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

Gateway Design and Implementation

Ben Crawford

Abstract

There is a clear need for gateways to bridge between systems with different properties in order to enable such systems to interoperate, and yet there is no clear understanding of how such gateways should be developed and managed. Prototyping work was therefore conducted to investigate design issues, and provide input into an architecture for interception.

This paper provides an introduction to some of the central issues for design of gateways. It introduces a general high level design, then describes prototyping work done to investigate and test this design; one of the prototypes was a gateway between the ANSAware Trader and the ORBIX MatchMaker. Bridging problems at both the platform and application level are considered. A number of important design issues are discussed in detail in the light of prototyping experience, and these are summarised at the end of the document.

APM.1303.01 **Approved** 6th October 1995
Technical Report

Distribution:
Supersedes:
Superseded by:

Gateway Design and Implementation

Gateway Design and Implementation

Ben Crawford

APM.1303.01

6th October 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

Table of Contents

1 Introduction

1.1 The need for gateways

The need for co-operation between an increasingly large and diverse range of information systems is widely accepted. There are fundamentally two ways to achieve co-operation:

1. Agree standards for protocols, processes, models and so on, which all members of a federation incorporate directly into their systems.
2. Let each member choose its own approach independently and build gateways to map between differences.

Ideally, standards are preferable because they reduce the interconnection and maintenance workload in the long term, if they are successful and widely adopted. However, in practice:

- A single universal standard which completely describes all aspects of systems is impractical, and it seems inevitable that there will always be a range of alternatives to accommodate.
- It may not be economically and technically feasible to support the necessary range of alternatives in each different system.

Co-operation between incompatible systems can be implemented by direct modification of the systems to make them compatible, but gateways are usually the preferred method (see §2.4 *Implementation of links*). Consequently, there will be a widespread, long term need for gateways to translate the information which passes between systems in order to reconcile the differences between them.

Thus there is benefit in trying to make the business of building, managing and maintaining gateways as inexpensive and reliable as possible. However, there is no clear understanding of how this should be achieved.

There is a distinction to be made between different kinds of gateways:

- **Application-specific gateways**, which are concerned with mapping between related but incompatible applications. These make existing applications available through different interfaces.
- **Platform-specific gateways**, which are concerned with mapping between different platforms. These allow existing services to be accessed from different platforms.

1.2 Overview of prototyping work

Development of gateway prototypes was necessary to gain experience and an improved level of understanding, so that an architecture for gateways and interception could be derived.

The prototyping work considered the following issues:

1. Building platform-specific gateways.
2. Building application-specific gateways.
3. Gateway management.
4. Monitoring of gateways.
5. Reuse of gateway software.
6. Automating gateway construction.

Two prototypes have been built to provide interoperability between two distributed system platforms, Orbix 1.2 and ANSAware 4.1.1.

1.3 The structure of this document

This document discusses the specification of links, outlines options and issues for gateway design and implementation, describes prototypes which have been built to investigate them, and draws conclusions about gateway design.

It is structured as follows:

1. Specification of relationships between systems (§2); this step is required even if gateways are not used to implement the relationship.
2. A general high level design for application level gateways (§3).
3. Description of prototyping work done to investigate gateway design issues. Differences between the two platforms used (ANSAware and Orbix) are considered (§4), and the two prototypes are described (§5, §6).
4. Discussion of important design issues that should be considered when developing gateways (§7).
5. A summary of the key issues (§8).

1.4 Related reading

A more general discussion of general federation issues, including an introduction to gateways and their uses, can be found in [APM.1514 95]. Some architectural terminology is drawn from the ANSA Interception model [APM.1507 95]. These two papers should preferably be studied in conjunction with this one.

This document aims to clarify and elaborate upon the commentary on gateways which is included in the ORB interoperability submission [OMG 94.9.32].

2 Specifying interoperation between systems

This section discusses the steps involved in specifying a relationship between systems and the link which implements it. These apply irrespective of whether gateways are used to implement the link.

The specification of a relationship should address three distinct areas:

1. The nature of the interoperation required.
2. Facilities required for managing the interoperation.
3. Non-functional requirements.

These areas are considered in turn below.

Once the relationship has been specified, a decision can be made about how it will be implemented. Relationships between incompatible entities may be implemented using gateways, or by modification of the entities themselves; the preferred approach is usually dictated by non-functional considerations.

2.1 Definition of the relationship

In order to establish relationships between systems, negotiation between the relevant authorities must occur for things such as access to resources. This document is concerned with technical rather than enterprise issues, so the areas of negotiation and agreement, whilst important, are largely ignored here. They are discussed at length in [APM.1514 95].

There is some relationship, or interface in a loose sense, required between the entities which must be defined and agreed. This relationship is expressed in terms of various aspects such as the communications protocol to be used, the operations which will be invoked, the information which will be exchanged, the type systems used to describe information and operations, security mechanisms, QoS requirements and so on.

The ways in which an entity represents these aspects are termed its characteristics, and areas where different characteristics hold are called domains [APM.1514 95].

The first step in enabling interoperation is to describe what the relationship between the entities will be for each of the aspects. These descriptions should address two things:

1. The information exchanged between the two entities.

   For example, “entity A requires an operation which it calls ‘Echo’, which takes a string and a reference to a string, and on return the reference should refer to a new string which is identical to the string passed in. Entity B will provide this functionality using an operation which it calls ‘bounce’, which takes a reference to a string and a string.”

2. For aspects where there is an incompatibility, the ways in which the information must be transformed as it is exchanged, so that it will be comprehensible to both entities.

   For example “the operation name must be translated between ‘Echo’ and ‘bounce’, the parameters must be reversed, and messages must be converted between the RPC which A uses, and the RPC which B uses.”

Some transformations may be very complex, and may involve information loss and use of defaults. Some of the aspects may not be represented at all by some of the entities being connected together (e.g. an entity may have no concept of security).

During this activity, it may become apparent that the effort required to rationalise the incompatibilities would be so great that it would outweigh the benefit; it may be cheaper to implement a new system.

Once the relationship has been completely defined, there is sufficient information available to specify the required link between the two entities.

2.2 Other functionality associated with links

Links may be required to include any or all of the following functions:

1. Translation of the information passing through the link, to overcome incompatibilities between the entities: this may require addition of information, or context information for services such as transaction management or security.
2. Additional functionality such as security which is not present in any of the entities being linked.
3. Monitoring of the information passing across the link.
4. Management functions, such as creation, deletion, provision of monitoring results, current load etc.
5. Control over what kinds of interaction may occur through the link, to allow separation of different administrative areas of the system.

The mappings should include all the information required for functions 1, 2 and 5.

Monitoring and management functions for the link may be provided using facilities supplied by the entities being linked, in which case these facilities should be connected using separate links in the interests of modularisation.

This is generally preferable, as it allows the link to be managed in a way which is consistent with the policies of the entities being linked, but in some circumstances it may be cheaper to implement such functions independently; if significant modification to the existing facilities would be required.

2.3 Non-functional requirements for links

Fundamentally, software implementing the link (link software) should be designed to maximise reuse and reliability, whilst minimising cost in terms of both run-time computing resources and development resources.

This requirement is discussed below in terms of two elements:

1. How software for translation between differing representations is structured.
2. How the commonly required functionality for monitoring, managing and controlling links is structured.

2.3.1 Reuse

Two distinct kinds of reuse are possible: reuse of general link software and reuse of translation software.

2.3.1.1 Reuse of general link software

Producing link-independent utilities for monitoring, management and control should pose no great problem. Translation functionality is likely to be specific, but it should be possible to find some reuse in this too, as often the same structures are being translated (e.g. operation invocations containing lists of parameters).

A good design should therefore abstract and decouple these kinds of functionality to enable them to be reused.

2.3.1.2 Reuse of translations

The worst case is a completely new agreement for each new link, with no use of existing standards or representations. However, many links are likely to be constructed for each:

- Processing environment
- Information service
- Organisation

There is likely to be commonality across the links specified for similar purposes, and this commonality should be exploited by agreeing representations which are widely applicable, and designing software to translate to and from such representations in a flexible manner.

2.3.2 Reliability

This is enhanced implicitly by modularisation and reuse, and in particular by auto-generation, as these make development and testing of links more effective.

Reliability is also improved by minimisation of complexity. Where a significant amount of processing, such as translation, is associated with a link, complexity is increased. This is likely to reduce the client-perceived reliability of a service, since it adds complexity: for services where high reliability is important, this may require special engineering in link software to maintain the required levels

It is also important to ensure that failure in link software can be distinguished by the infrastructure from failures in the entities being linked; this should be selectively transparent with respect to the linked entities where possible. It may be impossible to make the transparency selective, where these entities have not been developed with detection of link failure in mind.

2.3.3 Run-time efficiency

Efficiency requires that the link software is as simple as possible, though this may mitigate against reuse.

Software which implements links should be designed to enable load-balancing. There are often difficulties with this, since the entities may be remote from each other and under different managements.

Link software should maximise throughput and minimise latency whilst using computing resources shared with clients and servers in a controllable fashion. For example, it is of little benefit to run a high-performing gateway on the server platform if this compromises the performance of the server. The ability to flexibly re-distribute the link software is desirable.

2.3.4 Minimisation of development time

Modularisation is important to enable reuse and reduce resting time. Reuse, and especially auto-generation, is important in reducing development time.

Some method of using information describing the characteristics of the entities being linked to generate the software for specific translations would be of great benefit, even if only relatively simple transformations could be handled.

2.4 Implementation of links

Fundamentally, a link between incompatible entities may be implemented in two ways:

1. Modification of one or more of the entities to provide the required compatibility and functionality.
2. Development of software to implement the link in the form of one or more gateways.

The key issues for deciding which approach is preferable are:

1. Since gateways are decoupled from the entities which they connect, the links which they implement are easier to modify; for example they enable flexible redistribution. Management of the link is easier to decouple from management of the entities being linked. Similarly, gateways are easier to reuse than additions to existing systems.
2. Modification of the existing entities avoids the need for translation and reduces overall complexity, thus improving performance and reliability. This is beneficial provided that the software being modified is highly maintainable, but over time, successive modifications will tend to reduce maintainability.

Gateways may be relatively expensive for small and infrequent modifications, and they may introduce unacceptable performance overheads in extreme cases. However, in the absence of stable standards, gateways are usually the preferred solution because their higher modularity implies better testability and maintainability, despite higher overall complexity.

The remainder of this document discusses gateways, though much of the discussion is relevant to any approach.

3 General top-level gateway design

This section describes a high-level design for gateways, which aims to provide the requirements described in §2.

