Architecture for Open Distributed Software Systems

Kazi Farooqui, Luigi Logrippo

Department of Computer Science,

University of Ottawa,

Ottawa K1N 6N5, Canada.

(farooqui | luigi@csi.uottawa.ca)

Distributed applications consist of discrete software components which are spread across multiple systems. The interaction between these components constitutes distributed processing. If the discrete components are capable of exhibiting "openness", then the interaction between them can be characterized as open distributed processing.

In an ideal open distributed system, it should be possible for distributed applications developed in different environments to interact. This can be achieved if distributed environments conform to a common conceptual model or architecture.

This chapter describes a common conceptual framework for the design of distributed systems, which is gaining a wide degree of acceptance within the distributed systems research community. This is the basis for the standardization of Reference Model for Open Distributed Processing (RM-ODP).

The chapter is divided into five parts. Part-1 is an introduction to the architecture for open distributed systems. A set of five abstraction levels- *enterprise viewpoint*, *information viewpoint*, *computational viewpoint*, *engineering viewpoint*, and *technology viewpoint*, are identified in RM-ODP for the specification of distributed software systems. While all the viewpoints are relevant to the design of distributed systems, the computation 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 the most important; they reflect the software structure of the distributed application most closely. In this chapter, we concentrate on the computational and engineering viewpoints in Part-2 and Part-3 respectively. Part-4 is an illustration of the application of ODP architectural concepts in a simple client-server scenario. Conclusions are drawn in Part-5.

PART-1

1.1 Motivation for Open Distributed Systems Architecture

Today's distributed systems are complex structures, composed of many types of hardware and software components. In some systems, components are developed separately by different implementors, and then combined together resulting in a heterogeneous system. Well known examples of such systems, which fall within the scope of the distributed systems architecture addressed in this chapter, include telecommunication systems (advanced intelligent networks), computer communication networks (internet), automated manufacturing systems, office automation systems, client-server systems (banking and airline reservation systems), etc.

In order to reason about such systems, it is necessary to develop appropriate concepts. These concepts may vary according to the point of view from which the system is being considered. For example, from the end user's point of view, the system will be described in terms of user objectives and requirements. From the application designer's point of view, it will be described in terms of components communicating together in some way, each component performing some function. The system designer instead is concerned with the communication protocols, etc. required to accomplish the communication between application components. Finally, the technical personnel in charge of putting together the system will see the software and hardware products connected in some way.

Different conceptual frameworks have been devised over the years by implementors and researchers. However such frameworks are usually adapted to specific vendor's architectures, and fail for heterogeneous systems. This situation has been very damaging in practice, because such frameworks are essential in order to design and maintain heterogeneous systems.

Therefore, it should not be surprising that much of the existing work on these unifying concepts has been done within international standardization bodies, mainly the International Organization for Standardization (ISO) and the International Telecommunication Union (ITU, formerly CCITT).

At the time of this writing, a set of documents being put together by the committees of the ISO and ITU, called the Reference Model for Open Distributed Processing (RM-ODP) [1-4] constitutes what many researchers consider to be the most complete and authoritative statement of the state of research in this area. It is the result of the work of many researchers active around the world, who have put it together in many meetings over a number of years.

RM-ODP is based largely on preexisting research work in the field of distributed systems, especially the work done in UK on the Advanced Networked Systems Architecture project (ANSA) [5].

RM-ODP builds on other previously established ISO and ITU standards dealing with heterogeneous systems such as the standards for Open Systems Interconnection (OSI) [6] and Distributed Applications Framework (DAF) [7], Integrated Services Digital Network, and Common channel signalling system.

The well-known OSI Reference Model, structured in seven layers, is the basis of a whole family of protocols which are widely used for heterogeneous system interconnection. Layer7 of the OSI stack deals with the application-specific protocols. Standardization of this layer has resulted in several application-specific standards, but unfortunately in very few concepts of general use. The RM-ODP work has resulted in part from the attempt of developing such general concepts, although it is reaching far beyond this goal.

In contrast to OSI, the work on ODP is not restricted to communication between heterogeneous systems. It deals also with the provision of various distribution transparencies within systems, and with application portability across systems. RM-ODP deals with the application *interaction* problems rather than the pure *interconnection* problems addressed in the OSI model. In this sense, ODP encompasses, and extends OSI. OSI becomes a (communication) enabling technology for ODP applications, i.e., OSI and other related standards (ISDN, CSS#7) provide the communication protocols which are required to support the communication between distributed applications.

1.2 Introduction to ODP

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 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.

As mentioned above, 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).

To this end, the Reference Model of ODP identifies several types of interfaces at which standardization may be required, and places constraints only at and between these interfaces. Thus, the issue of heterogeneity is tackled by opening interfaces.

1.3 Objectives of ODP

As mentioned, the objective of ODP is to enable distributed system components to inter-work seamlessly, despite heterogeneity. One of the mandates of ODP architecture is to identify and define the functionality of mechanisms which mask underlying heterogeneity from users and applications. These mechanisms will address a set of fundamental *distribution transparency* properties.

Additionally, RM-ODP intends to provide a framework for the identification of interfaces (or reference points) where standardization is required for the purpose of making conformance statements at those interfaces and hence ensure openness of interfaces. Thus, one of the tasks of RM-ODP is the broad categorization of interfaces based on their architectural placement: such as man-machine reference point (e.g. graphics standards), networking reference point (e.g. communication interface), interchange reference point (interface to external physical storage medium), and programmatic reference point (e.g., application programming interface) [8].

1.4 ODP Framework of Viewpoints

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 architects that design it, the programmers that implement it, and the technicians that 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 [9].

Rather than attempt to deal with the full complexity of distributed systems, RM-ODP attempts to recognize these different interests by considering the system from different *viewpoints* or *projections*, each of which is chosen to reflect one set of design concerns. As shown in Figure 1., each viewpoint represents a different abstraction of the original distributed system, without the need to create one large model describing the whole of it [8].

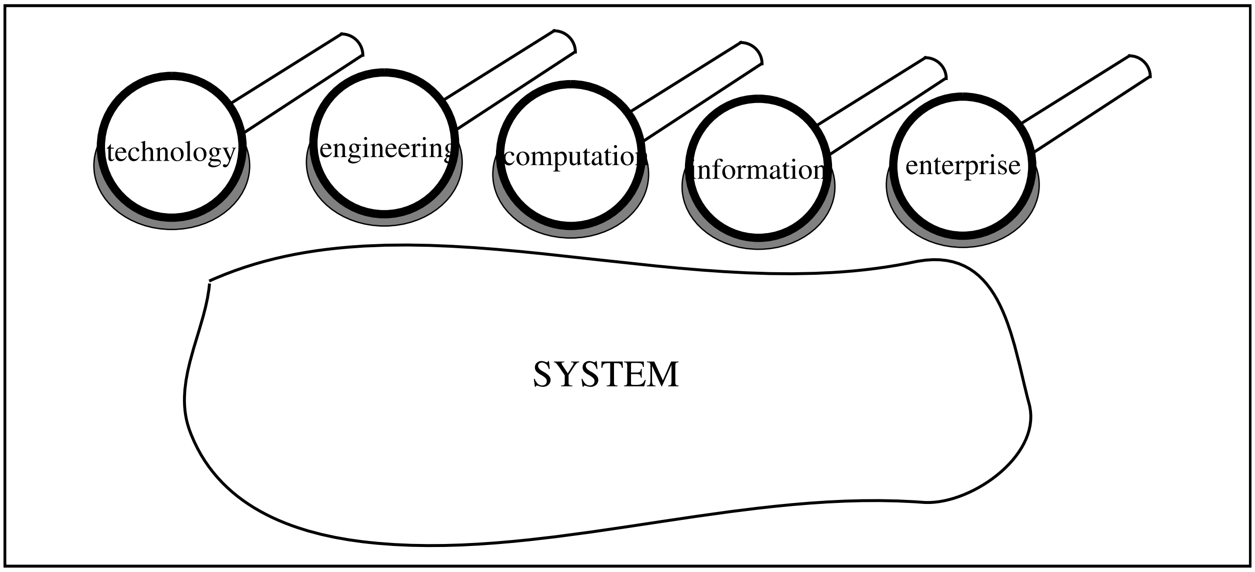

Figure 1. Viewpoints: Different Projections on the System

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 of the software engineering process, 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.

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.

Specifying a distributed system in each of the viewpoints allows an otherwise large and complex specification of distributed system to be separated into manageable pieces, each focussed on the issues relevant to different members of the development team.

The following sections take a detailed look at the ODP viewpoints. Each viewpoint is characterized by indicating the relevant issues, problems, and components visible from that viewpoint.

1.4.1 Enterprise Viewpoint: The Enterprise viewpoint is directed to describing the *needs* of the *users* of an information system. It provides the members of an enterprise in which information systems are to operate with a view of how and where a system is placed and used within the enterprise [10].

An enterprise view covers the enterprise *objectives* of an information system. It focuses on the *requirements* that an organization places on a distributed system and the *role* of the distributed system within the organization.

The Enterprise viewpoint is the most abstract of the ODP framework of viewpoints, stating high-level enterprise requirements, system management policies, and organization structures.

