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

Abstract and Automate

Dave Otway

Abstract

The pace of technological change coupled with the expanding scale of distributed systems are exploding the complexity of distributed software. The scale and rate of change are such that no modification can be propagated throughout the (world-wide) system before it is itself obsolete. This means that attempts to contain this complexity explosion solely by enforcing homogeneity via standardized protocols and Application Programming Interfaces (APIs) are not sufficient. The only hope is to apply increasing degrees of abstraction and automation to the process of building distributed programs.

APM.1020.01 **Approved** 23 May 1994
Technical Report

Distribution:
Supersedes:
Superseded by:

Abstract and Automate

Abstract and Automate

Dave Otway

APM.1020.01

23 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 Overview

1.1 Purpose

This document argues that standardized protocols and Application Programming Interfaces are necessary but not sufficient to solve the problems of programming distributed systems; and that language based approaches using increased abstraction and automation are required.

1.2 Audience

Strategic technical, marketing and purchasing decision makers looking 2 to 5 years ahead in companies supplying and/or using computer systems.

1.3 Relevance

Most programming techniques and tools currently being used or developed for the production of distributed applications will not scale to very large systems, work with federated management domains or cope with the explosion in technological heterogeneity that scale and federation bring in their wake.

These techniques and tools will work acceptably in medium scale, homogeneous single management domains; creating the illusion that they can evolve to cope with all distribution problems and leading the unwary to develop systems that are too complex to be debugged, managed and maintained.

There are technical solutions that avoid these problems and reduce the apparent complexity of distributed applications. These techniques are no more complex than those currently in use but are not direct evolutionary developments of them. Some changes need to be made in the way programs are developed because the current techniques are being stretched to their breaking point by the problems of distribution.

1.4 Main conclusions

- most of the world's computers are being interconnected
- distributed systems software is becoming too complex
- standardization of protocols and APIs is necessary but not sufficient
- now is the time to increase the degrees of abstraction and automation
- application programs can be constructed to be completely independent of their configuration, distribution engineering and supporting technology
- automatic tools can cope with optimization, evolution and reconfiguration, enabling technology to be chosen on a cost/benefit basis

1.5 Significance

- the problems of doing business in a world of large scale, heterogeneous, federated distributed systems will not go away if they are ignored
- most companies will only be able to afford one shot at getting it right
- it takes time to change a programming culture and development tools
- now is the time to start thinking about it

2 The opportunity

The world's computing and telecomms community is constructing larger and larger components of a world-wide distributed computer system. This will be the largest machine ever known. It will encompass and dwarf the current largest machine (the international telephone network) and be constructed in a much shorter timescale.

There are no technological or economic barriers to the construction of a world-wide distributed system which could interconnect most of the world's computers.

All the necessary hardware (mips, memory and bandwidth) plus the distribution software engineering is available and affordable. Its installation and interconnection is inevitable; the process can't be stopped. However badly it is designed and managed, it will still be built.

For those that succeed in getting their software to interwork, the benefits will be enormous. Electronic trading will become the norm.

3 The challenge

There are known and workable solutions to most of the technical problems of building distributed systems. The only one that may still hold any surprises is that of sheer scale.

All the remaining problems are management concerns.

- We (i.e. humanity) have never built anything as complex or as big. The telecomms industry only just copes and it has a narrow market structure, virtual monopolies and government regulation on its side. The computer market is very much less structured and more diversified, with the majority of the investment being made by the users not the suppliers.
- Nobody is in charge of this project, nobody knows what the design is, there is no implementation plan and most people involved in it don't even realise that they are part of it.
- The technology being employed is diverse; and both it and the organisations involved are changing at a much faster rate than the system being implemented.
- Even if we could design and build a system where everything could interwork (rather than just communicate), then we would be faced with the even bigger problems of debugging, managing and maintaining it.

Unfortunately, these problems won't go away if you ignore them. If a significant number of your competitors manage to get their act together then your company could be at such a disadvantage that it might go out of business before it can catch up. This is particularly true of any form of electronic trading. If you can't do it, you won't be in the market!

So you are forced to take a gamble; the trouble is that the costs are very large and, if you bet on the wrong technology, you may end up in a worse position than when you started.

