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**

APM Business Unit

Client/Server - Extending the Paradigm for Distributed Systems

Hugh Tonks

Abstract

A paper submitted to the BCS Client-Server Journal (?)

The business problem addressed is that client/server technology is being heavily promoted without explaining the principles behind it. It is therefore impossible to determine whether it is an appropriate solution to a particular business problem.

The technical problem created by that business problem is that existing systems have been design using assumptions that no longer hold for distributed systems..

The solution being offered is the use of ANSA.

[Lightly reformatted from the raw text: Chris Mayers]

APM.1213.00.02 **Draft** 20 May 1994

External Paper

Distribution:
Supersedes:
Superseded by:

Client/Server - Extending the Paradigm for Distributed Systems

Client/Server - Extending the Paradigm for Distributed Systems

Hugh Tonks

APM.1213.00.02

20 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 Client/Server - Extending the Paradigm for Distributed Systems

It is rare these days to read an article in a computer journal, or an advertisement, without seeing the words "client/server", "object-orientated" (these days spelled "object-oriented"), or worse still, "open". These terms are bandied about, often by marketing types who are never slow to jump on a good bandwagon; but they are often poorly understand, and incorrectly used. Consequently, there is still much confusion over their use and meaning.

Help is at hand, though, in the form of a forthcoming ISO/IEC standard known as the Basic Reference Model for Open Distributed Processing, to be ISO 10746. This is expected to mature as a Draft International Standard early next year, and is based largely on the work of the ANSA project in Cambridge (U.K.). This project started with Alvey, progressed through Esprit, and now continues with commercial funding (the current sponsors are BNR-Europe, Bellcore, BT, DEC, France Telecom, GEC Marconi, GPT, HP, ICL, and Open Connexion).

The ANSA project's main focus has been to derive an Architecture for Open Distributed Processing which provides new ways of thinking about the design and construction of object-orientated client/server distributed systems. The ANSA Architecture allows objects to interact remotely by providing services to one another,with the emphasis squarely upon building practical heterogeneous distributed systems rather than the application of fashionable terminology for its own sake.

This article sets out some of the major premises of ANSA and discusses what can realistically be achieved with objects and client/server principles. In doing so, it is worth explaining the rationale behind ANSA to justify its view of client/server, which is subtly different from the "classical" view.

1.1 Reversed Assumptions

Historically, few organisations owned more than one computer, being constrained by cost, knowledge base, and the lack of sophistication of the technology. A centralised approach was thus innate, and nobody questioned it; after all, it was better to have one computer than none. Programming languages, operating systems, and applications were developed for these central behemoths, and those techniques of analysis, design, implementation and configuration survive to this day, embodied in a thousand and one different places.

But the world really isn't like that - the world is naturally distributed. Information is distributed across libraries, offices, filing cabinets, pieces of paper, books, and brains. Things which process information - people and machines - are similarly distributed. People don't have a problem with dealing with this sort of world, but computers do, partly because their hardware and

software (and the way in which they are used) is constrained by archaic principles which work for centralised systems but fail, often spectacularly, in the distributed case.

These principles need a thorough overhaul if they are to form the foundation for distributed or networked computing. In fact, examination shows that many principles need to be reversed before progress can be made. For example, single machines provide a homogeneous, localised environment in which a reliable clock is provided, universal synchronisation is possible, and in which global naming and other global administration properties are acceptable (and work). For a generic network, few of these luxuries are available; one must be prepared for heterogeneous technology, spread over a wide distance, without an accurate global clock, with natural asynchrony, and the possibility that global naming and administration policies may not be enforceable. Most systems suffer to some degree from these infelicities.

ANSA has therefore re-written the basic assumptions to provide a proper foundation for distributed computing. If you believe that this is unlikely to apply to you, then think again - unless you have only one computer you are likely to run up against these problems, in the future if not already. With the growing trend towards inter-company computing (e.g. with non-interactive EDI, and other obsolescent technology), even users of single machines may experience difficulties when they try to access remote services provided by other organisations.

