DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
REPORT A-1998-2

Trading services in open distributed environments

Lea Kutvonen

UNIVERSITY OF HELSINKI
FINLAND

DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
REPORT A-1998-2

Trading services in open distributed environments

Lea Kutvonen

*To be presented,
with the permission of the Faculty of Science of the University of Helsinki,
for public criticism in Auditorium XII, Main Building,
on June 24th, 1998, at 12 o'clock noon.*

UNIVERSITY OF HELSINKI
FINLAND

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 941-45-8223-3
Computing Reviews (1998) Classification: C.2.0, C.2.4, D.2.11
Helsinki 1998

Trading services in open distributed environments

Lea Kutvonen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Lea.Kutvonen@cs.Helsinki.FI

PhD Thesis, Series of Publications A, Report A-1998-2
Helsinki, June 1998, viii + 231 + 6 pages
ISSN 1238-8645, ISBN 941-45-8223-3

Abstract

The current development of computing and telecommunication environments aims towards interoperability across separate platforms and organisations in a world-wide fashion. Interoperability means that software components can be exploited from arbitrary computers in such a way that the service semantics is preserved. The mechanisms supporting interoperability at the application level must mask heterogeneity of the compound computing environment.

The term 'open distributed processing' does not currently have a single, commonly accepted interpretation. Therefore, the main system architecture models are critically analysed in order to contrast the facilities for interoperation in each model. The focus of this dissertation is on the problems arising when the interoperating computing systems are controlled by autonomous organisations, i.e., problems of federated systems. Federations between sovereign systems involve exploitation of open interfaces and run-time information about the system facilities. Open systems, especially federated systems, must be based on a shared, very high level architecture model. The Open Distributed Processing (ODP) framework standard has been developed to enable world-wide computing services to evolve. Among other specifications, the family of ODP standards also identifies a set of fundamental services required from each participating system.

The trading service is one of the essential meta-information services of open systems. Trading presents a global mediator for information about available services and their properties. This dissertation presents work on the provision of the trading functionality. It analyses the requirements of trading designs and the ways trading function interacts with the system environment. The focus of this study is on federated trading, but trading problems in traditional distributed environments are also discussed. Furthermore, the design, implementation and performance of the DRYAD trading system is presented. In addition, exploitation scenarios for trading functionality in open distributed environments are analysed. Special attention is given for the use of trading in the explicit exchange, negotiation, and contract establishment required for interoperation in federated environments.

The trading functionality is a powerful tool to be used within the open infrastructure to support controlled cooperation between autonomous organisations. It allows construction of a world-wide computing environment that tolerates the constant evolution of services and applications. The exploitation of such a world-wide system requires software engineering tools that are based on the open system services. Therefore, major changes are expected in the areas of application architectures and software development processes within the next few years.

Computing Reviews (1998) Categories and Subject Descriptors:

C.2.0	Computer-communication networks: General
C.2.4	Computer-communication networks: Distributed Systems
D.2.11	Software engineering: Software Architectures

General Terms:
Design, Experimentation, Standardisation

Additional Key Words and Phrases:
Open systems, distributed systems, federated systems, open distributed processing reference model, trading function, middleware, distributed applications, software architectures

Acknowledgements

I would like to express my gratitude to my supervisors, professor Martti Tienari and Dr. Timo Alanko, for watching over my studies.

The work leading to this dissertation was carried out at the Department of Computer Science of the University of Helsinki. The department, headed by my supervisor professor Martti Tienari, has provided me excellent working conditions.

The constructive parts of the work were programmed in DRYAD project. The financial support of the Technology Development Center (TEKES), Telecom Finland, Helsinki Telephone Company, ICL Personal Systems, Finnish State Computing Centre, and Center for Scientific Computing for this project is gratefully acknowledged.

Involved to the DRYAD project were 22 of our undergraduate and M.Sc. students. I would like to thank them for their effort and contributions to the project.

The text of the dissertation was written during my study leave in Helsinki Graduate School of Computer Science and Engineering (HeCSE). The financial support of HeCSE is gratefully acknowledged.

I appreciate the occasional – but most useful – discussions about different research cultures and writing styles with professor Heikki Mannila. I would like to thank Dr. Timo Alanko, Mr. Max Hamberg, Dr. Teemu Kerola, Ms. Bonnie Kerola, Mr. Pekka Kähkipuro, Ms. Liisa Marttinen, professor Martti Tienari, Dr. Henry Tirri, and Dr. Hannu Toivonen for taking turns in commenting the publications that finally lead to this dissertation. Especially Timo has provided me with more than a handful of comments in the margins.

I would also like to thank the Finnish follow-up group for ODP standardisation for its interesting discussions, closely related to my work. Especially, I would like to thank Liisa for taking me to my first standardisation meeting and chairing the follow-up group during the first years, and Pekka for co-authoring some articles induced by the group discussions. I appreciate the cooperation of Finnish Data Communication Association (STY ry), Finnish standards organisation (SFS), and the steering committee of telestandardisation with the ODP group.

The international standardisation meetings have given me an excellent opportunity to discuss with the experts in the area. I appreciate the leadership of Dr. Mirion Bearman, professor Peter Linington, and Dr. Kerry Raymond in these intense and stimulating discussions. I thank Mirion also for encouraging my participation and studies early on.

The final preparation of this dissertation appeared to be a surprisingly pleasant experience, due to the good will of everyone involved. The practical cooperation worked seamlessly, and the feedback given on my research proved to be both healthily critical and encouraging.

I would also like to thank all my colleagues and friends for their refreshing company and their pursuit to interleave work and pleasure – often associated with impertinent and incisive discussions.

I take this opportunity to express loving and caring thanks to my parents, Rauno and Seija Summanen.

Above all, I thank my husband Petri for bearing my commitments and for encouraging me to bring this dissertation to a conclusion. He has supported my work by playing all roles from computer support organisation to co-authorship, not forgetting the care he has given me.

Helsinki, May 1998

Lea Kutvonen

Contents

Part One

INTRODUCTION

Chapter 1

Introduction

1.1 World-wide information services

During recent years, demand for a highly integrated and world-wide information processing network has grown. Examples of services in such a network can be easily found in everyday life: bank services, shopping, and travel agencies. Communication services, such as mail, telephone and fax, are already being replaced with electronic mail and video-conferencing. Even those who are not educated as computer professionals have skills for using electronic mail and access to such services. Moreover, during the last two years, the wide audience has been accustomed to TV and newspaper reports about the Internet and information services on it. In future, we can expect this tendency to expand. Perhaps information retrieval services offered by traditional libraries will become complemented with electronic libraries and even widened towards telepresence systems.

As information services are slowly being adopted as everyday utilities, the clients require that services are integrated and easily accessible throughout the world. An international money transfer system is an example of world-wide service integration. People even expect that the integrated services will be reachable via multiple alternative access technologies. For example, the demand for mobile phone access to Internet services is increasing. Moreover, people want to negotiate about the quality of service and to have a possibility to choose between several service providers. For example, when in a hurry, a user is ready to pay more for a fast service. In addition, the competition between service providers contributes to the overall quality of available services.

The software producers are not yet, even technically, able to fulfil the user requirements for world-wide services. While the users require world-wide service integration, exploitation of new technology, competitive choice of service providers and access technology, and negotiable quality of service, the software vendors and service providers try to answer these requirements. They try to embed new technology within their architecture solutions, and develop advanced services for the users. They aspire to these goals via cooperation in vendor consortia, and by developing more efficient tools for software production. The software producers work together in various vendor consortia in order to achieve shared recommendations and interface standards.

World-wide cooperation between computing systems and telecommunication facilities requires that the overall system architecture covers situations where services are supported by independent organisations in a multi-vendor computing environment. Such a system is continuously evolving: new services are created, and existing services are offered by a varying set of service providers. Furthermore, the history of the systems involved is different, thus introducing differences, for example in information contents, expressions, interfaces and processing technology. The major design problems in the multi-organisational environment are related to the cooperation mechanisms between the services supported by the sovereign organisations and their computing systems.

Combining the architecture goals of interoperation among sovereign systems, support for service evolution, and support for independent system administration lead to a situation where the public availability of interface specifications is not adequate. A set of infrastructure services is required for exchanging meta-information between sovereign cooperating systems about available services and their properties.

Traditionally, a unifying middleware layer is assumed. The middleware allows autonomy of each component system in many respects, but still, shared platform architecture, identical application service behaviour and interfaces, and expressions for meta-level information are expected. The decisions on system capabilities, such as the supported service interfaces, are external to the system and applied to all parts of the global system.

In order to achieve durable solutions, services need to be based on sovereign systems that are able to adapt to various cooperation schemes. In contrast to middleware architectures, a sovereign system has an independent administration that is able to select the used technology and strategies, e.g., operating and networking systems, implementation languages, communication protocols, access rights, application service behaviour and policies, remuneration, expressions for meta-level information about the system, and even interfaces. The decisions about system capabilities are captured in the system as meta-information. The system infrastructure includes functionality that allows exchange of meta-information with other sovereign systems, and also allows further decisions to be made based on that meta-information.

It is important that the multi-organisational system architecture model also captures the latest technical facilities developed in the areas of telecommunication and computing. The architecture should promote, for example, mobile computing and multi-media services.

1.2 Federated computing

In order to support evolution and system independence, the interoperating systems need interface information at run-time. Publicity of interfaces is a basic requirement, but not sufficient. Both evolution and sovereignity of systems require that information about services can be updated with a consistent, run-time mechanism throughout the integrated network.

A temporary interoperation relationship between independently administered and thus sovereign software components or subsystems is called federation. A federation involves two or more independent software components each fulfilling a similar application functionality on their local administrative domain. Having a local and independent service provider at each domain guarantees locally the continuity of service. The software components can then act as clients to each other, for cooperation between the domains. If the software components themselves are aware of their administrative and technical differences and take care of necessary transformations in their communication, they are said to interwork. If the software components are supported by infrastructure services so that they do not need to be concerned about system boundaries, the communication form is called interoperation.

Federation requires implicit or explicit contracts between the cooperating entities. The contract includes agreements of

- the communication semantics;
- information representation;
- data exchange protocols;
- quality of service, such as timeliness, trustworthiness, and precision;
- security related information; and
- failure semantics and recovery protocols.

The contracts can be established on two system levels, between application level objects and between platform level objects. The mechanisms for establishing federations are different on these layers. For federated computing, the focus of interest is on the establishment of application level federations on a platform that is composed from federated platform services.

Federation establishment between application objects requires supporting services that mediate contract related information, meta-information, within and across system boundaries. One of these services is trading service. The trading function is a mechanism that can be used to exchange interface information. The trading mechanism allows advertising of available services, service providers, and their interface properties, and supports retrieval of service provider information.

When cooperation contracts are explicitly expressed, the trading mechanism can be used for contract negotiation. Each potential partner in a federation can advertise its interface properties, location, and preconditions for the contract. When an initiative for establishing the federation is made, the trading mechanism can pick suitable candidates for the federation. The properties and the preconditions of the selected candidates are then merged to the contract, and the candidates are promoted to federation partners. The federation partners obey the regulations agreed in the contract.

1.3 Overview of the dissertation

This dissertation studies the architectural evolution of open distributed systems and the provision of trading functionality in such system environments. Special attention is given for federated systems.

Part II develops and analyses the federated system environment in contrast to traditional distributed and networked systems. The discussion points out the semantical differences in the founding concepts of each system model. The essential concepts and mechanisms for open systems are adopted from the Open Distributed Processing Reference Model, RM-ODP [91]. The ODP model is a joint standardisation effort of ITU (International Telecommunication Union) and ISO (International Organisation for Standardisation). The goal of RM-ODP is the coordination of existing platforms in order to make the integration of the platforms possible. The framework is based on the idea that the computing platforms cannot be forced to share their structures, to offer all the same services, or to share a single object paradigm. Part II includes an introduction to the RM-ODP and a comparison of major platform architectures against the ODP reference model.

