Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

An Overview of ANSA (for IEEE Network)

Andrew Herbert

Abstract

This is an article for IEEE Computer which explains the rationale for ANSA; the principles that underpin the architecture and uses a simple telecommunications oriented scenario to explore the architectural framework.

It is an amalgam of APM.1000 and the author's 1993 TINA Conference Keynote address..

APM.1093.01 **Approved** 25 November 1993
External Paper

Distribution:
Supersedes:
Superseded by:

An Overview of ANSA (for IEEE Network)

External Paper

An Overview of ANSA (for IEEE Network)

Andrew Herbert

APM.1093.01

25 November 1993

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	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

1.1 The ANSA objective

ANSA is an architecture which enables programmed telecommunications service and computer application components to work together despite diversity of programming languages, operating systems, computer hardware, networks, communications protocols and management and security policies. The Architecture is relevant to telecommunications, manufacturing, sales, cooperative working, banking health service, research, and other applications. It provides a framework for the design and implementation of distributed computer systems supporting networked information services. The framework ensures that different design choices which are made for particular applications can be related to one another.

1.2 Distributed systems

The various software systems used by an organization, be it a telecommunications operator or service provider, a factory or an office, collectively function as a distributed system whether or not they are designed as such, or managed as such. Ideally a software architecture should facilitate the coordination and evolution of software systems which support business processes in the organization.

The computer industry has developed three different approaches to meeting this goal:

- distributed databases and remote database access
- robust messaging and transaction routing
- networked client - server personal computers.

Each approach is based on distributed computing, but has quite different techniques and properties.

The database approach is oriented towards large stable bodies of data accessed through a query language such as SQL. Databases optimise the storage of data and maintain its integrity. This is easy to do because the data is all in one place, which is also a limitation because of the implied bottleneck and fixed structure.

The robust messaging approach is oriented towards reliable flow of transactions through a sequence of processing steps and often embodies powerful facilities for error recovery and security. The application program is expected to cooperate closely with the queuing and transaction functions, making it difficult to port the application to other systems.

The networked PC approach is oriented towards empowering users to organize data an applications to suit their own convenience. It brings much greater

autonomy and flexibility, but without the safety net intrinsic to the more controlled messaging and database regimes.

Each approach meets a different need. No single one of them will absorb the others. Instead users are faced with having to interoperate between them and achieve federations of legacy application islands.

1.3 The software burden

Currently interoperability is piecemeal, both in terms of technology and standards. For example, there are standards for accessing databases from PCs, but mostly without transaction support. The software burden of application and service integration unacceptably high.

A more comprehensive software architecture is needed for interoperability. The requirements upon this architecture are:

- Allow many different kinds of applications and services to coexist and interwork (including crossing traditional boundaries between office, factory, telecommunications and data processing)
- Scale from small (a few nodes) to very large (e.g. the telephone network)
- Allow applications and services to span organizational boundaries, whilst respecting the policies and constraints of those organizations.

The benefits that the architecture must deliver are:

- Improved time-to-market for services - rapid design, coding, testing and deployment
- Reduced costs - avoiding re-implementation, building on existing systems, and ensuring that the new legacy is more open and extensible than its precursors
- Increased the scope for systems interoperability and software portability - giving the user freedom to choose appropriate technology, to mix new and old technologies in a system, and to couple technologies together.

These criteria point to the need for simple system structuring techniques and a high degree of automation in production of system software as foundations of a successful architecture for software service management across federated, networked computer islands.

1.4 Building consensus

To enable the design and construction of systems with the above characteristics, consensus between the computing and telecommunications industries is desirable. The ANSA work programme has provided a focus for such consensus by being directly supported by more than 20 companies in these industries.[1]

Computer related standards communities often work on standards before products are available. This inevitably means that expertise in the area is less well founded than elsewhere. The ANSA Architecture and its prototypes help promote the confidence that standards for distributed computing, once implemented, will lead to systems with desirable characteristics.

