INTRODUCTION TO ODP ENGINEERING MODEL

BY

KAZI FAROOQUI,

LUIGI LOGRIPPO,

DEPARTMENT OF COMPUTER SCIENCE,

UNIVERSITY OF OTTAWA,

OTTAWA K1N 6N5, CANADA.

E-mail: farooqui@csi.uottawa.ca, luigi@csi.uottawa.ca

This paper presents a tutorial introduction to the major features of the RM-ODP Engineering Model. The elements of the engineering model are introduced and briefly explained. The model is described as an object-based distributed platform for the support of distributed applications modelled in the computational model. Major elements of the engineering model which enable, regulate, and hide distribution are presented.

1. What is an Engineering Model

The classical counterparts of the computational models are the engineering ones which are of interest to operating system and communication experts. The engineering model 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.

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

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

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 set of basic services and mechanisms, identified in the engineering model, are modelled as a collection of interacting objects which together provide support for the *realization* of interactions between distributed application components.

The engineering model can be considered as an extended operating system spanning a network of interconnected computers. In the *networked-operating system* view of the engineering 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. Engineering Model: Animation of Computational Model

The interest of the computational model is directly related to the existence of a mapping enabling it to relate to engineering concerns. This means, for instance, being able to map computational concepts onto the engineering structures.

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

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

4. Structure of Engineering Model

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

There are different kinds of engineering objects in the engineering model corresponding to different distribution (enabling, regulating, hiding) functions required in distributed environment. 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, 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 (figure 1). The objects and structures which are defined later in the text are italicized.

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.

BEO environment rules: It 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 *nucleus resources interface* of the *nucleus* object (to access nucleus resources)

* a *cluster manager* object in the same *capsule* (to enable object deactivation, checkpointing, migration, etc.)

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.

Cluster environment rules: It is a partition of BEOs in a *capsule* such that members of the partition 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.* A cluster is a:

unit of distribution: objects within a cluster must reside in the same capsule.

unit of storage: all objects in a cluster are stored and retrieved at the same time. It is a unit of activation and deactivation to storage medium.

unit of migration: all objects in the cluster must move to the destination capsule when migration takes place.

unit of operation: within a cluster there is no partial operation-if a cluster is operational (active) then all objects in the cluster are operational.

unit of interaction: objects within the cluster can communicate in non-ODP conformant ways.

unit of replication: if an object needs to be replicated, all objects in the cluster must be replicated.

