INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.600

(11/2000)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Distributed processing environment architecture

ITU-T Recommendation Z.600

(Formerly CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of Formal Description Techniques	Z.110–Z.119
Message Sequence Chart	Z.120–Z.129
PROGRAMMING LANGUAGES	
CHILL: The ITU-T programming language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

For further details, please refer to the list of ITU-T Recommendations.

ITU-T Recommendation Z.600

Distributed processing environment architecture

Summary

This Recommendation describes the Distributed Processing Environment (DPE) Architecture, which represents the run-time environment for telecommunication and information services and applications.

The purpose of the DPE Architecture is to provide detailed technical requirements leading to specifications, both to help the DPE vendors to develop their products and the application developer to understand the infrastructure support that the DPE provides.

The material herein is based on work done in the TINA Consortium by the TINA core team and several auxiliary projects in the member companies of TINA-C, supporting the core team.

This Recommendation contains:

- an explanation of the relationship between modelling concepts so far as such a relationship needs to be concerned in the computing architecture;
- a description of the Kernel Transport Network (KTN) which is the DPE analogue of the telecommunications signalling system;
- an interoperability framework for the DPE;
- requirements for the DPE kernel services.

A number of DPE engineering object services can be identified, that support the execution of telecommunication services. These DPE object services are identified and associated to the functions and transparencies of the Reference Model for Open Distributed Processing (RM-ODP). Detailed requirements on the DPE object services and their specifications are for further study.

The DPE, the DPE object services and the applications deployed on a DPE need to be managed. What are the requirements on management, and how can management of these entities be accomplished is also for further study.

Not all DPE kernel services identified in this Recommendation are required for all applications. DPE profiles that support different kinds of services and applications need to be defined. These profiles need to specify which kernel services are mandatory for a given profile. The definition of DPE profiles and their use are for further study. The DPE must support quality of service (QoS) as needed by the services and applications. How this is accomplished by the DPE is for further study.

Source

ITU-T Recommendation Z.600 was prepared by ITU-T Study Group 10 (2001-2004) and approved under the WTSA Resolution 1 procedure on 24 November 2000.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

ITU-T Recommendation Z.600

Distributed processing environment architecture

1 Scope

This Recommendation specifies the Distributed Processing Environment (DPE) which supports the execution of distributed telecommunication applications. The DPE can be regarded as the infrastructure on which distributed telecommunications applications such as multimedia and real-time applications can execute.

This Recommendation provides the basic requirements and functionality for the DPE to support the execution of distributed telecommunications applications. Any distributed telecommunications application designed according to RM-ODP will benefit from using the DPE as a platform. The DPE itself is based on the RM-ODP concepts and principles ([2] and [3]).

RM-ODP specifies a **viewpoint** (on a system) as a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system. The RM-ODP viewpoints are:

- enterprise viewpoint;
- information viewpoint;
- computational viewpoint;
- engineering viewpoint;
- technology viewpoint.

This Recommendation focuses on the RM-ODP engineering viewpoint. It constitutes a specialization and refinement of the RM-ODP engineering language to take into account specific requirements from the telecommunication domain.

RM-ODP specifies **distribution transparency** as the property of hiding from a particular user the potential behaviour of some parts of a distributed system. The RM-ODP distribution transparencies are:

- access transparency;
- failure transparency;
- location transparency;
- migration transparency;
- relocation transparency;
- replication transparency;
- persistence transparency;
- transaction transparency.

RM-ODP specifies **functions** to support Open Distributed Processing. The RM-ODP functions are:

- management functions;
- coordination functions;
- repository functions;
- security functions.

The RM-ODP distribution transparencies and the ODP functions are associated with DPE kernel services and object services. This Recommendation uses the engineering functions as a basis and, where appropriate, refines them or specializes them for the DPE.

The DPE adopts OMG CORBA 2 as the prime technology base and thus the CORBA common object services are incorporated where appropriate in the DPE.

In particular, this Recommendation encourages the study of Real-Time CORBA [6], since this OMG specification satisfies already a number of requirements identified in this Recommendation.

1.1 Purpose

The purpose of the DPE Architecture is to provide detailed technical requirements leading to specifications, both to help the DPE vendors to develop their products, and the application developer to understand the infrastructure support that the DPE provides. The stated requirements are based on validated results from implementation projects.

In summary this Recommendation contains:

- detailed technical requirements;
- statements of required functionality.

1.2 Objective

The objective of this Recommendation is to provide an abstract description of the DPE. Implementations of the DPE have to support RM-ODP compliant applications as developed according to RM-ODP concepts and methodology. This abstract description of the DPE is implementation independent and should be used by DPE platform vendors as a reference for constructing DPE compliant platforms.

1.3 Non-objectives

This Recommendation describes services that are supposed to be present in many DPE implementations. It provides descriptions of these services without indicating how those services should be combined.

Furthermore this Recommendation does not restrict the actual implementation of the DPE in any way.

2 Normative references

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

2.1 Identical Recommendations | International Standards

[1] ITU-T X.901 (1997) | ISO/IEC 10746-1:1998, *Information technology – Open Distributed Processing – Reference Model: Overview.*

[2] ITU-T X.902 (1995) | ISO/IEC 10746-2:1996, *Information technology – Open Distributed Processing – Reference Model: Foundations.*

[3] ITU-T X.903 (1995) | ISO/IEC 10746-3:1996, *Information technology – Open Distributed Processing – Reference Model: Architecture.*

[4] ITU-T X.931 (1999) | ISO/IEC 14752:2000, *Information technology – Open distributed processing – Protocol support for computational interactions*.

2.2 Other ITU-T Recommendations

[5] ITU-T Z.130 (1999), *ITU object definition language*.

2.3 Other specifications

[6] OMG Document formal/01-09-34, *The Common Object Request Broker: Architecture and Specification*, Revision 2.5.

3 Definitions

For the purposes of this Recommendation, the following definitions apply.

3.1 Terms defined in the ODP reference model: Foundations

This Recommendation makes use of the following terms defined in ITU-T X.902:

a) architecture (of a system);

b) domain;

c) interface;

d) object;

e) template;

f) thread.

3.2 Terms defined in the ODP reference model: Architecture

This Recommendation makes use of the following terms defined in ITU-T X.903:

a) binder;

b) capsule;

c) channel;

d) checkpoint;

e) cluster;

f) kernel (nucleus);

g) node;

h) stub.

3.3 Definitions for distributed processing environment architecture

This Recommendation defines the following terms:

3.3.1 business administrative domain: A business administrative domain is defined by the requirements of one or more business roles and is governed by a single business objective.

3.3.2 business role: The expected function performed by a stakeholder in a telecommunications business environment.

3.3.3 communication API: The application programming interface through which the communication services are accessible

3.3.4 distributed processing environment (DPE): The DPE is the infrastructure that provides the execution environment, including distribution transparencies, for distributed applications in a

system. The DPE provides a distribution transparent view to its users. The users of the DPE are the application and service designers and developers. The DPE consists of a collection of DPE nodes that are interconnected.

3.3.5 DPE API: The application programming interface through which the DPE services are accessible.

3.3.6 DPE reference point: The engineering view of a reference point.

3.3.7 engineering computational object (eCO): An eCO is the engineering representation of a computational object (one-to-one correspondence), which encapsulates state/data and processing.

3.3.8 engineering service: Engineering services support the DPE distribution transparencies. There are a number of engineering services that are required by a wide range of applications (e.g. repository, object adaptor) and are fundamental for the construction of applications that execute on the DPE.

3.3.9 general inter-ORB protocol (GIOP): The GIOP is a messaging protocol used for object communication between different DPE nodes. The GIOP is specified in [6].

3.3.10 Internet inter-ORB protocol (IIOP): The IIOP is the mapping of the GIOP to TCP/IP. The IIOP is specified in [6].

3.3.11 inter-domain reference point: Same as Reference Point. Unless otherwise stated in this Recommendation, an inter-domain reference point is implied.

3.3.12 intra-domain reference point: A reference point not visible outside a business administrative domain. An intra-domain reference point is mainly used for procurement.

3.3.13 kernel transport network (KTN): The kernel transport network is the network which transports invocations and responses between different DPE nodes.

3.3.14 native computing and communication environment (NCCE): The NCCE is a computer which generally consists of a set of resources (hardware and operating system) that together provide a distinct platform. The NCCE is outside the scope of the DPE. Typically each DPE node runs on a NCCE and uses its resources and services. For example a DPE node would use a TCP/IP stack provided by the NCCE for communication with other DPE nodes.

3.3.15 object request broker (ORB): The ORB provides mechanisms for transparently communicating distributed client object requests to server object implementations, and for returning any server object responses to the requesting client object. The ORB is specified in [6].

3.3.16 object service: Object services support the execution of distributed applications.

3.3.17 reference point: A reference point marks a boundary between business administrative domains. A Reference Point is a set of several viewpoint related specifications within a context, defining constraints for one or more reference points to operate under (see Annex A).

4 Abbreviations

This Recommendation uses the following abbreviations:

AAL	ATM Adaptation Layer
ACID	Atomicity, Consistency, Isolation and Durability
API	Application Programming Interface
CMIP/CMIS	Common Management Information Protocol/Common Management Information Services
CORBA	Common Object Request Broker Architecture

DCE-CIOP	Distributed Computing Environment – Common Inter-ORB Protocol
DPE	Distributed Processing Environment
eCO	engineering Computational Object
ESIOP	Environment Specific Inter-ORB Protocol
GIF	General Interworking Framework (X.931)
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interface Object Reference
IPC	Interprocess Communications
KTN	Kernel Transport Network
LAN	Local Area Network
MPEG	Moving Picture Experts Group
NCCE	Native Computing and Communication Environment
ODL	Object Definition Language
OMG	Object Management Group
ORB	Object Request Broker
POSIX	Portable Operating System Interface
QoS	Quality of Service
RM-ODP	Reference Model for Open Distributed Processing
SCCP	Signalling Connection Control Part
SNMP	Simple Network Management Protocol
SS7	Signalling System 7
TCAP	Transaction Capabilities Application Part
TCP/IP	Transmission Control Protocol/Internet Protocol
TINA	Telecommunication Information Networking Architecture
UDP	User Datagram Protocol

5 Introduction

This Recommendation provides the architecture and the requirements for building large distributed processing environments, supporting the execution of telecommunication services and applications, including support for real-time services and applications.

Middleware can be defined as a set of software components that supports the interconnection of distributed systems and services. Middleware are software products that can be positioned between or underneath applications and on top of system software, computing systems and communication networks and services. In most cases, middleware is hiding the heterogeneity of the underlying systems, e.g. programming languages, operating systems, computing systems and network protocols. From the system developer's point of view, middleware provides a homogeneous view of a well structured, heterogeneous distributed environment, where applications and services inter-operate through standardized open interfaces. The DPE is an instance of middleware.

The DPE is the infrastructure that provides the execution environment, including distribution transparencies, for distributed applications in a system. The DPE provides a distribution transparent view to its users. The users of the DPE are the application and service designers and developers. The DPE consists of a collection of DPE nodes that are interconnected.

This Recommendation provides an abstract description of the DPE. Implementations of the DPE have to support RM-ODP compliant applications as developed according to RM-ODP concepts and methodology. This abstract description of the DPE is implementation independent and should be used by DPE platform vendors as a reference for constructing DPE compliant platforms.

The DPE includes a DPE kernel, which provides support for object life cycle control and inter-object communication. Object life cycle control includes capabilities to create and delete objects during run-time. Inter-object communication provides the mechanisms to support the invocation of operations provided by operational interfaces of remote objects. The DPE kernel provides basic, and technology independent functions that represent the capabilities of most operating systems, i.e. the ability to execute applications and the ability to support the communication of applications with each other.

