2

RM-ODP: The Architecture

P.F. Linington

Computing Laboratory, University of Kent,
Canterbury, Kent CT2 7NF, United Kingdom

The Reference Model for Open Distributed Processing is a joint ISO/ITU Standard which provides a framework for the specification of large scale, heterogeneous distributed systems. It defines a set of five viewpoints concentrating on different parts of the distribution problem and a set of functions and transparency mechanisms which support distribution. The resulting framework is being populated by more detailed standards dealing with specific aspects of the construction and operation of distributed systems.

Keyword Codes: C.2.4
Keywords: Distributed Systems

1. INTRODUCTION

The Reference Model for Open Distributed Processing is a standard produced jointly by the International Organization for Standardization (ISO) and the International Telecommunications Union (ITU). Experts from these two organizations have been working together on this framework for Open Distributed Processing (ODP) for some seven years, and the resulting architecture has recently been approved for publication by ISO; ITU approval is expected at a meeting later in 1995.

This work is based on recent research and best practice; it has drawn upon a wide range of experience in the implementation of distributed systems and the formulation of a general architecture. Input has been taken from advanced industrial research, such as that in the ANSA consortium, from various ESPRIT and RACE activities, and from practical experience with platforms such as the OSF-DCE. In the standards world, ideas from OSI management and the security frameworks have been incorporated. Major input from the ITU has introduced the requirements from TMN, INA and TINA. More recently, the work has benefited from a strong, two-way liaison with the Object Management Group.

The resulting standard is a framework, which documents key decisions, relates components and sets the technical agenda for future, more detailed, standardization. The Reference Model for Open Distributed Processing (RM-ODP) is made up of four parts [1-4].

K. Raymond et al. (eds.), *Open Distributed Processing*

These are

Part 1: **Overview**: this contains a motivational overview of ODP, giving scoping, justification and explanation of key concepts, and an outline of the ODP architecture. It contains explanatory material on how the RM-ODP is to be interpreted and applied by its users, who may include standards writers and architects of ODP systems. It also includes a categorization of required areas of standardization, expressed in terms of the reference points for conformance identified in Part 3.

Part 2: **Foundations**: this contains the definition of the concepts and analytical framework for the description of arbitrary distributed processing systems. It introduces the principles of conformance to ODP standards and the way in which they are applied.

Part 3: **Architecture**: this contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards must conform.

Part 4: **Architectural semantics**: this contains a formalization of the ODP basic modelling concepts defined in Part 2. The formalization is achieved by interpreting each concept in terms of constructs of the different standardized formal description techniques.

The current paper concentrates on Part Three of the RM-ODP — the architecture which makes a distributed system be specifically an ODP system.

2. MODELLING FOUNDATIONS

The architecture is supported by a set of modelling concepts which provide the foundation for expressing it. These concepts are object based, and are very general; they can be applied in many different areas of the architecture.

The most basic concepts defined are those of object, action and interaction; an object encapsulates its state, and this state can only be modified by interaction with other objects or by the internal actions of the object. The interactions between objects take place at interfaces; an object can have any number of interfaces. Interfaces can be located at particular points in space. These interfaces are the basis for the description of configurations of objects and the transfer of information about the availability of objects.

A second set of concepts supports the structuring of specifications, introducing ideas of composition and refinement, type and class and of the instantiation of objects or interfaces from templates which describe them. Following this, a variety of more abstract organizational concepts, such as domain, contract and liaison are introduced to express relationships between objects.

Finally, there is a basic framework for the definition of conformance to the ODP specifications, and for the statement of where such conformance applies. The conformance of implementations to ODP standards can be tested in a number of ways, depending on whether the requirement is for interworking, software portability, correct generation of interchange media (such as removable discs), or correct interaction with human users and the outside world in general.

3. ARCHITECTURAL FRAMEWORK

Distributed systems can be very large and complex, and the many different considerations which influence their design can result in a substantial body of specification, which needs to be given structure if it is to be managed successfully. A good framework should allow different parts of the design to be worked on separately if they are independent, but should identify clearly those places where different aspects of the design constrain one another. There are two main structuring approaches used in the ODP architecture: the definition of viewpoints and the definition of transparencies.

3.1. Viewpoints

The RM-ODP defines five viewpoints. A viewpoint is a subdivision of the specification of a complete system, established to bring together those particular pieces of information relevant to some particular area of concern during the design of the system. The viewpoints are not completely independent; key items in each are identified as related to items in the other viewpoints. However, the viewpoints are sufficiently independent to simplify reasoning about the complete specification.

Each of the viewpoints in the set can be related to all the others. They do not form a fixed sequence like a set of protocol layers, nor are they created in a fixed order according to some design methodology. The architecture is expressed in terms of the complete set of related viewpoints, without laying down how this complete specification is to be constructed.