1.2 Transparencies

The extra complexity which distributed systems introduce has to be dealt with in some way. Life is already hard enough for the designer and programmer, without a whole new set of programming paradigms to absorb. The question is one of how to give the programmer access to the benefits of distributed computing (replication, migration, concurrency, fault tolerance, etc.) without getting bogged down in overcoming the problems (heterogeneity, separation of components, lack of global anything, and so on).

There is no doubt that these problems can be overcome by judicious design and extra code, but this in itself is a problem, in that it causes software to become even more complex. So rather than require programmers to write, by hand, this new code into all their software, ANSA advocates an "abstract and automate" approach. This is nothing new, as the technique has been successfully used many times in the history of computing; we have moved from machine code to assembler, from assembler to Fortran and COBOL, and thence to 4GLs, each time allowing programmers to specify their requirements in a high-level (i.e. more abstract) notation, and using tools (in this case, compilers, interpreters) to transform these requirements into machine-level instructions.

The goal towards which ANSA is moving is to give programmers a set of concise notations, each allowing the declarative expression of distribution-related requirements, which can then be turned into code by the appropriate tool. The advantages are legion - less code, hence less error (providing you trust the tool), less time to produce, shorter programs, more productivity, easier maintenance, and increased independence from hardware platforms. The overall effect is to add extra facilities to software transparently. All that is

added to the source code is a small number of statements about the desired behaviour of the program.

1.3 The ANSA Object Model

"Don't have more elements than you need" - thus spake William of Occam, and a fine principle it is, even though he said it in Latin. In keeping with this idea of not over-complicating matters, ANSA has at its heart one simple concept - that of the "service". A service is just a set of information processing functions, known to some as an API (Application Program Interface). What the service does is not important (that is a matter for the system designers) as ANSA is concerned with what can be said about services in general. The concept of service is far reaching, in that anything can be viewed as a service - including legacy systems.

What interests the programmer about a service is its signature, that is, information about the available functions, their arguments, their possible termination values and the results that may be associated with each possible termination. Note that having multiple terminations builds in at a fundamental level proper facilities for error-handling. A service may also have "quality of service" attributes associated with it, that allow the programmer to select a particular instance of a service from a collection of apparently identical services. For example, a print service may have a signature that shows three operations (print a document, cancel a print, examine the queue), several abnormal terminations (printer jammed, no paper, error in PostScript file, etc), and quality of service attributes relating to paper size, paper colour, dots per inch, cost per page, location of printer, and make of printer. These attributes would vary considerably between a cheap dot matrix and an expensive laser printer, and by making the appropriate service selection from those available based on quality of service, the programmer gets to use the right sort of printer.

That which is not necessarily of interest to the programmer is complexity which may be hidden by a transparency. This includes information about where the service is provided (location transparency), what kind of technology supports the service (access transparency), whether the service is provided by one's own organisation or not (federation transparency), whether the service needs activating before it can reply (resource transparency), and numerous others factors.

So, returning to the main theme, services are really all you need to think about, given that a service based view allows you to abstract away from many difficult problems. But services don't exist in vacuo, they require resources (storage, processing, communications) in order to run, and it is the object that provides the wherewithal for services to exist and operate.

Here ANSA diverges from the various "classical" object models. Common views of objects - the Smalltalk view, the C++ view - obscure the fact that objects are independent of the programming language that is used to implement them; this muddying of issues has not helped understanding. Because this kind of object is dependent on the syntax of a particular language, the sort of things one can do with objects are also similarly constrained. Smalltalk relies heavily on inheritance; C++ allows only one interface to each object; and other object models permit only a subset of the sort of facilities one might like.

ANSA has attempted to set objects free from this sort of arbitrary restriction, in that it provides arguably the most open and flexible object model. Like Smalltalk, it has methods (called "operations"); but ANSA allows the designer to parcel up sets of operations into interfaces, the interface being the unit of provision of a service. At this stage the interface is merely a specification - we have not yet discussed the implementation options. Traditionally-minded object people might think of an interface definition as a "class", from which interface instances (i.e. running services) may be instantiated.