Part III focuses on the trading mechanism. The trading concepts and implementation related aspects of the trading functionality are analysed. Also, the differences of trading service designs in traditional distributed environments and in federated environments are discussed. Finally, various trading realisations and specifications are compared; we analyse the ODP trading function standard, and discuss the trader software produced by the DRYAD project.

Part IV studies exploitation scenarios for trading. Chapter 6 defines a generic binding process between objects in a federated environment. The federated binding protocol is compared to the corresponding solution in a traditional distributed environment. Chapter 7 discusses the use of traders in an electronic commerce environment.

Part V concludes the dissertation and presents future work. The system model presented has consequences on the suitable application architectures for federated environments, and in addition, it also supports distribution of software engineering processes. We also discuss the effects meta-information services may have on distributed programming patterns and frameworks, as federation-transparent communication primitives are promoted.

1.4 Research history

The work described in this dissertation consists of both conceptual development and software construction. Some of the conceptual ideas presented in this dissertation have been published as articles [129, 124, 123] in international conferences. Also, an early version of this monograph was published as licentiate thesis [126]. The conceptual work has also served as a basis for contributions to the standardisation of the ODP reference model since 1994.

The implementation work in the DRYAD project has largely been done as undergraduate student work [113, 71, 19, 68] under the author's supervision. Some further software design ideas have been drawn out as master thesis work [244, 245, 259, 178]. The software design and some performance measures have been reported in a conference article [125] and in a journal article [130].

The research work has been done in liaison with educational and standardisation efforts. One of the concerns has been to evolve the understanding of open system architectures and to develop national vocabulary for discussing such architectures. Therefore, an ODP dictionary has been developed (Annex A.) Major parts of the dictionary were in test use in an ODP related seminar [235] during the spring semester of 1997 at the University of Helsinki. The ODP concepts and viewpoints were in practical use in one of the student projects [68], and the subject of study in a master thesis [67].

In the following, the international publications are commented on in chronological order.

The early model of trading covered, in addition to the current trading functionality, also binding and service invocation functionalities. Thus the first article [129] discusses a technique to integrate trading-based service invocation functionality into traditional distributed systems ('implicit trading'). The article also sets the federation concepts as the goal of the research work by stressing the importance of sovereignity of trading domains, and denial of direct control or access to other traders information contents.

The federation theme was continued in the second article [124] by the introduction of the concept of 'service interface autonomy'. In this dissertation this concept is termed 'interface heterogeneity'. The article discusses service invocation functionality, exploitation of trading within it, and explains ideas of liaisons and contracts.

Participation in the international standardisation work on the ODP trader raised some questions that called for suggestions. One of these questions was that of a policy concept and a policy framework. Those concepts were tried out [123] and were used as input to the standardisation work. The policy framework of the ODP trading function standard has been through several evolution steps since. The article also discusses concepts of federation and the need for standardising a single federation protocol. Important points were also made on concepts of domain and federation.

The journal article [130] discusses a more technical problem: how to support service offer objects within the DRYAD trader. The analysis of the repository requirements showed that the traditional database management system was not suitable, and therefore we selected a special-purpose database management system (Debbie). In this paper, the analysis and the performance measures are reported. Accompanying performance results reported in the licentiate thesis [126] and also further in this dissertation. The article also gives encouraging results on the feasibility of trading in TINA environment.

The technical design of the DRYAD trader software is described in article [125]. This article also gives a relationship between the concepts of contract and liaison to the technical structures of service offers and bindings.

The major change from the licentiate thesis [126] to this monograph is the inclusion of a federated system model description. The federation ideas are further applied to application management in a recent conference article [127]. The article expresses a concern that open service specifications and management specifications seem to be drifting too far apart in relation to federation aspects. The article also describes how trading mechanism can be exploited for homogenising the access view to a heterogeneous technology environment.

Part Two

OPEN DISTRIBUTED COMPUTING ENVIRONMENTS

The term 'open distributed processing' does not currently have a single, commonly accepted interpretation, but the interpretations vary depending on the goals of the term's user. Therefore, in this part, we identify three models for open distributed computing. This study is formulated as a ladder from networked systems, through traditional distributed systems, to federated systems. Although the ladder reflects the evolution history of the system architectures, all designs are still in use and needed today. At each step of the ladder, more demanding requirements have been raised, and the solutions are each time built on the services realized by the previous step.

Chapter 2 discusses the problems and the design of open distributed systems according to the three system models. This introduction sets some specific vocabulary for open systems – reflecting the current vocabulary in the area and bridging between the terminology used for distributed and federated systems. The chapter also briefly introduces the trading function, the focus of this dissertation, and its role in federated system environments as a meta-information exchanger. Meta-information is information about the capabilities of the systems themselves.

Chapter 3 reviews the standardised reference model of open systems, RM-ODP. The reference model stands for the current open system architecture and responds to the requirements of distributed and federated systems. The review defines the basic concepts, the design methodology of the reference model, and the ODP computing environment. This introduction focuses on features necessary for federated systems, whereas some other tutorials are biased on more traditional systems. We can expect the ODP reference model to have significant effects on the long-term plans of system vendors.

In this dissertation, we expand the facilities for the federated architecture, especially for the support of system flexibility and system evolution management. Software vendors should gradually move towards federative architectures. Some of this movement is already visible. Chapter 4 discusses the three well-known, distributed platform architectures: CORBA, TINA and ANSA. We study the concepts and services of these platforms and compare them to the requirements of federated systems and also to the requirements set by the ODP reference model.

Chapter 2

Design of non-centralised systems

The computing industry has for a couple of decades searched for savings and prospected profits by integration of individual computers to form an interconnected network. The benefits of sharing hardware and software are commonly acknowledged, as well as the profits through increased availability and reliability. Furthermore, the productivity of programmers has increased together with the expressive power of the programming tools.

In this chapter, we develop the idea of a federated system architecture as a next step on a ladder where networked systems and traditional distributed systems also appear. This ladder reflects on one hand the evolution history of non-centralised systems, on the other hand a hierarchy of abstract computing layers used together. In the historical perspective, the ladder reflects a continuum. Especially, the borderline drawn around distributed systems is arbitrary and aims at clarifying the areas of concern in each case.

The development of federated system architecture ideas presents a need for meta-information services in the system. Therefore, we analyse the available meta-information in each system model. The increase of available meta-information is related to the evolution of object models. Object models have evolved simultaneously with non-centralised systems, although the evolution lines were not closely interrelated. We can identify a typical object model used together with each system model, although such classification leaves out many interesting solutions. The discussion of object models indicates suitable software development and programming tools, although such tools are not studied in this dissertation. The main goal of this theme is to explicate the special characteristics of the object model selected for the federated system model.

2.1 Overview

In the evolution of computer networks we can discern three major design models starting from networked systems, leading through distributed systems, and culminating to federated systems. Although all three designs in general acquire benefits from joining computers together, there are major differences in the goals of these designs. The networked systems plainly allow remote access from one computer to another. The system includes loosely-coupled hardware with non-integrated software. The difficulty of using networked systems necessitated the development of distributed systems with integrated software in control. The distributed systems trust in a shared software layer that homogenises the computing environment from the point of view of programmers and end-users. The assumptions of homogeneous environment and a shared control over systems owned by independent organisations are not realistic when world-wide information services are aspired. Therefore, a federated system architecture is needed.

In each design model, the ultimate goal of computer interconnection is interoperation. Although the required semantic level of interoperation varies between the models, and the service components involved in the interoperation vary, we can still give a general definition 2.1. The interpretation of this definition in each design is discussed in the following sections, together with the definition of service in each case.

Definition 2.1 *"Interoperability: The ability of two or more computer and/or information service components of such systems to exchange information and to mutually use the information that has been exchanged" [104].*

In each design model, interoperability is based on 'openness' of the system components. However, the evolution of the models has forced the definition of open system to gradually change. Initially, open system was one that used shared assumptions about the services and the environment, later on open system was expected to include services that reveal such assumptions.

We give the following three definitions for open systems:

Definition 2.2 *Open system (networked systems): "An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalised as protocols" [221].*

In the case of networked system, the major requirement is the availability of coding rules for messages exchanged.

Definition 2.3 *Open system (distributed systems): "A comprehensive and consistent set of international information technology standards and functional standards that specify interfaces, services and supporting formats to accomplish interoperability and portability of applications, data and people" [104].*

Where

Interface: "A shared boundary between two computer systems, or the components of such systems, or such systems and users" [104].

In the case of traditional distributed systems, the requirement covers not only the message encoding but also the semantics of the functionality indicated by the message. Still, the publishing mechanism of interfaces and interface related information is external to the system.

Definition 2.4 *Open system (federated systems): An open system is one that*

- *is prepared to interoperate with other open systems,*
- *is able to restrict the interoperation to such services and to such interoperating systems that are acceptable by the open system administration, and*
- *is prepared to negotiate about the interoperation technique in terms of available services, interfaces, protocols, remuneration, and quality of service (QoS).*

In the case of federated systems, the requirement explicates that the publishing mechanism for interfaces and interface related information must be supported by the system. Furthermore, the system is required not only to enable interoperation but also to control and even restrict interoperation relationships based on externally applied policies in each participating system. The policies are expressed in terms of meta-information on the open system services. In the case of federated systems, more categories of meta-information are available than in the other two system models.

A major difference in open system design models is the use of the information about the interfaces available at a computer, i.e., meta-information. Therefore, we introduce the term meta-information to capture this idea in general:

Definition 2.5 *Meta-information is information about the service interfaces supported by an open system. The meta-information describes the supported service types, interface locations, protocols suitable for the interfaces, and quality of service offers. Meta-information may represent either the entities producing a service or the types of those entities.*

Examples of meta-information are given later, separately for each system model.

2.2 Networked systems

From generally accepted descriptions [221, 64, 35, 222], we can summarise the following definition for a networked system:

Definition 2.6 *A networked system is composed of a set of independent computers that are connected to each other with a (local area) network. The operating systems of the computers may be similar or different, however, the selected communication protocols must be consistent and open.*

The goal of networked systems is simply to allow access via the network from a remote computer. Each computer has its private set of users and its private access control. In addition, each computer supports protocols for remote sessions. Each user normally works on designated computers, and using a different computer requires a new session to that computer. Also, each computer has a private file system, and the users must themselves move files between computers with explicit file transfer commands. The users are themselves responsible for all remote tasks, and therefore, they must be aware of the networked system configuration.

Interoperation in networked systems is restricted to the exchange of files and acceptance of sessions initiated from a remote computer instead of a local terminal. Thus we can define the networked system services as follows:

Definition 2.7 *Service: Networked system services include operating system services and protocols. The operating system supports*

- *creation, suspension, continuation and termination of processes;*
- *accessing and locking files;*
- *memory management; and*
- *checking of access rights.*

The protocols accept requests for performing remote programs. Especially interesting programs include remote login, file transfer, remote printing, electronic mail and news transfer.

Only the protocols can be considered to support a form of interoperation. However, the user of the system must be aware of the differences of the local services in each node, as well as of the topology of the network.

The computers can be workstations, PC's and server computers. Examples of the networked systems include UNIX with Sun RPC [201, 18] – or OSF/DCE [180] for OSF/1 [179], for UNIX, or for MS-Windows [152].

2.2.1 Model for programming

The applications that use networked systems are controlled from a single point. Cooperation between computers can be implemented using file transfer as a communication mechanism. A typical client-server configuration includes a set of workstations requesting files, even start-up files, from a dedicated file-server computer.

More sophisticated solutions have been implemented on top of the transport protocol layer with finer grain software components. In these solutions, the communicating components are processes instead of programs, and communication entities are protocol messages instead of files. A client-server configuration now consists of a client process that sends a request to a remote server process which in turn sends a reply message. The request-reply protocol is often connectionless, in order to avoid overhead [221]. The communication protocols provide reliable delivery of messages between the computers. The concepts required for such operations include ports or sockets for communication together with transport level protocols, such as TCP, UDP, or OSI.

The request-reply protocol can be extended to a full RPC protocol (remote procedure call [18]) for the use of programmers. The RPC protocol is a request-reply protocol between client and server processes, but the messages may contain meta-information about the representation format of the procedure parameters. We discuss the process model and the inter-process communication via message passing in more detail in Section 2.3, because the development of RPC leads towards distributed systems.