In terms of software engineering, this viewpoint is related to requirements capture and transformation and to the early design of distributed system [17]. The design decisions made using the enterprise viewpoint concern *what* a system is to do and *who* it is doing it for. This allows the designer to develop a closed (i.e., bounded) model which represents all the real world requirements which the designer must incorporate, later in the design trajectory, into the structure of the system.

As a consequence of the large number of enterprises (Telecommunication, Computer

Integrated Manufacturing, Management Information Systems, etc.) to which ODP applies, the RM-ODP cannot sensibly prescribe an all-encompassing enterprise model. However, RM-ODP provides *descriptive* tools for use in constructing such models.

1.4.2 Information Viewpoint: The information viewpoint focuses on the information content of the enterprise. It defines the *information semantics* of the distributed system, i.e., the meaning that a human would ascribe to the data stored or exchanged between components of a distributed system. From this viewpoint, the information processing facilities are seen as black boxes. The parts of the information processing facilities that are to be automated are not differentiated from those to be performed manually [10]. In this model, the distribution of processing is not visible, although the natural distribution of the enterprise itself may, of course, need to be modelled. The model deals with the information, information processing, and information exchange aspects of a distributed system.

The information model is expressed in terms of abstract objects which represent the information elements manipulated by the enterprise. The information modelling activity consists of identifying: *information structures* (or elements) of the system, *constraints* and *manipulations* that may be performed on these information structures, and *information flows* (both the information sources and sinks within the system). These definitions are entirely implementation independent; no restrictions are placed on how the information is represented in a real system, or the means by which it is manipulated.

The information specification of an ODP application could be expressed using a variety of methods, e.g., entity-relationship models, conceptual schemas, Z language, etc. RM-ODP gives descriptive terminology and tools for information modelling.

1.4.3 Computational Viewpoint: The computational viewpoint represents the distributed system as seen by application designers and programmers. It deals with the logical partitioning of a distributed application, breaking it up on the basis of flows of invocation and provision of service. It is here that the idea of particular sets of application components being related by their roles as client or server in an interaction becomes important.

The computational viewpoint regards distributed processing in terms of application components and their interactions, *independent* of any specific distributed environment (operating system, communication system) on which they run. It *hides* from the application designer the details of the realization of the underlying abstract machine (engineering model) that supports it. A computational model may thus be characterized as focussing on applications rather than on the mechanisms used to distribute or, more generally, support them in the system [11].

The computational model describes the coarse grained structure of a distributed application, i.e., application components and their interaction (in terms of operation invocations) at an abstract, system independent level. From a computational viewpoint, the structuring of applications is independent of the computer systems and networks on which they run.

Each coarse-grained entity of a distributed application is represented by an object, called *computational object*, with a set of well defined interfaces, called *computational interfaces*. Computational objects may run concurrently and exhibit internal parallelism. The *computational modelling* of a distributed application consists of the structuring of the application into computational objects, identification and specification of computational interfaces, identification of application-level communication between computational interfaces - called *computational interactions*, and the identification of *environment constraints* associated with these interactions (some of which are deduced from the enterprise viewpoint). Computational objects are, by default, distributed, and hence remote computational interactions are expressed at a high level in a distribution-transparent abstraction in terms of application-level operations - *computational operations*, rather than in terms of physical messages.

The computational viewpoint provides a *service-oriented* view of the distributed application. Computational specifications are expressed declaratively, i.e., state *what* (kind of environment) is required (to support distributed application components and their interactions) and not *how* it is to be provided [9]. Hence, the computational view of a distributed application is expressed in terms of computational objects, computational interfaces, distribution-transparent interactions between these interfaces, and statement of environment constraints (requirements) for their realization (in the engineering model).

Computational viewpoint can be specified using an array of programming languages, interface definition languages, formal description techniques such as Lotos [12], SDL [13], Estelle [14].

1.4.4 Engineering Viewpoint: The engineering viewpoint addresses the issue of system support for distributed applications. It deals with aspects resulting from physical distribution of applications. It provides an infrastructure or a distributed platform for the support of the computational model. It provides generic services and mechanisms capable of supporting distributed applications and their interactions, specified in the computational model.

The engineering viewpoint is centered around the ways the application may be *engineered* into the (distributed) system. It is concerned with concrete application configurations, component placement and distribution, remote object communication and usage of underlying transport protocols, provision of distribution transparency mechanisms, and application-specific support services.

The engineering viewpoint is not concerned with the semantics of the distributed application, except to determine its requirements for distribution. The environment constraints, such as distribution transparency requirements, quality of service attributes, etc., visible in the computational viewpoint, are used to select the available engineering mechanisms, in the engineering model, to achieve the required form of distributed processing.

The ODP engineering model is not a detailed description of how to implement a particular environment. Rather, it identifies the functionality of basic system components that must be present, in some form or other, in order to support the computational environment described in the computational view. It identifies specific interfaces to identified components allowing implementation freedom. There may be several engineering models for a particular computational environment, reflecting the use of different system components (mechanisms) and their configurations to realize the same computational environment.

The mechanisms visible in the engineering model are processing and storage resources, distribution transparency mechanisms (access transparency, location transparency, concurrency transparency, migration transparency, replication transparency, resource transparency, failure transparency, federation transparency, etc.), communication support mechanisms (communication protocols), and other application-specific support mechanisms. As the notions of processor, memory, operating system play a more indirect role (in providing system-level support), the term 'engineering model' is used here in a more specific sense to describe a framework oriented towards the organization of the underlying infrastructure consisting of structures and mechanisms which enable and regulate distribution.

1.4.5 Technology Viewpoint: The technology viewpoint concentrates on the realized technical components, or the real-world artifacts, from which the distributed processing system is built. The technology model identifies the possible technical solution to the engineering mechanisms. The technical artifacts (realized components) from which the network of information system is built include the hardware and software comprising the local operating system, communication system components, etc. The technology model shows how these technical artifacts are mapped to the (technology independent) designs identified in the engineering viewpoint.

An example of technology specification is the prescription that the communication support will be provided by OSI stack, that inter-node communication will employ X.25, or the protocol used to convey file data will be FTAM. A more detailed technology view of a specific environment would also specify: specific support environment technologies, e.g., Sun running Unix, Vax machines running VMS, etc.

1.5 Summary of Viewpoints

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.

A summary of ODP viewpoints is given in Table 1. below.

Table 1: Summary of ODP Viewpoints

View-point	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.

View-point	Enterprise	Information	Computation	Engineering	Technology
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, stubs, binders, 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.

PART-2 COMPUTATIONAL MODEL

2.1 What is a computational model

The ODP computational model is a framework for describing the structure, specification and execution of the (components of the) distributed application on the distributed computing platform. It is the abstract model to express the concepts of the computational viewpoint.

The computational model provides a set of basic (albeit, abstract) concepts and elements for the construction of a distributed programming (specification) language for which the model does not provide any syntax. Using the computational model, one can specify (program) a distributed application without worrying about the details of the underlying distributed execution platform (the engineering model). The design principle of the computational model is to minimize the amount of engineering detail that the application programmer is required to know, yet at the same time allowing the programmer to exploit the benefits of distributed computing. For example, the computational model allows the programmer to code interaction between application components without having to deal with the distribution of program components. Similarly, the model allows the programmer to express the interaction requirements (such as distribution transparency requirements) without explicitly dealing with the details (mechanisms) of interaction support.

The computational model focuses on the organization of applications into distributable components, identification of interactions between application components, and the identification of the distribution requirements (from the underlying distributed execution environment) for the support of interactions between application components.

The *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.

2.2 Computational model: An Object-Oriented view of distributed application

The computational model is based on a *distributed-object model*. It prescribes an object-oriented view [33] of the distributed application. Applications are collections of interacting objects. In this model, objects are the units of distribution, encapsulation, and failure.

As shown in Figure 2., 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 [9]. 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 at the interfaces of that object [15].

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 objects is modelled as (remote) invocation of an operation [15].

The computational model provides a view of the underlying ODP platform as a distributed, multi-tasking abstract machine supporting (concurrent) objects and interactions between objects.

2.3 Distribution Issues in the Computational Model

The computational model places few constraints on the extent to which application programs can be distributed. Most of the constraints on distribution of application components stem from discussion in other projections, such as enterprise viewpoint or information viewpoint.

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.

2.4 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 [16].

The basic elements of the computational model are: *computational object*, *computational interface*, *operation* invocation at computational interface, *activities* that occur within a computational object, *environment constraints* on operation invocation, etc.

This section is a brief introduction of these basic computational elements out of which the *computational specification* of the distributed application is constructed. The definitions are introduced in terms of the *templates* (specification) of the corresponding elements.

2.4.1 Activity: Activity is agency by which computations make progress [15]. It is the unit of concurrency of the computational object. A computational object may have multiple activities threading through it, of which one or more may actually be executing on a processor at any one instant, depending upon the number of processors available. An activity may pass from one object to another by the first invoking an *operation* on the interface of the second. Activities may split into parallel sub-activities and later recombine. New activities can be initiated to proceed in parallel, independent of their initiating activity.