The five viewpoints defined are:

a) the enterprise viewpoint: a viewpoint on the system and its environment that focuses on the purpose, scope and policies for the system.

b) the information viewpoint: a viewpoint on the system and its environment that focuses on the semantics of the information and information processing performed.

c) the computational viewpoint: a viewpoint on the system and its environment that enables distribution through functional decomposition of the system into objects which interact at interfaces.

d) the engineering viewpoint: a viewpoint on the system and its environment that focuses on the mechanisms and functions required to support distributed interaction between objects in the system.

e) the technology viewpoint: a viewpoint on the system and its environment that focuses on the choice of technology in that system.

In each viewpoint, key terminology is established and constraints expressed which characterize the architecture, making the systems described distinctively ODP systems, not just any distributed systems. These terms and constraints are expressed as the initial terms and grammar of a set of abstract viewpoint languages. Specifications expressed in these languages conform to their grammar, and so the systems designed are ODP systems, at least from an architectural point of view.

The various viewpoint languages differ in the strengths of the constraints their use implies. Those concerned with organizing distribution and providing common solutions to its problems (the computational and engineering viewpoints) place a significant number of constraints that must be observed, and in so doing give guarantees of interworking between and portability of components. Those which express requirements for the system as a whole (the enterprise and information viewpoints) place fewer constraints. Constraints on the complete system effectively limit its scope, and so make the architecture less general-purpose. Therefore few language constraints have been defined in these viewpoints in order to support as wide a range of applications as possible.

It should be noted that the expression of the architectural constraints in the form of an abstract language does not imply any particular syntax or notation; there may be many notations consistent with the architecture, derived from a variety of programming practices or development methods, and embodying choices not expressed in the ODP architecture itself.

3.2. Transparencies

The second structuring approach taken is to identify a number of transparencies. When contemplating a distributed system, a number of problems become apparent which are a direct result of the distribution: the systems components are heterogeneous, they can fail independently, they are at different and, possibly, varying locations, and so on. These problems can either be solved directly as part of the application design, or standard solutions can be selected, based on best practice.

If standard mechanisms are chosen, the application designer works in a world which is transparent to that particular problem; the standard mechanism is said to provide a transparency. Application designers simply select which transparencies they wish to assume, and where in the design they are to apply.

The transparency approach can lead directly to software reuse. Selection of transparencies in the system specification can lead to the automatic incorporation of well-established implementations of the standard solutions by the system building tools in use, such as compilers, linkers and configuration managers. The designer expresses system requirements in the form of a simplified statement of the interactions required and the transparency properties that they should possess.

The transparencies defines in the RM-ODP are

a) **access transparency**, which masks differences in data representation and invocation mechanisms to enable interworking between objects. This transparency solves many of the problems of interworking between heterogeneous systems, and will generally be provided by default.

b) **failure transparency**, which masks from an object the failure and possible recovery of other objects (or itself), to enable fault tolerance. When this transparency is provided, the designer can work in an idealized world in which the corresponding class of failures does not occur.

c) **location transparency**, which masks the use of information about location in space when identifying and binding to interfaces. This transparency provides a logical view of naming, independent of actual physical location.

d) **migration transparency**, which masks from an object the ability of a system to change the location of that object. Migration is often used to achieve load balancing and reduce latency.

e) **relocation transparency**, which masks relocation of an interface from other interfaces bound to it. Relocation allows system operation to continue even when migration or replacement of some objects creates temporary inconsistencies in the view seen by their users.

f) **replication transparency**, which masks the use of a group of mutually behaviourally compatible objects to support an interface. Replication is often used to enhance performance and availability.

g) **persistence transparency**, which masks from an object the deactivation and reactivation of other objects (or itself). Deactivation and reactivation are often used to maintain the persistence of an object when the system is unable to provide it with processing, storage and communication functions continuously.

h) **transaction transparency**, which masks coordination of activities amongst a configuration of objects to achieve consistency.

In each case, the definition of the transparency involves both a set of requirements and a solution that satisfies it. The set of requirements states where the transparency is needed (i.e. which interactions it affects). This may simply be a statement that it applies throughout a system, or may be a more selective statement involving specific interfaces, defining, for example, the interactions which make up a transaction or selecting the objects and interfaces to be supported by replication. The solution takes the form of specific rules for the transformation from a specification in which the transparency is requested to a more detailed one which expands selected interaction or objects so as to include mechanisms which provide a means of interaction with the requested properties.

4. THE ENTERPRISE LANGUAGE

The aim of an enterprise specification is to express the objectives and policy constraints on the system of interest. This involves the identification of the main roles involved in the system. These roles represent, for example, the users, owners and providers of information processed by the system. Creating a separate viewpoint to convey this information decouples the objectives set for the system from the way it is to be realized.

One of the key ideas in the enterprise language is that of a contract, linking the performers of the various roles and expressing their mutual obligations. A contract can express the common goals and responsibilities which distinguish roles in a community, such as a business and its customers or an government organization and its clients, as being related in particular ways in a single activity or enterprise.

A federation is one particular kind of community; a federation is a coming-together of a number of groups answering to different authorities (and thus representable as distinct domains) in order that they may jointly cooperate to achieve some objective. Since the evolution of distributed systems will repeatedly result in the merging of existing, separately managed sub-systems to share information or support commercial interests, the creation of

federations and the expression of the rules which are to govern them forms an important part of system specification in the enterprise viewpoint.

