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

The Challenge of ODP

Andrew Herbert

Abstract

This Technical Report is a revision of an invited paper for the Berlin ODP Conference, October 1991.

This paper reviews the technical challenges that have to be overcome if Open Distributed Processing (ODP) standards are to achieve the goal of enabling interworking of applications and sharing of data across computer networks spanning organizational and national boundaries.

The choice of abstract data types for the ODP computational model and the provision of selective transparency in the ODP engineering model ares justified as the foundations which permit practical and efficient ODP systems to be built.

APM.1016.01 **Approved** 24 May 1994

Technical Report

Distribution:
Supersedes:
Superseded by:

The Challenge of ODP

The Challenge of ODP

Andrew Herbert

APM.1016.01

24 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 Introduction

The document reviews the technical challenges that have to be overcome if Open Distributed Processing (ODP) standards are to achieve the goal of enabling interworking of applications and sharing of data across computer networks spanning organizational and national boundaries.

These challenges are used to give a rationale for the organization of the Basic Reference Model of Open Distributed Processing (RM-ODP) as a set of five languages, called the computational, engineering, enterprise, information and technology languages respectively, for describing different views of distributed systems.

The choice of abstract data types as the basis of the ODP computational language and of selective transparency as the basis of the ODP engineering language are justified as foundations which permit practical and efficient ODP systems to be built.

The enterprise and information languages are introduced as the means to make the link between technical solutions and user requirements.

The technology language is not discussed in this document: it is the means for relating specifications in the other viewpoints to conformance statements about the products used to assemble an ODP system.

The document is written for a general technical audience: it does not assume prior understanding of either ANSA or the RM-ODP.

This document is a consolidation of the work of both the ANSA Team and ISO/IEC JTC1 SC21 WG7. The structure of the document is based upon a presentation written by David Otway of GEC-Marconi.

This Technical Report is a revision of an invited paper for the Berlin ODP Conference, October 1991. The revision aligns the description with progress in the development of ANSA and ISO 10746-3 Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model.

2 The scope of ODP

The scope of Open Distributed Processing (ODP) can be summarized as networked, computer-based systems that:

- have many vendors
- are implemented using heterogeneous technologies
- scale to sizes larger than the telephone system
- can evolve gracefully
- enable applications to interwork and share data with one another
- reduce development and operating costs.

The requirement for ODP to be supported by many vendors arises from the fact that successful distributed computing is dependent upon standards for the components that provide the infrastructure across which applications interwork. This infrastructure needs to be universally available and therefore must have an open specification. By implication, the open specification will make the distributed processing infrastructure a commodity available to all, rather than a unique attribute of a limited number of suppliers. In consequence it must be economical to put together and maintain.

Heterogeneity is inevitable given history. Evolution is inevitable given the pace of technological development. Evolution reinforces heterogeneity. Users will want to build ODP systems using the components they already have, and to grow these systems by adding new components and functions as they become available. Both the new and the old components will be required to interwork. Heterogeneity embraces computer architectures, network architectures, programming languages and application-building tools.

The issue of scale is an important one: while initially ODP systems may be small, they will grow by interconnection to other ODP systems. The limit to this growth is the available telecommunications connectivity in the world and, in consequence, the size of the ODP network will grow to meet the size of the telephone system and perhaps become larger.

For many years the telecommunications industry has prided itself on running the world's largest distributed application continuously and successfully. The same expectations will exist for the ODP network, which will have the additional burden of having to support many different applications and be more flexible in its approach to management. Development of the ODP network will be ad hoc: there will not be a central design or management authority and many of the people involved will not recognize the potential for linking and scaling up the systems they deploy.

Incremental change and continuous operation will be inevitable for ODP systems: it will be impossible to stop an entire system to repair bits of it. Users will not be able to afford the investment required to replace systems wholesale.

Interworking between applications and sharing of data between organizations is the primary goal of ODP: this is the functionality that users expect. Users have been promised interworking as a benefit of standards for many years, but somehow the promise has never come true. There is currently a tension between communications based integration on the one hand (e.g., via Open Systems Interconnection - OSI) and applications portability on the other (e.g., via Open Systems Environment - OSE) as ways to achieve integration. With the ODP Reference Model, the standards community now has a framework for understanding how interconnection and application portability can come together and be combined with a distributed processing infrastructure to come closer to the user's need.

