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

Distributed Objects - Improving Traditional Object Models

Hugh Tonks

Abstract

The notes for a paper given to Objects in Europe (?)

The business problem addressed is the need to react quickly to a changing environment. Object technology is a potential solution.

The technical problem created by that business problem is that traditional object model typified by Smallltalk suffer from implicit assumptions that break down in the real world of distributed systems.

The solution being offered is the application of ANSA when selecting an object model.

[Lightly reformatted raw text: Chris Mayers]

APM.1214.00.02 **Draft** 9 May 1994

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

Distribution:
Supersedes:
Superseded by:

Distributed Objects - Improving Traditional Object Models

Distributed Objects - Improving Traditional Object Models

Hugh Tonks

APM.1214.00.02

9 May 1994

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	**1**	**Distributed Objects - Improving Traditional Object Models**
1	1.1	Introduction
1	1.2	History
1	1.3	The Aim of ANSA
2	1.4	Reversed Assumptions
4	1.5	The ANSA Object Model
5	1.6	Comparison of the ANSA Object Model and Traditional Object Models
6	1.7	ANSA Project contact details:

1 Distributed Objects - Improving Traditional Object Models

1.1 Introduction

Neither object orientation nor distributed computing are new ideas, but their marriage has never been easy. This article concerns an object-based paradigm for distributed computing - the ANSA Architecture - and will cover ANSA's principles together with their justification. It is difficult to deal with such a wide-ranging subject in a short article; readers who would like to read more (or vent their spleen), are invited to contact the author.

1.2 History

ANSA's inception was in 1985, with the eponymous Alvey project; its sponsors were BT, DEC, GEC Marconi, GPT (as Plessey), HP, ICL, ITL, Olivetti, Racal, and STL. The focus of the project then, as now, was distributed computing. The Alvey phase concluded in 1989 with the publication of the ANSA Reference Manual, and the sponsors continued the work as Esprit project 2267 (Integrated Systems Architecture). New European sponsors were added for this phase, the complete list being AEG, Alcatel Research (Vienna), BT, CTI-Patras, DEC, Ericsson Telecom, France Telecom (SEPT), GEC Marconi, GPT, HP, ICL, Philips (now Origin), Siemens, STL, Televerket, and Voest Alpine.

With the end of the Esprit phase earlier this year, the project was opened up for full commercial funding by companies worldwide. The current sponsor list is Bellcore, Bell Northern Research - Europe, BT, DEC, France Telecom, GEC Marconi, GPT, HP, ICL, and Open Connexion (Australia). At the time of writing several new sponsors, mostly American end-user organisations, are in the throes of signing up.

1.3 The Aim of ANSA

The work on ANSA had an ambitious goal: to provide an Architecture which was completely generic in application scope, allowing the creation of distributed systems without constraints on size or technology mix, operated by multiple administrations each with their own organisational policies; but at the same time to reduce the complexity of such systems to a manageable state. Although there is much work still to do, there is sufficient work completed to realise this goal, and this foundation work forms the basis of a forthcoming ISO standard (ISO 10746 - the Basic Reference Model for Open Distributed Processing), which will also appear as X.901 - X.904.

The term architecture is used here somewhat differently to most people's expectations. "Where is the architectural block diagram?" I hear you cry - and the answer is that this is a meaningless question. ANSA is not (as most

architectures are) a design for the implementation of a particular distributed system; rather, it is a set of principles, basic components, recipes for their combination, rules, and guidelines on trade-offs, and these together provide the understanding to let you design any distributed system. Nothing in ANSA is compulsory - you can follow as many or as few of the principles as you wish, but you should be aware of the consequences of your decisions, should you (for example) sacrifice architectural integrity in favour of performance.

It is the basic components, and some of the underlying principles, that I shall set out here, to enable a comparison with traditional object thinking.

1.4 Reversed Assumptions

Early on in the ANSA project, we realised that the approach characterised by building a centralised system and then trying to distribute it would not suffice, that distribution was more fundamental than any application considerations. Our attention was turned towards a set of basic assumptions that would enable, rather than constrain, the designer. These assumptions can be derived simply by a consideration of the differences between a centralised and a distributed environment. It turns out that not only are many of the traditional assumptions invalid for networked systems, they must be reversed - hence the heading.

A comparison of some differences is instructive:

- local (traditional assumption) -> remote (reversed assumption)

  On a single host, everything is local; in a network, everything should be assumed to be remote. Some entities will be local to each other, but this would be a special case, which could be harnessed to allow an optimisation. There is always a trade-off involved; in this case, better performance can be achieved by using local rather than remote communications, but this impedes any attempt to separate those local entities in the future, and thus configuration flexibility is lost.

- homogeneity -> heterogeneity

  A single host provides one kind of hardware, one operating system, one filing system, one set of administrative procedures, and so on - these are luxuries which are not generally available in a mixed network. Ideally, the design process should be completely independent of the underlying technology, which allows for maximum flexibility and, as a side effect, ensures future-proofing.