Where appropriate, an enterprise specification will also express aspects of ownership of resources and responsibility for payment for goods and services in order to identify, for example, constraints on accounting and security mechanisms within the infrastructure which supports the system.

Different notations for enterprise specification can be expected to support specific organizational structures and business practices, but architecturally, ODP is neutral, requiring only that an appropriate specification be generated; few constraints are placed on the form that organizations should take.

5. THE INFORMATION LANGUAGE

The individual components of a distributed system must share a common understanding of the information they communicate when they interact, or the system will not behave as expected. Some of these items of information are handled, in one way or another, by many of the objects in the system. To ensure that the interpretation of these items is consistent, the architecture identifies the information viewpoint to specify the information to be handled, independently of the way the information processing functions themselves are to be distributed.

The information specification consists of a set of related schemata, just as in familiar data modelling activities. In the RM-ODP, a distinction is made between invariant, static and dynamic schemata. An invariant schema expresses relationships between information objects which must always be true, for all valid behaviour of the system. A static schema expresses assertions which must be true at a single point in time, and a dynamic schema specifies how the information can evolve as the system operates.

These schemata may apply to the whole system, or they may apply to particular domains within it. Particularly in large and rapidly evolving systems, the reconciliation and federation of separate information domains will be one of the major tasks to be undertaken in order to manage information.

Different information specification notations model the properties of information in different ways. Emphasis may be placed on classification and reclassification of information types, or on the states and behaviour of information objects. The approach to be taken will depend on the modelling technique and notation being used.

Since both the information and enterprise viewpoints consider the system as a whole, conformance to them must be assessed as a whole. Sets of observations at any of the points where system components are defined to interact should all be consistent with the requirements expressed in these viewpoints. If a set of observations is not consistent with the requirements of, for example, an invariant information schema, the implementation of the system does not conform to that part of its specification.

6. THE COMPUTATIONAL LANGUAGE

In distinction to the two viewpoints described so far, which consider the distributed system as a whole, the computational viewpoint is directly concerned with distribution. It does not address interaction mechanisms, but it does decompose the system into objects performing individual functions and interacting at well-defined interfaces. The computational specification thus provides the basis for decisions on how to distribute the jobs to be done, because objects can be located independently and communications mechanisms can be defined to support the behaviour at their interfaces.

The heart of the computational language is the object model which defines the form of interface an object can have, the way that interfaces can be bound and the forms of interaction which can take place at them. The computational language also defines the actions an object can perform, so that new objects and interfaces can be created and bindings established. This model provides the basis for specification languages, programming languages and communication mechanisms all to perform in a consistent way, thus allowing open interworking and portability of components.

6.1. Object interaction

There are two basic kinds of computational interface: operation interfaces, which support discrete interactions, and stream interfaces, which support continuous flows. Often, these interactions can be considered as indivisible, and nothing further needs to be said about how the interactions take place.

If, however, details of the progress of the interaction need to be expressed, for example, to define time delays or other aspects of quality of service, then the interactions can be structured into a sequence of signals, each of which has a precise location and time of occurrence. Timing requirements can then be stated in terms of the delay between particular pairs of signals or by more complex timing statements, as necessary. This use of signals as a common representation may in future be used to define new interaction types.

For the present, however, the kinds of interaction supported are strictly limited to those for which efficient communication mechanisms are known to exist. Thus there are only two kinds of operation interaction: interrogations, in which there is a request followed by a reply, and announcements, in which there is a single unidirectional transfer of information. More complex forms, such as rendezvous, which are expensive to provide in practice, are not supported.

Stream interfaces can be made up of a number of independent flows, each with a specified direction, so that a stream might consist of linked audio and video flows, or paired audio flows in opposite directions. Streams were introduced into the architecture primarily to support multimedia interactions, but they can also be used to represent other forms of unstructured information transfer, such as sequences of announcements giving regular updates from some sensor, if the exact repetition rate of the readings is not of major concern in the application design.

6.2. Binding

Before two objects can interact, the interfaces to be involved must be associated by creating a binding. In simple cases, one of the objects involved can perform a **primitive binding** action, linking it to another object. This is adequate for the expression of, for example, a straightforward client-server binding. However, in other situations, it may be necessary to model the binding process in more detail. This requirement may arise, for example, if there is a need to manage the quality of service of the expected interactions, or if the binding is to link more than two objects, supporting some sort of group or multicast interaction. In particular, stream bindings are often set up by third parties, and generally require some kind of explicit control during their lifetime.

To give the necessary control, the notion of a **compound binding** is introduced. In this form of binding (which can be expressed in terms of some piece of object behaviour involving primitive bindings), the originator of the binding instantiates a binding object, which, in turn, is bound to the set of objects which are to interact. The required rules associated with the interactions to be performed are then expressed via the behaviour of the binding object. Whenever a binding object is created, knowledge of a control interface is returned to its creator, so that changes in behaviour or configuration can be requested, or the binding object asked to terminate itself. Apart from the way its creation is parameterized by the set of interfaces to be bound, and its special status in encapsulating and controlling part of the communication function, a binding object is just an ordinary computational object.