To build ODP systems it will be necessary to increase the ease with which applications programmers can exploit distribution: distributed system technology cannot be locked away in the hands of a few networking and operating system specialists. This will only come about if ODP systems are based on simple, easy to understand models and if the ODP interfaces are narrow and well-structured. Huge application program interfaces (APIs) and a myriad of protocol options covering every possibility are not the best solution: the search must be for a small set of general, powerful ideas than can be combined over and over again in diferent ways to build the functions and structures that applications programmers need.

3 Distribution transparency

The central new requirement that arises in distributed systems is that of masking from applications the differences in mechanisms used to overcome problems and details of distribution. This is called **transparency**.

Distribution cannot be denied: applications programmers will have to deal with the possibilities of concurrent access to shared resources, variable latency in accessing resources and persistent failures disrupting access to resources. There are many different techniques for overcoming these problems, representing different trade-offs between for example consistency and availability, security and openness, strict management and flexibility, abstraction and fine-grained control. The architectural aim has to be to minimize the extent to which applications are polluted by detailed knowledge of the individual techniques. Sometimes applications will want to exercise control over distribution or participate directly in its provision. Transparency must therefore be **declarative**, **selective** and **modular**.

Transparency is obtained by matching transparency attributes for applications to an enhanced infrastructure for application software, containing appropriate mechanisms, positioned above the low level operating systems and communications facilities.

The details of distribution that should be masked by transparency are:

- the physical separation of components and the physical concurrency of execution amongst them
- heterogeneity of the computers, programming languages and networks involved
- knowledge of the physical configuration of components within a system
- boundaries between administrative domains within systems.

The details of distribution which should be simplified by transparency are:

- replication of components to increase reliability
- migration of programs or data to balance loads and reduce access times
- checkpointing followed by recovery at alternate locations to mask faults
- specialization of components to enhance performance of specific functions.

4 Designing for distribution

Transparency becomes possible if interfaces between applications and between applications and their supporting infrastructure are simple and uniform. These interfaces must respect a number of technical constraints contingent upon the details of distribution outlined in the previous section.

4.1 Separation

Because of the potential separation of one component of a distributed system from another, all access between components must be based on the exchange of request and response messages - in programming terms a procedural style of interaction. In any interaction, the sender of a request is called the **client** and the recipient the **server**.

The procedural style is selected rather than a data sharing paradigm since sharing is hard to scale,presents problems with respect to failure handling and managing concurrent access, and requires agreement to common representations which restricts heterogeneity and evolution.

The procedural style is selected rather than a pure messaging paradigm since the response carries an end-to-end guarantee of service. The procedural style can be used to implement queues as services accessed through a procedural interface.

The procedures provided by the server give access to a data structure (or a device or some other local resource) that is possibly remote from the client: thus the client is effectively referencing a <procedure, data> combination, rather than just a procedure by itself. Often there are several procedures that can be applied to the same body of data: together these procedures define a self-consistent set of **operations** providing a consistent **service**. The point of access to those operations is termed an **interface**.

The technical term generally used for a set of operations which encapsulate data is an **abstract data type** (ADT). To ensure distribution transparency, all accesses to data structures should be made by references to the interface of the ADT rather than to the data itself; such indirect addressing allows the appropriate communications capability to be inserted transparently in the path between client and server should they be physically separate.

In many programming languages abstract data types are represented as objects; however the object concept often carries other connotations with it that distract from the central requirements of ODP, which can be ignored, except when they conflict with the rules of the RM-ODP. (These are discussed in TR.18 *Distributing Objects*.)

The application programmer has to be prepared for several features of distributed systems which, although often present in local systems, have not been regarded as significant, namely:

- invocation is asynchronous and many clients may be attempting to use a service at the same time; concurrency is the norm in a distributed system and program executions are truly overlapped
- latency is variable: invocations may be delayed due to the distance of the client from the server, or because of transient communications problems; a server may become heavily loaded and respond slowly in consequence
- catastrophic failures may occur which cannot be masked by transparency mechanisms: a computer may fail for an extended period; a critical network link may be broken.

