Telecommunications
Information
Networking
Architecture
Consortium

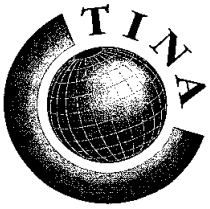
TINA-C Deliverable

Issue Status: Publicly Released

Computational Modelling Concepts

Version: 3.2

Date of Issue: 17th May 1996



Telecommunications
Information
Networking
Architecture
Consortium

TINA-C Stream Deliverable

Issue Status: **Reviewed baseline**

Computational Modelling Concepts

Version: 3.2

Date of Issue: **17th May 1996**

Abstract: This document describes the TINA-C Computational Modelling Concepts. These concepts provide the framework for the computational specification of distributed telecommunications applications. They provide the framework for describing an application in terms of computational entities that interact with each other, with discrete communication as well as continuous media communication.

Main Author(s):	**Many TINA-C Core Team Members**
Editors:	**Tom Handegård**
Stream:	**DPE**
Workplan Task:	**8a, 17, 194**
File Location:	**/u/tinac/96/dpe/docs/computational_model/v3.2/**
Archiving Label:	**TP_HC.012_3.2_96**

Extended Abstract

This document describes the TINA computational modelling concepts (CMC). These concepts provide the framework for the computational specification of distributed telecommunication applications. The concepts are based on emerging standards and specifications such as RM-ODP, [28], [29], and the OMG's object model [30] and CORBA architecture [31], and the results of several ongoing projects.

The TINA CMC is one of the parts of the TINA Computing Architecture (see Section 2.2), which is based on the RM-ODP viewpoints. Other aspects of the architecture are described in two other documents: *Information Modelling Concepts* [2], providing the framework for the information specification of distributed systems, and *Engineering Modelling Concepts* [3], providing the framework for the engineering specification of distributed systems.

In the computational viewpoint, a distributed application consists of a collection of computational objects (hereafter, unless qualified, the term "object" denotes a computational object) that interact with each other via interfaces (see Section 3). An object provides one or more interfaces to enable other objects to access its capabilities. Some of these interfaces may be created when the object is created while some may be offered (i.e. created) dynamically during the lifetime of the object. At least one interface, the object's initial interface, is created when the object is created.

Objects interact either by invoking operations and sending responses or by means of stream flows. Interfaces used for the former kind of interaction are called operational interfaces, and interfaces used for the latter kind are called stream interfaces.

The interactions that occur at an operational interface are structured in terms of invocations of one or more computational operations (or operations for short) and responses to these invocations (see Section 3.2). Operations are classified into two kinds: Interrogations and announcements. In an interaction via an interrogation operation, a result is passed back from the server to the client after the operation invocation has been processed by the server. In an interaction via an announcement, no result is passed from the server to the client, and the client is not informed of the outcome of the invocation.

A stream interface (see Section 3.3) is an abstraction that represents a communication endpoint that is the source for some information flows and a sink for some other 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.

The nonfunctional aspects of interfaces operations and stream flows, such as quality of service parameters, and configuration information are specified using trading attributes and quality of service attributes associated with the interfaces and operations and flows (see Section 3.1.5 and Section 3.1.6).

Computational specification structures are defined for defining object types and interface types (Section 3.5). A computational object type (or object type) is a template (or schema) for a class (or collection) of objects, and each member of this class is considered an instance of that type. Every object is an instance of some object type. Similarly, a computational interface type (or interface type) is a template (or schema) for a class (or collection) of interfaces, and each member of this class is considered an instance of that type. Every

interface is an instance of some interface type. The concrete syntax supporting the definition of object templates and interfaces templates, TINA-ODL (Object Definition Language), is described in a separate document [5].

In a distributed application, objects may interact with each other in complex ways and engage in several activities. While many of these activities are application specific, there are certain generic management operations that can be performed on objects, such as object creation, object deletion, interface creation, etc. These generic management functions are described in Section 4.1.

Establishment and control of interactions between computational objects may be done either implicitly by the infrastructure (the Distributed Processing Environment, DPE) or explicitly by applications. The former are called implicit bindings and the latter are called explicit bindings. Explicit bindings are permitted to enable applications to define complex forms of communication, such as group communication, and to enable application-level control of communication. Explicit binding is mandatory for stream interfaces. A binding object is an abstraction that represents a communication session, the session model being defined by the template of the binding object. A binding object encapsulates the mechanisms required for the control of the communication session that it represents, and provides operations that other objects can invoke to control the communication session. A binding object is instantiated as a result of an explicit request to a computational object that is capable of establishing sessions of one or more models, each model defined by a binding object template.

An object group is a collection of objects (and component groups) that is built, installed, maintained, and managed as a unit. It can be thought of as a “subsystem” or “software package”. An object group is an encapsulated unit, in the sense that only a subset of the interfaces offered by the objects in the group are visible outside the group. Interfaces that are visible outside the group are called contracts. Objects that are located in different groups interact only via contracts. Interfaces that are not contracts cannot be accessed by objects outside the group. Section 5 describes the basic properties of object groups as well as the operations necessary to perform basic life cycle management (i.e. create, delete) on them.

The question of transaction model for the CMC is currently being discussed. The current proposal can be found in Appendix A. In the proposed model, a transaction is an activity that spans one or more objects and has the properties of atomicity, consistency, isolation, and durability. A transaction is initiated when an operation that is designated as being transactional is invoked. Invocation of transaction initiation operations can be nested giving rise to nested transactions. The concept of conflicting transactional operations has been defined and it is used to define “serializability” as a possible correctness criterion for transactions. The transaction model also allows the relaxation of serializability to improve concurrency and allow semantic recoverability. An updated version of the transaction model is expected for the next version of this document.

The sections of this document are organized as follows:

- Section 1 (**Introduction**) describes the scope and provides some context for the document.
- Section 2 (**Requirements and Related Work**) describes the requirements on the computational modelling concepts and identifies related work.
- Section 3 (**Computational Structures**) describes the computational structures by which distributed applications are composed.

- Section 4 (**Object Management and Binding Model**) presents the dynamic behavior of objects and specifies the different forms of interactions that may occur between computational entities.
- Section 5 (**Object Groups**) describes concepts related to object grouping.
- Section 6 (**Further Work**) discusses problems that need further work.
- Appendix A (**TINA Transaction Model**) describes a proposed transaction model. It specifies how transactions spanning multiple objects are constructed and describes the synchronization and recovery requirements that are implied by the transaction model.
- A list of references, and an acronym list can be found at the end of the document. A glossary of terms for all the TINA-C deliverables can be found in [9].

The first appearance of a glossary term in the main text is emboldened.

Understanding Recommended Before Reading this Deliverable

A general understanding of distributed computing concepts is helpful to understand better the computational modelling concepts described here. However, this is not a necessary prerequisite for reading this document.

Readers who have never read any TINA-C output are referred to:

- [19] “TINA-C: Towards Telecommunications Information Services” ISS’95 — gives a brief overview
- [1] “Overall Concepts and Principles” 1994 baseline — gives a full overview

For better understanding of the TINA Computing Architecture, readers are referred to (in addition to this document):

- [2] “Information Modelling Concepts” 1994 baseline
- [3] “Engineering Modelling Concepts (DPE Architecture)” 1994 baseline
- [4] “DPE Infrastructure Description” 1995 report
- [5] “TINA Object Definition Language (TINA-ODL) Manual” 1996 baseline

Changes from Last Version

Major Changes Made from previous draft (Version 3.1)

The major changes from the previous version of the document are related to streams and object groups.

The definition/discussion of stream interfaces, stream interface compatibility rules, and stream binding has been updated to be in line as much as possible with both reviewers' comments and current results in TINA-C [8].

The section on object grouping has been restructured and expanded. An exhaustive example has been added to the chapter. The object group concept is now believed to be more concrete than before.

A third point worth mentioning is that the previous concept of *service attributes* has been replaced by two other concepts: *Trading attributes* and *quality of service attributes*. This is believed to reflect more clearly the intended uses of this feature(s).

As a final remark, note that Appendix A has not been updated since the previous version of the document (version 3.1), even though comments have been received on the proposed transaction model. The view is that further considerations and discussions are needed before a revised version of the transaction model can be provided. A revised version is expected for the next version of the computational modelling concepts.

Version History

Version Number	Date of Issue	Issue Status	Major Changes from Previous Version	
			Reviewer	Review Comments
1.0	Dec 1993	Baseline	• first official release of document, label TB_A2.NAT.002_1.0_93	
				•
2.0	Feb 1995	Baseline	• Second issue of baseline, document label TB_A2.HC.012_1.2_94	
				•

Version Number	Date of Issue	Issue Status	Major Changes from Previous Version	
			Reviewer	Review Comments
3.0	7th August 1995	Stream reviewed	• Document restructured to fit new document template. • List of transparencies moved from Section 1 to a more logical place in Section 2. • A subsection on graphical conventions added in Section 1. • A section about the relationship between Computational Modelling Concepts and the other parts of the TINA Computing Architecture has been added (Section 2.2.2). • The section on the relationship with the OMG object model has been extended (Section 2.3.2). • The old section 3 has been split into two sections. The current Section 3 now covers old 3.1-3.5, section 4 being old sections 3.6-3.7. • The definition of the dynamic aspects of interfaces have been changed. Interfaces now have a separate state, and can be dynamically created (Sections 3.1.2 and 4.1.1). • A section (3.1.4) describing service attributes has been added. The definition in this document is now consistent with the one in [5]. • A section (3.2) characterizing object composition/decomposition has been added. • A new definition of grouping has been proposed. The section on grouping (Section 5) is completely new. • The section on the transaction model (formerly section 4) has been extensively updated. It is currently placed in an appendix (Appendix A) since it needs to be assessed. • The section on functional separations (formerly Section 6) has been moved into an appendix because these principles have more to do with design methodology than with the Computational Modelling Concepts as such. The section may be removed in the next version of this document. • The ODL-definition (formerly Section 7) have been removed from the document. ODL has been updated, and is now documented in a separate document [5]. • ODL-examples (formerly Appendix B) have been removed because they were outdated and because ODL no longer is described in this document.	
			Core Team	• The descriptions of dynamic interface creation and interface specific states are not clear. • The section about interface bindings (4.2) is slightly confusing. • Clarify purpose of object grouping. Make more consistent use of terminology. Provide more examples. • Remove the appendix on functional separations.

Version Number	Date of Issue	Issue Status	Major Changes from Previous Version	
			Reviewer	Review Comments
3.1	11th October 1995	Core Team reviewed	• The extended abstract has been updated. • Description of dynamic interface creation, initial interface, and interface/object state clarified. • Removed section 3.7, “Computational objects and OSI agents” • Section about interface bindings (4.2) restructured. • The section about object grouping has been cleaned up: A description of purpose has been added, terminology has been made more consistent, and care has been taken to avoid confusion with OSI Management and TMN concepts. • The glossary was removed. A glossary for all TINA-C deliverables can now be found in [9]. • The appendix on functional separations was removed.	
			Reviewers	• see /u/tinac/95/dpe/docs/cmc/review/v3.1/comments.fm4
3.2	17th May 1996	Baseline	• Updated descriptions of transparencies to be in line with ODP • Slightly enhanced the comparison with OMG object model • The explanation of the multiple interface concept has been extended/clarified • A section on “initial interface” has been added • The previous and rather unclear “service attribute” concept has been replaced by two other concepts: “trading attributes” and “quality of service attributes”. • The semantics of announcements have been clarified • The definition of stream interfaces has been updated extensively to be in line with current thinking (e.g. as expressed in the document “Unification of Session/Connection Graphs, Stream Interfaces, and Channel Model” [8]) • The discussion of compatibility rules for stream interfaces has also been updated • The sections on object and interface creation/deletion/activation/deletion have been cleaned up • The section on interface binding has been completely restructured and expanded, especially in the area of stream interface bindings • The section on object grouping has been restructured and expanded. An exhaustive example has been added to the chapter. The object group concept is now more concrete than before. • Many minor edits/clarifications have been made...to many to mention here.	
				•

Table of Contents

1. Introduction

1.1 Purpose

1.1.1 Objective of the Document

This document describes the **TINA Computational Modelling Concepts** (CMC). These concepts provide the framework for the **computational specification** of distributed applications that can be found in telecommunication information networks. The computational specification of a distributed application describes the application in terms of computational entities (or program components) that interact with each other. It specifies the structures by which the interactions occur and also specifies the semantics or behaviour of these interactions. The computational modelling concepts described in this document provide the framework for such computational specifications. The applications under the purview of these modelling concepts may provide either generic distributed processing functions, network resource management functions, services to customers of the network, or functions for the management of such services.

1.1.2 Relationship with other TINA-C Work

The TINA CMC is one of the parts of the **TINA Computing Architecture** (see Section 2.2). Other aspects of the architecture are described in two other documents: “Information Modelling Concepts” [2], providing the framework for the **information specification** of distributed systems, and “Engineering Modelling Concepts” [3], providing the framework for the **engineering specification** of distributed systems.

The execution environment for applications built according to the computational modelling concepts described here is provided by an infrastructure called the **Distributed Processing Environment** (DPE). The TINA DPE is described in [4].

A language has been developed to support the development of computational specifications: **TINA Object Definition Language** (ODL), described in [5] (see also Section 3.5).

1.2 Audience

All members of the TINA-C Core Team and TINA-C auxiliary projects should read this document. Anyone interested in knowing about the distributed processing framework of the TINA-C Architecture should also read this document.

1.3 Graphical Conventions

In this document and other TINA documents, the following graphical conventions have been adopted for representing graphically computational specifications (derived from the ANSA Computational Model [22])

- Solid rectangles represent objects,
- Small darkened boxes represent operational interfaces,
- Small white boxes represent stream interfaces, small white circles represent sources of stream flow, and small darkened circles represent sinks of stream flow,
- A (thin) line from an object to an interface denotes that the object is a client of the interface,

- A thick line between a stream source and a stream sink indicates a stream flow.
- A grey/dashed rectangle enclosing a collection of objects represents an object group.
- An interface that touches/crosses an object group rectangle, represents a contract.

Figure 1-1 illustrates the diagrammatic conventions.

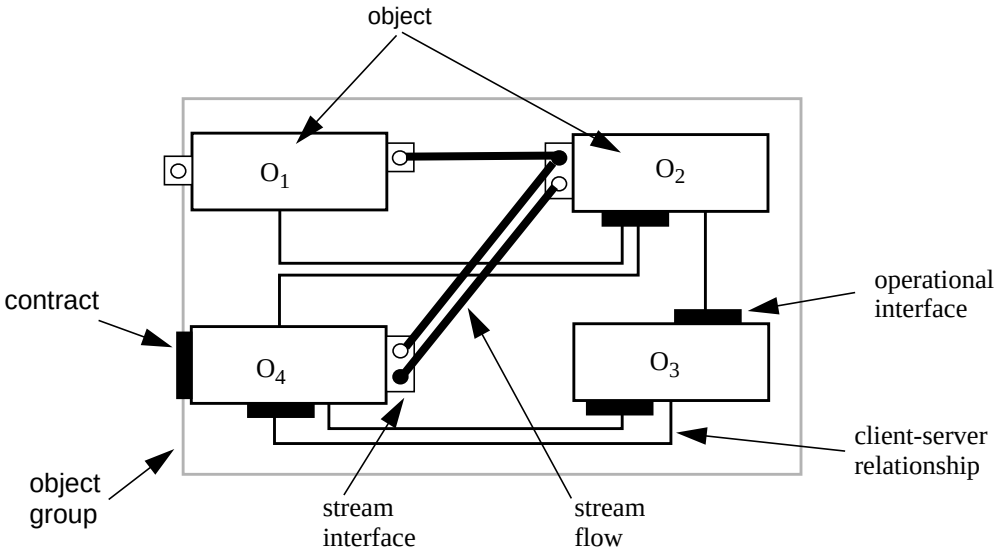


Figure 1-1. Graphical Representation of Objects

2. Requirements and Related Work

The first part of this section describes the objectives and requirements on the Computational Modelling Concepts. The other three parts describe the relationships between the CMC and the other parts of the TINA Computing Architecture, important related standards, and other related work.