ANSA allows an object to provide more than one such interface, and to have multiple concurrent instantiations of any of its interfaces. Conversely, it may also use many services (provided by one or more other objects). It may even both provide and use services at the same time. These radical notions are what allow the construction of any topology of interacting objects that can be imagined. In a generic Architecture, such features must be available for the Architecture to be credible.

Similarly, the lifetime of an interface instance need not be identical to that of the object that supports it; in ANSA, interface instances can be produced and destroyed at run time, often in response to changing demand for a service. This allows the object topology to change dynamically, providing the basis for dynamic re-configuration of a software system.

Finally, the way in which objects find and use each other is worthy of note. Traditionally, this has been fixed at compile time, with details of other objects hard-coded in. ANSA does not prohibit compile-time binding, but for flexibility's sake it recommends leaving binding until run time. In fact, ANSA hardly prohibits anything, the view being taken that rules are there to be broken, but one had better understand the consequences of so doing.

Objects may register the availability of the services they provide with an application called the Trader (cf. OMG's Object Request Broker); potential clients may interrogate the Trader to discover what services are available, and may extract services references which then enable them to use those services directly. Traders may be federated together, to form an organisation-wide directory of services. Astute readers may spot that Traders and Yellow Pages phone books perform a similar service.

1.4 Client/Server in ANSA

It is easy to define clients and servers given the above notions; a client is an object which uses a service, and a server is an object which provides a service. ANSA objects may therefore be both client and server (many times over) simultaneously, and thus the use of the terms client and server are with respect to a particular interaction between two objects. The terms define (temporary) roles, which are context-relative.

One can see from this that usage of the terms client and server to describe machines, or even pieces of software, is far too restrictive and leads to an a way of thinking that closes down possibilities rather than opening them up. It isn't unreasonable to talk about client objects and server objects, if that is the only function of those objects (i.e. a particular service is being discussed) but it should be understood that the context needs to be stated.

For example, the popular theory put about by database companies that client/server means primarily a database server and a GUI-based client application demonstrates a restricted view of the world. Another popular theory, that (in

hardware terms) client/server means PC clients and UNIX servers, suffers from the same lack of imagination. Why can't a PC host an object which can provide services? Why can't a UNIX box support an object which can act as a client? No reason at all - just hidebound convention.

1.5 What Are Objects Good For?

Three major benefits are commonly cited for objects - code re-use, inheritance, and encapsulation. Of these, only the last has much to offer in a distributed environment.

Code re-use is somewhat of a red herring. People rarely write programs from scratch these days, they tend to adapt existing software. Is this not code re-use? Libraries are another example of code re-use - and this is ancient technology. Objects do not appear to offer substantial extra benefits in this direction. Besides, the best opportunities for re-use exist when the objects are small, each providing limited but generic functionality; to get substantial re-use, you have to put up with having many objects, and this adds complexity to the configuration if not to the design. There is one other possible meaning of code-use, and this is discussed in 2) below.

Inheritance is not universally well understood, as there are several kinds, offering differing benefits. There are three common varieties:

1. Inheritance of specification at compile time: this allows programmers to ensure that a piece of specification - for example, the definition of an interface - is common to more than one object. This has obvious benefits, in that not having to replicate the definition increases consistency; the down side is that a dependency has been created between the specification and the objects that inherit it (and, implicitly, between those objects themselves), and while small numbers of dependencies are manageable, large numbers can be confusing.
2. Inheritance of code at compile time: this ensures that a particular piece of implementation will be common to multiple objects. The benefits can be greater - less code to write, and perhaps this is what is really meant by "code re-use" - but the dependencies are correspondingly more rigid.
3. Dynamic inheritance (of methods) at run time: this is, on anything but a small or local scale, a no-no. This type of inheritance works satisfactorily on a single user machine (cf. traditional Smalltalk). But when it is moved to a wide-area heterogeneous network things start to go wrong. The whole point of this type of inheritance is that you only ever need a single copy of a piece of code, which gets inherited by anything that needs those facilities. But with various hardware and operating systems, it becomes necessary to keep a version for each platform (a DOS version will not run on UNIX, nor will a SUN UNIX version run on HP UNIX, etc). Even if this can be overcome, you run into other problems, especially with update the root classes. A simultaneous update of all objects inheriting a root class implies a communications link from the root to those objects, and some kind of two-phase commit (or worse) for synchronisation. There are other more subtle problems, but the bottom line is that this kind of inheritance is of limited use when programming in the large due to the scaling issues.