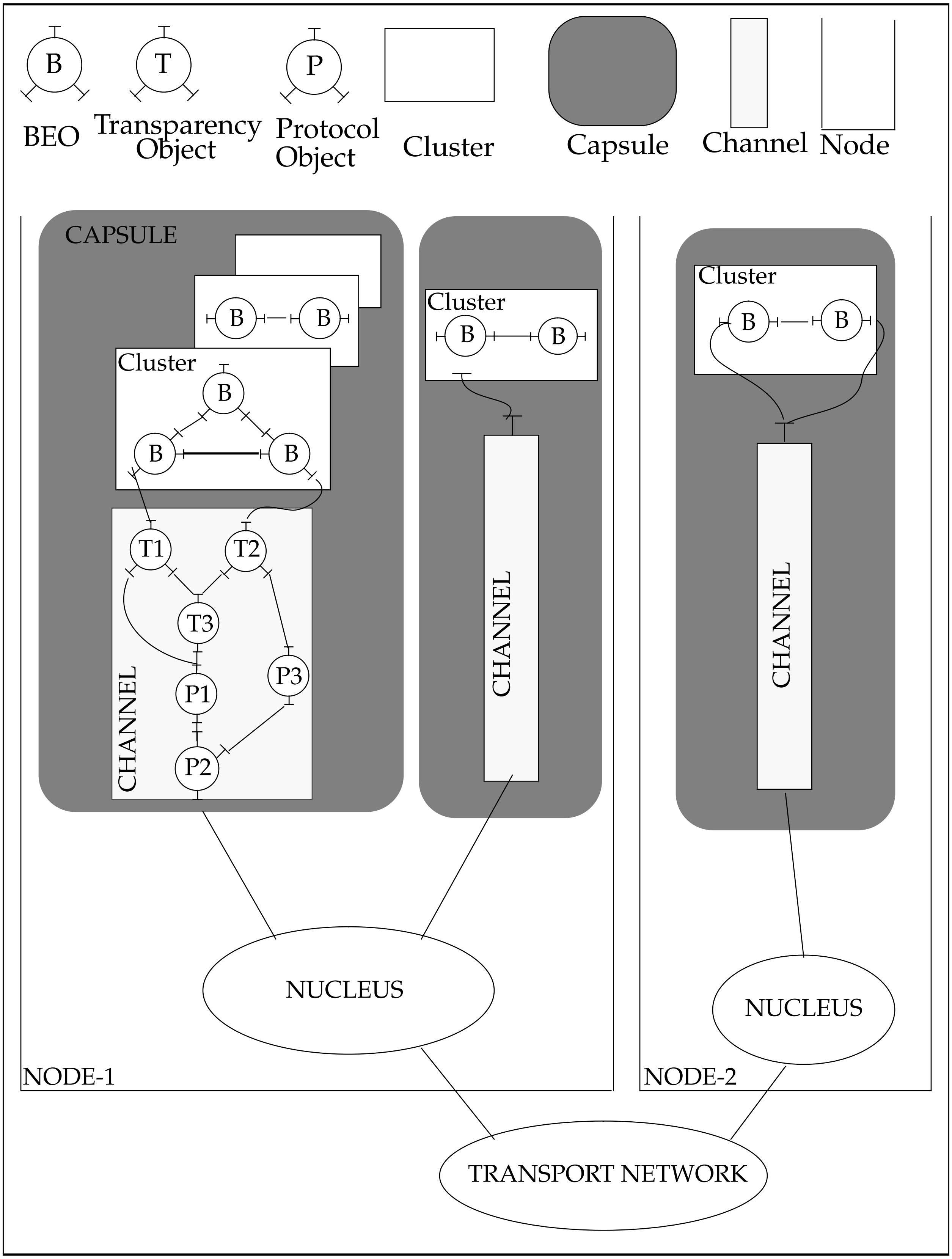

FIGURE 1. ODP ENGINEERING MODEL: Organization of Distributed Infrastructure

unit of security: the same security policy applies to all the objects in the cluster. Security is not an issue for interaction between objects in the same cluster.

unit of dependability: objects within the cluster have the same level of dependability.

unit of availability: objects within the cluster have the same level of availability. A cluster is not a unit of access control. Access control will have to be applied for each requests to use objects in a cluster.

The environment of the cluster is shown in figure 2. A *cluster template* specifies the initial configuration of objects in the cluster and an initialization activity. Cluster template instantiation is performed by a *capsule manager*. A cluster can be in one of two states:

active cluster: cluster instantiation creates an active cluster. An active cluster has nucleus resources allocated to its objects. All objects in an active cluster are immediately available for invocation.

passive cluster: deactivating an active cluster destroys the active cluster (object deletion) and creates a passive cluster representing the same state.

A passive cluster has no nucleus resources allocated to it and is not available for direct invocation. A cluster is passivated for resource management requirements.

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

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.

Capsule environment rules: It is a partition 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 nucleus). A capsule is a unit of failure-if a capsule fails, all clusters in the capsule fail. Clusters within the capsule cannot fail independently. A capsule consists of:

* active clusters;

* cluster manager objects, one for each cluster in the capsule;

* *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.

* a capsule manager. A capsule manager is bound to each cluster manager's cluster management interface.

An active cluster is always contained within a single capsule. A capsule is always contained within a single *node*. The structure of a capsule is shown in figure 3.

Capsule instantiation is performed by the *nucleus* object, with reference to a *capsule template* which specifies the initial configuration of objects in the capsule. A capsule can be destroyed when it no longer contains any clusters capable of further activity.

