as a variable length strings of bytes
2. all technology addresses are generated locally on demand and under the control of the engineering software
3. a technology address is just a hint to where the service probably is, based on where it once was

The first principle enables all technology addresses to be stored, compared, copied and marshalled by general purpose routines. The principle is only broken by the routines that directly drive a particular communications technology.

The second principle keeps technology addresses away from application programs and users. It also avoids the requirement for "well-known" addresses and drastically reduces the problem of stale addresses. For client/

server applications this principle requires the services of a trader [APM.1140], which begs the question: how does an application find the address of the trader?

Bootstrapping the address of a local trader is the only place where the second principle is broken and this is done by loading the trader's remote interface reference from the local environment; see [RM.100] for the various ways to do this.

The third principle allows for the migration, replication, recovery and passivation of a service. A more up to date location can be discovered from one of the relocators [§5.4] referenced by the service's remote interface reference.

5 Implicit binding

This chapter deals with when and how the (operational) interface bindings described in the previous chapter are established within the (asynchronous) computational and engineering models.

5.1 Computational view

The computational view of bindings is that they are always fully established and ready for use. As such their establishment is implicit.

5.2 Engineering view

A fully established binding on which invocations are being made consumes a significant amount of memory and communication resources. But many bindings are never used to invoke an operation, they are just stored in a database for later distribution or are merely in transit to the real client.

Therefore, fully establishing a binding all the way down the protocol stack as soon as it is created would waste a lot of resources. It would also create very severe scaling problems for many kinds of database server.

5.2.1 Engineering objectives

The objectives of the engineering are to provide the computational semantics while placing the least restrictions on the scale of applications and consuming the least resources. To do this it must:

- not allocate resources that are never used
- allocate resources as late as possible
- share resources as much as possible
- release resources as early as possible
- match the distribution of resources to the scale of the demand

5.2.1.1 Non allocation

If a binding is merely being stored for later distribution (such as a service offer in a trader) and it will never be used to invoke an operation, there is no need to allocate any communication resources to it at all.

The engineering should therefore only allocate communication resources when it is certain that the binding will actually be used.

5.2.1.2 Late allocation

If a server (e.g. a database) has millions of clients then few computers would be able to provide the resources to keep open millions of connections. But very

few of the clients which have bindings to such a server would actually be invoking operations on those bindings at any one time.

The engineering should therefore not allocate communication resources until the binding is needed.

5.2.1.3 *Resource sharing*

If a client capsule has a number of bindings to a server capsule then those bindings can share the same communications resources by serial multiplexing.

If a capsule has a limited number of communications endpoints available to it then these can be shared between the bindings on a usage basis.

5.2.1.4 *Early release*

Communication resources could be released as soon as an invocation had been completed and then reallocating them for the next invocation. But this would incur a processing and communications overhead in continually allocating and releasing resources for heavily used bindings.

A sensible trade-off would be to only release resources after a binding has been unused for a period of time, or when the resources are needed for another binding.

5.2.1.5 *Demand scaling*

If a server has millions of clients then even the resources needed to keep track of the existence of the clients would become prohibitive. On the other hand each client would only need to keep track of a single server. Given this, it is important that the fact that a client has a binding to a server, but is not currently invoking operations on that binding, should not require any resources to be allocated by the server.

This is why ANSA implicit bindings are unidirectional. That is, potential clients hold details about the server, but the server has no prior knowledge of its clients before they invoke it. In this way the resources used for dormant bindings [§5.2.2] are allocated to the objects generating the storage demand.

5.2.2 Dormant bindings

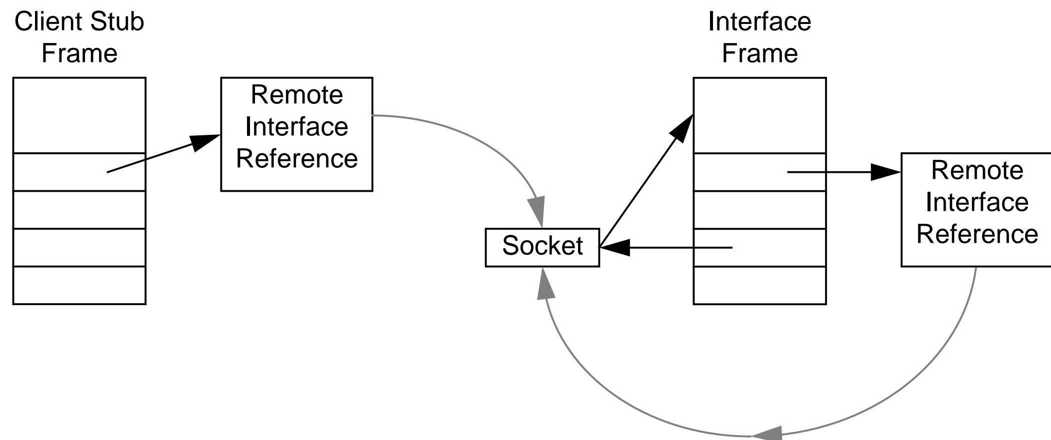
Following the non allocation and late allocation strategies, resources are only allocated when they are first needed. When a binding is first created it only needs enough resources to store the address information and can then lie dormant until it is first used to invoke an operation.

A remote interface reference is only needed when an invocation reference is first transmitted as an argument or result of a remote invocation. Since this needs to contain a socket address, a socket must be created. The remote interface reference will contain addresses in all the protocol stacks over which the server capsule is prepared to establish bindings. [The remote interface reference could be discarded after it has been marshalled and recreated on demand, but the socket must be kept in order to service any incoming invocations.]

