ELSEVIER

Computer Networks and ISDN Systems 27 (1995) 1215–1229

COMPUTER NETWORKS and ISDN SYSTEMS

The ISO Reference Model for Open Distributed Processing: an introduction

Kazi Farooqui [a], Luigi Logrippo [a], Jan de Meer [b,*]

[a] *Department of Computer Science, University of Ottawa, Ottawa K1N 6N5, Canada*

[b] *Research Institute for Open Communication Systems Berlin (GMD-FOKUS), Hardenbergplatz 2, D-10623 Berlin, Germany*

Abstract

The ISO Reference Model of Open Distributed Processing (RM-ODP) consists of four parts: an Overview of the reference model, the Descriptive Model, the Prescriptive Model, and the Architectural Semantics. The four parts provide the concepts and rules of *distributed processing* to ensure *openness* between interacting distributed application components. Openness is a combination of characteristics: i.e. scalability, accessibility, heterogeneity, autonomy and distribution.

The RM-ODP introduces the concept of *viewpoint* to describe a system from a particular set of concerns, and hence to deal with the complexity of distributed systems. While all the viewpoints are relevant to the description and design of distributed systems, the computational and engineering models are the ones that bear most directly on the design and implementation of distributed systems. From a distributed software engineering point of view, the computational and engineering viewpoints are again the most important; they reflect the software structure of the distributed application most closely. In this introductory paper, we concentrate on the computational and engineering viewpoints.

Keywords: Standardisation; Distributed processing; Viewpoint models; Architectural semantics; Specification process; Openness

1. Introduction

The Reference Model for Open Distributed Processing (RM-ODP) is an architectural framework for the integrated support of distribution, inter-working, inter-operability and portability of distributed applications. It provides an object-oriented reference model for building open distributed systems. It defines an architecture for distributed systems which enables multi-vendor, multi-domain, heterogeneous, networked-computing.

RM-ODP is a meta-standard to coordinate and guide the development of application-specific ODP standards. While individual ODP standards enable inter-operability and portability of conforming implementations, the RM-ODP enables:

(1) choosing the suitable levels of abstraction for the specification of ODP systems;

(2) using proper modelling concepts corresponding to abstraction levels;

* Corresponding author. E-mail: jdm@fokus.berlin.gmd.d400.de

SSDI 0169-7552(95)00087-5

(3) identifying and relating generic functions of ODP systems; and
(4) selecting adequate formal description techniques (FDTs) and associated methods for expressing, refining, and validating specifications of ODP systems.

RM-ODP identifies several types of interfaces at which standardization may be required, and places constraints only at these interfaces. Thus the issue of heterogeneity is tackled by opening interfaces. It identifies the functionality of the distributed platform, the ODP Support Environment (ODP-SE), required for the open and distribution-transparent interaction between application components.

The scope of ODP can be summarized as providing a framework for building open distributed systems out of networked systems that are heterogeneous in nature. Heterogeneity can include: *equipment heterogeneity*, *operating system heterogeneity*, *computational (programming or database) language heterogeneity*, *application heterogeneity*, *and authority heterogeneity* (e.g., where interaction between autonomous ownership domains is required).

RM-ODP prescribes a methodology for the design of distributed systems by describing different abstraction levels called *viewpoints*. The ODP framework of viewpoints is quite generic. A set of concepts, structures, and rules is given for each viewpoint, providing a *language* for specifying ODP systems in that viewpoint.

RM-ODP is based on precise concepts and, as far as possible, on the use of formal description techniques (FDTs) for the specification of architecture.

2. The structure of ODP RM

The set of documents which comprise the ODP RM consists of four parts. ODP RM is currently an ITU-T and ISO/IEC/JTC1/SC21/WG7 Committee Draft, except for part-2 and part-3 which are available as Draft International Standard:

Part-1: ISO 10746-1/ITU-T X.901: Overview
Part-2: ISO 10746-2/ITU-T X.902: Descriptive Model
Part-3: ISO 10746-3/ITU-T X.903: Prescriptive Model
Part-4: ISO 10746-4/ITU-T X.904: Architectural Semantics

Part-1 contains a motivational overview and guide to the use of RM-ODP. It explains the key concepts of the RM-ODP architecture. It introduces the concept of information distribution.

Part-2 gives precise definition of the concepts required to specify open distributed processing systems. It is a descriptive model. It contains *basic modelling concepts* such as object, interface, behaviour, state, interaction, etc.; *specification concepts* such as composition, decomposition, behavioural compatibility, refinement, trace, template, type, class, etc.; and *architectural concepts* such as organizational concepts (group, configuration), properties of systems and objects (distribution transparency, quality of service) and naming concepts.

Part-3 prescribes the ODP framework of viewpoints for the specification of ODP systems in different viewpoint languages. It contains the specification of characteristics that characterize a system as open distributed system. It is prescriptive in nature.

Part-4 deals with "architectural semantics", i.e., how the modelling concepts of Part-2 and viewpoint languages of Part-3 can be represented in a number of formal description techniques such as LOTOS, Estelle, SDL, and Z. None of the FDTs are completely suitable for the specification of concepts arising in all viewpoints. For example, Z is suitable for information modelling, SDL and LOTOS have been used for computational and engineering modelling.

All the parts of ODP RM are explained using the object-oriented paradigm. The object concept plays a central role in the modelling of ODP systems. An object stands for data abstraction, function encapsulation and modularity. However, different interpretations of the ODP modelling concept of an object are possible, i.e. a real-world thing, the subject of concern, a idealised thing, a

denotation of a model or program or the object itself as part of the real-world.