4 The explosion of complexity

From an engineering point of view the problems can be categorized as:

- *Complexity* too many details
- *Scale* too many components
- *Federation* too many chiefs
- *Heterogeneity* too many varieties
- *Separation* too many indirections
- *Concurrency* too many conflicts
- *Reliability* too many faults
- *Availability* too many demands
- *Security* too many threats
- *Evolution* too many changes

These all reinforce each other in adding to the first. None of them will go away and they will all get worse, resulting in an explosion of complexity.

4.1 Scale

The sheer number of components in a world-wide system means that their state can never be consistent because the time taken to check consistency is very much longer than the mean time between state changes. The system must tolerate inconsistency and ensure that it will eventually converge.

Similarly, the technology used to build the system can never be homogeneous because the intervals between technology changes are much shorter than their propagation times and life times. The system will either be obsolete before it is completed or it must tolerate heterogeneous technology.

4.2 Federation

The various sub-systems will be owned and operated by many organisations in many countries. There will be no central control, no strategic direction and no consistency. Agreements must be reached by negotiation and can only be enforced through the legal system or by withdrawing co-operation. The management controls traditionally used to coerce conformance inside a single organisation will not be available.

4.3 Heterogeneity

The system will be built out of many different varieties of technology (both hardware and software). This diversity is partly historical, providing

backward compatibility; partly specialisation, providing specific optimizations; and partly innovative, attacking new markets. There is also a healthy dose of competition, together with the usual fashions, dogmas and prejudices present in all human enterprises.

This diversity will never be eliminated because there are no universal optimizations. System design will always involve trade-offs between such parameters as: price/performance, security/convenience, availability/ consistency, latency/throughput, etc.

4.4 Separation

The main characteristic of distributed systems is that, in general, everything else is somewhere else. This means that not only do you have to find out where it is but also that you can't manipulate it directly; all access has to be indirect. In addition, it might not be where you were told it was, it might not still be where it was when you last used it, and there may be multiple copies and/or versions of it.

4.5 Concurrency

Distributed systems are naturally concurrent and it is impossible for clients to co-ordinate their access to services, presenting many opportunities for inadvertent conflicts. Only servers can coordinate their clients' access, and they must do this in ignorance of their clients' intentions.

4.6 Reliability

Distributed systems introduce the possibility of faults on every interaction plus the complications of partial failures and network partitions.

4.7 Availability

The demands placed on any component becomes less predictable as the scale of the system increases. Increased latency due to increases in scale and load may result in unacceptably slow response times. There are techniques for dealing with these problems (replication, migration, caching) but they all reduce consistency.

4.8 Security

Because there is no central authority, there can be no central trust. Each organisation must protect itself from all potential threats and police all permitted interworking. It will have its own security requirements, policies and mechanisms.

4.9 Evolution

The birth rate of new technologies is increasing while their gestation period is decreasing; product development cycles are now sub-annual while their

economic exploitation periods and propagation times are multi-annual. Therefore, multiple generations of a particular technology have to co-exist.

Every year the cost of software increases and the cost of hardware decreases, causing a constant shift in the relative economics of software which makes it more expensive to develop but cheaper to deploy.

The pace of organisational change (both internal restructuring and external merger/de-mergers) has increased to the point where it is significantly shorter than the lifetime of computer systems (particularly databases).

Application programs and databases are the longest lived components of a computer system and consume the major investment. We have to learn to construct them so that they can survive a complete change of both the technology that supports them and the structure of the organisations that own them. There is no alternative; technology or organisations which have their evolution frozen will not survive.

5 Current approaches

The computing community's current approach to dealing with the complexity explosion caused by the construction of ever larger distributed systems can be summed up as “standardize and automate”.

The standardization element is popularly known as “open systems” and has two main components: communications protocols and Application Programming Interfaces (APIs).

The protocols define the syntax and semantics of communications messages and provide an interworking capability. The APIs provide program portability and are used to interface applications to protocols, operating systems and standard services.

There is an increasing but piecemeal use of automation, particularly preprocessors and stub generators for the generation of remote invocations, communication stub functions and marshalling code.

5.1 Limitations of APIs

APIs consist of a list of function specifications which tend to have very few of the unifying principles needed to provide coherence. If a small number of orthogonal concepts are presented via an API then allowing for all their permutations leads to a large number of procedures or long lists of arguments. APIs are generally too large for a programmer to remember.