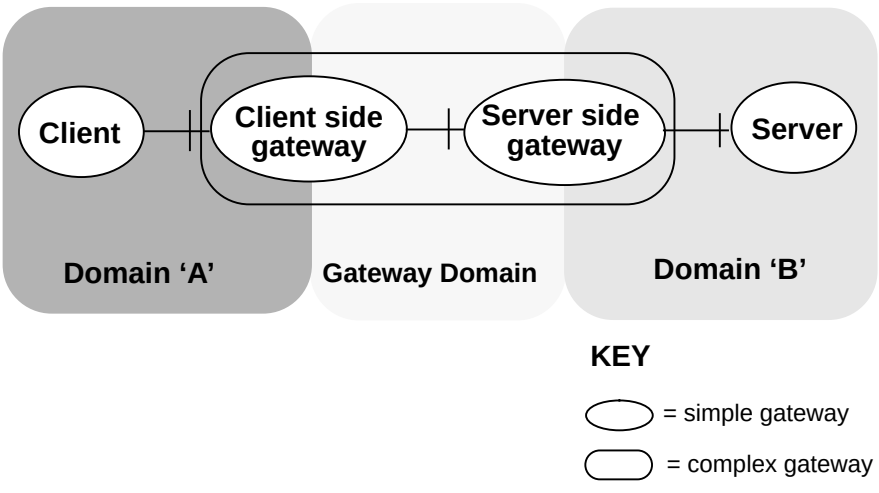
The ‘domain’ concept refers to areas of similarity in terms of one or more characteristics. As a simplification, discussion in this section groups all characteristics into one kind of domain and does not specialise, so ‘domain’ is roughly synonymous with ‘system’.

3.1 Composing complex gateways from simple gateways

There are two approaches to connecting existing domains: either to transform directly between the representations and mechanisms which they use, or to specify a third domain to use as an intermediate representation, and to use two transformations, first from the source domain to the intermediate domain, then from the intermediate domain to the target domain.

A gateway which translates directly between domains is termed a simple gateway, whereas a gateway which translates between domains via one or more intermediate domains is termed a complex gateway. A complex gateway is thus composed of two or more simple gateways (see figure 3.1).

Figure 3.1: Structure of complex gateways



The figure shows a complex gateway composed of two simple gateways, the client side gateway and the server side gateway (henceforth CsG and SsG). The CsG communicates with clients in the normal fashion for Domain ‘A’, and the SsG communicates with servers in the normal manner for Domain ‘B’.

Some method of communication between the simple gateways must be defined and agreed; decisions about inter-gateway protocols, models and mechanisms must be made with care, as they have important implications. These models, protocols and mechanisms effectively constitute another domain, called a gateway domain, which exists inside the complex gateway.

The advantages and disadvantages of gateway composition are discussed in the remainder of this section; more detailed design issues follow.

3.1.1 Benefits of gateway composition

Composition of gateways, rather than use of single gateways which exist in all the domains being connected, is sometimes unavoidable, and also has several important advantages.

3.1.1.1 Each domain retains control of interaction

Each domain is linked to the complex gateway via its own simple gateway, over which it has control. This enables domain management to prevent or enable interaction across domain boundaries, and thus protect itself against violations of agreements, for example by withdrawing the gateway if the load through it exceeds an agreed maximum. It also enables restrictions like security to be imposed by each member of the federation independently.

3.1.1.2 Domains allocate their own resources

Each simple gateway requires resources. For a domain to allow an external agent control of its computing resources implies a high level of trust, which may be unacceptable. Since each domain retains responsibility for creating and deleting its own simple gateways, it retains full control of how and when its computing resources are allocated.

3.1.1.3 Use of incompatible mechanisms

It is desirable for gateways to use mechanisms which are commonly employed in the domains being linked. There may be incompatibilities between these mechanisms or the software which implements them. For example, different RPC mechanisms are often implemented with conflicting assumptions about the control of their environment, thereby making it impossible for them to co-exist in the same process or address space. In such cases it may be easier to create a composite gateway from simple gateways, each with its respective RPC mechanism, linked by a third communication mechanism.

3.1.1.4 Improved maintainability

Gateways often need to be enhanced or upgraded, for example to cater for changes to the domains being connected (e.g. a new version of a software platform with a new RPC mechanism). Since new versions of each simple gateway may be inserted into a complex gateway without impact on the other simple gateways, maintainability is vastly improved in comparison with direct simple gateways.

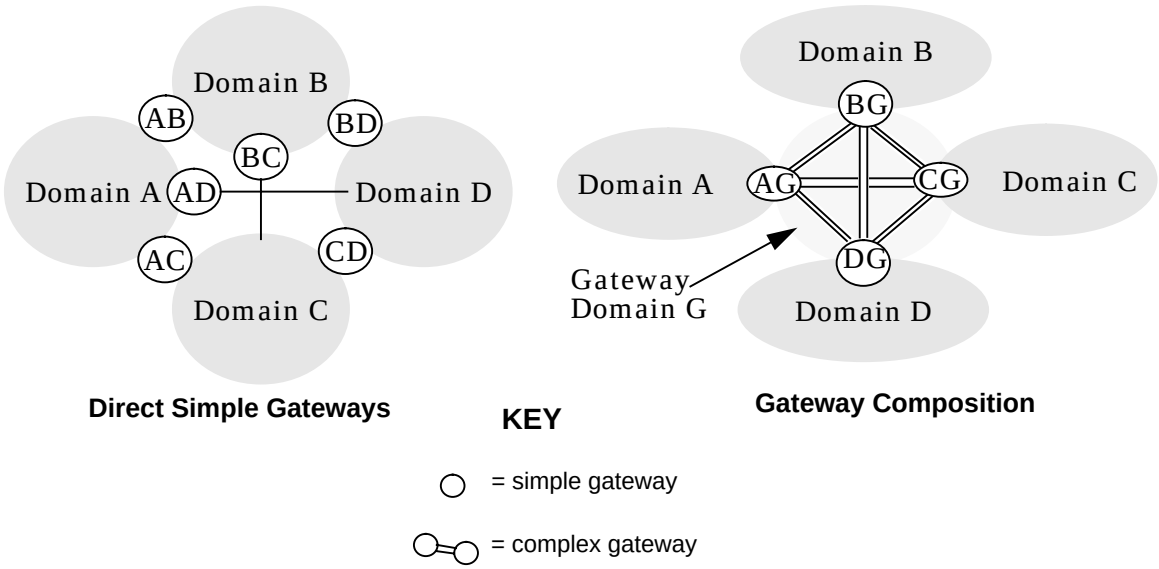
3.1.1.5 Reuse of translation software

When large numbers of domains are involved in federation, there is a proliferation of gateways. Use of the gateway composition approach can greatly reduce the total number of simple gateways that must be constructed, provided that suitable intermediate representations can be agreed.

The scenario illustrated in Figure 3.2 involves four domains, which are fully interconnected: six complex gateways are required. It can be seen that with gateway composition, four simple gateways are sufficient as opposed to the six required with direct gateways.

In general, gateway composition can reduce the number of gateways required to fully interconnect n different domains from ((n squared - n) / 2) to n.

Figure 3.2: Proliferation of gateways



A single agreement can rarely be achieved on a wide scale, but smaller scale sharing of gateway domains can enable significant reuse, and if desired this can be expanded by linking different gateway domains together using further gateways.

3.1.1.6 Dynamic re-configuration

Once gateway domains are shared, several different simple gateways may offer the same services in the same gateway domain. Consequently, a CsG may dynamically re-bind to different SsGs offering the same service, transparently to the client. This implies that a client can be bound via gateways to an instance of a new server type at run-time, provided that the service being consumed does not involve holding state.

With direct simple gateways, this can only be done through production of a new gateway implementation per server type per server domain. In order to transfer a client to a server of a different type, it would then be necessary to bind the client to a new gateway instance with a different implementation, and this might require state about the client session to be transferred between different gateway instances.

Composition thus offers more stratagems for resilience than direct simple gateways. For example, if a stateless server fails, and there are no other servers of the correct type in a position to offer service, the CsG of a complex gateway may be able to locate and bind to another SsG for a different server type or domain, enabling service to be maintained, possibly even without interruption. Similarly, this approach could be used to improve service, should a more suitable service type become available after a long term binding has been established.

Changing of servers in this way must be managed carefully to ensure that the policies of the client and its administration are upheld. Any decision to form a new binding should probably be referred to the clients infrastructure, and also to the client itself should it have elected to be concerned with such matters.

3.1.2 Disadvantages of gateway composition

There are also some disadvantages:

1. An intermediate domain, including appropriate mechanisms, protocols and models for representing interactions between the domains being linked, must be defined and agreed. It may be very difficult to define representations which have all the required characteristics (these are discussed in §7.3). In particular, the need to transform all information into a 'lowest common denominator' representation should not overly penalise performance, and the representation should be able to accommodate evolution of the domains being linked, such as the introduction of new technology and new administrative policies.
2. Use of a larger number of distinct gateways in a single end to end invocation path implies higher complexity. This in turn suggests lower reliability and performance.

   Performance is likely to be impacted significantly, because of the overhead of two translations instead of one. Messages from one source to multiple targets can be optimised by only translating the information into the gateway domain representation once.
3. It may be harder to identify where an error has occurred with a series of gateways. Gateways may pass terminations back to other gateways. In order for these to be interpreted, shared representations for terminations must be agreed. The more different representations are traversed, the more likelihood there is of information being lost. In addition, more gateways on the invocation path imply more chance of processing and communications errors and so more chance of lost messages. Complex gateways may therefore entail a higher maintenance overhead in some circumstances.

3.1.3 Conclusions regarding gateway composition

Gateway composition is the preferred approach in the general case. As configurations of gateways become more complex, the performance disadvantages become increasingly offset by optimisations, and are far outweighed by advantages of autonomy, separation, maintainability and reuse. As more and more services require interconnection, the effort expended on defining gateway domains will become increasingly easy to justify due to more foreseeable reuse opportunities.

However, for simple links, especially one-off gateways between only a pair of different domains, the use of direct simple gateways is perfectly legitimate.

3.2 General gateway algorithm

Essentially, the task performed by any gateway, whether client side, server side or both, includes the following stages:

1. Receive one or more incoming operation invocation(s) from one or more clients (clients may be gateways).
2. Map the information associated with the incoming invocation(s) (e.g. operation names, parameters, context) into the form required for the outgoing invocation(s)
3. Invoke the outgoing invocation(s) in the correct order
4. Collect the responses to the outgoing invocations.
5. Map the information (e.g. parameters, terminations, context) returned by the outgoing invocations into the form required by the incoming invocations. This may require further invocations, involving iteration of stages 3, 4 and 5, such as committing transactions, retrying or sending invocations with information returned by prior outgoing invocations.
6. Reply to the incoming invocations.