A client stub frame is created when a remote interface reference is first received as an argument or result of a remote invocation. [The copy of the remote interface reference must be kept because it cannot be recreated.]

The components of a dormant binding are shown in figure 5.1 There may be

Figure 5.1: Dormant binding



client stub frames in many potential client capsules, but the server requires only one socket (and one copy of the remote interface reference).

5.2.3 Active bindings

When a dormant binding is first used to invoke an operation, it must be fully established before the invocation can proceed. This requires a plug to be created and associated with the client stub frame, as shown in figure 4.6. When the plug is created a protocol stack must be selected from those in the remote interface reference that is also available to the client capsule.

The binder's job is now complete and there is a logical channel between the plug and socket. There will not, however, be any connections between the client and server capsules. The underlying communications protocols will make and break these as required from the information stored in the plug.

5.2.4 Sessions

When an operation is invoked on a binding, an end-to-end session connection must first be established. This is used to track the progress of the invocation at both ends:

- large messages must be fragmented into packets[1] and reassembled
- lost or corrupted packets must be replaced
- duplicate packets must be suppressed
- invalid packets must be discarded

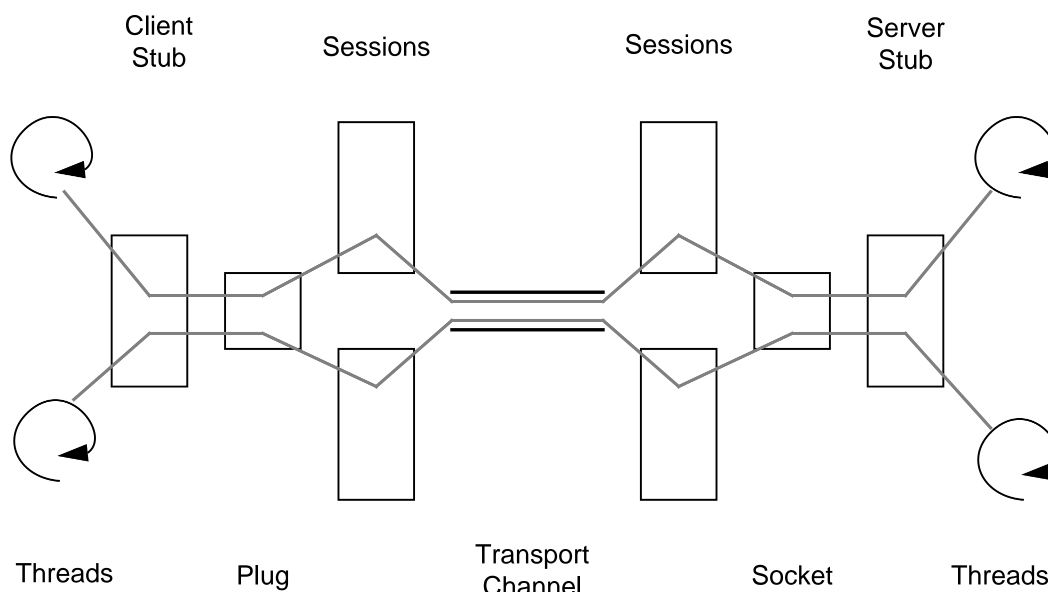
In addition, concurrent invocations of the same service by different clients, and by separate activities in the same client, must be kept apart. All these are the jobs of the session data structures and protocol.

For operational interfaces, the session protocol is always a Remote Procedure Call (RPC) protocol [RM.101].

1. The maximum packet size for a channel is the maximum data unit the protocol stack can handle, while message sizes are only limited by the virtual memory of the end systems.

A separate session entry is created for each client thread which invokes an operation on a binding. A matching session entry is created in the server when the invocation request is received. Between them the client thread, client stub, plug, client session, transport channel, server session, socket, server stub and server thread are the engineering resources which implement the computational view of a distribution transparent activity. Figure 5.2 shows the engineering components of two activities in a single client concurrently invoking the same service.

Figure 5.2: Components of concurrent distribution transparent activities



Note that the ANSAware RPC session protocol multiplexes sessions on the same binding down the same transport channel. Other clients of the same service will only share the socket and server stub.

The mapping between client thread and client session is fixed for the duration of a session because the client thread keeps local data on stack between invocations. But the mapping between server session and server thread may vary because the server operation keeps no local data on stack between invocations.

If each end deallocated its session entry as soon as it considered an invocation to be complete then this could cause confusion if the other end had experienced an error or delay and was out of step; especially if the session entry had been reallocated. Therefore the session entries are kept after the invocation is considered to be complete for long enough to be reasonably confident that there are no packets still in the network.

Allocating and deallocating sessions for each invocation would be wasteful considering that invocations on the same service are often bunched together quite closely. Also the RPC protocol can take advantage of closely bunched invocations by piggybacking an acknowledgement on the next request or reply, see [RM.101] or [BIRRELL & NELSON] for details, and the session indexes can be carried in the packets to provide fast lookup for existing sessions. This produces a protocol with the highly desirable characteristic of being more efficient under a high load than a low load.

Therefore sessions are kept for a period after they have become idle. This period will depend on network speed, scarcity of resources, average latency, connection costs, etc.; but is likely to be of the order of seconds rather than milliseconds or minutes.