APIs are the software equivalent of an instruction set and because the programmer directly codes to them, there can be no pre-execution check on their correct usage. Therefore, to avoid total chaos the APIs have to include extensive run-time checks; but these can only protect the integrity of the service behind the API, not the application program.

There are a number of APIs and they are arbitrarily different. They take a long time to deploy so as well as different APIs coexisting there will also be multiple versions of each one, particularly as different suppliers have their own product cycles.

Program portability is limited by API stability and portability.

5.2 Current limits to automation

Preprocessors and stub generators are inherently limited forms of automation in that they perform very specific transformations on specific components of the program, which usually have to be marked in some way.

Applying automation to programs written using APIs is very difficult because the APIs lack any grammar, making it very hard for the tools to decode the programmer's intentions. It is like trying to decipher an English text written by a Japanese person using only a Japanese to English dictionary and with no

knowledge of English grammar; the writer knows what the text means but the reader can only guess.

If the programmer makes the optimizations (i.e. writes for a specific API) then this makes it even more difficult to automatically translate the program to use an alternative API. Since the lifetimes of the implementation technologies are now much shorter than the lifetimes of application programs, this means that hand “optimized” programs are sub-optimal for most of their (mature) life.

5.3 Limitations of current commercial programming languages

The current generation of commercial programming languages were developed for single processor stand-alone machines. They don't have the facilities to cope with many aspects of distributed programming such as separation, concurrency, interaction failures, partial failures and dynamic binding. This deficiency has been made up for by the extension and addition of APIs.

5.4 The limitations of standardisation

Standards are necessary for any interworking or portability, but they are not sufficient to provide either in a distributed system that spans the planet.

The current standardization process attempts to solve the complexity explosion by enforcing homogeneity; but as even the most cursory examination shows, homogeneity is an impossible goal to achieve in a world-wide distributed system.

The functionality that standard protocols and APIs provide is necessary and the “open systems” approach they enable leads to a fairer market place; but this does not mean that application programmers need to be directly exposed to them.

6 Historical approaches

We've been in this kind of mess many times before and we have invariably solved it (eventually, and after much debate) by the “abstract and automate” approach; for example:

- octal/hex instruction codes to assemblers
- assemblers to high-level languages
- physical to logical to “semantic” data base schemas
- 4th generation languages

The only significant exception to this approach is the structured programming revolution, where we significantly reduced program complexity by exercising self-restraint, when the great majority of programmers were persuaded to stop using goto.

There has always been a fierce rearguard action (clinging to the efficiency argument) to any increase in the level of abstraction, but the flexibility, reduced complexity and increased productivity of the abstractions have always won through in the end.

In an era when mips are doubling and costs halving annually, the efficiency argument against increased abstraction is very short term. Compiler technology has improved to the point at which compilers now generate more efficient code than programmers, while hand coded optimizations only apply to one technology and have to be redone when the configuration is changed.

This increase in abstraction has also had the side effect of freeing the development of the underlying technologies from the dead hand of backwards compatibility - for example, would RISC processors have been developed if everybody was still programming in IBM system 360 assembler?

The first and last to adopt new abstractions have always suffered worst from the trauma of change, but the first ones have gained an advantage as well. The smart people who gained the greatest advantage and suffered the least pain have generally been the ones in the second wave after the pioneering work had been done by someone else; but this strategy requires a lot of luck or very precise timing (which means you must already have done the groundwork and be closely monitoring both the technology and the market).

The amount of automation employed in hardware development and production tends to increase at a constant rate because changes can be made relatively independently of one another. The amount of automation employed in software development/production tends to increase in large infrequent steps because everything is so intertwined that it all has to be changed at the same time. This process of change is so painful that it is put off for as long as possible by increasing the quantity (and thereby decreasing the average quality) of programmers until the situation gets out of control.

7 The way forward

The complexity of distributed systems has now surpassed the point where we can reasonably expect application programmers to be able to cope with it.

The time has come (yet again) to increase both the level of abstraction and the degree of automation in order to enable application programmers to write the vast amounts of software required to exploit the potential of the dramatically increasing price/performance of the hardware.