More detailed algorithms will vary according to the interaction model used, parallelism, and other issues.

3.3 Specific examples of gateway algorithms

High level algorithms are given below. They describe an RPC gateway composed of two simple gateways, which use traditional IPC to exchange serialised invocations. Some unspecified translation of operation names and parameters is illustrated.

Pseudo code for creation and deletion is given, followed by pseudocode for the bodies of both the client side and server side gateways.

These algorithms make simplifying assumptions, namely:

1. Single threading. This limits the gateway to executing one incoming invocation at a time. It is possible to have multiple clients per single-threaded server, since the gateway blocks pending invocations until the previous one has completed.
2. One server per gateway. In order to allow multiple servers and clients it would be necessary maintain information about servers in the gateway.
3. One to one mapping of incoming invocations to outgoing invocations. This, coupled with assumption 2 means that gateways need not hold state describing pending invocations.

3.3.1 Gateway creation and deletion

This example is for the client side gateway; the server side one is similar.

Figure 3.3: Gateway creation and deletion algorithms

```
startClientsideGateway(interfaceRef other_gate_ifref)
BEGIN
      Initialize resources;

      interfaceRef this_gate_ifref = createInterface();

      status = startGateway(this_gate_ifref, other_gate_ifref);
      /* If other_gate_ifref == NULL waits for setup call and sets
      /* other_gate_ifref else makes setup call to the other gateway */

      IF status != ok
            abort("Cannot bind with Serverside Gateway!");
      ENDIF;

      trader_ifref->Export(this_gate_ifref, Type,
                                          Advertizing requ's, Constraints);
      /* Clientside gateway may export its interface to trader */
      /* Serverside gateway would import here */

END startClientsideGateway

stopClientsideGateway()
BEGIN
      status = stopGateway(this_gate_ifref, other_gate_ifref);
      /* Notifies other Gate of my demise, severs comms link */

      destroyInterface(this_gate_ifref);
      Free resources;

END stopClientsideGateway
```

3.3.2 Client side gateway algorithm

The CsG provides operations to be invoked by the client; these phrase the operation request in the inter-gateway protocol and send it to the other simple gateway.

Figure 3.4: Client side RPC gateway algorithm

```
Op1(argType1 arg_name_1, ..., argTypeN arg_name_N)
BEGIN
      operationName operation_name = Op1;
      operationName out_operation_name = map_operationName(operation_name);

      terminationName termination_name;
      terminationName out_termination_name;

      outArgTypeM out_arg_name_M = map_argTypeN_outArgTypeM(arg_name_N);
      /* For each outgoing arg, may be n:1 map */

      gatewayMessage response_message(MAX_MESSAGE_SIZE);
      gatewayMessage invocation_message = marshall
                  (out_operation_name, out_arg_name_1, ..., out_arg_name_N);

      status = send(other_gate_ifref, invocation_message);

      /* Send the Domain B invocation request using whatever */
      /* comms mechanism has been agreed for the gateway domain */

      IF status != ok
            abort("Write to Serverside gateway failed!");
      ENDIF

      status = receive(other_gate_ifref, response_message);
      /* Wait for the response */

      IF status != ok
            abort("Read from Serverside gateway failed!");
      ENDIF

      response_message.getTerminationName(out_termination_name);

      CASE (out_termination_name)
      OK:
            /* Take appropriate action, e.g. commits,further invocations */
      TERMINATION_NAME_1:
            /* Take appropriate corrective action, such as retrying */
      TERMINATION_NAME_2:
            /* As above */
      ENDCASE

      termination_name = map_terminationName(out_termination_name);

      ResponseMessage.getArgN(out_arg_name_N);
      /* Unmarshalling for each modified outgoing parameter */

      arg_name_M = map_outArgTypeN_argTypeM(out_arg_name_N);
      /* For each incoming parameter, again may be n:1 */

      RETURN (termination_name, arg_name_1, ..., arg_name_N);
END Op1;

Op2(argType1 arg_name_1, ..., argTypeN arg_name_N)
BEGIN
      /* As with Op1 */
END Op2;
```

3.3.3 Server side gateway algorithm

The SsG analyses operation name, and invokes the appropriate routine to unmarshall the inter-gateway message and make the invocation on the server.

Figure 3.5: Server side RPC gateway algorithm

```
serversideGateBody()
BEGIN
      GLOBAL interfaceRef server_ifref =
                  trader_ifref$Import(Type,Advertizing requ's, Constraints);
      /* We may obtain server interface references by any one of */
      /* several means, use of a trader is only one example */

      operationNameType operation_name;
      gatewayMessage invocation_message(MAX_MESSAGE_SIZE);
      gatewayMessage response_message;

      WHILE (TRUE)
            status = receive(other_gate_ifref,invocation_message);
            /* Wait for incoming invocations */

            IF status != ok
                  abort("Bad invocation from Clientside gateway!");
            ENDIF

            invocation_message.getOperationName(operation_name);
            CASE (operation_name)
            OPERATION_NAME_1: Op1(invocation_message, response_message);
                              break;
            OPERATION_NAME_2: Op2(invocation_message, response_message);
                              break;
            ENDCASE
            status = send(other_gate_ifref, response_message);
      ENDWHILE
END serversideGateBody;

Op1(gatewayMessage invocation_message,gatewayMessage response_message)
BEGIN
      operationNameType operation_name = OPERATION_NAME_1;
      operationNameType out_operation_name;
      terminationName termination_name;
      terminationName out_termination_name;
      /* Also declare arg_name_x and out_arg_name_x ... */

      invocation_message.getArgN(arg_name_N);
      /* For each incoming parameter */

      out_operation_name = map_operationName(operation_name);
      out_arg_name_M = map_argTypeN_outArgTypeM(arg_name_N);
      /* For each outgoing arg, may be n:1 map */
      out_termination_name = server_ifref$Op1
                                    (out_arg_name 1, ..., out_arg_name_N);

      CASE (termination_name)
      OK:
            /* Take desired action */
      TERMINATION_NAME_1:
            /* Take desired corrective action */
      ENDCASE

      termination_name = map_termination_name(out_termination_name);
      arg_name_M = map_outArgTypeN_argTypeM(out_argName_N);
            /* For each incoming parameter, 1:n */

      response_message = serialise(termination_name,
                                    out_arg_name_1, ..., out_arg_name_N);
END Op1;
```

3.3.4 Comments on example algorithms

3.3.4.1 IPC between gateways

Much of the apparent asymmetry between the two example algorithms is due to the difference between the way these gateways interact with clients and servers (using RPC to invoke operations) and the way they interact with each other (exchanging serialised messages via traditional IPC).

This could be rationalised by using a procedure call mechanism (local or remote) in the gateway domain instead of IPC, though as already noted the platforms being connected may employ incompatible mechanisms which prevent this. Where IPC is used, a better approach is to wrap the IPC messaging in functions which model the normal procedure call model to make the rest of the code more reusable.

Ideally, any gateway should be implemented with a procedural interface which can either be accessed inline or wrapped up in a remote invocation mechanism.

3.3.4.2 Modelling mappings

Mapping functionality is modelled here as a collection of functions with names like *map_typeX_typeY*, where X is a type in the source domain and Y is a type in the target domain (note that *map_X* implies *map_X_X*). The full names of these should really be *map_domainA_typeX_domainB_typeY*, but depending upon the scope of the symbols, context relative naming can be used.

It is generally worthwhile providing standard functions for such mappings, as in most cases the mapping of a particular pair (or more) of types will be the same between any two domains. However, these general mappings may often need to be specialised for specific cases, since the semantics of types are rarely complete.

3.3.4.3 Use in case studies

The following chapters (§4, §5, §6) discuss prototyping work which was conducted to investigate gateway design issues, and in particular to test the design described in this chapter.

The prototypes described overcome a range of different incompatibilities, including RPC mechanisms; their algorithms closely resemble the examples given in this section.

4 Differences between ANSAware and Orbix

Both of the case studies involved links between ANSAware 4.1.1 and Orbix 1.2. There are several differences between the ANSAware and Orbix platforms which must be overcome in platform-specific gateways between them; these are discussed below. The two platforms are described in [APM ANSAware 93] and [IONA 94].

4.1 RPC mechanisms

ANSAware and Orbix both provide RPC mechanisms for remote invocation. However, they cannot co-exist in a single capsule. This implies that the CsG and SsG for ANSAware-Orbix gateways must reside in separate capsules.

4.2 Type systems

ANSAware uses C, and Orbix uses C++, so the language-supplied type systems are compatible. However, there are differences in the way in which the two platforms represent standard complex types like sequences and unions: it is possible to map ‘sequence of X’ and ‘union of X, Y’ given mappings for X and Y, though some information is lost, such as the maximum sequence length in Orbix which has no equivalent in ANSAware.

4.3 Exceptions and terminations

The result of an operation is a called a termination in ANSAware and an exception in Orbix. In ANSAware terminations are simple enumerated values indicating some fault, whereas Orbix defines a more complex exception with an id, completion status and arbitrary application-specific information.

4.4 Interface references

In Orbix these are strings built by concatenating strings identifying host, process, object and interface type. In ANSAware they are more complex, including lists of records describing alternative network addresses for an interface and a nonce. Neither platform makes any provision for transparent inclusion of foreign interface references, though ANSAware does support use of different protocols.

4.5 IDL

Both platforms use an IDL to specify interfaces and generate stubs. The two IDLs are different but mappable, so two IDL specifications are required, even for an application-independent gateway.

4.6 Implementation Language

Orbix objects are coded in C++, whilst ANSAware uses C. This makes it difficult to develop common functionality which can be reused for gateways in both platforms.

4.7 Library incompatibilities

ANSAware redefines some of the standard C library routines (e.g. IO routines altered to be threadsafe) which other C or C++ libraries generally need to make use of. This can cause integration problems where code is written under Orbix using these routines.

4.8 Management issues

The facilities for object management differ significantly between ANSAware and Orbix.

ANSAware provides independent trading, binding and factory services. Orbix, on the other hand, combines these services together and manifests them as the Orbix daemon. Thus, a 'bind' call to the Orbix daemon is equivalent to both an import and a bind in ANSAware, and it may also perform a factory role if the required server is not currently instantiated.

For some purposes this simplicity is beneficial, as it may hide unwanted complexity from the client, but it imposes restrictions and allows clients less control.