2.4.2 Computational Operation: Computational objects may support multiple interfaces as *service* provision points. A *service* is an association between object state (some data) and the programs that operate upon them [15]. The ways that a user can interact with a service are completely defined by the set of *operations* that the service supports. Operations affect the state of the object. An operation is a service primitive. Each operation has two parts: the *operation signature* which defines how the operation is invoked by a use of the service (*client*), and the *operation bod*y, which is the piece of program code executed by the provider of the service (*server*) when that operation is invoked.

An *operation signature* template has three parts [15]:

1. The *operation name* is an intrinsic part of the operation. When a client wishes to invoke an operation in a particular server interface it identifies it by its name within that interface. To ensure that there is no ambiguity, no two operations in the same service may have the same name.

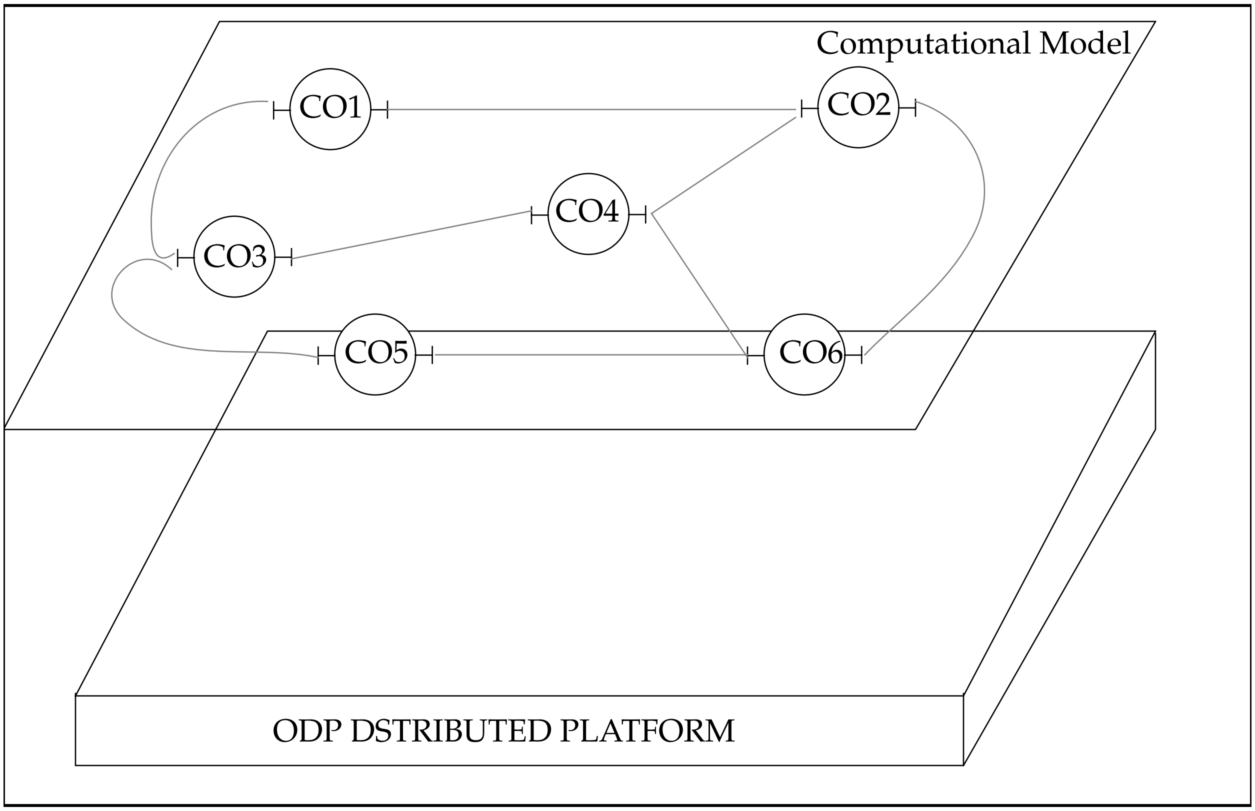

Figure 2. ODP COMPUTATIONAL MODEL: An object world

2. The *parameter part* of an operation specifies the number and types of the parameters and the order in which they are passed to the operation when it is invoked.
3. The *result part* of an operation specifies the number and types of result for each possible outcome from the operation.

Operations have distinct outcomes, each of which can convey different numbers and types of results. An operation's possible outcomes are called *terminations*, and are distinguished by their names. For convenience one outcome from each operation can be left unnamed; this is called the *anonymous termination*, and is conventionally used to represent the normal or expected outcome, while named terminations are often used to represent unusual or unexpected results.

In the computational model, the (engineering) infrastructure failures in invoking an operation on a (remote) interface are reported (to the clients) by the infrastructure objects through the use of termination mechanisms. This permits the detection of invocation failures in the infrastructure.

2.4.3 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 application components and of their requirements of distribution are captured in *computational interface templates* (see Figure 3).

Computational interfaces model different interaction concerns of computational objects. An application component acting as a client may request a number of other components to perform operations and thus needs a different interface with each of these. Similarly, the application component acting as a server may perform actions requested by a number of client components. There is no reason to restrict a server to provide interfaces with identical specifications to each of its clients. Allowing a server to provide multiple interfaces with distinct specifications enables a computational specification to directly reflect the different roles identified in the enterprise specification, especially with regard to access control. Multiple interfaces also enable the knowledge of the services provided by the object to be more tightly scoped [16]. For example, a client which is allowed access to an interface of server object (and may not have access to other interfaces of server object) can observe only a subset of the service provided by the 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, interactions are specified in terms of either operational or non-operational interfaces.

2.4.3.1 Operational Interface: The specification of an operational interface template consists of: Operation Specification, Property Specification, Behavior Specification, and Role Indication.

Operation Specification: The definition of operations that are supported by the interface. Operation specification includes:

1. Operation name: Each operation has a local name within an interface template. No two operations, within the interface, may have the same name.

Data Specification:

2. The number, sequence, and type of arguments that may be passed in each operation.
3. The number, sequence, and type of results that may be returned in each termination.

The operation name together with the type of argument and result parameters constitutes the *operation signature*. Both the operation names and the arguments can be represented as abstract data types.

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 [5]:

* information affecting how the infrastructure supports the interactions; this informa-

tion constrains the type of distribution transparencies, choice of communication protocols, etc. that must be placed in the interaction path between the interacting objects.

* the behavior (or the semantics) of the service offered at the interface; an interface is viewed as a projection of an object's behavior, seen only in terms of a specified set of observable actions.

As a result, signature compatibility is less discriminating than interface compatibility. Interface compatibility includes not only the signature compatibility of the operations (supported by the interface) but also the behavior observable at the interface and the check on the property specification (see below).

Property Specification: The property specification in the computational interface template defines the following attributes:

1. distribution transparency requirement on operation invocation (e.g., migration transparency, transaction transparency, etc.).
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 (e.g., an object be located on a fault-tolerant computer system).
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. Property specification is an important component of the computational interface template and has a direct relationship to the realized engineering structures and mechanisms.

Behavior Specification: It defines the behavior exhibited at the interface. All possible ordering of operation invocations at or from this interface can be specified. This includes ordering and concurrency constraints between operations as well as sequential and parallel operation invocations. The behavior constitutes the protocol part of the interface.

Role Indication: In general, an interface specification may be bi-directional and specify the operations each member of a pair of application components could request the other to perform. However, the ODP computational model also contains uni-directional interface specifications which directly support client-server interaction.

Often an object assumes the role of either *client* or *server*. All interactions of an object, both as a client and as a server, between it and its environment occur at object interfaces. It is convenient to partition client-role interaction concerns from server-role interaction concerns in different interfaces.

2.4.3.2 Non-operational Interface: The computational objects may both 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 *non-operational interfaces* (also known as *stream interfaces)*.

The non-operational interface is a set of information flows whose behavior 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 [18].

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 non-operational or 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 Constraints: 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.

2.4.4 Computational Object: The components of a distributed application are represented as computational objects in the computational model. The computational objects are the units of (application) structure and distribution. A computational object is an encapsulation of (application-specific) state and mechanisms which are not directly accessible to any other object. The computational objects model both the application