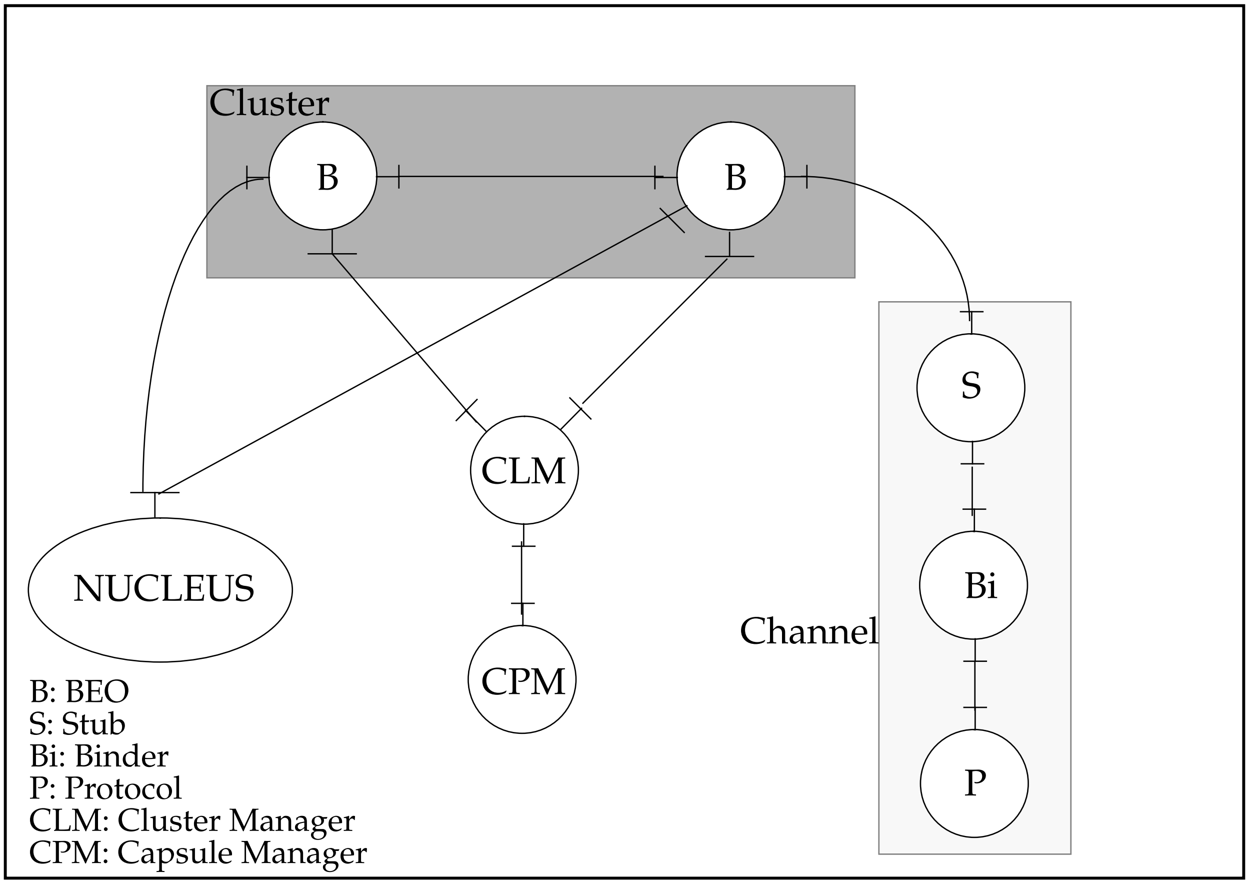

FIGURE 2. Structure Supporting Cluster of BEOs

Each capsule has an interface to the nucleus which provides:

* operations for (object's) threads execution and synchronization (access to processing resource of nucleus);

* operations for resource management (access to storage resources of nucleus);

* communication *plugs* and *sockets* (access to communication resources of nucleus)

4.5 Capsule Manager: The capsule manager is responsible for the management of clusters in the capsule. Each cluster manager in the capsule is bound to the capsule manager.

4.6 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 (figure 1). A nucleus has the capability of interacting with other nuclei (through its communication function), providing the basis for inter-capsule and inter-node communication.