2.1 Objectives of the Computational Modelling Concepts

2.1.1 Requirements

The TINA-C computational modelling concepts should satisfy the following requirements:

- Object Interaction Description:
  - Concepts should be defined for describing interactions between application components.
  - The concepts should encompass both discrete communication and continuous media communication, such as audio and video communication.
- Object Service Description:
  - The modelling concepts should address both functional and nonfunctional aspects of distributed applications. The nonfunctional aspects are those related to quality of service, configuration, and enterprise policy issues such as accounting and security policies.
- Object Structuring:
  - The modelling concepts should address the evolutionary nature of distributed applications. Concepts to describe units of application components that can be independently released or upgraded should be defined.
- Object Template Description:
  - The modelling concepts should define the framework for the specification of application interfaces including both structure and semantics. Notations and specification structures for computational specifications should be specified.[1] This is the objective of TINA-ODL, described in [5].

An important requirement on the modelling concepts supporting the objectives listed above is to build on existing standards and results wherever possible.

2.1.2 Distribution Transparencies

A central theme in the design of computational modelling concepts is that the complex details of mechanisms required for interaction between application components that are remote from each other and that may not have been developed together should be invisible

1. It is noted here that some computational models in the literature, such as the Reference Model for Open Distributed Processing (RM-ODP) [29], consider specification notations and structures to be outside the scope of the computational model. Although this separation is laudable from the viewpoint of genericity of the model, it has been required that the TINA-C computational modelling concepts should prescribe specification structures. This requirement is imposed to facilitate communication and exchange of specifications among TINA-C Core Team members and participants of TINA-C auxiliary projects.

in the computational specification of a distributed application. This process of hiding the effects of distribution is known as **distribution transparency** [28], [29]. Several dimensions of distribution transparencies have been identified in the literature, and some of the major transparencies are listed below.

- **Access transparency**: A transparency which hides details of marshalling messages according to specific message formats and data representation conventions that are required for communication between heterogeneous computing systems. This transparency will generally be provided by default.
- **Location transparency**: A transparency which hides the location of an application component from other components identifying and binding to it. This transparency enables flexible placement of application components. It requires that components are named in a way that is independent of actual physical location.
- **Relocation transparency**: A transparency that hides relocation of a component from other components bound to it. This transparency enables applications to continue to operate even if some components change location or are replaced.
- **Migration transparency**: A transparency that hides location changes from the component being relocated. This transparency enables the component to be relocated preserving its state across the migration without disrupting other components interacting with it. Migration can be used for e.g. dynamic load balancing and reconfiguration.
- **Failure transparency**: A transparency which hides the failures and recovery actions of an application component from other components or itself. Many different classes of failure can be identified for distributed systems; interaction failure (communication failure, security failure,...), binding failure, instantiation failure, etc. When failure transparency is provided for a class of failures, the designer can work in an idealized world in which the corresponding failures does not occur.
- **Replication transparency**: A transparency which hides the replication of an application component from other components that interact with it. Replication is used for attaining high availability and/or performance.
- **Persistence transparency**: A transparency which hides from an application component the deactivation and reactivation of other components or itself. Deactivation and reactivation of components are often used when a system is unable to provide it with processing, storage and communication resources continuously.
- **Transaction transparency**: A transparency which hides the coordination of activities amongst a configuration of components (i.e. interactions between the application components and the components that realize the transaction control functions) to achieve consistency.

Different applications may have different transparency requirements. Hence, it is important that the computational modelling concepts enable application designs that use transparencies selectively. All the transparencies described above may be used selectively. A transparency may be used throughout a complete system, or just for a part of it.

If a particular transparency is not selected, this means that the concerns covered by the transparency must be solved directly as part of the application design. If the transparency is selected, this means that the designer relies on “standard” solutions to be used. These standard solutions can e.g. be incorporated by the system building tools in use, such as compilers, linkers and configuration managers. The designer then works in a world which is transparent to that particular concern.

2.2 TINA Computing Architecture

2.2.1 Objectives

The **TINA Computing Architecture** defines the modelling concepts that should be used to specify object-oriented software in TINA systems. It also defines the distributed processing environment (DPE) that provides the support system allowing objects to locate and interact with each other. These concepts are based on the Reference Model for Open Distributed Processing (RM-ODP) [28][29], and especially on the notion of **viewpoints**. TINA Computing Architecture defines modelling concepts for three viewpoints, used for specifying a telecom system:[2]

- **Information modelling concepts** [2] focus on the semantics of information and information processing activities in the system. It provides the framework for the **information specification** of distributed systems. An information specification describes a distributed system in terms of information held by the system components and information exchanged between the components. The information may relate to the operation and use of the system as well as to the management of the system itself. The information specification of a system focuses only on the information elements of the system and their relationships. It describes the system without regard to the distribution machinery and does not deal with explicit interaction among the information elements.
- **Computational modelling concepts** focus on the decomposition of the system into a set of interacting objects which are candidates for distribution. It provides the framework for the **computational specification** of distributed systems. It is the objective of the current document to describe this framework.
- **Engineering modelling concepts** [3] focus on the infrastructure required to support distribution. It provides the framework for the **engineering specification** of distributed systems and on the description of an abstract infrastructure, called the **Distributed Processing Environment** (DPE) [4]. An engineering specification specifies the distributed system in terms of components that are actually distributed on a set of computing nodes, and the models and mechanisms to support their execution and interaction.

In addition, related to the **enterprise viewpoint**, a business model for TINA is currently being developed, focusing on the purpose and scope of TINA [14].

2. RM-ODP defines in addition the technology viewpoint, which focuses on the choice of technology to support the system. The TINA Computing Architecture does not define modelling concepts for this viewpoint.

2.2.2 Relationships Between Computational and other Modelling Concepts

2.2.2.1 Information Modelling Concepts

The information modelling concepts described in [2] and the computational modelling concepts described here have an intimate relationship.

The information specification and the computational specification of a system can be related in two ways. In one, the information specification occurs earlier in the design phase, and the information specification forms the basis for the computational specification which occurs later in the design phase. This approach seems natural in data intensive applications. In the other method, the computational specification and the information specification are developed together, and the information specification is used for a precise characterization of the behaviour of the computational entities. In this approach, the information elements are used for representing the information exchanged and the effects of information exchange between the computational entities.

2.2.2.2 Engineering Modelling Concepts

The engineering modelling concepts described in [3] relates to the computational modelling concepts described here in two ways.

First, an engineering specification describes how the computational specification will be realized in terms of the components (clusters, capsules) that are actually distributed on a set of computing nodes in the distributed environment.

Secondly, the concepts, mechanisms, and distribution transparencies assumed in the computational specifications of TINA applications have to be supported by the infrastructure (i.e. the DPE, see [4]). The engineering modelling concepts allow the description of this infrastructure. The DPE description includes the description of support for:

- *Object interaction*: A list of distribution transparencies have been identified TINA Computing Architecture (Section 2.1.2), and is assumed for computational specifications of TINA applications. Therefore, DPE mechanisms supporting these transparencies are described. Moreover, concepts and mechanisms for supporting object interactions have to be defined in order to provide a uniform model for interaction to application developers. This communication model defines the concepts of channels, stubs, binders, protocol adapters, etc.
- *Object life-cycle*: A common support from any DPE platform to the life-cycle of TINA objects (object deployment and withdrawal, object creation and destruction, object activation and de-activation, etc.) requires the definition of concepts, models, and mechanisms in the DPE Architecture.

The DPE may provide some additional functions, especially some DPE Services and management functions [4], that are not required by the computational modelling concepts, but are very useful for distributed applications.

2.3 Related Standards

2.3.1 RM-ODP Prescriptive Model

The computational modelling concepts prescribed in the RM-ODP Prescriptive Model [29] have been used as the major baseline for the modelling concepts described in this document. The key concepts borrowed from RM-ODP are the following:

- A distributed application is composed of objects that interact with each other.
- An object offers its capabilities to other objects via one or more interfaces.
- Interaction between objects is either by means of operation invocations and responses, or by means of stream flows where each flow is a unidirectional bit stream.
- To facilitate dynamic configurations of objects, the concept of trading is used. An object that needs to use some capability need not have the reference to any specific interface providing the capability, but can discover an appropriate interface during execution by interacting with a match-maker object called the trader.
- Transactions with ACID properties (see Appendix A for details) are initiated by invoking operations that are designated as being transactional operations.
- To enable applications to define new forms of interactions, such as multicast group communication, that are not built into the basic interaction model, and to enable application level control of such interactions, the notions of explicit binding and binding objects have been defined. See Section 4.2 for details.

While RM-ODP has been adopted as the basis for the modelling concepts, it was also felt that some additions are needed to RM-ODP concepts. Modelling concepts to describe collections of computational objects that have some properties in common will be helpful in describing the structure of large applications.

In addition to these new modelling concepts that are needed, the additional concepts introduced in this document and in [5] are related to specification structures for computational specifications. In this respect, the scope of this document goes beyond the scope of RM-ODP computational model.

2.3.2 OMG Object Model

The Object Management Group (OMG) has developed an object model for distributed applications and has specified an architecture for the infrastructure, called Object Management Architecture (OMA), that supports the object model [30], [31]. The OMG object model is viewed as a subset of RM-ODP computational model, and hence it was decided to use RM-ODP model as the basis for the modelling concepts described in this document.

Some differences have to be noted however between OMG Object Model and the object model defined in this document (hereafter referred to as the TINA Object Model). OMG Object Model focuses on the support for object interaction. As such, it defines a notation called Interface Definition Language (OMG-IDL, defined in [31]) that serves as a description of the interfaces of objects supporting operation invocations. CORBA Objects are each seen as a single interface providing services through invocation of its operations.[3]

The objectives of the TINA Object Model, developed as a support for the definition of telecom applications, extend the ones of OMG. The support to object interaction encompasses discrete interaction (operation invocation), but also continuous interactions (data streams).The support to continuous interaction between objects is provided by the definition of stream flows, which is an addition to OMG interaction model. The notion of explicit binding of interfaces included in the TINA Object Model is also a powerful extension compared to the OMG object model. It allows the application designer to define complex forms of interactions between computational objects without requiring them to be built-in modelling concepts.

Moreover, the TINA Object Model support objects with multiple interfaces that can be dynamically created and deleted. This is a feature that can be used as a basis to solve several issues; e.g. it allows an object to support several distinct services operation on a common state (e.g. one interface for normal usage, and one interface for management of the normal service), or to support different versions of the same service. It may also be used to maintain per-client session contexts. Yet another possibility is to use interfaces as the granularity for access control.

The TINA Object Model also supports the concept of object groups; collections of objects that are built, installed, maintained and managed as units. Object groups help designers and system administrators cope with the complexities of large distributed systems by adding modularity and a framework for application management (see Section 5).

Given the extended objective of TINA object model compared to OMG object model, its supporting language (TINA-ODL) has been defined as an extension to OMG IDL (see [5]).

See also [21] for a more thorough comparison of the OMG and TINA object models.

2.4 Other Related Work

The following is a list of work that are similar to and have influenced the TINA Computational Modelling Concepts.

2.4.1 ANSA Computational Model

The Advanced Network Systems Architecture (ANSA) Computational Model defined in [22], [23] is the basis for the definition of the RM-ODP model. Hence, the description contained in Section 2.3.1 applies here.

2.4.2 INA/OSCA Computational Model

The Information Networking Architecture (INA) Computational Model described in [26] has several similarities with the CMC. This is not surprising since the INA Computational Model also adopted RM-ODP as its basis. Compared to the INA Computational Model, the notion of objects is used in a more fundamental manner in this document. In the INA model, applications are composed of *building blocks*, and a building block is a package made up of a collection of objects. Interfaces of objects within a building block that can be used by objects in other building blocks are called contracts. (The notions of building blocks and contracts

3. The OMG has now issued a Request for Proposals (RFP) called “Multiple Interfaces and Composition”. This RFP requests technology that can support objects with multiple interfaces, very much in line with the TINA object model.

were originally introduced, without the characterization using objects, in the OSCA architecture [32].) The INA model was focussed primarily on building blocks and contracts and less on the internal object structure of building blocks.

2.4.3 MODA

Another related work is the CSELT computational model MODA (a Model for an Object Distributed Architecture). This model also builds on the object model of RM-ODP and extends it to incorporate the notions of building block and contract.

2.4.4 SERENITE

The SERENITE Computational Model builds strictly on the object model of RM-ODP [35]. The SERENITE model includes some refinements for quality of service specification and explicit binding. Further, it proposes template structures for abstractions that are useful in telecommunication applications, such as binding object templates for multimedia services, and templates for transport network entities. These templates have been proposed as specializations of the templates prescribed in the RM-ODP model.

2.4.5 ROSA Object Model

The RACE Open Service Architecture (ROSA) Object Model can be viewed as a subset of the object model of RM-ODP [34]. For example, in the ROSA model, an object has only one interface, and transactional operations have not been defined. On the other hand, the specification notation, called COOLISH, that supports the ROSA object model has some features that in the future possibly can be incorporated in the TINA computational specification language ODL.

3. Computational Structures

This section identifies the basic computational structures (or entities or elements) by which distributed applications are composed. All interactions between computational entities occur via interfaces offered by the entities. The interaction model defines a type system for the classification of interfaces and specifies the notion of compatibility between interfaces.

Additional computational structures are defined in Section 5.

3.1 Object Structure

In the computational viewpoint, a distributed application consists of a collection of **computational object**s. Hereafter, in this document, unless otherwise qualified, the term “object” denotes only a computational object.

3.1.1 Objects and Interfaces

An object encapsulates data (or state) and behavior. An object provides a set of capabilities (or functions) that can be used by other objects. This collection of capabilities is grouped into one or more subsets. Each such subset of capabilities is called a **service**.[1] Thus, an object provides one or more services that other objects can use. To enable other objects to access a service, the object that provides the service offers (or provides) a **computational interface** (hereafter simply referred to as an **interface**) which is the only means by which other objects can use the service. Thus, an object may offer several interfaces, each providing a service. The interfaces are the only interaction points for other objects. An object is the smallest unit of distribution in the distributed system.[2]

Objects process operations. Operations on different objects are processed concurrently to one another. The activity structure within an object is not specified in this interaction model. In particular, an object may process concurrently several operation invocations and responses, and thus may have several concurrent activities within it. Depending on the semantics of an interface, concurrent usage of the interface by multiple objects may or may not be possible. For example, concurrent use of an interface for fetching information from an object may be allowed, while concurrent use of an interface for managing (i.e. updating) the object may not be allowed.

The specification of an interface includes both functional aspects and nonfunctional aspects. The nonfunctional aspects are aspects such as performance and other quality of service parameters, some forms of distribution transparency constraints, configuration

1. What is described here is the computational notion of service. In the telecommunications area other notions of service exist. E.g. a “telecommunication service” is defined as a set of capabilities provided by the network to a customer of the network. This notion of service belongs to the enterprise viewpoint. Telecommunication services are defined for the purpose of being marketed and sold to customers. These “enterprise services” will typically be realized by one or more “computational services”. Furthermore, a single computational service may be reused in different enterprise service offerings.
2. The design process of a computational specification may use decomposition or composition to organize the design cycle in successive steps, each one dealing with a different level of abstraction. This permits a complex object to be decomposed into a number of simpler objects which may also be decomposed at a lower level of abstraction. In the finalized computational specification, however, there will be no objects that can be considered as compositions of other computational objects.

information such as the identities of physical resources encapsulated, and cost of using the service. These non-behavioral aspects are specified using **quality of service attributes** and **trading attributes** associated with interfaces.

Depending on the nature of interactions that occur at an interface, the interface is classified as being either an **operational interface** or a **stream interface**. Each of these interface kinds is described in Section 3.2 and Section 3.3.

3.1.2 Multiple Interface Objects

An object may provide multiple interfaces. Some of these interfaces may be created when the object is created while some may be offered (i.e. created) dynamically during the execution of the object. The multiple interfaces provided by an object may be of different types, they may be multiple instances of the same interface type, or a mix of both.

Different interfaces of *different types* may be used by an object to provide services differ in the nature of interactions. Consider, for example, an object that represents a video camera. Control operations on the camera can be accomplished via an operational interface while transmission of video signals can occur via a stream interface. Interfaces of different types may also be used to provide logically distinct services. E.g. consider an object that provides file storage and access services. Such an object may provide two interfaces: one for sequential file access, and another for random access. As a third example, interfaces of different types may by used by an object in order to simultaneously support different versions of the same service.

Different interface instances of *the same type* may be used to offer the same service simultaneously to several clients by providing one interface to each client. The object is able to distinguish invocations of the same operation on different interfaces, and there is a local state associated with each interface instance (see below). This can e.g. be used by the object to maintain a separate session context (session state) for each client, or to perform different functions, such as billing different accounts, depending on which interface instance is invoked. Different interface instances of the same type may also differ in the values of their service attributes. This may be used by an object to provide the same service with different quality of service (and price) characteristics.