5.1 DPE architecture

The DPE architecture as illustrated in Figure 1 is the unifying entity that encompasses all the required functionality described in this Recommendation, into a standardized implementation. It does this by presenting to applications a homogeneous view of a well structured, heterogeneous distributed environment, where applications and services interact through standardized open interfaces. This architecture hides the heterogeneity of the underlying systems, e.g. programming languages, operating systems, computing systems and network protocols.

The DPE architecture provides the middleware for interaction between engineering computational objects on remote DPE nodes. It also provides tools for diagnosis and configuration for use by network administrators.

The components of the DPE Architecture are categorized in:

- DPE Kernel;
- DPE Object Services;
- DPE Support Tools.

A DPE node is controlled by one DPE kernel (see also Annex B).

There can be a DPE reference point between any two DPE nodes.

5.2 DPE reference points

The concepts **Business Administrative Domain** and **Reference Point** are used as explained in Annex A.

A Reference Point, which is defined as a set of related specifications, is positioned between two business administrative domains. The DPE reference point is the engineering view of a reference point and supports the service related reference point as illustrated in Figure 2. The DPE API is used by the services and applications to access the functionality of the DPE.

A reference point marks a boundary between business administrative domains, thus a DPE reference point marks a domain boundary between DPE nodes operating under distinct business administrative domains.

Communication between distinct DPE nodes takes place over a DPE reference point.

Communication between services or between DPE nodes can take place in two different ways:

- **Client-server**: A common way to communicate between objects, in which software is split between server tasks and client tasks. A client sends requests to a server, according to some protocol, asking for information or action, and the server responds. There may be either one centralized server or several ones. This model allows clients and servers to be placed independently on nodes in a network, possibly on different hardware and operating systems appropriate to their function, e.g. fast server/cheap client.
- **Message passing**: A technique for communicating between parallel processes. A service running on one node may send a message to a service running on the same or another node. The execution environment handles the actual transmission of the message. A message passing environment provides primitives for sending and receiving messages. These primitives may by either synchronous or asynchronous or both. A synchronous send will not complete (will not allow the sender to proceed) until the receiving process has received the message. This allows the sender to know whether the message was received successfully or not. An asynchronous send simply queues the message for transmission without waiting for it to be received. A synchronous receive primitive will wait until there is a message to read, whereas an asynchronous receive will return immediately, either with a message or to say that no message has arrived. Messages may be sent to a named service or to a named mailbox, which may be readable, by one or many services. Messages may be typed or non-typed. They may have a priority, allowing the receiver to read the highest priority messages first.

5.3 Extensions to CORBA

OMG CORBA 2 [6] is a mapping of the DPE Architecture into a specific technology. Although the DPE architecture is based to a great extent on CORBA 2, it does not repeat CORBA 2 functionality and describes the additional requirements that arise in the telecommunications domain.

The DPE kernel requires additional functionality other than what is provided by OMG CORBA 2. Examples of additional required functionality are:

- support for an enhanced object model, e.g. support for multiple interfaces and stream interfaces;
- support for and integration of media streams;
- integration of quality of service aspects;
- support for real-time services and applications;
- support for different networks for inter-DPE communications.

At the level of object services, many OMG common object services as specified in OMG CORBA Services [7] can be reused to build a DPE.

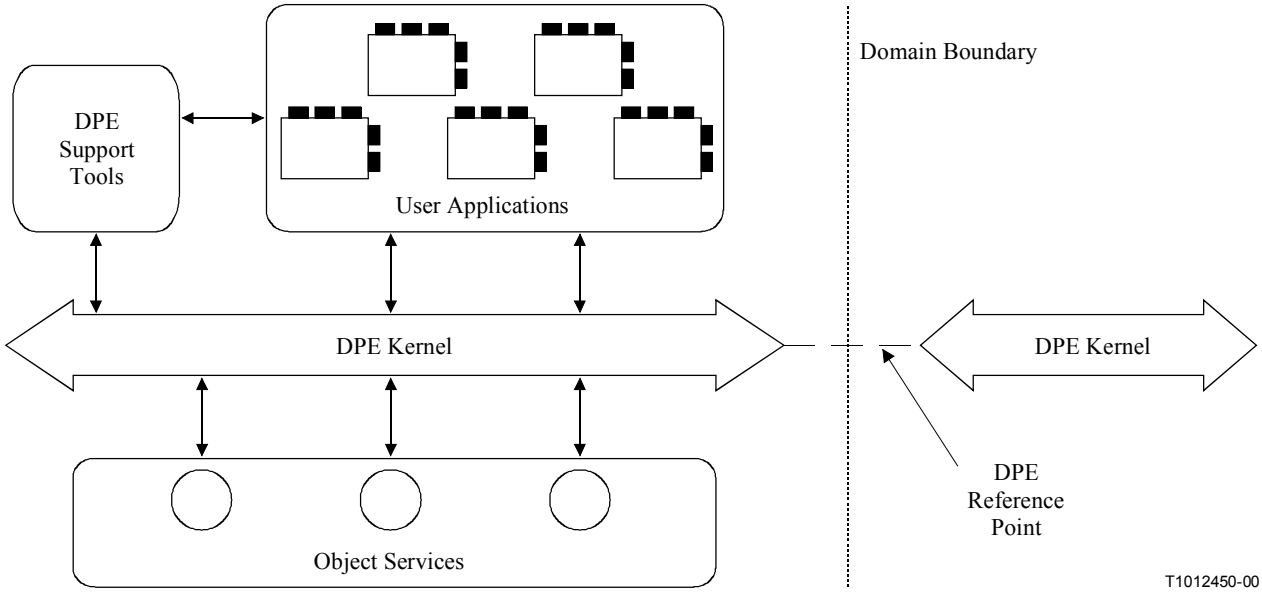

Figure 1/Z.600 – Architecture

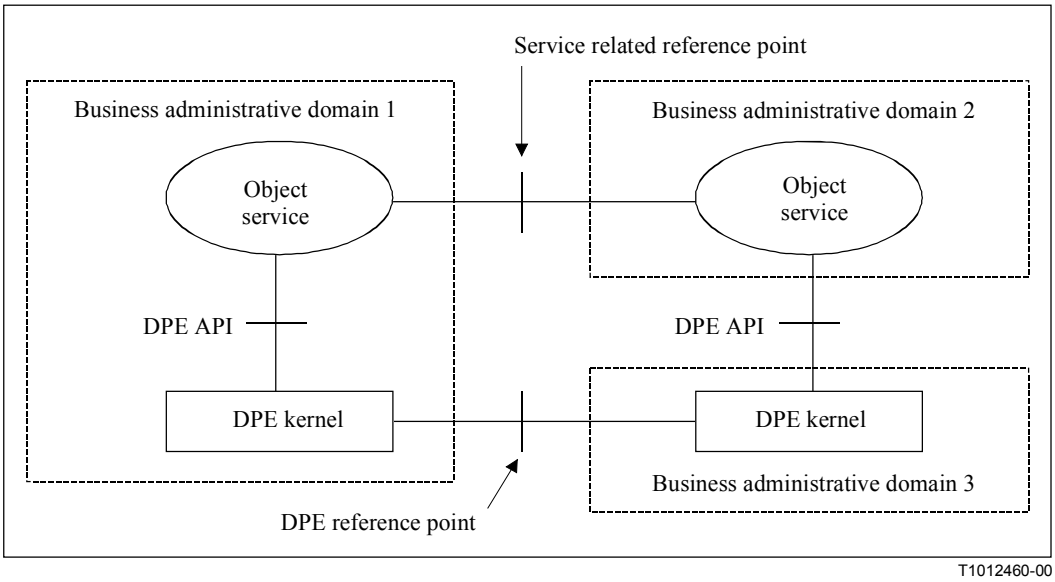

Figure 2/Z.600 – Reference points and APIs

6 Engineering modelling concepts

This Recommendation uses the basic engineering modelling concepts as defined in [3] and described in Annex B. In particular Annex B elaborates on the notions of:

- DPE node;
- DPE kernel;
- capsule;
- cluster;
- channel;

- object life cycle;
- communication in the DPE:
  - operational communication;
  - stream communication.
- bindings.

6.1 Object life cycle

Several stages have been identified for the life cycle of engineering objects. The stages relevant to the DPE are illustrated in Figure 3 are the following:

- **Deployment**: Executable code is deployed onto one or several DPE nodes in the form of eCO templates, enabling the DPE node to create an instance of an eCO for which it has execution code.
- **Creation**: An eCO instance is created within a cluster.
- **Activation**: Activation of an object sets it into an active state, i.e. computing resources are allocated to the object instance and the object instance is ready for receiving and answering invocations.
- **Deactivation**: Deactivation sets an object into a non-active state, i.e. computing resources are retracted and interfaces are not able to receive and answer invocations. Deactivation of an object also involves storing a checkpoint that can be used to reactivate the object later.
- **Deletion**: An eCO instance is removed from a cluster.
- **Withdrawal**: The eCO template is removed from the node.

A typical scenario for the object life cycle is found in Appendix I.

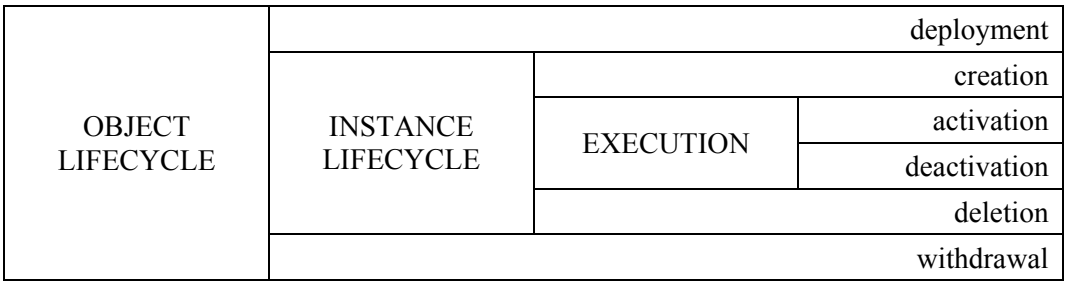

Figure 3/Z.600 – Stages of the object life cycle

6.2 Communication in the DPE

This clause describes the computational modelling concepts regarding the communication between objects in a DPE.

Computational interfaces are the only means for providing access to services that a computational object offers to its environment. A computational object interacts with other computational objects by invoking the computational operations (operations hereafter) they offer, or by exchanging stream flows with these other objects. Operational interfaces and stream interfaces are distinguished.

6.2.1 Operational communication

The interactions that occur at an operational interface are structured in terms of invocations of one or more operations and responses to these invocations. Operations are classified into two kinds:

- interrogations; and
- announcements.

Unlike an interaction via an interrogation, in an interaction via an announcement, no result is passed back from the server to the client, and the client is not informed of the outcome (success or failure) of the invocation. An operation message consists of a finite length data set.

6.2.1.1 Interface reference

Computational objects define computational interfaces as interaction points for other objects in terms of interface types and service attributes. The engineering interfaces of the eCOs reflect the interfaces of the respective computational objects.

Interfaces of eCOs are described by engineering interface references which specify the information needed to uniquely identify an engineering location of an interface and to bind to this interface.

Engineering interface references are capable of being passed across heterogeneous DPE nodes and are comparable for equality (for example for enabling the identification of interfaces after migration of an object instance).

6.2.2 Stream communication

A stream interface is an abstraction that represents a communication end-point that is:

- the source for some information flows;
- the sink for some other information flows; or
- both a source and a sink for information flows.

When objects interact via stream interfaces, the information exchange occurs in the form of stream flows between the objects, where each stream flow is unidirectional and is a bit sequence with a certain frame structure (data format and coding) and quality of service parameters. A stream message may consist of infinite-length information.

With reference to a stream flow, the object that is the source of the flow is called the producer and the object that is a sink is called the consumer. An object that offers a stream interface specifies the stream flows that can occur at that interface, and specifies, for each flow, whether the object is the producer or the consumer. For more details, see ITU-T Z.130.

NOTE – Streams containing objects are for further study. This depends on the definition of an object in MPEG.

6.3 Computational to engineering view mapping

See Annex B.