2.2.2 Meta-information

There is little need for explicit meta-information exchange and thus very little meta-information available in the networked systems at operation time. Interoperation is governed by decisions made at system configuration time.

We can, however, find some examples. The first set of examples covers facilities designed for end-users. The UNIX command 'ping' (supported by ICMP echo service) allows users to study whether a communication line exists between two computers and whether the remote computer is in working condition. Also, end-users can remotely query for processor load. The second set of examples covers facilities for communication protocols. Domain Name Service, DNS [159, 160], allows query of the IP address of a named computer, and ARP queries (address resolution protocol) [188] allow the computers to find out Ethernet identifiers based on their IP addresses.

2.2.3 Benefits

The benefits of interconnecting individual computers together are straightforward: There are plenty of tasks that can be performed with cheap microcomputers, such as management of electronic mail. However, the tasks often require that the results are communicated to remote locations. For example, a department store can use microcomputers to collect credit card events during the day, but communicates the events to the credit card company in the evening.

In the form described in this section, the networked systems are too simple for extensive distributed computing. However, the networked model serves as a starting point for true distributed systems.

Currently, typical systems combine features of networked and distributed systems. A typical system can include, for instance, a set of UNIX workstations, connected with NFS (Network File System) [215] to a large UNIX file server. Additional services may include software distribution to support system maintenance, automated backup for user file systems; USENET news access, WWW browsers; RPC programming tools, and tools for object oriented programming.

2.2.4 Problems

For networked systems, the facilities for interoperation are founded by standardisation and established through implementation of standard interfaces. Changes in such an environment are very slow and costly. Furthermore, the protocol standards allow several optional features. Combinations of such features are called ‘profiles’. In some cases, internal policies of organisations force a certain profile to be used in all communication. Unfortunately, it is possible that interoperation is not possible between all profiles of a standard.

Heterogeneity in the implementation of operating and networking system services is to some extent allowed, but each heterogeneous solution requires that the end-users of the joint system understand the differences in the cooperating systems. In practice, this leads to a wish that each node should support a similar, shared environment. Only such configurations allow end-users and programmers to experience a homogeneous and comfortable working environment. A homogeneous environment can be achieved only through the cooperation of all node administrations. Thus, the node administrations would lose autonomy on selecting local operational policies.

It is typical that a networked system has little or no support for fault tolerance, except the recovery mechanisms of the communication protocols used. The platform gives, for example, little support for building replication groups or using multi-cast communication, and thus, in practice, redundancy is not exploited in normal systems.

The interconnection mechanism restricts the possibilities of preserving privacy and autonomy of the nodes. The authorisation control of each node is based only on access control of user accounts. If a user needs any access to a node, an account is required, but then most procedures and resources are accessible.

2.3 Traditional distributed systems

Traditional distributed systems are designed to have integrated, consistently administered software that controls a set of interconnected computers. From the literature, we have selected the following definition:

Definition 2.8 *"A distributed system is a collection of independent computers that appear to the users of the system as a single computer" [221].*

A distributed system ensures that end-users and programmers view a distributed system as a virtual uniprocessor, and they do not see differences between local and remote resources.

However, the components of a distributed system may be heterogeneous. Therefore, the development and maintenance of software for distributed systems are difficult and expensive.

The interoperation model within distributed systems is based on the concepts of processes and inter-process communication. While networked systems interoperate between computers, the distributed systems interoperate between processes. Therefore, we need to introduce a distinction between the terms service and server:

Definition 2.9 *Service: The specification of the primitives, parameters and actions available for a client. Service is an interface description and does not define the implementation of the activity offered.*

Definition 2.10 *Server: A server is a process that runs on a single computer and implements the service or part of the service [218].*

Thus, a service is an abstraction of a functionality that can be invoked at the system, while a server is a specific implementation that can be executed at the system. For simplicity, we have here chosen to use such a definition for server that does not allow a single server to be split to several nodes. This choice was made because many distributed object management systems allow object components to reside in different nodes and exploit a kind of interoperability interface for communication. Such solutions are further discussed in Section 2.3.4. A server, as defined here, can be used as an object component. A service must always be accessed through an interface available at a node, and thus composition of several servers to support a service does not essentially change the model. The localised servers are also easy to map on the interprocess communication model discussed in Section 2.3.3.

Typical platform services in distributed systems are distributed file systems [218], global name services [135], directory services (like X.500 [105]), distributed time services [134], and transaction management [45]. The communication services are extended by broadcast and multi-cast.

Example systems include Amoeba [223], Clouds [41], OSF/1 [179], Mach [195], SunOS/Solaris [216], OSF/DCE [180], MS-Windows NT [153], and OS/2 [80].

2.3.1 Architectural characteristics

In order to achieve a virtually unified computing environment, the programmers need a homogeneous interface to the whole system. This requires that the system supports distribution transparency, i.e., includes functionality that masks heterogeneity and remoteness of system components from programmers and end-users by automatic mechanisms and selected conventions.

As part of the distribution transparency notion, the following aspects are considered important [221]:

- Location transparency hides the location of hardware and software resources from the users. For example, adjoint file system names like `uhecs:/usr2/summanen/.profile` are not allowed because they in fact reveal the allocation of the files to separate computers.
- Migration transparency allows resources to move between locations and still keep their names unchanged.
- Replication transparency masks, from a user, the existence of additional copies of resources (e.g., files) created by the system. Usage of replications can improve system performance and fault tolerance.
- Concurrency transparency hides, from a user, the existence of other simultaneous users of the same resource.

- Parallelism transparency hides, from a user, the existence and the use of multiple processors for parallel executing (parts of) a single service.

These transparency terms are widely used in literature with the above described meaning. Nevertheless, in Section 2.4 we will define standardised transparency terminology used in RM-ODP, and some of the terms are redefined with higher requirements. From Section 2.4 on we will use only the RM-ODP definitions. The transparency terms are first introduced here in order to avoid the potential confusion created by the differing use of the same terms at closely related areas.

Another important goal in the development of traditional distributed systems is the flexibility of the system, i.e., the modifiability of the system with a reasonable cost. Flexibility can be increased by the correct modularisation of the operating system. An attempt for this is the micro-kernel model [221, 223]. In the micro-kernel model, most of the traditional operating system tasks are externalised to user-level processes that can be allocated to any processor in the system. The micro-kernel itself is run on all system processors. The micro-kernel architecture has the benefit of allowing multiple implementations of a service to be present simultaneously, for example, multiple file servers for different file structures. Because the micro-kernel supports communication, it has also a major role in building the system dependability. Dependability should cover both availability of the system services, and fault tolerance of the operating system.

Finally, the communication mechanism should be scalable, i.e., the selected communication model should have good performance, even if the number of processors in the system is considerably increased. The design task is difficult, because the global information required for optimisation. Decentralised algorithms cannot expect any computer to have complete information of the system state but allow each computer to make decisions based on local information. Furthermore, no global time can be expected. Of course, the algorithms must behave reasonably, even if some of the computers fail to participate in the protocol [221].

In order to achieve the above mentioned design goals, the distributed system must include some additional services on top of the operating systems of each node. These services take care of communication within the distributed system and control the global behaviour of the system. These controlling services are jointly called middleware. For programmers the middleware supports concepts of transparency. Especially when the hardware or the operating systems are heterogeneous it is convenient that an extra software layer homogenises the interface used by the programmers (a locally known example is AHTO [1]).

The interesting distributed system mechanisms include

- management of distributed processes: remote interprocess communication, synchronisation, naming and process management, and
- resource management: resource allocation, detection of deadlocks, protection and security, and types of services to be provided.

The following sections discuss these mechanisms.

2.3.2 Process and object models

A distributed system can be designed either using a process model or an object model [64]. At the logical level, the models can be mapped to each other, but on the technical level, there are differences for example in organising access to data, and synchronisation.

For the purposes of discussing distributed systems, we define a process to consist of one or more execution threads of program code that can access a single, colocated memory space. The definition allows parallel processing to take place, thus enabling a good level of throughput and performance within a computer. The process management functions include process creation, termination, allocation to a processor, suspension, and reactivation.

With the process model all the operating system functions and applications are constructed as sets of processes that interact by sending and receiving messages. Messages are received at ports representing message queues, and can be denoted with a protocol specific address format.

An object is a programmer-defined abstract model of a resource that encapsulates a data structure [109]. It consists of internal data, the object state, and presents methods that operate on the object state. To access or modify the internal state, the client process must invoke one of the methods. This property is called information hiding [186]. The object concept provides a mechanism to replace the actual data manipulation with more abstract method calls. Instead of including the data manipulation code to the client processes, the code is captured together with the data structure within the object. Therefore, object-based systems are, in principle, easier to modify than process-based systems where the data structures can be manipulated by client processes without an intermediate abstraction. These kinds of objects are never active, and there must always be a client process to invoke methods on them.

We did not formulate the descriptions of objects as definitions at this point, because we will later introduce an object concept more suitable to the federated systems we are interested in. The object concept that was discussed in this section is later on referred as an encapsulated data structure.

We continue the discussion of typical object models for distributed systems in Section 2.3.4.

2.3.3 Middleware

The term 'middleware' has come to mean any facilities that support communication between application layer clients and server processes. An attempt to particularise the middleware definition can be found in [204]: "Middleware is software that allows elements of applications to interoperate across network links, despite differences in underlying communications protocols, system architecture, operating systems, databases, and other application services".

This definition concentrates on the communication facilities of the system. However, in most commercial cases, the components of middleware can currently be considered to cover not only the communication software, but also the rest of the execution environment (name management, security, and other core services; and application specific services like transaction management and SQL optimisation) and deployment facilities (administration tools like configuration and performance management) [204, 16]. This interpretation is consistent with a view that middleware services are general-purpose services that are able to exploit operating system services of various vendor architectures and support a considerably large variety of applications [16].

In the following, we review the concerns of middleware services in traditional distributed systems. We start by discussing the process management problems, covering aspects on processors and memory under a consistent control through the distributed system middleware, and continue with discussion on interprocess communication, covering aspects of security.

Process management

In the distributed system, a new process is allocated to one of the processors available. Several processor allocation algorithms have been developed, differing in the way processor load information is used, and the members of the decision making. The allocation algorithms use the private process (and thread) management methods and information of each operating system involved. After the process allocation, each process is controlled by the local operating system, i.e., the process scheduling is normally a local functionality. In some cases, the application is composed in such a way, that a special co-scheduling scheme is required to avoid unnecessarily high network load.

In a distributed environment, process persistence is a preferred property. Persistent processes survive temporary failures of the operating system or the network system services. Persistence can be maintained either by replicating the processes or by storing and recovering intermediate state information in non-volatile storage. A well-known system with persistent processes is ISIS [17]. The ISIS system also supports process migration, as the same communication mechanism can be used for trapping a lost connection and for reconstructing it to a new location.

The memory management activities in a distributed system are dependent on the system organisation. Usually, traditional distributed systems are built as multicomputers (each CPU has its own private memory). In a multicomputer environment, execution of a task induces interprocess communication. In order to ease frequent message passing scenarios, distributed shared memory concept was introduced [138, 139]. To the basic mechanism, several optimisations have been suggested. Especially, organising the shared memory to shared variables (e.g., [15]), or shared object space (e.g., [62]) improves the system performance. These optimisation methods reduce the amount of shared data, and simultaneously increase the amount of meta-level information that can be used for further optimisation.

Interprocess communication

The basic interaction form in distributed system is passing messages between processes using some transport protocol, like ISO TP, TCP/IP or UDP/IP. Whichever transport protocol is used, the programmer has to consider synchronisation of the processes, the reliability of transport, the reliability of the processes communicating with each other, and the independent failure of the processes. Also, if multiple processes must receive the same messages, the programmer has to simulate multi- or broadcast by sequential send operations.