Whilst it is not a standard in its own right, ANSA is being developed in close association with a number of standards-making bodies including the Open

Software Foundation, Unix International, the Object Management Group, ISO and the ITU-T (formerly CCITT). ANSA has been a significant input to the joint ISO/IEC and ITU-T development of a Reference Model for Open Distributed Processing [ISO 93a], [ISO 93b]. At the level of detail in this paper, the Reference Model for Open Distributed Processing and ANSA are identical. ANSA is a cornerstone of the work of the TINA Consortium [BARR 93]

Technology based on ANSA principles has been submitted by sponsors of the ANSA work programme to OMG, OSF, and UI; aspects of their specifications can be traced back to ANSA.

1.5 Is ANSA real?

It is not possible to offer proof of concept through the pages of any paper. Many of the ideas expressed in ANSA have been taken to a fine level of design and implementation [ANSA 91]. Prototypes are continually being developed to illustrate the practical application of the principles which underpin the Architecture. Several large scale distributed systems have been built using ANSA technology including:

- a data retrieval and remote processing overlay for the Internet called the NASA Astrophysics Data System [MURRAY 92]
- a management and monitoring overlay for an OSI MAP protocol-based manufacturing system
- a document transfer system for a commercial PC-based hospital management system.

ANSA is the basis of two products produced by ANSA sponsors - ICL's DAIS distributed applications platform, and a PC-based video conferencing product soon to be launched by GPT.

The technology and concepts are also used in many universities as a basis for both research and teaching distributed computer system fundamentals.

1. ANSA has been developed in three phases. The first phase was concerned with scoping the architecture and was supported under the UK Alvey Programme and involved 12 UK and US companies. The second phase was concerned with development and demonstration of the architecture. The work was supported by the ESPRIT programme and involved over 20 European and US companies. The third phase started in early 1993 as a fully industrially based activity to extend the architecture in the direction of distributed service management. The programme currently has 10 major sponsors.

2 The architecture

2.1 Organization of ANSA

The ANSA Architecture is not a design for a distributed computing product, in the same way that the gothic architecture is not the same as the Notre Dame. Instead, ANSA specifies:

1. a set of components, which form the basic building blocks and tools of the architecture;
2. a set of rules, which constrain the way in which such components can be combined in designs that conform to the architecture;
3. a set of recipes, which provide advice on how to combine basic components using the tools to obtain subsystems with certain properties;
4. a set of guidelines, which help designers make design decisions if their own preferences do not offer sufficient basis for such decisions.

Using the components, rules, recipes and guidelines (i.e. the Architecture) helps ensure that the differences between the resulting systems remain such that applications can be ported and their interoperation can be achieved more easily.

The Architecture allows many different designs. Each may be suited for a particular application or application area. For each area different optimisations will be appropriate, and individual designs must reflect these. The Architecture can be used as a framework to compare the different design optimisations and to help designers reason about combining dissimilar systems.

2.2 Projection models

The only way to tackle the complexities that result from the requirements expressed in §1.3 is to provide for separation of concerns within the architectural framework.

ANSA uses five complementary **projection models**[1] to describe architectural components:

1. an *enterprise model* for expressing system boundaries, policies and purpose;
2. an *information model* for expressing the meaning of distributed information and information processing tasks;

1. called “viewpoint languages” in the ISO Reference Model for Open Distributed Processing. (The ANSA term comes from the mathematical concept of projection - giving an image of the total structure when viewed from a particular angle.)

3. a *computational model* for expressing functional decomposition of systems into distributable units with well-defined interfaces;
4. an *engineering model* for describing components and structures needed in support of distribution;
5. a *technology model* for describing the make up of a system in terms of components which conform to appropriate standards.

Each of these models is object-based. They differ in the way objects are used, and in terms of what is being modelled. For example, in information models, relationships have prominence; in computational models interfaces are key; in engineering models the focus is on structure.[1]

A system specified using ANSA consists of statements made using some of all of the models. A correct implementation of such a specification is one which satisfies all the statements made about it. A specification containing statements drawn from all the models is likely to leave less implementation freedom than one using fewer of them.

2.3 Functional models

Within the framework of projection models, specific models of a more functional in nature (including an Atomicity Model, a Naming Model, a Model for Groups, and a Storage Model) lay down the definition and usage of architectural components such a traders (see §3.4).

Most other “architectures” known in the computer and telecommunications industries have a fixed structure similar to a systems design. Such designs can often be represented by a block diagram. By way of contrast, ANSA is characterised by the properties of its components and the nature of the composition rules. These cannot be expressed in a simple diagram.