5.3 Technology view

Establishing a transport channel for a connectionless protocol is straightforward; the client's plug already contains the server's transport address and the client's return address is sent in the invocation request packet. Each end then keeps the other's transport address in its session entry.

For connection oriented protocols, the server's transport address in the plug (and remote interface reference) is a contact address. The client establishes a connection to the server's contact address and this connection is then transferred[1] to a newly allocated server address for the duration of the connection. Each end then keeps its local connection identifier in its session entry.

The communications engineering may multiplex all channels between a given pair of capsules down the same transport connection.

It may also make and break transport connections in systems where transport endpoints are scarce or where transport connections are costly. This is done on a usage basis without disturbing the session connections.

If all the sessions using a transport connection have been deallocated then the connection can safely be disconnected.

5.4 Relocation

Services in a distributed system can change their location(s) for a variety of reasons, such as migration, replication, recovery and passivation. This change of location can occur before a channel is established or during an active session on a channel.

Such a location change may cause an attempt to establish a channel or invoke an operation to fail. In both cases, an updated remote interface reference must be obtained from a relocation service before the channel can be (re)bound.

To cater for this situation, a remote interface reference [RC.268] contains a sequence of one or more interface records. The first record is always for the service being referenced and any further records are for a set of relocation services [APM.1003, APM.1021, RM.101].

Relocation is done by successively invoking the relocation services identified by the relocation records in the existing remote interface reference, until one is found which returns an updated remote interface reference. If there are no relocation records or none of the relocation services has an updated remote interface reference then a binding error termination is returned to the client.

If an updated remote interface reference is found then an attempt is made to establish a channel and (re)invoke the operation which generated the initial

1. For protocols that can't transfer calls: the newly allocated transport address is returned to the client capsule, which then establishes a connection to it.

bind failure. If this is not successfully then another relocation is attempted using the relocation records found in the updated remote interface reference.

If the initial bind failure occurred during an invocation, then the invocation must be restarted. The at-most-once semantics of the RPC session protocol [RM.101] will ensure that the operation is not invoked twice by the server.

5.5 Garbage collection

The computational view of bindings is that all reachable bindings are usable. It is the job of the engineering to preserve this illusion in the real world of scarce and costly resources.

Since the engineering allocates resources without any intervention from the application program, it can't expect much help when it comes to deciding when to deallocate resources.

Locally, this is fairly easy to do. As discussed above, the transport and session resources can be efficiently garbage collected. The invocation references and client stub frames, plus their associated plugs and remote interface references are purely local resources and can be garbage collected by one of the many variants of the standard techniques of reference counting or mark and sweep.

The problem gets *interesting* when it comes to interface frames, plus their associated sockets and remote interface references, because references to them can be passed all round the planet. This considerably complicates the standard approaches to garbage collection because:

- reference counts may be lost during node crashes
- exhaustive search techniques scale very badly
- loops are much harder to detect without a reliable identity comparison[1]

This situation results in distributed garbage collection requiring significantly more resources to keep track of remote references than local garbage collectors require for local ones. But fortunately, most of the information needed to keep track of remote references is needed or is useful for other purposes, e.g.

- information about which remote references have been transmitted and received on each binding can also be used for optimizing their marshalling [§4.3.2.2] on subsequent transmissions
- (the absence of) proxy references [APM.1245] in gateways can be used to restrict the scope of garbage sweeps

1. It is impossible to provide unique identifiers for transient objects in a functional large scale federated system and relocation may invalidate address hints.

6 Explicit endpoint binders

This chapter discusses the programming language constructs required to implement explicit binding of operational and stream interfaces.

6.1 Explicit endpoints

With implicit bindings, all the active clients can have their plugs (logically) connected to a single socket because:

- the bindings are asymmetric (i.e. established from the client side)
- they all share the same Quality of Service (QoS)

As explicit bindings are symmetric and must be established from both sides, the clients need an endpoint reference that they can pass to the server.

Since clients use explicit bindings to obtain different and independent QoS guarantees, the server needs a different endpoint for each binding to a client so that:

- the server can pass a different endpoint reference to each client
- the server can differentiate its bindings

Stream interfaces are symmetric and both ends need explicit endpoints.

6.2 Endpoint binders

Endpoint references contain both technology addresses and engineering parameters (i.e. they are equivalent to remote interface references [§4.3.2.2]). To avoid exposing this information to an application, they are wrapped up in an interface which provides operations for an application program to manipulate them.

The application at each end of a binding has to invoke a local binder to establish the binding. A generic local binder that could establish a binding to any type of interface would either sacrifice type safety or require a dynamic type check every time a binding was established.

Explicit binding can be made type safe if the interface that the endpoints are wrapped up in is to an instance of a local binder that is specific to the type of the interface being bound. A compiler can generate the type of this binder interface and check that it is correctly used by the application. The actual binder provided at run-time will be a generic one that can bind any type of interface.

Both ends of an explicit binding must be established [§7] before the binding can be used and the binders must ensure that each endpoint is only involved in one binding at any one time.

6.3 Client endpoints

For implicitly bound operational interfaces, there are no client endpoints for a server to bind to; so a new construct needs to be added.

What is required is an expression which creates an instance of an endpoint binder of a particular type to represent each client endpoint; along the lines of:

```
clientEndpoint= "client" "of" TypeExpression
```

which would generate and return a new client endpoint binder of the specified type.

The (binding to the) local client endpoint binder can be passed to a server so the server can use its local endpoint binder to bind to the client's endpoint. The local client endpoint binder can be invoked to bind the client endpoint to a server endpoint and deliver a binding which can be used to invoke the server.