3. The viewpoint approach of RM-ODP

For any given information processing system, there are a number of user categories — or more accurately, a number of *"roles"* — that have an interest in the system. Examples include the members of the enterprise who use the system, the system analysts, who specify it, the system designers, who implement it, and the system administrators, who install it. Each role is interested in the same system, but their relative views of the system are different, they see different issues, they have different requirements, and they use different vocabularies (or languages) when describing the system. RM-ODP attempts to recognize these different interests by defining different *viewpoints*.

Rather than attempting to deal with the full complexity of distributed systems, the RM-ODP considers the system from different viewpoints or projections, each of which is chosen to reflect one set of design concerns. Each viewpoint represents a different abstraction of the original distributed system, without the need to create one large model describing it.

The ODP framework of viewpoints partitions the concerns to be addressed in the design of distributed systems. A viewpoint leads to a representation of the system with emphasis on a specific set of concerns, and the resulting representation is an abstraction of the system, that is, a description which recognizes some distinctions (those relevant to the concern) and ignores others (those not relevant to the concern). Different viewpoints address different concerns, but there is a common ground between them. The framework of viewpoints must treat this common ground consistently, in order to relate viewpoint models and to make it possible to assert correspondences between the representations of the same system in different viewpoints. This framework allows the verification of both the completeness of the various descriptions and of the consistency between them.

The ODP viewpoints can be used to structure the specification of a distributed system, and can be related to a design methodology. Design of the system can be regarded as a process that may be subdivided into phases related to different viewpoints. Each of the viewpoints can be used as problem analysis technique as well as a solution space of the relevant issues of the problem domain.

These viewpoints should not be seen as architectural layers, but rather as different abstractions of the same system, and should all be used to completely analyse the system. With this approach, consistent and complete system models

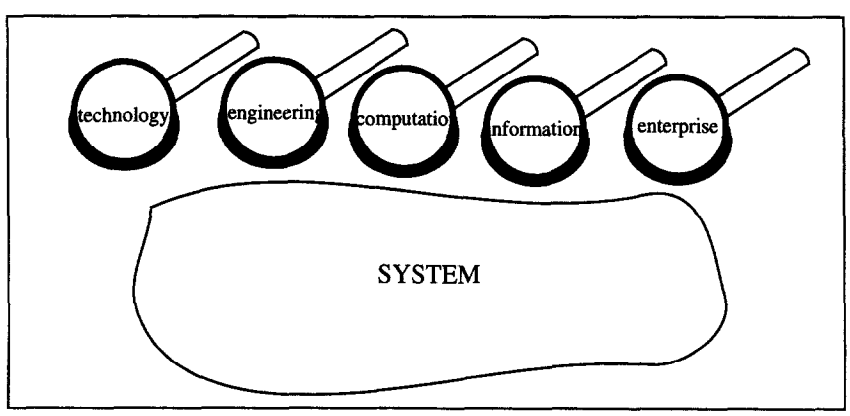


Fig. 1. Viewpoints: different projections on the system.

may be described and developed based on concepts and methods still to be designed for individual viewpoints.

RM-ODP defines the following five viewpoints. Together they provide the complete description of the system: *enterprise viewpoint*, *information viewpoint*, *computational viewpoint*, *engineering viewpoint*, and *technology viewpoint*, see Fig. 1. The concerns addressed in each of the viewpoints are briefly sketched below:

(1) *Enterprise viewpoint*: It is directed to the *needs* of the *users* of an information system. It describes the (distributed) system in terms of answering what it is required to do for the enterprise or business. It is the most abstract of the ODP framework of viewpoints stating high level enterprise requirements and policies.

(2) *Information viewpoint*: It focuses on the information content of the enterprise. The information modelling activity involves identifying *information elements* of the system, *manipulations* that may be performed on information elements, and the *information flows* in the system.

(3) *Computational viewpoint*: It deals with the logical partitioning of the distributed applications independent of any specific distributed environment on which they run. It *hides* from the application designer the details of the underlying ma-

Table 1
Summary of ODP viewpoints

Viewpoint	Enterprise	Information	Computation	Engineering	Technology
Areas of concern	Enterprise needs of IS; Objectives and roles of IS in the organization.	Information models, information structures, information flows, information manipulation.	Logical partitioning of application, application components, component interfaces, component interactions; service-oriented view of distributed application.	Distributed platform infrastructure; distribution transparency, communication support, and other distribution enabling, regulating, and hiding generic mechanisms; system-oriented view of distributed application.	Technological artifacts required for realizing engineering mechanisms.
Main concepts	Agents, artifacts, communities, roles, etc.	Schemas, relations, integrity roles, etc.	Computational object, computational interface, environment constraints, computational interactions, etc.	Basic engineering objects, transparency objects, protocol object, nucleus, etc.	Technological solutions corresponding to engineering mechanisms and structures
Whom does it concern	System procurers, corporate managers.	Information analysts system analysts, information engineers.	Application designers and programmers.	Operating system designers, communication system designers, system designers.	System integrators, system vendors.
Language/ notation	Requirement description languages.	Entity-relationship models, conceptual schemas, etc.	Application programming environments, tools, programming languages, etc.	Distributed platforms, engineering support environments, etc.	Technology mappings, identification of technical artifacts, etc.
Role in software engineering	Requirement capture and early design of distributed system.	Conceptual design and information modelling.	Software design and development.	System design and development.	Technology identification, procurement, installation.

chine (distributed platform) that supports the application.

(4) *Engineering viewpoint*: It addresses the issues of system support (platform) for distributed applications. It identifies the functionality of the distributed platform required for the support of the computational model.