The rules and recipes of the architecture describe the various ways in which components can be combined into systems, the ways in which different systems conforming to the architecture can be made to interoperate and how software can be ported between them.

1. In the ISO Basic Reference Model of Open Distributed Processing, the common properties are the models are collected into Part 2 of the standard, called the “Descriptive Model”. The five models are defined in Part 3 of the Reference Model, which is called the “Prescriptive Model”.

3 Using ANSA

Imagine a municipal authority which wants to encourage new industry to come into a development zone. To facilitate developers, the authority provides electronic access to its Planning and Land Registry units. Developers can submit plans for approval electronically; they can obtain maps and images of the development zone electronically. An architect's practice augments these facilities by linking them to its own Computer Aided Design tools so that the architects can show clients drawings of new building projected onto images of the terrain.

This scenario illustrates several aspects of the rationale for ANSA - the time-to-market for the system is short requiring rapid development and deployment: the system will have a short life, requiring development costs to be kept low. It involves integration of several different technologies (databases, messaging, graphics) and depends upon telecommunications.

Immediately two questions arise:

- How do we analyse the scenario to identify potential distributed services?
- How do we structure the service designs to be sure that
  - all design options are considered?
  - all design constraints are satisfied?
  - the design can evolve to accommodate new requirements and adapt to new technology?

These questions can be answered by using the ANSA projections.

3.1 Policy issues

Policy issues are the domain of the enterprise model. Systems are characterized as federations of interacting agents consuming resources and generating information. The enterprise model of a system is obtained by asking "**who** uses **which** information and for **what** purpose?

Behaviour in the enterprise model is explored from the point of view of **obligations** and **liabilities**. For example in the scenario

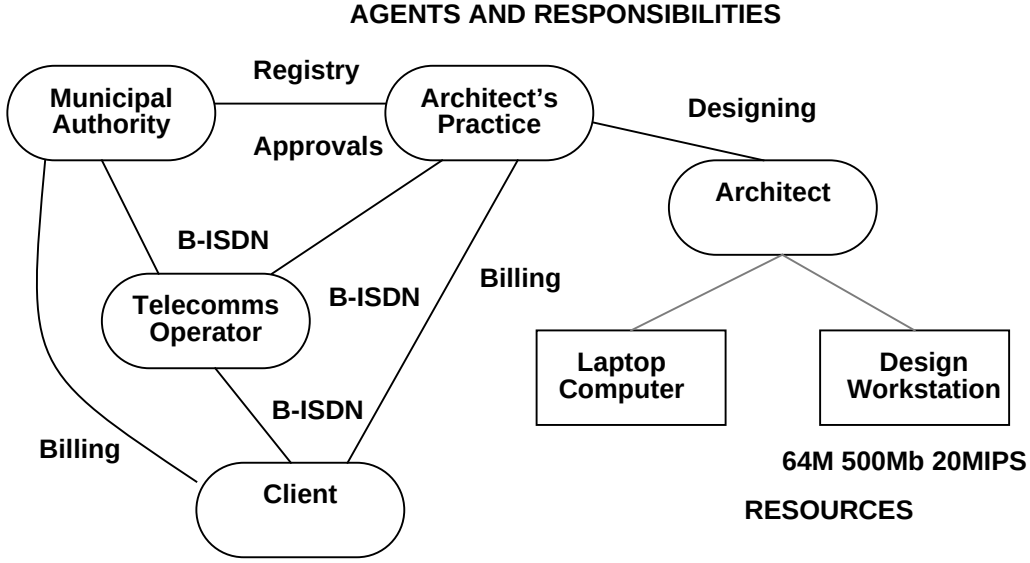
- which information sensitive?
- what is the accounting / charging model?
- what are the requirements on ease of installation, access, user training, maintenance?
- what regulatory requirements must be respected?

The issues can be decided can applying techniques of **Responsibility Analysis**.

Discussion of resources provides a basis for exploring the dynamics of the system in terms of throughput and storage requirements. Used as input to a

simulation tool, an enterprise model of a system is a vehicle for reviewing alternative system structures. A fragment of an enterprise model for the scenario is illustrated in Figure 3.1

Figure 3.1: Example enterprise model



Obligation and resource are tied together by the concepts of **value**, **negotiation** and **sanction** which in turn depend upon the notion of **boundary** to delineate authority. The introduction of boundaries brings an object-based flavour to the analysis, congruent with the use of objects in the other projection languages.