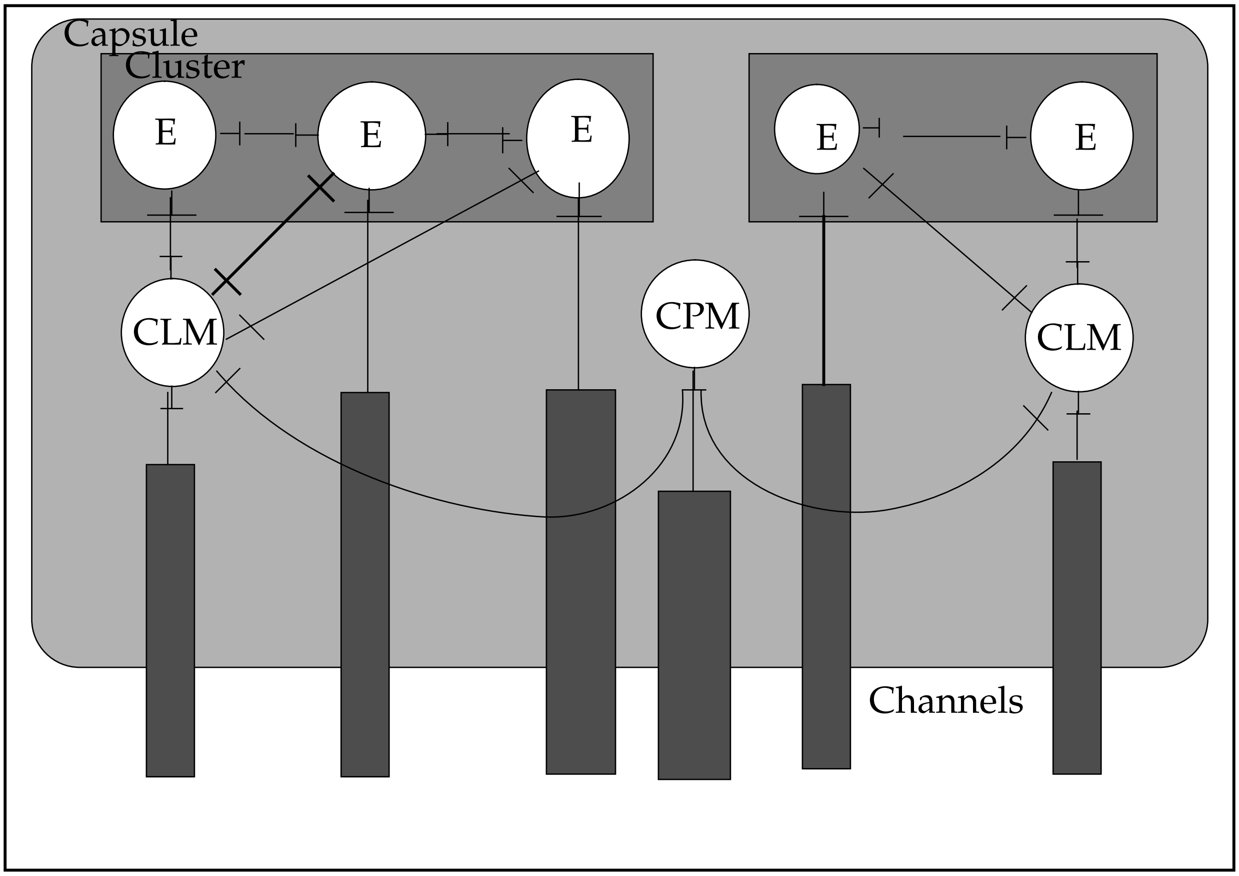

FIGURE 3. STRUCTURE OF CAPSULE

Nucleus environment rules: Nucleus supports the following interfaces:

* interfaces to access storage resources, called nucleus *resource interface.*

* interfaces to access nucleus communication facilities, called *plug* and *socket.*

* interfaces to access processing resources, called nucleus *interpreter interface*.

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

A node is an engineering unit of resource independence; a node is a resource management domain. The procedure for node instantiation is outside the ODP framework.

4.8 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.

4.9 Channel: A channel is a configuration of *transparency objects*, *protocol* objects, and *interceptor* objects 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.

4.10 Supporting Object: A supporting object is an object, outside a channel, which cooperates with objects within the channel for the provision of distribution transparency. 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 1: System Abstractions in Engineering Model

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

5. Structure of 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.

A *channel template* specifies a configuration of *transparency objects*, *protocol 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 engineering objects in the channel provide the medium through which (remote) interactions between 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.* 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 4 is a simplified view of the channel that illustrates the object types used in the structure. In practice, a channel may be much more complex than this and may contain several different subtypes of stub objects, binder objects, etc., depending on the transparency properties required.

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

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

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

Stub objects add further interactions and/or information to interactions between interacting basic engineering objects to support distribution transparency. As an example, a stub object may provide adaptation 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 object* and *concurrency transparency object* discussed in the next section.

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

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

Binder objects maintain the binding between basic engineering objects, even if they migrated, reactivated at new location, or are replicated.

Examples of binder objects include *location transparency object, migration transparency object, replication transparency object, failure transparency object,* and *resource transparency objec*t.