It would be intolerable if applications programmers had to deal with these details at every point: the benefit of transparency is that such things can be deferred to a level where measures intended to deal with 'errors' perpetrated by their users provide the required resilience, for example to deal with entry of incorrect data. In an ODP system the programmer has to think harder about error handling not just within an application itself, but also within the context of the larger system to which the application belongs.

Transparency mechanisms reduce the probability of errors occurring - but they cannot guarantee that things will always work perfectly. The application designer has to consider application-level resilience to concurrency, latency and failure. Adopting a model in which all interaction is potentially remote ensures the programmer recognizes this point and leads to safer designs than those which treat remote procedure call as an extension of local procedure call into a distributed environment.

Along with understanding how to handle errors, the ODP application programmer should also be prepared to exploit parallelism to overcome communication delays and to make full use of the multi-processing capability of a distributed system.

The requirement that application programmers learn how to exploit distribution is a potential inhibition to the success of ODP. The inhibition can be reduced by increasing the potential for checking program structure for correctness and automatic generation of low-level detail. A related problem exists at the higher level too: current approaches to application design assume a model of a central self-consistent database at the hub of a set of related applications programs. Design methods need to be extended to cope with multiple sources of data and a distributed set of applications under separate autonomous management.

The challenges in this context are first to provide application designers with the necessary tools to model, manage and program distributed applications and second to define a simple but comprehensive application programming viewpoint on ODP.

4.2 Federation and heterogeneity

Distributed systems arise as people try to interconnect previously isolated applications for the purpose of sharing existing information, applications or computer resources. Then applications are built using the resources of the interconnected system and further applications are connected in. At each stage of connection decisions have to be taken about which local controls are given up in the interests of interworking and which ones are retained. The end result is a **federation** of interworking, autonomous systems.

Organizations have to be able to apply their own management policies and make their own design trade-offs for the systems they own. There is no way to prevent them from doing so or of enforcing conformance to standards. Therefore it is unreasonable to assume a hierarchical management structure in which some organization give management authority of themselves to others. The reality is that of peer-to-peer federations of organizations intercting with each other according to agreed contracts and retaining their autonomy within the federated structure. A consequence of this requirement is that approaches to management and naming in distributed systems that depend upon strict hierarchies cannot span federations.

Within each federation there may be islands of homogeneity where similar computers are used, the same protocols exist and the same programming language is available. It is less likely this homogeneity will exist over the whole federation, since systems installed at different times and for different purposes are likely to select different technologies. Therefore, in the general case, a client cannot predict the format of remote data or which data types are supported by network protocols; to preserve distribution transparency, the client should be saved from requiring knowledge of the format of any data, local or remote. This requirement is met by the use of indirection and abstract data types.

At the boundaries between organizations there will necessarily be gateways to enforce the security and accounting policies of each organization and oversee the interactions between them. The gateways, or **interceptors,** can also take resposibility for translating between differences in protocol used to support client-server interaction across the boundary.

For interception to be economical, there must be a commonly accepted standard for interworking. This standard must be kept to a minimum in order to increase the chances of adoption. It must be policy independent and support federation.

Abstract data types provide a suitable interworking standard since they provide a high level of abstraction and hence technology independence: they require only that clients be able to invoke ADT operations on servers and that clients and servers be able to pass references to ADT interfaces as arguments and results.

The move to open systems based on standards is often presented as the solution to the problem of heterogeneity: such a view of standards as route to homogeneityl is over simplistic; a single set of standards its unlikely to ever be agreed. Even if this were acheieved it would be impossible to arrange an overnight world-wide change-over. For the foreseeable future it will be necessary to live with differing technologies. The challenge is to develop the means to build bridges between them automatically.

4.3 Configuration

There are many features of distributed systems that require considerable flexibility in the means of configuring of system components (growth, enhancement, fault avoidance, load balancing):

- to change configurations dynamically, indirection (i.e. late binding of clients to servers) is essential; world-wide system builds and reboots will not be practical

- for maximum safety, all accesses must be type checked; to achieve this in a dynamic system, it must be possible to find out the description of any component on-line; early type checking reduces the risks of unpredictable behaviour - it requires that type checking be an integral part of the configuration process
- for maximum flexibility, all components, both large and small, should be equally configurable and manageable.