6.4 Client binders

The simplest client binder is an interface with just a bind operation, which takes a server binder as an argument and delivers a binding to the server as the result.

The type customisation is done by generating an interface type with a bind operation which takes an argument of the server's binder type and delivers a result of the server's type. So given the expressions:

```
  ServerType = type ( boo() ->() hiss() ->() )
; clientBinder = client of ServerType
```

the type of the newly generated `clientBinder` will be:

```
ClientBinderType= type
     ( bind (server:type ( bind (client:ClientBinderType)
                                ->()
                            )
             )
          ->(ServerType)
     )
```

Note that the `ClientBinderType` is type specific to the `ServerType` because its bind operation has an anonymous termination with a result of type `ServerType`. It is also type specific to the corresponding `ServerBinderType` [§6.6] because the type of the argument to its bind operation contains a bind operation whose argument type is `ClientBinderType`.

The two binder types are mutually recursive, but can be defined independently of each other by each type definition including the top level structure of the other type and using itself as a fixed point. For instance, the type of the argument of the bind operation in `ClientBinderType` is defined as a type expression containing a bind operation whose argument type is the very `ClientBinderType` currently being defined.

6.5 Server endpoints

The server endpoint for implicit binding is generated by the interface expression:

```
interface = "interface" "(" { signature operationBody } ")"
```

Whenever an interface expression is executed, it generates a new instance of the interface.

What is required for explicit binding is a way of generating a new server endpoint binder to a specific interface instance that can be passed to a client for explicit binding and which can be used locally to explicitly bind to a client endpoint. Following the style of the client endpoint generation, this would be along the lines of:

```
serverEndpoint = "server" "of" unit
```
[1]

which would generate and return a new type specific server endpoint binder for the specified interface instance.

A server endpoint can only be generated for a local interface instance. This can only be checked at compile time by flow analysis; otherwise any infringement can easily be caught at run-time.

The server endpoint binder can be invoked to bind the server endpoint to a client endpoint. The (binding to the) local server endpoint binder can also be passed to a client so the client can use its local endpoint binder to bind to the server's endpoint.

6.6 Server binders

The simplest server binder is an interface with just a bind operation, which takes a client binder as an argument and delivers a termination with no result. A result is not provided because the server does not require a binding on which to invoke the client.

The type customisation is done by generating an interface type with a bind operation which takes an argument of the client's binder type. So given the expressions:

```
  ServerType = type ( boo() ->() hiss() ->() )
; service = interface ( boo() ->()
                             [ output.string("boo") ]
                        hiss() ->()
                             [ output.string("hiss") ]
                      )
; serverBinder = server of service
```

The type of the newly generated `serverBinder` will be:

1. A unit is defined as: unit = name | invocation | block | object
and delivers a binding to an interface instance.

```
ServerBinderType= type
     ( bind (client:type ( bind (server:ServerBinderType)
                                 ->(ServerType)
                              )
             )
           ->()
     )
```

6.7 Stream endpoints

Stream binding is always explicit because there is no standard default QoS. The only asymmetry in a stream is the direction of the flows and therefore the types of the two endpoints will have their flows reversed with respect to one another.

In order to save a lot of trivial rewriting of stream types, a type operator is introduced to generate a type which has the reverse flows of another type:

```
TypeExpression= type | [“reverse”] unit
```

The reverse operator is only valid on types consisting of flows.

Both endpoints of a stream can then be generated by the expressions of the form:

```
streamEndpoint= “stream” “of” TypeExpression
```

where the TypeExpressions for both endpoints can be written out in full or one can be defined as the reverse of the other.

6.8 Stream binders

As for operational interfaces, a stream endpoint is an instance of a type specific local binder for the particular instance of the stream binding that it will be used to establish.

The stream endpoint binder is similar to the client and server endpoint binders, but both stream endpoints have a binder with the same type structure, so that given the expressions:

```
  StreamType = type ( >> ( boo() hiss() ) )
; streamBinder = stream of StreamType
```

the type of the newly generated `streamBinder` will be:

```
StreamBinderType= type
   ( bind (stream:type ( bind (stream:StreamBinderType)
                             ->(reverse StreamType)
                          )
           )
         ->(StreamType)
   )
```

The only difference between this stream endpoint binder and the one for the other end is the direction of flow in `StreamType`. The other binder type can be constructed by reversing the flows in the `StreamType` definition or by using the reverse[1] type operator for every occurrence of `StreamType`.

6.9 Client control of implicit bindings

Implicit bindings are established automatically without any involvement of the client or server applications. If a server wants to exercise control over a binding it can make it explicit and force its clients to cooperate. But in general a client has to put up with what the server provides.

However, it is possible to customise the client's binder to some extent without requiring any changes to the server's binder or application. The things that can be changed are those aspects of the binding over which the client binder has control, such as the timing of binding establishment, the opening and closing of sessions, and the choice of protocol stacks.

Therefore, to enable a client of an implicit operational interface to take limited control over the binding without any cooperation from the server, a client control binder generator is introduced:

```
clientBinder = "control" "of" TypeExpression
```

So given the expression:

```
clientBinder = control of ServerType
```

and the type of the simplest binder generated would be:

```
ClientBinder = type
     ( bind (server:ServerType) ->(ServerType) )
```

This would enable a client to control the time of binding establishment.

If the server's implicit binding is invoked, it will still be implicitly bound.