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

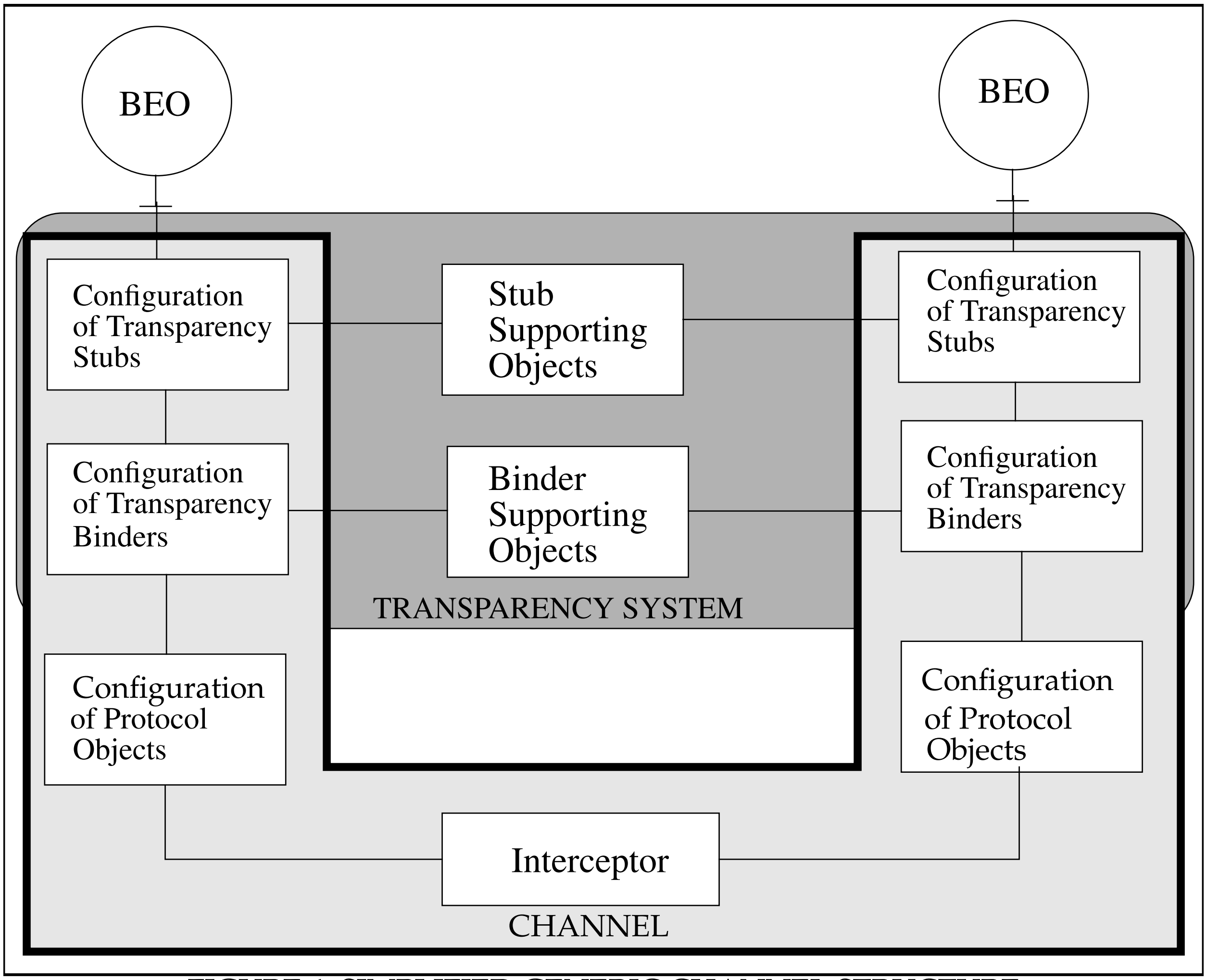

FIGURE 4. SIMPLIFIED GENERIC CHANNEL STRUCTURE

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

Protocol environment rules: When protocol objects are in different (administrative) domains they interact via an interceptor. When they are in same domain they interact directly.

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

Interceptor Rules: In order to perform interception, interceptors need access to the types of the basic engineering object interfaces they interconnect.

For interaction to occur, basic engineering objects must agree on a (number of) transfer syntax capable of representing data types including interface references, operation names, termination names, etc.