The configuration requirements are for strong type checking and dynamic configuration: abstract data types and interface references provide these. Furthermore, a pre-existingbody of data or legacy application can always be encapsulated behind an ADT interface, thereby adding the capability for the structure to interwork and be configured as any system component.

4.4 Computational view

The analysis of separation, heterogeneity, configuration and federation demonstrate the requirement for a programming model constrained by the intrinsic properties of distributed systems. Abstract data types are the best foundation of such a model.

The principles of ADTs are very simple:

- 'state' is represented by references (distribution transparent 'pointers') to ADT interfaces; primitive data types such as integers and strings can be modelled as ADTs as well as complex types such as bank accounts and databases
- 'values' are ADTs that only provide operations to access state; 'variables' are ADTs that provide operations to both access and modify state
- all arguments and results are passed by copying references to ADT interfaces: the client and server thereby share access to the interface
- a set of abstract data type implementations which have common state need to be encapsulated so that the state is protected.

This programming model is a simple high-level view suitable for applications programmers: it abstracts out distribution detail, whilst exposing major distribution properties. It gives the application programmer the potential to write applications that are neutral to the kind of transparency provided in the environments where they operate so that the application can be ported between different environments. If the application does have specific environmental constraints, such as dependability or performance guarantees, these can be specified declaratively. The application does not have to be bound to a specific transparency mechanism, or even to an interface to a class of transparency mechanisms. This gives the freedom for the application programming interface and the systems interfaces to evolve independently.

The ODP computational language is a formalization of a programming model based on ADTs suitable for specifying distributed applications.

4.5 Engineering view

Clearly a simplistic implementation of abstract data types would be very inefficient, because of the amount of indirection implied and the overhead

associated with abstract representations of frequently used types such as integers and characters.

Several optimizations are available to make significant improvements in performance:

- direct local access can be used for co-located data - trading off flexibility and portability against performance; in many cases it is easy to identify data which can be co-located conveniently: for example all the paragraphs that make up a document
- compilers can use efficient formats for data, regaining the efficiency of non-abstract languages like C and FORTRAN. If the program is subsequently moved to a new environment it can be recompiled in the context of an alternative compiler back-end that generates the data formats of the new environment; importantly, the source remains unchanged
- objects which have constant state can be copied without breaking computational semantics - the copy will behave identically to the original and be indistinguishable from it. Abstract types corresponding to everyday data such as integers, booleans, strings and so forth all have this property. Consequently such types can be copied across network links that support concrete representations of them, in place of interface references.

Thus alongside the abstract programming model, there needs to be an engineering model of how to organize a system for transparency, and how to optimize implementations where the aplication allows it and transparency is not required. Transparency is achieved by linking transparency mechanisms into the access path to an interface so that effects due to distribution are filtered.

From a computational viewpoint, transparency requirements are expressed asenvironment constraints within interface specifications. The environment constraints detail the properties required of the engineering infrastructure that supports those objects. For applications structured according to the rules of the computational language, transparency requirements can be processed automatically by editing the code generated when programs are compiled to add the extra functionality needed to achieve transparency.

These same techniques can be extended to engineer other qualities and constraints: for example to impose a synchronization discipline over the dispatching of the operations in an interface.

Thus ODP is concerned not just with runtime structures and protocols, but also with the tools used to assemble, compile and link programs. The declarative approach relies on specifying application programs (or at least those parts of them affected by distribution issues) from a computational view and using automated tools to transform this abstract form into an engineering implementation. Such an approach has significant benefits:

- applications are easier to write because distribution is declarative. Programs are labelled with constraints requesting that particular guarantees be applied to selected interfaces rather than mixing application code with calls to low-level system procedures. The engineering is separated from the application
- applications are less error-prone because distribution details are automated. Once the automated tools have been debugged by systems

experts they can be used safely by applications programmers with little systems knowledge

- programmers are more productive because they concentrate on applications rather than distribution details (and have to write less code)
- applications are more portable because the use of ADTs makes them more technology-independent
- applications are future proof because the rigorous and simple semantics of ADTs will survive automated changes in their representation.