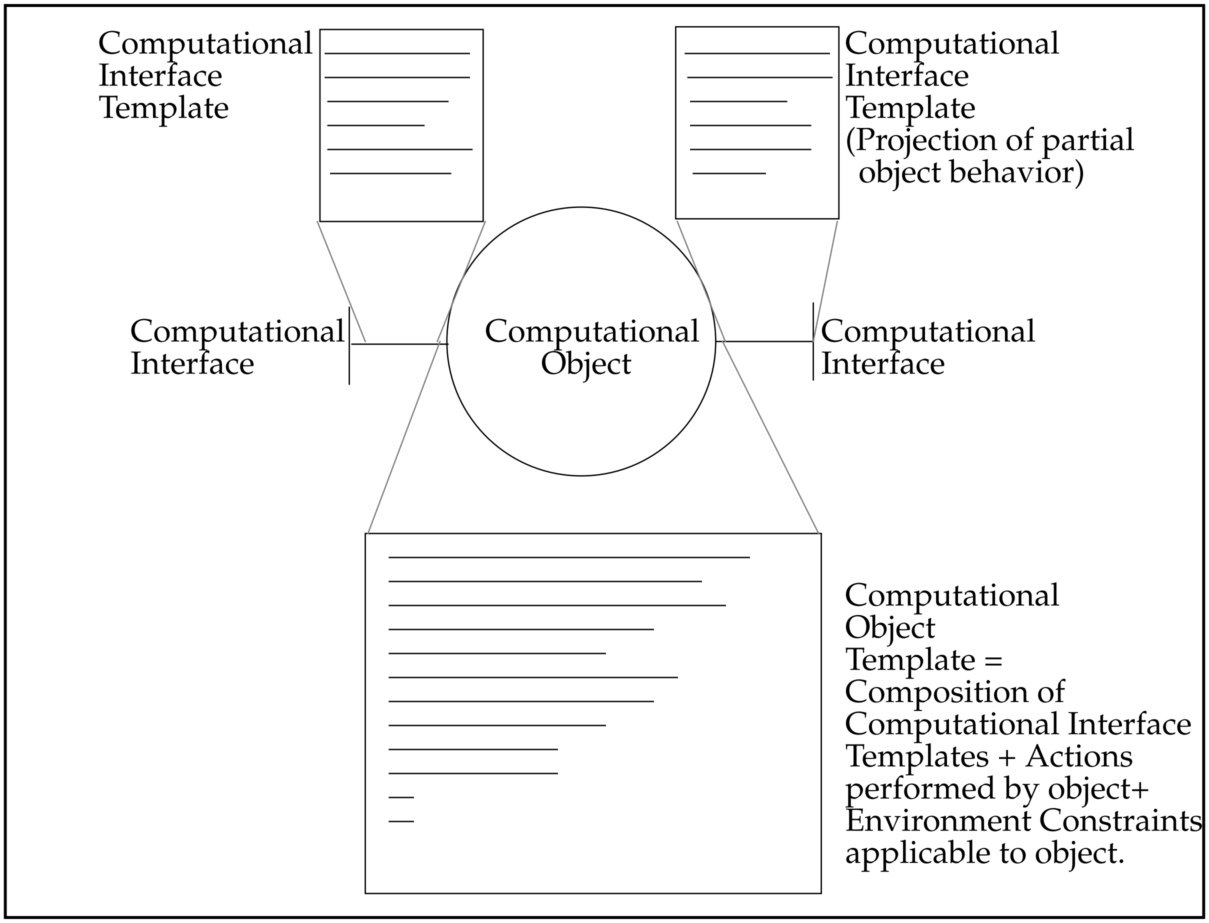

FIGURE 3. ODP Computational Model Concepts

components that perform information processing and those components that store the information. Objects can create interfaces or stop them during their lifetime.

As shown in Figure 3., a *computational object template* consists of:

1. a set of computational interface templates (both operational and stream) which the object can instantiate.
2. an action template for initializing the state of new instances of the object.
3. a specification of environment constraints applicable to the object as a whole.

A computational object can perform the following *activities* [3]:

1. instantiation of interface templates (creating an interface),

2. instantiation of object templates (creating an object),

3. binding to an interface,

4. invoking an operation at an operational interface,

5. reading and writing the state of the object,

6. spawning, forking, and joining actions,

7. stopping of interfaces,

8. stopping of object.

These basic actions can be composed in sequence or in parallel.

2.5 Multiple views on the Computational Model

There are several ways in which the general computational model can be described. This section identifies its major aspects. The computational model can be viewed as:

1. *interaction model* - an environment for interaction between computational objects.
2. *construction model* - construction of the configuration of computational objects.
3. *programming model* - an application programming environment.

Together, these aspects address the issues related to the functional decomposition of the distributed application, inter-working, and portability of application components.

2.5.1 Interaction Model: One view of the computational model is as an environment that supports the existence of and the interaction between computational objects. Computational objects interact by invoking operations at their interfaces. The interaction model defines an *invocation scheme* and a *type scheme* [19].

The invocation scheme describes the permitted forms of interaction, i.e., how clients may use the interfaces provided by the server. It defines the mechanisms for parameter passing between interfaces.

The type scheme provides a set of types into which computational interfaces can be classified. It defines a conformance relation over interface types which are a set of matching rules between interfaces which must be satisfied before a binding between interfaces can be established (Figure. 4).

The interaction model (invocation scheme) is simple and uniform. It is based on the concept of *operation* invocation. The interaction between computational interfaces is via operation invocations which carry input argument parameters. The result of operation execution is returned to the invoker of the operation via *termination*.

The interaction model (invocation scheme) supports two styles of interactions between computational objects (or more precisely between computational interfaces): *interrogations* and *announcements*, to model interactions with and without the reply respectively.

Interrogation is a synchronous *request-response* invocation style; the *activity* that invoked the interrogation passes (via *operation*) to the object that provides the invoked operation, and subsequently returns (via termination) to the object from which the invocation was made. There is no change in the degree of concurrency of the system using an interrogation type of invocation.

Announcement is an asynchronous *request-only* invocation style; a new *activity* is created in the object that provides the invoked operation, and the invoking activity continues in the object from which it made the invocation. Invoking an announcement increases the concurrency in the system, the completion of the evaluation of the body of an announcement decreases the concurrency in the system. The object that invoked the announcement is informed neither of the completion of evaluation (of body of operation) nor of the results delivered (if any). The concept of announcement on the ODP computational model supports the idea of spawning concurrent and independent activities.

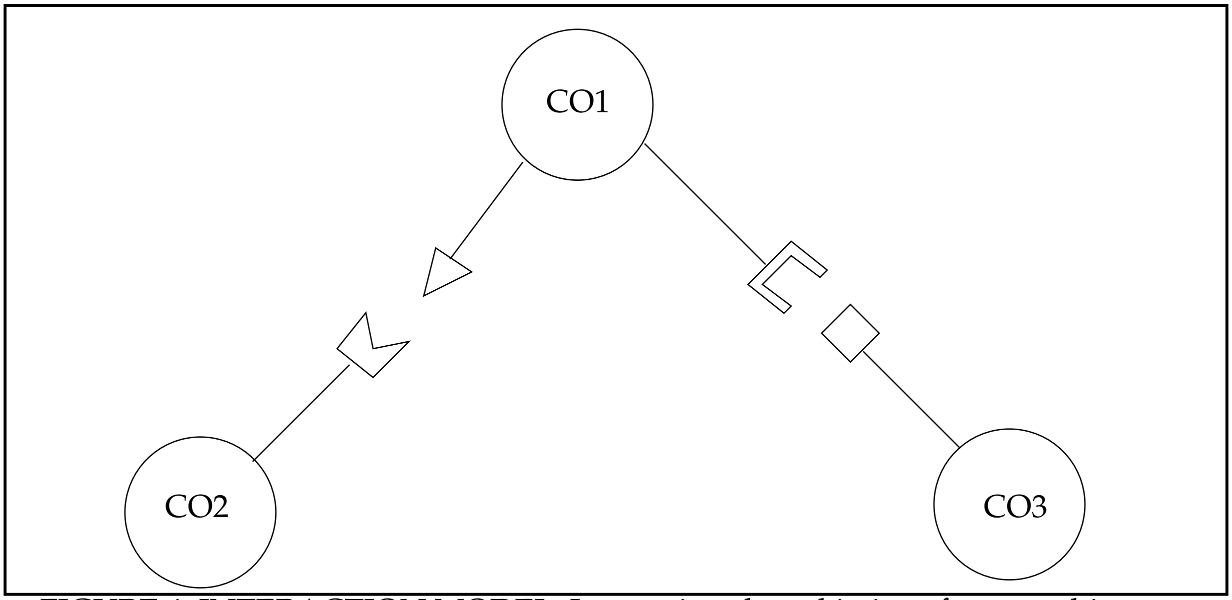

FIGURE 4. INTERACTION MODEL: Interactions based in interface matching

The interaction model is independent of the kind of computational objects that participate in the interaction as well as of the way in which a computational object has been structured internally. The interaction model thus supports the notion of encapsulation and information hiding. This model establishes the interpretation of parametrization and gives failure semantics for the interaction.

2.5.2 Construction Model: The construction model is concerned with the construction of the configuration of computational objects, and supports the creation of complex networks of interacting objects, giving the rules which govern object composition and decomposition.

The computational objects can be connected in various ways, and networks of such objects can be treated as a single computational object. Similarly, a single computational object can be decomposed into networks of computational objects [5].

2.5.3 Programming Model: The computational model provides an abstract, distribution-transparent, language-independent *specification* and *programming model* for distributed applications, and of their execution and interaction semantics. Concerns in this viewpoint essentially include specification/programming language and programming system interface issues. The computational model expresses the *programmability* of the distributed platform [11].

The language-independent *programming framework* offered by the computational model provides:

1. Application programming interfaces (APIs).
2. Programming concepts and abstractions necessary for the development of distributed applications (an abstract programming language).

From this viewpoint an ODP system appears as a large programming environment capable of building and executing applications on the supporting engineering infrastructure. The distributed programming model provided by the computational model, abstracts away, in an integrated framework, the generic set of distributed services provided by the engineering model from distributed applications designers and programmers. The ODP engineering model, that describes the structure and organization of these distribution enabling and regulating services, constitutes a *virtual machine* model for executing distributed programs conforming to the ODP computational model [20].