Channel Rules: 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.

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, it need not be the case that the application designer has to deal explicitly with these properties, if these properties are made transparent. The complexities of distributed systems may be hidden through the notion of distribution transparencies defined by ODP.

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

This section describes a distribution transparency system that binds a pair of basic engineering objects within the *channel* of the engineering model. The engineering model, currently, 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, so far 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.

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. The *transparency system* is composed of stub objects and binder objects in the channel, and supporting objects outside of 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 necessitated, in distributed systems, by any of the following reasons:

1. load balancing, reduction of access time, etc. This aspect of distribution is masked by the *location transparency binder* and the *migration transparency binder*.

2. failure of object at one location and its reactivation at another location. This aspect of distribution is masked by the *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 the *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 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. The location transparency binder provides the basic service. All other binders require the support of the location transparency binder.

6.2 Selective Transparency: 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 explicitly 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.

7. Distribution Transparencies

The following transparencies have been identified in the ODP engineering model, as important in distributed systems. The concept of transparency is viewed as the corner stone of ODP standardization. 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 applying the transparency and a brief description of the transparency mechanism:

1. Access Transparency

2. Concurrency Transparency

3. Location Transparency

4. Migration Transparency

5. Replication Transparency

6. Resource Transparency

7. Failure Transparency

8. Federation Transparency

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

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

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.

7.1 Access Transparency

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

Transparency Mechanism: The mechanisms for the provision of access transparency (between interacting client and server objects) are encapsulated in *client access transparency stub* and *server access transparency stub*. These two stubs together provide access transparency to the client and server basic engineering objects.

The client access transparency stub acts as a proxy for the server object, offering the server interface locally. It accepts the client's local invocation (e.g., a procedure call) and converts (*marshals*) the operation name and parameters into a request message using some agreed transfer syntax.

The request message is sent to the server access transparency stub object which *unmarshals* the operation name and parameters from the request message and makes a local call to the server.

The termination name and the associated result parameters returned by the server (in response to operation invocation) are marshalled into a response message by the server access transparency stub.

When the response message arrives at the client access transparency stub, it is unmarshalled and the response is returned as an operation termination (e.g., a local call return) to the client basic engineering object.

Access transparency implies that there is only one invocation mechanism for both local and remote services. The services must also appear the same and this can be done either by making the service appear to be local or by making it appear to be remote regardless of which has actually been provided.

Making the service appear remote means that the user of the service must allow for communication failure responses although the probability of such a response from a local service is zero.

Making the service appear local means introducing a transparency mechanism which intercepts the communication failure responses and does something about them. The mechanism may incorporate a recovery strategy that attempts to achieve the desired result or it may report some kind of failure that is possible for the local service. The options for the recovery strategy may depend upon which other transparencies are in force.

Result of application of transparency: Access transparency enables interworking across heterogeneous computer architectures operating systems and programming languages.

7.2 Concurrency Transparency

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

Basically, this transparency masks scheduling of operation invocations that act on shared state to satisfy the ACID properties discussed below.

Transparency mechanism: The mechanisms for the provision of concurrency transparency (to the client object from the concurrency existing at the server object) are encapsulated in the *concurrency transparency stubs* in the client and server side of the channel. This transparency is achieved by the realization of the following properties, referred to as ACID properties:

1. *Atomicity*: This property ensures that the effect of an operation is “all-or-nothing”; it should either complete or leave the state of the system unchanged. This can be achieved by adding ‘commit’ or ‘revert’ results in the terminations of *atomic* operations. It requires the use of commitment, coordination and recovery mechanisms by which an activity threaded through multiple computational objects can negotiate whether to confirm to a set of potential (distributed) state changes, or to move back (*revert*) to the original (distributed) state.

2. *Consistency*: The “serializable” property of consistency preserves the integrity of distributed object state by preventing interference between concurrent activities. It ensures that invocations are scheduled with maximum concurrency and without conflict. It can be achieved by associating ordering predicates with interfaces, where the predicate describes the permitted sequence of non-conflicting invocations. It requires the use of mechanisms for the generation of serializable (or consistent) schedules and of the locking mechanisms for reading and writing shared state.

3. *Independence*: The independence or *isolation* property controls the way in which the results of atomic activities (and their sub-activities) become visible to others. The property ensures that partial results of an operation (intermediate states) are not visible to other operations until they are 'committed' or 'reverted' back. This property not only avoids the problem of interference but also the related problem of cascading reversions.