These abstractions need to provide what good abstractions have always provided:

- hide those details which are irrelevant to application programmers
- hide arbitrary syntactic differences between technologies
- keep concepts as independent and orthogonal as possible
- impose the simplest possible conformance rules
- provide the maximum configuration flexibility

and the automation tools need to:

- compile application programs onto any suitable technology
- optimize application programs for any particular configuration
- transform declarative requirements into imperative statements
- check that the application program will execute correctly

8 Objectives

This section considers the specific objectives of increased abstraction and automation and lists the requirements needed to satisfy them. As there is considerable overlap in the requirements, these are only listed here and discussed in detail in the following section.

8.1 Abstraction

Abstractions for distribution have four major objectives: interworking, portability, modularity and transparency. Of these, interworking takes precedence; but in practice the requirements do not clash and have much in common.

8.1.1 Interworking

To allow for separation, we need to abstract over the possible remoteness of the component that we are interacting with; this requires that:

- all bindings are indirect
- all accesses to external state are procedural
- all procedure invocations may report failures

To allow for reconfiguration and migration, we need to abstract over the location of the component that we are interacting with; this requires that:

- all bindings are dynamic
- all bindings are indirect

To allow for heterogeneity of communications, we need to abstract over the concrete syntax of state representations in messages; this requires that:

- all interfaces are Abstract Data Types (ADTs)[1] [§9.2]
- all arguments and results are references to ADTs

To allow clients to be independent of each other, we need to avoid any requirement for them to communicate with each other; this requires that:

- all servers must be able to cope with being accessed concurrently

8.1.2 Portability

To allow for different supporting technologies, we need to abstract over the concrete representation of application state; this requires that:

- all application state is represented by ADTs

1. An Abstract Data Type hides the concrete details of its data representation behind a purely procedural interface. Users of an ADT cannot manipulate the data representation but can only ask questions and make update requests via a predefined set of procedures.

- all access to application state is procedural
- the only state which can be directly manipulated by the program are bindings to ADTs

To allow for different implementations of concurrency, we need suitable abstractions for representing and controlling concurrent activities; this requires abstractions for:

- evaluating a set of expressions concurrently and collecting the results
- evaluating an expression completely independently of everything else
- separating the evaluation of potentially interfering invocations
- ordering invocations to protect the integrity requirements of servers

8.1.3 Modularity

Hiding the implementation details of an object (module) requires that:

- all interfaces are ADTs
- all arguments and results are references to ADTs

Minimising the propagation of unnecessary consequential changes caused by changes to operations that aren't being used requires:

- a type system based on conformance rather than equality
- multiple interfaces to an object

8.1.4 Transparency

Transparency is the process of hiding the details of distribution from those programmers who don't need to know it.

Distribution engineering mechanisms take one of three forms. The extra functionality can be inserted (and hidden) in the invocation path, which requires that:

- all bindings are indirect

Alternatively, the required functionality can be added by changing the implementation or by inserting calls to engineering libraries. Hiding these changes to an implementation requires that:

- all interfaces are ADTs
- all arguments and results are references to ADTs
- all application state is represented by ADTs

8.2 Automation

The objectives of increased automation are to translate, optimize, transform and check application programs.

8.2.1 Translation and optimization

Automatic tools can only be used on programs that they can fully understand; this requires that:

- those parts of the programs to be transformed must be constructed from a small coherent set of clearly defined semantics

- there must be precise boundaries drawn around any parts of the program not using semantics known to the tools

In order to apply the tools across a wide range of languages and technologies, we must make as many of the tools as possible independent of the source and target representations; this requires that the tools use:

- an abstract representation of the semantics of a program

8.2.2 Transformation

Most of the engineering details of distribution are of interest to the application programmer only to the extent that they are correctly configured and implemented. Such engineering details should be as *transparent* as possible to the application programmer, but the programmer needs to be able to control whether they are present or not. The programmer just declares what engineering properties are required and uses an automated toolset to supply the necessary engineering mechanisms. This requires that:

- declarative statements are used to control the distribution engineering

8.2.3 Checking

In order to check that the separate components of a distributed application will interwork correctly when executed, it is necessary to define:

- a type system that is independent of the implementations

In order to allow the widest choice of bindings, it is necessary to define:

- a type system based on conformance rather than equality

9 Requirements

This section discusses the detailed requirements for increasing the level of abstraction and automation used in the construction of distributed systems.