5 Types of transparency

For transparency to be selective, it is necessary to distinguish between transparency options. There are no hard and fast rules for doing so, but the following list of ‘transparencies’ have been derived for the RM-ODP, as a set which separate out significant differences in function:

- access transparency - masking any differences in representation and operation invocation mechanism
- concurrency transparency - masking overlapped execution
- replication transparency- masking redundancy
- location transparency - masking differences in object naming between systems and enabling the continued use of objects that have moved
- failure transparency - masking recovery of objects after failures
- resource transparency - mask changes in the representation of an object and the resources used to support it (e.g. automatic retrieval and storage of objects between volatile memory and a stable object repository)
- migration transparency - masking movement of an object from one application to another
- federation transparency - masking administrative and technology boundaries.

5.1 Access transparency

The requirement for access transparency is met by providing an invocation scheme for abstract data types: a service is presented as a set of operations. Two kinds of interaction need to be supported:

- a request-reply structure (Interrogation) for procedural interaction where activity is temporarily transferred to the invoked interface
- an asynchronous request-only structure (Announcement) for spawning a new activity to perform the requested operation.

For both kinds of invocation, communications quality of service constraints must be specified (either explicitly or by default). In the case of interrogation, failure to meet the constraint can be reported to the invoker, whereas in the case of announcement this is not possible. All arguments and results should be references to (local or remote) interfaces. Each operation should be permitted to have a range of possible outcomes, each one of which carries its own package of results:

- the range of outcomes is needed to provide a means to signal different kinds of failure - this can be extended by the application to signal different outcomes to the client’s request.

- the ability to return multiple results in each outcome is required to minimize latency - without this facility the client would have to call the server over and over again to extract the results one at a time.

For access to be type-safe, there must be prior agreement that the client activity is requesting an operation provided by the server interface. This places a requirement for type checking to be based on interface signature checking: if the interface type includes the operations required by the client (with appropriate arguments and outcomes) it is suitable. (The alternative is to name types and declare type name hierarchies; however this fails to meet the requirements for federation and evolution.)

From a description of the signatures of the operations in an interface, a compiler can automatically generate code to marshal data from the local representation format to a network format and vice versa; it can also generate a dispatcher to accept incoming requests from the network to the application procedures that process them. A binder must be provided in the engineering infrastructure to manage the relationship between local procedures and data and external references to them.

5.2 Concurrency transparency

To mask the effects of overlapped execution it is necessary to augment the interaction model with the so-called ‘ACID’ properties, so that sequences of interactions can be treated as ‘transactions’:

- **atomicity** - ensuring that the effect of transactions is all-or-nothing; this can be achieved by adding ‘succeed’ or ‘fail’ attributes on terminations to select the desired effect of an operation and retaining of versions of object state until the overall fate of a transaction is decided. When atomicity is provided a request-reply style invocation will carry an atomic activity into the invoked operation, whereas a request-only invocation can be interpreted as starting a new, independent activity
- **consistency** - ensuring that transactions leave the object involved in a consistent state.This can be achieved by associating ordering predicates with interfaces, where the predicate describes the permitted sequences of invocations within a transaction
- **isolation** - ensuring that transient modifications to object state due to one transaction are not visible to another. Isolation can be achieved by associating separation constraints with interface specifications indicating which operation and argument combinations potentially interfere
- **durability** - ensuring the result of transactions persist, for example by using replication techniques to increase the stability of storage.

Separation constraints can be interpreted to automatically generate a concurrency control manager which governs access to the ADT interface being made atomic. The concurrency control manager will also control the version store for holding the intermediate results of transactions. Additionally it will need to interact with a deadlock detector so that applications do not hang indefinitely if transactions suffer locking conflicts.

5.3 Replication transparency and object groups

Redundancy in systems can be exploited to enhance both reliability and performance. In a distributed system there is an opportunity to allow the application programmer to exploit redundancy for example:

- replicating services and data so that the failure of a computer or network link will not render the data unavailable
- enhancing availability by spreading the demand for a service over a set of identical services
- using N-version programming to provide a defence against programming errors in addition to hardware errors.