4. *Durability*: The "permanence of effect" property of durability is concerned with making the effects (commit or revert) of atomic operations permanent. It requires the provision of failure resilient repositories for recording copies (versions) of the effects of committed and reverted atomic activities.

Result of application of transparency: The ACID properties together aid in the provision of concurrency transparency and mask any *effect* due to other concurrent users of a service from individual users of the service.

7.3 Location Transparency

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

Transparency mechanism: Objects (and their interfaces) in a distributed system may change their location to achieve load balancing or to reduce communication latency, they may be checkpointed and restarted at different locations, or an object (and hence its interface) can be accessed by several (protocol) paths and hence several network level names by which it is known.

Location transparency implies that there is some sort of mapping from the symbol that identifies a service to something that identifies the location of the provider of the service. Like other transparencies, location transparency within the engineering model controls the visibility of certain kinds of failure (in the computational model). As an example, an operation invocation may fail if the symbol that identifies a service contains location information and the service is not at the specified location (due to migration), and a response given to the invoker saying "service not known here". With location transparency in place, such a response in not possible. The location transparency mechanisms locate the service and redirect operation invocations to changed location.

In the ODP engineering model, the mechanisms for the provision of location transparency between two basic engineering objects are encapsulated in the *location transparency binders* in the client and server of the channel and in the supporting object (outside the channel) known as the *relocator*. A relocator is a repository of interface locations which is updated whenever an interface (supported by the object) changes its location. The location binder objects have two main functions:

* to inform the relocator of the location of the interface it supports

* to obtain the location of the migrated interface from the relocator.

The location transparency binders typically cache location information. If the location of the interface changes, the use of old location will cause an error. The location transparency binder object will then obtain the new location from the relocator object and re-invoke the operation. This activity remains transparent to the communicating basic engineering objects.

Result of application of transparency: Object invocations are location independent.

7.4 Migration Transparency

Hides: 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.

Transparency mechanism: A service is migration transparent if a user of that service is unaware of change of location that takes place while an invocation is in progress. A non-migration-transparent service includes a "moved" response that indicates that the service has moved to a different location during the invocation of an operation. A migration transparency mechanism would catch this response and take recovery action.

Migration transparency requires mechanisms to cleanly disconnect an object (i.e.,*cluster*) from the source location (source *capsule*) and plant it in destination location (destination *capsule*). All local interface references in the object must be converted to remote ones and, if interactions with other objects are in progress, the state of these interactions must be preserved at the destination site and the appropriate redirections left behind at the source site. It may be that requiring that an object be idle before it migrates is a reasonable restriction. Objects must be passivated before migration, and reactivated afterwards. The bottom level mechanism will be an operation *freeze* which returns a passive representation of the object (cluster) and a *thaw* operation to convert the passive representation back into an active object.

Result of application of transparency: The relocation of the servers from one location to another, while the clients are interacting with them, is made transparent to clients.

7.5 Replication Transparency

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

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

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

Replicated service providers: A non-transparent replicated service appears as a set of services. The user of the service would invoke some or all of these services *separately*, and then *collate* all of the results.

A *replication transparent* provision of this service would appear to be of the same type as the one of the services in the set (referred to as the *server group*). The transparency mechanism is responsible for selecting the members to which the client operation invocations are to be directed and then collating the results; it has the non-transparent view described above. This full transparency is typically used to provide failure resilience, dependability, performance, etc., but may also be used to spread the work over a number of servers.

Replicated service users: The user of a service may be replicated, and in this case it will be called a *client group*. This means that there will be multiple requests from different objects for what is really a single invocation. The transparency mechanism must intercept all of the requests and present a single request to the service provider (which may itself be replicated). It must also deliver the response to all of the requestors.

Replication types: There are two types of replication schemes: *active replication* and *passive replication*.

In an *active* replication mechanism every member of a replica group receives every invocation made on the group and computes its own result.

In a *passive* replication scheme only one member of the replica group computes a result in response to an invocation, while the other members record a trace of requests and responses. The active member checkpoints itself to other members. If the active member fails, another member takes over.

Transparency mechanism: It is possible for a singleton client to invoke a replicated server, for a replicated client to invoke a replicated server, or for a replicated client to invoke a singleton server. The case of a singleton client invoking a singleton server may be treated as a special one where the client and server group membership is one.