9.1 Indirect bindings

Any state that we allow application programmers to access directly is permanently restricted to being co-located with the code that accesses it. If we wish to allow any application state to be remote and to have different representations of it then we must hide the knowledge of both the location and format of all application state from the application programmer. This requires that all program bindings must be indirect and all access to the application state must be procedural.

If we really want to write distributed programs, we must refer to all state via pointers, but must either discipline ourselves not to explicitly de-reference those pointers or write in languages which cannot do it.

Refraining from explicit pointer de-referencing will bring similar benefits to the data integrity of distributed programs that refraining from using goto statements brought to the control paths of structured programs.

Once direct bindings have been eliminated then dynamic linking becomes easy, and extra levels of indirection can be transparently inserted in the invocation path to implement various distribution mechanisms.

9.2 Abstract data types

The minimal requirements for interworking between two components of a distributed program are that each side can ask questions of the other about a particular data set and understand the answers. This only requires:

- a way of identifying the data set of interest
- a way of discriminating the questions
- a way of discriminating the answers
- a way of referring to other data sets

The data set of interest and any other data set referred to in the questions or answers are indirect bindings. The discriminators are strings chosen from a limited set associated with each data set and do not need to be globally unique. Both questions and answers consist of a discriminator and a list of indirect bindings.

Questions are asked (operations are invoked) by quoting the binding to the data set of interest (server) along with a question (operation) and bindings to other data sets (argument list). The answer (termination) and bindings to other data sets (result list) is then returned to the questioner (client).

This kind of interface where information is exchanged by inference from questions and answers, instead of by copying concrete representations (values), is known as an Abstract Data Type (ADT).

Conversations can be conducted via an ADT interface without any reference to concrete data representations; and the correctness of such conversations can be checked before they take place by comparing the ADT definitions used by both sides.

In this way the client and server source programs can be made completely independent of each others implementation language and the protocols used to communicate between them.

Anything beyond this minimal functionality is an optimization. Typical optimizations are to transfer concrete copies (values) of immutable state for those data types that are understood by the presentation protocol being used. These optimizations can be made without changing the source or compiled version of the application program and may vary from link to link. Both ends of the link may also cooperate to define concrete representations for a wider range of immutable data types using the data structuring facilities of the presentation protocol in order to make the link more efficient.

ADTs can also be used to hide details of local state from application source programs, providing very high portability.

9.3 Invocation semantics

If the distinction between local and remote invocations is visible in the source program, the distribution boundaries will have been fixed and the possible configurations severely constrained. There have been many attempts to overcome this problem by making the syntax and semantics of a remote procedure call (RPC) as similar as possible to a local procedure call.

These attempts have never entirely succeeded because certain undesirable failure properties of RPCs cannot be completely abstracted away. Good distribution engineering can drastically reduce the frequency of failures but it can never reduce the failure rate to zero.

A more sustainable abstraction is to assume that all invocations are remote and to construct application programmes which cope with the failure of any procedure call. A program that expects a failure which never happens is inherently more robust than one which suffers a failure it never expected.

Note that the remote invocation abstraction does not reduce the efficiency of local invocations because the compilers and linkers can take advantage of their knowledge of the configuration not to insert the distributed engineering mechanisms between co-located components.

9.3.1 Invocation failures

Distributed application programmes must be constructed to take account of the possibility of the following failures on every invocation.

9.3.1.1 Communication failure

It is not possible to make a communications channel completely reliable. No matter how much redundancy is incorporated into the channel, there will

always be some catastrophic (set of) circumstances in which it is impossible to communicate.

It is also impossible to distinguish a (silent) failure of an end system node from a break in the communications channel.

9.3.1.2 Partial failure

When a program calls a local procedure, both the caller and the callee are inside the same failure boundary. If one fails then both fail, so neither has to take account of the possibility that it might survive the failure of the other.

For a remote invocation, both the caller and the callee may fail independently of the other before, during or after the invocation.

9.3.1.3 Partition

The communications channels may break in such a way as to partition the network into two (or more) sub-networks which are unable to communicate with each other. If caller and callee end up in different sub-networks then both may still be running and able to interact with other components but not with each other. It impossible to maintain consistency between them unless both parties refrain from making any further changes. This problem is particularly acute with replicated objects.