Hence, the computational model provides the equivalent of a programming language, for use on top of the *abstract machine* realized by the engineering infrastructure. Such a computational model will contain programming language features which are commonly found in advanced object-based distributed platforms. As such, the computational model can be seen as some form of implementation language for building applications on top of ODP systems [21].

The ODP computational model is based on both synchronous (send-blocking) and asynchronous (non-blocking) call and a lightweight threads style of programming. This can very easily be noted since, in the computational model, all object interactions are considered remote and the invocation of interrogation corresponds to the remote procedure call. Such a style is a natural extension of the procedural style found in the majority of programming languages.

The computational model hides the actual degree of distribution of an application from its programmer, thereby ensuring that application programs contain no deep-seated assumptions about which of their components are co-located and which are separated. Because of this, the configuration and degree of distribution of the underlying platform on which ODP applications are run can easily be altered without having a major impact on the applications software [22]. This desirable characteristic is called *distribution transparency*.

Since the main objective of ODP is to provide a generic architecture for distributed systems, the role of the computational model is particularly important. It masks the effects of distribution and heterogeneity, when required from applications.

By conforming to the computational model, application programmers are given a guarantee that their programs will operate in a variety of different quality environments, without modification of the source. The engineering model offers standardized system programming interfaces for supporting the computational programming environment [23].

The computational model concentrates on the problems and the opportunities presented by the execution of application components on distributed computing systems. It identifies the functions that must be available to the programmer and the constrains on the (application) program structure necessary to enable distribution, rather than a particular syntax of the computational language. The outcome of this approach is that all programs, in whatever language, are written with the same abstract (distributed) machine as their target. Porting a program from one system to another is then a matter of only changing the local representation of the abstract machine as it appears in the application programming language, which does not require any changes to the application program itself [3].

PART-3 ENGINEERING MODEL

3.1 What is an Engineering Model?

The engineering model is an abstract model to express the concepts of the engineering viewpoint. It contains 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 [11].

The engineering model is still an abstraction of the distributed system, but it is a different abstraction than that of 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 [24]. 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.

3.2 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 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*[1] 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.

3.3 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 [15]. 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 [17].

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, distri-

1. In contrast, in the distributed operating systems view, the system management would be global and individual computers have little autonomy.

bution requirements); the engineering model prescribes solution as to *how* to realize these interactions, satisfying the stated requirements.

3.4 Structure of the 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 distributed environment. This is illustrated in Figure 5. Some engineering objects correspond to the application functionality and they are referred to as *basic engineering objects* while those which provide distribution functions are classified as *transparency objects*, *protocol objects*, *supporter 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.). However, 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.

3.4.1 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.

A basic engineering object is the corresponding computational object (computationally) enriched with extra interfaces to interact with objects in the channel. In general, a computational object can be mapped onto a single basic engineering object, but (because of refinement, decomposition, and replication) a computational object will often map to several basic engineering objects.

A BEO is an object all of whose interfaces are bound to either other basic engineering objects in the same *cluster* or to (objects in the) *channel*. A BEO is always bound to a *cluster manager* object in the same *capsule* (to enable object deactivation, checkpointing, migration, etc.)

3.4.2 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.

A cluster is a collection of BEOs in a *capsule* such that members of the cluster have no interfaces bound directly to interfaces of objects in other clusters. Objects within a cluster communicate directly, whereas objects in different clusters interact through *channels.*

3.4.3 Cluster Manager: A cluster is associated with a cluster manager which coordinates the management of the cluster. The cluster manager performs the operations of activating a cluster, passivating a cluster, checkpointing a cluster, migrating a cluster, and other

.

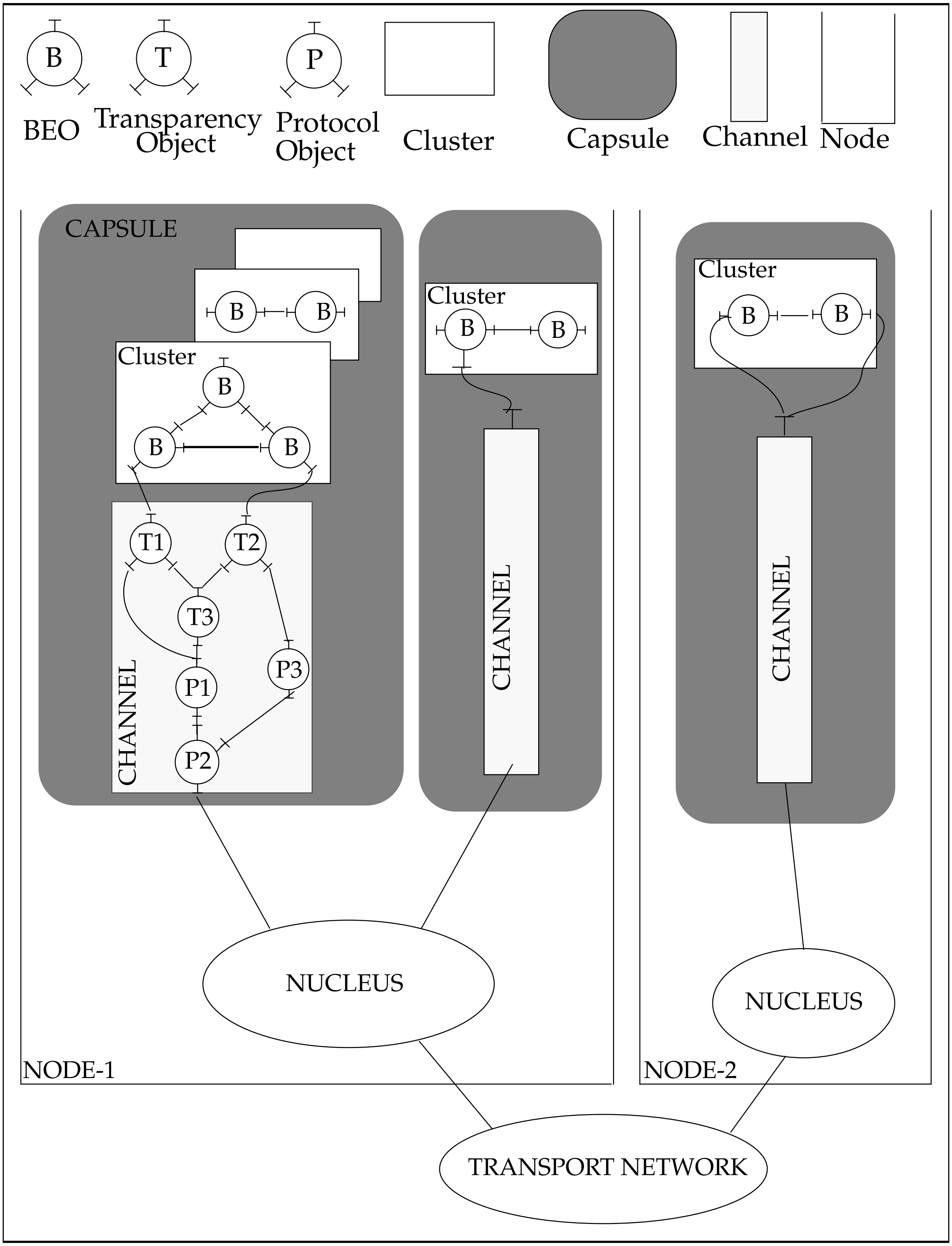

FIGURE 5. ODP ENGINEERING MODEL: Organization of Distributed Infrastructure

policy specific operations.

3.4.4 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.

A capsule is a collection of basic engineering objects, transparency objects, protocol objects in a *node* such that objects in the capsule have no interfaces bound directly to interfaces of objects in other capsules (except via the nucleus). A capsule consists of:

1. active clusters;
2. cluster manager objects, one for each cluster in the capsule;
3. *transparency stub, transparency binder* and *protocol objects* for each *channel* bound to an interface of a basic engineering object within any of the active clusters.
4. a capsule manager. A capsule manager is bound to each cluster manager's cluster management interface.

A cluster is always contained within a single capsule. A capsule is always contained within a single *node*.

3.4.5 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.

A nucleus supports the following interfaces:

1. interfaces to access storage resources, called nucleus *resource interface.*
2. interfaces to access nucleus communication facilities, called *plug* and *socket.*
3. interfaces to access processing resources, called nucleus *interpreter interface*.

3.4.6 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.

3.4.7 Node Manager: The node manager performs the bootstrapping of the node. It initializes the services on the node. It is a repository of capsule templates.

3.4.8 **Channel**: A channel object 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 distribution function requirements of the interaction between basic engineering objects. A general structure of the channel is described in the next section.

3.4.9 Supporting Object: A supporting object is an object, outside of a channel, which cooperates with objects within the channel for the provision of distribution transparency. The supporting objects are shown in Figure 6. The supporting objects are the repositories of information required by the *transparency objects* and *protocol objects* within the channel. For example, the *location transparency binder* object registers and retrieves object locations via a supporting object known as the *relocator*.

Table 2: System Abstractions in the 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	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.

Table 2: System Abstractions in the Engineering Model

Engineering object	System representation
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.