(5) *Technology viewpoint*: The technology model identifies possible technical artifacts for the engineering mechanisms, computational structures, information structures, and enterprise structures.

A summary of ODP viewpoints is presented in Table 1.

Using the five ODP viewpoints to examine system issues encourages a clear separation of concerns, which in turn leads to a better understanding of the problems being addressed: describing the role of the enterprise (enterprise viewpoint) independently of the way in which that role is automated; describing the information content of the system (information viewpoint) independently of the way in which the information is stored or manipulated; describing the application programming environment (computation viewpoint) independently of the way in which that environment is supported; describing the components, mechanisms used to build systems independently of the machines on which they run; and describing the basic system hardware and software (technology viewpoint) independently of the role it plays in the enterprise.

4. The ODP computational model

The ODP computational model is a framework for describing the structure, specification and interactions of (components of) a distributed application on a (distributed) computing platform.

The computational model provides a set of basic (abstract) concepts and elements for the construction of a programming (specification) language for which the model does not provide any syntax. Using the computational modelling concepts, one can specify (program) a distributed application without worrying about the details of the underlying distributed execution platform. The design principle of the computational model is to minimize the amount of engineering details that the application programmer is required to know, yet at the same time allowing the programmer to exploit the benefits of distributed computing.

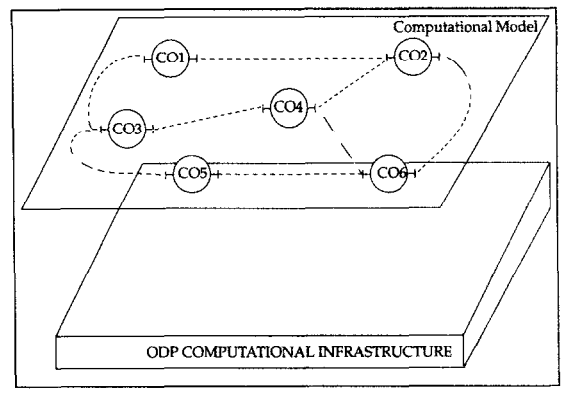


Fig. 2. ODP computational specification: an object world supported by distributed platform.

A *computational specification* of a distributed application consists of the composition of *computational objects* (which represent application components) interacting, by *operation invocations*, at their interfaces. It identifies the activities that occur within the computational objects, and the interactions that occur at their interfaces, (*computational interfaces*), see Fig. 2.

4.1. Computational model: a object-oriented view of distributed application

The computational model is based on a distributed-object model. It prescribes an object-oriented view of the distributed application. Applications are collections of interacting objects. In this model, objects are the units of distribution, encapsulation, and failure.

The computational model is an "object world" populated with concurrent (computational) objects interacting with each other, in a *distribution-transparent* abstraction, by invoking operations at their interfaces. An object can have multiple interfaces and these interfaces define the interactions that are possible with the object.

"Activity" is a unit of concurrency within an object. A collection of (computational) objects may have any number of activities threading through them. The state encapsulated by the object can be accessed and modified by the activities executing the operations in the interfaces of that object.

A distributed computation progresses by operation invocations at object interfaces. The activity in an object (invoking object) can pass into another object (invoked object) by invoking operations in the interface of the invoked object. Activities carry the state of their computations with them, i.e., when an activity passes into an operation it carries the parameters for that invocation, and returns carrying results. In the computational model, concurrency within an object and communication between objects are separate concerns. While concurrency is modelled by the concept of activity, communication between object is modelled as (remote) invocation of an operation.

4.2. Distribution issues and the computational model

Computational specifications are intended to be distribution-transparent, i.e., written without regard to the specifics of a physically distributed, heterogeneous environment. However, the expression of *environment constraints* in the computational interface template provides a hint of the application requirements from the distributed platform, e.g., distribution transparencies, security mechanisms, specific resource requirements, etc..

At the computational level, user applications are unaware of how the underlying distributed platform is structured or how the distribution enabling and regulating mechanisms are realised.

4.3. Elements of the computational model

The design philosophy of the computational model has been to find the smallest number of concepts (elements) needed to describe distributed computations and to propose a *declarative* approach to the formulation of each concept. This section is a brief introduction of some basic

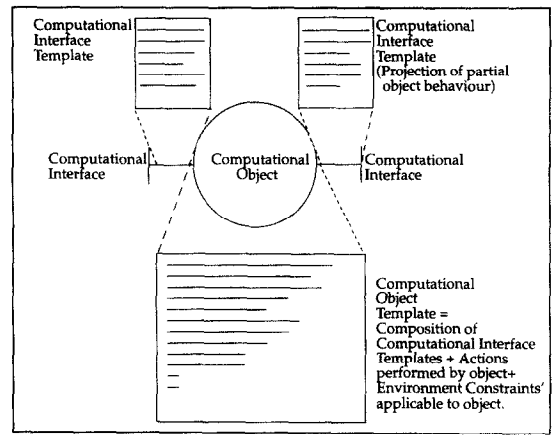


Fig. 3. ODP computational model concepts.

computational elements out of which the *computational specification* of the distributed application is constructed.

The basic elements of the computational model are: *computational object*, *computational interface*, interface invocation mechanisms such as *computational operation*, and the abstraction to model the communication between the computational interfaces — *binding object*.

Computational Object: The components of distributed application are represented as computational objects in the computational model. The computational objects are the units of (application) structure and distribution. The computational objects model both the application components that perform information processing and those components that store the information.

As shown in Fig. 3, a computational object template consists of a set of computational interface templates which the object can instantiate.