Taken together the enterprise model concepts - there are few rules, but many guidelines relating to preserving autonomy - provide a powerful basis for modelling the business processes to be supported by the computer system

Including policy issues in a design discussion forces the designer to recognise that the systems users and the system itself are all participants in negotiating an end-to-end **contract** to provide and use services.

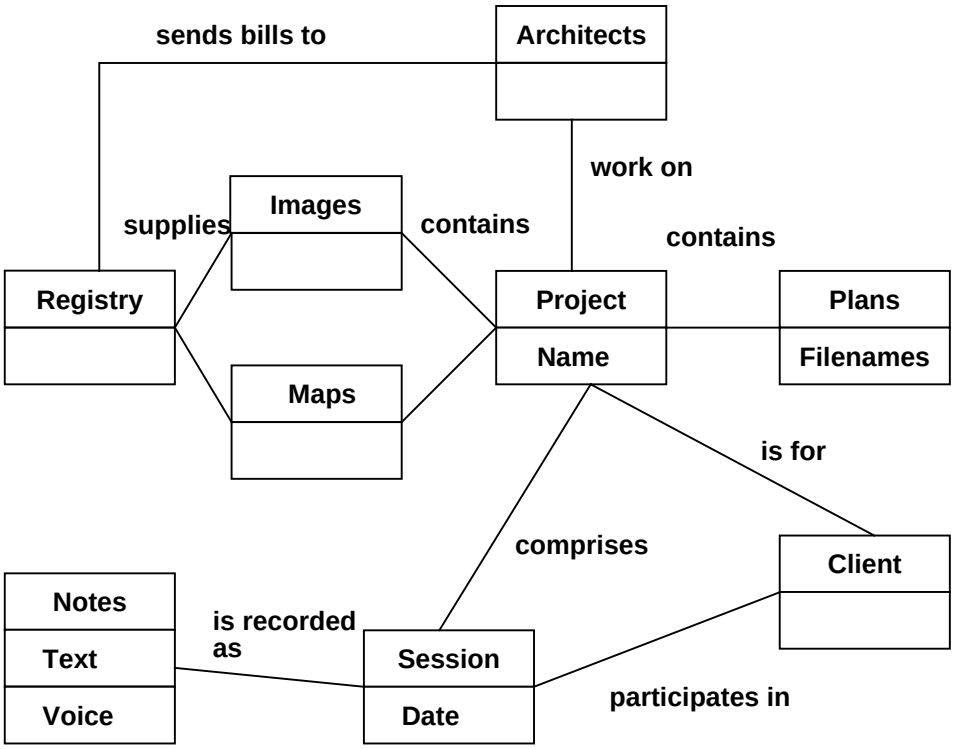
3.2 Information issues

In the scenario several kinds of information were exchanged between the parties, including maps, plans and images. For this information to be meaningful, and the dialogues successful there must be a consistent interpretation of both data and processes.

The information model provides the basic tools of objects and relations to do this job. Very little more needs to be said since the techniques of object-oriented information modelling are well known (for example see [RUMBAUGH 91]). Illustrative fragments of an information model for the scenario are shown in Figure 3.2.

More care is needed in modelling distributed systems because of the problems of inconsistency and convergence explained in§4.1. The information model of a system will often look for generalizations that can be made so that changes in

Figure 3.2: Example information model



the enterprise model are successfully anticipated and planned for the technical system.