3.5 Structure of a Channel

This section describes the generic structure of the channel which provides the binding between basic engineering objects. A channel supports *distribution transparent* interaction between a pair of (interfaces to) basic engineering objects located in different clusters.

As shown in Figure 6, a *channel is* a configuration of *transparency objects*, *protocol objects*, *application-specific supporting objects*, and *interceptor objects*. It is parametrized by a set of communication interfaces. The configuration of the channel can be dynamically negotiated when establishing the binding between basic engineering objects.

The configuration of objects in the channel provides the medium through which (remote) interactions among basic engineering objects pass.

The channel is composed of a variety of *transparency objects.* The transparency objects that make up the channel are classified as either *stub objects* or *binder objects* (see Figure 6). Both stub objects and binder objects contribute to the provision of distribution transparency between interacting basic engineering objects, but they differ in that the stub objects actually modify the information exchanged across the channel, while binder objects control various aspects of the binding between the interfaces of remote basic engineering objects.

Figure 6 is a simplified view of the channel[1] that illustrates the object types used in

1. In spite of the resemblance of figure 6 to the layered communication models, such as OSI, there is not necessarily a peer-to-peer relationship between the objects in the two halves of the channel. There may exist peer-to-peer relationship between the protocol objects. In contrast, binder objects may obtain the required information from support objects outside the channel.

the structure. In practice, a channel may be much more complex than this and may contain several different types of stub objects, binder objects, etc., depending on the transparency properties required [24].

The figure shows the client half and server half of a single channel object. Note that the whole channel is a single object, even if the BEOs are on different nodes.

3.5.1 Stub Object: A stub is 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).

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 functions such as marshalling and un-marshalling of operation parameters to enable *access transparent* interactions between interfaces of basic engineering objects.

Examples of stub objects include *access transparency objects* and *concurrency transpar-*

ency objects discussed in the next section.

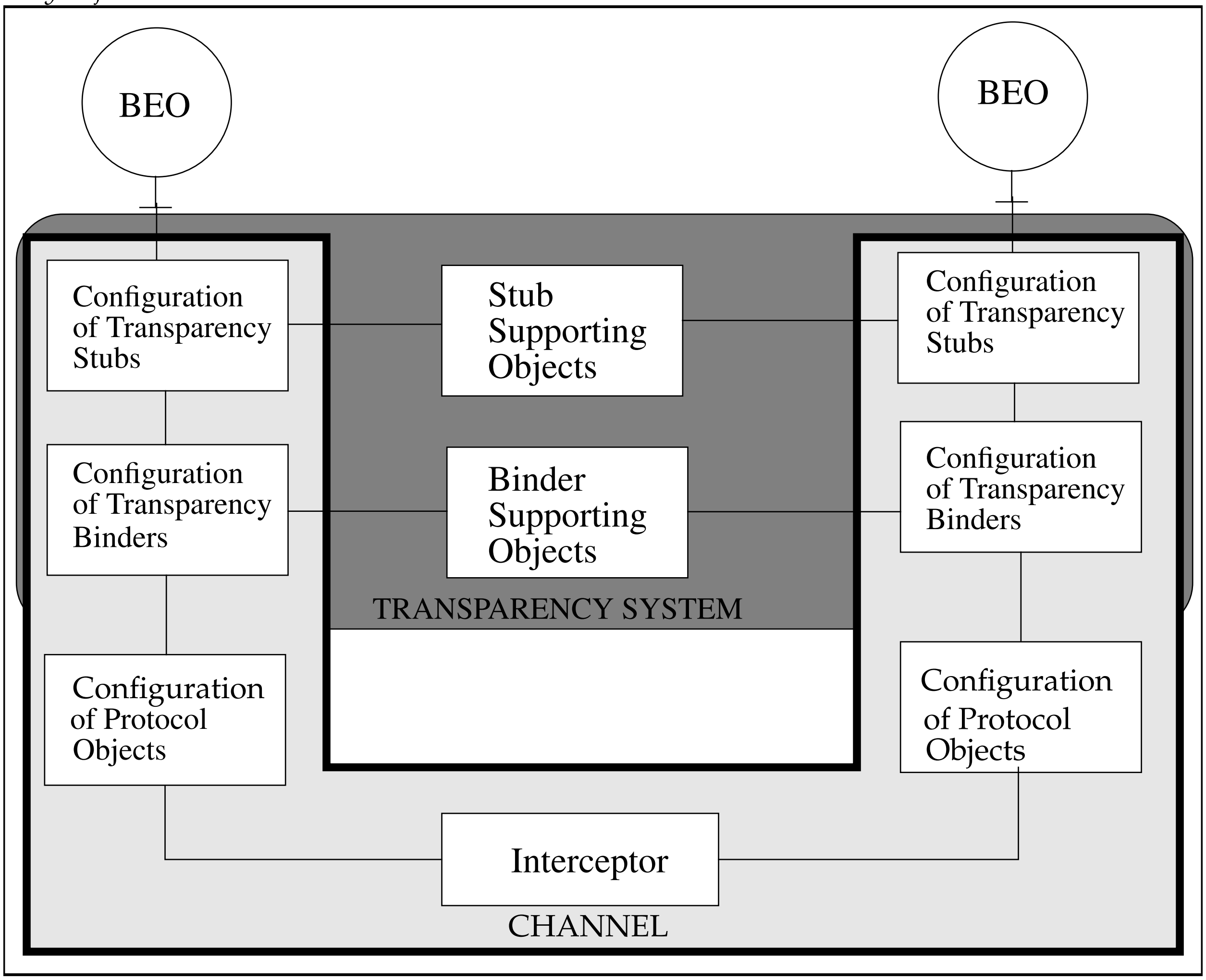

FIGURE 6. SIMPLIFIED GENERIC CHANNEL STRUCTURE

Basic engineering objects are always directly bound to the stub objects. Stub objects within a channel can interact using other objects in the channel, or via interaction with supporting objects outside of the channel (Figure 6). Transparency support through the combination of stubs and binders is discussed in section 3.6.1.

3.5.2 Binder Object: A binder is 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 replicated. Examples of binder objects include *location transparency objects, migration transparency objects, replication transparency objects, failure transparency objects*, and *resource transparency objects*. Transparency support through the combination of stubs and binders is discussed in section 3.6.1.

Stub objects are bound to binder objects (Figure 6). Binder objects interact with each other to maintain the integrity of the binding between the interacting basic engineering objects. Binder objects in the channel can interact with each other using other objects in the channel, or via interaction with supporting objects outside the channel. Binder objects are interconnected by protocol objects.

3.5.3 Protocol Object: A protocol object 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.

When protocol objects are in different (administrative) domains they interact via an *interceptor*. When they are in same domain they interact directly.

3.5.4 Interceptor Object: An interceptor is 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*.

When a channel connects basic engineering objects in capsules supported by a common nucleus object, i.e., in the same node, all of the protocol and interceptor objects in the channel structure can be omitted.

When a channel connects basic engineering objects with no requirement for distribution transparency to support interactions between them, the stub and binder objects can be omitted from the structure.

3.6 Transparency System

Distributed systems exhibit a number of properties, inherent in distribution, not found in centralized systems. Consequently, an application designed to work on a distributed system must take these additional properties into account. However, the application designer does not have to deal explicitly with these properties, if they are made transparent. The complexities of distributed systems may be hidden through the notion of distribution transparencies defined by ODP.

The concept of *transparency* is related to the notion of *abstraction*, where irrelevant details are ignored. Transparency is the property of *hiding* from the user (in the computational environment) some aspects of the potential behavior of the underlying ODP infrastructure [25].

This section describes distribution transparency system (Figure 6) that binds a pair of basic engineering objects within the *channel* of the engineering model. Currently, the engineering model identifies a set of transparency mechanisms, which are by no means exhaustive. There is scope for the definition of more generic distribution transparencies in the engineering model. The distribution transparencies, currently identified, can be used in a broad range of enterprise domains. However, enterprise specific transparency requirements will be identified in the enterprise specific engineering models. It is through the definition of a suitable repertoire of transparency objects that the ODP infrastructure can be made sufficiently flexible to meet a wide range of enterprise requirements [1].

3.6.1 Transparency Support through Stubs and Binders: The transparency objects cooperate to perform the *transparency function* by bringing uniformity to some aspect of the distribution of the engineering objects they support. Some forms of transparency require supporting services: for example, if basic engineering objects can move from one location to another, a means of recording and discovering the current location of the object is required. Supporting functions are modelled as engineering objects so that the architecture provides a maximum degree of configuration flexibility. As shown in Figure 6, the *transparency system* is composed of stub objects and binder objects in the channel, and supporting objects outside the channel.

As mentioned in the previous section, the engineering model classifies transparency objects as either stub objects or binder objects. While stub objects address masking of some aspects of distribution - those arising due to the presence of *heterogeneity* and *concurrency* in the distributed system, the binder objects address aspects of distribution resulting from change of location of objects. The migration of the object may be required for any of the following reasons:

1. load balancing, reduction of access time, etc. This aspect of distribution is masked by *location transparency binder* and *migration transparency binder*.
2. failure of object at one location and its reactivation at another location. This aspect of distribution is masked by *failure transparency binder*.
3. unavailability of (nucleus) resources at one location and its (re)activation at another location. This aspect of distribution is masked by *resource transparency binder*.
4. replication of objects at different locations, for example, if the server object is replicated, then it is required to maintain the binding between the client and the set of replicated server objects. Changes to the membership of the replica group, such as addition of a server object, would require establishing the binding with the new member.