Computational Interface: While computational objects are the units of structure and encapsulation of (application-specific) services, interfaces are the units of provision of services; they are the places at which objects can interact and obtain services.

The distributed application components (modelled as computational objects) may be written in different (programming) languages and may run on heterogeneous environments. In order for a component to be constructed independently of

another component with which it is to interact, a precise specification of the interactions between them is necessary. The specification of interaction between computational objects, and of their requirements of interaction are captured by interface templates. The computational interfaces model different interaction concerns of an object.

A computational object may support multiple computational interfaces which need not be of the same type. Interfaces of the same type may be provided by objects which are not of the same type. Each object may provide interfaces which are unlike those provided by the other object.

In the ODP computational model two kinds of interfaces are identified: operational interfaces and stream interfaces.

Operational Interface: The specification of operational interface template consists of:

(1) Operation specification.

(2) Behaviour specification.

(3) Environment contract.

The operation specification includes the operation name together with the number, sequence, and type of arguments that may be passed in each operation invocation and its response(s). This is called *operation signature.*

The behaviour specification defines the behaviour exhibited at the interface. All possible orderings of operation invocations at or from the interface are specified. The behaviour constitutes the protocol part of the interface.

Most interface specifications, to date, have concentrated on the syntactic requirements of the interface such as the operation signature. Aspects other than pure syntax are also important in facilitating the interaction between a pair of objects. This additional semantic information falls into two categories:

(a) Information affecting the way in which the infrastructure supports the interactions; this information constrains the type of distribution transparencies, choice of communication protocols, etc. that must be placed in the interaction path between the interacting objects.

(b) The behaviour (or the semantics) of the service offered at the interface; an interface is viewed as a projection of an object's behaviour, seen only in terms of a specified set of observable actions. As a result, signature compatibility is less discriminating than interface compatibility.

The environment contract in the computational interface template defines the following attributes:

(1) Distribution transparency requirement on operation invocation.

(2) Quality of service (including communication quality of service) attributes associated with the operations.

(3) Temporal constraints on operations (e.g., deadlines).

(4) Dependability constraints (e.g., availability, reliability, fault tolerance, security etc.)

(5) Location constraints on interfaces (and hence their supporting objects).

(6) Other environment constraints on operations (e.g., those arising from enterprise and information viewpoint).

These attributes may be associated with individual operations or the entire interface. The environment contract is an important component of the computational interface template and has a direct relationship to the realized engineering structures and mechanisms.

Stream Interface: The computational objects may perform the information processing task as well as act as containers of information. There is a need to model not only the interfaces which provide "service", but also those interfaces which model "continuous" information flow. Such interfaces are modelled, in the computational model, as *stream interfaces.*

The stream interface is a set of information flows whose behaviour is described by a single action which continues throughout the life time of the interface. Information media such as voice and video inherently consist of a continuous sequence of symbols. Such media are described as *continuous* and the term *stream* is used to refer to the sequence of symbols comprising such a medium.

Examples include the flow of audio or video information in a multimedia application, or the continuous flow of periodic sensor readings in a process control application. The computational description does not need to be concerned with detailed mechanisms; the fact that the flow is

established and continues during the relevant period is enough.

The template for a stream interface consists of

Stream Signature: A specification of the type of each information flow contained in a stream interface and, for each flow, the direction in which the flow takes place.

Environment Constraint: Continuous media have strict timing and synchronization requirements. The environment constraints that are relevant to stream interfaces include synchronization and clocking properties, time constraints, priority constraints, throughput, jitter, delay, media-specific communication quality requirements, etc., in addition to the properties applicable to operational interfaces.

Role: A role for each information flow, e.g., a producer object or a consumer object.

Binding Object: Interactions between computational objects are only possible, when their interfaces are bound. There is a concept of implicit and explicit binding in the computational model. When objects get implicitly bound in the computational model, it is assumed that the underlying platform (the engineering infrastructure), will provide the service of checking the consistence between the interfaces to be bound.

The computational objects are explicitly bound through a binding object. The template for the binding object specifies the interaction patterns between the bound computational objects. The binding object contains control interfaces which allow dynamic modification of number and types of objects involved in the binding.

5. Engineering model

The engineering model is an abstract model to express the concepts of the engineering viewpoint. It involves concepts such as operating systems, distribution transparency mechanisms, communication systems (protocols, networks), processors, storage, etc. As the notions of processor, memory, transport network play a more indirect role in a distributed system, the term "engineering model" is used here in a more general way to describe a framework oriented towards the organization of the underlying distributed infrastructure and targeted to the application support. It mostly focuses on what services may be provided to applications and what mechanisms should be used to obtain these services. The term *platform* is used to refer to the (configuration of) services offered to applications by the infrastructure.

The engineering model is still an abstraction of the distributed system, but it is a different abstraction than the computational model. Distribution is no longer transparent, but we still need not concern ourselves with real computers or with the implementations (technology) of mechanisms or services identified in the engineering model. The engineering model provides a machine-independent execution environment for distributed applications.

Unlike the enterprise, information, and computational models which deal with the semantics of distributed applications, the engineering model is not concerned with the semantics of the distributed application, except to determine its requirements for distribution.

5.1. Engineering model: an object-based distributed platform

The ODP engineering model is an architectural framework for the provision of an object-based distributed platform. The set of basic services and mechanisms, identified in the engineering model, are modelled as a collection of interacting objects which together provide support for the *realization* of interactions between distributed application components.

The engineering model can be considered as an extended operating system spanning a network of interconnected computers. In the *networked-operating system* view of the model, the linked computers preserve much of their autonomy and are managed by their local operating systems which are enhanced with mechanisms to enable, regulate and (if desired) hide distribution.