All of these forms of redundancy place a requirement for a client to be able to transparently invoke a group of replicas of a service - in other words the client sees the replicated group as it if were a singleton, but with increased reliability or availability.

To provide such a consistent view, the group must arrange that all the members process invocations from clients in the same order - the members do not have to run in exact lock-step, but they must all do things in the same order if they are to maintain a consistent image of the group state as a whole. Between the members of the group there must be some sort of ordering protocol to agree when received invocations can be dispatched. This ordering protocol should be tolerant of failures in members of the group and of changes of membership of the group so that new members can join and current members can leave.

Such a basic group execution mechanism provides the foundation on which more specific replication facilities can be provided: for example, hot-standby, where one member provides the service, with other members waiting to switch in if the active one fails; active replication where all the members are in service so that there is no fail-over period and more general distributed 'blackboard' structures.

5.4 Location transparency

Location transparency requires that a reference to an interface be usable without requiring a client to know or track the location of a service.

Interfaces in a distributed system may change their location for a variety of reasons:

- an object may be checkpointed and then restarted at a different place after a computer fails
- load balancing may move an object from one computer to another in order to increase performance
- an object may be moved from its current location to be co-located with one of its clients to reduce access time and network traffic
- resource management may cause an object to be passivated when it is not in use - for example by removing it from main memory and putting it on disc (see Section 5.5)
- the apparent location of a replica group will change as the membership of the group changes

- there may be several protocols by which an interface can be accessed, and perhaps several network level names by which it is known
- network management operations may lead to an interface's network level name changing.

Different protocol access paths may exist either because of heterogeneity in the system, or because different protocols provide different qualities of service in terms of bandwidth, error handling and so forth.

All these things have to be represented in the description of the location of an interface in a way such that the location transparency mechanism in the client does not have to know the server's migration, passivation or checkpointing structure.

To avoid scaling problems, relocation mechanisms should only require the registration of changes in location because the majority of interfaces in a system can be expected to be temporary and stationary.

5.5 Migration, resource and failure transparency

The computational model outlined in Section 4.5 requires that interfaces persist while they are accessible from the current activities in a system. In engineering a distributed system, interfaces may have different forms optimized for space utilization versus speed of access; for example objects may migrate between machines to balance load or reduce communications latency, taking their interfaces with them. Objects that are not actively in use may be transferred from the execution environment to storage (e.g., in a database or file system). Objects may write snapshots of their state to storage and log interactions so that the object can be reinstated an alternative location after a failure.

Migration, resource and failure transparency management policies may lead to interface instances being requested to move from one location to another and undergoing a change in representation. They require atomicity to ensure consistency. Consequently there is a great deal of sharing of mechanism possible between the several transparencies possible. Transparency is therefore an effect rather than a mechanism.

An object has to take the responsibility for moving itself and its interfaces, since this provides for the opportunity to represent its state in a more compact or resilient form than if the data space of the active representation was simply copied out. It also allows the object to delay the migration until a time convenient to other activities using the object. This is an instance of a general principle that objects should manage themselves, itself a specialization of the concept of encapsulation.

It may well be that operations to snapshot and move an object are provided by an automated tool that scans an object specification and generates appropriate code, in effect providing the desired transparency.

In the case of migration transparency, the snapshot is moved to another location and immediately re-activated.

In the case of resource transparency, the object is moved not to another active location, but rather to a storage device for later retrieval an activation. This passive location can be advised to the relocation mechanisms and subsequent reactivation made transparent to clients of the object.

Finally, in the case of failure transparency, the snapshot must be associated with a log of outstanding interactions, so that when recovery occurs, the replacement object can mirror exactly the state of its predecessor.

5.6 Federation transparency

Federation transparency is concerned with crossing boundaries: either technological ones or or administrative ones. In either case some kind of interception of interactions across the boundary is required.

For a technology boundary the interceptor must stand on the boundary itself and translate between the two domains. The translation may be simple conversion, or it may be that the interceptor has to set up proxy objects in each domain that stand as representatives of objects on the other side of the boundary.

For an administrative boundary the interception may occur within the interaction computers themselves, checking permissions and exchanging adminstrative data. In this context it is natural to think of the interceptor acting as a 'guard'. This is discussed further in section 7.1 on Security.