For example, the Orbix factory service will not instantiate multiple instances of a single persistent service. This is because the Orbix daemon cannot be used to bind to services unless they are registered in the implementation repository. Servers which need to be instantiated externally (i.e. not using the Orbix daemon factory service; these are termed 'persistent servers') must be registered as CORBA 'shared servers' and Orbix places a restriction that there cannot be more than one instance of a shared server active at one time.

Such differences make it difficult to standardise gateway management functionality across the two platforms.

5 Case Study 1: A simple application gateway

This study was primarily aimed at platform issues. In order to investigate platform gateways, it is best to begin by linking clients and servers for very simple services, to minimise the complexity and confusion caused by application differences.

The ANSAware Echo example is an appropriate choice.

5.1 Design and implementation issues

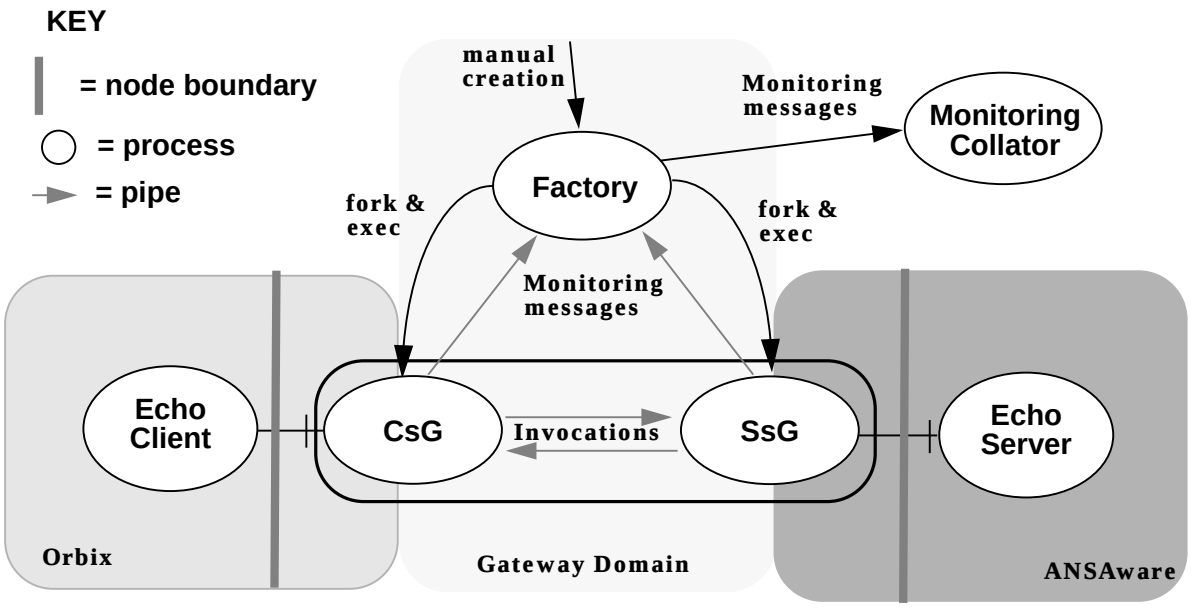
The logical design and algorithms used were essentially as described in §3 *General top-level gateway design*.

The approach selected was to use a factory process to spawn two child processes, one for each simple gateway - in this case, an ANSAware gateway and an Orbix gateway. Each simple gateway interacts with other objects in its own platform using the platforms own RPC, and they communicate with each other using some other agreed mechanism. In the long term this would be IIOP, but for prototyping purposes simple unix pipes were used.

The two gateways produced monitoring information which was sent, again via pipes, to the factory process which also acted as a monitoring collator.

The physical design is depicted in figure 5.1.

Figure 5.1: Simple application gateway physical design



The factory process is invoked manually. It creates and initialises the pipes and then forks and execs the two simple gateways, passing them the handles for the appropriate pipes.

The factory process then alters function, and listens for monitoring messages from the gateways which are written to pipes as shown. The collator partially orders these messages using Lamport clocks and writes them to disk for later use.

Four simple gateways were implemented, producing two complex gateways, one for the ANSAware-Orbix direction and one for Orbix to ANSAware.

Monitoring messages were chosen to enable visualisation of the clients, servers and simple gateways with the 'DEMON' visualisation tool, despite the fact that monitoring functionality was only present in the gateways, not in clients and servers.

Mapping problems were relatively trivial, the only differences in the applications being operation names and termination names.

5.2 Management and setup issues

Work focused on the creation and initialisation of the gateways. Deletion issues were not considered in this study; they are more naturally tackled when gateway creation is automated.

Because of the architectural differences between Orbix and ANSAware described in §4.8 *Management issues*, the setup process for the two gateways differs considerably.

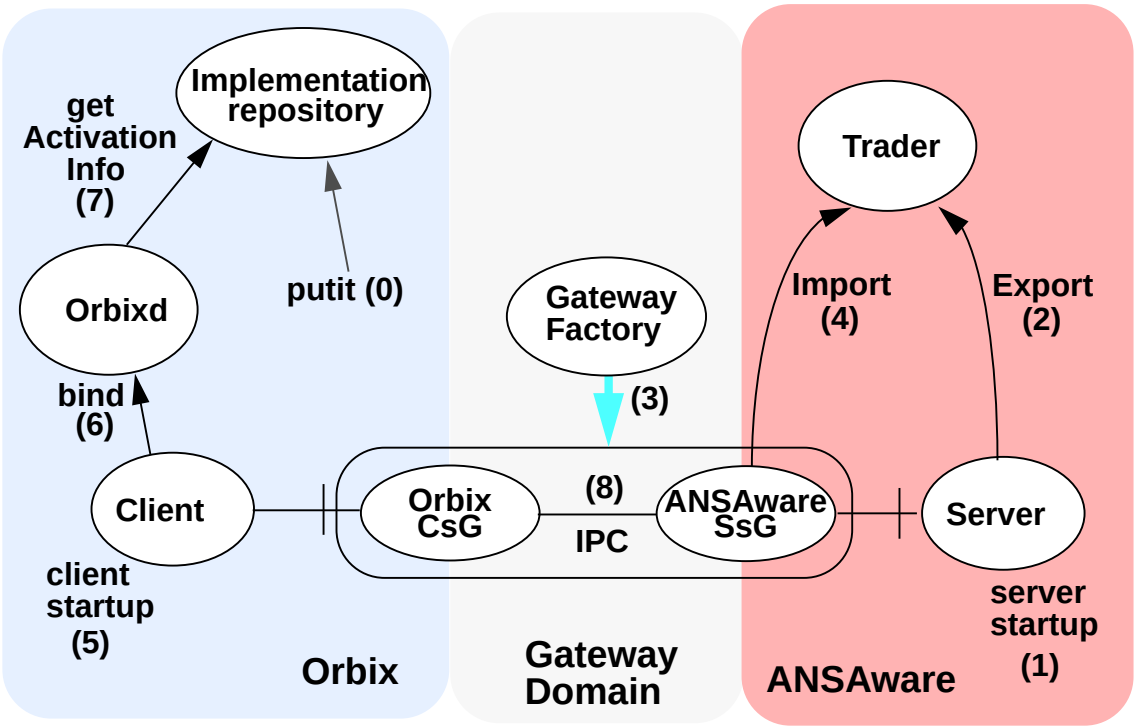
The ANSAware gateways used the ANSAware Trader to import and export interfaces, whilst the Orbix gateways used the Orbix daemon via CORBA *bind*() and *impl_is_ready*() calls.

This is not the preferred approach: if two domains have been fully linked, a gateways is created as a result of the transfer of an interface reference across the boundary, and so the gateway factory would pass this server interface reference to the gateway in the factory call. Use of the trader by the gateway to obtain the server reference was a short cut used in this artificial example, because there was no parent gateway between the domains through which an interface reference could be passed.

5.2.1 Orbix client - ANSAware server

The Orbix client - ANSAware server set-up sequence is shown in Figure 5.2.

Figure 5.2: Setting up an Orbix-ANSAware gateway



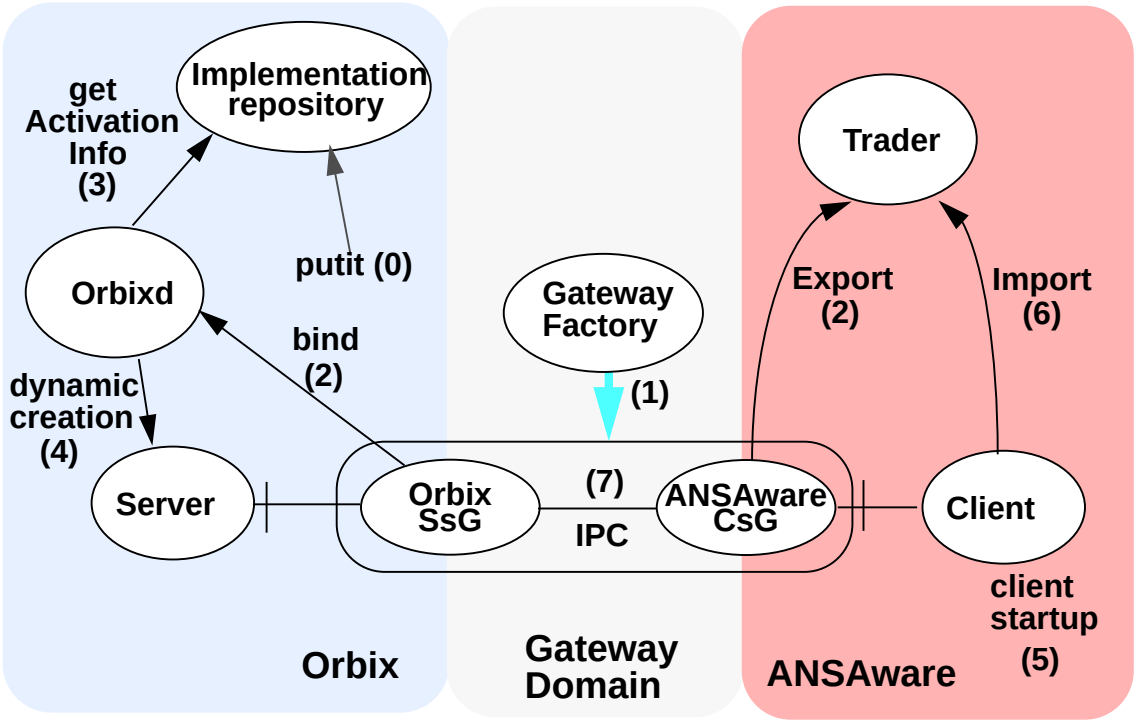
Ordering of events:

0. The Orbix CsG interface must be registered with the implementation repository using *putit*(), to allow clients to bind to the gateway. This may happen any time before stage 4.
1. The ANSAware server is executed manually.
2. The ANSAware server exports its interface to the ANSAware Trader.
3. The gateway factory is then invoked manually to create the gateway.
4. The SsG imports the server interface reference from the ANSAware trader and the CsG advertises itself to the Orbix daemon.
5. The client is executed manually.
6. The client binds to the gateway via the Orbix daemon.
7. The Orbix daemon looks up activation information for the CsG type.
8. The client invokes the ANSAware server via the gateway.