3.3 Functional distribution

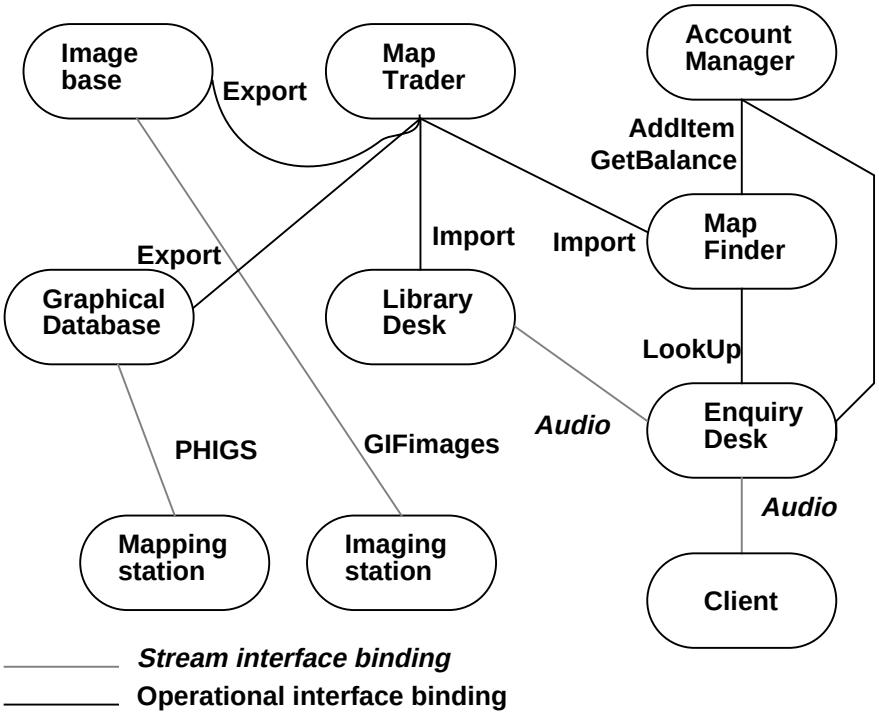
The ANSA computational model imposes strong constraints upon the structure of systems so that they can be distributed transparently. These constraints reflect the principles presented in Chapter 4.

The core of the computational model is the use of **objects** as units of management and replacement. An object has one or more **interfaces** which are the points of provision and use of service. Interfaces are of two kinds: **operational interfaces** which contain a set of named operations (i.e. procedures, or methods) and **stream interfaces** in which communication is structured as a set of linked directional information flows. A computational model of part of the image and CAD subsystems in the scenario are illustrated in Figure 3.3. Operational interfaces are shown by bold lines, stream interfaces by dotted lines.

Operational interfaces have the usual remote procedure call style of interaction - operations are invoked with a set of arguments and a response is returned. The response consists of a named outcome and a set of results. This model generalizes results and exceptions as found in many programming languages and recognizes the symmetry inherent in network communication.

Arguments and results to invocations consists of references to other interfaces. For example the Map Finder will return a reference to a map held in the

Figure 3.3: Example computational model



Graphical Database. This interface will contain operations to interrogate the map and extract data from it.

The effect of an interaction is that the client and server “share” access to the argument and result interfaces. This forces the designer to recognize the intrinsic concurrency in a distributed system and construct a server that manages concurrency to control consistency, without compromising overall throughput. This model makes each interface an **abstract data type**.[1]

Stream interfaces are used to describe unstructured communication - for example video and voice streams in multi-media systems or other forms of communication such as electronic mail being managed by an ANSA system.

Streams are an obvious motivation for objects that have multiple interfaces - it enables objects such as filters and transducers to be described. In the case of operational interfaces, each interface typically denotes a context in which the object can be used. In the scenario it may be that Graphical Database creates a new interface as the result of each query. The interface contains operations for interrogating the map identified in the initial query. Separate queries will result in separate interfaces. This facility is very powerful because it saves the

1. It might be argued this model is inefficient and that some form of copying semantics are needed in addition. There are two ways in which copying can be applied without extending the model. One is for the programmer to instruct an object to migrate from one node to another. Alternatively, the object can be replicated so that a copy exists at both client and server, with a coherence protocol between the two. Appropriate mechanisms for these are defined in the engineering model. Note that in the case of immutable objects (e.g. simple data types like “Integer”) the replicas do not require a coherence protocol - they can be replicated with no overhead.

programmer from having to name and manage contexts. It provides a foundation for security and accounting.

The interface binding rules in the computational language require that interfaces be type checked before bindings are established. The type checking rules impose **subtyping**.

In the case of operational interfaces, the server is expected to provide at least the operations required by the client, the client is expected to accept at least the set of outcomes generated by the client. The arguments supplied by the client can be super (i.e. simpler) types than those required by the server; vice versa for results provided by the server. These rules have two properties:

- safety - the client is guaranteed at bind time that the server will not respond at invocation time with a “method not supported” error
- evolution - the server interface can be extended to be a richer subtype without requiring any change to client programs.