6.10 Binding failures

The binding service may not be able to establish a binding when requested due to a variety of reasons, such as:

- no common protocol
- cannot guarantee QoS [§7.4]
- endpoint already bound

These binding failures will need to be notified by (as yet unspecified) failure terminations added to the bind operations.

6.11 Customised binders

The endpoint binders discussed above are the simplest possible ones and are only specific to the type of the interface being bound. It is possible to generate binders with more operations (e.g. unbind, openSession, closeSession) and with extra arguments (e.g. QoS, multiple channels). Examples of more complex binders are given in the next chapter.

1. reverse reverse StreamType = StreamType

The generation of customised binders can be controlled by attributes.

But when more than just argument types are changed, what is available to the application programmer will be severely limited by the capabilities of the local binding engineering software. This will take the form of an attribute library containing binder templates plus a matching run-time library. Note that the type safety of the applications will depend on the type correspondence between the attribute and binder libraries as well as the correctness of the compiler's type checker.

The attribute controlled endpoint binder generators described in this paper enable an application programmer to succinctly select binders from the available options and have their subsequent use strongly type checked before run-time.

A programmer can then be confident that the required binders are available and will not be incorrectly used before an application is executed.

7 Explicit binding management

This chapter discusses the process of negotiating and coordinating the establishment of an explicit binding, and then monitoring and controlling its use.

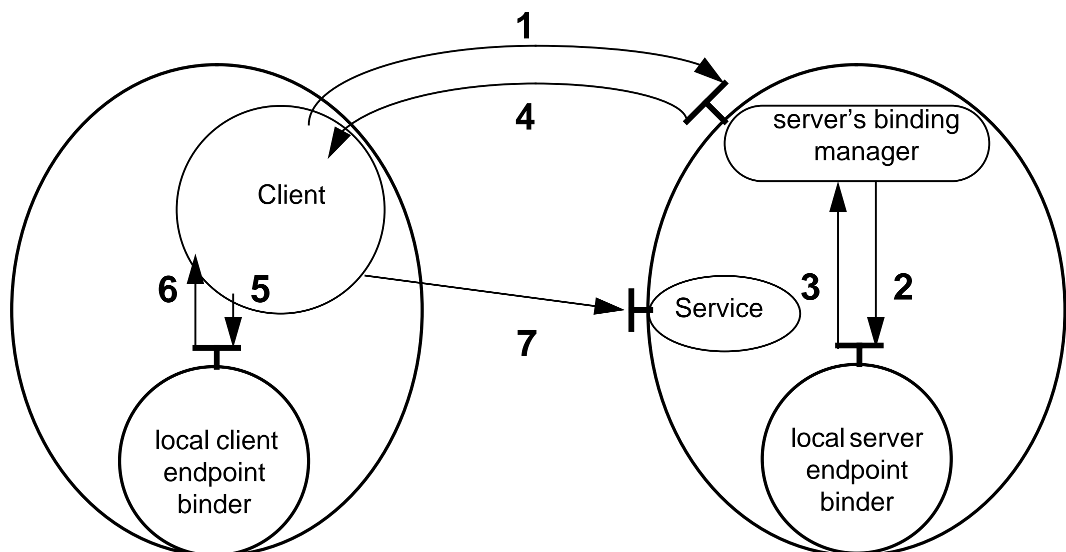
In accordance with the architectural principles of the binding model, all binding negotiation and coordination is done as a distributed application before each party establishes its local binding.

There are various ways of organising the establishment of an explicit binding, but they all follow the same basic principles.

7.1 Server binding managers

The simplest form of explicit binding management is where only the server has a binding manager as shown in figure 7.1. Because the server cannot

Figure 7.1: Using just a server binding manager



distribute the same service endpoint for use by all clients [§6.1], it is must always have a binding manager for explicit bindings. Therefore, it distributes a reference to a binding management service that makes new application service endpoints on request.

After the client has obtained the reference to the server's binding manager (either directly or via some third party such as a trader), an explicit binding is established as follows:

- the client generates a new client endpoint binder and passes its binding to the server in an implicitly bound invocation of the server's binding manager [request 1]
- the server's binding manager generates a new server endpoint binder for the service it is managing and invokes it to bind the new server endpoint to the client endpoint it has just been passed [request 2 and reply 3]
- the server's binding manager then returns the server endpoint binder to the client [reply 4]
- the client then invokes the client endpoint binder it created earlier to bind it to the server endpoint it has just received and gets back a binding to the service [request 5 and reply 6]
- the client then invokes or transmits to the service [request or signal 7]

This basic dialogue can be extended in a number of ways depending on the application requirements, e.g.:

- a set of multi-media streams can be established at the same time
- the single invocation of the server's binding manager can be extended into a negotiation about Quality of Service, billing arrangements, costs, etc.