5.2.2 ANSAware client - Orbix server

The Orbix server - ANSAware client set-up sequence is shown in Figure 5.2:

Figure 5.3: Setting up an ANSAware-Orbix gateway



Procedure:

0. The Orbix server interface must be registered with the implementation repository using *putit*(), to allow clients to bind to it. This may happen any time before stage 2.
1. The gateway factory is invoked manually to create the gateway.
2. The ANSAware CsG exports its interface to the ANSAware Trader. The Orbix SsG makes a bind call to the server via the Orbix daemon.
3. The Orbix daemon looks up activation information for the server type.
4. The Orbix daemon instantiates a new server if a suitable one is not already instantiated.
5. The ANSAware client is executed manually.
6. The ANSAware client imports the CsG interface reference from the Trader
7. The ANSAware client invokes the Orbix server via the gateway.

6 Case study 2: A gateway between traders

Whilst the first case study investigated the issues of building platform-specific gateways, this study was primarily aimed at overcoming application differences, though it also involved tackling some deeper platform issues.

By constructing an application-specific gateway to provide a Trader-like ANSAware interface to the Orbix MatchMaker, it was hoped to investigate:

1. Immediate resolution of interface references crossing domain boundaries, by automated creation of child gateways.
2. Complex mappings between applications

A trader gateway was also an important step towards achieving sufficient infrastructure spanning the Orbix and ANSAware domains to investigate more complex issues such as distributed management and trader federation.

The ANSAware Trader is described in [APM ANSAware 93], whilst the MatchMaker is described in [APM 1384 94]. A basic knowledge of traders is assumed in the rest of this chapter.

The ANSAware Trader offers several interfaces, including *Trader*, which is used for import and export of offers, and *TrFed*, *TrType* and *TrCtxt*, which are used for managing the Trader. This prototype only offers the *Trader* interface; this is by far the most complex one.

The MatchMaker offers various interfaces. In order to implement the *Trader* interface, two of these were utilised; the *ExportManager*, and the *OldTrader*, which closely resembles the *Lookup* part of the ANSAware *Trader* interface.

6.1 Design and implementation issues

The design of the two simple gateways produced was again essentially as described in chapter 3.

The Trader-Matchmaker gateway (henceforward TMMG) is created manually using a gateway factory of the kind used in the first case study. Monitoring messages are structured as with the Echo gateway.

A modified factory was produced for automated child gateway creation, called the child gateway factory. This functions in a similar fashion to the standard gateway factory, but is invoked with the interface reference which is passed through the parent gateway during *Export* and *Import* operations, so that the SsG of the child gateway can bind to the server whose interface reference is being passed. The child gateway factory passes back the interface reference of the child CsG to the parent - in this case, the TMMG.

Monitoring requirements were more complex because it was necessary to include parent and child gateways, so Lamport clocks were passed with the factory call. Collation software was modified to collect and order monitoring messages for the desired set of gateways in a configurable way.

Some functionality is common to both Orbix and ANSAware gateways. This was implemented in a C++ library, which was used directly in the Orbix simple gateway, and was used via a C wrapper in the ANSAware simple gateway. This approach was useful in reducing code volume and testing time, and improving maintainability.

6.2 Mapping between applications

Mapping is vastly more complex than with the first case study; a variety of difficult problems were encountered due to fundamental conceptual differences between the two applications. To understand these it is necessary to consider the operation mappings in detail.

6.2.1 Trader.Register to ExportManager.Export

	Register			Export	
IN	InterfaceSpecificationName	STRING	IN	Properties	sequence <NamedProperty>
IN	NamingContext	STRING	IN	Properties	sequence <NamedProperty>
IN	PropList	STRING	IN	Properties	sequence <NamedProperty>
IN	Ref	ansa_InterfaceRef	IN	Properties	sequence <NamedProperty>
IN	Capsule	CapsuleRef	IN	Properties	sequence <NamedProperty>
			OUT	Result	OfferId
OUT		Result	OUT		Exception (Failed)

In the Trader, an offer is identified by the interface reference and properties associated with it. In the MatchMaker, an offer need not have an interface reference associated with it, and is identified by a unique *OfferId*. This difference requires no work here though, since the ANSAware client is not expecting any information about the offer to be returned.

An offer in the Matchmaker is described by an arbitrary list of properties, whereas the Trader treats some properties (interface type name, naming context, interface reference and capsule reference) as special. The special Trader properties need to be mapped to members of the MatchMaker property list.

The *Result* expected from the Trader is a structure holding a boolean indicating success/failure and a flag indicating the cause of any failure; the MatchMaker raises an exception to indicate failure, and this holds an ID which indicates the cause of failure.

6.2.2 Trader.Delete to ExportManager.Withdraw

	Register			Export	
IN	Ref	ansa_InterfaceRef	IN	Offer	OfferId
IN	MatchingConstraints	STRING			
OUT		Result	OUT		Exception (Failed)

Differences here appear less significant on cursory inspection, but in fact are more problematic.

The Trader *Delete* operation takes an interface reference and a constraint list to use to find the correct offer. This is not necessary in the MatchMaker because of the use of a unique *OfferId*, but this *OfferId* cannot be used across the TMMG because ANSAware clients cannot hold such *OfferId*s.

There are two ways of overcoming this difference. The first is to hold the ANSAware interface reference and ANSAware properties list as properties in the MatchMaker; then when a *Delete* invocation arrives, a *Search* for this information in the MatchMaker yields the correct *OfferId*, which is then used in a *Withdraw* call. This is inefficient, but requires the least development work. It is only possible because the MatchMaker service holds state in a flexible way, however; this solution is not applicable for most gateways.

Another solution is for the gateway to hold a map from (*Ref* + *MatchingConstraints*) to (*OfferId*). This is not a pleasant solution, and it introduces other problems in itself, but it was adopted in the interests of studying the problems of application specific gateways at a more general level.

6.2.3 Trader.Lookup to OldTrader.Select

	Lookup			Select	
IN	InterfaceSpecificationName	STRING	IN	InterfaceType	ServiceDescription
IN	NamingContext	STRING	IN	ConstraintExpression	MatchingCriteria
IN	MatchingConstraints	STRING	IN	ConstraintExpression	MatchingCriteria
IN	Policy	LPolicy			
			IN	SearchPolicyName	LinkSearchPolicy
OUT		LResult	OUT	NamedProperties	ServiceProperties
OUT		LResult	OUT	any	Service
OUT		LResult	OUT		Exception (Failed)

The *Trader.Lookup* operation maps to *OldTrader.Select* only if the *Policy* parameter has a value of *Lookup_Random*.

Mappings of *InterfaceSpecificationName* to *ServiceDescription* and *MatchingConstraints* to *MatchingCriteria* are fairly trivial.

The MatchMaker does not use a hierarchy of contexts for offers, but the name space used in the trader can be mapped to offer properties in the MatchMaker, so *NamingContext* can be treated as another property to add to *MatchingCriteria*. However, the constraint specification language supported by the MatchMaker is not powerful enough to match subcontexts properly, so additional processing is needed in the gateway to eliminate MatchMaker offers which have inappropriate contexts.

LResult is a *CHOICE* of reason for failure or a sequence of structures describing offers found. Each of these structures contains context, type, properties and an interface reference, and since we are using *Lookup_Random*, the sequence will only hold one member.

The structure describing the offer can be built entirely from *ServiceProperties*, except for the interface reference which comes from *Service*. The reason for failure can be taken from the contents of any exception returned by *Select*.

6.2.4 Trader.Lookup to OldTrader.Search

	Lookup			Select	
IN	InterfaceSpecificationName	STRING	IN	InterfaceType	ServiceDescription
IN	NamingContext	STRING	IN	ConstraintExpression	MatchingCriteria
IN	MatchingConstraints	STRING	IN	ConstraintExpression	MatchingCriteria
IN	Policy	LPolicy			
			IN	SearchPolicyName	LinkSearchPolicy
			IN	PropertyNameProfile	ServicePropertyNames
OUT		LResult	OUT	RepositoryInfo	Result
OUT		LResult	OUT		Exception (Failed)

The *Trader.Lookup* operation maps to *OldTrader.Search* only if the *Policy* parameter has a value of *Lookup_All*.

Mappings of *InterfaceSpecificationName*, *MatchingConstraints* and *NamingContext* are as with *Select*.

ServicePropertyNames is a list of *PropertyNames*, which governs the order in which properties are given in property lists returned by the operation. It also acts as a second filter, since matching offers must have all the properties in this list. There is no equivalent to this in *Lookup*; ANSAware clients expect the properties back in whatever order they were specified when registered.

Result is a list of matching offers, which take the form of *OfferId*s, each with a list of properties. These properties can be translated back into the information required for *LResult*, as they include the interface reference.

However, there is a subtle but very dangerous difference between the lists of interface references passed back by *Trader.Lookup* and *OldTrader.Search*. Where the ANSAware Trader encounters *ProxyOffer*s, these are resolved to produce a directly usable reference which supplies the requested service. However, this is not so with Monitored offers, which are the MatchMaker equivalent: *Search* passes back the reference of the Monitor, so this must be resolved by the gateway using *ReferenceManager.LookupReference* to produce a directly usable offer which can be passed to the client.

Another important problem is posed by this operation. The approach taken with *Export* was to use immediate resolution, and create a child gateway for the interface reference as it passed through the trader gateway. However, with the *Search* operation, many interface references are passed and the chances are that most of them will not be used. Immediate resolution is therefore inappropriate. This highlights an interesting point, namely that resolution strategy is not a simple trade-off of performance and resource usage issues, but may often be decided by application semantics.

6.2.5 MatchMaker to Trader mapping

The Trader to MatchMaker direction is the easier direction of the two, as the MatchMaker interfaces provide a near superset of the functionality offered by the Trader. It should be noted that the one to many mapping of interfaces, calls into doubt the concept of a gateway per interface; in many cases we may find a many to many mapping of interfaces, especially in larger systems.