6 Configuring distributed systems

Clients within an open distributed system need to be able to find out which services are offered by servers. (Some applications may be both client and server simultaneously). This process is called **trading**. Servers describe the services they provide (the types and properties of their interfaces) and the locations of each interface. Clients describe the type and desired properties of services they want to use to a trader, which in turn supplies the client with references to suitable servers.

The requirements on trading are that

- the set of service offers should be structured so that separately administered portions can be clearly identified
- that service offers can be qualified with properties to distinguish them
- that a client is only told of service offers which provide at least the operations it requires (otherwise the trading would breach the type safety guarantees implicit in the computational model).

Trading is intimately concerned with type-checking: a trader needs access to descriptions of the types of the services it offers: it may be convenient to gather these description up within a type manager. The type manager can impose additional constraints on type matching beyond those implied by the type system of the ODP computational language. Taken together, traders and type managers provide within an ODP system a description of its capabilities: self-describing systems are more open-ended and scale better than those which have a fixed external description.

Federation requires cross linking of autonomous traders: such a structure is inevitably an arbitrary graph, and therefore names are potentially ambiguous, since their meaning depends upon where they are interpreted: there is no canonical root. The ambiguity can be overcome by extending names with information about how to get back to their defining context whenever they are sent as argument or results - this is called **context-relative naming**. Within any particular federation, by agreement on common conventions and a management structure for policing them names can be made unambiguous across the whole federation and contextual information only has to be added to names that cross the borders of the federation.

Trading will sometimes need to be linked to resource management: for example it may be useful to activate a passive object if one of its interfaces has been imported by a client. However it is undesirable to build a particular resource management strategy in the trading system itself and therefore it must be possible to link offers to a resource manager which can take what ever actions are required when the offer is selected during a trading operation.

A related aspect of configuration is setting up the initial configuration of servers in a system. This requires the provision of a node manager for each computer in an ODP system which links the computer into the system after a restart, creating any servers on that machine which are required by default

and advertising them via the trading system. This node manager can be extended to provide a management service, accessible from other computers, for starting and stopping servers on its own node.

7 Other issues

7.1 Security

Security in a distributed system is founded upon trusted encapsulation and the management of shared secrets between objects. Encapsulation gives confidence that information within an object is secure. Shared secrets provide the basis for authenticating interactions and achieving integrity and confidentiality.

In simple terms security involves three parties: the client, the server and the supporting infrastructure. The client has a policy about who it will interact with, similarly the server and the infrastructure will have a policy about which interactions they are prepared to allow.

The client can impose its policy directly by choosing which services to use: by sharing secrets with those services authentication, integrity and confidentiality can be achieved.

The infrastructure can intercept both requests to set up bindings between clients and servers, and individual interactions, although the latter is much harder since the infrastructure will need to include much of the transparency mechanism within its security perimeter to access arguments and results. This is contrary to the accepted wisdom that security infrastructures should be minimal.

A server can decide whether to process an interaction or not. Shared secrets with the client can be used to achieve authentication, integrity and confidentiality.

In an ODP system an interface reference for accessing an object cannot itself be secure - the engineering mechanisms for relocation, migration, replication and so on need to be able to read and modify references. It is possible for any object to assemble a reference, therefore a secure object must check that any access is from a valid source.

This checking is another example of the kind of engineering detail which can be generated automatically from a declarative statement of security policy. For each interface of the object, a guard can be generated to police use of that interface. The guard must be included within the encapsulation boundary of the secure object so that interaction between the object and the guard are secure.

An application (or its guards) may choose to devolve some of the checking to other objects, providing, for example, authentication and authorization services, but these services are only advisory in nature since the object may ignore their verdicts.

7.2 Multi-media

Multi-media capabilities are becoming increasingly important and will be an important part of ODP systems. Multi-media brings questions of how to handle complex data types, how to handle synchronization between streams of voice, video and data and how to integrate the switching and interconnection of such streams into the trading and binding architecture for services described by ADTs.