6.4 Bindings

According to RM-ODP, a binding provides a contractual context, resulting from a given establishing behaviour. In order for two objects to communicate via their interfaces, the execution environment (i.e. the DPE) must provide the mechanisms to bind them. Elaborations on bindings are found in 6.6 and Appendix II.

6.4.1 Operational vs stream binding

Bindings can be categorized according to the interface kind they bind, into:

- operational binding;
- stream binding.

6.4.2 Explicit vs implicit bindings

Bindings can be further categorized according to the kind of the establishing behaviour, being:

- explicit binding, resulting from explicit binding actions of the objects that will take part in the binding;
- implicit binding, being performed by an external party, i.e. when the user does not express binding actions.

See Figure 4.

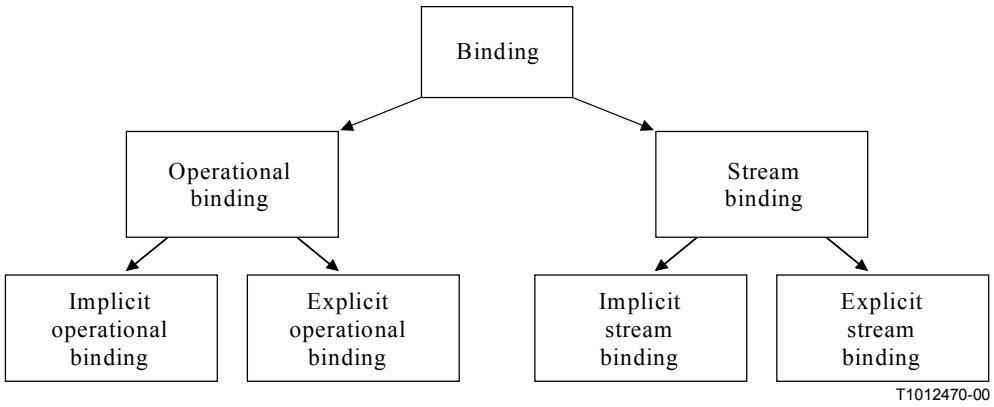

Figure 4/Z.600 – Binding relationships

6.5 Channels

In order to support explicit binding, the ODP notion of a channel is used. The engineering functionality of a channel derives from the requirement to support distribution transparent interactions of computational objects. Three fundamental channel services are defined:

- the stub service;
- the binder service;
- the protocol object service.

These services can be described as being split into closely related services for the client and the server side. Thus, a channel gets a symmetric appearance as illustrated in Figure 5.

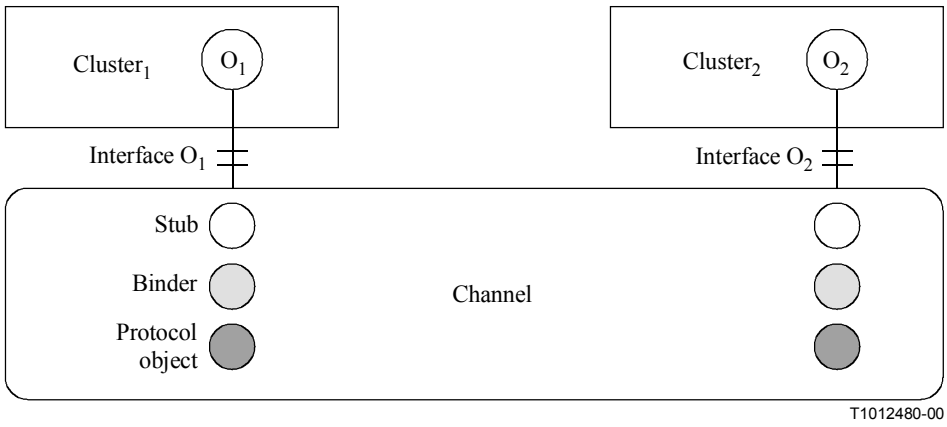

Figure 5/Z.600 – Stub, binder, and protocol object within a channel

The **stubs** support the channel interfaces to the engineering representation of the interacting computational objects. For operational communication the stubs add further interactions and/or information to interactions between the computational objects. They execute preludes and postludes before and after the actual operation invocation. They provide adaptation functionality in the sense that they make the transformations necessary to provide transparencies such as access transparency and transaction transparency. For stream communication the stubs provide compression and decompression functionality, as well.

The **binders** associated with the respective channel end-points at the respective interacting eCOs, interact with one another to maintain the integrity of the binding. They maintain information about the channel such as the association context or a data buffer. They are also responsible for validating the interface reference and for interacting with a relocation service or any other supporting service in order to keep consistent information about the interface location in case of a binding error (due to a failure, a migration, a deactivation, etc.).

Stub and binder services, as described above, provide primarily access and location transparency. In order to realize other transparencies, additional functionality may be required for stubs and binders, such as relocation or transaction management services.

The **protocol objects**, provide the mechanisms within a channel, which allow the related computational objects to interact remotely. To provide communication between different nodes, the protocol objects support a protocol that ensures that the interaction semantics are guaranteed. The protocol objects provide the access transparency functionality which is used by the stub object, which has knowledge of a specific computational interface, bound to it.

NOTE 1 – Intra-DPE node interactions may be feasible without the support of protocol services.

NOTE 2 – Remotely here means crossing DPE node boundaries.

6.6 Elaboration on the bindings

6.6.1 Introduction

In the DPE implicit binding for computational interfaces is supported through the GIOP protocol, which is mandatory for every DPE. In this case, only the Interface Object Reference (IOR) is known to the client object which invokes operations on this interface without having control of the binding. How the implicit binding is implemented is extensively elaborated in the CORBA 2 specification [6].

In order for two objects to interact by means of stream flows between them, each object has to offer a stream interface and the two interfaces must be bound.

The basic difference between interactions via operational interfaces and interactions via stream interfaces is that interactions via an operational interface are structured in terms of operation invocations and responses, whereas no such structure is imposed on interactions via stream interfaces. Once the stream interfaces of two objects have been bound, a set of *information flows* has been set up between the objects with some specific quality of service parameters. Thereafter, the producer of a flow inserts information into the flow, and the consumer retrieves information from the flow and consumes it. No explicit interaction occurs between the producer and the consumer during this information exchange. The control of these flows is achieved using operational interfaces. Eventually, either the producer, the consumer, or some third party object breaks the binding between the producer and the consumer. Thus, the paradigm for interaction via stream interfaces is that of asynchronous message passing, while the paradigm for interaction via operational interfaces can be that of a remote procedure call, or that of asynchronous message passing.

The stream channel adopts a stub-binder-protocol adapter structure similar to that for the operational channel.

Support for stream binding in the DPE can be accomplished in two different ways:

- Programmable stream interfaces.
- Control and Management of stream interfaces.

6.6.2 Programmable stream interfaces

The requirements to support stream interfaces in the DPE encompass:

- the ability to define and reference stream interfaces in a type-safe manner;
- the ability to bind together several stream interfaces through various stream binding objects and ideally with some QoS constraints;
- the ability to program stream interfaces; and
- the ability to synchronize multiple flows.

Satisfying these requirements yields a DPE that offers native support for creating, referencing and programming stream interfaces. A Multimedia Stream Object Adapter offers the following services (directly or through delegation):

- the explicit (type-safe) binding of multimedia stream interfaces with the ability to require QoS constraints;
- resource configuration, reservation and admission control;
- set-up of connections using multi-media transport protocols;
- the proper handling, generation and interpretation of stream interface references.

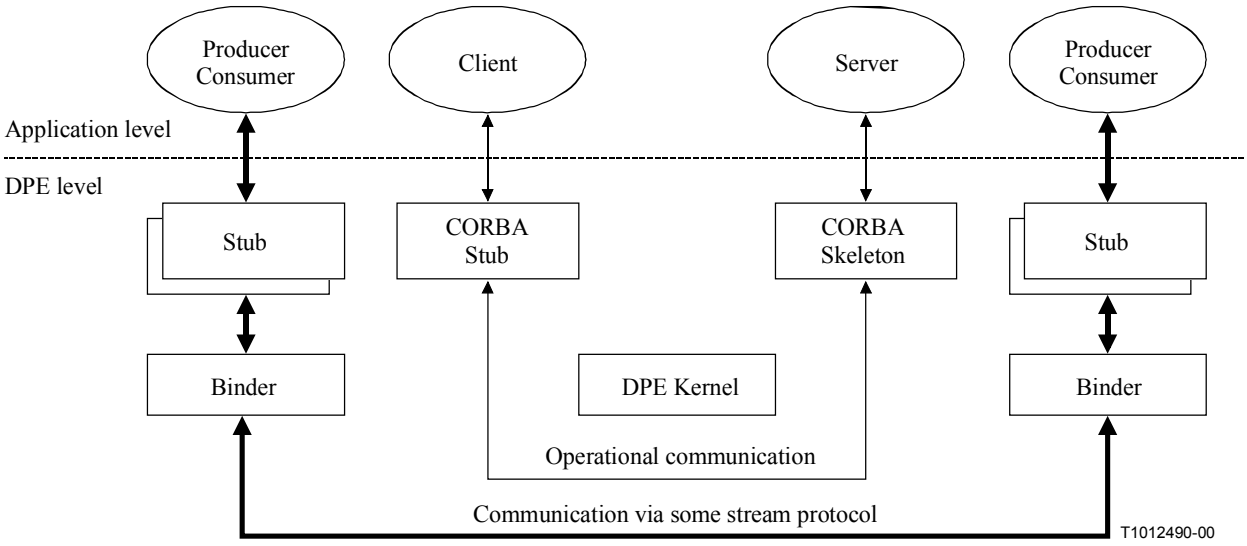

Figure 6/Z.600 – A multimedia DPE

Figure 6 illustrates the architecture needed for supporting programmable stream interfaces. It should be noted that with this approach the DPE kernel is also responsible for connection management (from the user's perspective) and that the stream flows are transported through the DPE.

Issues for the support of stream interfaces are elaborated in Appendix III.

NOTE – ITU-T Z.130, ITU object definition language, supports the specification of stream interfaces.

6.6.3 Control and management of media streams

This clause identifies a set of interfaces that implement a distributed media streaming framework. The associated specification has been adopted by the OMG as an object service for Control and Management of Audio/Video Streams [9].

Figure 7 shows a stream with a single information flow between two stream end-points, one acting as the source of the information flow and the other as the sink. Each stream end-point consists of three logical entities:

- a stream interface control object, that provides IDL defined interfaces for controlling and managing the stream;
- an information flow source or sink object that is the final destination of the information flow; and
- a stream adapter that transmits and receives frames over a network.