Mapping in the other direction, MatchMaker to Trader (i.e. a MMTG), is only examined in brief here, and with particular reference to the problem of mapping from superset to subset. Full examination would be time-consuming, and it is not clear that many additional issues would be identified.

6.2.5.1 *Flexible structure of offers*

Some properties of offers in the Trader are treated as special; offers cannot be registered without them. These are interface type and interface reference. The MatchMaker stores offers as arbitrary collections of offers, and what is more allows individual properties of existing offers to be added or deleted at will, see the operations *AddProperties*, *DeleteProperties* on the interface *MatchMaker.ExportManager*.

It might be possible to use reserved dummy values (for example, *InterfaceSpecificationName* "*Unspecified*" or "*Uninvokable*") when passing offers without type and interface reference to the Trader, but care is needed, since clients of the Trader normally expect to receive interface references with imports.

Manipulation of individual properties could be achieved in a MMTG gateway by retrieving the existing offer, deleting it, removing or adding properties to the retrieved information and then re-registering it. This again is a clumsy solution which performs poorly.

6.2.5.2 *Control of link traversal*

NamingContext governs which parts of the ANSAware Trader namespace are searched, and this is the only mechanism by which ANSAware clients can control which links to other traders are followed. In ANSAware, traders may be federated by being bound into each others namespaces, or by posting of *ProxyOffer*s.

In the MatchMaker, traders are federated using explicit links or monitored offers, but in either case, *LinkSearchPolicy* can be used to control access.

Bi-directional mapping of the correct level of control can be achieved here by managing the contexts in which *ProxyOffer*s are posted in the ANSAware Trader so that they are consistent with the contexts of bound namespaces. However, in achieving this, other uses of the ANSAware Trader naming hierarchy are prevented: the context tree is essentially one-dimensional.

6.2.6 Multiple gateway instances

Problems could arise when the same two services are linked by more than one state-holding gateway; it is worth considering the issues in order to forestall such problems.

There are two cases to consider here; two TMMGs running in parallel, and a TMMG in parallel with a MMTG.

6.2.6.1 *TMMGs in parallel*

The problem here is the state held by the gateway - in this case the mapping between *OfferId* and (Interface reference and properties). There is no guarantee that:

1. A *Delete* invocation will pass through the same gateway instance as the corresponding *Register*.
2. There will only be one *Delete* call made for each *Register* call.

The first point requires that map state be shared between gateway instances, which implies that this state should be persistent, as it must outlive single gateway instances.

As to the second point: one or more *Delete* calls may be made through gateways once the offer has been deleted - how should such calls be handled? One approach would be to keep state about offers which have been deleted, so that such invocations could be forwarded to the server and hence the appropriate “offer not known” message produced. A performance optimisation would be to catch such invocations in the gateway, but great caution must be exercised: it should be possible to distinguish these calls from erroneous calls or errors in gateway software.

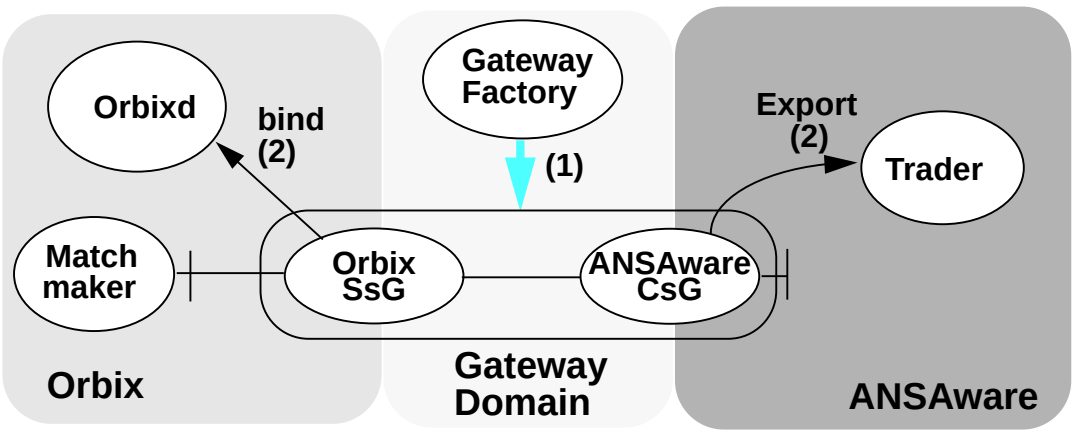
There is a garbage collection issue here, since information about offers needs to be deleted eventually; this includes cases where no *Delete* call is received.

6.3 Management and setup issues

6.3.1 Trader-MatchMaker gateway initialisation

The setup of the TMMG itself is much as with the Echo gateway.

Figure 6.1: Setting up the Trader-MatchMaker Gateway



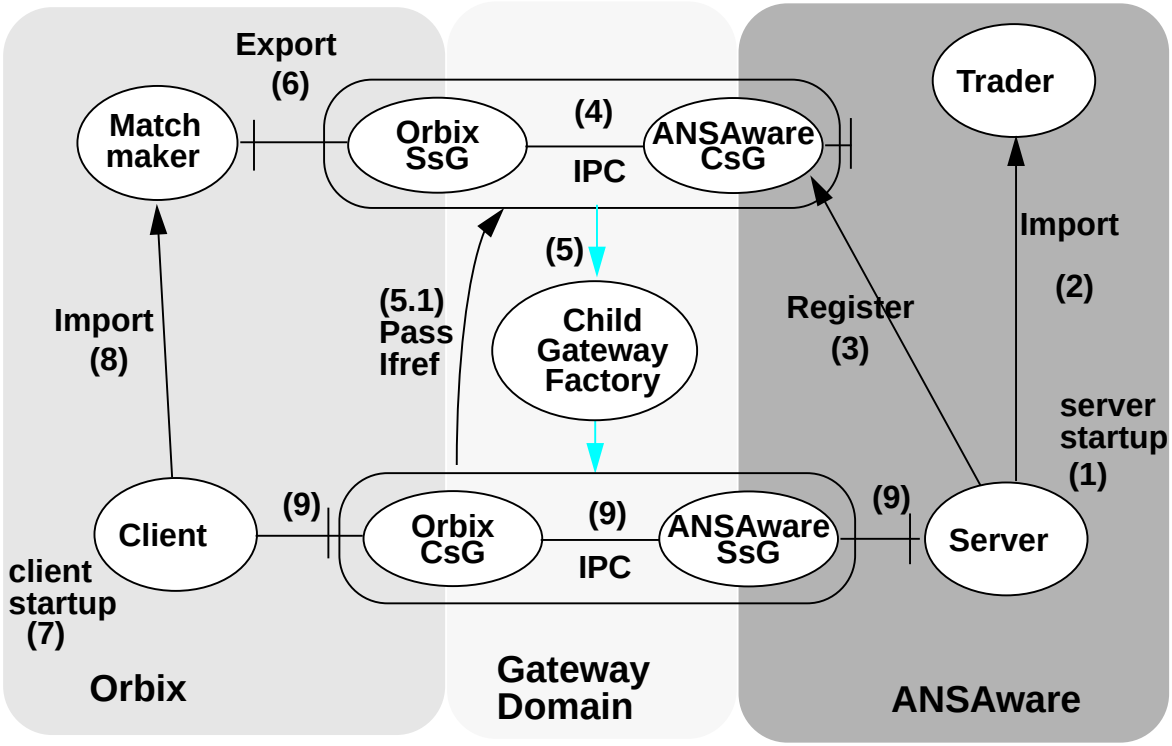
Procedure:

1. Manually invoke the gateway factory to create the TMMG. The Orbix SsG binds to the MatchMaker via the Orbix daemon; the ANSAware CsG Exports its interface to the Trader.

6.3.2 Trader-MatchMaker gateway usage

The ANSAware server may perform *Register*, *Lookup* and *Delete* operations through the gateway. Figure 6.2 shows an example scenario where an ANSAware server *Register*s itself with the MatchMaker via the gateway (3), causing a child gateway for the server to be automatically created (5: this is immediate resolution), and an Orbix client importing the reference of this gateway from the MatchMaker (8) and using the gateway to make invocations on the ANSAware server (9).

Figure 6.2: Using the Trader-MatchMaker Gateway



Procedure:

1. (1) Create AW Echo server. The following sequence of events occurs:
   - (2) Echo server Imports TMM gateway service.
   - (3) Echo server Registers its services to TMM Gateway.
   - (4) Invocation passes between simple gateways.
   - (5) TMM gateway invokes gateway Factory to create Echo gateway.
   - (5.1) Echo gateway passes its Orbix interface to TMM gateway.
   - (6) TMM gateway invokes Export on Match-Maker passing the Echo gateway interface.
2. (7) Create Orbix client. The following sequence of events occurs:
   - (8) Orbix Echo client imports Echo service and gets Echo gateway interface.
   - (9) Client invokes server through Echo gateway

Automated gateway creation introduces the problem of how to automate gateway deletion. This is an interesting problem, because deletion of the offer which caused the gateway to be created is not grounds for deletion of the gateway, which might well remain in use. One reasonable time for the gateway to be deleted is when the last active client discards the interface.

This is not the preferred architectural picture for trading in general. Ideally, clients and servers should not be directly exposed to traders other than their own (steps 2 & 3), and the traders should be federated directly.

7 Important Design Issues

This chapter discusses several important issues for gateway design. Note that ‘clients’ and ‘servers’ in the context of this discussion may themselves be gateways.