7.2 Client and server binding managers

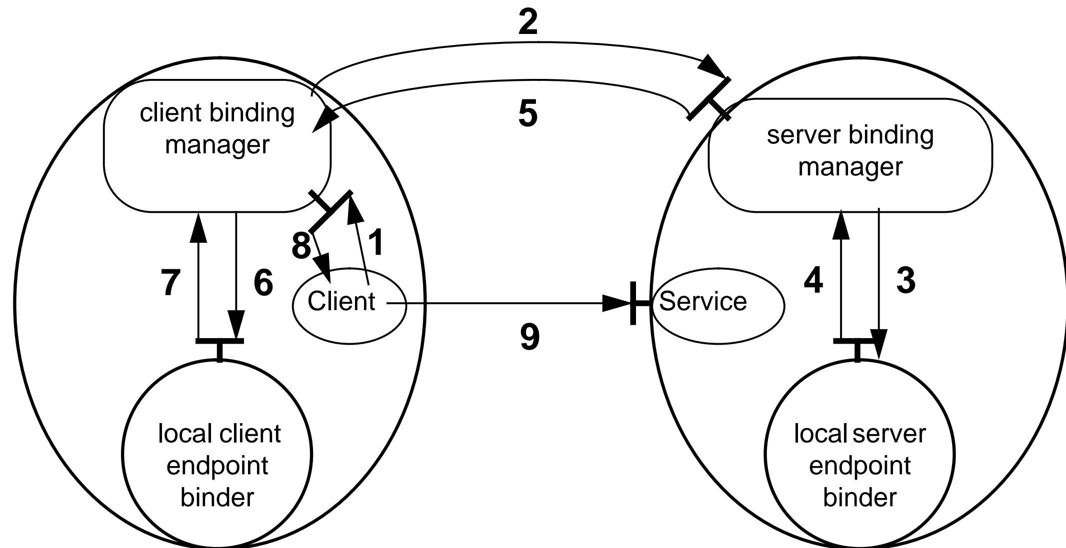
Although managing the establishment of explicit bindings as an application gives complete freedom to do whatever is required, most bindings are likely to conform to a small set of standard styles. These will lead to the development of a library of off-the-shelf server binding managers for the standard kinds of explicit binding.

Since the client's options are limited by what the server's binding manager is willing to provide, a matching library of off-the-shelf client binding managers will evolve to remove the drudgery from clients willing to use a standard binding.

The following simple dialogue between a matching pair of explicit binding managers is shown in figure 7.2:

- the client generates a new client endpoint binder and invokes its own binding manager with its new endpoint binder and the server's binding manager as arguments [request 1]
- the client's binding manager invokes the implicitly bound server's binding manager with the client endpoint binder as an argument [request 2]
- the server's binding manager generates a new server endpoint binder for the service it is managing and invokes it to bind the new server endpoint to the client endpoint it has just been passed [request 3 and reply 4]
- the server's binding manager then returns the server endpoint binder to the client's binding manager [reply 5]
- the client's binding manager then invokes the client endpoint binder to bind it to the server endpoint it has just received and gets back a binding to the service [request 6 and reply 7]
- the client's binding manager then returns the service binding to the client [reply 8]
- the client then invokes or transmits to the service [request or signal 9]

Figure 7.2: Using both client and server binding managers



Again, this basic dialogue can be extended in a number of ways, e.g.:

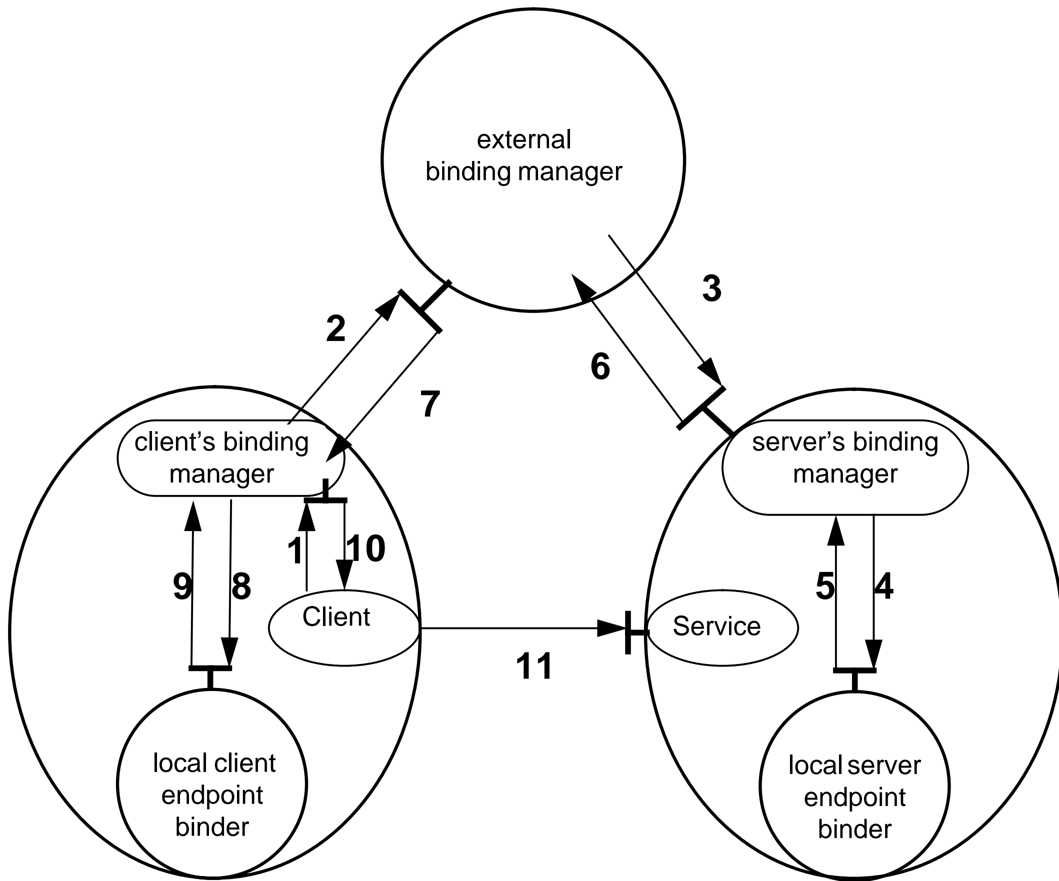
- the client's binding manager can interact with traders [APM.1140] to select and import the server's binding manager on behalf of the client
- the client's binding manager could negotiate federation agreements and organise the instantiation or cooperation of gateways [APM.1142]