9.3.2 Latency

Latency is the most difficult aspect of distribution to hide. Bandwidth can be increased indefinitely, subject to financial constraints, but the speed of light cannot be changed.

For short communications paths (e.g. within the same building), increased bandwidth may reduce the apparent latency by compressing the message, since at higher bandwidths the gap between the start of a message and the end is shorter even though they both have the same transmission delay as at lower bandwidths. Over wider areas, the transmission delay dominates; e.g the North Atlantic is always going to be at least 0.015 seconds wide by cable and at least 0.5 seconds by satellite.

There are techniques for reducing the effects of latency (e.g. caching, lazy updates, migration, replication), but it will still have a significant effect on performance.

System performance is affected by traffic patterns; so even though latency can't be eliminated, it is worth not making its presence visible in the application source code in order to have the freedom to tune the configuration.

Latency makes it important that the maximum value is extracted from every message. For local invocations, it was merely a matter of programming style whether single or multiple results could be returned; but for (potentially) remote invocations, there is a real payback on returning multiple results.

9.4 Concurrency

Distributed systems are inherently concurrent and since it is impossible to provide 100% reliability, to enforce complete separation or to schedule the system as a whole, each component must be involved in solving its own reliability, separation and scheduling problems.

Standard approaches to providing reliability require well defined concurrency mechanisms with obvious failure semantics. Replication requires a predictable activity structure in order to maintain replica determinism and transactions require a well defined activity structure that can be cleanly unravelled. Restricting each independent activity to a strict tree structure of sub-activities (which is preserved across remote interactions) enables each branch to be predictably labelled and the error reporting and recovery mechanisms to be correctly nested. Using asynchronous messages to provide concurrency in a distributed system would produce even worse spaghetti control structures than using goto statements.

Separation can be achieved using semaphores or some similar mechanism to avoid conflicts but this has the disadvantage that the programmer has to sprinkle calls to a particular mechanism around the application program. If declarative separation predicates were provided at the interface level then any appropriate separation engineering could be added automatically and some potential conflicts could be avoided altogether by pre-scheduling.

Scheduling mechanisms enable concurrency to be exploited. Specifying scheduling constraints declaratively at the interface level enables the appropriate mechanisms to be inserted automatically.

9.5 Semantic straitjacket

Automatic tools can only be used to translate, optimize, transform or check programs that they can understand. But most programming languages are overly rich in features, which provide many ways of achieving the same effect and many ways of using each feature. This results in complex programs from which it is difficult to automatically recover the programmer's original design. Also, all current programming languages have features that will not distribute.

It is much harder to build a tool which checks that a programmer has used a protocol correctly than it is to build a tool which uses that protocol itself. Programs written in languages that allow direct manipulation of concrete representations are very difficult to port to technologies with different concrete representations.

Therefore, if we wish to increase the amount of automation used, it is imperative that we write application programs at a higher level of abstraction using a small set of well understood semantics. The tools can then translate the application program to use the concrete representations provided by the underlying technology. They can also optimize the program to take advantage of the underlying technology and re-optimize it when the configuration is changed. Once this is in place the tools can be enhanced to transform application programs to add various forms of distribution engineering and to check that the program will correctly execute.

9.6 Precise semantic boundaries

We still need to use programs that manipulate representations directly, because a considerable number of them already exist and because in some cases specialised non-distributable languages will be more appropriate for the application. But once the semantic genie escapes from the bottle it is very difficult to get it back in, so we must have very precisely defined and tightly

scoped boundaries around any code which manipulates concrete representations of state.

In order for the distribution tools to stand a chance, we must define precisely where the boundaries lie between the different sets of semantics. We can't do this if these sets of semantics are intermixed; it is only possible if one set of semantics is nested in the other and the boundary conditions are well defined.

Because the external parts of the program will always have to deal with distribution, the top-level of the application program should always be written using the distribution semantics. The semantic boundaries can then be nested as required.

9.7 Decoupling the semantics from the syntax

When we have precisely defined the semantics of the abstractions needed for distributed application programs then we will require a concrete syntax if these abstractions are to be used to write source programs; but they can also be represented as a data structure known as an Abstract Syntax Tree (AST).