3.4 Trading for services

Key to the completeness of the computational model are the notions of trading and factory services. Trading services provide the means to locate instances of a service; factories create new instances. Factories are found by trading.

The availability of service is advertised by an **exporter**. The exporter nominates a trader service and describes the exported service in terms of a service specification and a reference to the interface that provides it. An **importer** makes a request to a trader describing the service required and the trader selects a matching export.

The service specification includes

- the interface signature (i.e. function prototypes for the operations available at the interface), sometimes called the **interface type**)
- an abstraction of the behaviour of the operations
- additional properties, such as location, ownership, quality of delivery and management constraints on use of the service.

An example contract might appear as follows (the syntax is purely illustrative)

```
BankAccount:
     operations {
     credit (amount: money)
                         -> (newbalance: money)
          precond amount > 0
          postcond newbalance >= amount
     debit (amount: money)
                         -> (newbalance: money)
                         -> overdrawn ()
          precond amount > 0
          postcond newbalance >= 0
     balance () -> (balance: money)}
          postcond balance >= 0
     environment {
               authentication using kerberos
               auditing on
               use dce rpc}
```

The signature information is used to ensure bindings are type safe. The behaviour information, when available, enables service semantics to be agreed. The additional properties determine the infrastructure required between service user and service provider.

A **trader** differs from a **broker** in that a trader performs a directory and selection function. It does not take part in interactions between client and server. By contrast a broker (e.g. as defined by the OMG Object Management Architecture) takes responsibility for delivering requests and replies between client and server. This presumes trading has taken place at an earlier stage to establish which server to use.

For trading to be possible service specifications must be visible in the network. The trader cooperates closely with type managers which act as repositories for service specifications, and which can check subtype relationships between specifications. Making service specifications visible enables incremental, dynamic integration and evolution for systems.

Factories, alongside the architectural support for portability, interconnection and migration make **service implementations** available from the network. This enables re-use of services between applications and therefore reduces system complexity.

3.5 Engineering issues

Engineering distributed systems is about making trade-offs. For example increasing abstraction hides detail and simplifies systems, but fine-grained control is lost. Services can be made more available by replicating them, but then the risk of inconsistency arises. Consistency can be obtained by keeping a single copy, but then availability is compromised. Autonomy gives more freedom in how systems are managed and used, but lead to complexity; uniformity reduces complexity, but is an imposition. Security conflicts with convenience.

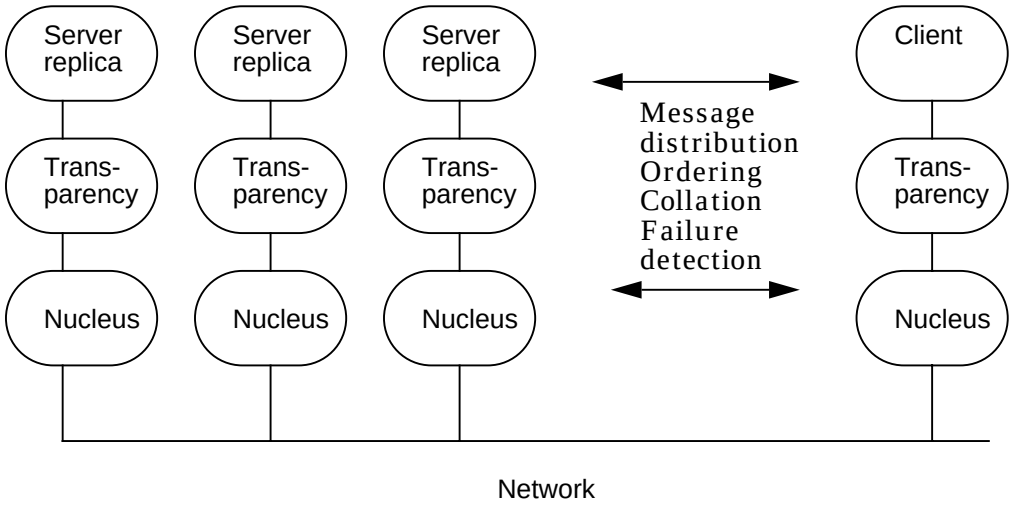
There are many such trade-offs.[1] The key point is that there is no unique answer, and therefore the goal of a ubiquitous distributed computing environment is ill-founded. Instead ANSA proposes a simple nucleus which has very basic facilities. Above this nucleus are layered “transparencies” that supply the additional guarantees. This is illustrated in Figure 3.4 which shows replication functions being used transparently. As far as the client and server objects are concerned the transparency objects provide a proxy for the remote client or server, where the proxy hides network communication and any other functions required to support more demanding transparencies.