- direct bindings -> indirect bindings

  The hard-coding into software of information about the location of other programs and services, is a major obstacle to flexibility. Indirection is a key idea in solving this problem, giving us the useful notion of references to services (about which more later).

- sequential execution -> concurrent execution

  On a single processor machine, everything happens sequentially, even with the illusion of multi-tasking provided by operating systems. With many processors, concurreny is the norm, and trying to get things to happen sequentially takes a deal of effort. Concurrency makes the task of debugging and monitoring a distributed system much harder than for the

single machine, given race conditions and the potential irreproducibility of sequences of events.

- global synchronisation -> asynchrony

  Single machines have single clocks, and everything can be made to march in time. With many machines, the global synchronisation of many clocks becomes infeasible - it suffers from scaling problems. This is not to say that you may not have a notion of common time across several machines, but that it should not be relied upon as a generally available property of systems.

- single instance -> replicated group

  Replication of software is a facility which has little use on a stand-alone machine. In a distributed environment it is a powerful feature, which can be used as the basis for fault-tolerance, data sharing between databases, load balancing, and other applications.

- fixed location -> migration

  Moving a program from one section of memory to another inside a machine is not particularly helpful. Moving programs, especially while running, from one machine to another can be beneficial; it aids system management, and can reduce response time and network loading by moving communicating objects near to each other.

- shared memory -> disjoint memories

  Shared memory is an efficient but inflexible method of communication between two or more programs. However, when working in an object-based environment this can violate the principle of encapsulation, which is discussed below.

- single name space -> federated name spaces

  Global naming is easy if you are the only user. Indeed, global naming is possible if there are many organisations involved - but it can only be achieved by agreement. In general, name spaces will consist of a collection of local name spaces with some federating principle that allows communication between them. Local name spaces may have widely varying rules for naming, and these must be respected by external users.

- * total failure -> partial failure

  When monolithic programs fail, the whole applicaation dies, and no further processing happens. With a system comprising interacting parts, any individual part may fail independently of the others. This requires software to be written to tolerate or even compensate for this kind of failure.

There are other minor reversed assumptions, but even a cursory examination of those listed above is enough to realise that the traditional assumptions are built into hardware, operating systems, programming languages and applications. It is no wonder that the distributed systems nut has been so hard to crack, given the obstructions placed in the path by the very technology that is supposed to help us.

1.5 The ANSA Object Model

With an understanding of the reversed assumptions, together with requirements that any solution must scale well, be both generic and flexible, and not place arbitrary restrictions on the designer, it is possible to piece together a programming model which facilitates distribution. The benefits afforded by encapsulation strongly suggest that an object-based model is appropriate. With the growing need for large scale, distributed, co-operating systems than span geographical, political and organisational boundaries, the requirement that objects be distributable must come first - properties of objects, however sacred, which do not meet these needs will be at best a hindrance.

Curiously, the ANSA Object Model does not start with objects, but with the contribution each object makes to the whole system, in terms of the services which they provide to one another. The principle of encapsulation, which states that no object may look inside another object except via a predefined access route, holds the key, and it is the routes into an object which are important. Smalltalk fans will recognise that I am talking about methods, or operations (in ANSA parlance). There is a reason for using a different term, and that is that we are not talking about quite the same thing. The basic idea is the same - that an operation is an information processing function that may be invoked by an external object, passing arguments and expecting results - but ANSA operations differ from Smalltalk methods in a number of important ways.

First, ANSA operations may be grouped together to form a service, with operations grouped according to common functionality, common security characteristics, or any other guideline. It is the service that is the basic building block of ANSA. Whereas Smalltalk methods exist for the exact lifetime of their object, ANSA services have more independence, in that an ANSA object may support more than one service concurrently, and may support concurrent multiple instantiations of the same service (this is one way to deal with multiple clients), and furthermore, ANSA service instances may be dynamically manufactured and destroyed by the object that supports them. Thus it is possible to represent with a diagram the potential structure of interaction of objects in an ANSA system; though at any moment the true structure will depend on the particular interactions and service links between objects that exist at the time. The ANSA object model is free of restriction, and in fact designed to be completely flexible; this is essential if it is to appeal to all possible users.

This object model lends itself naturally to the client/server model of interaction, with the proviso that the terms client and server are roles, that is, they must be used within the context of the relationship between two objects. This is necessary, because ANSA objects may concurrently be both client (of several services, perhaps provided by different objects) and server (of several copies each of several services). This would allow, amongst other things, an ANSA object running on a PC to be a server as well as a client, contrary to conventional "wisdom" which regards PCs as client engines running GUI front-ends.

Because of the requirement for indirect binding, ANSA deals in references to services rather than hard data about the services themselves. Service references are currency - that is, an object which possesses a service reference may use any of the operations provided by that service, and service references

may themselves be passed as arguments and results in operation calls. It is this last facility that allows the bootstrapping of an ANSA system into existence.