In all these cases the binding between the basic engineering objects is susceptible to be broken down, resulting in a disruption of the service to the client. The binder objects attempt to maintain the integrity of the binding between basic engineering objects.

Hence, they are called transparency *binder* objects. Location transparency binder provides the basic service. All other binders require the support of location transparency binder.

In ODP, the application designer can *select* the level of transparency needed in a design and have full control of other aspects by turning off some transparencies. As a general rule, a transparency is supported by placing the corresponding transparency object between the user and the system, which acts as a filter to hide unwanted system features from the user. By removing the object (i.e., turning off the transparency) the user can directly deal with the system.

ODP permits distribution transparency to be selectively enabled in any binding between basic engineering objects and specifies channel configuration rules to achieve or avoid specific transparencies.

The following transparencies have been identified in the ODP engineering model, as important in distributed systems. A brief description of each transparency, based on the concept of client and server objects (or client and server interfaces) is outlined below with respect to what aspect of distribution is masked by the transparency, the result of the application of the transparency, and a brief description of the transparency mechanism.

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 the distribution transparent programming environment offered by the computational model. The technique for providing transparency services is based on the principle of replacing an original service by a new service which combines the original service with the transparency 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 [26].

Since the interaction 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.

3.6.2 Access Transparency: Access transparency 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).It hides the difference between local and remote provision of the service. It enables interworking across heterogeneous computer architectures, operating systems, and programming languages.

3.6.3 Concurrency Transparency: Concurrency transparency 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.

3.6.4 Location Transparency: Location transparency hides from a user (client) the location of the object (server) being accessed.

3.6.5 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 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.

3.6.6 Replication Transparency: Replication transparency, also known as *group transparency,* hides the presence of multiple copies of services and the maintenance of 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 a service and need not be concerned about making multiple operation invocations or dealing with multiple responses.

3.6.7 Resource Transparency: Resource transparency 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. This 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 a 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.

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 relocation of objects during the "bind-session".	Clients are unaware of the dynamic migration of the server.

Transparency	Central Issue	Result of Transparency
Replication	Multiple invocations of replicated objects, multiple responses, and consistency of replicated data.	Client invokes a replicated server group as if it were a single server. Distribution of requests, 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.

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

3.6.9 Federation Transparency: Federation transparency hides the effects of operations crossing multiple administrative boundaries from the clients. It permits interworking across multiple administrative and technology domains.

PART-4

4.1 Application of ODP Architectural Concepts: An Illustration

This section illustrates the application of ODP architectural concepts discussed in Part-2 and Part-3 of this chapter through a simple client-server example. The distributed application consists of a set of distributed file servers and file users (clients). The clients and servers are located on different *nodes* of a distributed platform. The distributed platform conforms to the ODP engineering model.

In the following we illustrate the *computational* and *engineering* modelling of this client-server distributed application. We focus on a single file user (F-CLIENT) and a file server (F-SERVER) interaction in order to illustrate the issues involved in the computational and engineering modelling of distributed systems. Henceforth the modelling activity is restricted to a pair of client and (possibly replicated) server(s) and interactions between them.

Computational Modelling: From the computational viewpoint, the application consists of a file user and a file server interacting in a distribution-transparent abstraction. The computational modelling activity consists of the identification of:

1. *Computational Objects*: The computational objects in this example are F-USER and F-SERVER. The activities performed by these objects are specified in the corresponding *computational object templates.*
2. *Computational Interfaces*: In this example, each computational object consist of a single computational interface. (More realistically, the file-server object may possess multiple interfaces to model its interactions with different clients).
3. *Computational Operations*: The operations that are supported by (server) interfaces and invoked from (client) interfaces are identified for each computational interface. They are: F-Open, F-Read, F-Write, F-Close, etc.
4. *Environment Constraints*: The environment constraints associated with the computa-

tional object and their interfaces are specified.

The environment constraint associated with the F-SERVER object may include, for example, certain *security* requirements (with respect to the *node* in which that object is placed).

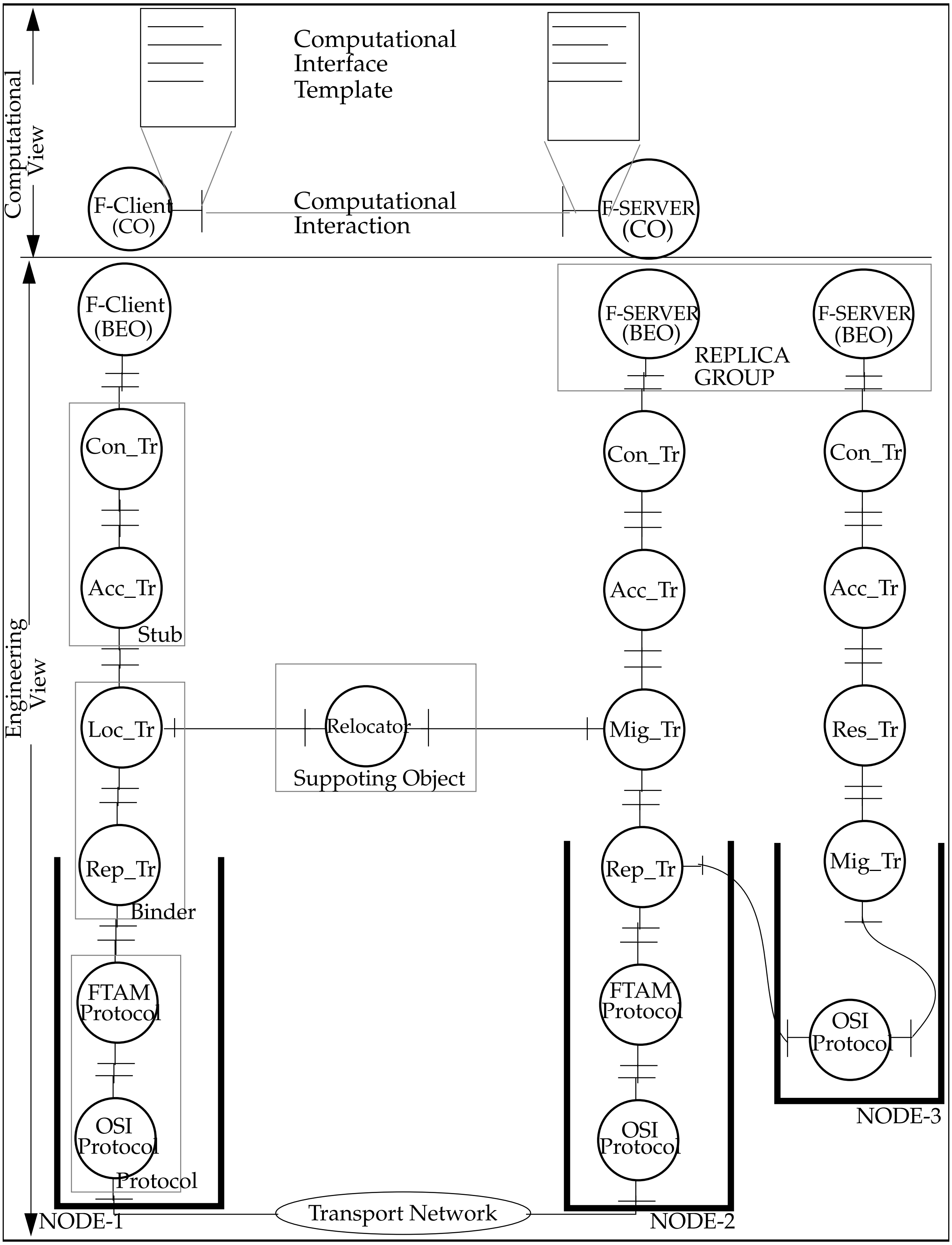

FIGURE 7. CLIENT-SERVER Interaction in an ODP Environment

The environment constraints associated with the F-CLIENT and F-SERVER interfaces include distribution transparency requirements such as access transparency, concurrency transparency, location transparency, replication transparency, communication protocol requirements such as (a specific file transfer protocol) FTAM [27], and connection-oriented session and transport protocols of an OSI stack. The specification of environment constraints in the computational interface template has a direct relationship to realized engineering structures and mechanisms.

5. *Computational Interactions*: The F-USER and F-SERVER objects interact by exchanging operations at their interfaces. The rules for operation exchange constitute the behavior (exhibited by the computational object) at the interface.

Items 3, 4, and 5 are specified as part of the *computational interface template*. The F-CLIENT and F-SERVER interfaces are bound on the basis of the matching of their computational interface templates, performed by a special ODP system object called the *trader* [28, 29]. The computational view of this application is shown in Figure 7.

Engineering Modelling: In passing from the computational viewpoint to the engineering viewpoint, concerns shift from the specification of computational structures (e.g., computational objects, computational interfaces, etc.) and statements of necessary *properties* of interactions between object interfaces (e.g., distribution transparency requirements) to engineering mechanisms capable of realizing these properties.