Note that an object *does not have to* offer a separate interface instance for each client accessing it. In many cases it will make perfectly sense for several clients to access the same interface instance. This depends on the service provided by the interface.

Figure 3-1 shows an example of an object offering three interfaces: two operational interfaces (IO_1 and IO_2) and one stream interface (IS). IO_1 and IO_2 may be of different types or of the same type.

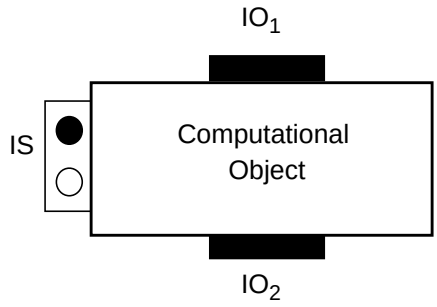


Figure 3-1. Object with multiple interfaces

Each interface instance offered by an object may have a separate state associated to it. This “local” state is accessed only when operations are invoked on this particular interface instance. Thus, the state encapsulated by an object is divided into two parts: One part that is local to each interface instance and is accessed when operations are invoked on this particular interface instance, and one “core” (common) part that is accessed by all operations irrespective of which interface that was invoked. As explained above, the notion of an interface specific part of the state is useful when an object offers multiple instances of the same interface type. E.g. if an object provides a separate interface instance to each client accessing a service, the interface specific part of the state could be used to hold the identity of the interface’s client.

It should be noted here that although an interface has an associated state, it is not a separate entity. An interface is a part of the object offering that interface.

3.1.3 Initial Interface

At least one interface, the object’s **initial interface**, is created when an object is created. The initial interface must be an operational interface. A reference to the initial interface will be returned to the creator of the newly created object. The initial interface will typically contain operations to initiate the newly created object, and operations to create additional interfaces. See also Section 4.1.1.

3.1.4 Interface Reference

In order for an object to use an interface, it needs to possess a **reference** to the interface. A reference to an interface (also called an **interface reference**) is a descriptor that unambiguously identifies the interface. An object may obtain a reference to an interface by one of the following means:

- Receiving the reference as an argument or result in an interaction with another object. The latter may be the object that offers the interface or some third party object.
- Creation of an object. The reference to the initial interface will then be returned to the creator. The request to create a new object must be directed to a third-party object that is capable of creating new objects, i.e. a factory object. The third-party object will return the initial interface reference to the object that requested creation.

- Creation of an interface. An interface is created upon request from the object that supports/offers it (the instantiating object). As a result of the creation, an interface reference is returned to the instantiating object. The reference can be passed to other objects as arguments or results of interactions.

It is possible that more than one object possess a reference to the same interface. It is also possible that an object possesses references to several interfaces. An object may use interfaces offered by itself.

Note that interface references may be either arguments of operations, results, or both.

3.1.5 Quality of Service Attributes

Quality of service attributes are associated with operations and stream flows to specify non-functional aspects.[3] The quality of service attributes associated with an operation or a flow address the timing constraints, degree of parallelism, availability guarantees and so forth, to be provided by that component (i.e. stream flow or operation). Quality of service specifications allow statement about the “level of service” offered by components. Quality of service information deals with the provision of the service, rather than the service itself.

One ore more quality of service attributes can be associated with each operation or flow. The attributes are typed. In general, values can be provided for the attributes at interface specification time, or dynamically assigned at interface creation. The values may even be altered during the lifetime of the interface, e.g. at interface bind time.

Examples of quality of service attributes for operations are:

- Maximum response time of the operation;
- Minimum invocation interval of the operation;
- Maximum invocation interval of the operation.

Examples of quality of service attributes for stream flows are:

- The throughput of the flow;
- The maximum jitter of the flow;
- The sampling rate of the flow (e.g. for an audio flow).

Note that this document only provides the concept necessary to associate quality of service statements with the operations and flows. It does not provide the *semantics* of these quality of service specifications. The problem of semantics may be illustrated when considering an example; say the quality of service attribute “bandwidth” is associated with a flow, and assigned the value “2 Mbits”. This may e.g. be interpreted as

- a statement of a mandatory capability, i.e. the interface cannot be offered unless it can source/sink 2 Mbits, no more, no less,
- a statement of expectation, i.e. typically connections are made at 2 Mbits, but other values can be negotiated, or
- a statement of support, i.e. 2 Mbits is the maximum supported bandwidth of connections are 2 Mbits, and only lower values may be negotiated.

3. The idea of associating quality of service attributes with interfaces and objects is for further study.

A quality of service framework that specifies the semantics of the quality of service attributes is outside the scope of this document. An initial draft of a quality of service framework for TINA can be found in [15].

3.1.6 Trading Attributes

The DPE component of the TINA architecture supports the notion of a Trading service [4]. Objects can advertise the references of their offered interfaces at the Trading service. Potential clients of the offered interfaces can search the Trader service for interfaces offers. Typically, the clients base their search on a combination of interface type, context (a hierarchical partitioning of the search space), and **trading attribute** values.

Trading attributes are associated with interfaces and may be used to specify properties of the interface that are of interest during trading. Examples of such properties are:

- The owner of the interface, i.e. the name of the organization that provides the service, and
- The price for using the service.

Each interface type has an individual set of zero or more trading attributes. The trading attributes have a name and a type. Values are assigned to the attributes on a per interface instance basis. The values must be assigned before interfaces are advertised at the Trader service. It is expected that the trading attributes as specified in the interface templates are visible to application developers. The attributes are intended to be utilized in application code when exporting and importing interfaces to the Trader service.

3.2 Operational Interfaces

3.2.1 Description

The interactions that occur at an operational interface are structured in terms of invocations of one or more **computational operations** (or **operations** for short) and possibly responses to these invocations. With reference to an operational interface, the object that provides the interface is called the **server** and an object that invokes operations defined in the interface is called a **client**.[4]

4. Note that “client” and “server” are roles played by an object in its interactions with other objects. An object may be a client in its interaction with one object and be a server in its interaction with another object.

Figure 3-2 shows an example of client - server relationship. An object may be a client of several interfaces. (See object O_4 in Figure 3-2.) Note also that an object may be a server of one interface while being a client of another interface. (See objects O_2, O_3, and O_4 in Figure 3-2.) It is also possible that two objects exist such that each is a client of an interface offered by the other. (See objects O_3 and O_4 in Figure 3-2.)

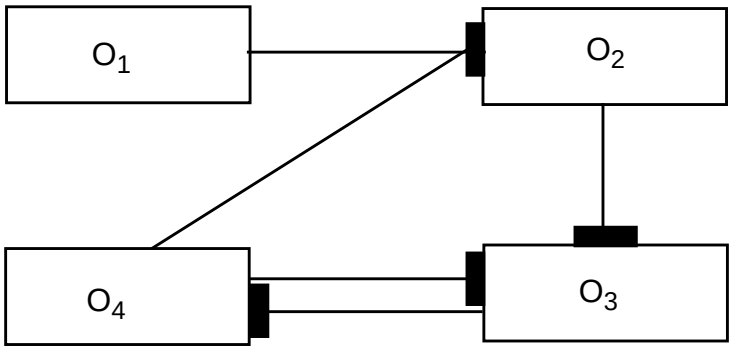


Figure 3-2. Client- Server Relationship

The specification of an operational interface is done by means of an **operational interface template**, see Section 3.5.3.1.

3.2.2 Operations

An operation in an operational interface is the interaction mechanism by which a capability (or function) provided by the server can be accessed by a client. Operations are classified into two kinds: **interrogation**s and **announcement**s.

3.2.2.1 Interrogations

An interaction between a client and a server by means of an interrogation consists of the following actions:

- Passing of zero or more arguments from the client to the server
- The processing of the invocation by the server
- Passing of zero or more results from the server to the client upon completion of invocation processing. Irrespective of whether the server returns any result or not, the client is *always* informed (by the infrastructure) of the outcome of the invocation, i.e., whether the invocation completed successfully or failed due to some reason, such as communication failure, and unavailability of the server.

An invocation of an interrogation can be either **blocking** or **nonblocking**. In a blocking invocation, the invoker waits until a response is received for the invoked operation. In a nonblocking invocation, the invoker does not wait for the response to be received, and retrieves the response at some later time. The definition of an operational interface does not specify whether an interrogation is to be invoked in blocking or nonblocking mode. This is purely a matter of the internal organization of the invoker object.

A response always follows the corresponding invocation, however no order of delivery is guaranteed between different invocations.

Furthermore, an interrogation can be identified as transactional. A **transaction** is an atomic unit of consistent and reliable computing activity composed of a sequence of operations involving one or more objects (see Appendix A of a description of the transaction model).

3.2.2.2 Announcement

An interaction between a client and a server by means of an announcement consists of the following actions:

- Passing of zero or more arguments from the client to the server
- The processing of the invocation by the server

Unlike an interaction via an interrogation, in an interaction via an announcement, nothing is passed back from the server to the client, and the client is not informed of the outcome (success or failure) of the invocation. Delivery of the invocation to the server is not guaranteed, i.e. the invocation semantics are “best effort”.

All invocations of announcements are nonblocking. No order of delivery is guaranteed between different invocations, i.e. the invocations may no be delivered to the server in the same sequence as they were invoked from the client.

3.2.3 Examples

The following are some examples of interfaces that are operational interfaces:

- An interface that provides operations for establishment and release of network connections between two termination points in a network
- An interface that provides creation, deletion, retrieval, and update operations on one or more instances of some information object types (i.e., object types identified in a specification from the information viewpoint)
- An interface that provides control operations on a video camera such as start, stop, and zoom.

3.3 Stream Interfaces

3.3.1 Description

A **stream interface** is an abstraction that represents the endpoint of a **stream**. A stream is an aggregation of one or more **stream flows** (or flows for short). When objects interact via stream interfaces, the information exchange occurs in the form of stream flows, where each stream flow has a type that is capable of being supported by the underlying infrastructure (e.g. “audio” or “video”). The flows are unidirectional. They are terminated in **flow endpoints** within stream interfaces. Each flow endpoint is either a **source** or a **sink** of the flow. A stream interface can contain a mixture of source and sink flow endpoints.

The type of each flow endpoint characterizes the data that is produced or consumed by the endpoint, e.g. “audio”, “video”, “MPEG1”, or “unrestricted bitstream”. One or more quality of service (QoS) parameters are also associated with each flow endpoint. A distinction is made between **offered QoS** for source endpoints, and **required QoS** for sinks. The num-

ber and types of the QoS parameters depends on the flow type. Typically, they may include timing requirements (delay, throughput, etc.), and synchronization requirements between flows.

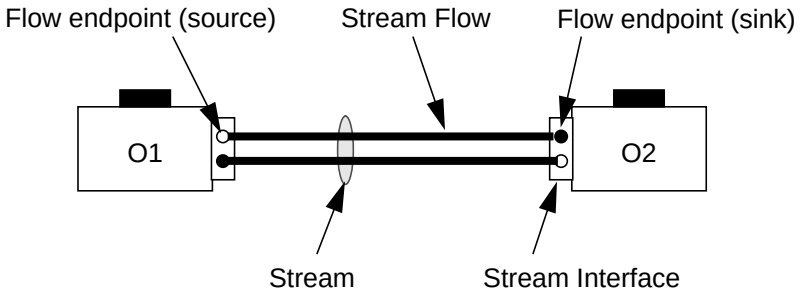


Figure 3-3. Stream communication concepts

3.3.2 Stream Interaction

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. When stream interfaces are bound, the flow endpoints of the interfaces are connected to each other. A precondition for a successful binding action is that the types and QoS parameters of the involved flow endpoints can be “matched”. See Section 4.2 for further discussion about interface bindings.

The exact nature of the interactions through a stream interface is not specified within these modelling concepts. The idea is that once the stream interfaces of two objects have been bound, a set of “information pipes” have been set up between the objects with some specific quality of service guarantees. Thereafter, the producer of a pipe inserts information into the pipe, and the consumer retrieves information from the pipe and consumes them.

Broadly speaking, there are two alternatives for how data enters and exits the stream flows. The first alternative is that the computational object that offers a stream interface is also be the producer and consumer of the data entering and exiting the flows within the interface. This alternative could informally be described as the “flows being terminated in software”. From an implementation point of view, one can imagine that the implementor of the computational object has specific programming language functions (e.g. “read_data” and “write_data”) at his disposition to interact with the stream flows.[5]

The second alternative is that the data are entering and exiting the flows directly from hardware devices. This alternative could informally be described as “flows being terminated in hardware”. When streams are used to carry multimedia information such as audio and video, this may be a desirable solution for performance reasons. The flow endpoints are then representations of these hardware devices. The computational object offering the interface will not interact with the flows, but only control them. Control of the streams is executed by creation of interface instances, binding of the interfaces, and manipulation of the bindings.

5. Such a programming model for streams is outside the scope of the Computational Modelling Concepts. It can be defined as part of an ODL language mapping.

3.3.3 Examples

The following are examples of stream interfaces:

- An interface that is used for video distribution. The interface typically consists of two flow endpoints: A source for an audio stream flow and a source for a video stream flow
- An interface that supports one party in a multimedia conference service. It typically consists four flow endpoints: A video sink, a video source, an audio sink and an audio source
- An interface that is used for bulk data transfer between computational objects (e.g. implemented using TCP over frame relay). The interface typically consists of two flow endpoints (to support data transfer in both directions): A data sink and a data source.

3.4 Operational Interfaces versus Stream Interfaces

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 precise structure is imposed on interactions via stream interfaces. Loosely we can say that the paradigm for interaction via stream interfaces is that of asynchronous message passing, while the paradigm for interaction via operational interfaces is that of a remote procedure call.[6] The asynchronous message passing paradigm is particularly useful for continuous media communication.

Another difference between operational and stream interfaces is that in the case of operational interactions, the object that offers the interface always takes part in the interactions, while in the case of stream interactions, the object offering the interface may be restricted to just controlling the communication.

A third difference between operational and stream interfaces relates to binding. Operational interfaces can be either explicitly or implicitly bound, the latter alternative probably being most commonly applied. Stream interfaces are always explicitly bound. Note also that in the case of operational interfaces, only the server object offers an interface, which the client is then bound to. There is no notion of a client interface. This is different for stream interfaces, where both the parties involved in a binding must offer an interface. See Section 4.2 for further discussion about interface bindings.

A fourth difference is related to interface specification and typing. For operational interfaces there are general rules for interface compatibility, stating under which conditions an interface of one type can be transparently substituted for an interface of another type.[7] For stream interfaces, the notion of interface compatibility is more complicated. Whether or not one stream interface can be regarded as compatible with another one depends much more on the capabilities of the binding mechanism and the supporting infrastructure than in the operational case. No general rules for stream interface compatibility are defined in this document. See Section 3.5.4 and Section 4.2 for further discussion of these issues.

6. This not imply that clients have to be blocked upon operation invocation, as explained in Section 3.2.
7. In ODL, these compatibility rules are associated with subtyping and inheritance.

3.5 Types and Compatibility Rules

3.5.1 Types and Instances

Specification structures are defined for defining computational interface, object, and object group types.[8] A **computational object type** (or **object type**) is a template (or schema) for a class (or collection) of objects, and each member of this class is considered an instance of that type.[9] Every object is an instance of some object type. Similarly, a **computational interface type** (or **interface type**) is a template (or schema) for a class (or collection) of interfaces, and each member of this class is considered an instance of that type. Every interface is an instance of some interface type.

The concrete syntax supporting the definition of object templates and interface templates is called TINA Object Definition Language (TINA ODL), and is defined in [5].

3.5.2 Object Type Declaration

An object template specifies the following information:

- The capabilities provided by the object in terms of supported interface types (operational or stream). This specifies that the object instances has the potential to offer interfaces of these interface types.
- The behaviour of the object, in terms of the functions performed by the object in providing services at its interfaces.[10] The behavior specification also includes identification of required interfaces, which specifies interface types used by instances of the object type to provide their services.
- An initialization specification that identifies an interface type, a reference to which will be returned to the creator when an instance of the object is created. The initial interface will typically be used to initialize the newly created object, and to get references to other interfaces supported by the object.

3.5.3 Interface Type Declaration

An interface type which is the template for a class of operational interfaces is called an **operational interface type**. An interface type which is the template for a class of stream interfaces is called a **stream interface type**.