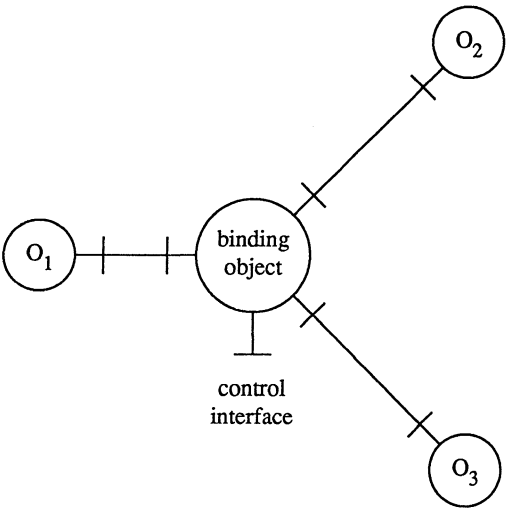

Figure 1 - An example of a compound binding

The behaviour required from a binding object can be quite complex. For example, a number of full duplex interfaces may be linked by a binding object which encapsulates the rules of a conference system for allowing the audio flow from a selected producer (the current talker) to be delivered to all consumers. Varying degrees of application control might be

exercised via the binding control interface to provide explicit floor control. Simpler binding objects might represent full duplex audio links or synchronized audio and video flows.

However, in the simple client-server case, using a default quality of service, the visibility of the binding adds little to, and may complicate, the specification. In such cases, the designer of a computational notation may opt to conceal the whole binding process, providing **implicit binding**. If so, the notation specifies with each interaction being invoked the identity of the other interface involved; it is then left up to the supporting infrastructure to create any necessary binding and either to discard it after the interaction completes or to retain it, for a while, for possible reuse. Some notations may support a mixture of implicit and explicit binding for different interfaces.

6.3. Interface types and subtyping

Every interface has an associated type which characterizes it. One important component of this type is the interface signature, which expresses the static aspects of the interface — the operations available and their associated parameter types (or, for a stream, its flow types), together with the interface's role in the cause and effect sequence of interaction.

The computational language does not require that interfaces participating in a binding be of identical types. To do so would create a barrier to system evolution by making the phased introduction of new services and functions more difficult. Instead, a set of subtyping rules is defined, and the constraints on interface binding expressed in terms of them.

The computational language specifies appropriate type matching rules for each of the kinds of interface defined. The rules are in terms of signatures because they are easy to check during the interaction; behavioural aspects of the interface type may require much more knowledge of the previous history than is available. For operation and signal interfaces, a "no surprises" rule is applied. Interfaces can be bound only if none of the interactions involved introduce unexpected behaviour or fail to supply expected information. However, not all features of the responding party in the interaction need be exercised. Thus, in a client-server interaction, for example, the server may support operations unknown to the client, and thus not used, but the client must not invoke operations which are unknown to the server.

The situation for stream bindings is more complex, because there is an element of application choice in the selection of the binding rules. The computational language requires that flows be coupled only if they are compatible, but admits the possibility of applications in which some flows may be left without a matching counterpart. Thus, for example, one might choose to allow participation in a video conference from a mobile (non-video) 'phone.

6.4. Support for portability

Further definitions are provided so that the computational language stipulates which actions an object can perform (effectively defining an object based virtual machine) and enumerating the possible failure modes of these actions.

A set of portability rules, using the actions defined, identifies the requirements on a computational notation which is to be used to support the portability of objects between different environments. Notations may be referred to as basic or complete, depending on the sets of actions they support.

7. THE ENGINEERING LANGUAGE

The engineering language focuses on the way object interaction is achieved and with the resources needed to do so. Thus the computational viewpoint was concerned with when and why objects interact, but the engineering viewpoint is concerned with how they interact. In the engineering language, the main concern is with the support of the individual interactions between computational objects. It is here that one of the most direct links between viewpoints is found; computational objects are visible in the engineering viewpoint as **basic engineering objects** and primitive computational bindings are visible as channels or local bindings.

7.1. Clusters, capsules and nodes

The engineering language deals with the basic engineering objects and with various other engineering objects which support them. It relates these objects to the available system resources by identifying a nested series of groupings.

At the outer level, objects are physically located and associated with processing resources by grouping them into **nodes**, which can be thought of as representing independently managed computing systems. A node can be anything which has a strongly integrated view of resources, as long as the system designer can consider it as a whole. Thus a tightly coupled parallel processing system can be considered a node, so long as it has one scheduling and allocation policy — one operating system.

The node is under the control of a **nucleus** which is responsible for initialization, for creating groups of objects, for making communications facilities available, and for providing basic services like timing and the source of unique identifiers.

Within a node, there may be a number of **capsules**. A capsule owns storage and a share of the node's processing resources. It can be thought of in terms of a traditional protected process, with its own address space. A capsule is thus the unit of protection and is generally the smallest unit of independent failure supported by the operating system. There is a special object, called the capsule manager, associated with each capsule, and for descriptive purposes, a capsule is controlled by interactions with this manager.