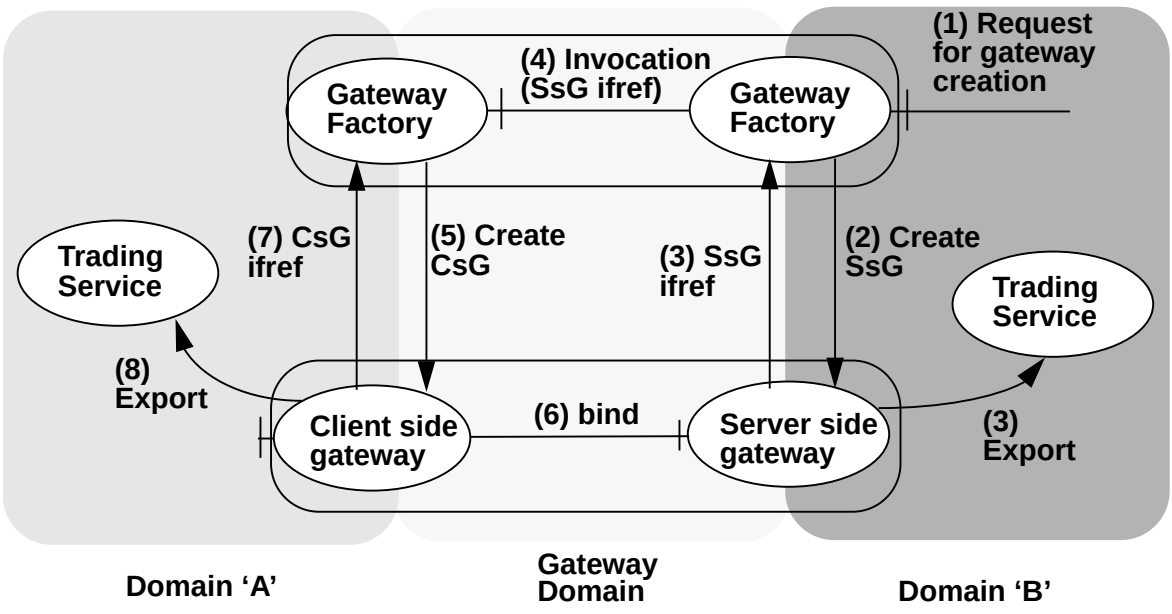
7.1 Gateway management

Before interaction between clients and servers can occur through gateways, these gateways must be put in place and managed. In terms of the ANSA interception model, this is resolution. Regardless of when the resolution is done or who requests it, there must be a way of managing this process so that it is transparent to both clients and servers.

7.1.1 Gateway creation

The general gateway creation process is depicted in figure 7.1.

Figure 7.1: Gateway Creation



The Gateway Factories would often be invoked by a parent gateway as a result of an invocation reference passing the domain boundary. One example of such a parent gateway is the Trader-MatchMaker gateway discussed earlier in this document (chapter 6).

The message numbering in figure 7.1 is exemplary only. The gateways can be created independently of each other; the comms resources which they use may be allocated at invoke time, gateway creation time, or earlier.

Each gateway may elect to export itself to the trading service in its own domain, as defined by policy.

Once both simple gateways and the necessary comms resources have been created, the simple gateways can bind to each other. They are then ready to be bound to clients and servers as required.

Note that a gateway may be created in order to link to a specific server or servers which already exist, and whose interface references are being passed to another domain. In this case, the factory may be passed the necessary interface references (1), so that these can be passed to the SsG (2) and a binding can be set up when the SsG initialises (2.1).

Alternatively, gateways may bind to servers dynamically using a trading service.

7.1.2 Gateway usage and destruction

Once gateways are in existence, more functionality is required to manage the gateways during their lifetimes, and then to delete them when they are no longer required.

Gateway usage issues include:

- Determining whether new clients and servers should be bound to existing gateway instances or new ones; put another way, managing the number of instances of each particular gateway type. This is a form of load-balancing, but issues of availability and reliability must be considered in addition to throughput and latency.
- Version control of gateways.
- Management of chains of linked gateways. Gateways may be composed from other gateways recursively; when operating on a single simple gateway, management functionality should consider the effect on any complex gateways which depend upon it.
- Identifying when gateway instances are no longer in use, or whose existence is no longer beneficial to performance, and removing them cleanly.

Prototyping activities have not addressed these issues in depth, so further discussion is outside the scope of this document.

7.2 Monitoring in gateways

There is often a need for monitoring in gateways. It may be required for debugging, traceability of the interactions between domains, load-balancing or other forms of management, or visualisation.

7.2.1 Monitoring of different projections

Fundamentally, we may be interested in any or all of the ODP viewpoints of the system [ISO/IEC 10476], yet all we necessarily have to work with at run-time is the engineering model.

It is sometimes possible to deduce interactions at the computational level from information about engineering entities, but this is less than ideal and is dependent upon the mapping between the engineering and computational projections. Usually, if we wish to monitor projections other than the engineering one, we have to provide extra information in the system, perhaps by annotating the software.

Initial prototypes concentrated on the computational model, since our initial goal was to visualise the behaviour of the gateway prototypes. Some consideration was given to other models, because gateways often have a meaning in several projections (e.g. enterprise).

7.2.2 Problems with monitoring in gateways

Monitoring distributed systems is not easy at the best of times, but it is particularly difficult in gateways.

The following problems are often encountered:

1. Gateways usually link two different infrastructures. Often, it will be difficult to relate the information supplied by one infrastructure to the information supplied by the other.
2. Infrastructures often support a range of transparencies, for example location transparency. Many infrastructures actually enforce these transparencies, so that it can be extremely difficult to obtain information which is required for application-level monitoring.
3. The two domains being linked may be configured to monitor different aspects of the system, to different levels of detail. In the extreme case, monitoring may be switched on in one domain, but not the other.

A detailed discussion of these issues is outside the scope of this document. Monitoring facilities in prototypes have concentrated on the computational model and have been restricted to simple gateways, without attempting to integrate facilities in the platforms being connected; in fact there are none provided in ANSAware 4.1 or Orbix 1.2.

7.3 Choice of gateway domain

Protocols, information models, interaction models and the like must be agreed for communication between gateways. There are advantages to considering these as forming another domain: it is a domain whose characteristics should be chosen to make gateways as efficient, reliable and maintainable as possible whilst preserving the autonomy of the members of the federation.

There are three issues to consider:

1. The nature of the agreement required: what the agreement will enable.
2. The nature of the gateway domain: what common forms to choose.
3. Use of industry standards.

7.3.1 The scope of the agreement

1. It may be an agreement to use one or more industry standards, or to use private models and protocols, or a mixture of both.

2. Interoperability may be required at different abstraction levels; for example, specific operations on low-level object interfaces, complex transactions between large systems or shared information models.
3. Interoperability may only be required at the level of normal operation invocations, in which case binding would be static and agreed a priori outside the systems. Alternatively, binding and trading services might also need to be federated, to allow free and flexible co-operation between systems. Some position in between these two may be preferred.
4. Management of the agreement, including issues such as measuring conformance to the agreement, changes to the agreement, withdrawal from the agreement and so on, may be handled entirely outside the system, or some of it may be automated; in either case, policies for managing the agreement should form part of the agreement itself.

7.3.2 The nature of the gateway domain

Characteristics of the gateway domain may often be dictated by economic or political constraints on the agreement, such as the use of standards or the need to satisfy specific requirements. In some cases, the gateway domain may not differ from one of the domains being federated, in which case the gateway linking this domain to the gateway domain may be null.

Where a choice can be made, the following issues should be considered:

1. The gateway domain must be capable of representing all of the information and interactions in the intersection of the domains being federated. This means in practice that the gateway domain will tend to be a superset of the others, in terms of any characteristics where there is a difference to be overcome.
2. The amount of processing required to transform information should be minimised. This can be facilitated by selecting a gateway domain which is as similar as possible to all of the domains being linked.
3. The likelihood of reuse is increased by selection of a more flexible gateway domain. For example, where there is a many to one mapping of operations between two domains, it is more likely that a gateway domain which represents the many operations will be mappable to a third domain than will a gateway domain which only represents the one. The representations in a gateway domain should act as 'lowest common denominators' for those in other domains.
4. It is beneficial to separate concerns as far as possible. Any distinctions which can be made between different properties of the domains being linked will promote reuse. For example, interaction mechanisms should be decoupled from interaction models, and application level issues should be separated from platform issues.
5. Given that the definition of a relationship may be organised according to the properties being transformed, different gateway domains may use different properties for describing relationships. For example, one gateway domain may represent RPC mechanisms and interaction models separately, whereas another may combine them. Again, it is better for the gateway domain to act as a 'lowest common denominator'.

7.3.3 Use of industry standards

In general, industry standards aim to be reusable agreements. If an industry standard is suitable for part or all of a gateway domain, it should be used. However, the issues described above make it difficult to agree standards; historically, IT standards have met with limited success, often being too volatile, poorly defined or subject to commercial politics.

Because of this, even when an international standard is used to achieve co-operation, it is usual for the design of the system to be kept independent of the definition of the standard as far as possible: effectively, the system is using a gateway to map between its own internal representation and the standard one.

Whilst standardisation bodies offer a top-down approach to reuse of agreements, it is also possible to take a bottom up approach, attempting to reuse existing gateway domains, and gradually enhancing them to apply more widely. In the long term, this may offer the best route to stable industry standards, through ad-hoc evolution from local standards.

7.4 Internal design of gateways

This section discusses a range of issues for the internal design of gateways. Other, related issues are discussed in [APM.1514 95].

7.4.1 Statefullness of gateways

The question of whether a gateway will hold state is very important, as shown in the second case study. Application specific gateways may be forced to hold state because of the nature of the mapping required. Alternatively, it may be deemed desirable to include other functionality such as load-balancing which requires state describing server characteristics.

This question is important, and other decisions will become simplified once it is answered. The worst trap is to decide upon statelessness and then have the decision overturned after detailed analysis of the application mapping.

7.4.2 Unidirectional or bidirectional gateways

Two or more simple gateways can be combined to provide a link between two domains; this link may allow operations to be invoked in both directions, or only one. In architectural terms the most simple unit is a unidirectional simple gateway, and a bidirectional link between two domains can be composed of four or more of these. However, where bidirectional links are needed, a design must consider whether to build unidirectional or bidirectional simple gateways.

In general, bidirectional gateways allow some optimisations, such as batching of transformations and invocations or load balancing economies through resource sharing, at the cost of higher complexity. Where the service provided in the client side domain is of a type which has no direct equivalent in the server side domain(s), multiple services will be needed to implement the service and a bidirectional gateway may become very complex.

7.4.3 Scope of gateway types

In general, a type of gateway makes one or more services available from one or more domains to one or more domains.

There are a range of options:

1. A gateway per client domain: one gateway which provides interfaces to all the service types required by objects in a particular domain, no matter what domains these service types are offered by.
2. A gateway per server domain: one gateway provides interfaces to all the service types required from objects in a particular domain.
3. A gateway per service type: one gateway provides an interface for a service type, which allows access to all servers which can provide it, no matter what domain they reside in.
4. A gateway per service type per server domain: one gateway provides an interface to the servers of a service type in a specific domain.
5. A gateway per service type per client domain: one gateway provides an interface for the clients of one service type in a specific domain.
6. A gateway per service type per server domain per client domain: one gateway provides an interface between two specific domains for one service type only.

As illustrated by the second case study (§6.2.5), the notion of a gateway per service type becomes very weak for application specific gateways, where there is not a one to one correspondence of types in the domains being connected.