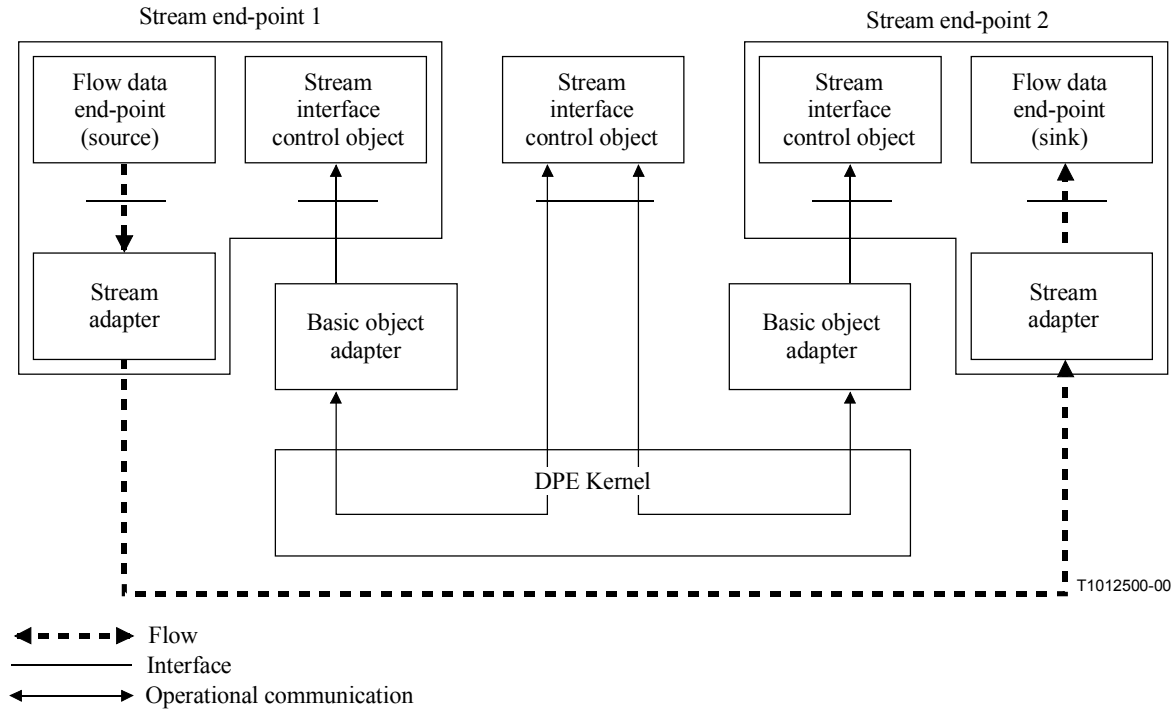

Figure 7/Z.600 – OMG stream architecture

The stream interface control object is using an object adapter, which transmits and receives control messages in a CORBA 2 compliant way.

When a stream is terminated in hardware, the source/sink object and the stream adapter may not be visible as distinct entities.

The Control and Management of A/V Streams OMG adopted specification provides definitions of the components that make up a stream and for interface definitions onto stream control and management objects, and for interface definitions onto stream interface control objects associated with individual stream end-points.

In particular, CORBA 2 interface references for stream interface control objects are used to refer to all stream end-points in parameters to stream control operations defined in the specification. Thus, in the case of control and management of media streams, the specification does not need to define a new IDL data type for stream interface references.

7 Kernel transport network

The Kernel Transport Network (KTN) is the network that transports invocations and responses between different DPE nodes. This clause gives certain guidelines for the construction of a KTN; however, it does not restrict the way the KTN maps onto the underlying network technology.

7.1 Messaging layer

The transportation of the invocations between different DPE nodes is done via some standardized messaging protocol. In order to guarantee interoperability between DPE nodes, a mandatory messaging protocol is specified. This protocol is the General Inter-ORB Protocol (GIOP) as specified in CORBA 2 [6]. In specific environments the use of GIOP is either not reasonable or not possible. In this case CORBA 2 describes the approach for the specification of an Environment Specific Inter-ORB Protocol (ESIOP). The same approach is adopted for the DPE Architecture.

The GIOP specification consists of the following elements:

- The Common Data Representation (CDR) definition: CDR is a transfer syntax mapping OMG IDL data types into an isomorphic low-level representation for "on-the-wire" transfer between ORBs and Inter-ORB bridges (agents).
- GIOP Message Formats: GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- GIOP Transport Assumptions: The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

7.2 Transport layer

A number of different transport protocols may be used to transport GIOP messages. In order to guarantee interoperability between DPE nodes, a GIOP mapping onto a mandatory transport protocol is specified. The mandatory transport protocol is TCP/IP, and the mapping of GIOP onto it is called Internet Inter-ORB Protocol (IIOP). The IIOP specification in CORBA 2 [6] is adopted for the DPE Architecture.

The IIOP specification adds the following element to the GIOP specification:

- Internet IOP Message Transport. The IIOP specification describes how agents open TCP/IP connections and use them to transfer GIOP messages.

The IIOP is not a separate specification; it is a specialization, or mapping, of the GIOP to the specific transport protocol TCP/IP. The GIOP specification (without the transport-specific IIOP element) may be considered as a separate conformance point for mappings to other transport protocols.

7.3 DPE requirements

There is a clear requirement for the DPE to support protocols other than IIOP for the KTN. Within the telecommunications domain, two of the most popular protocol stacks which require support are the SS7 protocol stack (i.e. the TCAP or SCCP layers) and the ATM Adaptation Layer 5 protocol. To support these protocols and guarantee interoperability, mappings from GIOP to these protocols need to be specified.

NOTE – Mappings from GIOP to alternative protocols are for further study.

The issue of interoperability over multiple protocols raises the broader problem of multiple protocol support in the DPE. The DPE must support the ability to communicate over multiple protocols such as:

- GIOP based;
- ESIOPs; or
- based on other means of communication such as message queues.

The mechanisms for multiple protocol support must be transparent to the application developer. Nevertheless, it must be possible for the application developer to set the parameters associated with a KTN connection. For example, the application developer must be able to specify bandwidth requirements for a KTN connection if the underlying protocol supports this parameter.

8 DPE interoperability

This clause provides the interoperability framework for the DPE. The interoperability framework explains the interoperability requirements at several different levels and provides the context for the Kernel Transport Network (KTN).

The details of the protocol support for computational interactions are defined in ITU-T X.931 [5]. In particular, ITU-T X.931:

- defines a General Interworking Framework (GIF);
- within the GIF, defines a family of functionally-related service primitives;
- specifies the mapping of the GIF service primitives and their parameters to the messages and fields of particular protocols, including CORBA GIOP.

DPE interoperability specifies an approach to support the seamless inter-operation of objects running on distinct DPE nodes. The approach is flexible in the sense that it allows several different combinations to support the specific needs of different environments. The basic concept of a homogeneous executing environment in which objects (services and applications) execute is maintained.

This clause addresses the interoperability at the DPE kernel level and the interactions between the DPE kernel level and the communication level.

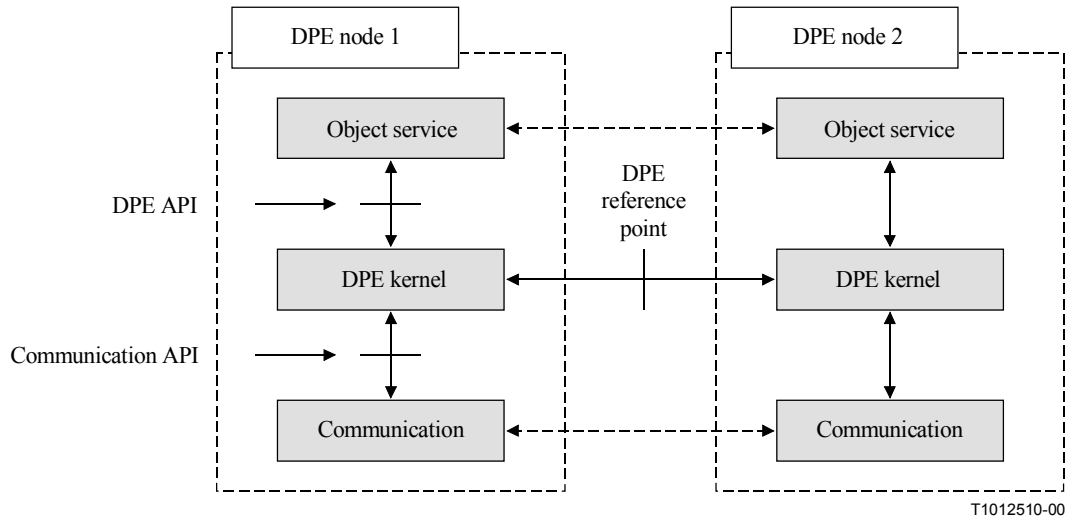

Figure 8/Z.600 – Interoperability framework

8.1 Interoperability framework

When observing two distinct DPE nodes that need to inter-operate, three different levels of inter-operation can be identified:

Object service: When service interfaces are specified in IDL or ODL, no information is available about the way peer objects perceive and process information. While the semantics of basic services (i.e. object services) is generally well understood, the internal semantics of specialized applications is only known to the application designer. At this level the interoperability concern lies in the internal semantics of the services or applications. This level of interoperability is outside the scope of the DPE.

DPE kernel: At this level service requests and service replies are transported between peer objects by mechanisms provided by the DPE. The details of these mechanisms involve:

- binding of the object interfaces (implicitly or explicitly); binding is supported by functions provided by the DPE API and the DPE reference point;
- data transfer involving the conversion of operation parameters and operation results into a common format for transmission.

Communication: This level is responsible for the interoperability of the transport protocols, which are used to transmit the service requests and service replies at the DPE level. The communication level is also out of scope of the DPE.

Figure 8 illustrates the interoperability framework.

8.2 DPE extensions to CORBA 2 interoperability

For the DPE, a clear separation between the network dependent parts of the DPE and network independent parts is required. This separation allows developing an adaptation to a networking protocol independent from the core of the DPE. Through this adaptation the development of networking protocols for inter-DPE communication becomes de-coupled from the rest of the DPE. The mechanism of pluggable protocol objects enables integration of the DPE kernel and network adaptation modules of different vendors. It should be possible to have multiple network adaptations in use at the same time.

The following requirements extend CORBA 2 interoperability. For inter-DPE communication, the DPE implementations should support:

- selection of protocol objects during initial configuration of a DPE node;
- dynamic selection of protocol objects at run-time of a DPE node;
- dynamic insertion of protocol objects (plug-in) at run-time of a DPE node;
- dynamic selection or insertion (plug-in) of protocol objects, according to some QoS requirements;
- concurrent use of multiple protocol objects should be possible.

An Open Communication Interface (OCI) is introduced. The OCI complements the definition of the inter-ORB communication as defined in CORBA 2 and provides a solid basis for the implementation of the Kernel Transport Network. The inter-DPE protocol relationships are illustrated in Figure 9. More specifically it allows the use of IIOP as a pluggable transport protocol.

For telecommunications domain message protocols a different approach could be used. In this case the message protocol actually replaces the CORBA GIOP message layer. Using the CORBA ESIOP approach, one can also define a mapping for environments using TCAP (see Figure 9).

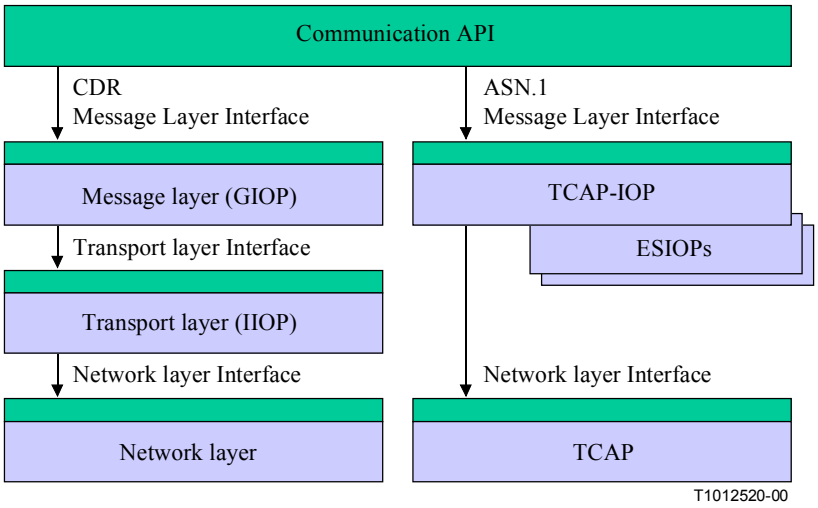

Figure 9/Z.600 – Example of inter-DPE protocol mappings

9 Engineering services provided by the DPE kernel

This clause identifies DPE kernel services necessary for the support of telecommunications applications. Not all DPE kernel services identified in this Recommendation are required for all applications. DPE profiles that support different kinds of services and applications need to be defined. Each profile specifies a set of mandatory kernel services. The definition of DPE profiles and their use is for further study.

9.1 Introduction

The DPE requirements are categorized into the following categories:

- modularity requirements;
- functional requirements;
- non-functional requirements.