In terms of the OMG CORBA specification the nucleus can be positioned as a basic Object Request Broker, the transparencies as higher level stubs and object adaptors that add extra characteristics to the Object Request Broker.

There is an interesting split in the kinds of functions required to support transparency. Some, such as object location, binding and authentication are usually independent of the individual requests made to a service - they establish end-to-end guarantees that persist as long as a binding is in place.

1. This particular set were presented by Andrew Birrell of DEC SRC at the 1992 ACM SIGOPS European Workshop at Le Mont Michel, France.

Figure 3.4: Transparency and the engineering model



Since they are generic, these functions can be completely transparent and simply configured as extra elements in the end-to-end communication path.

Others, such as replication, transaction and migration are interface specific - they require knowledge of object state and which operation is being invoked at any instant. These can only be made transparent if they (or at least the application-specific parts) are compiled into programs. In effect the program is transformed by the compiler to provide a form of transparency. Transformation is an important tool for providing abstraction to the programmer and automation of the production of system code.

Techniques for embedding application code into wrappers that specify the required distribution characteristics and then generating the required transparency have been an important part of the ANSA work. It has been used for:

- transactions and concurrency control
- object replication
- object migration
- object persistence.

4 ANSA Principles

The design of ANSA is governed by four dominant principles:

- define **services** in terms of **function**, **quality of delivery** and **ownership**
- **reverse** centralized, single processor, single manager **assumptions** to exploit distribution and federation
- **separate concerns** using the projections to gain clarity
- **hide irrelevant detail** to reduce complexity, **expose necessary detail** to obtain control.

4.1 Separation

Most systems are designed to work within a fixed context, either within a single computer, or in some static relationship with other systems. Major difficulties are often encountered when such systems are required to cooperate with systems outside this context. This is because built in assumptions and optimisations prevent the extensions required for effective interworking with other systems. Flexibility brings greater complexity and management overheads. This can be kept under control by using tools [WILDE 93].

It should be assumed from the outset that all components are physically or logically remote from one another. Careful choice of local optimizations guarantees interworking remains possible. Separation and remoteness produce a requirement for distribution transparent access.

It then follows that since local data is viewed as the co-located special case of remote data, all manipulation of data (as seen by the programmer) must be indirect and an object can only be requested (not instructed) to change the data it holds. There is an implied contract between service user and service provider. The parameters of interactions between user and provider are references to object interfaces.

It is not possible to know the reliability of a remote system, nor of the communication system to it. Any request might fail. Systems architecture must tolerate all types of failure, including partial ones. Mechanisms may be provided to hide and possibly correct failures from the application programmer (see: §4.4).

Physical separation implies that there can be no common assumptions about time. It is often cheaper to determine a logical ordering of events than to relate then to real time. Simultaneity or particular ordering of events cannot be assumed, it has to be determined explicitly. It must not be assumed that some inherent system characteristic will cause processes to synchronise.

4.2 Diversity and scale

Large distributed systems will include many significantly different individual systems, connected using many different communications systems and protocols.Differences in data representation, however problematic, must be accepted. Differences in performance and quality of service cannot be made uniform without going to a lowest common denominator, so they must be accommodated. Gateways (**interceptors**) will exist at boundaries between technology and administrative domains. Within each domain local properties can be exploited - between domains properties must be matched or mapped to ensure interworking.

Do not assume the existence of a universal data pool that can be accessed directly from everywhere. Integration of databases at the conceptual schema level imposes constraints of scale, since high levels of throughput, availability and consistency cannot be accommodated at once. Inconsistencies cannot be avoided and must therefore be accommodated. Local data, however, can always be treated as consistent, unambiguous and definitive within itself as it is under local control.