5.2. Engineering model: animation of computational model

The interest of the computational model is directly related to the existence of a mapping

enabling it to relate to engineering concerns. This means, for instance, being able to map computational concepts onto the engineering structures.

The engineering model provides an infrastructure or a distributed platform for the support of the computational model. The model provides generic services and mechanisms capable of supporting distributed applications specified in the computational model. The model is concerned with *how* an application, specified in the computational model, may be *engineered* onto the distributed platform. The selection of distribution transparency and communication (protocol) objects, among many other support mechanisms, tailored to application needs, forms an important task.

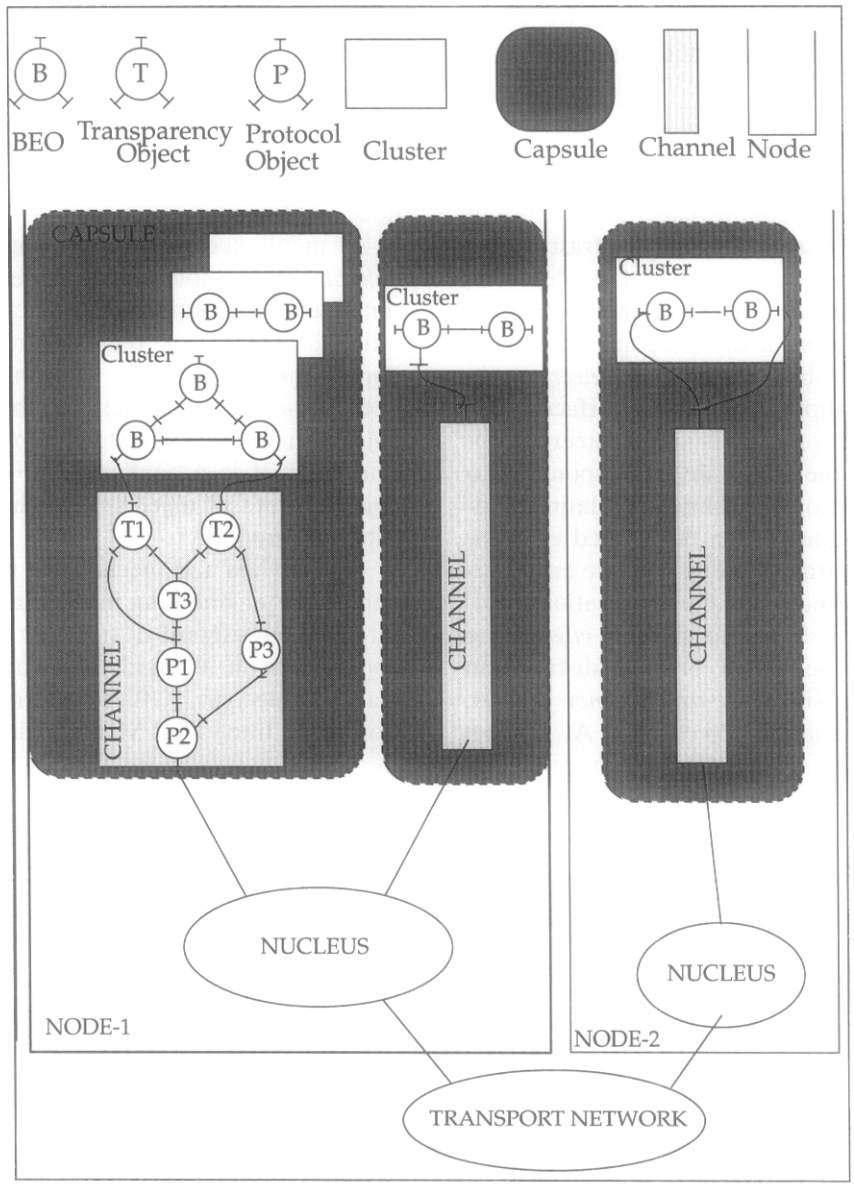


Fig. 4. ODP engineering model: Organization of distributed infrastructure.

The engineering model identifies the *functionality* of basic system components that must be present, in some form or other, in order to support the computational model. Hypothetically, there may be several engineering models for a particular computational environment, reflecting the use of different system components and mechanisms to achieve the same end. The issue in the computational model is *what* (interactions, distribution requirements); the engineering model prescribes solution as to *how* to realize these interactions, satisfying the stated requirements.

5.3. Structure of engineering model

The engineering model reveals the structure of the distributed platform, the ODP infrastructure which supports the computational model. The services or mechanisms which enable, regulate and hide distribution in the ODP infrastructure, are modelled as objects, called *engineering objects*, which may support multiple interfaces.

There are different kinds of engineering objects in the engineering model corresponding to different distribution (enabling, regulating, hiding) functions required in a distributed environment. This is illustrated in Fig. 4. Some engineering objects correspond to the application functionality and are referred to as *basic engineering objects* while those which provide distribution functions are classified as *transparency objects*, *protocol objects*, *support objects*, etc. At a given host, the basic engineering objects belonging to an application may be grouped into *clusters*. A host may support multiple clusters in its addressing domain, known as *capsule*. A capsule consists of clusters of basic engineering objects, a set of transparency objects, protocol objects and other local operating system facilities.

From an engineering viewpoint, the ODP infrastructure consists of interconnected autonomous computer systems (hosts), which are called *nodes*. Each node supports a *nucleus object* and multiple capsules. The nucleus encapsulates computing, storage, and communication resources at a node. All the objects in the node share common processing, storage, and communication resources encapsulated in the nucleus object of the node.