The basic requirements that must be supported in all DPE implementations are associated with operational invocation handling at the peer object sides. These requirements are satisfied by CORBA 2 and include the requirements for the Dynamic Invocation Interface, the Static Invocation Interface and the Portable Object Adapter [6].

A DPE for telecommunication systems should support a wide variety of systems:

- low end network equipment such as routers, cross-connects, etc.;
- high end information processing nodes;
- systems with real-time behaviour;
- high performance systems;
- high availability and fault tolerant systems.

Requirements are presented in this clause, and no assumption is made about how the DPE will be implemented on top of an underlying operation system and hardware resources. Possible implementations range from standard application-level libraries and servers to a DPE fully integrated within the supporting operating system. In this regard, the DPE kernel requirements stated here go beyond the scope of a DPE and imply requirements on the supporting operating system.

In addition to the basic requirements, a DPE may incorporate any service such as transaction or security handling.

NOTE – Some object services may have a direct impact at the DPE kernel level. These services cannot be developed as standard services on top of the DPE kernel, or only at an unacceptable price in terms of performance. Support for such services falls within the scope of these requirements.

9.2 Approach to real-time

In this Recommendation, a real-time system is defined as a system that allows:

- specification of timing constraints, e.g. deadline and priority;
- specification of throughput requirements;
- guarantees about the fulfilment of these requirements;
- allocation and control the resources needed for predictability.

The nature of guarantees provided may vary from:

- best-effort, where the system provides no quantitative guarantee of how well or how often it will meet application QoS requirements;
- deterministic, where the system guarantees that application requirements will be strictly met throughout the lifetime of the application.

Any reasonable guarantee provided by a system can only be met relative to given QoS dependability levels.

An important distinction can be made between closed and open real-time systems. A closed real-time system is a system that supports a fixed and predetermined set of applications. Examples are classical embedded real-time systems. An open real-time system is a system whose set of supported applications is not known beforehand and which may vary over time. This distinction is important relatively to the nature of tools used to ensure that QoS guarantees can be provided to applications. In a closed real-time system, application designers can plan relevant schedules and resource usage off-line. This is typically not possible in an open real-time system, where strong (i.e. stronger than best-effort) guarantees can be provided only via some form of run-time admission control.

Providing timing and throughput guarantees in an open distributed real-time system is a difficult task. For this reason, a real-time DPE should, as a general principle of separation between policy and mechanism, refrain from embodying any particular resource management policy. Instead, a real-time DPE ought to be flexible enough to accommodate different policies and should provide interfaces for direct control of system resources such as processors, memory and communication.

9.3 Requirements on the execution model

When a DPE kernel receives a request, the policy implemented by the kernel for switching between threads in a multi-treaded environment must not corrupt the real-time behaviour of the other parts of the application running on an operating system.

Multi-threading is defined here as sharing a single processor between multiple tasks (or "threads") in a way designed to minimize the time required to switch between threads. This is accomplished by sharing as much as possible of the program execution environment between the different threads so that very little state information needs to be saved and restored when switching between threads.

When deploying a DPE on a physical platform, the execution model must allow customization of the internal behaviour according to the real-time requirements:

- control over the server threads (number of threads, priority, scheduling class, stack size);
- control over the invocation model (policy when a request arrives).

9.4 Modularity requirements

9.4.1 Flexible DPE architecture

The architecture of the DPE allows:

- the introduction of new binding and communications mechanisms;
- the ability to incrementally add transparency services (security, transaction, persistence, migration etc.).

In addition, the DPE may allow on-demand loading of kernel services. The corresponding APIs of the DPE must be made public to the application or system developers allowing them to tailor the platform to the particular requirements of their applications.

The DPE architecture provides modularity because:

- it is simpler to meet stringent performance and size constraints by omitting components from a DPE architecture than attempting to strip down a monolithic DPE;
- the cost of developing a specialized DPE for (often low volume) niche or emerging markets is reduced by reusing the architecture and libraries of replaceable components;
- interworking between DPE nodes is supported, because they are specialized from the same DPE architecture.

The DPE architecture has to provide:

- general abstraction mechanisms, allowing many different resource management policies and processing schemes, including those which enable application level control; and
- mechanisms to allow designers to compose these resources. The design of those abstraction mechanisms should be modular enough to give designers a free choice of components used in constructing a specific DPE. Modularity is associated here with the classical properties of simplicity, orthogonality and sufficient narrowness of interfaces.

9.4.2 Multi-protocol support

The DPE should allow for:

- the introduction and simultaneous execution of multiple communication protocol stacks, suited for different types of applications and QoS requirements;
- the ability to dynamically indicate which protocol stack to be used and to control the level of communication resource multiplexing (e.g. to request the creation of a binding using a previously allocated communication channel);
- on-demand loading of protocol code.

The availability of the DPE API as implied in 9.4.1 is a prerequisite to fulfil this requirement. The stubs generated by the IDL or ODL compiler must be protocol independent.

The following example protocols may by supported:

- telecom protocols: TCAP, CMIP/CMIS, SNMP, etc.;
- ORB and RPC protocols: IIOP, DCE-CIOP, etc.;
- real-time protocols for video/audio transportation;
- real-time signalling protocols.

The support for multiple protocols is a prime requirement in a telecommunications environment. Legacy protocols are taken into account as well as new protocols addressing new functional needs (e.g. multicasting or group communications).

9.4.3 Generic communication scheme

A DPE incorporates a generic communication scheme to allow the exploitation of different communication resource multiplexing policies and the construction of any protocol. Such a scheme features:

- Protocol function composition: It is possible to build protocol stacks out of modules and to reuse modules across several stacks. Dependencies between modules must be minimized.
- The nature of protocol stacks is completely de-coupled from data presentation issues. For example, it should be possible to use a specific marshalling policy with any of the communication protocols.
- Protocol configuration: Protocols are often platform specific, and should only be configured when necessary support exists. Protocol configuration can be done either at compile time, or dynamically (at run-time).
- Independence of concurrency and multiplexing, which are the major sources of protocol overhead and inflexibility of protocol architectures.
- Generic threading scheme to allow the adaptation of different concurrency handling techniques.
- A generic buffer management scheme to allow different data presentation, fragmentation, and protocol processing techniques.

A generic communication scheme helps protocol designers and implementers to extend the DPE with new protocols and facilitates the reuse of standard implementations of protocol components. Such a scheme also enables the fine tuning of distributed application implementations, by allowing the exploitation of different design trade-offs.

9.4.4 Support for flexible binding

A DPE provides support for any binding between objects. In particular it:

- allows application developer defined bindings;
- provides both explicit (i.e. initiated in application code) and implicit bindings;
- provides support for QoS-constrained bindings;
- supports any binding scenario, including multi-point binding for group communication and binding by a third-party;
- allows application-level control over established bindings.

An elaboration on flexible binding can be found in Appendix II.

9.5 Functional requirements

9.5.1 Support for stream interfaces

The DPE should provide support for stream interfaces, including:

- definition of strongly-typed stream interfaces;
- ability to bind together multiple stream interfaces;
- application-level processing of streams;
- ability to synchronize multiple flows.

The notion of stream interface is necessary to describe object interfaces that are capable of generating or consuming continuous flows. With the introduction of stream interfaces, it becomes necessary to provide support for:

- defining stream interface signatures;
- programming of incoming or outgoing information flows associated with stream interfaces;
- control of communication paths for information flows between different stream interfaces;
- control of generation or reception of information flows at stream interfaces.

The control of communication paths between stream interfaces, i.e. their establishment, configuration, monitoring and deletion, is accounted for by the requirements in 9.4.4.

9.5.2 Multithreading support

The DPE kernel should provide support for multithreading. The DPE must be thread-safe and re-entrant both on the client and the server side. Locks used to protect the internal data structures of the DPE must be documented and their granularity must be small enough to achieve a certain level of concurrency. The DPE should provide a thread abstraction layer which isolates the developer from system-specific thread packages. This abstraction layer covers:

- thread handling (e.g. creation, destruction, etc.);
- thread scheduling: specification of scheduling parameters including temporal parameters (e.g. deadlines);
- synchronization facilities: semaphores, mutual exclusion, condition variables, etc.
- communication/invocation policy: how does the DPE wait for request and dispatch them to the servant;

- servant policy: how a request is processed.

Multithreading is a classical requirement in distributed systems and is necessary to achieve the levels of concurrency required by distributed applications in telecommunications.

9.5.2.1 Ensemble of threads

Once the thread abstraction layer is properly defined, it allows the definition of an ensemble of threads sharing common characteristics. The characteristics include:

- the definition of thread parameters:
  - the common POSIX parameters (scheduling class, stack size, priority);
  - the MAX and MIN number of threads;
  - the policy to be applied when all the threads or the ensemble are busy (exception or request queuing).
- the definition of the request queue of the ensemble of threads:
  - the maximal number of request that can be queued;
  - the time during which a request may be queued and not processed. When the time has elapsed, the request is extracted from the queue and rejected with an exception.

9.5.3 A flexible event-to-thread mapping

Application programmers should be able to build and define mappings between events and threads. Events can be requests, signals, etc. Events related to timers or to request handling (e.g. receipt of an invocation) are particularly important. The DPE should provide the means to specify the mapping of requests to threads on an object basis:

- one thread per object;
- a dedicated pool of threads, etc.

and should allow the request dispatching to be taken over by the developer. The typical thread mapping policy of incoming requests can be itemized as follows:

- the incoming request will be processed by the current (usually protocol) thread;
- the incoming request will be processed by a new created thread;
- the incoming request will be processed by a thread selected from a thread Pool.

Many different event-processing schemes and object implementations are possible. There is no single way to implement communicating, possibly multithreaded objects for best performance and versatility. Different trade-offs have to be considered based on operating system functionality and application requirements. For this reason, a real-time DPE should allow different implementations and processing schemes. This function ultimately depends on the ability to associate in different ways event occurrences and threads responsible for handling these occurrences.

9.5.4 Concurrency management

In a client-server architecture, a server will typically have several clients and is therefore likely to receive overlapping requests. If a server is only able to handle one request at a time, then the requests are queued, and if the processing time is substantial or there are many requests, the delay imposed on the clients can become excessive. Thus, a server will normally be expected to handle (or appear to handle) multiple requests in parallel. In addition, a server must not only handle several requests at the same time, but it must do so safely, ensuring that there is no conflict in data access between parallel requests. Thus concurrency mechanisms within a capsule are required. This is termed lightweight concurrency and is usually applied in situations where all components involved in the parallel activity reside in the same address space. Concurrency between capsules is not addressed here.

9.5.4.1 Overview of concurrency model

Lightweight concurrency can be supported through the use of **thread** and **task mechanisms**. Lightweight concurrency can be viewed in the following way:

- A logical unit of concurrency, known as a **thread**, is an independent execution path through an application, which can be executed concurrently with other logical units of concurrency.
- A physical unit of concurrency, known as a **task**, is the set of resources required to enable execution of a logical unit of concurrency.

A thread can only execute when it is associated with a task. Tasks are shared and reused by threads in an application. Once a thread has completed execution, the task it is executing on is released and can be used by another thread. By separating logical and physical concurrency through the definition of threads and tasks, we can control the resources used to provide the concurrency in an application without affecting the logical concurrency of the application (see also 9.5.2).

There may be several thread/task pairs bound and waiting to execute in the DPE kernel at any one time. The DPE kernel must organize these activities in some ordered manner; this process is commonly known as scheduling.

9.5.5 Generic scheduling scheme

A real-time DPE must offer a scheduling framework with a clear separation between application-selectable scheduling policy modules and a shared set of scheduling mechanisms. Support is needed for both priority and deadline-based scheduling policies.

Real-time requirements from applications in an open environment can only be met with a certain level of guarantee if the scheduling of tasks implementing these applications can make use of temporal information such as worst-case execution time, deadline, period, etc. Such information is absolutely required for admission control, and can be exploited by scheduling algorithms (e.g. earliest deadline first or rate-monotonic). Since there is no single best scheduling algorithm or admission control policy or even real-time task model, appropriate scheduling policies for the applications must be selected.