3.5.3.1 Operational Interface Template

An operational interface template specifies the following information:

- The signature of each operation that can be invoked by clients of interfaces that are instances of the type. An operation signature specifies the following information:
  - the name of the operation
  - an indication whether the operation is an interrogation or an announcement

8. Specification structures for object groups are described in Section 5.
9. In general, a type can be defined by any arbitrary form of predicate. However, throughout this document, only a particular predicate form is used for defining types, namely template structures. Thus, associated with every type is a template that defines the type.
10. Currently only natural language (e.g. English) is used, see [5] for details.

- the type of each argument of the operation
- in the case of interrogations, a set of **termination** signatures, where each termination signature describes one possible termination of the operation by identifying the termination name and the types of result parameters of the termination, and
- The name and type of each quality of the service attribute associated with the operation. In the case of interrogations, a QoS attribute may e.g. be used to indicate the maximum response time of the interrogation.

• The behavior or the semantics of each operation, including sequencing constraints, and concurrency constraints applicable to the operations. At the moment, this part is specified informally using a natural language (e.g. English).

• The name and type of each trading attribute.

3.5.3.2 Stream Interface Template

A stream interface template specifies the following information:

• The signature of each stream flow that can occur at interfaces that are instances of the type. A stream flow signature specifies the following information:
  - the name of the flow,
  - the information type of the flow,
  - a producer/consumer attribute that specifies whether an object that offers the interface is a producer or consumer of the stream flow, and
  - The name and type of each quality of service attribute associated with the flow. A QoS attribute may e.g. be used to indicate the required throughput of the flow.

• The behavior of the interface. Currently this part is specified informally using a natural language (e.g. English).

• The name and type of each trading attribute.

3.5.4 Interface Compatibility Rules

Quite often, in a large telecommunications network, new functionalities are introduced by incrementally refining existing functionalities. Thus, it is possible that within a network, there are two services A and B, offered by two different objects such that service A includes all capabilities of service B and some additional capabilities. For example, service A may provide multi-party communication capabilities, while service B provides two-party communication capabilities. In such cases, the flexibility of transparently substituting service A for service B is quite desirable. It allows gradual phasing out of old services and a smooth transition to new services.

To support this flexibility, the notion of **interface type compatibility** is introduced in this interaction model. If an interface type A is compatible with another interface type B, then interfaces of type A can be transparently substituted for interfaces of type B. For example, when a client invokes operations defined in an operational interface of type B, the DPE may choose a server that provides an interface of type A that meets the client's quality of service requirements to receive the invocations and provide the necessary responses. Notice that the compatibility relationship is not necessarily symmetric; i.e., type A being compatible with type B does not necessarily mean that type B is compatible with type A.

3.5.4.1 Compatibility Rules for Operational Interfaces

For operational interfaces, the notion of compatibility is defined more precisely below. This definition uses only the operation signatures information and is based on the subtyping rules for operational interfaces defined in RM-ODP [29]. Note that the rules only define *signature compatibility*, which is sufficient to ensure that a substituted interface can consistently interpret the structure of any interactions that may occur. Signature compatibility is a necessary, but not sufficient, condition to ensure *behavior compatibility*.

An operational interface type A is compatible with another operational interface type B if the following conditions are met:[11]

- For every operation signature X defined in B, there is an operation signature Y defined in A such that the following conditions are met:
  - if X is an interrogation so is Y; conversely, if X is an announcement so is Y
  - the name components of X and Y are identical
  - X and Y have the same number and ordering of arguments[12]
  - the type of each argument of X is *compatible* with the type of the positionally corresponding argument in Y
  - all termination names in Y are also in X
  - for every termination name that is in Y (as well as X), the termination signatures in both X and Y have the same number and ordering of result parameters, and the type of each result parameter of the termination in Y is *compatible* with the type of the positionally corresponding result parameter of the termination in X.
  - for each QoS attribute defined in X there is a QoS attribute in Y with identical name and type, and vice versa.[13]
- For every trading attribute U defined in B, there is a trading attribute T defined in A with identical name and a type such that the type of U is compatible with the type of T.

In the above definition, primitive data types, such as integer, boolean, and character, are also viewed as predefined types. Compatibility rules for such types are assumed to be predefined.

The above definition of interface type compatibility is intended to serve as a guideline for application designers when they introduce new interface types and to help the designers to determine whether a new interface type is compatible with some existing interface types. These rules can also be used by a function within the DPE to determine whether two interface types are compatible based only on operation signatures and service attribute specifications.

---

11. These rules are adequate only for nonrecursive interface types. Rules for recursive interface types (without the service attributes) have been defined in [29].
12. The positional correspondence requirement can be relaxed in situations where the infrastructure permits parameter names (or tags) to be included in invocation and response messages.
13. It is recognized that this clause may be too restrictive. The compatibility requirement with respect to Quality of Service attributes depends on the quality of service architecture, which outside the scope of this document.

3.5.4.2 Compatibility Rules for Stream Interfaces

It is not possible to define completely general compatibility rules for stream interfaces since these rules depend on the semantics of the interactions that are being abstracted in the definitions of the streams. For example, a composite audio and video interface might be considered compatible with an audio interface, to allow a remote telephone user to communicate with a user of a video phone. As another example, a PAL video flow might be considered compatible with an NTSC video flow, to allow terminals using different video codings to connect to each other.

Any type system defined for stream interfaces will have an associated set of compatibility rules. As explained above, these compatibility rules differ from the compatibility rules for computational interfaces in that they may allow communications between objects whose interfaces offer different capabilities. Definition of a specific type system for stream interfaces is outside the scope of this document. It is possible to envisage that different type systems are defined for different applications. Binding objects (see Section 4.2) will be responsible for enforcing the compatibility rules of any given type system at binding time.

As a general guideline, one can say that compatibility rules for stream interfaces will consist of two steps [27]:

- identification of corresponding flow endpoints in the two interface types, and decision on whether the correspondences found are sufficient for compatibility to exist; and
- comparison of the types and QoS aspects of each pair of corresponding flows, to determine whether compatibility exists.

3.6 Examples

3.6.1 An Example: Service Interfaces and Management Interfaces

As discussed in Section 3.1.2, in general, interactions with an object via one interface of an object are observable and controllable through interactions via other interfaces of the object. A scenario in which this is particularly useful is one in which an object provides one or more interfaces, called service interfaces, for some functionalities, and one or more interfaces, called management interfaces, for managing or controlling the functionalities accessible via the service interfaces. Thus, in general, an object provides one or more service interfaces and one or more management interfaces. This is illustrated in Figure 3-4. Note that the concepts of service interfaces and management interfaces are introduced here only as a typical example of a scenario where the concept of objects with multiple interfaces is very useful. The service interfaces - management interfaces structure described here is not an architectural requirement.

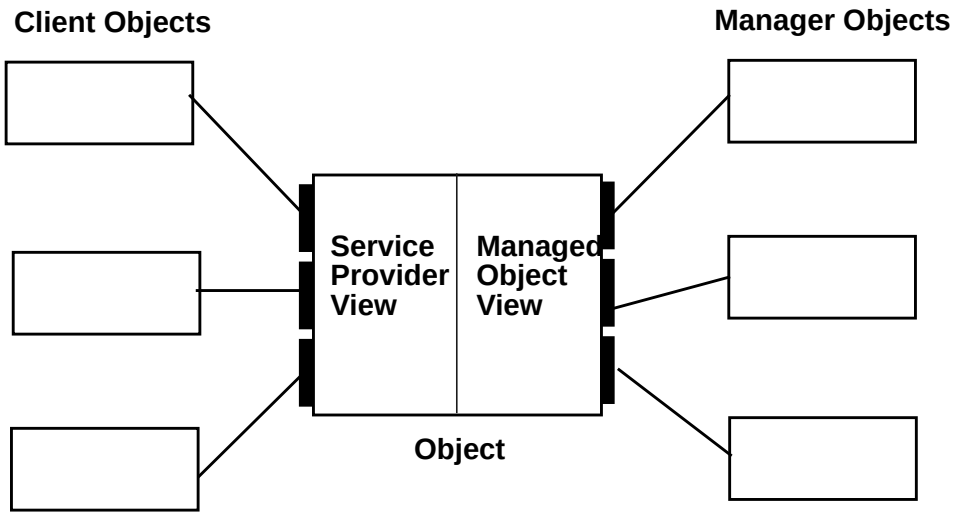


Figure 3-4. Service Interfaces and Management Interfaces

These two kinds of interfaces offered by an object provide, to other objects, two kinds of views on the object. The service interfaces provide a service provider view and the management interfaces provide a managed object view. (See Figure 3-4.) The client-server relationship that exists between objects when they interact via an interface is specialized into a manager-managed object relationship when objects interact via a management interface. The object that provides the management interface is called the managed object and the object that uses the management interface is called a manager.

It is important to note that the terms “manager” and “managed object” are relative to an interface and describe roles played by objects that interact via the interface. In general, an object may be a managed object with respect to its management interfaces while being a manager with respect to management interfaces of some other objects. It is also possible that the “core capabilities” provided by an object are capabilities for managing some other objects. Thus, invocation of an operation defined in a service interface of an object may result in that object invoking an operation defined in a management interface provided by another object. Thus, a service interaction in one level may cause management interactions elsewhere.

The different management interfaces provided by an object may provide different kinds of management functions. For example, one management interface may provide accounting meter control (start, stop, suspend, resume), another interface may provide operations for state changes (enable, disable), and yet another interface may provide operations for receiving notifications of threshold violations and state changes emitted by some managed objects.

Section 3.2 and Section 3.3 introduced the notions of operational interfaces and stream interfaces. A service interface may be either an operational interface or a stream interface. However, it is recommended that all management interfaces are specified as only operational interfaces. This is suggested so that management interactions occur in terms of operation invocations and responses, and not by means of low level abstractions such as

stream flows. A service interface that is an operational interface may provide operations that return as results references to management interfaces associated with the service interface.

Note again that the agent - manager structure separation described here is provided only as an example of a possible usage of multiple interface objects. It is not an architectural requirement.

4. Object Management and Binding Model

In a distributed application, objects may interact with each other in complex ways and engage in several activities. While many of these activities are application specific, there are certain generic management operations that can be performed on objects. This section briefly describes these generic management operations. Concepts related to establishment and control of interactions between computational objects are also described in this section.

4.1 Object Management

In general, the management operations described in the following subsections can be performed on objects during the lifetime of the objects. Note that this is only a general list. Depending on an application design, some of these operations may not be applicable to a specific object.

4.1.1 Creation and Deletion of Objects

Objects can be created. From the computational viewpoint, creation of an object is accomplished by instantiating an object template, as defined in Section 3.5.2. It should not be construed that a computational object template contains *all* information needed for creating an instance of the template. On the contrary, it contains only the relevant information required to specify object creation from the computational viewpoint. Additional information required for object creation is contained in some engineering templates (such as the cluster template, see [4] for details).

An object cannot create itself. To create an object, the object that needs the creation, *the instantiating object*, interacts with an entity that is capable of creating the object, the *instantiator*. The instantiator creates the object by instantiating the object template. At least one interface, the object's initial interface, is created when the object is created. A reference to the initial interface will be returned to the instantiating object via the instantiator. The initial interface can be used to initialize the newly instantiated object. The creation mechanism is illustrated in Figure 4-1. Note that this is an idealized depiction. In practice, the instantiator may be a programming language implementation itself (the creation process being a language type instantiation), or it may be a *factory object*. (The reader is referred to [4] for a discussion on factory objects that is related to this issue.) It is a matter of system design and implementation which entities are responsible for the creation of which objects.

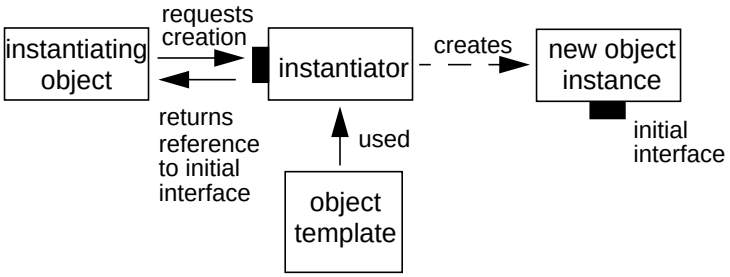


Figure 4-1. Schematic model of object creation

Of course, objects can also be deleted. When an object is deleted, all interfaces of the object are also deleted. Depending on the application design, the deletion operation may be provided by the object itself or by some other entity.

4.1.2 Creation and Deletion of Interfaces

From the computational viewpoint, interfaces are created by instantiating a computational interface template. As with objects, it should not be construed that a computational interface template contains *all* information needed for creating an instance of the template, but only the relevant information required to specify interface creation from the computational viewpoint.

The model for interface creation is similar to the one for object creation, see Figure 4-1. To create a new interface instance, the object that supports the interface, *the instantiating object*, interacts with an entity that is capable of creating the interface, the *instantiator*. The instantiator creates the interface by instantiating the interface template. The interface template must be one that is supported by the instantiating object, as described in Section 3.5.2. A reference to the new interface instance will be returned to the instantiating object from the instantiator. Note again that this is an idealized depiction. In practice, the instantiator may be a programming language implementation and/or the underlying processing environment (e.g. DPE kernel).

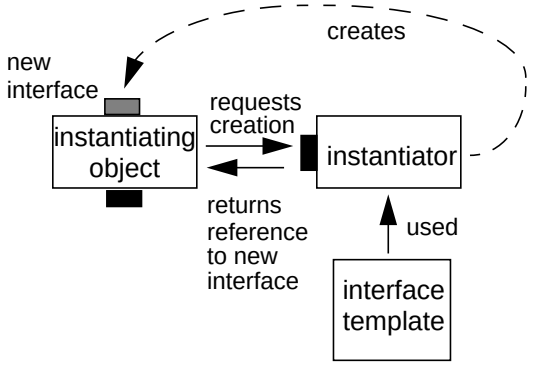


Figure 4-2. Schematic model of interface creation

After creation, the new interface is an offered interface of the instantiating object. Other objects that are potential clients of the new interface must obtain its reference either from the Trader service (if the interface is exported to the trader after being created), or by interacting with the instantiating object via another interface.

Interfaces can also be deleted. From the computational viewpoint, an interface is deleted by the object that supports it (which is the same as the instantiating object above).

4.1.3 Deactivation and Activation of Interfaces and Objects

Interface instances can be deactivated and activated. Deactivation means that no interaction is possible via that interface until the interface is activated. An object's initial interface cannot be deactivated. When an interface is activated the object is requested to resume interactions via the specified interface.

Objects can also be deactivated and activated. Deactivation of an objects means that all its interfaces except its initial interface are deactivated. When an object is activated, it will activate all of its interfaces that are currently deactivated.

Deactivation of objects and interfaces is useful e.g. to make sure that objects are kept in a consistent state during certain management operations, such as software upgrade. Deactivation of interfaces or objects can e.g. prevent “normal service usage” to take place during these management activities.

4.1.4 Migration of Objects

During its lifetime, an object may be moved from one location (or node) to another.

4.2 Interface Binding

4.2.1 Binding

Modelling concepts for describing the establishment and control of interaction sessions involving multiple objects is an important aspect of the interaction model. An interaction session involving multiple objects is called a **binding**. A binding may involve operational or stream interfaces. Bindings can be either explicit or implicit. Explicit binding is defined for both operational and stream interfaces, while implicit binding is defined only for operational interfaces. The requirement that stream interfaces must be explicitly bound is based on the view that the quality of service, timing, and synchronization requirements needed for stream flows require that network and computing resources be set up prior to start of the flows.

4.2.2 Implicit Binding

In some cases, an application-level control of operational bindings may not be needed. This is indeed the case in simple operational bindings where no new party is to be added or removed dynamically, where there is no need to distinguish between the binding establishment and the interaction, and where there is no negotiation of quality of service. Such bindings can be established implicitly by the infrastructure, and are called **implicit bindings**. Implicit bindings are not visible at the computational level. Only the interactions are visible.

Implicit binding takes place when a client object invokes an operation on an interface to which the client is not bound. The following steps are involved:

- The client is bound to the server interface. This function is a basic mechanism provided by the DPE and the underlying network (i.e the kernel transport network (kTN), see the engineering modelling concepts [3] for more details).
- The operation is invoked on the server interface.
- Optionally, the binding is deleted.

Note that there is no rule for when the implicit binding will be deleted. The binding may be deleted immediately after the completion of the operation invocation, or at some later point in time. The latter alternative will allow the same binding to be used for several interactions between the client and the server.