This can be done by regarding the client and server operational interfaces described so far as a special case of a more general interface concept of a **stream interface** which represents a point at which any form of interaction to occur, including continuous flows such as video. A stream is described in terms of its type and its quality of service requirements. A stream interface can be traded and passed in arguments and results just as an operations (i.e. ADT) interface: there is however no means for ADT style interaction at a atream interface.

Operational interfaces can be implicitly bound - holding an interface reference is sufficient to enable interaction. For streams a means of explicit binding must be defined. Explicit binding is parameterized by a template specifying which information flows are enabled between the various interfaces being tied together. It may be that the flows need to be controlled or that events occurring within the streams should be monitored: this is made possible by arranging that the binding process produces an interface containing control and management functions.

The concept of stream interfaces and explicit binding is extremely powerful: it can be used to model many other structures than multi-media ones, for example groups of sensors interacting with a control system, conferencing applications and so forth.

7.3 Garbage collection

The ODP computational model is based on interfaces to objects being accessed via references: this implies that objects must persist for at least as long as there are clients holding references to their interfaces. This potentially puts a server's resources at the mercy of its clients. There are several ways to overcome this:

- provide a means to explicitly close an interface: subsequent attempts to access the interface produce an error indication as their outcome
- objects which have been preserved for archival purposes can be progressively moved out to less and less accessible storage media: they can be moved back on demand.

In addition, distributed garbage collection can be used:

- only passive objects need be considered - active ones cannot be garbage by definition
- many of the computers in large distributed systems spend significant periods idle (overnight for example) and can contribute resources towards the garbage collection process
- careful consideration by application programmers of when to use resource, location and migration transparency can also help reduce the number of references to track down.

7.4 Management

ODP requires extension of concepts of network management to cater for application management. Traditional topics of management such as configuration control, error handling and resource management remain relevant, but become intimately related to providing distribution transparency.

The links to management required for ODP include:

- identification of points where network and system management information can contribute to the provision of transparency
- identification of management interfaces for monitoring transparency mechanisms and changing transparency parameters
- management guidelines about when to select particular transparencies and what kinds of resource management policy to apply.

8 Enterprise, information and technology

This report has focused on the computational and engineering challenges of ODP, since they are a source of constraint on possible implementations. However an ODP system is built to meet a user requirement. A computational and engineering description alone is not sufficient to make the link from technical solution to the user's need. For example, whilst intuition about the names of the operations in an interface may imply something about the service being offered, it is dangerous to do so - the names are just symbols used by the programmer and are not required to be meaningful outside the program.

Two further languages are provided by the RM-ODP - the **enterprise language** and the **information language** - with which meaning and purpose can be modelled.

The information language builds upon familiar notions of objects, relations and information flows to enable description of the entities relevant to the users of a system, and the interactions between those entities. Information modelling is well-established, however ODP adds a new challenge of having to deal with issues of inconsistency and conflict between multiple versions of the same information held by different parties in a federated environment.

The enterprise language focuses on the ideas of communities (i.e. organizations of one sort or another), roles within communities and the objectives of a community. An understanding of these issues provides the design rationale for placing security and dependability requirements on the components of an ODP system, and for deciding where flexibility should be retained versus where it can be left out. For example mission critical resources should be carefully protected; contractual interactions should be subject to audit and so forth. These issues are separated out from the information language to reinforce the point that there may be many ways to organize information to meet enterprise objectives (and vice versa).

Finally, the components of an ODP system have to be related to standards and products: to this end the RM-ODP includes a **technology language** which provides the conceptual foundations for an ODP conformance tsting methodology by which the suitability of some product to meet an ODP requirement can be assessed.

The complete description of an ODP system (or at the smaller scale, a component of an ODP system) will be an amalgam of computational, engineering, technology, information and enterprise statements. There is no sense of priority to any of the individual statements; they are all talking about the same system, but from different points of view.

9 Summary

The goal of ODP is to provide a framework of integrated standards for building world-wide open computer-based systems. The requirement for ODP is undeniable: achieving that goal is a technical challenge because of the many constraints that limit the technical approach.

The foundations of the RM-ODP are the computational language as the basis for application interface standards and the engineering language to structure the provision and interoperation of transparency mechanisms. These are in turn supported by the enterprise, information and technology languages to relate technical solutions to user requirements.