9.5.6 Time service

A time service in a real-time DPE must provide application developers with the means to obtain the current time with a known precision and to generate time-based events through timers. Typically, this service is used for QoS-constrained applications and to enable real-time monitoring and control.

9.6 Non-functional requirements

9.6.1 Object granularity

The DPE should be scalable to support applications handling a large number of objects and to support many simultaneous connections to remote objects.

The memory cost of an unused remote interface reference in a given capsule should be of the order of a standard language pointer.

The handling of many simultaneous connections to remote objects hinges on the ability of the generic communication scheme to allow sharing and reuse of communication resources to minimize remote communication overhead.

A typical telecommunications environment comprises objects of very different granularity in both space (memory size) and time (object lifetime and duration). A scalable DPE must support objects at different levels of granularity, and minimize the overhead associated with the handling of small and volatile objects and of connections to remote objects.

9.6.2 Small memory footprint

A real-time DPE must be lightweight enough so that its memory footprint allows it to be deployed on top of minimal hardware platforms: e.g. embedded systems, set-top boxes, personal digital assistants. The memory overhead incurred by the use of the real-time DPE must be properly documented and kept to a minimum.

NOTE – An exact characterization of the memory overhead mentioned above is not necessarily easy, considering the various factors that must be taken into account. A first attempt for such a characterization can be done by considering the memory cost associated with the use of the real-time DPE for supporting communicating objects located in two different capsules, compared to the native communications facilities provided by the supporting operating system.

9.6.3 Documented time behaviour

In order to provide applications with QoS guarantees, the DPE must provide information on the temporal behaviour of the supporting infrastructure. The applications must know the characteristic latencies of the supporting mechanisms under relevant load and environment constraints. This requirement is akin to knowing the temporal characteristics of basic operating system primitives (in a given implementation).

For a given hardware and basic software (i.e. operating system) environment, a real time DPE implementation should be documented with a complete list of performance measurements and locking constraints for all DPE kernel functions (e.g. delays for worst-case execution paths, critical section duration, etc.). The characteristic latencies of a real-time DPE must be identified and documented.

9.7 Multiple interfaces

The DPE should support computational objects with multiple interfaces. Some of these interfaces may be created when the object is created, while some may be created dynamically during the execution of the object. The multiple interfaces provided by an object may be of different types and there may be multiple instances of the same interface type, or a mix of both.

An object may use various interfaces of different interface types:

- to provide services having different nature of interactions;
- to provide logically distinct services; or
- to reflect different access rights by different users.

An object may use different interface instances of the same type:

- to offer the same service simultaneously to several clients by providing one interface to each client.

NOTE – A computational object does not always have to offer a separate interface instance for each client accessing it. In many cases it will make sense for several clients to access the same interface instance.

ITU-T Z.130, ITU Object Definition Language (ITU-ODL) [4], supports the specification of multiple interfaces.

10 Object services

This clause describes object services needed to support the execution of telecommunication applications. Some of these object services are generic enough in nature to be considered as general purpose computing services, while other object services are specific to the telecommunications domain.

10.1 Introduction

The DPE is an infrastructure, which provides support for telecommunications applications. This requires the provision of services by the DPE that provide functions to support the object interaction mechanisms. These services are sufficiently generic to be described as part of the DPE. These services have a computational interface, i.e. their functionality can be invoked as if they were computational objects.

The following object services are currently considered as DPE object services.

Life Cycle Service: The life cycle service provides functions for creating, deleting, copying, and moving individual objects or collections of objects. It also provides capabilities for deactivation, reactivation, replication, recovery and migration.

Naming Service: The naming service is an fundamental DPE Service. Its objective is the localization of interfaces, especially of object implementations. The naming service provides a mapping between a human readable name and an interface reference.

Trading Service: The trading service supports late binding between two objects, the exporter and the importer. To do so, it administers information about service offers, the associated interface references and service attributes. The exporter offers its services (interfaces); the importer seeks services and uses the trading service in order to get hold of them.

Security Service: The security service manages confidentiality, integrity, accountability, and availability within the DPE. The security service counteracts threats of disclosure, deception, disruption, and usurpation of telecommunication data and services.

Notification Service: The notification service enables objects to emit or receive notifications without being aware of the set of objects with which they are communicating. Similarly, it enables objects to receive notifications without having to interact with emitter objects. The service acts as a broker between emitters and recipients.

Transaction Service: The transaction service provides transactional communication between objects guaranteeing consistency of applications with properties collectively referred to as ACID properties: atomicity, consistency, isolation and durability. Open nested transactions are provided for real-time applications.

Concurrency Control Service: Concurrency is a paramount concern in a real-time system, where a trade-off exists between high availability on the one hand and application consistency on the other. A Concurrency Control Service (CCS) enables multiple threads to coordinate their access to shared objects. When many concurrent threads access a shared object, any conflicting operations by the threads are reconciled so as to preserve the consistency of the object state.

Persistence Service: The persistence service allows for the management of the persistent state of objects. It ensures the integrity of the data of a given database by ensuring the consistency of the objects.

Interrogation Service: The interrogation service provides interrogation operations on collections of objects. It can be used to return collections of objects that are either selected from a source query-able collection or produced by a query evaluator. Interrogations are specified using a query language and may perform general manipulation operations such as selection, updating, insertion and deletion on collections of objects.

Messaging Service: In order to achieve their aims of high performance, scalability and throughput telecommunications applications may use asynchronous communication in combination with an event based programming model. The messaging service satisfies the requirements for a truly asynchronous method invocation model.

Migration Service: A migration service as part of the DPE object life cycle service requires realization of two distribution transparencies:

- relocation transparency masks a migration of an object from other objects bound to it;
- migration transparency masks location changes from the object being relocated.

Licensing Service: The licensing service enables accounting of access and use of telecommunication services and software applications.

Event Logging: For the purpose of managing any system, it is necessary to have the ability to trace the activity history of the system. A service for making event notifications persistent and storing them (logging) is a basic requirement.

Topology: The Topology service provides a general service for managing the topological relationships (associations) between distributed objects. Its purpose is to relieve application objects from the burden of managing associations by providing a service for storing topological information independent of a specific application.

Software Distribution/Installation: The Software Distribution/Installation service provides run-time support when software needs to be distributed and/or installed on a large number of nodes. For example when deploying new telecommunications services, support by the DPE is needed in order to properly coordinate the deployment.

Software Configuration Management: The Software Configuration Management service provides the functionality to configure various services running over the DPE, in the same way as management functionality is used in telecommunication networks to set up and modify parameters of the physical equipment. When such functionality is added to the DPE, the users can manage both software and hardware entities in a seamless fashion. This service might be integrated with the Integrated Management Service.

Integrated Management: The Integrated Management Service targets the integration of user application management, platform management and network management.

Control and Management of Audio/Video Streams: The Control and Management of Audio/Video Streams Object Service adopted the OMG [9] specifies in detail this service.

NOTE – The object services can be associated to the ODP distribution transparencies and functions (see ITU-T X.903 [3]). The actual association is for further study.

11 Bibliography

[7] OMG Document number formal/98-12-09, CORBA Services: Common Object Services Specification, Revised Edition, 31 March 1995, Last Updated December 1998.

[8] HOLZ (E.), KATH (O.), GEIPL (M.), LIN (G.), VOGEL (V.): The CAMOUFLAGE Project – Introduction of TINA into Telecommunications Legacy Systems, in *Proceedings of TINA'97*.

[9] OMG Document Number telecom/97-05-07, Control and Management of Audio/Video Streams, 30 May 1997.

[10] GIFFORD (D.), GLASSER (N.): Remote pipes and procedures for efficient distributed communication, *ACM Transactions on Computer Systems*, Vol. 6, No. 3, 1998.

[11] COULSON (G.), BLAIR (G.S.), HORN (F.), HAZARD (L.), STEFANI (J.B.): Supporting the Real Time Requirements of Continuous Media in Open Distributed Processing, *Computer Networks and ISDN Systems, Special Issue on Open Distributed Processing*, Vol. 27, No. 8, July 1995.

ANNEX A

Business modelling concepts – A framework for the propagation of requirements in an open telecommunication market

A.1 Scope

The business modelling concepts provide the means to model telecommunication and information services in a multi stakeholder business environment.

The business modelling concepts specify a common business framework for all stakeholders in an open telecommunication market.

The instantiation of the abstract business model for a particular service enables:

- the identification of the business roles needed to provide a particular service;
- the association of the business roles with the involved stakeholders;
- the identification of the business relationships between the business roles and the business administrative domains owned by the stakeholders involved;
- the specification of the reference points implementing the business relationships.

A.2 Terms and definitions

This annex defines the following terms:

A.2.1 business administrative domain: A business administrative domain is defined by the requirements of one or more business roles and is governed by a single business objective.

A.2.2 business relationship: An association between two business roles.

A.2.3 business role: The expected function performed by a stakeholder in a telecommunications business environment.

A.2.4 contract: A contract is the context defining constraints for one or more reference points to operate under.

A.2.5 reference point: The manifestation of a business relationship in the telecommunication system. The reference point consists of several viewpoint related specifications governed by a contract.

A.2.6 stakeholder: A party that holds a business interest or concern in the telecommunications business. A stakeholder owns one or more business administrative domains.

A.3 Business modelling concepts

A.3.1 Framework

The basis of an open telecommunication system are the information, computational and engineering objects owned by business administrative domains and separated by reference points. In order to specify the policies and interactions between business administrative domains, one needs to specify the visibility and rights on each type of object in the domain with regard to related domains. These rights and visibility are included in a contract. The contract is established between business administrative domains and can be negotiated.

A.3.1.1 Contract

A contract provides the basis for the contexts defined in the supported viewpoints. Within the constraints specified in the contract, the contexts in the supported viewpoints can be modified by negotiation. However, the contract can never be modified as a result of the negotiations within the supported viewpoints, since a single viewpoint only provides a partial view of the interactions between the business administrative domains and might violate the policies negotiated for the other viewpoints.

A.3.1.2 Business administrative domain

A business administrative domain is defined by the requirements of one or more business roles. Business administrative domains interact with each other through reference points, which are the implementations of the business relationships between business administrative domains.

The concept of business administrative domain is based on ownership. Ownership implies the universal privilege of managing the entities inside the domain.

A.3.1.3 Business roles and business relationships

The business roles are identified by analysing the current and expected future business needs in telecommunication and information services. The definition of the business roles are driven by the following types of business separations:

- **Technical**: Areas of different development speed of technology are placed in different business roles.
- **Economic**: Business roles which are considered consumers and producers of services in today's information market are assigned to different business roles.
- **Regulatory**: Due to regulatory constraints, certain separations of business roles are induced.

All business roles play the role of user and provider towards specific other business roles. Whether a provider or user role is played is determined by the contract governing the interaction between the business roles.

Business roles can be combined in business administrative domains to suit the needs of the stakeholder for its particular business.

A business relationship expresses the interaction requirements between two business roles. The manifestation of a business relationship between two business administrative domains is the reference point.

A.3.1.4 Reference point

The reference point consists of several viewpoint-related specifications governed by a contract. A reference point specification is split into reference point segments. Each segment is a meaningful, self-consistent specification.

The reference point is the aggregation of the specifications of all supported viewpoints. The reference point shall contain the following specifications:

- **Business part**: Scope limitations, functional and non-functional requirements posed on the business relationship by the business roles. It is derived from the requirements of the business roles for their interaction.
- **Information part**: Defines the information that is shared between the business administrative domains.
- **Computational part**: Defines interfaces on computational objects to be made accessible to the other domain.

- **Engineering part**: Defines the separations of the supporting distributed infrastructure in nodes, signalling and control links, supporting operating systems and protocol stacks needed for interactions between the business administrative domains.
- **Miscellaneous part**: Defines other constraints, e.g. limitations on other specifications imported into a reference point specification, allowed limitations on compliance, etc.