4.2.3 Explicit Binding

In some cases, controlling the binding is really essential for fulfilling the service requirements. This is e.g. always the case when binding stream interfaces. In such cases, the binding has to be established by an explicit request from a computational object. Such a binding is called an **explicit binding** and each such binding is modelled as a **binding object**. A binding object is an object that encapsulates the binding mechanisms and provides operations for controlling the binding (i.e., the session), including addition and removal of participants as well as modification of quality of service parameters. The binding object is an ordinary computational object.

The different entities involved in the creation and control of an explicit binding are shown in Figure 4-3. A binding object is an instance of a *binding object template*. Several different binding object templates may be available in a system. A binding object is instantiated as a result of an explicit request to a **binding factory**. Such an operation takes as arguments the required binding object template name, references to the initial set of the interfaces to be bound, and some other parameters. The operation returns an interface reference to an interface provided by the created binding object; this interface (generally called a control interface) then allows other computational objects to control the created binding. The binding factory is simply a computational object that is capable of creating one or more types of binding objects, where each type is defined by a specific binding object template. Note that the request to create a binding may come from one of the objects involved in the binding or from a third party.

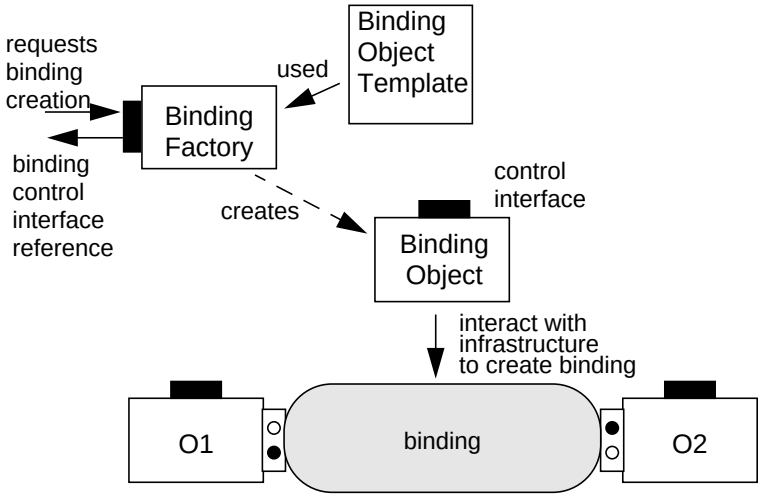


Figure 4-3. Creation of an explicit stream binding

An interface may be simultaneously involved in multiple bindings. A flow endpoint may however only be involved in one binding.

The following two subsections investigate in more detail the different characteristics of explicit stream and operational bindings.

4.2.4 Explicit Binding for Stream Interfaces

A stream binding may allow creation and control of simple or complex communication structures. The binding may involve to or more interfaces of the same or different types, possibly forming complex multi way bindings.

The behavior of binding objects reflect the communications semantics they support. The complexity of binding objects may range from a simple object only supporting point-to-point connection between two interfaces of the same type, to complex binding objects supporting multi way communication between interfaces of different types. The computational modelling concepts does not restrict the types of binding objects, however useful types of binding objects could be standardized. Some examples of useful binding types may be:

- A simple full duplex point-to-point binding between two interfaces of the same type.
- A binding type that encapsulates a conference system, allowing multipoint communication between interfaces that support different video and audio encoding, and offering a control interface that allows control of the conference
- A binding type that allows the combination of flows from different sources to a single sink, e.g. a video source from one interface and an audio source from another may be bound to a single “television” interface as image and associated commentary.

The behavior of a particular binding object depends on the object type, however we can identify some functions that will typically be performed, either by the binding object or the binding factory, during binding establishment. These are:

- Interface type compatibility checking and matching of flow endpoints: The operation used to request a new binding is parametrized by two or more interface references, one for each interface involved. The binding will only succeed if the interfaces are compatible and have types supported by the binding object. If the interfaces contain multiple flow endpoints, corresponding flow endpoints must be matched (e.g. matching video source in one interface with video sink in another).[1]
- QoS negotiation: Inclusion of an interface in a binding may involve negotiation of several quality of service parameters between the participants of the binding. Compatibility has to be ensured between the quality of service attributes of the interfaces involved and the supporting infrastructure capabilities.
- Allocation of resources: Communication paths for providing the connectivity between the participants of a binding needs to be established. Establishment of communication paths may involve allocation of both network and computing resources. If the interfaces involved in the binding are of different types, other resources, such as code converters may need to be localized and allocated.

Once the binding has ben established, it can be controlled through the control interface offered by the binding object. In the case of stream bindings, the control operations may e.g. include one or more of the following:

1. Depending on the interfaces offered by the binding factory and the binding object, matching of flow endpoints may also be specified explicitly by the client.

- Operations for adding and deleting parties to the binding. In general, the set of interfaces bound together in a single binding may vary over time; i.e., interfaces may be added and removed during the lifetime of the binding. When an interface is to be included in a binding, the existence and accessibility of the interface has to be verified.
- Operations for controlling the start and stop of the stream flows.
- An operation which requests notification of errors disrupting the binding. This operation takes as argument an interface reference indicating where the binding object should invoke a notification operation if failures disrupt the binding.
- Operations could be defined to manipulate specific QoS parameters associated with the binding. This form of control would be particularly useful in multimedia applications.
- An operation which requests notification of events of interest to the application which are detected by the infrastructure supporting the binding. For example, an event may be signalled at the start or end of a silence in an audio flow.
- An operation for deleting the binding. The effect of deleting a binding to the binding object is determined by the behavior of the binding object.

Note that in the case of stream interfaces, the binding object represents the control of the binding only, not the actual flow of data between the stream interfaces, which is considered to take place “outside” the binding object.

4.2.4.1 Example: Connection Session Manager

This example illustrates a form of binding that is supported by an application that is built on top of the DPE, see Figure 4-4. This application, called *communication session manager* (CSM), provides operations for the establishment, control, and release of communication sessions involving two or more stream interfaces (see [11] for a more detailed description of CSMs and other objects related to the TINA Connection Management Architecture). To create a communication session, a client object requests the CSM factory to instantiate a CSM. The CSM establishes the communication session using the services provided by network resource managers (such as connection managers) for the allocation of network resources, and the service provided by the DPE for the allocation of computing resources such as buffers. Upon establishment of the communication session, the CSM creates a computational interface that provides operations for controlling the communication session. The client invokes these operations, such as add party, remove party, and change quality of service parameters to control the session. Thus, in this case, the CSM plays two roles: one as the creator of bindings, and the other as the provider of operations for controlling the bindings that have been created.

4.2.5 Explicit Binding for Operational Interfaces

Explicit operational bindings allow the creation and control of more complex or specialized computational communication sessions than the implicit bindings. Examples of some useful explicit operational binding types are:

- A binding type that support group communication, allowing a client to invoke operations on a group of interfaces of the same type (i.e. a one-to-many communication structure).
- A binding type that allows the selection and monitoring of the underlying protocols used for the communication.

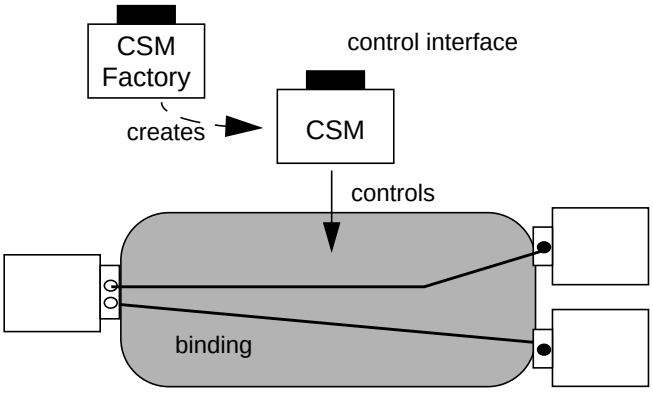


Figure 4-4. Explicit Binding Provided by CSM

Interface compatibility is a less important issue for operational bindings than for stream bindings, since interface compatibility is determined by generic rules (i.e. subtyping relationships). However the following functions may still have to be done during establishment of the binding:

- QoS negotiation, e.g. if there are requirements on the QoS needed from the underlying infrastructure.
- Allocation of resources, e.g. communication protocols and buffers.

It will often be necessary for the binding object to offer an invocation interface in addition to the control interface. The invocation interface will be used by the client to interact (i.e. invoke operations) through the binding. An example of such an invocation interface is shown in Figure 4-5.

A subset of the operations mentioned as typical control operations for stream bindings are also relevant for operational bindings (see Section 4.2.4), e.g.:

- Operations for adding and deleting interfaces to the binding.
- An operation which requests notification of errors disrupting the binding.
- Operations to manipulate specific QoS parameters associated with the binding, e.g. with the underlying communication protocol stack.
- An operation to delete the binding.

4.2.5.1 Example: Multicast Binding

A binding in this example is a multicast group[2] involving two or more operational interfaces (of the same type), see Figure 4-5. The binding object, called the *multicast manager*, provides two types of interfaces. It provides a control interface and one invocation interface for each group that it creates. The control interface provides operations for establishment, control, and release of multicast groups. The operation for group establishment accepts as an argument the list of interfaces that are the initial members of the group, and returns as result

2. Note that the concept of multicast group discussed here is unrelated to the concept of object group discussed in Section 5.

a multicast interface that is of the same type as the type of the group members. When a client invokes an operation on the invocation interface, the multicast manager invokes the same operation on each of the component interfaces of the group. The multicast manager collates the results of individual invocations and returns the collated result to the client. The control interface also provides operations for changing the group membership during the lifetime of the group.

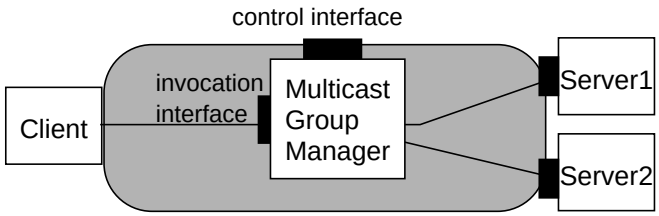


Figure 4-5. Explicit Binding for Group Communication

The form of binding illustrated in this example can be supported by the DPE or by an application on top of the DPE.

With reference to the Notification Server discussed in [3] and [4], note that the Notification Server provides a particular form of group communication. Specifically, the Notification Server supports dynamic multicast groups composed of notification interfaces.

4.2.6 Bindings Supported by DPE vs. Bindings Supported by Applications

From the computational perspective, the concept of explicit application level control of bindings by means of binding objects is rather powerful. It allows extensibility of the interaction model; i.e., applications can define complex forms of interactions between computational objects without requiring them to be built-in modelling concepts. This means that a system designer has the freedom to decide which forms of bindings (and binding templates) are to be supported by the DPE and which are to be supported by applications. It is important to note that even in the case of implicit bindings, establishment of these bindings involve objects such as binding factories. The difference is that in such cases, it is the DPE that (implicitly) requests instantiation of (conceptual) binding objects.

It is important to note also that, both in the case of implicit binding as well as explicit binding, the DPE does not deal with management of network resources (when network resources are necessary), but use the services of computational objects that are specifically designed to manage these resources.

5. Object Groups

5.1 Introduction

A service, application, or system, in a TINA environment consists of a collection of object instances typically distributed over multiple computing nodes. When designing a large TINA system, remaining at the abstraction level of objects becomes a limitation. The complexity becomes overwhelming. This limitation can be overcome by raising the abstraction level to deal with collections of objects as units in themselves (subsystems). For example, a distributed system comprising 100 interacting objects may be better designed[1] as, say five loosely interacting components of about 20 objects each.

Also, TINA systems will typically require systematic management: of configuration, in conditions of fault, of security mechanisms, and of accounting and performance information. It is expected that a single large and complex TINA system will be supervised by management systems operated by different organizations. In essence the implemented objects of TINA systems need to be managed. However, if the number of these objects is large, then there is a motivation to manage collections of objects together as units instead of isolated objects.

This section defines the concept of **object group**, which is intended to help cope with the design complexity and management issues noted above. An object group is a “software package”, or “subsystem”, that contains one or more objects. It is built, installed, maintained, and managed as a unit. Introducing this abstraction has the following benefits:

- It provides support for modularity. Object groups are encapsulated entities that only communicate with one another through special external interfaces (called **contracts**). The internal structure of an object group is hidden, and can be modified without affecting the operation of other groups.
- It provides a framework for flexible software management. An object group is intended to be as much as possible a self-managed entity. An external manager can also manage the object group as a single unit through a contract that acts as a single well-defined point of management access.

Note that the term “group” as used in this paper is an *application structuring* mechanism that is unrelated to the notions of *interface groups* and *group communication*, two terms associated with the *interaction models* of applications as frequently encountered in the distributed systems literature.

Object groups should not be confused with the notion of *domains* as used in TINA. A domain is a collection of resources, e.g. transport network resources, under the purview of a single control/management function. An object group is a collection of computational objects, and can be used to structure and control those control/management functions.

This section describes the basic properties of object groups as well as the operations necessary to manage them.

1. Better design in the sense of: Managing complexity, better reflecting the conceptual structure of the system, reflect distribution requirements, and so forth.

5.2 Object Group Structure

An object group is a collection of objects[2] that is built, installed, maintained, and managed as a unit. It can be thought of as a “subsystem” or “software package”. An object group is an encapsulated unit, in the sense that only a subset of the interfaces offered by the objects in the group are visible outside the group.

Interfaces that are visible outside the group are called contracts. Objects that are located in different groups interact only via contracts. Interfaces that are not contracts cannot be accessed by objects outside the group. An object group may offer one or more contracts.

All the objects located in a group may potentially invoke operations on contracts offered by other groups. The corresponding interface references could be obtained through the trader, or elsewhere.

One of the component objects of an object group is designated a **group manager**. The group manager is responsible for mediating management operations on the component entities of the group. At least one of the interfaces offered by the group manager must be a management contract for managing the object group itself as a whole and service contracts. The management contracts provide management functions such as creation and deletion of component entities, management of the state of component entities and mediation of notification emissions from the object group. The group manager is discussed in more detail in Section 5.4.

It is not an architectural requirement that group managers provide only operations that can be classified as “management” operations. The group manager can also usefully provide operations that allow clients to interact with multiple objects (i.e. members of the group) with a single operation.

A group can have other groups as components in addition to objects. There are a number of motivating reasons for having this feature. One view of a group is that it is akin to a management domain. Having hierarchical management domains lends support to the notion of having hierarchical groups. Further, from the point of view of application structuring, nesting of groups allows an existing group to be reused as a component within the design of a new group. Nesting of object groups also allows program structure to better reflect the conceptual domain of the application. For example, if some of the objects in a group need to be located remotely from the others, but together, “packaging” them in a separate autonomous unit (a subgroup) provides support for this conceptual structure. Finally, hierarchical object groups require no additional support from the Distributed Processing Environment than is needed for non-hierarchical object groups.

The components of an object group are potentially distributed, in the sense that component groups *can* be located remotely from the enclosing group. The component groups may be distributed amongst different *capsules*[3] on a given computing node, or amongst different capsules on different computing nodes. Component *objects* (including the manager object) of an object group are always located within the same capsule.

2. And “subgroups”, see below.
3. A capsule [3] can be thought of as a Unix like process. A capsule owns storage and a share of the node’s processing resources. A capsule is a unit of management, control and protection with respect to the operating system of the node. It is generally also the smallest unit of independent failure supported by the operating system.

Since, in general, object groups encapsulate data and behavior and provide operations at interfaces (i.e. contracts) for manipulation of that data and behavior, it could be tempting to regard groups as a kind of (composite) objects. However, groups are *not* objects, for the following reasons:

- groups have an internal structure that in some cases is visible, e.g. to a manager that has the appropriate authorizations. A computational object has no visible internal structure. It is not possible to “peek” inside a computational object, and
- a groups instance may be distributed, while an object instance is a unit of distribution.

Even though there is a distinction between groups and objects, it is expected that their encapsulation properties allows them to be “easily” substituted for each other during some phases of system design.

An example of a group containing two component objects and one component group is shown in Figure 5-1.