A capsule will typically contain many objects; the grouping of objects into capsules is done to reduce the cost of object interaction. This is because communication between traditional processes is slow and expensive, because of the checks which need to be performed; however, the compiling tools that build capsules can be trusted to validate and structure the interactions between closely related objects to a sufficient extent to let them share resources. Resources within a capsule will be controlled by some kind of language-specific run-time system.

The smallest grouping of objects is into a set of **clusters** within a capsule. The objects in a cluster are grouped together in order to reduce the cost of manipulating them. The objects in a cluster can be checkpointed together, transferred to persistent storage, reactivated or moved to another node altogether. This manipulation of complete clusters as a single operation opens the way to the management of very fine-grain object-based systems at reasonable cost. For example, a geographical information system might consider data about individual points on a map to be objects, but could not sustain the cost of giving each of these objects a completely separate existence. Communication between objects in a cluster can be highly optimized, since the objects are created together, in the same language, and are expected to stay together.

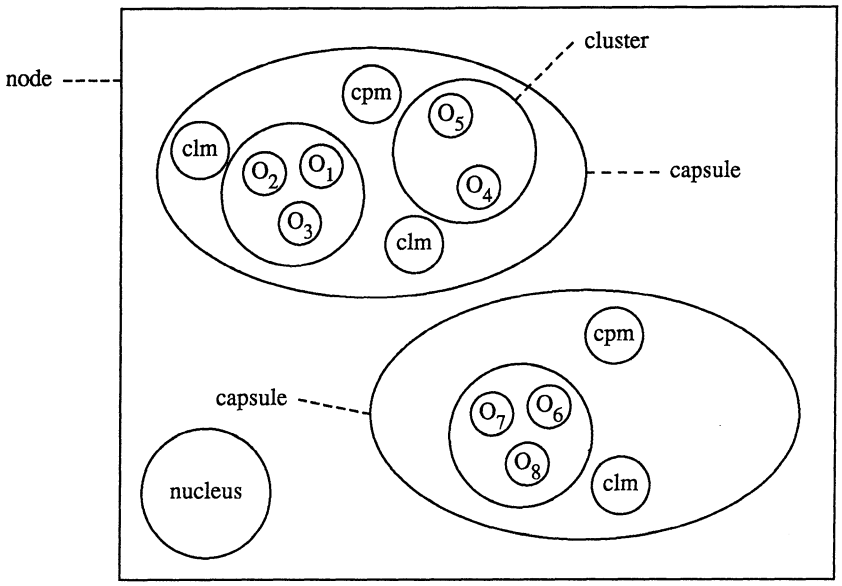

Figure 2 - Clusters, capsules and nodes

Interaction within a cluster might therefore be supported by a simple local method invocation or equivalent.

Clusters are controlled and actions on them initiated by interaction with an associated cluster manager object.

7.2. Channels

When objects in different clusters interact, there is a need for a good deal of supporting mechanism. Even if the objects are currently within the same capsule or node, mechanisms are needed to cope with the possibility of one or other of them terminating, failing or moving elsewhere. The set of mechanisms needed to do this constitute a channel, which is made up of a number of interacting engineering objects.

The objects within a channel can be divided into three types, based on the job that they do. **Stubs** are concerned with the information conveyed in an interaction, **binders** are concerned with maintaining the association between the set of basic engineering objects linked by the channel, and **protocol objects** manage the actual communication.

Stubs interact directly with the basic engineering objects they support, and perform functions such as the marshalling and unmarshalling of parameter, or the logging of information about the interaction being performed. Thus the stubs need access to information about the type of the interaction, or, more generally, the type of the interface that is being supported. This distinguishes them from binders and protocol objects, which transfer complete messages without concern for their internal structure.

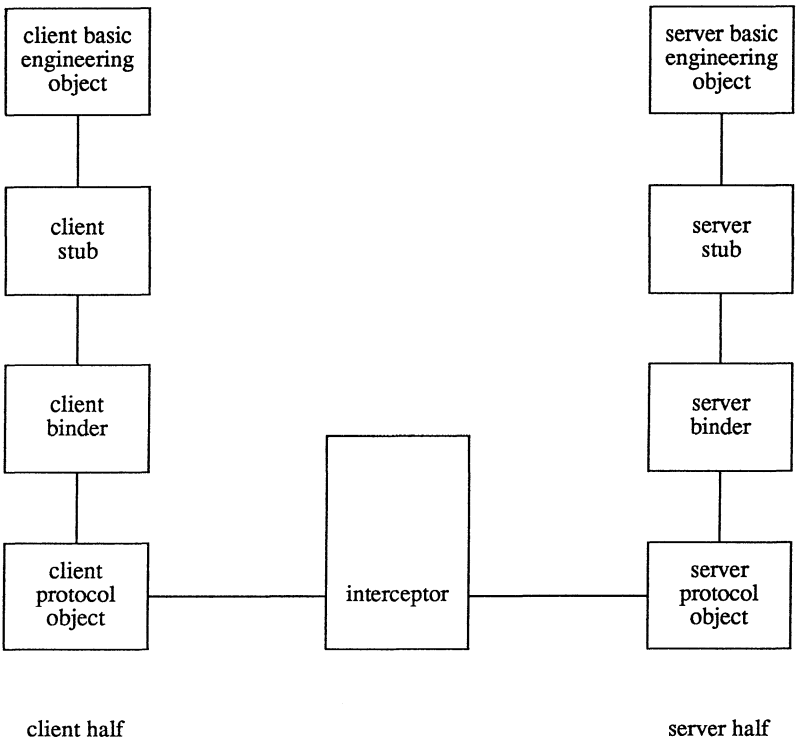

Figure 3 - An example of a client-server channel

Depending on the design of the system, a stub may be directly associated with a particular basic engineering object, or it may be shared between a number of such objects. Sharing will generally imply the need to transfer some additional information to identify, and thus distinguish between, the objects being supported.