A.3.2 Segmentation of reference points

A reference point specification is segmented. A reference point segment is a meaningful and consistent cross-section of a reference point specification.

The segmentation into access and usage is driven by the isolation of functionality controlling and managing the business administrative domain interaction (access functionality) from the other functionality providing and managing services (usage functionality).

The generic segments of a reference point are:

- access segment; and
- usage segment.

The Usage segment is further segmented into:

- primary segment; and
- ancillary segment.

The reference point segmentation depends on the business relationships that are combined into the reference point and the actual functionality introduced by the business relationships into the reference point. Further segments are possible.

A.3.2.1 Access and usage segmentation

See Figure A.1.

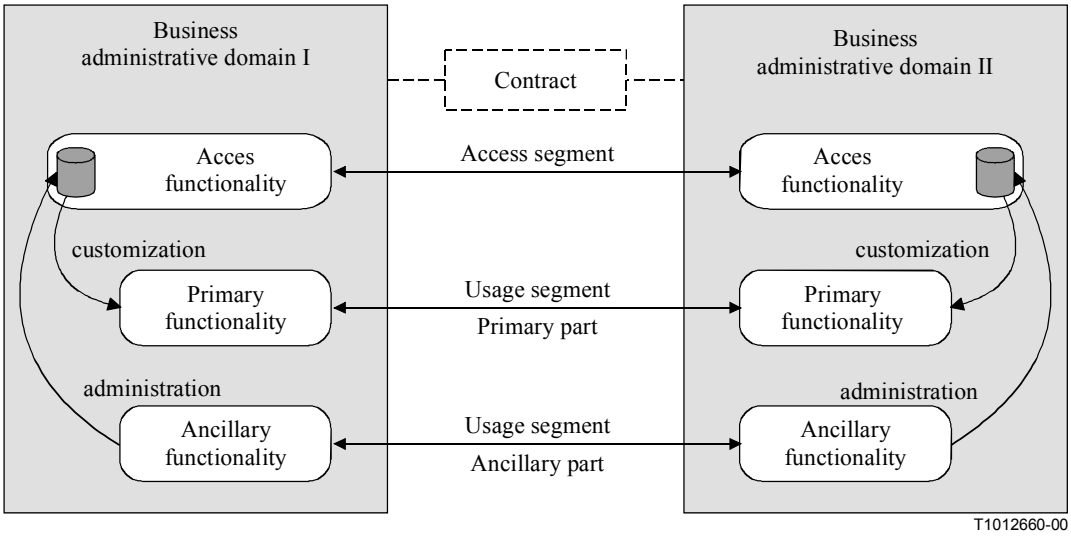

Figure A.1/Z.600 – Generic reference point segmentation

The functionality of the Access segment of a reference point are:

- initiate dialogue between the business administrative domains;
- identify the business administrative domains to each other;

NOTE – Either business administrative domain may remain anonymous depending on the requested interaction.

- establish a secure association between the business administrative domains;
- set up the context for the control and management of usage functionality:
  - the context specifies which services are offered and under which conditions;
  - the context can be changed dynamically over time.
- initiate the Usage segment of the reference point between the business administrative domains.

A.3.2.2 Primary and ancillary segmentation

The Usage segment is specific to the actual services provided between the business administrative domains. The Usage segment can be further segmented according to:

- the use (direct: e.g. a Video-on-Demand (VoD) service versus indirect: e.g. fault management for VoD);
- the impact in the domain (e.g. provisioning vs e.g. management vs e.g. administration).

The Usage segment of a reference point is segmented into:

- primary usage functionality;
- ancillary usage functionality.

The Primary usage segment covers the main objective of the contract between the business administrative domains. The functionality of the primary usage segment of a reference point are:

- control of the service life-cycle;
- exchange of service content.

The Ancillary usage segment includes the support functionality of the primary usage or access segments. The functionality of the ancillary usage segment of the reference point are:

- set and manage the context for a specific service or set of services;
- set and manage the context for domain administration, i.e. functions that are not specific for a single service or a set of services;
- carry out administration functions;
- carry out management functions;
- control the life-cycle and attributes of ancillary usage services.

The ancillary usage segment does not fulfil the primary contractual purpose of a business relationship; it has no independent value, but rather adds value to the primary usage segment. The ancillary usage segment may modify information and policies that are used for decision making in the access segment.

Different versions of business relationships can exist. Each of these different versions is called a profile of the business relationship. A profile is implemented by the reference point.

A.3.3 Combination of business roles into business administrative domains

As shown in Figure A.2, one or more business roles can be combined into a single business administrative domain which is owned by a single stakeholder.

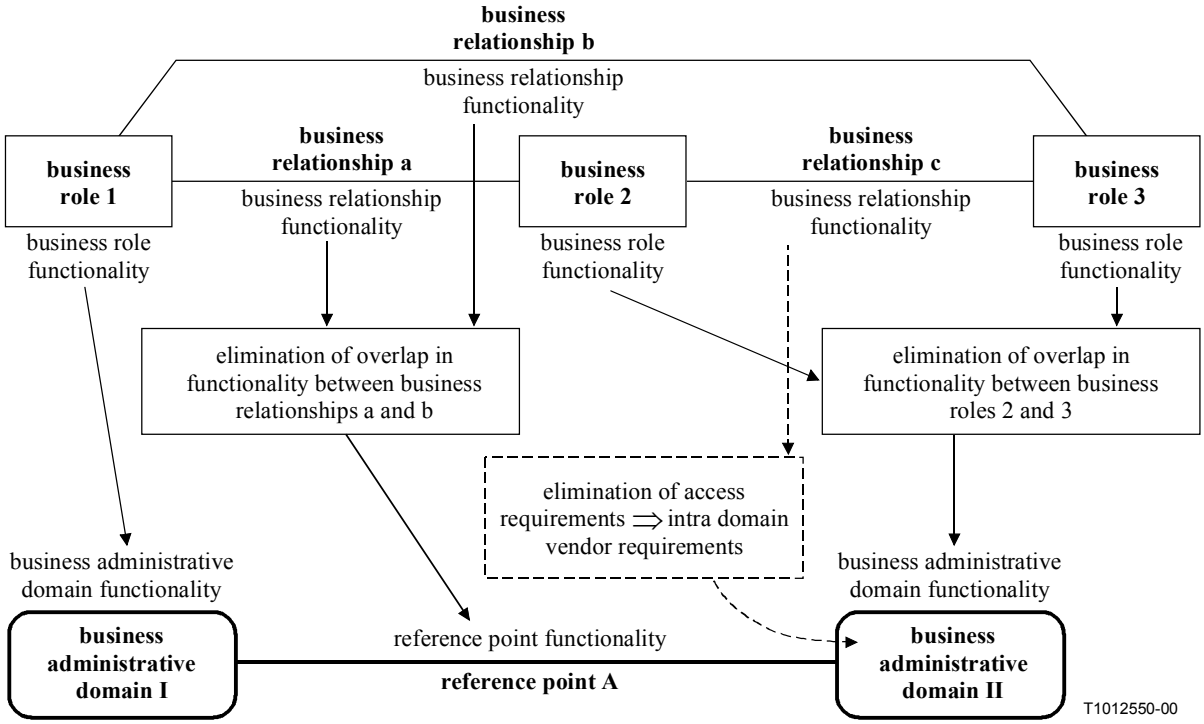

Figure A.2/Z.600 – Combination of business roles into business administrative domains

The interactions between business role 1, business role 2 and business role 3 are expressed by the business relationships a (between business roles 1 and 2); b (between business roles 1 and 3); and c (between business roles 2 and 3).

The business role 1 is performed in the business administrative domain I, and the business roles 2 and 3 are performed in the business administrative domain II.

The reference point A, between the business administrative domains I and II is defined as the combined functionality of the business relationships a and b, after the elimination of the overlap in functionality among the business relationships a and b.

Similarly the business administrative domain II performs the combined functions of business roles 2 and 3, after the elimination of the overlap in functionality among the business roles 2 and 3.

The implementation of the business relationship c in not visible outside the business administrative domain II. A reference point that implements the business relationship c is not mandatory. It may still be implemented within the business administrative domain II, in many cases though by eliminating the access segment, since there would be no need for access control within a single business administrative domain.

A.3.4 Delegation

The segmentation of reference point specifications allows:

- reuse of reference point segments in other reference point specifications; and
- shared functionality between reference points.

This is illustrated between reference points A and B in Figure A.3 and is called delegation.

Delegation can be static or dynamic:

- Static delegation is used where the delegated segments do not change during the duration of the contract.

- Dynamic delegation is used when the delegated segments may vary over time during the duration of the contract.

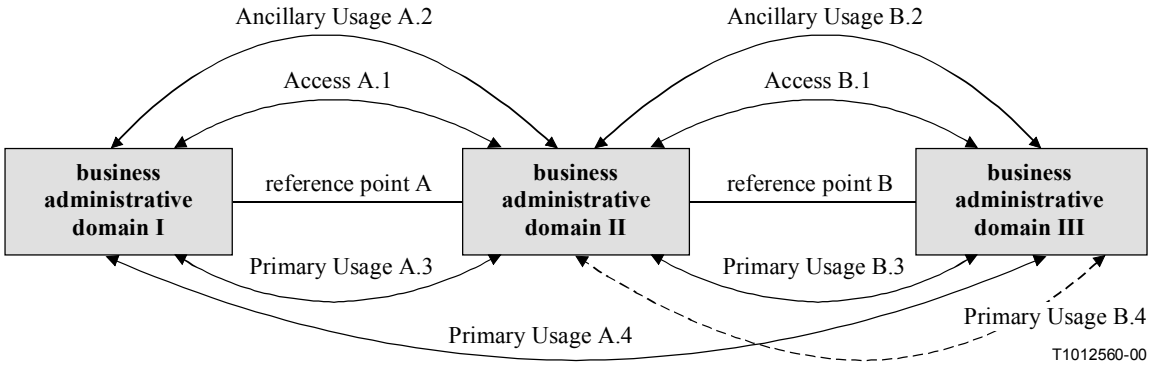

Figure A.3/Z.600 – Delegation of reference point segment functionality

Reference point A, between business administrative domain I and II is segmented into:

- Access A.1;
- Ancillary Usage A.2;
- Primary Usage A.3; and
- Primary Usage A.4.

Reference point B, between business administrative domain II and III is segmented into:

- Access B.1;
- Ancillary Usage B.2;
- Primary Usage B.3; and
- Primary Usage B.4

Business administrative domain II does not add value to reference point segment A.4. Reference point segment B.4, part of the business relationship between business administrative domain II and business administrative domain III, is functionally identical to A.4.

If allowed by the respective contracts between business administrative domains I, II and III the reference point segment A.4 can be implemented directly between business administrative domains I and III as a segment of reference point B.

ANNEX B

Engineering modelling concepts

This annex provides an explanation of the relationships between RM-ODP modelling concepts so far as such relationships need to be concerned in the DPE architecture. This annex also provides the appropriate references to RM-ODP (X.900 series of ITU-T Recommendations) and ITU ODL (ITU-T Z.130) for the explanation of these relationships.

NOTE – Although RM-ODP specifies an engineering language, it does not provide a framework for the mapping of a computational specification, provided in ODL or IDL, onto the engineering language. There is clearly a lack of methodology for supporting this mapping. Direct mapping of IDL and ODL specifications to a specific programming language (i.e. C++, Java, etc.), are available today ([6] and [4]) bypassing the engineering language. The direct language mapping introduces a dependency from the target programming environment on the computational specification, so that at least the deployment of applications onto several nodes depends on the capabilities of the target environment. The issue of computational to engineering view mapping is for further study.

B.1 Basic concepts

A series of engineering concepts introduced in the RM-ODP are fundamental to the DPE architecture. These involve the use of objects, capsules, and clusters in DPE nodes and which are reviewed below and illustrated in Figure B.1. These concepts can be mapped to available distributed object technologies like OMG's Common Object Request Broker Architecture (CORBA) or other proprietary object communication technologies.