To support an easier programming interface for interprocess communication, the remote procedure call mechanism was introduced [18]. The mechanism allows programs to call remote procedure implementations in the same way as they call local procedures. The message passing mechanism that implements RPC is not visible to the programmer. The system internally marshals the procedure parameters to messages and again unmarshals them for processing. The marshaled messages can be split into parameters based on shared and inherent knowledge about the number and the type of the operation parameters. The message only carries the name of the operation and the parameter values or references. Several advanced RPC mechanisms are able to manage reference parameters of some programming languages (for example [72, 117]). Several advanced RPC systems have been developed that are able to mix programming languages, for example see [63].

Although the RPC concept strives for imitating a local procedure call, additional features and problems are introduced, because of the distributed nature of the mechanisms. First, the client must bind to a server before communication can succeed. By the binding operation, the client process becomes aware of a receiving port of the server and is then able to send messages to it. If the server address is encoded to the client process, the system is not very flexible nor fault tolerant. Therefore, some distributed systems use dynamic binding, i.e., the server is selected at the time of RPC initiation either from a mapper process or an external repository. In both cases, the dynamic binding mechanisms may become extremely heavy when the number of nodes and processes increases. Second, the remoteness of nodes creates potential of failures that affect only some parts of the system. The client and server processes and the data transport primitives may fail independently from each other. Timeout mechanisms are used for recovery. Advanced RPC mechanisms make automatic recovery attempts, governed by execution rules like 'at most once' and 'at least once'. Such rules denote that the client should know how many times a service has been executed when the RPC call has successfully terminated.

The RPC management mechanisms do not include much support for service evolution. The RPC implementations usually use numbers for identifying the requested service. This service number can be supplemented by a version number, in order to allow gradual evolution of the services reachable by RPC. In addition, a mapper process is commonly used to map the service and version numbers to the port number reserved for the actual server process [214].

The RPC concept itself is continuously evolving. RPC realisations have been supplemented with various additional features, like transaction properties, failure recovery features, and membership services for group communication (e.g., [17]).

In a distributed system, not all resources are accessible to any user for privacy reasons, remuneration reasons, or information integrity. The services related to securing the resources and the information exchange between processes or computers include authentication of users, authorisation of processes to use resources, and encryption messages for transfer. In a simple case, the security services can be based on a centralised security authority that holds user identification, an access control list for the resources, and finally deliver cryptographic keys for processes. Some distributed systems trust in a security service like Kerberos [165], some others trust in services like Network Information Service (NIS) (formerly known as yellow pages service) [216]. NIS stores among other information also pairs of user names and passwords. For the programmers, secure interprocess communication can be offered via secure RPC packages. However, the programmer must instrument the RPC call with encryption keys, and user identifiers.

Other general services

Some general services, that were independently supported by operating systems in each node, have been replaced by distributed versions. Such distributed services have become an essential part of the distributed system middleware.

For example, distributed file services provide a consistent view to a shared file system that may have components at each of the nodes of the distributed system. The method for implementing such a view can be based on copying files between file servers as access to them is requested by clients. Another solution is to split files to smaller partitions and copy only the partitions between the servers. The partitions can be for example pages, or native structures for the files themselves (records, etc.). For accessing files the naming system of the files is required, i.e., the file directory service. It should offer location transparency for the client: the name of the file should be the same regardless in which computer the file is stored. This can be achieved by two-level naming schemes, where the client is offered a symbolic name that is within the file servers mapped to a system specific address. Well-known examples of distributed file services include Network File System (NFS) [215], and Andrew file system (AFS) [212].

2.3.4 Model for programming and program design

The realisation of distributed software is affected by the programming languages available, the design patterns guiding the realisation, and overall design methodology governing the realisation work. These areas are very rigorous with incompatible suggestion. Therefore, we only make some remarks on object-related languages, methodologies, and tools.

Distributed object models

Interoperability within a distributed system enables two processes to exchange data or data references and thus use a shared data space. Thus, a single application can be spread to multiple computers. The processes belonging to an application are under a shared control, but some forms of independence are present: the processes can fail independently from each other, the processes may be allocated to processors without a joint scheduling control, the processors may introduce heterogeneity at the instruction and data representation level, the code executed in the processes may originate from different programming language environments, etc.

The distributed application processes are typically constructed by the programmers using procedural languages, or object-oriented programming languages, together with advanced RPC libraries (examples of programming tool sets can be found by OMG [176]). The commonly used object-oriented programming languages, like C++, actually present an abstraction very close to the process model discussed in Section 2.2. The passive objects represent abstract data structures; the active objects present abstract data structures with an initial method that covers the full lifetime of the object [33]. Such an initial method typically includes method calls for other objects.

The benefit of using object-oriented programming languages is that they offer a structuring tool for collecting the common behaviour characteristics of essentially similar processes into a single description. Furthermore, the descriptions can be reused for new objects through inheritance. The object-oriented programming languages do not themselves offer a design technique, but they increase the efficiency of application production merely by insisting similar structures to be used throughout the production team. The design support for object oriented software can be found in object-oriented design patterns (e.g. [57]). A pattern describes a set of roles and interfaces associated with them. A typical application utilises a set of patterns and each object adopts a role in some of these patterns. The design patterns actually introduce some protocols between the actual objects. The patterns are selected from successful applications and the design is then reused in later software projects.

The programmer view to object model can be further enhanced by properties like object durability, persistence, immutability, and replication (see for example [54]). The programmer view to the communication between objects can be further simplified by hiding many transformation and control mechanisms within the middleware between them, like described for example in [252].

A further development on object languages, dynamic (or reflective) object technology, enables applications to be modified during development and runtime, without changes to the program code itself [131, 34, 142]. This feature essentially decreases the software system maintenance cost. The dynamic object language implementations maintain meta-information about object classes and inheritance relationships. Modifications on the object structure are applied to this meta-information instead of the object implementations themselves.

Interface definitions

In object-oriented programming, the important goal is to define correct interfaces for the objects. However, languages like C++ do not support a separate interface concept, but concentrate on defining the semantics of object methods. Languages like Interface Definition Language (IDL) from OMG [177] are more suitable for defining interfaces. One of the drawbacks in IDL usage is that IDL does not define the semantics of the defined operations at all, only the structure of the object interface in terms of exchanged data values and message contents. Software development projects typically use a combined set of these tools. The shared IDL definitions are used for generating RPC stubs and other lower level communication related parts, and the method semantics is defined by adding more traditional programming language sections. This approach enables considerably easy use of multiple programming languages in parallel.

Beside inheritance, polymorphism is an essential concept in object-oriented programming languages. Polymorphism allows an operation to be mapped to different kind of methods at the object implementation at run-time. The selected method depends on the actual information contained in the service request invoking the operation. A simple and often used way of selecting the method to be executed is to require the operation name and the method name to match [256].

Object databases and distributed object management systems

Another line of constructing distributed systems is structured around database system models. Distributed databases are frequently used as a basis for implementing traditional business applications such as office information systems, engineering databases, and medical information systems. Distributed object management systems (DOMS) merge the distributed relational database management system technology with object oriented data base technology [184].

The design goal of DOMS is to provide network, replication, and fragmentation transparency of transactions [184]. The network transparency hides the network management duties and the distribution of data. The replication transparency masks the physical copies of a single logical data item. The fragmentation transparency masks the physical distribution of database objects partitions.

The database objects are managed by a global transactional access management, implemented jointly by local manager processes. The object representation is not required to be homogeneous at the sites. Instead, an external schema is used for formulating user queries and transaction access between local manager processes. The external schema is defined over a global conceptual schema which is partitioned to local conceptual schemata at each site. The local conceptual schemata are then mapped to local internal schemata to describe the physical organisation of data at that site. The database objects can be clustered based on their type (shared method information), or each object can be partitioned. In horizontal partitioning, the objects can be clustered by other criteria, and for each relevant site, the type information is duplicated. In vertical partitioning, both the type information and the object data are partitioned to several nodes. This causes situations where both methods and data can be either local or remote in respect to the manager controlling a method execution [184].

Integration of existing object database systems can be supported by standardised interfaces and object models. The OMG consortium has been involved in an open data management recommendation, ODMG [29]. The recommendation introduces an object definition language, ODL [233], and its language bindings to some programming languages. The ODL language contains basic type construction tools, built-in collection of type generators, and alternatives for composite object semantics. The recommendation also defines an abstract object query language, OQL to match the data model.

An interesting example of distributed object management systems is the DOMS of DTE [144]. It clearly shows integration of heterogeneous, distributed database systems, and exploits CORBA platform design to support the local managers and their communication. The work also shows interfaces as objects and allows the interface objects to be polymorphic. The idea of polymorphic interfaces will be further developed for federated system model in Section 2.4.

Object-based tools

The increased use of object-oriented programming languages has lead to development of object-oriented design methodology. A well-known example is the object modelling technique, OMT [203]. The technique is based on objects that consist of a unique identity, public and private data, and public and private operations. Public operations are called services. For the specification of an object system, OMT uses three basic models: object model, dynamic model and functional model. Each model captures only a part of the specification and the models must be combined to get a whole system view. The object model describes the static structure of the system. It is represented by class diagrams showing active objects, data stores and messaging between objects. The dynamic model describes the potential changes in the system state. The model is represented by a state graph for each class of objects showing potential object states and transitions. The functional model describes changes of the data in the system. The functional model is presented as client-server relationships between object classes and the execution order of object methods. The technique encompasses four phases of the software engineering process: analysis, system design, object design and implementation. All phases basically describe features of implementation level objects, i.e., processes and abstract data structures. The OMT technique is being replaced by UML (unified modelling language) [175] that basically includes the same fundamental concepts.

For independent application areas object frameworks have been developed. An object framework is a model program that covers the essential functionality of the application, networking support for communication between application components, and usually also a generic graphical user interface [53]. The programmers need only to modify this model program to suit the requirements of the specific application. The benefits of this approach are apparent: New applications are fast to construct and have a similar appearance for the end-users. The programmers need not to be specialists on all technical areas, e.g. graphic and network programming, but can concentrate on the business model of the application area. Object frameworks are usually implemented as libraries. We do not yet have such concepts embedded into any programming languages nor into any middleware services. However, such solutions are clearly to be presented soon.

2.3.5 Meta-information

As a distributed system is governed by a single administration, meta-information is mainly needed during system or service design and implementation. Meta-information is required for ensuring the interoperability of objects that can be implemented either separately or in a heterogeneous programming environment. Meta-information is also used during the system operation time for administrative purposes. Such run-time meta-information is mainly created and modified by human administrators. However, facilities like process migration require meta-information to be present.

Typical meta-information interesting for distributed application development include object and interface definitions, and topology of the distributed application across the network. Interface definitions can be considered as service-related meta-information. However, such information is not widely used at run-time.

A simple form of service information can be found in RPC management, because the service requests are checked for conformance against the service and version numbers of the server. In addition, data representation formats can be captured either as a design rule or carried within the messages exchanged between application components. Many RPC mechanisms trust in self-describing data, and transport data in forms like ASN.1 [89], or XDR [214]. A more elaborated example of run-time service information is found in dynamic object systems, where explicit meta-information can be modified during system run-time.

The topology of a distributed application can be captured in terms of location information. The DNS and ARP services are commonly known implementations for location services in the Internet. In addition to the topology, authentication information is also required for remote communication. Such information is supported, for example, by NIS (Network Information Service). It stores pairs of user names and passwords (authentication service), computer names and network addresses (name service), group names and user names of the members (membership services), etc.

In the design of distributed applications, an important aspect is the expected and achieved quality of service. Quality of service concept has been traditionally used in relation with multimedia applications [38]. The quality of service is expressed as a measurable property, like latency, throughput, jitter, and availability time. The values for these attributes can either denote comfortable conditions for the users, or unacceptable behaviour of the service. The quality of service management is often covered by capacity planning, load balancing techniques, or denial of new tasks when the system capacity is reached. However, quality of service can also be monitored during the system operation. Monitoring information is commonly collected within the system, for example, in RPC stubs, and can be used for example for load sharing between the processors in the system.

We can summarise that run-time meta-information available in distributed system includes

- name and location services,
- membership services,
- data type information,
- quality of service monitoring information, and
- service information.