7.3 External binding managers

Sometimes it is more convenient to manage the bindings from an object which will not be using the binding. This is particularly useful for multi-party bindings [§7.7] such as conferences. A simple example of an external binding manager dialogue for a two party binding is shown in figure 7.3:

- the client generates a new client endpoint binder and invokes its own binding manager with its new endpoint binder and the server's binding manager as arguments [request 1]
- the client's binding manager invokes the implicitly bound external binding manager with the client endpoint binder and the server's binding manager as arguments [request 2]
- the external binding manager then performs its own management functions, such as: billing, QoS negotiation, resource scheduling; then invokes the server's binding manager with the client endpoint binder as an argument [request 3]
- the server's binding manager generates a new server endpoint binder for the service it is managing and invokes it to bind the new server endpoint to the client endpoint it has just been passed [request 4 and reply 5]
- the server's binding manager then returns the server endpoint binder to the external binding manager [reply 6]
- the external binding manager then returns the server endpoint binder to the client's binding manager [reply 7]

Figure 7.3: Using an external binding manager



- the client's binding manager then invokes the client endpoint binder to bind it to the server endpoint it has just received and gets back a binding to the service [request 8 and reply 9]
- the client's binding manager then returns the service binding to the client [reply 10]
- the client then invokes or transmits to the service [request or signal 11]

7.4 QoS specification

One of the main requirements for explicit binding is the provision of quality of service guarantees. Ideally, the QoS specifications should be passed to the binding engineering in a type safe manner. But QoS specifications are specific to a QoS domain [APM.1151] and therefore will require binders to have different types and numbers of QoS arguments.

This problem can be solved by parameterising the endpoint expressions defined in [§6.3], [§6.5] and [§6.7] with attributes that instruct them to generate QoS domain specific endpoint binders containing a bind operation with the required number and types of QoS arguments. The syntax of a stream endpoint would then be:

```
streamEndpoint= "stream" "of" [attributeList] TypeExpression
```

and the other endpoint expressions would be similarly enhanced.

For example, an isochronous stream endpoint might be generated by:

```
  StreamType = type ( >> ( boo() hiss() ) )
; streamBinder = stream of <isochronous> StreamType
```

and the type of the newly generated binder might be:

```
StreamBinderType= type
   ( bind (stream:type ( bind (stream:StreamBinderType
                              bandwidth:BitsPerSec
                              maxJitter:MicroSecs
                             ) ->(reverse StreamType)
                               ->QoS_mismatch(String)
                               ->resource_limitation(String)
                       )
          bandwidth:BitsPerSec
          maxJitter:MicroSecs
         ) ->(StreamType)
           ->QoS_mismatch(String)
           ->resource_limitation(String)
   )
```

If the QoS arguments provided to both endpoint binders are incompatible or the infrastructure can't provide the necessary resources then the bind operations will return failure terminations [§6.10].

Note that the binder templates must already exist in the attributes library and be accessible at compile-time. The compiler just fills in the type of the endpoint argument of the bind operations. The run-time binding engineering must also match the binder templates, but its endpoint arguments will be written to accept any type. This is still type safe because the type check done by the compiler will be for the binder type it generates from the template.

7.5 Monitoring and control

As well as controlling the establishment of bindings an application may want to monitor and control the use of a binding. This can be done by requesting an enhanced endpoint binder in the same way as a QoS specific endpoint binder. The endpoint binder would have extra operations included which could be used to monitor and/or control the binding.

The exact monitoring and control facilities available would depend on what the local engineering could provide. For example, a monitoring and control interface for explicit session management might be generated by:

```
clientSessionBinder = client of <explicitSession> serverType
```

and the type of the generated binder might be:

```
ClientSessionBinderType= type
    ( bind (server:type ( bind (client:ClientSessionBinderType
                                action: type ( open() ->()
                                               close()->() )
                                ) ->()
                           open() ->()
                           close() ->()
                          )
            action: type ( open() ->() close() ->() )
           ) ->(ServerType)
      open() ->()
      close() ->()
    )
```

Each endpoint binder has two extra operations which enable a session to be opened or closed after the channel has been established by the bind operation. Each bind operation has an extra argument to allow the application to provide an interface for the local binder to call when the other binder opens or closes a session.

Much more elaborate monitoring and control operations can be added to binders and there is no requirement for the two endpoints to have identical binders as long as each conforms to the type expected by the other.

7.6 Multi-channel bindings

An application may wish to batch together a set of bindings (e.g. a multimedia stream) so it can bind and control them as a single entity. Again, this can be done using enhanced endpoint binders.

The binders could be generated using attributes to add extra channels to the main one, but this makes it difficult to associate other attributes with a particular channel. Therefore, the TypeExpression clause in all the endpoint binder generators is replaced by a TypeList which has a “batched” option to combine multiple channels into a single binding:

```
TypeList = TypeExpression
         | “batched” “(“ { [AttributeList] TypeExpression} “)”
```

Attributes can then by attached to the individual channels or to the whole binding.