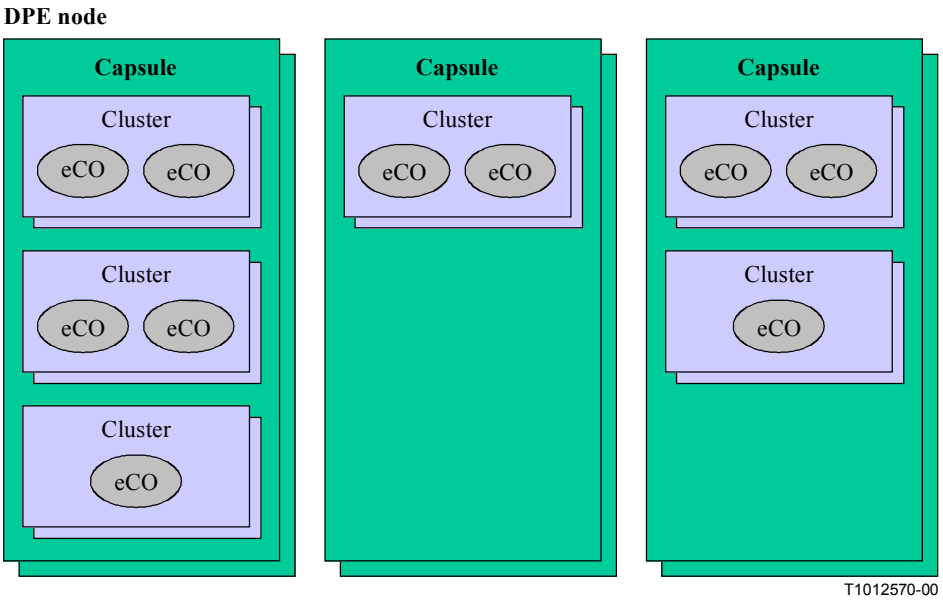

Figure B.1/Z.600 – Basic engineering concepts

A computational object in ITU-T Z.130 is an aggregation of interfaces. During the design phase a specification of a computational object with multiple interfaces can be refined and represented as a collection of computational objects, each supporting a single interface. CORBA 2.2 can only support the representation of single interface computational objects. Supporting computational objects with multiple interfaces in CORBA can be accomplished by the instantiation of run-time supporting engineering objects, which keep track of the aggregation of the interfaces belonging to a computational object with multiple interfaces.

DPE node: A DPE node is the engineering abstraction of a computing system. It is a model for a collection of resources that are computing autonomously. Being autonomous means that a node is capable of operating independently of other DPE nodes, but in an actual case the DPE node may interact with other DPE nodes and be managed by other nodes. A DPE node provides the mechanisms to support distribution transparency and hides the underlying computing infrastructure from the application developer. The computing resources of a node are:

- Processing resources; e.g. processes, tasks, threads, scheduler.
- Memory resources; e.g. volatile memory, persistent memory.

- Communication resources; e.g. intra-node communication mechanisms (e.g. IPC) and network access mechanisms (e.g. communication drivers, protocol stacks).

A DPE node consists of:

- a DPE kernel;
- engineering services;
- engineering computational objects (eCOs); and
- protocol objects.

The DPE kernel provides a node management service at one or more node management interfaces.

A DPE node is capable of interworking with other DPE nodes using some standardized protocol. A DPE node is a unit of network connectivity and network management. The network's responsibility interconnection and management ends at a node's boundary. For connectivity inside a node, the node assumes responsibility.

Examples of DPE nodes are:

- a single computer (with CPU, memory, hard disk) capable of independent operation;
- a parallel processing system;
- a system distributed over a local area network (LAN) with a distributed operating system;
- a cluster of computers arranged in a way to provide a high availability node.

DPE kernel: A DPE node is under the control of the DPE kernel which is responsible for initializing the node, creating groups of objects, making communications facilities available, and providing essential services like timing, threading, and scheduling. The DPE kernel is a set of functions that controls the DPE node. The DPE kernel may be implemented on top of NCCEs. From an application or management point of view one sees only the abstracted functionality provided by the DPE kernel instead of the functionality of the NCCE.

For example in a DPE that utilises different proprietary platforms, the DPE kernel would have different specializations for these types of NCCEs.

Capsule: A capsule is the engineering unit for allocation and encapsulation of a DPE node's computing resources. A node will typically contain several capsules. A capsule can be regarded as a virtual machine or process with some specific Quality of Service (QoS) guarantees. A capsule is a unit of management, control and protection. The objects deployed to this unit share the same allocation policy for allocating resources from a DPE node's kernel. This allocation policy is different from the allocation policies of other capsules in the same DPE node.

Depending on the kernel's approach to resource management, the node's resources are shared among the node's capsules or partitioned and exclusively used by each capsule. In the latter case, the objects in one capsule have their own pool of resources and are independent of objects in other capsules. In the former case, resources are shared and the objects observe the same delays or resource shortages.

A capsule manager provides the capsule management service at a capsule management interface.

A capsule will typically contain several clusters.

Cluster: The smallest grouping of objects is the cluster. The objects in a cluster are grouped together in order to reduce the cost of manipulating them. A cluster is a set of eCOs which combines at the same time the properties of being a unit of placement, activation, deactivation, checkpointing, reactivation, recovery and migration. Clusters are encapsulated in a capsule. No eCO can belong to two different clusters.

The relationship between the eCOs within a cluster is always local. The following rules hold for their interactions:

- **Inter-cluster interaction**: The DPE concept for modelling interactions between eCOs in different clusters is the channel.
- **Intra-cluster interaction**: Interaction mechanisms for eCOs within a cluster are not prescribed by the DPE.

A unit of instantiation is not considered as a property of the cluster; objects can be added to and removed from a cluster dynamically.

A cluster manager provides the cluster management service at a cluster management interface.

Channel: A channel is a set of supporting mechanisms to allow communication between clusters. The set of mechanisms needed to manage and control interactions between clusters are part of a channel, which is made up of a number of interacting engineering objects (specifically stub, binder and protocol objects).

APPENDIX I

Object life cycle scenario

This appendix elaborates on the object life cycle presented in 6.1.

A typical scenario for the object life cycle is the following:

1) An application is developed as a set of computational objects (COs), encompassing the source codes (in C++ for example), and the objects' structure and interface descriptions (in ODL).
2) Object source codes and interface descriptions are translated into executable code.
3) **Deployment**: Object executable codes are deployed onto one or several nodes in the form of (or a set of) eCO templates. Having the eCO template and execution code gives the node the ability to create an eCO instance. Cluster templates can also be deployed to enable instantiation of clusters.
4) **Creation**: Sets of eCOs are created in clusters and initialized.
5) **Activation**: Clusters are activated for execution by allocating computing resources to the objects and enabling their interfaces.
6) **Deactivation**: Clusters are deactivated.
7) **Deletion**: Clusters are deleted.
8) **Withdrawal**: eCO templates are withdrawn.

Step 1 is supported by a service creation environment, which is not described within the DPE Architecture. Step 2 is supported by compilers such as C++ compilers and an ODL compiler.

The DPE Architecture provides the model and the functions to support steps 3-8. The Software Distribution/Installation service addresses steps 3 and 8. The Life Cycle Service addresses steps 4-7.

APPENDIX II

Flexible binding

This appendix elaborates on bindings as described in 6.4 and 6.6.

Classically, binding in computer systems denotes a set of actions or processes for associating or interconnecting different objects of a computing system. Binding implies setting up an access path between two objects, which in turn typically comprises locating the target of the access path, checking access rights, setting up appropriate data structures in support of the access path to enable communication between objects following this path. This notion of binding covers both operating system level binding, which typically takes place between different address spaces, and language level binding, which typically establishes a mapping between a variable and a target object, and takes place within the same address space. In distributed systems, binding usually combines features of operating system level binding and language level binding and includes, in addition, the mobilization of communication resources to support remote (i.e. on different DPE nodes) interactions between objects.

Examples that combine operating system and language level bindings are RPC systems, where binding sets up typed object stubs and a supporting connection to a server, and persistent object systems where binding sets up a memory cache object itself bound to a disk location.

Providing a flexible integration of system and language level binding is an important requirement for open distributed systems. Even in the standard client-server case, there is a wide variety of operation invocation semantics and implementation mechanisms that reflect different application requirements. For instance, servers may be persistent, replicated, or may use caches managed with various policies to improve performance, etc. More generally, in particular multimedia computer-mediated communication, communications between objects may take many different forms, including many-to-many interactions and exchange of continuous flows of data. These applications typically require the support of a combination of different communication schemes and of specialized protocols. For example, it has long been noted that RPC does not provide good support for the exchange of (a-temporal) streams of data [10]; this is even more obvious for the exchange of continuous streams such as audio or video [11]. These applications also require explicit control over configurations of objects, and typically, over the setting up of communication paths between objects.

Since a single universal mechanism for all these cases is highly unlikely, an open distributed system should provide a framework in which many different sorts of binding with different communication semantics can be efficiently supported and combined.

APPENDIX III

Issues for the support of stream interfaces

This appendix elaborates on stream interfaces as described in 6.6.2 and 6.6.3.

Support for stream interfaces raises a number of issues:

- what extensions, if any, are needed to define stream interfaces?
- what does a stream interface consist of? A unique flow, multiple unidirectional flows, bidirectional flows, etc.?
- how to manage groups of stream interfaces?
- what kind of language mapping will be provided for stream interfaces?

It is worth making a comparison between the multimedia stream and binding support advocated here and other approaches such as the IMA MSS[1]. The IMA MSS specification provides support for continuous flows and allows explicit control over the configuration of multimedia flows between objects as follows: The MSS specification models media devices as objects with multiple interfaces called virtual devices. A virtual device possesses a server interface which we shall call its control interface (see Note) and which provides its clients with means to observe media stream positions and to control the flow of media data. A virtual device also comprises ports which correspond to input or output mechanisms for the virtual device. Ports within a device are designated by indexes, but do not provide any visible interface. Each port is associated with a format interface that provides an abstraction of data type information (e.g. type of encoding, frame format, etc.). Ports can be linked by virtual connections. Instantiating a virtual connection from an appropriate factory results in a control interface of a type similar to that of the control interface of a virtual device. The virtual connection control interface offers a connect operation to link an output port to one or more input ports (depending on the type of the virtual connection, unicast or multicast).

The TINA and ODP concept of binding object subsumes that of a virtual connection in the MSS specification and the TINA and ODP notion of stream interface generalizes the notion of port in the MSS specification. The MSS specification, however, does not fulfil the requirement for stream interfaces stated here. The MSS specification does not explain how ports can be implemented and accessed. In contrast, the present requirement calls for programmable stream interfaces: programmers can thus provide implementations for stream interfaces, much in the same way they provide implementations for operation interfaces. Just as mandated by RM-ODP, stream interfaces could be manipulated in the same way as operation interfaces. In contrast, ports in the MSS specification are not viewed as interfaces and are subject to ad hoc handling (for instance, they can only be referred to by means of integer indexes within a given virtual device).

The requirement on the programmability of stream interfaces is motivated by several reasons. The availability of low-cost general-purpose platforms with multimedia capabilities allows a software handling of continuous flows. Programmers building applications handling flows (e.g. enhancing a Web browser with live audio and video) should be insulated from low-level media format, protocol-dependent characteristics and the details of specific hardware devices.

NOTE – The MSS specification uses the term "stream object" to denote the control interface. To avoid confusion, we stick to the ODP terminology, and rename MSS stream objects into MSS control interfaces.

[1] The Interactive Multimedia Association (IMA) has redirected its efforts away from creating technical standards for interoperability and more into Intellectual Property and other market issues and has merged with the Software Publishers' Assocation.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	**Languages and general software aspects for telecommunication systems**

Geneva, 2002