Binders need to solve many of the problems of distribution. They are responsible for maintaining the end-to-end integrity of the channel, and so have to handle changes of configuration and communication or object failures. The binder has to establish the binding when the channel is created, and has to keep track of the other endpoints if objects move or fail and are replaced; this is the process of object relocation. The binders are thus involved in the provision of many of the distribution transparencies.

The protocol objects provide for communication of sufficient quality and reliability between the binders they serve. In addition to handling whatever peer protocols are in use, they will provide access to supporting services, such as directory services for translating addresses, where necessary.

Any of these three kinds of engineering object may itself need to communicate with other parts of the system, in order to obtain the information it needs to do its job, or to supply management information to other objects. Such communication may itself need the various distribution transparencies, and so the communication from these objects to elsewhere is by means of a channel; from this point of view, the objects within one channel can play the role of basic engineering objects in another. Similarly, any of these objects can support control interfaces, via which they can be managed. For example, a protocol object may provide a control interface through which the target quality of service for the channel can be adjusted.

In cases where the channel crosses some technical or organizational boundary, there may be a need for additional checks or transformations to match the requirements on the two sides. These functions are performed by **interceptors**, which form part of the channel. They may need to perform format or protocol conversion, or may provide accounting or access control checks. An interceptor may be built up from protocol objects, binders and stubs, depending on the nature of the job it has to do.

For simplicity, channels have been described here as linking two basic engineering objects. However, channels with many endpoints can be defined, supporting various forms of group communication or multicast. In such channels, the binders are responsible for coordinating communication, but the multicast mechanisms may be provided by either binder or protocol, depending on the technology available. Multi-endpoint channels are used to support replication transparency.

7.3. Interface references

When an interface is created, an interface reference for it is generated. The nucleus is involved in this process, so as to make the reference unambiguous, and sufficient resources are allocated and initialized for the objects in that node to participate in bindings if asked to do so.

The interface reference is the key for access to a large amount of information. Given such a reference, it is possible to discover the type of the interface, a communications address at which binding to it can be initiated, and other information about the expected behaviour of stubs, binders and protocol objects within the channel, which is needed for a subsequent binding to succeed. It is also the starting point for calling upon the functions needed to handle errors; knowledge of an interface reference makes it possible to contact an appropriate relocator.

This does not imply, however, that the information is all encoded as part of the interface reference; to do so might make it a very big item to manipulate. The architectural requirement is that there should be some prescription for obtaining the necessary information, starting from the interface reference, but the exact prescription, in terms of decoding and enquiry from other objects, can be chosen differently in different system designs.

In addition to these design variations, there will also be variations arising from the existence of multiple naming domains and the allocation of references with respect to these domains. For both these reasons, it will be necessary for interceptors, or other objects in the channel, to transform interface references when they are passed across domain boundaries.

7.4. Binding

There are two kinds of engineering binding. Within a cluster, or between the objects which cooperate within a node to provide a channel, there are **local bindings**, which are provided by system-specific mechanisms. Such bindings are regarded as primitive in the architecture. On the other hand, the bindings supported by channels provide appropriate distribution transparencies; these are called **distributed bindings**, and creating them will generally involve some interaction between a number of nodes to establish the channel.

7.5. Conformance

The structuring of the engineering specification into clusters, capsules and nodes, and the support of interaction by structured channels gives rise to a large number of interfaces, any of which can be selected as a conformance point, allowing for observation and conformance testing.

The various interfaces can be used to provide the different kinds of conformance. The interface between protocol objects is an interworking conformance point, providing for familiar methods, like OSI testing, based on observation of the communication behaviour. Most of the other interfaces are internal to a node, and represent boundaries between software modules; they are programmatic reference points and allow testing for software compatibility and portability. Some of the interfaces to basic engineering objects may allow other forms of conformance testing, for interchange or perceptual conformance (correct interaction with the real world).

8. THE TECHNOLOGY LANGUAGE

The technology viewpoint provides a link between the set of viewpoint specifications and the real implementation, by listing the standards used to provide the necessary basic operations in the other languages. The aim of the technology language is thus to provide the extra information needed for implementation and testing by selecting standard solutions for basic components and communication mechanisms. Such a selection is necessary to complete the system specification, but is largely divorced from the rest of the design process.