Encapsulation, however, is a much more useful facet of objects. The black-boxness of objects is due to encapsulation, which enables the specification of a service (the signature) to be separated from its implementation (the object).

This separation of concerns is vital to avoid muddling issues and hence botching the design by the inclusion of irrelevant factors. Encapsulation implies that the only way in which one can see inside an object (to get at data, for example) is by invoking a service provided by that object, and examining the results. Encapsulation further implies that objects need a greater automony than they have previously enjoyed; a view which says that objects are responsible for themselves and for the way in which they interact with the outside world has important implications for distributed security models.

ANSA objects are designed primarily to be distributable; there is no requirement to use an "object-oriented language", nor facilities which are of doubtful worth, and indeed the major benefit of using object-orientation is considered to be that of encapsulation.

1.6 Client/Server and Legacy Systems

If there is one big computing problem facing nearly all organisations, it must be that of the legacy system. For those not familiar with the term, a legacy system is one which may be supported by technology that is hard to change, which provides a service on which the organisation depends, but for which the cost of replacement cannot be justified in terms of the risks and problems it presents.

In a sense, a solution to the legacy systems problem is the case against rather than for downsizing. It is a fact that organisations have already invested in technology, but few are making as good use of it as they could, if they knew how. Rather than embark on a crusade against the mainframe, merely because the idea is fashionable, it seems much more sensible to consider how the mainframe, and the software it supports, might form part of a modern networked environment running advanced distributed client/server technology. At some point, the user will want to replace the mainframe, but this should be a properly scheduled event, performed at a time when the cost of so doing has been brought down sufficiently to make replacement justifiable.

To achieve this, one can appeal to some of the ANSA concepts. The first is that of the service - a legacy system can be viewed as one or more services, just in the same way as any other program. Having stated this, ANSA then offers help with integrating legacy services into the network; location transparency hides the fact that the service is running on the mainframe, access transparency hides the technology - to a user, a legacy service could be located, accessed, traded and used in exactly the same way as any other service. The re-presentation of a legacy system as a set of networked services is, surprisingly, neither expensive nor especially complicated.

By initially designing the system at a high enough level, such system differences disappear, and one can concentrate on the more important issue of how the software can enable the organisation to be more efficient and profitable. Choice of technology for implementation is a decision that can well be deferred until one knows exactly what the system is supposed to do, and why. Similarly, configuration of a system may be deferred until run time, as this brings the flexibility and responsiveness which system managers need.

1.7 Conclusion

ANSA has shown the importance of both object-orientation and the client/server paradigm in distributed computing. A few well-chosen principles, several fundamental components, and the rest of the Architecture follows - and is thus justifiable, if one accepts the original premises. It has proved itself in a variety of application areas, including multi-media, telecomms management, databases, process control, CASE tools, and hospital systems. As the basis of ISO 10746 it will have far-reaching effects worldwide.

It is also worth noting that the ANSA principles may be used freely, as a result of their development under Alvey and Esprit. If this really is the age of distributed computing, what are you waiting for?

1.8 ANSA Project contact details

Hugh Tonks
Architecture Projects Management Ltd
Poseidon House
Castle Park
Cambridge CB3 0RD
Tel: 0223 323010
Fax: 0223 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 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.