Replication requires the use of functions such as *distribution*, *collation*, *ordering*, *coordination, recovery*, and *group membership*. The basic mechanism to achieve replication is the use of *atomic broadcast protocols*.

In the ODP engineering model, the replication transparency mechanisms are encapsulated in the *replication transparency binders* in the client and server side of the channel.

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

7.6 Resource Transparency

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

Resource transparency, also known as *liveness transparency*, masks the automated transfer of *clusters* from a *capsule* to a *storage* object and back again, to optimize the use of the node's nucleus resources (processor, memory, etc.). *Active objects* have nucleus resources (e.g., processor, memory, etc.) required for execution, while *passive objects* do not.

Transparency mechanism: Objects that are not actively in use may be transferred from the execution environment to storage (*passivated*) and then brought back to the execution environment (*reactivated*) when a (remote) operation invocation is made on them. Thus, resource management policies may lead to objects (hence their interfaces) being requested to move from one location to another and undergoing a change in representation.

Resource transparency requires the functions for passivating and activating objects. The resource transparency mechanisms are encapsulated in the *resource transparency binder* object on the server side of the channel.

Result of application of transparency: Clients can invoke operations on the server irrespective of whether the server is currently active or passive.

7.7 Failure Transparency

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

Transparency mechanism: Failure transparency requires the mechanisms for periodically *checkpointing* the object state, and for its *recovery*. Failure transparency is achieved by saving the checkpoints of the (server) object at another location with independent failure characteristics and subsequently reactivate the object from that checkpoint if the active object fails.

Failure transparency mechanisms are encapsulated in the *failure transparency binder* object on the server side of the channel.

Result of application of transparency: Clients do not get a failure response, for certain types of failures of server object, in the operation termination.

7.8 Federation Transparency

Hides: Federation transparency hides the effects of operations crossing multiple administrative boundaries from the clients.

Transparency mechanism: Servers of the clients may be located in administrative domains or technology domains other than the one in which the client is located. Administrative domains may impose their own access control policies for such purposes as security, accounting, monitoring, etc. Between technology domains protocol conversion, name translation, etc., may be required.

Federation transparency mechanisms are provided by *client federation transparency stub*, *server federation transparency stub, gateways* and *interceptors*.

Result of application of transparency: It permits interworking across multiple administration and technology domains.

Table 2: ODP Distribution Transparencies

Transparency	Central Issue	Result of Transparency
Access	The method of access to objects (invocation mechanism and data representation).	Clients need not be aware of *access* mechanisms at the server interface.
Concurrency	Concurrent access to objects in the distributed system.	Clients are masked from the effects of concurrent access to the server interface.
Location	Location of object in the distributed system.	Clients are unaware of the physical location of the server.
Migration	Dynamic re-location of objects during the “bind-session”.	Clients are unaware of the dynamic migration of the server.
Replication	Multiple invocations on replicated objects, multiple responses, and consistency of replicated data.	Client invokes a replicated server group as if it were a single server. Distribution of 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.

8. 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.

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

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

[5]. Proceedings of the IFIP TC6/WG6.4 International Workshop on Open Distributed Processing (October 1991), North Holland 1992.

[6]. Proceedings of the International Conference on Open Distributed Processing (September 1993), Berlin.

[7].Proceedings of the First Telecommunication Information Networking Architecture Workshop, (TINA 90), Lake Mohonk, New York, USA, June 1990.

[8]. Proceedings of the Second Telecommunication Information Networking Architecture Workshop, (TINA 91), Chantilly, France, March 1991.

[9]. Proceedings of the Third Telecommunication Information Networking Architecture Workshop, (TINA 92), Narita, Japan, January 1992.

[10]. Proceedings of the Fourth Telecommunication Information Networking Architecture Workshop, (TINA 93), L'Aquila, Italy, September 1993.

[11]. ANSA Reference Manual, Volume A., Release 01.01, Advanced Projects Management Limited, Cambridge, U.K., JUly 1989.

[12]. ANSA Atomic Activity Model and Infrastructure, AR.004.01, February 1993, Architecture Project Management Limited, Cambridge, U.K.

[13]. ANSA: A System Designer's Introduction to the Architecture, RC.253.00, April 1991, Architecture Project Management Limited, Cambridge, U.K.

[14]. ROSA Architecture (Release Two), 5th Deliverable, RACE Project R1093, May 1992.