There are consequences of the technology selection, however. One area in which the selections in the technology viewpoint feed back to other aspects of the system design is in the provision of a specific quality of service. The selections in the technology viewpoint determine the performance costs of interactions and thus, indirectly, the quality of service which can be achieved by the behaviour defined in other viewpoints.

The technology language plays a major role in the conformance testing process. It supplies the information needed to interpret the observations a tester can make in terms of the vocabulary and concepts used in the other viewpoints of the system specifications. For example, it allows valid interactions to be recognized, so that their appropriateness can be checked against some specified object behaviour.

9. CONSISTENCY BETWEEN VIEWPOINTS

The five viewpoint specifications constructed must be linked by defining the relations between key terms in them. It is these statements of the relationships between viewpoints that make them specify a single system, rather than being completely independent documents. See [5] for some examples of demonstration of viewpoint consistency.

Many of the links needed will be provided implicitly by the notations used, resulting from correspondences between names. However, some of the key constraints need to be stated explicitly. In the architecture, constraints are placed on the relations between terms in the viewpoint languages themselves, establishing some limits on the mappings which can be established. Most of the constraints placed are between terms in the computational and engineering languages, and are defined so as to create consistent interpretations when system components, such as those supporting the ODP functions, are specified separately.

Clear mappings between viewpoints are necessary if the processes of identifying interfaces and of providing transparencies are to be supported automatically by development tools. For example, a computational object may be realized as a set of linked engineering objects, but a single engineering object cannot represent multiple computational object; a computational interface cannot be divided into separate engineering interfaces supported by unconnected channel structures; computational interfaces can always be identified unambiguously by engineering identifiers. These kinds of constraint help to ensure that common engineering mechanisms will be able to support the full range of possible computational behaviours.

10. ODP FUNCTIONS

In addition to the five viewpoint languages, the RM-ODP gives brief definitions of a number of common functions. Most of these are either introduced in the engineering language to provide support needed for its structures, or form convenient building blocks for the provision of transparencies. Functions are provided by objects, although it is generally left for more detailed standards or individual implementors to decide whether each function is provided by a single object, or several functions by one object, or a function provided by a set of interacting objects.

The specification of how one of these functions is to be provided may be complex and may itself needs to structured. It amounts to the design of a small, special purpose distributed system, in which the interactions between the object providing the function and objects having roles which relate to it are defined. It is natural, therefore, to structure the specifications by using the RM-ODP framework, considering the provision of the function as a small enterprise with its own information and computational models.

10.1. Management functions

Management functions are needed to control the lifecycle of objects and of the various groupings of objects identified in the engineering language. For each of the management functions, there will be a corresponding management interface type, the details of which will depend on the kind of grouping being managed. Management functions exist to control individual objects, clusters, capsules and nodes.

For individual objects, a management function can request checkpointing of the object's internal state or deletion of the object. Object checkpoints are combined into cluster checkpoints, which are essentially templates for recreating the cluster in the state it had when the checkpoint was taken, if necessary. Cluster management is primarily concerned with using this information to deactivate, move, reactivate or recover clusters. These activities form the basis of a number of the transparency mechanisms.

The capsule management functions perform a similar job, but at a coarser level of granularity, controlling the resources of the cluster as a whole and instructing the capsule managers to create or remove their clusters.

At the coarsest level, the node manager controls the basic resources of the node, and makes allocations from them on request. It handles execution resources such as threads and timers, communication resources and naming responsibilities. It creates new capsules when necessary.

10.2. Coordination functions

The second class of functions is concerned with the coordination of distributed activities and the management of distributed groups of objects. These functions support various kinds of consistency and information dissemination mechanisms.

The first coordination function deals with event notification — the establishment of objects which maintain historical records of which events have happened and take responsibility for informing other objects of when events occur. The event notification function thus makes it possible for a coordinated group of objects to maintain an interest in the occurrence of selected events in a consistent way.

The next set of functions is concerned with the coordinated maintenance of checkpoint records of the state of the clusters which an application depends on. The architecture distinguishes checkpointing and recovery functions, which are concerned with keeping and using records, from deactivation and reactivation functions, which control the activity of interest directly.

The group function coordinates some number of objects which are participants in a multi-party binding; it is concerned with group membership, distributed management of group interactions, and combination of results to ensure a consistent outcome. It has a specialized form supporting exact replication of a number of objects, coordinating their interactions such that they remain replicas of each other at all times.

Built on these is the migration function, which supports the movement of a cluster from one capsule to another. Migration can be achieved in one of two ways, depending on the performance requirements. It can be achieved by deactivation followed by reactivation in the new location, or it can be achieved by replication of a new copy in the desired location, followed by deletion of the original copy. The first involves less communication, but the second provides a more continuous service.

Sequences of actions can be coordinated to achieve a consistent result by using the transaction functions. Both a general transaction function and a specialization of it to provide ACID properties are defined.

Finally, an interface reference tracking function is defined to maintain records of what interface references exist and where copies of references are held, and to support whatever garbage collection policy the system designer selects.

10.3. Repository functions

The repository functions are all concerned with persistent storage. There is a general storage function and then a number of specializations of it, supporting different types of repository. The basic storage function just allows an object using it to make any data item persistent, that is, to have a longer lifetime than the object itself.