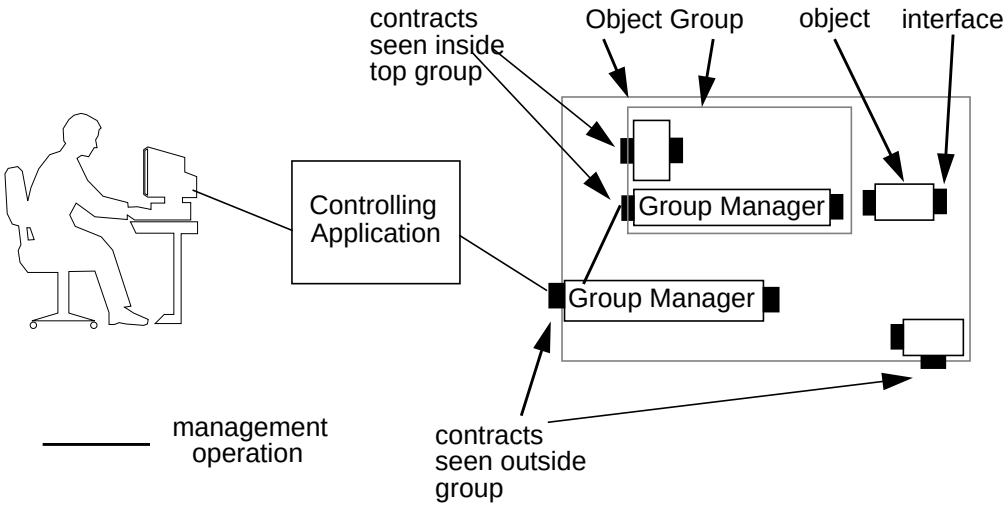


Figure 5-1. Illustrative diagram of object group instances, and their relationship to a user and a controlling application.

5.3 Contracts

Any type of interface can be a contract. However, contracts have additional constraints placed on them when compared to the internal interfaces of a group. For example, contracts can be subject to supporting release independence (backwards compatibility). Contracts may also be a natural point to enforce security constraints.[4]

The release independence constraint requires that if an object group is upgraded, resulting in changes in a particular contract, the upgraded object group must contain objects that offer contracts that are compatible with the old version of the contract. Release independence is imposed to facilitate evolutionary growth of distributed applications; when an object group

4. A similar notion of contracts exists in Bellcore’s Information Networking Architecture [17] [18].

is upgraded, resulting in changes to interfaces used by objects outside of the package, the latter objects do not have to be upgraded concurrently. The period for which such backwards compatibility is maintained by an object group is an enterprise level policy issue that may be different for different contracts of the group. Since only contracts are visible outside a group, the internal structure and interfaces of a group may be modified without affecting other groups.

Object groups may be considered units of security. In this case, contracts are natural points to enforce security checks, including authentication and access control checks. The granularity of access control is a design issue. The access control policies may be different for different contracts of an object group. Interactions between objects within the object group are not subject to security checks. Furthermore, all objects within an object group are considered to belong to the same security principal. That is, the object group has an identity that can be authenticated, and all objects within the group have the same set of access privileges.

As mentioned above, an object group must offer at least one *management contract* for managing the objects within the group, either individually or in aggregation. The granularity of management, and the management operations provided is a design issue and an aspect of group manager behavior. The group manager is discussed further in Section 5.4.

Since the interface references of interfaces inside a group are not intended to be used outside the group, they will normally not be exported to the open system environment (i.e.to the “public” trader or name server). The references to the contracts typically would be exported to the trader to be made available for binding by external components not known a priori.

5.4 Group Managers

Each group is associated with a group manager object. The group manager is always the first object to be instantiated in a group. Other objects may be created and deleted dynamically during the lifetime if the group, but not the group manager. Via a group manager, a group is expected to be, as much as possible, an autonomous or self managing entity.

In practice, a group manager is expected to embody concerns from a number of viewpoints [20]. These include the enterprise viewpoint (such as enforcing access, as well as other, policies), computational viewpoint (such as knowing details of the group and object types supported), and engineering viewpoint (such as knowing which node it is currently executing on, and other issues of distribution and execution). Having one entity embody multiple viewpoints is not seen as problematic, but care needs to be exercised to avoid misleading ambiguity when discussing group manager issues.

The group manager plays a central role in the activity of a group. It is responsible for mediating control/management activity between entities external to the group and objects and groups inside its group. As mentioned above, the group manager always offers at least one contract that can be used to manage/control the group. A typical example of this control activity occurs when an object external to the group (e.g. a controlling application) requests the creation of a new object in the group, as depicted in Figure 5-2[5].

5. The means of instantiation of component entities, in an engineering sense, depends upon where they are to be instantiated, see Section 5.6.2.

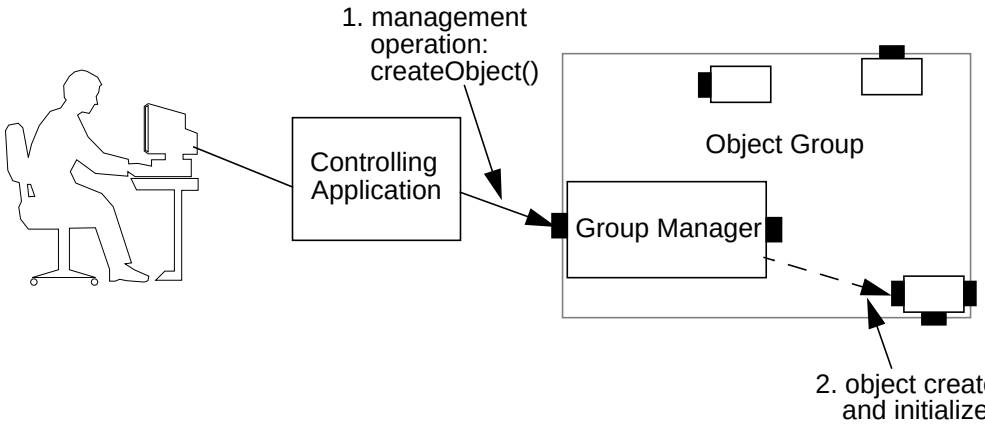


Figure 5-2. Human operator creating an object in a group via interaction with a group manager.

A group manager can also be a natural point of co-ordination for entities within the group. A typical example of this co-ordination activity occurs when an object inside an object group requests the creation of a new object, as depicted in Figure 5-2. In the illustrated example, the group manager first creates the new object instance, and then emits an event notification through some notification service. The event is forwarded through the notification service to controlling applications that need to keep an updated state of the object group.

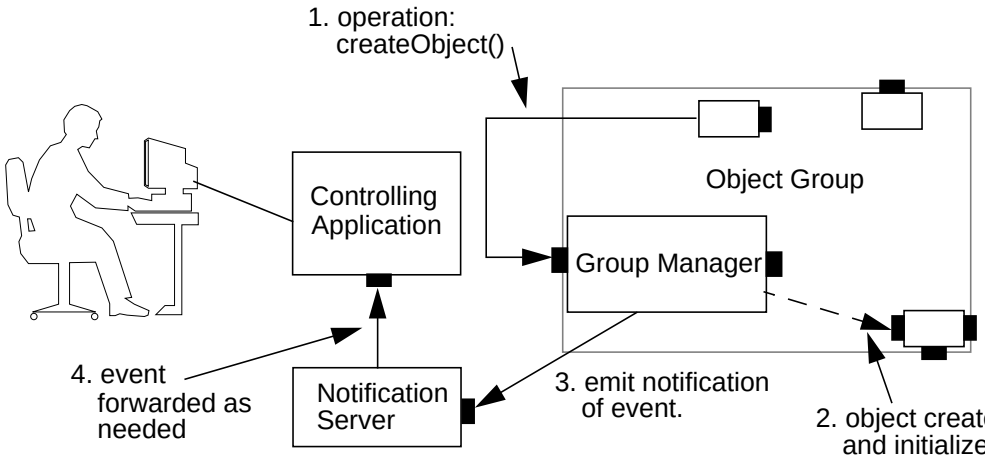


Figure 5-3. An object in a group creating another object via interaction with a group manager.

5.5 Object Group Declaration

5.5.1 Types and Instances

A **computational object group type** (or **object group type** or **group type**) is a template for a class (or collection) of object group instances, and each member of this class is considered an instance of that type. Every group is an instance of some group type. The essential components of a group type are its component object types and group types. For a specific group type, its component types cannot change dynamically.

An object group instance (or group instance) is an individual entity corresponding to an object group type. During the lifetime of an object group instance the number of component instances may vary. Typically, an initial configuration of component objects (at least the manager object) will be established upon creation of a group. Later, the initial configuration may be altered.

Interface instances of types that are identified as contracts of a particular group are in principle available to objects outside the group. Of course, a given group implementation may still choose to prevent individual instances of contract interface instances to become externally available e.g. by not passing the interface reference to any objects outside the group, including the trader.

5.5.2 Object Group Template

An object group template is a syntactic representation of a computational object group type. It specifies the following information:

- Identification of the object templates *and/or* group templates that compose the object group template. Each such object or group template is called a *supported object template*.
- Identification of a subset of the supported interface templates of the supported object templates as being *contracts*.[6]
- Identification of one of the supported object templates as the *group manager object template*. When this group template is instantiated, an instance of the group manager object template is created, and a reference to the initial interface of the group manager object is returned. The initial interface of the group management object defines operations for managing instances of object templates (and interface templates) supported by the group template. These management operations typically include creation and deletion operations, see Section 5.6.
- The behavior of the group, including semantics of operations and relationships and dependencies between objects in the group (e.g. constraints on the sequence for creation of object in the group.

The concrete syntax (i.e. TINA ODL) supporting the definition of group templates is defined in [5].

6. Currently, no template is specified for contract definition, see Section 6.

5.6 Object Group Management

This section describes a number of generic management operations that can be performed on groups during the lifetime of the groups. Note that this is only a general list. Depending on application design, some of these operations may not be applicable to specific groups.

Some groups may implement other management operations than the ones mentioned here. In principle, it may be relevant for the group manager to support functionality encompassing all the management functional areas: Fault, configuration, accounting, performance, and security (FCAPS). Currently all the operations mentioned in the following subsections fall within the category of configuration management.

5.6.1 Creation and Deletion of Groups

From the computational viewpoint, creation of a group is accomplished by instantiating a group template as described in Section 5.5. Initially, when a group type is instantiated, an instance of its group manager is instantiated, and no other entities of the group.[7] A reference to the group manager's initial interface is returned to the group's creator. After the group manager is instantiated, the initial entities of the group are instantiated. The process of establishing this initial configuration is an aspect of group manager behavior. The group manager may immediately instantiate the initial components, or wait for the invocation of some initialization operation. The initial configuration itself may be established in a number of ways: explicitly coded in the group manager implementation; explicitly stated in a well defined configuration language external to the application code (say in a file known by the group manager instantiator, or as a data structure known by, or communicated to, the group manager instantiator); or presented by some other means.

Here we do not make any decisions about how the initial configuration of groups is implemented, other than assuming that each instance of a group type is capable of having a different initial configuration. In addition to an initial configuration, each group is subject to constraints that may limit its possible subsequent configurations.

The group deletion operation deletes all objects in the object group, including the object manager. If the object group contains subgroups, they will be recursively deleted.

5.6.2 Creation and Deletion of Component Objects and Groups

Group managers support the creation of component objects or groups within the group it is responsible for. The means of instantiation of component entities, in an engineering sense, depends upon where they are to be instantiated. For the implementation of groups we assume that component objects of a group (including the group manager) must exist in the same capsule as the group manager, while component object groups can exist in other capsules. In this case, object instantiation can be achieved by programming language type instantiation. If a subgroup needs to be instantiated in the same capsule as the group, then programming language instantiation will suffice also. If a subgroup needs to be instantiated in a different existing capsule, then the *capsule manager* (see [4]) of the remote capsule is requested by the group manger to create the subgroup. If a subgroup needs to be instantiated in a different new capsule, then an entity capable of creating capsules (e.g. *a factory)*

7. Group creation can therefore be considered as similar to object creation, see Section 4.1.1.

needs to be contacted first by the group manager to create the capsule, then activity can proceed as if creating a new subgroup in an existing capsule. See also the discussion on object creation in Section 4.1.1.

In some cases it may be appropriate for the group manager to return an interface reference to one of the interfaces of the created object (i.e. a contract interface). In other cases the new object will only be referred to by a name. The group manager is responsible for ensuring that the names of all its components are unique.

5.6.3 Creation and deletion of contracts

A group manager will generally offer operations to create and delete contract instances in its group. The group manager will perform these operation by forwarding the requests to the objects offering the contracts in question.

5.6.4 Activation and Deactivation of Contracts and Groups

Contracts can be deactivated and activated. When a contract is deactivated, no interaction is possible via that contract until the contract is activated. A group manager will generally offer operations to deactivate and activate contracts in its group. The group manager will perform these operations by forwarding the requests to the objects offering the contracts in question.

Operations may also be supported to deactivate and activate individual objects within the group (affects all the contracts of an object), or the group itself (affects all the contracts of the group, except the management interface).

5.6.5 Group Migration

During its lifetime, an object group may be moved from one location (or node) to another. Migrating an object group, relocates all the component objects (including the group manager), but not component groups. The group manager may also support operations to migrate subgroups.

5.7 Example

We will consider here a multi-party video conferencing service. The TINA Service Architecture [16] describes concepts and principles for structuring services. The Service Session model is a part of this architecture that models service session configurations and is valid for a broad range of multimedia services. We will consider here, as an example of grouping, the components of the Service Session model as used to form a multi-party video conferencing service session. The key entities of the Service Session model are the *Service Session Manager* (SSM) and the *User Session Manager* (USM). Typically, a user of a service invokes operations on their assigned USM (such as add-party to a call), which in turn invokes operations on the associated SSM (say, to create state information in the SSM, and create a new USM for the party to be added). The SSM maintains state and behavior needed by all the parties of a communication session. The further description of these entities is somewhat involved, and the reader is advised to refer to Figure 5-6 as they read the following paragraphs.

The SSM coordinates and serves all the requests from the members of the session. The SSM groups several components each offering different types of interfaces. The first one is the GSC (global session control) which offers a generic *session control* interface providing

operations for both connectivity and control relationship requests. It allows users to join or leave the service session (e.g. a video conference session). The GSC coordinates the behavior of the service session. Operations to suspend and resume involvement in a service session are also provided. The second object in the SSM is the GSP (global service part) which offers an interface providing *service specific* operations.

The second composite entity, USM, represents a particular user in a session. The USM groups two objects each offering a different type of interface. The first one is the USC (user session control) which provides the user with a *generic* session control interface (common to all services). Through the generic session control interface offered by the USC in the USM a user can add and delete other session members. The second object is the USP (user service part) which provides a *service specific* interface.

In addition to the Service Session model entities, the examples below involve the Service Factory (SF) entities. These entities are responsible for instantiating Service Session Managers. Typically, when starting a video conference service session, a user sends a message to a SF indicating that they want to initiate a service session manager capable of handling the a video conference service session. The SF initiates creation of a SSM, which in turn creates a USM for the originating part of the service/call. When a potential new party of the service is invited to join the service session, the SSM initiates creation of a USM for the new party.

We will assume that the Service Factory (SF) for this service is in one group, and that Service Session Managers (SSM) and associated User Session Managers (USM) are in a separate hierarchical group to the SF. This supports the view that the software for each service session or call instance is seen as one manageable entity. This is shown diagrammatically in Figure 5-6 where there is one SF group and one SSM group supporting three USM groups (indicating there are three parties participating in the conference). The primary ra-

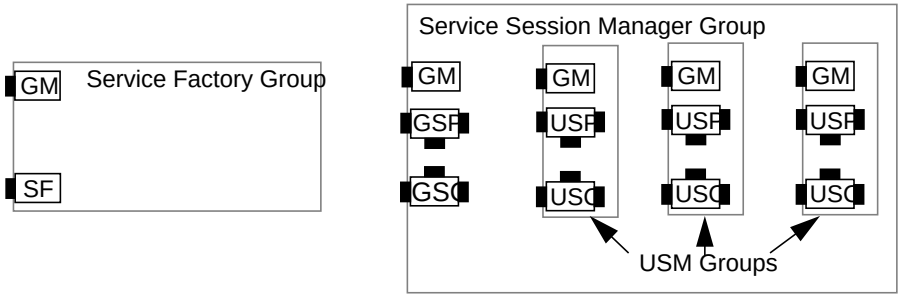


Figure 5-4. One possible grouping configuration of session control entities.

tionale for structuring the SSM in this manner is the lifetime dependencies of the objects involved. All of a USM's component entities are created and destroyed together as parties are added and deleted to and from a service session. If they are in a group the group can be created and destroyed as a unit. Similarly, SSM entities are created and destroyed as a unit as services are invoked and exited.