At the heart of the separation between the ODP computational and engineering models is the idea of a tool-driven transformation between the abstract computational description of a distributed application and its mechanization in terms of the engineering model. The engineering model animates the computational model [30].

The engineering modelling activity consists of identification of *basic engineering objects* corresponding to computational objects, realization of interactions between computational interfaces through the configuration and instantiation of appropriate *channel* structures between the (corresponding) interfaces of basic engineering objects, and the placement of engineering objects in the appropriate environments (*nodes*). The composition of objects in the channel must satisfy the environment constraints specified in the computational interface template.

1. *Basic engineering objects*: The F-CLIENT and F-SERVER basic engineering objects are obtained through a compilation or any other transformation of the corresponding computational objects, and may result in the decomposition of computational objects and/or identification of additional interfaces to BEOs to interact with objects in the channel. The computational interfaces are not decomposed.
2. *Channel*: The channel structure between the basic engineering objects carries the operations invoked by the interfaces of the BEO. In this example, the channel between the F-USER and F-SERVER BEOs is composed of the following engineering objects, which satisfy the distribution transparency requirements of the computational interface.

2.1 *Access Transparency Object*: The F-CLIENT and F-SERVER objects may be coded in different programming languages and compiled on different machines. An access transparency object is interposed to enable them to talk to each other. Each of the access transparency objects on the client and server half of the channel convert the operations into messages in the network format and vice versa. For example the client may be coded in Smalltalk and the server in C. The client uses the Smalltalk *methods* to invoke the server. However the server (written in C) cannot understand Smalltalk *methods*. It can only respond to C *function calls.* The access transparency objects on both sides of the channel convert the local invocation and data parameters into messages in the network format and vice versa.

2.2 *Concurrency Transparency Object*: The F-SERVER may have multiple clients invoking operations on its interface. The concurrency transparency object hides the effect of concurrency existing at the F-SERVER from the F-CLIENT [31].

2.3 *Location Transparency Object*: The F-SERVER object may be migrated on a different node (due to load balancing). If an operation for a migrated F-SERVER is received, the location/migration transparency object on the server side of the channel informs its counterpart on the client side of the channel. The client location transparency object obtains the current F-SERVER address from the *relocator* (a supporting object) and redirects the client operation invocation.

2.4 *Replication Transparency Object*: The example shows that there exists two replicas of F-SERVER. If the F-CLIENT wishes that all its operations on the F-SERVER be transparently performed on server replica (instead of making separate invocations), then the replication transparency objects on both ends of the channel perform the functions of distribution of client requests and server responses, collation, and ordering in order to perform consistent replication [32].

2.5 *Resource Transparency Object*: If the F-CLIENT wishes that the deactivation and reactivation of F-SERVER be transparent between operation invocations, the resource transparency object on the server side performs the reactivation of a passivated F-SERVER when an operation invocation is received for it.

Note: As discussed before, transparencies are selective. Only those transparencies which are specified in the computational interface template are included in the channel. Some transparencies require peer objects (on each side of the channel), because they require peer-to-peer protocols to achieve the transparency.

The objects satisfying the communication requirements of the computational interface are as follows.

In this example, the File Transfer Access and Management (FTAM) protocol is configured in the channel along with the appropriate subset of OSI session and transport protocols to support the communication of operation exchanges between F-USER and F-SERVER over the unreliable transport network.

PART-5

5.1 Conclusion and directions for future research

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 field of Open Distributed Procesing offers numerous research opportunities, related to both theoretical and practical aspects of viewpoint models. One of the main research problems is ensuring consistency between viewpoint specifications. This includes identifying the requirements and constraints that originate in the viewpoint models, identifying the consistency constraints between the models, and defining consistency-preserving transformations between the models. Similarly, it is required to ensure consistency between the system design and the viewpoint specifications.

The computational and engineering viewpoints are the most important from the point of view of distributed software engineering. They offer uniform and consistent abstraction levels for the specification of the system and its engineering on the distributed infrastructure. They offer powerful design concepts for the development of application programming environments and distributed platforms.

The ODP Model is very generic. It can be applied in various application domains. Currently, it is being used in the field of Advanced Intelligent Networks, Distributed Network Management, etc. Its application in other domains is an area of active interest.

References

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

[2] Draft International Standard ITU-T X.902 / ISO 10746-2: Basic Reference Model of Open Distributed Processing - Part-2: Descriptive Model, 1994.

[3] Draft International Standard ITU-T X.903 / ISO 10746-3: Basic Reference Model of Open Distributed Processing - Part-3: Prescriptive Model, 1994.

[4 Draft Recommendation ITU-T X.904 / ISO 10746-4: Basic Reference Model of Open Distributed Processing - Part-4: Architectural Semantics.

[5] ANSA Reference Manual, Volume A, B, C., Release 01.01, Advanced Projects Management Limited, Cambridge, U.K., July 1989.

[6] International Standard ISO 7498: Information Processing Systems-Open System Interconnection-Basic Reference Model, 1984.

[7] CCITT Com. 7 Rxx, "Study Group 7, Q19/7 Distributed Applications Framework (DAF) Report", February 1990.

[8] P.F.Linington, "Introduction to Open Distributed Processing Basic Reference Model", Proceedings of the IFIP TC6/WG6.4 International Workshop on Open Distributed Processing (October 1991), North Holland 1992, 3-14.

[9] S.Proctor, "An ODP Analysis of OSI Systems Management", Proceedings of the Third Telecommunication Information Networking Architecture Workshop, (TINA 92), Narita, Japan, January 1992, 23.2.1-23.2.22.

[10] J.J. van Griethuysen, "Enterprise Modelling, a necessary basis for modern information systems", Proceedings of the IFIP TC6/WG6.4 International Workshop on Open Distributed Processing (October 1991), North Holland 1992, 29-68.

[11] G.Bregant, "Platform Modelling Requirements from the ROSA Project", Proceedings of the Third Telecommunication Information Networking Architecture Workshop, (TINA 92), Narita, Japan, January 1992, 11.1.1-11.1.15.

[12] International Standard ISO 8807: Information Processing Systems-Open System Interconnection- LOTOS- A Formal Description Technique based on temporal ordering of observational behavior, 1988.

[13] CCITT Specification and Description Language (SDL), Z.100, Geneva, March 1988.

[14] International Standard ISO 9074: Information Processing Systems-Open System Interconnection- Estelle- A Formal Description Technique based on Extended State Transition Model, 1989.

[15] ANSA: An Application Programmer's Introduction to the Architecture, TR.017.00, Advanced Projects Management Limited, Cambridge, U.K., November 1991.

[16] ANSA: An Engineer's Introduction to the Architecture, TR.03.02, Advanced Projects Management Limited, Cambridge, U.K., November, 1989.

[17] A. Schill, M.Zitterbart, "A Systems Framework for Open Distributed Processing", Proceedings of the International Workshop on Distributed Systems: Operations and Management, 1992.

[18] ANSA Technical Report: Integrating Multimedia into ANSA Architecture, TR.028.00, Advanced Projects Management Limited, Cambridge, U.K., February

1993.

[19] ANSA Computational Model, AR.001.01, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[20] J.B.Stefani, E.Najm, "A Formal Semantics for the ODP Computational Model", to appear in the CN&ISDN special issue on Open Distributed Processing.

[21] J.B.Stefani, "Open Distributed Processing: The Next Target for the Application of Formal Description Techniques, Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'90, 427-442.

[22] ANSA Technical Report: DPL Programmers Manual, TR.031.00, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[23] J.B.Stefani, "Towards a Reflexive Architecture for Intelligent Networks", Proceedings of the Second Telecommunication Information Networking Architecture Workshop, (TINA 91), Chantilly, France, March 1991, 1-13.

[24] C.J.Taylor, "Object-Oriented Concepts in Distributed Systems", Computer Standards and Interfaces, Vol(15), Nos 2/3, 1993.

[25] 5th Deliverable, The ROSA Architecture, Release Two, RACE Project R1093, May 1992.

[26] ANSA: A System Designer's Introduction to the Architecture, RC253.00, Advanced Projects Management Limited, Cambridge, U.K., April 1991.

[27] International Standard ISO 8571: Information Processing Systems-Open System Interconnection-File Transfer Access and Management, Part-1 - Part-4, 1988.

[28] J.Indulska, K.Raymond, M.Bearman, "A Type Management System for an ODP Trader", Proceedings of International Conference on Open Distributed Processing,

ICODP-93, Berlin, September 1993, 169-180.

[29] A.Goscinski, Y.Ni, “Object Trading in Open Systems”, Proceedings of International Conference on Open Distributed Processing, ICODP-93, Berlin, September 1993, 145-156.

[30] A.Watson, ISA Project Report: Types and Projections, Ref: APM/RC/.258.03, Advanced Projects Management Limited, Cambridge, U.K., April 1992.

[31] J.P. Warne, O.Rees, “ANSA Atomic Activity Model and Infrastructure, AR.004.01, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[32] ANSA: A Model for Interface Groups, AR.002.01, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[33] J.Rumbaugh, M.Blaha, W.premerlani, F.Eddy, W. Lorensen, “Object-Oriented Modelling and Design”, Prentice Hall, 1993.