This AST can be made to be independent of the syntax of both the source and target languages. If all tools (except parsers and code generators) use an AST as both their input and output representation of a program then they too can be independent of the syntax of the source and target languages.

Target language independence has the advantages of coping with technological heterogeneity and minimising the total investment in the tool set. Source language independence provides the flexibility to cope with backward compatibility and application language specialisation. The source language choices available include:

- distribution abstractions only
- distribution abstractions wrapped around a non-distribution language
- distribution abstractions added to an existing language

Distribution abstractions can be implemented in more than one concrete syntax; in particular, when used to wrap or extend another language, they can be made compatible with the syntax of that language.

If the AST is also provided with multiple views, by implementing it as a set of inter-related ADTs, then each tool can have its knowledge of the AST restricted to that which is strictly necessary so that the tools can be made as independent as possible of one another.

Design support tools can also use AST representations. For instance, if all interfaces were stored in a data dictionary as an AST representation of an ADT, client/server stubs and marshalling routines could then be automatically generated for any combination of programming language and presentation syntax.

9.8 Declarative transparencies

Application programmers don't want or need to know the details of the engineering required to make their programs function in a distributed system. They only need to control whether and where a particular distribution

transparency is to be present. This can be specified by properly scoped declarative statements.

The engineering of the distribution transparencies which hide these necessary but unwanted details is done by the mechanical application of precisely defined rules. This approach is ideally suited to programs and very badly suited to humans.

It is much more productive and less error prone to do this with an automatic toolset than it is to require all programmers to follow a set of detailed rules. It is also more portable because tools can insert different engineering implementations for different configurations.

More importantly, it is also more future proof, since applications can be adapted to take advantage of improved technologies by updating the toolset rather than the application programs.

9.9 Type checking

In order to build safe distributed systems it must be possible to check in advance that a program will execute correctly. It is beyond our current capability to perform a complete check on the correctness of a (non-trivial) program but we can easily check for the most common kind or error: incorrect interaction between separate components.

The ADT definitions for both ends of an interaction can be checked for conformance to ascertain that the client will not ask a question that the server can't understand and the server will not give an answer that the client can't understand. This check is then applied recursively to all the arguments and results, but note that the conformance check for operation arguments is in the reverse direction.

An implementation of an ADT that provides more operations and/or fewer terminations than are required is a subtype of the required type and will still interact correctly because the extra operations will never be invoked and the extra terminations will never be returned; while the reverse situation will clearly cause problems. The essential principle is that neither end is ever surprised.

The interaction of statically bound components can be completely checked at compile or link time but dynamically bound components will have to be dynamically typed checked or have the validity of previous checks verified.

9.10 Minimising the scope of consequential changes

Conformance is the minimum check needed for correct interaction and provides a continuous subtyping relationship. Equivalence is too restrictive even if subtyping is allowed, since it forces subtypes to be predefined as a set of discrete steps, resulting in redundant edits and re-compilations when unused operations or terminations are changed.

Providing an object with multiple interfaces enables each interface to be specified as narrowly as possible so as to limit the spread of consequential changes to where they are really necessary. For example, the spool interface to a printer (and all its clients) should not need to be changed if the printer's auditing interface is updated.

9.11 Optimization

If the application source code can be completely divorced from the details of the underlying technology and distributed engineering mechanisms then the following benefits can be realised:

- the program can be automatically optimized for its current configuration and required transparencies
- the program can be automatically re-optimized for any new configuration
- the transparencies can be easily changed and the program re-optimized

Thus, both portability and flexibility are increased. This is crucially important because the most appropriate optimizations will depend on the facilities offered by the supporting technology and because the design process always involves trade-offs between conflicting requirements. The best balance between the requirements is not always easy to predict and invariably changes over the lifetime of an application, particularly with respect to changes in technology and costs.

These benefits will not be easy to achieve even when the appropriate abstractions and tools are in place, because all our programmers have been trained in a culture of efficiency. They will need to be re-educated to avoid (and actively restrained from) introducing premature optimizations into their programs.

10 Related issues

It should be obvious by now that the proposed distribution abstractions are compatible with object-oriented and client-server architectures. However, great care needs to be taken with inheritance, particularly with respect to encapsulation, security, federation and typing.