In comparison to the meta-information available in the networked systems, the above items allow more flexible cooperation between separate computers: Location services are used when processes are migrating and communication channels must be reconstructed, name services can be used to ensure that an interesting service can actually be accessed at a remote node; and furthermore, service information can be used to ensure consistent communication semantics between two processes.

2.3.6 Benefits

Distributed systems allow effective exploitation of computers in such a way that the overall system capacity and usability are compatible to that of a centralised system. The system availability may even be better for a distributed system. From the programmers point of view, the distributed systems offer a more convenient working environment in comparison to the networked systems.

2.3.7 Problems

When we consider distributed systems as the support mechanism for world-wide information services, we notice some conceptual and some technical problems. Furthermore, the object-oriented tools involved in the development of distributed platform and application services have their inherent limitations.

The conceptual problems arise from the distributed system design goal of a unified system environment. This goal leads to solutions that trust in a shared administrative control over the system, and also, to an assumption of a homogeneous middleware interface. However, these assumptions are not reasonable for a system that spans multiple organisations. Organisations cannot be forced to use same operational policies in their computing systems, nor forced to evolve their computing services in the same pace. A single de-jure and de-facto standard middleware interface is unlikely to arise. Such homogeneous middleware interface would stop both competition between vendors and evolution of middleware services for new demands.

The technical problems arise from the concepts of distribution transparency as a mechanism to hide differences of systems and remoteness of computing. When we adopt the multi-organisational system model, we have to modify the transparency requirements as well. For example, in a multi-organisational environment, end-users are prepared to be informed about service failures caused by mismatch of organisation policies, because such mismatches appear in communication between people as well. Thus there is no need to always hide all organisational boundaries from end-users or programmers.

Another technical problem category arises from the object-oriented software development environments that trust in inheritance as a basic mechanism for ensuring interoperability of objects. Thus interoperation is restricted to cases where the conformance of their interfaces can be checked at compilation time. However, in a multi-organisational environment, cooperation relationships should be possible with objects that are introduced some time during the system operation. The middleware should be flexible enough to adopt new application services from other organisations and to offer them for the local end-users. In a traditional distributed system, such flexibility is not supported. Instead, new services must be introduced through administrative actions and usually require a proxy or a gateway to be installed (or even programmed).

2.4 Federated systems

Federated systems are composed of sovereign systems that include middleware for interoperability support.

A system is considered to be sovereign when it has an autonomous administration. In a sovereign system, decisions – for example on system architecture, services supported, programming and interface definition languages, remuneration, authorisation policies, and communication protocols – can be done independently from any other systems.

Interoperability support includes mechanisms that negotiate about the shared capabilities of the systems and establish an interoperation relationship at system run-time. Such mechanisms are needed in a multi-organisational environment because the interacting software components cannot inherit properties that would ensure interoperability.

Cooperation ability between software components located to separate sovereign systems becomes separated to two concepts: interoperation and interworking. Interoperation between (application) objects means that the supporting infrastructure manages aspects arising from system sovereignity. In an interworking relationship the objects need to manage those aspects internally.

The characterising goal of federated systems is to enable world-wide information service systems, but still allow each member system to offer a localised computing environment for the end-users. A shared goal with the traditional distributed system evolution is the inclusion of new technology, like mobility, multi-media streams, and multi-party communication relationships.

We consider a federated system to be a community of federable systems that dynamically enter and leave federations. The federation and the federable systems we define as follows:

Definition 2.11 *Federation is a state of agreement between two or more systems about interoperation. Federations can be established and terminated during the operation of the systems. The federation agreement covers general communication related aspects, such as protocols and locations of interfaces, and service specific aspects, such as quality of service contracts.*

Definition 2.12 *A federable system is a sovereign system that contains middleware services that support federation establishment and management.*

2.4.1 Architectural characteristics

The goal of the federated system model is to improve the facilities for accessing services (not servers) from computing systems belonging to other organisations. Federation can be established between such systems that include similar facilities and that are allowed to federate by their owners. There is no predefined shared goal for the joint operation nor a shared control for the joint operation, but clients at any federable system can request services that are eventually performed at another federable system.

Composition of a world-wide system involves interconnection of the independent systems. The scope of potential federations is determined by the amount of common communication facilities and the amount of applications that are suitable for cooperation. A fully connected communication network between the federable systems is not necessary, as all services are not used at all federable systems and in some cases an intermediating system can be used. Furthermore, reachability of individual servers from all locations is not an objective. Therefore, practical limitations of interconnectivity do not invalidate the model. However, the communication services across the federated system should be able to join various communication technologies. For example, the communication services should be able to adapt the failure detection and recovery mechanisms to fixed and mobile networks.

In the world-wide federated system, it is not reasonable to expect that all organisations would offer a similar computing platform or application repertoire for their users. Instead, each organisation should be allowed to offer localised interfaces. We can adopt the concept of personalisation (from the context of micro-kernels [136, 238]) as a design pattern that separates the actual execution of services from the mechanism through which the users exploit the service. In the federated system, we allow transformation of generic service concepts to a local representation format at each member system. Such an architectural opportunity is beneficial for vendor competition as well: The vendors of federable system platforms and applications are granted a possibility of commercial competition with their products within the federated systems.

The federation management mechanisms are easier to build if federations are not established between whole systems but per individual services. Each service federation can be established independently and the federations can be reconstructed whenever changes at the services or service implementations are introduced. The service-wide federations require standardised contract schemata. Such work has already been initiated, for example within the business object modelling special interest group (BOMSIG) of OMG [170].

We introduce a federated system application architecture that contains sovereign objects establishing liaisons, i.e. contractual relationships, among themselves. The liaisons ensure that the interacting objects are technically able to exchange information and perform services for each other in a semantically consistent way. The technical capabilities are controlled using meta-information about the objects. The meta-information is maintained by platform level services.

Figure 2.1 illustrates the global system view where each service is essentially formed by application level objects and platform level objects. The properties of platform level objects effect essentially the interconnectivity potential of an application object, and thus platform objects must be included in the global system view. In distributed systems, the network of platforms is homogeneous and fully connected, and the global system view can be abstracted to contain only the application level objects. In federated system model, the interconnectivity and heterogeneity of the platforms must be considered. The federated system global infrastructure may consist of a set of distributed systems. The network of platforms is not fully connected. Instead, it obtains a shape depending on the shared protocols, the awareness of other systems, and the authorisation of remote users at each individual system. We can consider each federable system as an independent management domain [209].

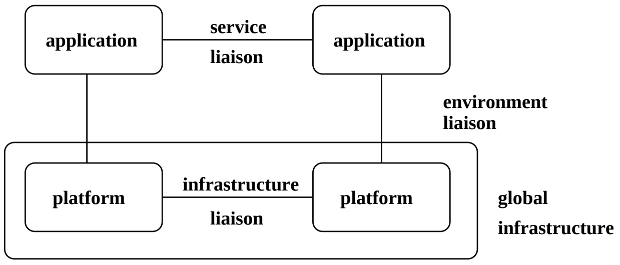


Figure 2.1: Global system view of federated system.

Examples of typical federated system applications include services for electronic commerce, i.e., authentication, billing, payment, and retail. Some services common in distributed systems, like distributed file systems or global naming services, do not appear as joint services of the federated system. Instead, such services can be supported by the federated systems and federated at will.

Typical federated system platform services include those of distributed system platforms, but especially additional services like trading, type repositories, federated binding management, and federated naming services. These services represent meta-information services fundamental for federated systems, and services that exploit meta-information for creating object federations. Some platform services are federated by nature themselves. We will further study these services when we discuss federated system middleware.

Currently, there are not yet any federated system implementations. However, examples of federated system characteristics can be found in TINA, CORBA, and ANSA systems, that are discussed in Chapter 4. The ODP framework specifies ODP systems that are allowed to be either traditional distributed systems or federated systems.

2.4.2 Basic concepts

The federated system model requires a more rigorous set of concepts for maintaining interactions between objects than traditional distributed systems. In a federated environment, objects cannot have inherited information about the behaviour or communication capabilities of each other. Instead, each object of a sovereign system must explicate its properties and negotiate about the federations to be entered.

Services and interfaces

Federated systems are basically founded on the concept of service. In order to facilitate federation negotiations, the views of the same service are separated based on whether a service is supported for others to use, or whether a service is requested. In addition to separating object views to the same service ideas, additional concepts are presented by the requirements of supporting modern communication technologies. Therefore, concepts are required, for example, for quality of service negotiation and stream interfaces. However, these concepts are not characteristic to federated systems alone, but to all modern systems.

Definition 2.13 *Service: Behaviour that can be invoked through an object interface. For a client, the service concept denotes a functionality that can be performed by its system environment. For a set of service providers, the service concept denotes a potential sequence of interactions that may be performed as a result of a defined signal from the client. The sequence of actions is governed by agreed policies on the joint behaviour of interacting object and the state of system environment at the time the service is performed.*

For traditional distributed systems we defined service (Definition 2.9) as a description of interface structure. Definition 2.13 captures the interface description as one of the items involved, but enhances the concept further to include aspects that are interesting for the organisations. The organisations consider the services also from the point of view of merchandises. Thus, quality of service, remuneration, and monitoring of the actual behaviour against the liaison become interesting. Furthermore, the Definition 2.13 requires that we promote interfaces to capture object behaviour in addition to the interface structure. Behaviour can have either operational or stream-exchanging semantics, as defined in ODP reference model (see Definitions 3.7 and 3.11).

Federation establishment mechanisms also require the refinement of interface concept. In the object models characteristic to distributed systems, the interface is the abstraction of functionality that is shared between the communicating objects. In a federated environment, the existence of such shared knowledge cannot be assumed. Instead, all objects have their private views to any interfaces and services. A commonly used interoperation scenario is that of a client and a server object. For those objects we can define a client-role interface and a server-role interface. When one of these interfaces is expected to receive a signal (e.g., operation invocation delivery), the other one should include statements of sending the corresponding signal. If the interfaces do not match perfectly, but are reasonably similar, the objects can still interoperate, but via an interceptor (see Definition 3.16) that transforms the signals while they are transported between the interfaces .

Definition 2.14 *Client role interface: a requester view to a service. A client role interface definition expresses the client's assumptions and restrictions on the service type and the access technology for a requested service.*

Definition 2.15 *Server role interface: a supporter view to a service. A server role interface definition expresses an implemented service type and required access technologies for the supported service.*

In order for the sovereign systems to negotiate about cooperation, the views to offered and requested services have to be captured. Therefore, we give two additional definitions:

Definition 2.16 *Service offer defines*

- *the type of server interface supported by a set of objects,*
- *the service properties in terms of quality of service, and*
- *the conditions under which the advertised properties are expected to be valid.*

Definition 2.17 *Service request defines*

- *the type of client interface supported by the requesting object,*
- *the service properties expected in terms of quality of service, and*
- *the requirements for the validity of the advertised properties in an acceptable service offer.*

All the above definitions use the term 'type' when referring to interfaces or services. We here use 'type' in the sense of predicate over an object. An object is of a given type if it fulfils the predicate, irrespective of how the object was created. In a federated environment, the service liaisons must be based on the type conformance of their interfaces. For the creation of objects a separate concept of template is used. A template is a type that is suitable for object creation in a given environment. For example, program code can be considered to form an object template. Object templates are private for each platform architecture; types are suitable to be used for global negotiation of services. We will return to these concepts in Section 2.4.3 and more formally in Section 3.2.

Interface instances are identified by interface references:

Definition 2.18 *Interface reference: An interface reference denotes an access point of a service. The interface reference structure is specified by the service type of the interface. Via an interface reference information about service type, location of the interface (e.g., communication address), and suitable access protocols can be retrieved.*

A more detailed definition is adopted from ODP reference model, see Section 3.2.

Binding liaisons for federated interoperation

The foundation for federated interoperation is the federated binding process that establishes liaisons. In this process, meta-information about services and supporting objects is exchanged by middleware functions between the sovereign systems involved. (The basic idea of liaisons is present in ODP reference model, see Definition 2.19 and the definitions in RM-ODP [92, clause 13.2.4].)