As mentioned before, the engineering model *animates* the computational model. The computational-level interactions between a pair of computational objects (or their interfaces) are supported through *channel* structures in the engineering model. A channel binds basic engineering objects in different clusters, capsules, or nodes. The channel is a configuration of transparency objects, protocol objects, etc. which provide distribution support.

The services and mechanisms currently identified in the engineering model are generic in nature and can support distribution requirements of applications in a broad range of enterprise domains (Telecoms, Office Information Systems, Computer Integrated Manufacturing, etc.). How-

Table 2
System abstractions in engineering model

Engineering object	System representation
Node	Single computer system, network of workstations managed by a distributed operating system, any autonomous information processing system with independent nucleus resources and failure characteristics.
Nucleus	Abstraction of an operating system providing processing, storage, and communication resources of a node.
Capsule	The concept of address space in operating systems.
Cluster	The concept of "linked" modules to form an executable program image.
BEO	The program module which may not be executed in isolation.
Channel	The run time "binding" between distributed BEOs
Transparency object	Special purpose modules which enhance the operating system environment of the node and can be dynamically linked into the distributed application program.

ever, domain-specific supporting functions will be defined in the domain-specific engineering models (which are the specialization of ODP engineering model).

The following is a brief description of the engineering objects and structures currently identified in the ODP engineering model. The objects and structures which are defined later in the text are italicized. Table 2 gives a relationship between the engineering objects and the real world system.

Basic Engineering Object: Basic Engineering Objects (BEOs) are the run time representation of computational objects (obtained through compilation, interpretation or through some other transformation of computational objects) which encapsulate application functionality.

Cluster: A cluster is a configuration of basic engineering objects. Clusters are used to express related objects (which belong to the same application) that should be local to one another, i.e., those groups of objects that should always be on the *same* node at all times.

Capsule: A capsule consists of clusters of basic engineering objects, *transparency objects*, and *protocol objects* bound to a common *nucleus* in a distinct address space from any other capsule. A capsule provides to its clusters access to the objects in the *channel* and to the nucleus to which it is bound.

Nucleus: A nucleus is an object that provides access to basic processing, storage, and communication functions of a *node* for use by basic engineering objects, *transparency objects*, *protocol objects*, bound together into capsules. A nucleus may support more than one capsule. A nucleus has the capability of interacting with other nuclei (through its communication function), providing the basis for inter-capsule and inter-node communication.

Node: A node consists of one nucleus object, a node manager, and a set of capsules. All of the objects in a node share common processing, storage, and communications resources.

Channel: A channel is a configuration of *transparency objects*, *protocol* objects, *application specific supporting objects*, etc. providing a binding between a set of interfaces to basic engineering objects, through which interaction can occur. The structure of the channel is dependent on the

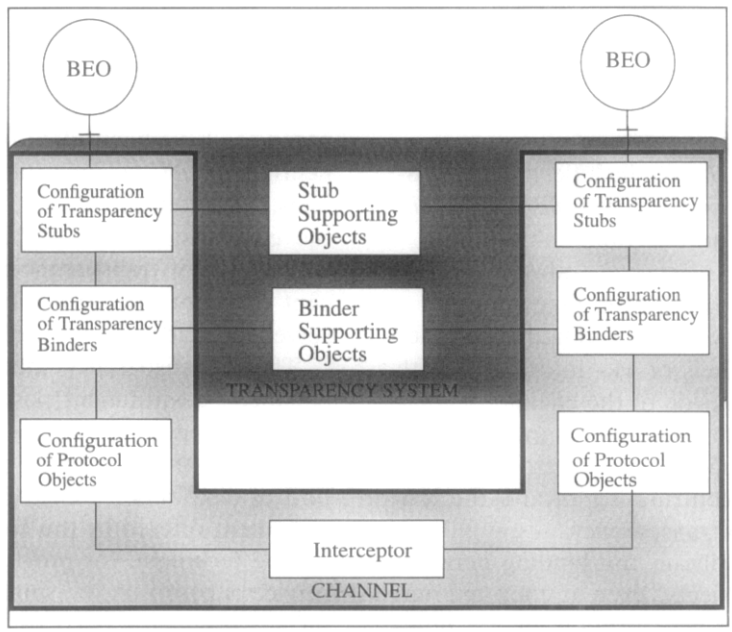


Fig. 5. Simplified generic channel structure.

distribution function requirements of the interaction between basic engineering objects.

Fig. 5 shows the client-half and server-half of a single channel object. If the objects being bound are on different nodes, there is still conceptually only one channel object created, i.e., there is not one channel object on one node and a different channel object on the other.

Stub Object: An object which acts to a basic engineering object as a *representative* of another basic engineering object located in different clusters, thus contributing towards distribution transparency. Stub objects are bound to the basic engineering objects for the purpose of hiding certain aspects resulting from distribution (or heterogeneity).

The stub objects have direct access to the basic engineering objects. The operation invocations on the interfaces of basic engineering objects are *intercepted* by stub objects to hide some aspects of distribution such as concurrency in the system or to modify the information exchanged between basic engineering objects, thus masking the heterogeneity in the distributed system.

Stub objects add further interactions and/or information to interactions between interacting basic engineering objects to support distribution transparency. As an example, a stub object may provide adaptation, such as marshalling and unmarshalling of operation parameters to enable *transparent* interactions between interfaces of basic engineering objects.

Examples of stub objects include *access transparency object* and *concurrency transparency object* discussed in the next section.