To help illustrate the advantages of fine grained software management provided by groups and group managers, we will consider the situation where a new video conference service request is made to a service factory, and an add-party request is made to the SSM[8]. For

simplicity of diagramming we will assume that the video conference SF group and SSM groups are all located in one “video conference” capsule. Extending the example to cover a number of capsules is trivial, but diagrammatically complex.

A sequence of activities associated with initiating a new service session is depicted in Figure 5-6 and a sequence of activities associated with an add-party request is shown in Figure 5-6. The reader should spend a moment to examine these figures. In Figure 5-6, a user requests use of a service at step 1. The service factory then uses the capsule manger to create a new SSM group at step 2. Creation of the SSM group involves creation of a new group manger at step 3. The group manager then creates its initial component entities, a GSP object, a GSC object and a USM group (manager) in steps 6 through 7. The USM group manager then creates its initial component entities in steps 8 and 9. The activity in Figure 5-6 is similarly understood. A point to note is that the (SSM) group manager is responsible for creating entities within the group. In both cases it should be remembered that the groups are part of a number of larger (conceptual) systems, such as a system of services, a managed/controlled system, and so forth. This is highlighted in Figure 5-6, where step 4 involves notifying external systems of what is occurring within the video conference service system.

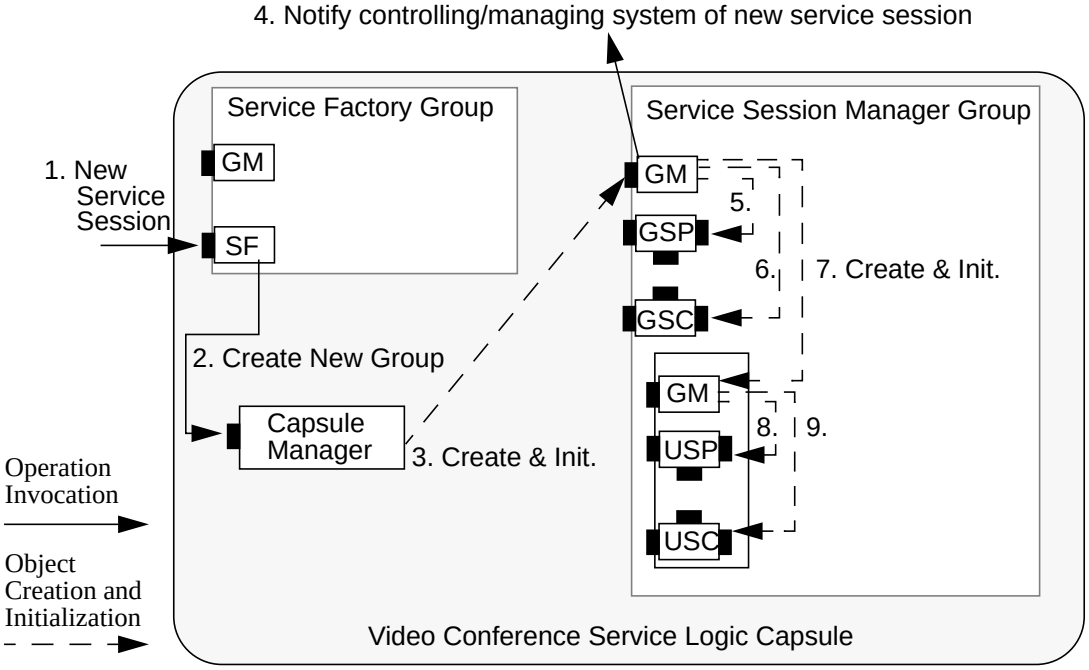


Figure 5-5. Processing a “new video conference service session” request.

With regard to Figure 5-6 and Figure 5-6, a controlling or managing system can have fine grain control over the system software through the presence of group managers (having identical interfaces), and the presence of an event forwarding infrastructure. Potential use of this control capability includes:

8. Note, even though the SSM is an aggregation of components, the notions of encapsulation and contracts support treating a SSM instance as a single entity.

a. Investigation of system configuration and state in cases of fault. For example, a Customer Service Representative may receive a complaint that the voice connection of a party is not present. The Customer Service Representative can investigate the state of entities in the SSM group to help ascertain the basis of the fault.

b. Manage configuration and state. For example, a customer may experience a fault in that they cannot exit a service session. A call to the Customer Service Representative could prompt an investigation of the appropriate group to discover that the group is in an inactive state. Depending upon the wishes of the customer the group can be reactivated, or deleted.

c. Maintain a view of the activity of the distributed system. For example, the event forwarding infrastructure can be used to maintain a view of the activity and distribution of the detailed state of service software.

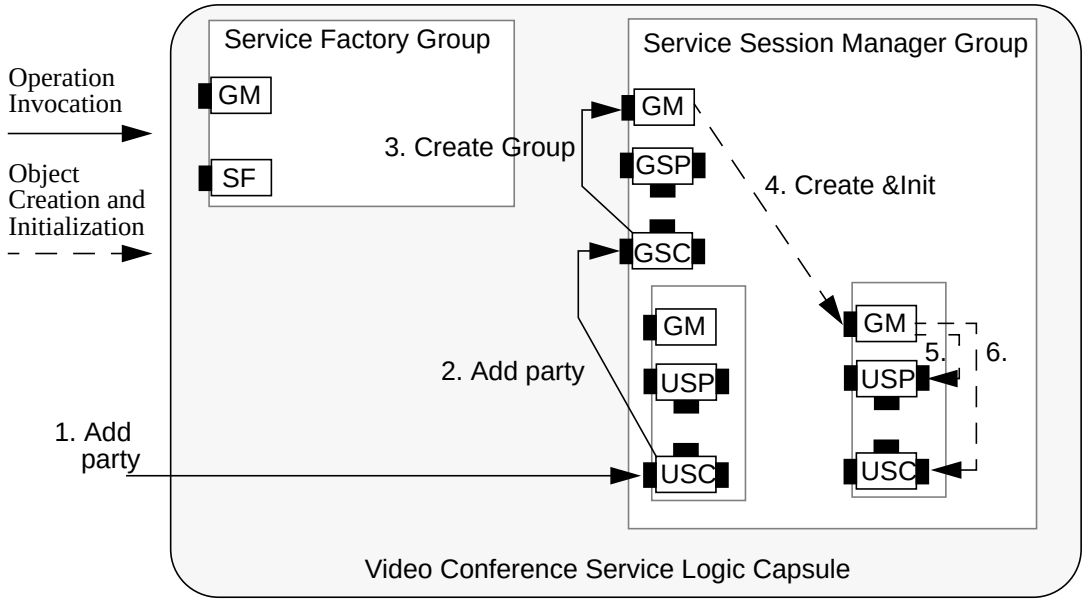


Figure 5-6. Processing an “add party” request.

6. Further Work

There are several major issues that need to be investigated further. These are summarized below. (Many minor issues have been identified in the individual sections.)

- Several details related to stream interfaces and stream flows need to be worked out. The specification concepts needed for describing stream flow signatures and synchronization constraints on stream flows are issues for further investigation. The relationship between stream interfaces and explicit binding needs to be understood better (see [7] in particular).
- The current compatibility rule for interface types requires equivalence of quality of service (QoS) attribute types. Relaxing this to allow compatibility of QoS attribute types is a problem of further study. This may require development of a *QoS architecture*, including more general specification structure for QoS attributes.
- The adaptation of the transaction model described in Appendix A to the requirements of telecom applications needs to be assessed. In particular, the feasibility of its support by a DPE needs to be established. A revised version of the transaction model is expected for the next version of this document.
- Work is needed on integrating security principles with the computational modelling concepts. This work should identify concepts and structures needed for specifying security aspects in computational specifications. It should also enable identification of computational objects that are specifically needed for ensuring security in interactions among computational objects.
- Several aspects related to object groups need further study:
  - Currently an object can only belong to a single group. It has been suggested that it is too restrictive to limit group composition to hierarchies. Instead, groups should be graphs, whereby an object may belong to more than one group. Further study is to clarify whether such a feature is needed and can be supported.
  - A separate template for contracts may be useful, to allow specifications of features that are not specified in interface templates, i.e. release independence policy, security policy, etc.
  - More work is needed to detail how all of the FCAPS management areas are supported by the group manager.

7. Acknowledgements

The editor want to thank Ennio Grasso from CSELT. The content of Appendix A is based on a proposal from him.

The editor of the current version of the document would like to thank all the people who gave comments and suggestions to improve this document, especially the reviewers. In alphabetical order these people are: Martin Chapman (BT), Heine Christensen (TeleDanmark), Man San Chung (Korea Telecom), Dennis Doherty (AT&T), Nigel Edwards (HP), Patric Farley (BT), Ennio Grasso (CSELT), Mikael Jørgensen (TeleDanmark), Yan LePetit (France Telecom), Osamu Miyagishi (NTT), Narayanan Natarajan (Bellcore), Bertil Nilsson (Ericsson), Ajeet Parhar (Telstra), Stephane Pensivy (France Telecom), Victor Perebaskine (France Telecom), Valere Robin (France Telecom), Victoria van der Meulen (KPN), and Art van Halteren (KPN).

In addition, the authors want to thank all members of the TINA-C Core Team for their contributions to the document. We particularly acknowledge the contributions of the authors and editors of earlier versions of this document. They are Fabrice Dupuy, Narayanan Natarajan, and Nikolaus Singer, Heine Christensen, and Nicolas Mercouroff.

Tom Handegård
Telenor
Norway

Appendix A: TINA Transaction Model

Editor's note: *This appendix has not been updated since the previous version of the document (version 3.1) was released, even though comments have been received on the proposed transaction model. Further considerations and discussions are needed before a revised version of the transaction model can be provided. A revised version is expected for the next version of the computational modelling concepts.*

This section has been prepared based on last year's version of TINA transaction model and inputs from Ennio Grasso (MODA project in CSELT). A list of issues related to this object model can be found in [10].

A.1 Transaction Support

A **transaction** is an atomic unit of consistent and reliable computing activity composed of a sequence of operations involving one or more objects. Traditional transactions have the following properties collectively referred to as **ACID properties** [44]:

- *Atomicity*: either the activity completes successfully or it is as if it never commenced;
- *Consistency*: upon the completion of the transaction, all objects that participated in the transaction are in a consistent state; (consistency is defined by a set of invariants in the specification of objects)
- *Isolation*: the effects of (i.e., object state changes made by) a transaction are invisible to other activities until the transaction completes;
- *Durability*: the effects of a transaction persist after the successful completion of the transaction.

Among the distribution transparencies identified in RM-ODP, **Transaction Transparency** masks the coordination of operations that have the ACID properties. It masks the mechanisms for rolling back any operation that has failed, allowing concurrent operations on a shared state, ensuring consistency within the invoked object, and making the results of committed operations durable.

The current version of TINA Transaction Model is tailored for offering transaction transparency to the developer of computational specifications. It is therefore assumed that the application developer declares, in the object or interface template, that an interrogation is transactional.

As for the other distribution transparencies, it is assumed that the environment provides support to this mechanism by ensuring the coordination of the action of the object with a transaction coordinator, according to the transaction model defined. This would be transparent to the programmer, who does not have to invoke any calls to the transaction manager in the implementation code (in C++ for example).

A.2 Transaction Modelling Concepts

A **transaction model** is characterized by the *transaction structure* and the correctness criterion of the allowed transactions [42], [44].

- The transaction structure specifies the *structural dependencies* between related transactions; complex transactions consist of a set of consistent transactions and a set of constraints on the execution structure of those constituent transactions.
- The correctness criterion determines the *dynamic dependencies* between transactions, i.e., the histories that result from the concurrent execution of a set of transactions which does not violate the consistency of the objects accessed.

In the transaction model proposed here, initiation of transactions and inclusion of actions within the scope (or umbrella) of a transaction are all specified on the basis of some tags associated with operations and the invocation of such operations. For the purpose of the transaction model, an operation provided by an object may be tagged as being one of the following kinds:

- **Transaction initiation operation**
- **Transaction join operation**

Only an interrogation operation can be tagged as a transaction initiation operation or a transaction join operation.

A transaction is initiated in the following manner. When an object invokes a transaction initiation operation, the invocation triggers initiation of a transaction. The completion of the invoked operation signifies completion of the transaction. The operation may complete either normally or abnormally. A normal completion signifies that the transaction has **committed**, i.e., all its effects are durable. An abnormal completion signifies that the transaction has **aborted**, i.e., all its effects have been undone and it is as if the operation was not invoked. The signature of a transaction initiation operation specifies at least two terminations, one that signifies a normal completion, and the other an abnormal completion.

During the execution of a transaction initiation operation, an object may invoke operations provided by other objects. Depending on the tag associated with an operation, the semantics of the invocation varies as described below:

- If the invoked operation is a transaction initiation operation, the invocation triggers initiation of a **subtransaction** or **nested transaction**. The notion of subtransaction is explained further later in this section.
- If the invoked operation is a transaction join operation, the processing of the invocation is viewed as a part of the transaction in which the invocation has occurred. Thus, the scope of the transaction is expanded to include the invocation processing. Normal completion of a transaction join operation means that the effects of the invocation processing are temporary until the transaction commits later. If the transaction subsequently aborts, the effects of the invocation processing are undone or discarded. Abnormal completion of a join operation means that the transaction is aborted. This implies that the invoker operation must also abort.
- If the invoked operation is a non-transactional operation, the processing of the invocation is outside the scope of the transaction. That is, the subsequent transaction commit or abort will have no impact on the invocation processing.

A.3 Synchronization Requirements

The basic approach for specifying correctness criteria is based on the definition of **conflicting operations**.

Definition: Two transaction initiation or join operations A and B of an object (A and B need not be distinct) **conflict** with each other if starting from some initial state of the object, the effect of executing A after B differs from executing B after A in at least one of the following ways:[1]

1. The final state is different in the two cases
2. The result parameters returned by A are different in the two cases
3. The result parameters returned by B are different in the two cases.

The traditional operations that transactions execute are sequences of *Read* and *Write* operations. It is well known that a *Write* operation conflicts with other *Read* and *Write* operations on the same object. The table derived from the conflict relationship between Read and Write is shown below:

Table 7-1. Conflict Relationships for Read and Write operations

	Write	Read
Write	Yes	Yes
Read	Yes	No

In general, object operations are not simply *Read* and *Write* but may be more abstract, such as *Increment* and *Read* for a counter object. If the semantics of object operations is taken into account it may be possible to permit much more concurrency so that waits will be caused by conflicts based on the application semantics and will occur less frequently. For example, two *Increment* operations on the same object can be admitted concurrently:

Table 7-2. Conflict Relationships for Increment and Read operations

	Increment	Read
Increment	No	Yes
Read	Yes	No

The specification of conflicting operations is the basic approach toward the definition of several correctness criteria. The most common correctness criterion for concurrent transactions is *serializability.*

Serializability requires that any history of concurrent execution of a set of transactions be equivalent to a serial execution represented by a serial history. An interleaved execution of transactions will be correct if there is a possible serial execution of the same set of transactions such that the order of conflicting operations is the same in both execution. More precisely, consider two concurrent root transactions T1 and T2 corresponding to two different transaction trees. By concurrent, it is meant that both transactions have been initiated, but

1. Since objects encapsulate state, it does not make sense to define conflict relationships between operations on different objects.

neither is yet to commit or abort. Suppose a transaction initiation or join operation A has been executed within one transaction, say T1, prior to the execution, within T2, of a transaction initiation or join operation that conflicts with A. Then, T2 is serialization dependent on T1. A concurrent execution of T1 and T2 is serializable if and only if the serialization dependency induced by the execution is acyclic; i.e., it should not be the case that some conflicting operations are performed in T1 prior to T2 and some conflicting operations are performed in T2 prior to T1. This serializability requirement is extended in a natural manner to cover a set of concurrent transactions, T1, T2,..., Tn. In this case, the concurrent execution is serializable if and only if the serialization dependency among the transactions is acyclic.

A.4 Nested Transactions

As discussed in the previous section, a nested transaction is initiated when a transaction initiation operation is invoked during the execution of a transaction. Invocation of transaction initiation operations may be nested giving rise to a tree structure (or hierarchy) for transactions [46]. This is illustrated in Figure 7-1.

Figure 7-1 illustrates the hierarchy for some particular transaction T. Each node in the tree corresponds to a transaction initiation operation that has been invoked during the execution of T. The transaction T was initiated by invoking an transaction initiation operation A defined in the interface I_1. During the execution of T, another transaction initiation operation B defined in the interface I2 was invoked initiating a subtransaction T_1. Note that this invocation may occur anywhere within the scope of T, i.e, either during the execution of A or during the execution of a transaction join operation invoked within the scope of T. Within the subtransaction T_1, a subtransaction T_2 was initiated, and upon completion of T_2, another subtransaction T_3 was initiated. Upon completion of T_1, a subtransaction T_4 was initiated within T.