The federated binding process contains negotiation of potential contracts. We call the service requests and service offers jointly as potential contracts. A contract schema defines what kind of agreements the federating objects must reach before a binding liaison can be established, and thus defines also the required structure of potential contracts.

For each service type different properties are natural and therefore also the contract schema is different. In general, the following object properties are interesting:

- the supported services (e.g., data storage and retrieval),
- the plausible quality of service (e.g., delay, mean time between failures),
- failure recovery behaviour in case that the object experiences unacceptable quality of service from other objects (e.g., timeout and termination of a delayed operation), and also
- the expected communication facilities (e.g., access with NFS protocol).

The actual cooperation relationships between object instances are called binding liaisons. The liaison eventually determines what kind of communication channel can be created between the service interfaces.

Definition 2.19 *Binding liaison: Binding liaison is a context where the shared facilities supporting the client and server role interfaces have been selected and will be deployed.*

Binding liaison is a concept that does not appear in traditional distributed systems. In distributed systems, it is the responsibility of a client object or a client process to form and maintain a binding, i.e., be aware of the communication details with a server. In federated systems, the system services are responsible of the communication details, and an abstraction that captures information for communication realization is required. In distributed systems, the environment is in many aspects homogeneous, and the liaisons can be implicit or at least static. In federated systems, the same aspects are gathered as meta-information to the binding liaison, to be dynamically exploited, tailored, maintained and modified. Aspects of heterogeneity include differences in object or process management, and representation format of meta-information.

The service concept essentially captures both the application level interface (both structure and behaviour) and the platform facilities that are required for the communication between the service requester and the service provider. Therefore, we separate the concepts of service liaisons that represent agreements between application level object, environment liaisons that represent agreements between application and platform level objects, and infrastructure liaisons that represent agreements between platform objects of different systems. These liaisons are captured to Figure 2.1. A binding liaison between two application objects requires a consistent set of service, infrastructure and environment liaisons to exist.

Objects

The required service behaviour is finally implemented by some computing and communication activities. In an abstract way, the activity can be considered as a single, very large object; in a concrete way, the activity can be considered to be produced jointly by a set of (programming language) objects interacting with each other. These two levels of abstraction are illustrated in Figure 2.2.

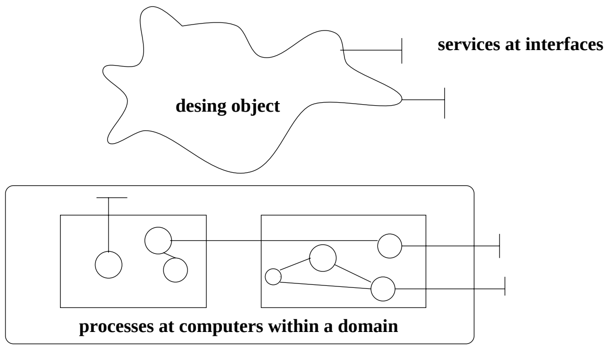


Figure 2.2: Modelling and realization of services.

Definition 2.20 *Object: Object is a modelling concept for real world entities like resources, threads of processing, abstract data structures, and programming language objects. Objects can be composite. An object has behaviour initiatable by requests on its interface and it may encapsulate information, entities, interactions among those entities, and self-induced internal activities.*

Only application level service packages, like teleconferencing services or bank interfaces, can reasonably be expected to interoperate in a federated environment. We exclude processes and programming language objects from the federation discussion, because the mechanism is far too heavy for such detailed integration. Objects of the same granularity are elsewhere called megamodules [251]. Also megamodules encapsulates – besides procedures and data – also types, knowledge and ontology (concepts and interpretation paradigm).

In contrast to programming language objects and processes of a computing system, the federated system objects can encapsulate not only data but also activities within some defined community of entities. This property allows federations to be build on application level services (banking services, teleconferencing) instead of restricting federation to platform level services (files, memory, processors).

Also in contrast to programming language objects, only the behaviour at the object interface is public – no public data is allowed. Instead, associated with every service is a set of public properties representing meta-information about the service. The meta-information covers technological restrictions for accessing the service, policies affecting the service, etc.

2.4.3 Middleware

For the federated systems, the middleware model presented by the ODP standards (see Chapter 3) is appropriate. The ODP middleware services form an abstract computing platform model onto which actual platforms can be mapped. In the federated system middleware implementations, threads of processing, data, and algorithms vary, and the variance is managed by using meta-information describing the components. More difficulties arise because also the meta-information representation forms vary between service implementations.

The interesting federated system middleware services are

- the establishment of conceptual liaisons between objects, and the actual dynamic binding process realising the liaisons; and
- the various meta-information exchange mechanisms, like trading and type repositories.

In the following we take a short look at the meta-information services. Part III elaborates these functions, especially the trading services in distributed and federated environments.

Trading

The trading function is used to mediate information about interfaces at run-time. It is a world-wide, federated mechanism that provides the means to advertise and to discover a particular service type [96]. The trading community includes 'importers', a 'trader', and 'exporters'. The client is an application object that wishes to find a server. The server represents a remote application object of interest. Neither of these objects need to be involved in the trading action themselves, but they have representatives, importers and exporters. The trader stores information about available servers as 'service offers'. Service offers are grouped so that each set of service offers represents providers of one abstract service type. Each service offer includes an interface reference that conveys where and how the service can be invoked, and a set of property values that convey other aspects of the service, such as quality of service. The properties can be either static (e.g., processor type) or re-evaluable at usage-time (e.g., queue length). The service offers can also convey policy choices of the objects, e.g., what kind of search algorithm is used in a database.

Trading itself is just a mechanism. The term only defines the operations available and the syntactical structure of the information controlled by the operations. In order to create an interesting trading service, the semantical contents of the controlled information must also be specified.

The trading function standard is already relatively stable [96]. Research projects studying trading are numerous: ANSA [44], BERKOM Y [254], COSM TRADE [149], RHODOS [166], MELODY [26], DRYAD [126], etc. Commercial products have also been already developed, for example, traders appear within ICL DAIS, Bellcore INA, etc. For CORBA, products have been offered from (at least) ICL, BNR, APM, DSTC [173]. We return to the comparison of the trader specification and implementations in Chapter 8.

Type repository

The binding process requires a consistent view to available interface types. Otherwise communication between objects cannot be guaranteed to be type safe. Type safety means that the sender and the receiver process are guaranteed to agree on the structure and semantics of an exchanged message.

The type repository function can be used to mediate the type information and to translate it to template information. The type repository function [97] resembles the trading function by its behaviour, but instead of interface instances it mediates type information. The key concept for type repository is 'type description'. A type description expresses an abstract service class by presenting a set of concrete expressions for it, in different languages. Thus, the type description contains a set of 'type definitions'. The different expressions allow replacing one type definition by another. This is especially important when a declarative type expression can be related to a template. A supplementary concept for type repository is 'type relationship' that is used for expressing conformance between type definitions administered separately. For the federation of the systems the relationship is essential: conformant type descriptions are interchangeable (or interceptable).

The type repository function for the ODP family of standards is under development [97]. Currently, this work is being merged with OMG Meta Object Facility, MOF [174].

Naming

Both the trading function and the type repository function trust in a federated naming service. Several solutions have been suggested, for ANSA [243], for CORBA [177], and for general cases (for example, [164, 14]). The ODP naming framework standard is still under development [98]. It defines a federation mechanism between global naming systems with independent management. However, it requires a globally distributed management of a single naming system and does not exploit federation facilities on this aspect [110]. The scalability problem created is clearly visible when the amount of nameable entities is increased [40].

Binding service

Communication between objects can only occur after a successful binding process between the involved interfaces. As a result of the binding process a binding liaison is created. The liaison ensures that a communication channel is either already created or the infrastructure is prepared to create the channel. The channel creation can be delayed due to optimisation of resource allocation until the channel is actually used for the first time. The characteristics of federated channels are discussed in Section 2.4.6 and in Chapter 9.

When a binding between a client and a server interface has been requested, the binding process should ensure that

- the interfaces exist or can be instantiated,
- the interfaces are compatible, and that
- there is no (security related) prohibition for the binding.

The compatibility requirements for interfaces capture

- conformance of interface signature types,
- conformance of interface behaviour,
- availability of a shared data transport mechanism,
- shared understanding of the names used for types, behaviours, and protocols in the binding process, and
- suitable combination of the roles for objects involved in the communication (e.g., producer and consumer, client and server).

As the result of the binding process the agreed values for the binding liaison have been decided for each contract schema item. Even within these values, some alternatives can be present. In such a case, the liaison is able to adapt to changes in the binding during its lifetime. Especially, the infrastructure liaison can be changed while the service liaison is persistent. A channel may, for instance, recover from a too high jitter by changing the transport protocol.

The nature of bindings between interfaces differs from that in distributed systems. In distributed systems, the binding process makes the communicating parties aware of each other's location directly so that they can start messaging with each other. In federated systems, there are two levels of object interfaces (computational and engineering levels) reflecting the application and platform level objects (see Figure 2.1). In federated systems, we want only the computational binding to be visible to programmers, although technically, the engineering interfaces must be bound.

The federated binding service (that we discuss in further detail in Chapter 9) follows the requirements of distributed binding specified in ODP reference model [93] and the more specific binding framework [102]. However, the federated binding model is more specific in aspects that involve sovereign systems, for example, in procedures for liaison maintenance.

Factories

Factory is an abstraction of a process that instantiates objects and object configurations, such as interfaces and channel components. A factory is realized for example by an operating system nucleus that supports operations for process creation from a program file.

A type-parametrisable factory is a factory that is able to select the necessary templates itself, when it has been given a type for the outcome. A type repository can be used for mapping types to suitable templates at each real platform.

The type-parametrisable factories do not resemble generic types supported by some programming languages (e.g., Ada). Generic types manage a form of inheritance and they are used for maximising code reuse. The type-parametrisable factories map shared knowledge to local technologies. The factories cannot use any form of inheritance for their cooperation. The factories manage a form of polymorphism.

The ODP reference model does not discuss factories, but discusses instantiation facilities instead.

2.4.4 Meta-information

The essential meta-information in federated systems is captured by contract schemata, and therefore, we discuss separately the major contract components. These include interface type or service type, quality of service attribute values or thresholds, failure detection and recovery protocols, and conditions on which the service and the quality of service agreement can be held.

Because meta-information related to types and templates has an essential role in federated systems, we first study type and template hierarchies. Types are necessary for the establishment of binding liaisons; templates are required for implementing the agreed service. Second, we are interested on items such as policy frameworks on service behaviour and quality of service contracts, within the service type specifications. Finally, other meta-information classes are briefly discussed. More detailed discussion on meta-information and middleware services follows in Chapters 6 and 9.

Service types and related templates

Successful communication between objects requires that the object interfaces share an information exchange structure (information items revealed and expected) and a semantics for the joint behaviour.

A service type is an abstraction that denotes a known behaviour pattern. A service type is represented as a description of a functionality and its access method through an object interface.

It can be either expressed as a service type name, or described by an interface signature together with a behaviour description or a behaviour name. An operation interface signature expresses names of operations available at the interface together with their parameter names and value types. A stream interface signature expresses names and protocols of the flows within the stream. Service type also lists quality of service concerns, for example, where to use the service, on what platform, and what kind of remuneration policy is used.

A service type can be captured as a set of alternative interface types. For example, the accounting service can be implemented in two ways: First, it can be represented by a single interface with two operations, one for receiving log entries, and another for producing the report on request. Second, it can be represented by two separate interfaces, one for each operation.

An operational interface type is defined by a set of operation types. Operations can be expressed in various languages, for example in IDL, ODL, or C++. However, an interface type expression may be valid as a template at some platforms. Most distributed object management systems conceptually separate object methods from operations. Operations included to the interface type are then further mapped to methods. The platform aspects of a service type can also require a concrete expression format. Two service types can differ based on platform or representation aspects alone.