Strict encapsulation of each object in a distributed system is necessary to mask heterogeneity and to enforce security. Some object-oriented languages have a form of re-active inheritance whereby changes in a super-class are immediately reflected in all its sub-classes. This kind of inheritance breaks the encapsulation of the sub-classes and causes immense security and management problems in federated systems.

Some object-oriented languages also have type systems based on inheritance relationships. These do not allow language heterogeneity and are overly restrictive in that they prevent a client from using a server which provides the required service merely on the grounds that it was not constructed from a particular inheritance graph.

Inheritance can be a very useful development tool, but multiple inheritance graphs are notoriously difficult to align and merge. Inheritance can only be safely exploited as an implementation tool for one (or a set of closely related projects) if it is non-reactive and the type of an interface is defined purely by its signature.

The use of higher level distribution abstractions does not conflict with all the current standardisation efforts going on in ISO and industry. The functionality provided by these standards is certainly necessary and any contribution they can make to reducing unnecessary diversity will be welcome.

Using the abstractions outlined in this paper will enable:

- multiple standards to be supported alongside each other
- the set of supported standards to be changed
- the evolution of the supported standards to be tracked

11 Economics of change

The users' biggest investment is in applications and databases; these have a lifetime of decades. They evolve constantly because requirements and technology keep changing, but incrementally because it is too slow and expensive to keep re-implementing them from scratch.

Again, this situation can be improved and the costs constrained if the application source code is decoupled from the details of the technology and the distribution transparencies are specified declaratively and inserted automatically.

In the short term, connectivity is always more important than efficiency and for the majority of interactions efficiency is irrelevant in the long term as well. If a complete division is made between the applications source code and the underlying engineering and technology, the application program can then be exclusively concerned with algorithms and correctness leaving efficiency (and resource utilisation) to the toolset.

Interaction based purely on ADTs requires very little common functionality to provide interworking; only references to ADTs and strings (to name operations and terminations) need to be exchanged. These interactions will be inefficient because every subsequent interaction with an argument or result will involve a call to wherever it is implemented; but the decision to improve efficiency by investing in the design and construction of compatible data types whose values are exchangeable can then be made purely on a cost/benefit basis.

Past investment in existing applications poses a big problem for an organisation moving to (or changing the way it implements) distributed systems. There is no way of automatically carving up and distributing a non-distributed application; But it can be wrapped up as an Abstract Data Type and made remotely accessible so that new applications that require to interwork with it can be distributed. This can even be done without access to the source code, but may require the wrapper to emulate a terminal or another application.

This wrapping technique can even be used to enable a new distributed database to masquerade as the old non-distributed database to old applications.

In these ways the abstractions developed to hide heterogeneity in distributed systems can be exploited to provide backwards compatibility and smooth out the evolution path.

12 Risk factors

The two main risks that any organization using computers faces are:

- getting out of step with your suppliers and customers (and therefore your competitors)
- becoming locked into an inappropriate technology

Both of these risks can be avoided by implementing all of your interfaces as Abstract Data Types. There will still be some costs involved if the rest of the world changes but this will be confined to updating your toolset and developing some wrappers.

Finally, at every stage of the program development process it should always be remembered that any premature optimization to the current technology will inevitably reduce portability and flexibility, thereby increasing risk. The tools are not only there to do the optimization automatically, but also to enable it to be left as late as possible.

13 Productivity

Increases in productivity will come from three main factors:

- greater use of automation
- more pre-execution checks
- better adaptability (and re-use)

The abstractions proposed in this paper increase these productivity factors.

14 Conclusions

This paper has argued that:

- most of the world's computers are being interconnected
- the hardware is available and affordable
- but current program development methods can't cope with the explosion of complexity in the software
- standardization of protocols and APIs is necessary but not sufficient
- it is time to increase the degrees of abstraction and automation
- the required abstractions and tools are well understood
- automation can only be applied to well understood programs
- application programs can be constructed to be completely independent of their configuration, distribution engineering and supporting technology
- new (inefficient) interconnections can be established at low cost
- optimizations can then be implemented on a pure cost/benefit basis
- automatic tools can cope with evolution and re-configuration
- backwards compatibility can be achieved by wrapping up non-distributed applications for remote access
- abstraction and automation will reduce costs
- abstraction and automation will reduce risks
- abstraction and automation will increase productivity