One of the few rules that ANSA insists on is that if you need run-time binding, you must have a Trader in your system. A Trader is an application which provides a service that trades other service references - similar to a yellow pages facility. The trading service must be in a pre-defined and well-known place. A server object may, after generating a service instance, advertise with the Trader the availability of that service (a process known as Exporting). A client object may go to the Trader to extract references to suitable services that it needs (Importing). Traders exist only to effect introductions, and after that they do not play gooseberry to future interactions between client and server.

1.6 Comparison of the ANSA Object Model and Traditional Object Models

There are three frequently cited benefits of using objects, namely encapsulation, code re-use, and inheritance. Whereas all three work well in localised environments, this is not true for distributed environments, which case I shall consider below.

Encapsulation is the most useful of the three. The shell around an object, which prevents tampering by other objects, allows the distinction to be made between what and how - between the abstract services which the object may provide or use, and the way in which the object is implemented. Encapsulation allows us to separate a computational view of an object from its engineering, a separation which is vital to good design. Enforcement of encapsulation also permits a good model of distributed security, in which each object is responsible for its own protection (although objects may choose to delegate this responsibility). In short, encapsulation is a Good Thing.

Code re-use may be divided into two categories. The first of these has more benefit, and that is the re-use of whole components. There is a balance to be struck; the smaller a component (in terms of its functionality), the better its potential for re-use, but the less it does. The second is the re-use, possibly with modification, of sections of code from other objects. This may be useful, but you must be very careful to understand what it is that is being extracted and modified, often out of context. There is great scope for error here.

Inheritance is a term that may mean three different things. The first is the static inheritance of specification. That is, the re-use, at compile time, of a service (or method, object, or whatever) specification. The advantage is the consistency this brings to two or more objects; the disadvantage is that this is also a dependency which needs managing, especially in the event of the modification of the inherited specification.

The second meaning of inheritance is the static inheritance of implementation, i.e. the re-use in one object of code (and possibly also variables) from another object. Even greater consistency results, together with immediate savings on programming time, but the dependency which is introduced is correspondingly more restrictive. To use this approach, you should be sure of what you are doing, and be confident of coping with future changes.

The third meaning of inheritance, the one so beloved of Smalltalkers, is the dynamic (reactive) inheritance of other object methods. This kind of inheritance assumes the instant magical propogation of changes throughout

the system, and as such, is largely nonsensical in real-world environments. ANSA does not forbid the use of reactive inheritance, but counsels you to be very careful where and to what extent you use it, and not to rely on it as a generically suitable mechanism for building systems.

There are a number of factors stacked against dynamic inheritance, and the reversed assumptions play their part:

- accessing inherited variables at run time violates encapsulation
- the latency involved in propogation of changes makes it difficult to decide where to locate and execute methods for a class instantiated on many nodes
- updating a root class method implies a communication channel to every instance; simultaneous update further implies a two-phase commit mechanism or worse; and this assumes you can find all the instances!
- in a multi-administration environment, who owns the root classes, and do you trust them to retain the integrity of those classes?
- in a heterogeneous environment, classes built for one platform won't work on other platforms (I discount the interpreted languages here); keeping one copy of a class built for each platform removes any elegance the scheme once had
- the redefinition of upper level classes will change the types of classes that inherit, and this in turn forces you to use a fully dynamic abstract type checking system.

There is an elegant answer to these problems, and that is to get access to another object's services via a Trader. Don't inherit - import instead. Because the ANSA Architecture has been specifically designed not to suffer from this sort of problem, service trading does not have the same drawbacks as reactive inheritance.

It is important, therefore, when considering the implementation of a heterogeneous distributed system, to choose one's object paradigm very carefully. Unfortunately most object models come ready-made, courtesy of particular programming languages. This is unfortunate, as it often forces an unnatural approach to be taken. If necessary, do without some of the object features that languages like C++ provide, if it allows you to take a more appropriate view of the way your objects should work. There is no reason why C++ (or C, for that matter) cannot be connected to an ANSA-conformant infrastructure, thus giving you a distribution-efficient way of writing object-based client/server systems.

In conclusion, I am happy to be able to report that this technique has been implemented, works, and works well. Why bother with reactive inheritance when you don't need it?

1.7 ANSA Project contact details:

Hugh Tonks
Architecture Projects Management Ltd
Poseidon House
Castle Park
Cambridge CB3 0RD
Tel: +44 223 323010

Fax: +44 223 359779 Email: ht@ansa.co.uk

Biographical notes (if needed):

Hugh Tonks has been with the ANSA project for nearly five years, both as a member of the research team and latterly as business manager. Prior to that, after graduating in Computer Science from Cambridge, he worked for GEC and Torch, as a consultant to the thoroughbred bloodstock industry, and for the US Department of Customs in Saudi Arabia, specialising in database systems of various kinds. His hobbies include exotic cookery, playing jazz piano, and juggling (mostly family, job and sleep).

References

[ANSA 91]

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