For the realisation of communication in a federated environment the types must be mapped to structures that can be executed. Therefore, we study interface templates that invoke object behaviour and refer to information exchange structures. An interface template is an interface specification that is usable at a specific platform: the platform must have a suitable invocation scheme for the operations listed. Most object-oriented programming language environments map operations directly to executable methods.

Interface specifications have a dual role as types and templates. The term 'interface' is therefore often used alone, not specifying whether a type or a template is denoted. This causes confusion, because the assumed level of abstraction is not expressed. In type specifications, operation parameters denote information to be exchanged; in template specifications, operation parameters denote the format and grouping of data to be transferred.

Both individual type specifications and template specifications may have a hierarchy for defining similarity or inclusion of properties. In case of types, this hierarchy expresses the capability of an object of one type to replace an object of another type. The similarity hierarchy can be formed at two levels of abstraction: either at service type level, or at interface level. In case of templates, the hierarchy is an inheritance hierarchy. The hierarchy denotes inclusion of object properties to a new object. The new object can be modified with some additional or replacing properties. The main goal is to reduce maintenance work in software production. Therefore, the inheritance hierarchy is only about the interfaces.

Unfortunately, both hierarchy structures are called subtyping, although the rules differ. The essence of subtyping rules for objects in a federated environment (and in ODP systems) is contravariance [93, 28, 27]: the offered information flows must include at least the information expected by the receiver. In other object models, subtyping is in most cases based on covariance: the replacing object can both expect to receive and offer more information that the replaced object expects and offers.

In a run-time environment, the type and template hierarchies are mapped together. Although inheritance usually leads to a subtype relationship, this is not always true. Therefore, both hierarchies must be maintained. In addition, a mapping must be maintained from each type to those templates that realize the type.

The general type concepts discussed here are called the target concepts of the type system. Different type systems may have different sets of target concepts. The inheritance or similarity hierarchies are constructed among the members of the representatives of each target concept separately. As a summary, the abstraction levels can be described as in Figure 2.3. The figure illustrates the containments and mappings of types on the left side, and the containments and mappings of templates on the right side. The arches joining types and templates denote a possibility of a template to realize or implement a type.

The usage of type information differs in traditional distributed systems and federated systems. Type information services in distributed system environments manipulate interface descriptions to be used mainly by design and implementation time tools. For example, the interface repository in CORBA model is such a service. In a federated system environment, the type repository function has run-time related tasks. Type services for federated environments support trading by providing information on service types and related properties, and transformations between interface implementations.

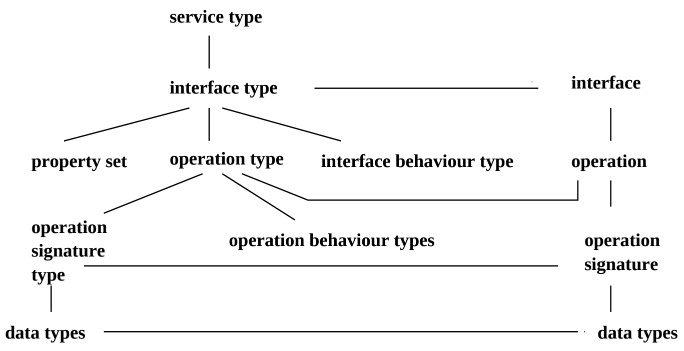


Figure 2.3: Target concepts for a type system.

Co-behaviour and policies

During the negotiation of binding liaisons, the co-behaviour of two or more independently administered objects need to be expressed. Currently, no acceptable method is available for representing object behaviour in a general form.

There are two separate behaviour aspects involved: first, the overall service semantics associated with an interface needs to be expressed; second, the object performing a service is governed by an administrative policy. The overall service semantics is a static property of the interface. To clarify the concepts, we can consider an interface that supports a printing service. The administrative policy restricts the overall behaviour of all objects under that administration. For example, all clients are forbidden to use printing services that cost more than 10 FIM per document. The service properties of the interface can be separately regulated by the administration, even changed during the interface life-time. For example, the price of an offered printing service may change.

The co-behaviour of federated objects is governed by the administrative policies on each domain involved. For example, a document can be printed on a named printer only if the price of

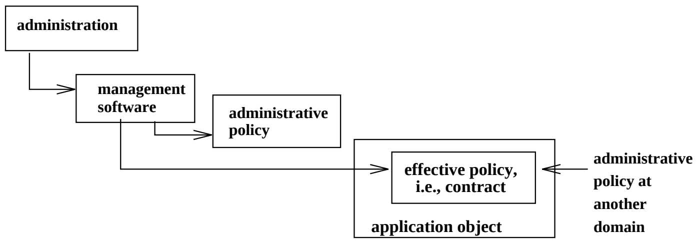


Figure 2.4: Restricting co-behaviour by administrative policies.

the job is under 10 FIM and the client has a billing contract with the printing service provider. The restricting rules of a service execution are inherited from the administrative policies of all involved domains, as illustrated in Figure 2.4.

As a current solution, we suggest to use behaviour names: We can name general service patterns and a set of acceptable variations of each pattern. The characterisation of such a behaviour pattern with its variations is called a policy framework. We require that the federating objects adopt their behaviour within standardised policy frameworks. Often, the policy that governs object behaviour is implicitly encoded to the object implementation, but even then, the specific features of the behaviour can be named. For example, a trader that supports dynamic property values, must be able to serve a query operation that denies the use of dynamic property evaluation [96]. The trader policy framework includes in this respect two alternative behaviours: evaluating or ignoring a property value.

Practically, this means that a general service name and a more detailed feature name can be together used as a description of the object behaviour.

In some cases, the policy rules are not related to qualitative behaviour changes but to quantitative parameters. For example, in the above printing example, we referred to printing prices. Such values should be captured as properties with standardised names and standardised value sets. Although individual sovereign systems would use different naming or value sets, interceptors could be used in federation establishment. A wide range of suitable standards already exists, including date and currency formats.

The variety of languages for expressing administrative policies does not form a specific federation problem, because detailed object management policies need not to be expressed in the binding liaisons. The federated architecture model separates federated control of object co-behaviour from traditional management activities, such as resource control [127]. Therefore, only a few aspects of a service behaviour need to be expressed in the binding liaisons.

Quality of service

The concept of quality of service (QoS) is essential in many modern system architectures, especially, in those supporting multi-media or telecommunication applications.

Traditionally, QoS refers to the properties of data transport in terms of throughput, jitter, and dependability (rate of lost or damaged messages). In this case, the QoS management is simple: the QoS requirements are used as parameters for the data transport protocols and when the transport layer fails to fulfil the requirement, no corrective actions are made at run-time. The QoS management is considered to be a design and installation time activity, requiring suitable capacity planning [38].

In modern systems, QoS concepts have been enhanced to cover QoS contract negotiation and dynamic QoS management. Also, the variety of QoS characteristics has been enlarged to cover more aspects of objects and services in general. QoS attributes can differ depending on the service type and whether they are associated with data, service, or objects [103].

The modern QoS management model [103] includes statements concerning QoS requirements, QoS offers, QoS contracts and QoS observations. The QoS requirements can be produced during the system design or during the system operation. The QoS requirements have been captured as part of service request in Section 2.4.2. The QoS requirements state the expectations of an object towards its environment, i.e., the infrastructure and the server eventually selected to perform the requested service. The QoS offers are used for advertising QoS capabilities of object configurations. There can be a separate refinement mechanism that expresses how the individual objects contribute to the offered QoS measure. However, there is no requirement for the offered QoS to be directly derivable from the actual object capabilities. The QoS contracts include statements resulting from a negotiation process over the QoS characteristics. The QoS contracts can be created either a dynamic negotiation mechanism or be implicit in the system design. The QoS observations denote the information created by monitoring the object configuration governed by a QoS contract.

The QoS requirements, QoS offers and QoS contracts have been captured as part of service requests, service offers, and service liaison (see Section 2.4.2). The QoS observations can be used in failure detection and recovery protocols for service liaisons, as explained in Section 2.4.2.

Examples of QoS measures include

- jitter of analogous data flow (e.g. a video signal),
- throughput (e.g. of a telephone line or a processor),
- timeliness (e.g. in real-time systems or in analogous signalling),
- service availability (e.g. probability of a disk being accessible, or a properly defined name to be resolvable at a given time),
- time-to-live (e.g. validity time of a name-address pair in name server), and
- immutability (e.g. promise of not changing a name-address pair after storage) [103].

Examples of communication agreements that can be based on QoS arguments can be found from multi-media applications [38]. Three kinds of service liaisons are frequently used:

- agreement on 'best effort QoS level', which gives no assurance nor remedy activities,
- agreement on 'threshold QoS level', which monitors the service performance, but notifies the client if the QoS drops below the negotiated level,
- agreement on 'compulsory QoS level', which gives no guarantee on performance, but the service is aborted if the measured quality does not meet the negotiated level, and
- agreement on 'guaranteed QoS level', which monitors the service performance, but does not abort the service if the QoS drops below the negotiated level.

In federated environments, the QoS contracts (i.e., the service liaisons) must be expressed and negotiated in such a manner that the sovereignity of the involved objects is not violated.

2.4.5 Model for software composition

Adoption of the federated architecture affects the style of constructing application software. In this section we study the differences.

A federated application architecture model does not federate processes nor computing systems, but instead, entire applications, like teleconferencing, and billing. Still, the software is constructed by current programming tools, like object-oriented programming languages. There are only a few additional rules to be applied with object-based programming:

- Each application must have an interface suitable for federation.
- At the federation interface, no shared memory nor referential parameters are allowed.
- The federation interface must be described in some interface definition language and the definition registered to a local type repository.
- The interface reference for the federation interface must be exported to a local trader.
- Communication between processes within a distributed system should utilise the federation-based communication scheme as well.

The federated system model allows objects to have multiple interfaces. Reasons for the multiple interfaces are various: aspects for extending object to support multiple independent roles [200, 207], to meet liaisons [73], and to allow combination of partial classes as in subject-oriented programming [70].

When objects are allowed to have multiple interfaces, the expressive power of objects increases in comparison to traditional procedural programming. Objects that have multiple interfaces and may request further information during a method execution by interacting with their environment are called 'interactive objects'. For example, a user interface object that is able to seek further advise from a user, is an interactive object. Introduction of interactive objects – that are able to make 'intelligent' decisions based on their environment conditions – present a major paradigm shift from procedural programming [249]. Interactive objects can be considered intelligent because they can retrieve information not only from the operation invoker but also independently from their real environment. A characteristic of object systems is that the observable behaviour of the joint system appears to be nondeterministic and cannot be described by sequential algorithms. The object's behaviour is not merely algorithmic, but is dependent also on the object environment and internal state that reflects its past history.

The paradigm shift is fundamental, because interactive objects are more powerful as a concept than the Turing machines [249]. Turing machines can take input, perform some computing, and produce output. However, the Turing machines do not allow external information to affect the computing phase.

The paradigm shift has important consequences for system design and behaviour: it introduces unavoidable elements of incompleteness and unpredictability to the systems. The goal of specifying a complete system behaviour should be rejected, and replaced by partial system specifications through interfaces and views [249]. Each partial specification has two roles: they essentially constrain the discouraged system behaviour and at the same time describe the useful system behaviour. A standardised framework for such partial specifications can be found in the RM-ODP, as described in Chapter 3.

2.4.6 Model for channel composition

Adoption of the federated system model changes the responsibilities of the platform services. Instead of instantiating a static channel structure, the platform services must be able to generate variable channel structures based on the binding liaisons.

A channel is a configuration of intermediate objects that are able to route signals (operation invocations, terminations, flow messages) from one application object to another. The end-points of the channel are determined either by the binding initiator or the binding process itself, i.e., the interfaces to be interconnected can be either identified or searched based on their properties. This also means, that the computational interfaces are bound together, instead of creating a channel between the technical addresses at which the interfaces are initially located. A channel does not necessarily form a static circuit through the network; a channel can be based on connectionless protocols.

A generic channel structure (that equals the generic channel structure in RM-ODP, see Definition 3.15) is represented in Figure 2.5. The actual channel structure varies depending on which distribution transparencies are selected by the user, and which communication protocols are in use. The actual channel structure varies also depending on the platform architecture and administrative rules on the systems that support partial channels.