There is a further scoping issue concerned with the nature of domains - since each domain is an area where a set of properties holds, a gateway between two domains may resolve differences in terms of several different properties. In architectural terms we can envisage a gateway per property, with option to combine these at design time. Consequently, each of the above options may be further specialised with the following options:

1. One gateway for all properties.
2. One gateway for each property.
3. Some position in between these two.

As with bidirectionality, there is a trade-off of optimisation against increased complexity.

There are also reuse issues, since gateways composed from simple per-property gateways are likely to be more modular and therefore more reusable. However, a well-designed multiple-property gateway should be constructed from software which is equally modular, so only run-time re-usability is necessarily compromised by building multiple-property gateways, and this is rarely important.

Management is also easier with fewer gateway instances.

7.4.4 Scope of gateway instances

No matter what the scope of a gateway type, there are also a range of options for parallelism in instances. These are:

1. Multiple simultaneous clients and servers for each gateway instance.
2. Multiple simultaneous clients and one server for each gateway instance.

3. One client and multiple simultaneous servers for each gateway instance.
4. One client and one server at a time for each gateway instance.

Support for multiple simultaneous clients implies that some state about server invocations is held, either in the invocation messages to servers or in gateways, to relate these invocations to clients, so that a response from a server can be returned to the correct client.

Support for multiple simultaneous servers introduces a more subtle problem, and a potential trap. The critical question here is whether the server is stateful or not. If the server is stateless, then invocations from clients may be multiplexed across available servers freely, on the basis of server availability.

However, stateful servers imply that invocations from a particular client must always be directed to the same server, even though both client and server may migrate, and the client may invoke different gateway instances at different times (depending upon the scope of the gateway type). Gateways may need to maintain shared persistent state in these circumstances.

It may seem that this problem should not be encountered, since stateful servers must be accessed in this controlled way even where no gateways are present, so the servers themselves should handle multiplexing. However, servers may assume that they are the only provider of a particular service: an assumption which may hold good within the servers own domain, but be invalidated by federation as gateways allow access to similar servers from other domains.

For example, if there is only one printer on a network, a series of “send-a-page-to-the-printer” invocations will produce the desired result. But if further printers become available via gateways, this series of invocations might cause a document to be scattered across several printers, unless the appropriate steps are taken.

7.5 Auto-generation of gateways

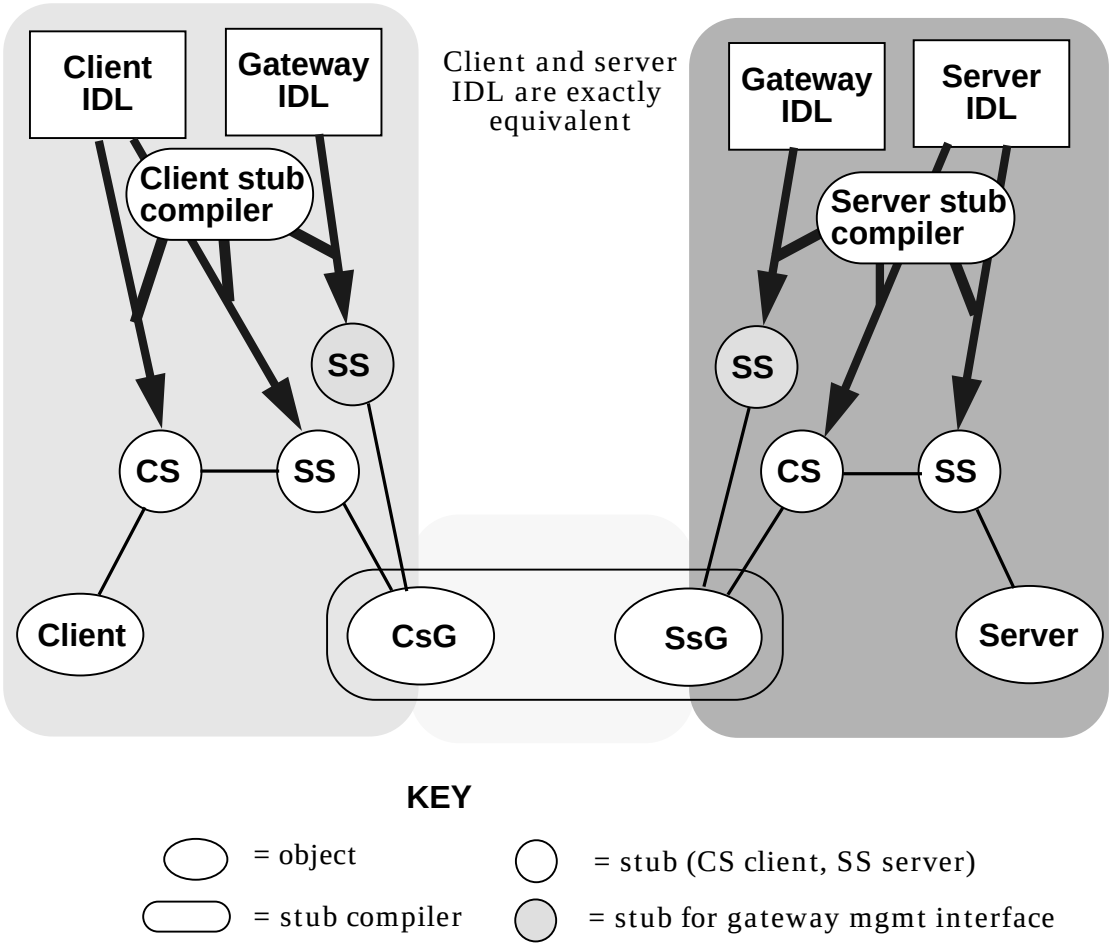
There are tools for most distributed processing environments, which enable stub generation from IDL (or similar) specifications. These stubs are often primarily aimed at location transparency, but they may include a wide range of functionality.

This kind of stub generation technology can be used to generate gateways.

7.5.1 Generating platform-specific, application-independent gateways

These are gateways which make services available in new environments, but do not do any application specific mappings - in other words, where the IDL is equivalent on both sides. These gateways can be generated with existing technology as shown in figure 7.2.

Figure 7.2: Auto generation of application independent gateways



Gateway management interfaces should be similarly treated; these need not be equivalent on each side, though discrepancies will need to be rationalised in order to manage complex gateways. Stubs may also be generated for use between gateways - these are not shown.

Also, a single stub compiler can be used to generate stubs for a range of environments; this allows auto-generation to be better managed [APM 1524 95].

7.5.2 Generating application-specific gateways

These gateways may be built by inserting functionality into application independent gateways. This functionality could conceivably be auto-generated from a specification of the application-level mappings, but little work has been done to identify an appropriate general method of describing such mappings, and so there is little likelihood of full auto-generation in the near future.

However, simple mappings may be described without great difficulty, and these may suffice for a significant proportion of cases. It may be possible to build up a library of translations between different types which can be reused for new operations.

Differences can be described using data structures to model the relationships between operations, parameters and terminations. This removes the need for cumbersome case statements and makes gateway code reusable for a given

pair of domains, though each new gateway would require new functions to be written, pointers to which could be held in the data structures and invoked by the general code according to data placed in the data structures by a specific initialisation routine.

Insertion of general functionality such as security or monitoring may be achieved by developing the appropriate libraries and inserting calls to these into the generated stubs

7.5.3 Type-independent platform-specific gateways

Gateways can be developed using dynamic interfaces such as the CORBA DSI and DII interfaces, which effectively offer application independent per-server-domain gateways which can process any interface type.

However, some application-specific functionality is often required (e.g. we may wish to selectively monitor a few key interfaces). These gateways should be considered in the light of the compromises discussed in §7.4.3 *Scope of gateway types*; in particular, significant application-specific functionality may produce a maintainability problem.

8 Summary

The prototyping work described here has been very useful, and the initial high level design has, so far as it goes, stood up well to the problems tackled. A number of key points have emerged:

1. The approach of composing complex gateways from simple ones as introduced in §3 is very important, and provides a wide range of benefits.
2. There are a wide range of options for scope and parallelism; these decisions are hard to generalise about (§7.4.2, §7.4.3, §7.4.4).
3. The issue of whether a gateway will hold state is critical (§7.4.1).
4. Definition of the gateway domain is an important task, and more work should be done on guidelines for this.
5. Gateway domains should be chosen using the 'lowest common denominator' approach to improve reusability (§7.3, §7.4.3).
6. Application independent gateways are good candidates for automation, and the technology to do this is widely available already, in the form of stub generation and DII/DSI (§7.5).
7. Gateways can be developed according to a standard design, and many components are independent of the nature of the differences being overcome. A common gateway skeleton which could be configured to solve particular problems would be a good investment.
8. Application specific gateways may involve very difficult problems, and can be much more difficult than gateways between platforms. It would be useful to produce some guidelines for estimating the effort needed to produce an application gateway for a given degree of difference.
9. It is possible that study of more examples of application gateways would reveal common patterns, and it may transpire that a relatively small set of tools would be sufficient to tackle the vast majority of application mappings.
10. The choice between developing a gateway or porting an application may be very difficult. If further links are likely, a gateway is probably the best option.
11. The addition of even a single extra operation have a very great impact on the design of the gateway: a mapping is less maintainable than a single model. It is therefore especially important that the link required is fully specified before development work begins.
12. Prototyping work suggests that management of gateways needs to be integrated at a low level with other platform components such as binders, relocators and traders; this may have significant impact on the architecture; see [APM.1514 95].

References

[ANSA 91]

ANSA, “A Systems Designer’s Introduction to the Architecture”, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD U.K., April 1991.

[APM.1514 95]

Yigal Hoffner & Ben Crawford, “Federation and Interoperability”, APM 1514, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD U.K., 1995.

[APM.1507 95]

Yigal Hoffner, “Interoperability and Distributed Platform Design”, APM 1507, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD U.K., 1995.

[APM ANSAware 93]

ANSAware Version 4.1 Manual Set, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD U.K., 1993.

[APM 1384 94]

Gray Girling, Mike Beasley, “The Property Repository”, APM 1384, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD U.K., 1994.

[APM 1524 95]

Youcef Laribi, “Stub compiler architecture and design”, APM 1524, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD U.K., 1994. In progress.

[IONA 94]

Orbix Programmer’s Guide and Advanced Programmer’s Guide, version 1.2 February 1994, IONA Technologies, 8-34 Percy Place, Dublin, Ireland.

[OMG 94.9.32]

“Universal Networked Objects”, OMG RFP Submission, OMG TC Document 94.9.32, Object Management Group, 1994.

[ISO/IEC 10476]

“Open Distributed Processing Reference Model”, ISO/IEC 10476, ITU-T Reccomendation X.900 (1995) Parts 1, 2, 3.