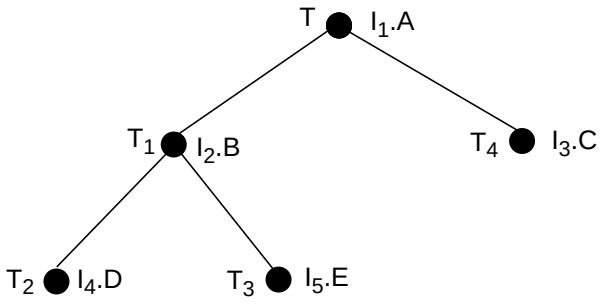


Figure 7-1. A Transaction Tree

With reference to a transaction tree, the standard terminology associated with tree structures, such as root, parent, child, descendant, and ancestor are used in this transaction model. The activity denoted by an entire transaction tree is referred to as the **root transaction**. Since there is a one-to-one correspondence between nodes in the tree and subtransactions, the notions parent, child, ancestor, and descendant are used to relate subtransactions in a natural manner.

The tree structure induces the following structural dependencies between related transactions:

- begin-dependency:

  a subtransaction must start after its parent transaction and complete before it;

- commit-dependency:

  the commit of a subtransaction is conditionally subject to the commit of its parent and all its ancestors up to the root transaction. Hence, a nested transaction can finally commit only if the root transaction commits.

- abort-dependency:

  if a (sub) transaction, at any nesting level, abort, the whole subtree of subtransactions whose root is the failed (sub) transaction is also aborted, whether or not some of the subtransactions have performed their local commit. Conversely, if a subtransaction fails, its parent is not required to abort but can perform its own recovery by:

  - ignoring the failure;
  - retrying the execution of the child transaction;
  - attempting to execute an alternative subtransaction.

  The abort-dependency provides a powerful mechanism for fine-tuning the scope of abortion. In contrast to flat transactions where a failure necessary undoes the transaction to its start point, nested transactions allow the rollback at the granularity of a single subtransaction. This limits the damage to a smaller part, making it less costly to recover.

The nested transaction model described here guarantees full ACID properties at the root level. To achieve global serializability, a subtransaction transfers all the serialization dependencies raised during its execution to its parent at commit. This **transfer rule** is mandatory as a committed nested transaction may later be rolled back should one of its ancestors abort.

Consider two concurrent subtransactions S1 and S2 each belonging to a different transaction tree and suppose that S2 is serialization dependant on S1. Let S be the parent of S1 9this may be the root transaction or another subtransaction). When S1 commits, S2 will become serialization dependent on S. The transferral of the serialization dependencies continues recursively up to the root transaction.[2]

Execution of concurrent subtransactions within a transaction tree must also be serializable. In this case however, the transferral of the serialization dependency of S2 on S1 continues up to their least common ancestor.

A.5 Open Nested Transactions

Although the requirements of full isolation, atomic commitment, and global serializability may be mandatory for traditional database applications, in complex domains, such as telecom applications, the need to selectively relax ACID properties according to application semantics and improve concurrency becomes a crucial issue. For example, a transaction

2. Note that the transfer rule only applies to committed subtransactions. Indeed, if S1 aborts the dependency of S2 on S1 will be relinquished.

connection set up activity may need a great amount of time if compared to a simple database operation. In this case global serializability may be a too severe consistency criterion because it greatly reduces concurrency.

The transaction model proposed here allows the specification of transaction join operations and transaction initiation operations with *closed*[3] and *open* semantics in any arbitrary combination.

Open nested transactions are a generalization of *sagas* [43] and *multilevel transactions* [47], [48], [41]. Open nested transactions maintain a tree structure and thereby retain the *begin*, *commit*, and *abort* structural dependencies between related transactions; yet, they relax *Isolation* by allowing the effects of committed subtransactions to be visible to other concurrent transactions. In fact, open nested transactions do not follow the transfer rule and guarantee a weaker form of serializability called *quasi-serializability*. Again, consider two concurrent **open** subtransactions S1 and S2 and suppose that S2 is *serialization dependent* on S1[4]. This time, when S1 completes, the dependency of S2 on S1 will be relinquished immediately.

Since the serialization dependencies on an open subtransaction are relinquished when it commits, the structural abort-dependency induced by a parent transaction on its children cannot rely on traditional rollback: rolling back an open subtransaction may unintentionally disrupt the effects of a second possibly interleaved transaction. Hence, abort recovery of an open subtransaction is performed by means of a *compensating transaction* that semantically reverses the effects of the child transaction that has committed. This relaxes *Atomicity* since a compensating transaction need not necessarily undo all effects of the transaction compensated for, but can result in a semantic specific repair action.

For example, suppose that a transaction operation inserts a key into a B-tree index causing a page-split where the index is stored. The compensation of this operation can simply remove the inserted key without restoring the page-split. A formal approach to recovery by compensating transactions can be found in [45].

Since open nested transactions use compensation for abort recovery, the definition of a transaction initiation operation with open semantics, say A, must include the specification of a second transaction initiation operation, say A^{-1}, that must be executed if A needs to be compensated.[5] The inverse operation is simply one transaction operation provided by the object and, as such, can also be offered in an interface as a normal operation. The input parameters for the invocation of the inverse operation are bound to the input/output parameters of A. This declarative approach does not reduce the expressive power of the model as some (or all) of the input parameters of the inverse operations B can be bound to the output parameters of A and, as such, are determined during the execution of A.

Since compensating transactions should always be allowed to execute, the definition of conflicting operations must be restricted to allow the correctness of compensation:

3. The term “closed” refers to the traditional nested transaction model [46]
4. S1 and S2 may belong to the same transaction tree or be two unrelated transactions.
5. The specifications of inverse operations shouldn’t be a major problem since object interfaces usually provide inverse operations.

Definition: Let A and B be two open transaction initiation operations of an object, and A^{-1} and B^{-1} be their inverse operations respectively, A and B conflict with each other if starting from some initial state of the object, the effect of interleaving the execution of the following couples of operations: A&B, A^{-1}&B, A&B^{-1}, A^{-1}&B^{-1}, differs in at least one of the following ways:

1. The final state is different in the two cases
2. The result parameters returned by the first operation are different in the two cases
3. The result parameters returned by the second operation are different in the two cases

Intuitively, since an open transaction operation A may be compensated later in time, any transaction operation that follows the execution of A must not prevent the execution of the inverse operation A^{-1}, i.e. it must not conflict with either of them.

In the extended transaction model, any combination of open and closed transaction initiation operations are allowed. The abort recovery scheme is then extended as follows:

> when a transaction operation aborts, all its children must be rolled back or compensated; specifically:
>
> - every active subtransaction receives an abort exception. Then it behaves as if it had decided to abort itself;
> - every committed closed subtransaction is rolled back; and
> - every committed open subtransaction is compensated by executing the inverse transaction initiation operation. The inverse operation must contain enough semantics to undo the effects of the whole subtree of transactions: no nested compensation is required. Yet, an inverse operation is itself a transaction initiation operation and may invoke other operations provided by different objects.

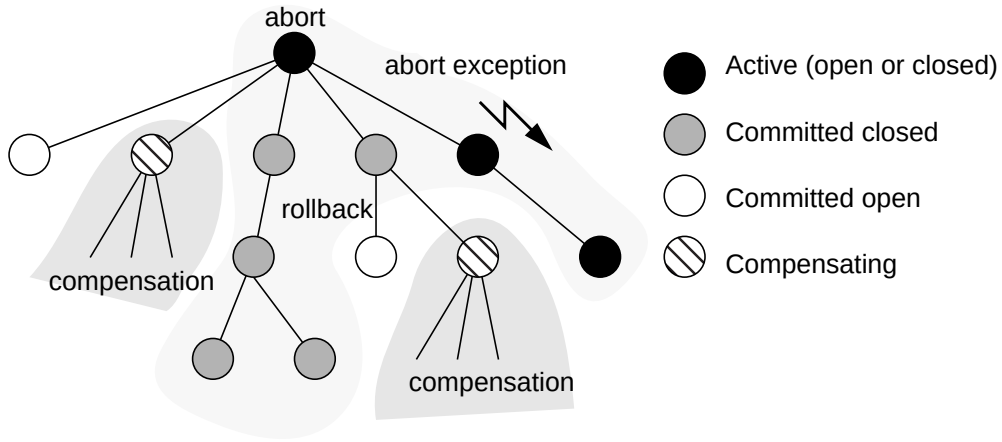


Figure 7-2. A Transaction Tree

In Figure 7-2 the abort recovery mechanism is shown.

Acronyms

ACID Atomicity, Consistency, Isolation, and Durability

ANSA Advanced Networked Systems Architecture

CMC Computational Modelling Concepts

DPE Distributed Processing Environment

IDL Interface Definition Language

INA Information Networking Architecture

ODL Object Definiton Language

ODP Open Distributed Processing

OMG Object Management Group

ORB Object Request Broker

QoS Quality of Service

RM-ODP Reference Model for Open Distributed Processing

ROSA RACE Open Service Architecture

TINA-C Telecommunications Information Networking Architecture Consortium

References

TINA-C documents

[1] *Overall Concepts and Principles of TINA*, Document No. TB_MDC.018_2.0_94, TINA-C, December 1994.

[2] I*nformation Modelling Concepts*, Document Number TB_EAC.001_1.2_94, TINA Consortium, April 1994.

[3] E*ngineering Modelling Concepts (DPE Architecture)*, Document No. TB_NS.005_2.0_94, TINA-C, December 1994.

[4] *TINA Distributed Processing Environment (TINA-DPE)*, Document No TR_PL.001_1.3_95, TINA-C, December 1995

[5] *TINA Object Definition Language Manual*, Document No TR_NM.002_2.2_95, TINA-C, April 1996.

[6] *Object Grouping and Configuration Management*, TINA Report No EN_TH.003_1.1_95, TINA-C, August 1995

[7] *TINA-C Stream Channel Model*, Version 1.3, EN_MFJ.001_1.3_95, TINA-C, July 1995.

[8] *Unification of Connection/Session Graphs, Stream Interfaces, and Channel Model*, Document No TR_PL.001_1.3_95, TINA-C, April 1996.

[9] *TINA-C Glossary of Terms,* Document No. BL_GL.001_1.0_960412, TINA-C, April 1996.

[10] *TINA-C Transaction Model: Proposal and Open Issues*, Document No EN_NM.009_1.1_95, July 1995.

[11] *Connection Management Architecture,* Document No. TB_JB.001_2.0_94, TINA-C, December 1994.

[12] *Connection Management Specifications ,* Document No TP_NAD.001._1.2_95, March 1995.

[13] *Security Architecture*, Document No. TR_EGL.004.2.0_1996, TINA-C, March 1996.

[14] *The Market for Information Services and its Demands on TINA-C (TINA-C Enterprise/ Business Model)*, Document No. TB_MB.001_2.0_95, TINA-C, March 1996.

[15] *Quality of Service Framework,* Document No. EN_EX.PFM.001_1.0_94, TINA-C, December 1994.

[16] Berndt, H. & Minerva, R. (eds.) *Service Architecture*, TINA-C Baseline, Doc. Label TB_MDC.012_2.0_94, TINA-C, March 1995.

Published Papers

[17] Natarajan, N. & Anderson, E., *Experiences in using a Distributed Processing Platform for Network Management*, Proc. DSOM 95, Ottawa, Canada. October 1995.

[18] Natarajan, N. *INAsoft DPE: A Platform for Distributed Telecommunications*

Applications, Proc. TINA 95, Melbourne, Australia. February 1995.

[19] F. Dupuy, G. Nilsson, Y. Inoue, *"The TINA Consortium: Towards Networking Telecommunications Information Services"*, ISS '95

[20] Nilsson, G., Dupuy, F. & Chapman, M. *An overview of the Telecommunnications Information Networking Architecture*, Proc. TINA 95, Melbourne, Australia. February 1995.

[21] B. Kitson. *CORBA and TINA: The Architectural Relationships,* Proc. TINA 95, Melbourne, Australia. February 1995.

Other documents

[22] ANSA, *The ANSA Computational Model*, AR.001.00. Architecture Projects Management Limited, August 1991

[23] [ANSA, *ANSA Atomic Activity Model and Infrastructure*, AR.004.00. Architecture Projects Management Limited, 1992.

[24] *Using Path Expressions as Concurrency Guards*, TR.022.00, Architecture Projects Management Limited, 1993.

[25] P. G. Bosco, G.Giandonato, C.Moiso, *A Distributed Processing Model for Telecommunication Services and Operations Software*, *TINA'93 - The Fourth Telecommunications Information Networking Architecture Workshop* Proceedings Vol II pp 77-92, September 1993.

[26] Bellcore SR-NWT-002282 , *INA Cycle 1 Framework Architecture,* Issue 2, April 1993.

[27] ISO/IEC 10746-2.2 / ITU-T Draft Recommendation X.901, *Basic Reference Model of Open Distributed Processing - Part 1: Overview,* International Organization for Standardization and International Electrotechnical Committee, 1995.

[28] IISO/IEC 10746-2.2 / ITU-T Recommendation X.902, *Basic Reference Model of Open Distributed Processing - Part 2: Foundations,* International Organization for Standardization and International Electrotechnical Committee, 1995.

[29] ISO/IEC 10746-3 / ITU-T Draft Recommendation X.903, *Basic Reference Model of Open Distributed Processing - Part 3: Architecture*, International Organization for Standardization and International Electrotechnical Committee, 1995.

[30] OMG Document Number 92.11.1, *Object Management Architecture Guide*, Revision 2.0, December 1993.

[31] OMG Document Number 95.xx.yy, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995.

[32] Bellcore, TR-STS-000915, *The Bellcore OSCA Architecture*, Issue 2, October 1992.

[33] ISO/IEC DIS 7498-4/ ITU-T Recommendation X.700, *Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 4: Management Framework*, International Organization for Standardization and International Electrotechnical Committee, April 1992.

[34] RACE Project R.1093 (ROSA) Deliverable 93/BTL/DS/A/010/b1, *Foundation of*

Object Orientation in ROSA, Release Two, RACE, May 1992.

[35] Jean Bernard Stefani. and Yann Lepetit, *The SERENITE Long Term IN Architecture*, *TINA'93 - The Fourth Telecommunications Information Networking Architecture Workshop*, L'Aquila Italy, September 27-30, 1993, Proceedings Vol I pp 45-63, September 1993.

[36] *Object Transaction Service*, OMG document No. 94.8.4, August 1994

[37] *X/Open Distributed Transaction Processing*, Reference Model (G307) and specifications (P209, P415, P306, P305, C193, …).

[38] *Advanced Transaction Semantics for TINA*, M. Fazzolare, B. Humm, R. Ranson, Proceedings of TINA'93

[39] *Less-Than-Transactional Semantics for TINA*, R. Ranson, Proceedings of TINA'95

[40] *Exploiting an Extended Transaction Model for Connection Management Applications*, E. Grasso, G. Spinelli, Proceedings TINA'95

[41] *Multilevel Transaction Management, Theoretical Art or Practical Need?* C. Beeri, H.J. Schek, G. Weikum; in Advances in Database Technology - EDBT'88, Springer-Verlag, 134-154, 1988.

[42] *A Transaction Model for Active Distributed Object Systems*, A. Buckmann, M. Tamer Ozsu, M. Hornick, D. Georgakopoulos, F.A. Manola; in Database Transaction Models for Advanced Applications; Morgan Kaufmann Publishers, 1992.

[43] *Sagas*, H. Garcia Molina, K. Salem; in Proc. ACM SIGMOD Int. Conf. on Management of Data, May 1987.

[44] *Transaction Processing: Concepts and Techniques*, J. Gray, A. Reuter; Morgan Kaufmann Publishers, 1993.

[45] *A Formal Approach to Recovery by CompensationgTransactions* H. Korth, E. Levy, A. Silberschatz; Proc. of 16th Conf. VLDB, 1990.

[46] *Nested Transactions* E. Moss; MIT Press, 1985.

[47] *Principles and Realization Strategies of Multilevel Transaction Management* G. Weikum; ACM Transaction Database Systems 16(2):132-180, June 1991.

[48] *Concepts and Applications of Multilevel Transactions an Open Nested Transactions* G. Weikum; in Database Transaction Models for Advanced Applications; Morgan Kaufmann Publishers, 1992.