Systems will always change, grow, and merge. This introduces differences in scale: geographical, numerical, time etc. Architectural components must be efficient in small systems whilst sufficiently functional to suit very large configurations. Where possible components should be able to interwork with ones outside the architecture, even if the non-architectural components do not interwork themselves.

4.3 Federation

Federation deals with the demarcation of authority in distributed systems. It is important to retain (local) control of local resources. Allowing other authorities to control ones resources implies risk. Organisational structures which surround large distributed systems are subject to continuous and unpredictable change: no merger was ever pre-planned. Architectural components concerned with naming, management and security must accommodate such changes. Objects are the unit of encapsulation of policy: objects are responsible for their own management of resources, internal state and security.

The federal model is one of interworking between groups of closely coupled applications. Each group is independent of the others and is organised to best suit the constraints upon it. Each group chooses which facilities to make available (export) to thegroup with which it interconnects; this group in turn select which facilities to use (import). Federations are therefore peer to peer rather than hierarchical structures.They depend upon the conept of trading.

To allow smooth cooperation between members of different communities, there is a need to unambiguously refer to entities. This is often achieved by agreeing a name for an entity, but this is in conflict with the need to remain in charge of one's own naming policies.The solution is to accept that every name is relative to a context and to extend the name when it passes into a wider context.

4.4 Transparency

Transparency requires that complexity be hidden from the application programmer. It is better for the programmer to express requirements and work to a high level interface which is mapped via tools and libraries on to the low level book-keeping required to drive the mechanisms directly.

Sometimes software writers will want to exercise control over distribution or participate directly in its provision for instance by specifying a specific recovery procedure for certain failures. Transparency must therefore either be selective or allow programmers to influence a generic mechanism by providing specific parameters.

4.5 Concurrency

A clear distinction must be made between application and implementation views of concurrency. Programmers are encouraged to indicate where parallelism is possible and where it is required, but without any reference to how or from where its is obtained. Tools can then used to map the application to available processors, threads, etc where possible or necessary.

Interaction should be synchronous - execution halts until the (remote) activity is complete or there is a failure. Asynchronous interaction can be achieved by generating (locally) concurrent synchronous interactions. This unifies local and remote interaction into a single framework and ensure failures in any remote interaction are explicit.

4.6 Configuration

The configuration of a large distributed system is never fixed. New parts are added, old parts removed. A maintenance program is carried out which continuously changes the configuration. Fault treatment and load balancing algorithms also influence the system configuration.

Late binding of clients to services is essential. All components, both large and small, should be equally configurable and manageable.All accesses must be type checked to reduce the risk of unpredictable behaviour. It must be possible to obtain the description of any component on-line. Type checking should be an integral part of the configuration process.

5 Acknowledgements

The description of the rationale for ANSA and its principles is taken from a Architectural Report written by Rob van der Linden, manager of the ANSA core team. The scenario is loosely based on a multi-media conferencing system built upon prototype ANSA software by Professor Peter Linington and his team at the University of Kent at Canterbury, England.

References

[ANSA 91]

ANSAware Version 4.1 Manual, Architecture Projects Management Ltd, Cambridge, UK, May 1992.

[ISO 93a]

ISO/IEC 10746-2, ITU-TS Recommendation X.902: Basic Reference Model of Open DIstributed Processing: Descriptive Model, (2nd CD draft) June 1993.

[ISO 93b]

ISO/IEC 10746-3, ITU-TS Recommendation X.903: Basic Reference Model of Open DIstributed Processing: Prescriptive Model, (2nd CD draft) June 1993.

[MURRAY 92]

Stephen Murray, “The NASA Astrophysics Data System, Smithsonian Institution Astrophysical Laboratory, published by Architecture Projects Management Ltd, Cambridge, UK, April 1992.

[RUMBAUGH 91]

James Rumbaugh, Michael Blaha, William Premerlani, Federick Eddy and William Lorensen, Object-Oriented Modeling and Design, Prentice Hall, 1991.

[BARR 93]

Willian J Barr, Trevor Boyd and Yuji Inoue, “The TINA Initiative”, IEEE Communications Magazine. March 1993, pp70-76.

[WILDE 93]

N Wilde, P. Matthews and R Huitt, “Maintaining Object-Oriented Software”, IEEE Software, Vol. 10, No. 1, January 1993.