Basic engineering objects are always directly bound to the stub objects. Stub objects within a channel can interact with one another using other objects in the channel, or via interaction with supporting objects outside of the channel.

Binder Object: An object which *controls* and *maintains* the binding between interacting basic engineering objects, contributing towards the provision of distribution transparency.

Binder objects maintain the binding between basic engineering objects, even if they are migrated, reactivated at new location, or are replicated. Examples of binder objects include *location transparency object*, *migration transparency object*, *replication transparency object*, *failure transparency object*, and *resource transparency object*.

Stub objects are bound to binder objects. Binder objects interact with one another to maintain the integrity of the binding between the interacting basic engineering objects. Binder objects in the channel can interact with one another using other objects in the channel, or via interaction with supporting objects outside the channel. Binder objects are interconnected by protocol objects.

Protocol Object: An object which encapsulates communication protocol functionality for supporting communication between basic engineering objects. A channel may be composed of a number of protocol objects corresponding to different communication support requirements of interactions between basic engineering objects. Protocol objects interact with other protocol objects to support interaction between basic engineering objects.

Interceptor Object: An object which masks administrative and technology domain boundaries by performing transformation functions such as protocol conversion, type conversion etc. It enables interactions to cross administrative and communication domains, thus contributing towards *federation transparency*.

Distribution Transparency: The following transparencies have been identified in the ODP engineering model, as important in distributed systems. The concept of transparency is viewed as the corner stone of ODP architecture. A brief description of transparencies, based on the concept of client and server objects (or client and server interfaces) is outlined below:

These transparency mechanisms provide an enhanced environment positioned on top of the low-level operating systems and communications facilities of the distributed platform, for the support of distribution transparent programming environment offered by the computational model.

The technique for providing any transparency service is based on the single principle of replacing an original service by a new service which combines the original service with the trans-

parency service, and which permits clients to interact with it as if it were the original service. The clients need not be aware of how these combined services are achieved.

Since the interactions between the objects occur at their interfaces, these transparencies are applicable to individual interfaces or to specific operations of the interfaces. An interface may have a set of transparency requirements which may be different from those of other interfaces of the same object.

A summary of transparency mechanisms is presented in Table 3.

Access Transparency: It hides from a client object the details of the access mechanisms for a given server object, including details of data representation and invocation mechanisms (and vice versa). Access transparency hides the difference between local and remote provision of the service.

Access transparency enables interworking across heterogeneous computer architectures, operating systems and programming languages.

Concurrency Transparency: It hides from the client the existence of concurrent accesses being made to the server. Concurrency transparency hides the *effects* due to the existence of concurrent users of a service from individual users of the service.

Location Transparency: It hides from a user (client) where the object (server) being accessed is located.

Migration Transparency: Migration transparency hides from the user of the service (client) the effects of the provider of the service moving from one location to another, during the provision of the service (between successive operation invocations).

Location transparency is a static transparency in the sense that it is assumed that once located

Table 3
ODP distribution transparencies

Transparency	Central issue	Result of transparency
Access	The method of access to objects (invocation mechanism and data representation).	Client need not be unaware of access mechanisms at the server interface.
Concurrency	Concurrent access to objects in the distributed system.	Clients are masked from the effects of concurrent access to the server interface.
Location	Location of object in the distributed system.	Clients are unaware of the physical location of the server.
Migration	Dynamic re-location of objects during the "bind-session".	Clients are unaware of the dynamic migration of the server.
Replication	Multiple invocations on replicated objects, multiple responses, and consistency of replicated data.	Client invokes a replicated server group as if it were a single server. Distribution of request, collation of responses, consistency of data, and membership changes are hidden.
Resource	Resource management policies of the node (deactivation and reactivation of objects).	Client unaware of the deactivation and reactivation of the server.
Failure	Partial failure of object in the node.	Client unaware of the failure of the server and its subsequent reactivation (possibly at another node).
Federation	Pan-organizational boundaries.	Clients unaware of interactions crossing administrative and technology boundaries.

the interface remains at its location (until the binding between the involved interfaces is broken). Migration transparency is the dynamic case which arises if the server interface can move while the client object is interacting with it, without disturbing those interactions.

Replication Transparency: Replication transparency, also known as *group transparency*, hides the presence of multiple copies of services and maintaining the consistency of multiple copies of data, from the users of the services.

It enables a set of objects (their interfaces) organized as a *replica group* to be coordinated so as to appear to interacting objects (or their interfaces) as if they were a single object (interface).

There are two main aspects of replication transparency. The first hides the difference between a replicated and a non-replicated provider of a service from users of that service, and the second hides the difference between replicated and non-replicated users of a service from providers of that service.

Users are unaware of multiple providers of the service and need not concern about making multiple operation invocation or dealing with multiple responses.

Resource Transparency: It hides from a user (client) the mechanisms which manage allocation of resources by activating or passivating (server) objects as demand varies. It also implies the hiding of deactivation and reactivation of (server) objects from the clients.

Resource transparency, also known as *liveness transparency*, masks the automated transfer of clusters from a capsule to a storage object and back again, to optimize the use of node's nucleus resources (processor, memory, etc.).

With resource transparency in place, clients can invoke operations on the server irrespective of whether the server is currently active or passive.

Failure Transparency: Failure transparency masks (certain) failure(s) and possible recovery of server objects from the client objects, thus providing fault tolerance.

Federation Transparency: Federation transparency hides the effects of operations crossing multiple administrative boundaries from the clients. It permits inter-working across multiple administration and technology domains.