The type of channels that can be batched will depend on the local engineering. For example, a three channel binder might be generated by:

```
multiStreamBinder = stream of batched ( AudioType MouseType
                                        VideoType )
```

and the type of the newly generated binder might be:

```
MultiStreamBinderType= type
    ( bind (stream:type ( bind (stream:MultiStreamBinderType)
                              ->(reverse AudioType
                                 reverse MouseType
                                 reverse VideoType)
                        )
           ) ->(AudioType MouseType VideoType)
    )
```

Only the single bindings to the multi-channel endpoint binders have to be passed around the system, and these will deliver three stream bindings when their bind operation is invoked. If all the channels cannot be established then none of them will be and a bind failure will be returned.

Multi-channel binders can be used with operational as well as stream interfaces, and with explicit combinations of both.

7.7 Multi-party bindings

So far all bindings have been assumed to have only two endpoints, but this is unnecessarily restrictive. Bindings with multiple endpoints can be very useful and can easily be established using attribute controlled endpoint binders. For example, a binder for a three party telephone call might be generated by:

```
threePartyBinder = stream of <threeParty> PhoneType
```

and the type of the newly generated binder might be:

```
ThreePartyBinderType= type
   ( bind (second:type ( bind (second:ThreePartyBinderType
                               third:ThreePartyBinderType
                             ) ->(PhoneType)
                       )
           third:type ( bind (second:ThreePartyBinderType
                               third:ThreePartyBinderType
                             ) ->(PhoneType)
                       )
          ) ->(PhoneType)
   )
```

This will only be successful if PhoneType is completely symmetrical (i.e. exactly the same signals can be transmitted as received), so the use of the threeParty attribute should trigger such a check.

For a conference call with a variable number of parties, the arguments of the bind operations would have to be lists of the appropriate type.

Symmetrical multi-party bindings could also be used for operational interfaces, provided each party consisted of an exactly matched pair of client and server endpoints. These can’t be constructed with any of the generators so far described, but could easily be generated by something like:

```
clientEndpoint ServerEndpoint = “pair” “of” [AttributeList] unit
```

Note: It would be interesting to examine whether asymmetrical multi-party binders could be constructed for operational interfaces and whether they could be usefully applied to client and server groups [APM.1002].

Note: It would also be interesting to explore the use of multi-cast addresses [VAN JACOBSON] for multi-party bindings.

7.8 Future work

Once the imperative mechanisms have been built and tested, a more declarative approach will be explored along with the development of more sophisticated binding managers and policies.

Some of the approaches that look promising are:

- attaching QoS and policy attributes to type and interface definitions so that implicit bindings can have application controlled QoS and binding policies
- using scope to control the establishment and duration of bindings (e.g. establish all bindings used in this block before entering it, or keep open a session while and only while a binding is in scope)
- use the trader to store QoS specifications and invoke binding managers

References

[APM.1000]

An Overview of ANSA, Rob van der Linden, APM Ltd., Cambridge UK.

[APM.1001]

The ANSA Computational Model, Owen Rees, APM Ltd., Cambridge UK.

[APM.1002]

A Model for Interface Groups, Ed Oskiewicz and Nigel Edwards, APM Ltd., Cambridge UK.

[APM.1003]

The ANSA Naming Model, Rob van der Linden, APM Ltd., Cambridge UK.

[APM.1007]

Security Services, John Bull & Andrew Herbert, APM Ltd., Cambridge UK.

[APM.1014]

DPL Programmers' Manual, Andrew Watson *et al.*, APM Ltd., Cambridge UK.

[APM.1015]

DPL Reference Manual, Dave Otway, APM Ltd., Cambridge UK.

[APM.1021]

ORB Interoperability, Andrew Herbert, APM Ltd., Cambridge UK.

[APM.1108]

Streams and signals , Dave Otway, APM Ltd., Cambridge UK.

[APM.1140]

A Designer's Introduction to Trading, Yigal Hoffner, APM Ltd., Cambridge UK.

[APM.1142]

A Model of Intercetion, Yigal Hoffner, APM Ltd., Cambridge UK.

[APM.1151]

A Model of Real-Time QoS, Guangxing Li, APM Ltd., Cambridge UK.

[APM.1245]

Introduction to Inter-operability and Interception, Yigal Hoffner, APM Ltd., Cambridge UK.

[ODP-3]

Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model, IOS/IEC JTC SC21 N8538.

[RC.268]

ANSAware 4.0 Interface References, Cosmos Nicolaou, APM Ltd., Cambridge UK.

[RC.273]

DPL Engineering, Dave Otway, APM Ltd., Cambridge UK.

[RM.100]

ANSAware 4.1: System Manager's Guide, APM Ltd., Cambridge UK.

[RM.101]

ANSAware 4.1: System Programming in ANSAware, APM Ltd., Cambridge UK.

[BIRRELL & NELSON]

Implementing remote procedure calls. Birrell A. D. & Nelson B. J., ACM Transactions on Computing Systems 2,1 Feb 1984.

[NAJM & STEFANI]

A formal operational semantics for the ODP computational model with signals, explicit binding, and reactive objects. Elie Najm, Ecole Nationale Supérieure des Télécommunications (ENST) <najm@res.enst.fr>; and Jean-Bernard Stefani, Centre National d'Etudes des Télécommunications (CNET) <jean-barnard.stefani@issy.cnet.fr>. October 3 1994.

[VAN JACOBSON]

Multimedia Conferencing on the Internet, Van Jacobson, SIGCOMM '94 Tutorial, University College London, London, UK.