The information organization function stores information about the various objects and interfaces in the system, and supports structured queries on the information stored. It can be used to maintain information about relationships between and attributes of objects.

The relocation function provides a specialized store of information about interface references, which can be used to update an interface reference if the object concerned moves, of fails and is restarted. For this mechanism to work, it is necessary for each of the mechanisms which might be involved in altering the interface reference data to record the appropriate information with the relocator.

The type repository function provides a source of information about the various type definitions supporting the system, particularly interface types, and can record type identifiers, type definitions and assertions of subtyping relationships between them.

In contrast, the trader stores information about interface instances and properties associated with them. These properties can give information both about the expected behaviour at an interface and less tangible properties of the service available from it. The trader can be queried for services by type and by properties, allowing suitable instances of the service to be discovered. Service offers are placed in the trader's records by a service exporter and queries performed by an importer, which intends to use or pass on knowledge of the service. The trader is one of the most important ODP functions, because it allows the dynamic configuration and evolution of distributed systems. Objects using it can seek out the services they need. Because of its importance, it is the first of the ODP functions to be the subject of formal standardization.

10.4. Security functions

The RM-ODP identifies a full range of security functions, largely by reference to the established standard security frameworks. It identifies functions for access control, security audit, authentication, integrity, confidentiality, non-repudiation and key management.

11. ODP TRANSPARENCIES

The ODP transparencies are defined by giving a prescription for translating from a set of system specifications in the computational, information and enterprise viewpoints to an engineering specification which incorporates the various ODP functions needed; the functions are used in a coordinated way so as to provide the necessary transparency. The transformation may result in changes to the original object's behaviour and interface

signatures, in order to incorporate any control interactions and information needed to guarantee the transparency. Thus, for example, the transaction transparency may require additional commit interactions and the addition of transaction identifiers to the parameters carried by existing interactions.

11.1. Access and location transparencies

Access transparency is normally provided as part of the basic function of the engineering stub object, and so the transformation to be performed to provide it is a straightforward refinement to introduce the channel structure.

In a similar way, location transparency will be provided by some combination of the stubs and protocol objects.

11.2. Failure transparency

Failure transparency is requested in terms of the kinds of failure that should not be allowed to disrupt the application. It can be provided in a number of ways. Firstly, the objects involved can be placed in an environment which is inherently sufficiently reliable, such as a non-stop system. Secondly, the checkpointing and recovery mechanisms can be used to overcome faults when the occur. Thirdly, the replication mechanisms can be used to make faults non-damaging to the application.

Which of these approaches is to be taken will depend on the relative cost and performance objectives to be met, particularly whether the system has to give real-time guarantees. These choices will be made on the basis of the enterprise policies which have been established.

11.3. Migration transparency

The migration transparency is expected to minimize the effect of movement of objects. One part of the information needed to determine the support needed for migration is the set enterprise policies which constrain object mobility, since if the objects do not move, there is no problem.

The decision as to whether objects should be moved will have to take into account issues of resource management, performance targets and security. Once the mobility constraints are established, suitable migration strategies can be determined and the mechanisms needed to support them incorporated.

11.4. Persistence and relocation transparencies

Both the persistence and relocation transparencies will involve the use of the relocation function. This is because they are both involved in changes which may invalidate current interface references, causing subsequent attempts to create a binding to fail.

In both cases, the relocation function will need to be used in a way which is coordinated with the resource management and recovery activities, to ensure that sufficient information is provided for the relocator to do its job.

11.5. Replication transparency

The provision of replication transparency is potentially complex because of the need to consider both the client and server roles of any object being replicated, to ensure that the behaviour of the system as a whole remains consistent.

The transformations involved to support replication are therefore potentially less localized than in some of the other transparencies. The cluster managers supporting the replica copies, at least, will need to be involved in the coordination.

Similar concerns also apply to transaction transparency.

12. CONCLUSIONS

The reference model described in this paper provides a firm basis for the construction of families of open distributed processing systems, capable of supporting a wide range of applications.

ISO and the ITU now plan to populate this framework, drawing on the work of the more forward looking of the industry consortia, where appropriate, to speed the process. The framework is now stable, and has been defined with sufficient flexibility and with a broad enough scope to satisfy the needs of distributed system builders for many years to come.

REFERENCES

[1] ITU Recommendation X.901 | ISO/IEC CD 10746-1, Open Distributed Processing - Reference Model - Part 1: Overview (1994).

[2] ITU Recommendation X.902 | ISO/IEC 10746-2: 1995, Open Distributed Processing - Reference Model - Part 2: Overview.

[3] ITU Recommendation X.903 | ISO/IEC 10746-3: 1995, Open Distributed Processing - Reference Model - Part 3: Overview.

[4] ITU Recommendation X.904 | ISO/IEC CD 10746-4, Open Distributed Processing - Reference Model - Part 4: Overview (1994).

[5] Bowman, H., Derrick, J., Steen, M., "Some Results on Cross Viewpoint Consistency Checking", Proc. ICODP'95, Brisbane, Australia, February 1995