6. Conclusion

Using the five ODP viewpoints to examine system issues encourages a clear separation of concerns, which in turn leads to a better understanding of the problems being addressed: describing the role of the enterprise (enterprise viewpoint) independently of the way in which that role is automated; describing the information content of the system (information viewpoint) independently of the way in which the information is stored or manipulated; describing the application programming environment (computation viewpoint) independently of the way in which that environment is supported; describing the components, mechanisms used to build systems independently of the machines on which they run; and describing the basic system hardware and software (technology viewpoint) independently of the role it plays in the enterprise.

The purpose of the RM-ODP framework of viewpoints is to position services relative to one another, to guide the selection of appropriate models of services, and to help in the placement of boundaries upon ODP. The framework of viewpoints is used to partition the concerns to be addressed when describing all facets of an ODP system, so that the task is made simpler.

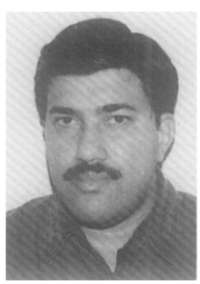
Acknowledgements

This research was funded in part by the Telecommunications Research Institute of Ontario and the European RACE II project R2088 "Tools for Protocol and Advanced Service Verification in IBL Environments (TOPIC)".

References

[1] Draft Recommendation ITU-T X.901/ISO 10746-1: Basic Reference Model of Open Distributed Processing–Part-1: Overview.

[2] International Standard ITU-T X.902/ISO 10746-2: Basic Reference Model of Open Distributed Processing–Part-2: Descriptive Model.
[3] International Standard ITU-T X.903/ISO 10746-3: Basic Reference Model of Open Distributed Processing–Part-3: Prescriptive Model.
[4] Draft Recommendation ITU-T X.904/ISO 10746-4: Basic Reference Model of Open Distributed Processing–Part-4: Architectural Semantics.
[5] *Proceedings of the IFIP TC6 / WG6.4 International Workshop on Open Distributed Processing* (October 1991), North-Holland 1992.
[6] *Proceedings of the International Conference on Open Distributed Processing* (September 1993), Berlin.
[7] *Proceedings of the First Telecommunication Information Networking Architecture Workshop*, (TINA 90), Lake Mohonk, New York, USA, June 1990.
[8] *Proceedings of the Second Telecommunication Information Networking Architecture Workshop*, (TINA 91), Chantilly, France, March 1991.
[9] *Proceedings of the Third Telecommunication Information Networking Architecture Workshop*, (TINA 92), Narita, Japan, January 1992.
[10] *Proceedings of the Fourth Telecommunication Information Networking Architecture Workshop*, (TINA 93), L'Aquila, Italy, September 1993.

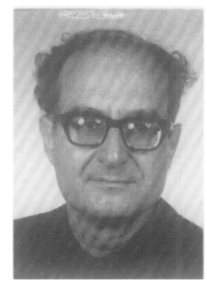


Kazi Farooqui received the Bachelors degree in Electronics and Communication Engineering from Jawaharlal Nehru Technological University, Hyderabad, India in 1984 and a Masters degree in Computer Science from University of Roorkee, India in 1987 specializing in Computer Network Architectures, Communication Protocols, and Open System Interconnection.

After obtaining the bachelors degree he worked for the Defence R&D Laboratory, Hyderabad on scientific software development and later for the Indian Institute of Technology, Bombay on a UN sponsored project, ERNET, and also for CMC Limited, New Delhi on various networking and OSI software development projects.

Currently he is pursuing Ph.D. in Computer Science at University of Ottawa. His research interests include Communication Protocols, Object-Oriented Distributed Systems, Open Distributed Processing, Programmability and Open Distributed Platforms for Intelligent Networks.

He is a Member, of the Institute of Electrical and Electronic Engineers.



Luigi Logrippo received his "laurea" from the University of Rome (Italy) in 1961. Until 1967, he worked with Olivetti, General Electric, and Siemens as a programmer and systems analyst. From 1967 to 1969 he was a Research Associate at the Institute for Computer Studies, University of Manitoba, where he obtained a M.S.c in computer science in 1969. He then obtained a Ph.D. in computer science at the University of Waterloo in 1974. Since 1973, he has been with the University of Ottawa, where he is now an Associate Professor. He has published in parallel program schema theory, computer-assisted analysis of music, software Methodology, and data communications protocols. His current research interest include Software methodolgy (specification, verification, formal development from specifications, and testing techniques). Data Communications Protocols (the same subjects with application to protocols), and standardization of Open Systems Interconnection. He has participated in the design of the language LOTOS, and currently is pursuing research on its application.

Jan de Meer is a member of the ACM since '76 and of the IEEE and the German Society for Informatics (GI) since '89. After he got his Bachelor's of Electronics in '72 and his Master's of computer science in '79 he began to work on the design and implementation of Computer Networks, first at the Hahn-Meitner Institute for Nuclear Research (HMI), and later at the National Research Centre GMD, both in Berlin. Since '83, he has been involved in the formal specification of Communication Protocols. Also in '83 he became a member of the German Standardization Institute (DIN), and volunteered for the working groups WG1(OSI) and WG7(ODP) of the International Standardization Organization (ISO). Since '91 he has been chairing both groups at DIN. Since '87 he is the head of the System Engineering and Methods research group of the FOKUS-Institute of the GMD in Berlin. At HMI he was involved in the implementation of the first computer network in Germany based on the ISO reference model for Open Interconnected Systems. At GMD-FOKUS he is responsible for R & D projects of ESPRIT and RACE dealing with formal specification and tooling for Network Performance and Quality of Service Specification, Testing and Verification. He participates actively in the various IFIP conferences PSTV, FORTE, IWPTS and ICODP.