In a federated environment, the channel creation process is not under a single controller, but distributed. Therefore, Figure 2.5 shows two channel halves: each section of a channel is instantiated independently of the other endpoints of the channel at different administrative domains. The instantiation process only uses the information captured to the binding liaison and interface references, including channel type and channel parameters like location.

The overall responsibilities of the channel components are traditional:

- Binders monitor the transport service for failures (crashes and breach of QoS contracts), and initiate recovery from the failures, potentially recreating the lower layers of the channel using different protocols. Binders can also act as binding membership controllers that allow communicating partners to join or leave the service liaison.
- Representation stubs marshal and unmarshal data, also when representation format is not equal in both ends of the channel. Persistence stubs check-point the communication, and support migration of the interface so that either a proxy interface is left back or a re-locator is informed of the new location. Transaction stubs execute the transaction protocols.
- Protocol objects transport data across the network. Different kinds of protocols are used for streams and operations. The protocols vary also depending on selected transparencies, abilities of the platforms, security requirements and transport line properties. The negotiation is done at the phase where object interfaces are matched.
- Interceptors make security related decisions, and transform information representation or grouping.

These components appear also in traditional RPC realisations. However, in federated system channels, the stubs can be selected at run-time, instead of compilation time as in many RPC implementations. Also, several stubs can be active concurrently, using the same protocol link underneath. Separate concurrent protocols for channel components are, for instance, group management [181], and QoS management [247]. In a general case, the stubs are not self-sufficient, but require services from management functions like authentication services [116].

Additional objects involved in the channel structure include channel controllers. A channel controller allows the channel configuration and parameters to be modified during the service liaison duration. Changes can involve, for example, multi-cast group membership or timeout values when a fixed network line is switched to a mobile network line. A channel controller is a direct client to all of the channel component's management interfaces, and therefore it offers a combined control interface to all of them. Because of its many connections, the channel controller has not been illustrated in Figure 2.5. The channel controller has a specific object at each domain, and those objects may cooperate in order to offer a joint binding liaison management service.

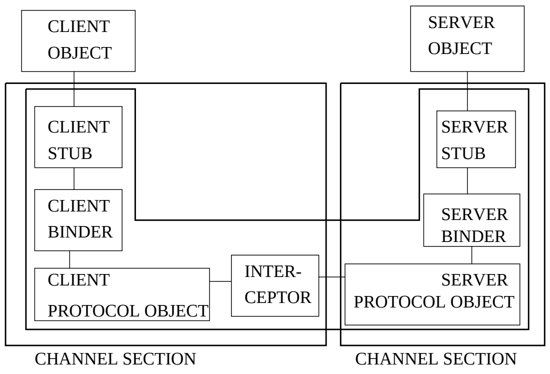


Figure 2.5: Channel structure.

2.4.7 Benefits

The federated system architecture is suitable for world-wide interoperation between independently developed systems. The architecture covers aspects of evolution, automation of cooperation relationships, and controlled access. Most importantly, the federated system architecture also preserves the sovereignity of systems in their cooperation. In addition, the architecture is scalable in respect to the number of data items passed through the network.

The scalability and the system sovereignity are reached by focusing on abstract services instead of low-level objects. The style of modularisation minimises the need of shared control or shared data between independent systems. For example, a shared file system is not necessary between independent systems, if files can be federated. The federated files can be manipulated in a manner similar to trader federations described in Section 5.3. The mechanism is based on caching mechanisms and a set of polling and announcement protocols for change management. However, data origin and data ownership are in all cases respected.

In a world-wide community, the services at each federable system inevitably evolve asynchronously. New services emerge, old services are no more used, and old services are modified to include new features. The federated system model has an embedded evolution mechanism, provided by the meta-information exchange functionality. The availability and the properties of services can be negotiated automatically at the system operation time. Furthermore, the meta-information services are neutral in the sense of information exchanged. Commercial interests are not affected by the mechanisms, because users of federated services can easily give preference to products with certain extra features. Even the traditional distributed systems can benefit from the meta-information services by employing them for software portability support and for simultaneous support for multiple versions of same services.

The object model used in the federated system architecture allows a fast and distributed software production process to be established. The federated architecture model also addresses the needs of modern systems, in areas of new technology and quality of service management.

The architecture adopts multiple technologies, such as streams for multi-media applications, and various data transport protocols to allow service mobility. Separation of systems is also naturally visible to end-users as the failure model of the communication primitives can be expressed in a considerably precise manner.

Adoption of quality of service as a run-time issue gives new tools for controlling the correct behaviour of a computing system. Recently, an interesting failure related to run-time QoS control was reported in Finnish newspapers [236, 143]: The railway traffic-control system controls the movements of trains and spaces them via semaphores so that there is only one train per section of a railroad. Because of its critical nature, the system has replicated hardware. The secondary computer is designed to receive and store replica data with a certain frequency. However, one day, a paper clip had jammed the space bar at the keyboard of this secondary computer, causing enormous amounts of interrupt signals. The computer was too busy to receive data from the primary system at the designed speed and therefore caused long message queues at the primary system. As a consequence, the primary system could not process new train information at its normal speed, but kept the trains safely but inconveniently far apart from each other for several hours. With QoS monitoring, the problems could have been avoided.

2.4.8 Problems

The overall design of federable systems covers several aspects still requiring further development:

- standardised contract schemata for formulating liaisons;
- creation of meta-information together with the actual processing entities and interfaces;
- common platform services for exchanging meta-information about the interfaces and the supporting services;
- a binding protocol for establishing binding liaisons and realising them as channel structures;
- self-healing communication channel structures; and
- factories for instantiating objects (processes) for a given type (that is replaced by a locally relevant template).

The major problem of federated system architecture is its immaturity and lack of practical implementations. There are not yet standards for contract schemata for service types, although the work has already been started. There are no widely used tools for developing federable software. The standards allowing federated system architecture to be exploited, are still under development and thus too vague for vendors to sell products based on them. However, for example within OMG, vendors have already started developing meta-information exchange services for their platforms.

An outstanding problem in federated systems is the security of federated communication. The current networks are easy to join and there is, so far, no mechanisms for ensuring that the joining system can be trusted.

Chapter 3

The ODP reference model

In this chapter, we discuss the Open Distributed Processing Reference Model, RM-ODP. In the previous chapter, the federated system model is described based on a systematic interpretation of ODP concepts. Therefore, we claim that the federated system model is consistent with the ODP reference model, but not the only acceptable interpretation. This chapter aspires to an unbiased tutorial on ODP reference model to support the consistency claim and to fill in some of the referred definitions. In addition, we indicate those ODP term definitions that were enhanced for the federated system architecture in Section 2.4.

3.1 Overview

The open distributed processing reference model, RM-ODP, captures the basis for the evolution of open systems. It is a joint standardisation effort of ISO [91] and ITU [107], started already in 1989. The ODP reference model is a family of general standards. Therefore, it can be utilised for specifying distributed systems [52, 190, 202], as well as federable systems. The model is targeted for architecture designers, standardisation bodies, and vendor consortia, to guide them in the design of software architectures. The ODP model is being developed in interaction with other consortia working on distributed system architectures, including OMG (founded in 1989) and TINA-C (founded in 1992). Therefore, the abstract infrastructure model of ODP can serve as a long-term prediction for software markets in future.

The aim of ODP standardisation is the development of standards for distributed information processing systems that can be exploited in heterogeneous environments and under multiple organisational domains. In addition to the requirement of using published and standardised interfaces, ODP systems are required to

- provide software portability and interoperability;
- support integration of various systems with different architectures and resources into a whole without costly ad-hoc solutions;
- accommodate system evolution and run-time changes;
- federate autonomously administered or technically differing domains;
- incorporate quality of service control to failure detection mechanisms;
- include security service; and
- offer selectable distribution transparency services [94].

These goals are acquired by the ODP model through three already standardised aspects of the basic reference model (completed in 1996 [92, 93]):

- a division of an ODP system specification into viewpoints, in order to simplify the description of complex systems [93],
- a set of general concepts for expressing the specifications [92],
- a model for an infrastructure supporting, through the provision of distribution transparencies, the general concepts that it offers for specification purposes [93].

The ODP standardisation work is being continued by the specification of essential middleware services as component standards. These services include trading service (completed in 1996) [96] and naming framework [98]. Components that are currently under work include type repository function [97], and interface binding framework [102] together with the supporting protocols [99]. At the same time, an open system management architecture (ODMA) [90] is being developed, although the current concepts are still more OSI related [86] than ODP related. In addition, further development is started on viewpoint languages.

The ODP family of standards also includes formalisations of the fundamental concepts [95]. The formalisation work has been hindered by the inadequacy of current formal languages. For example, concepts that require dynamicity, autonomy, and heterogeneity are difficult to express. The formal aspects of ODP modelling are not discussed in this dissertation. Formal approaches can be found for example in [208, 163]. However, further work is required in this area.

In the following sections, we study the ODP basic concepts, the viewpoint specification rules for object systems, and the abstract infrastructure services identified in the RM-ODP to support the specification and implementation of open object systems.

3.2 The ODP concepts

The basic ODP concepts include terminology on objects and their communication, and terminology on organisation of objects to various structures. In addition, terminology is given for explaining the scope of ODP specifications through conformance statements. In the following subsections we discuss each of these areas in turn.

The ODP object model differs from object models represented for object-oriented programming languages, object-oriented design methodologies, and distributed object management systems. The major differences include

- the flexible use of the object concepts for a variety of abstractions, ranging from implementation details to abstract business model considerations,
- the use of multiple interfaces,
- separation of client and server sides of the interface through which the objects communicate,
- variety of mechanisms for introducing objects to a system,
- concepts related to the type safety of communication through interfaces, and
- interactions between objects are not constrained – they can be asynchronous, synchronous or isochronous, and atomic or non-atomic.

The communication model between ODP objects improves the communication primitives offered to programmers by introducing selective distribution transparency. The communication model is realisable through the binding model of objects.

Characteristic of the ODP reference model is the use of structuring concepts of community, domain and federation. These concepts can be considered to be either static concepts exploitable at design time, or dynamic concepts exploitable at operation time.

The conformance statements are mainly required by standardisation activities: further standards must claim the suitable conformance testing methods. However, the classification of reference points restricts the areas of conformance testing in such a way, that all implementation decisions cannot be revealed.

ODP object model

The ODP object model covers the terms of

- object,
- interaction point for specifying the object location,
- object behaviour for specifying behavioural compatibility required by the interface binding process, and
- object types and object templates for selecting and instantiating objects.

An ODP object is an identified abstract entity that represents the properties of a real-world phenomenon. The state of the object is considered to be the information contents of the object at a given instant in time. This information content determines all possible interaction sequences that can be started from that state. Therefore, the object state and the future behaviour of the object reflect the same abstraction.

Object is defined by RM-ODP as follows:

Definition 3.1 *"Object: A model of an entity. An object is characterised by its behaviour and, dually, by its state. An object is distinct from any other object. An object is encapsulated, i.e. any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment.*

An object interacts with its environment at its interaction points.

Depending on the viewpoint, the emphasis may be placed on behaviour or on state. When the emphasis is placed on behaviour, an object is informally said to perform functions and offer services (an object that makes a function available is said to offer a service). For modelling purposes, these functions and services are specified in terms of the behaviour of the object and of its interfaces. An object can perform more than one function. A function can be performed by the cooperation of several objects" [92, clause 8.1].

This definition is very general and allows objects to represent entities of arbitrary granularity. It also allows the term to be used differently in the five viewpoint languages. For example, a complete information system is represented as an object in a typical enterprise viewpoint specification. At the same time, a small component participating communication protocols inside the system can also be represented as an object when specifying the implementation of the system [94, clause 7.1.1]. The Definition 2.20 on sovereign objects in federated environment is conformant with the ODP object model.

Objects interact at interaction points. An interaction point is specified both in time and in space. Several interaction points may exist at the same location, and an interaction point can even be mobile. Furthermore, a logical interaction point may be technically distributed, because by extension, the location of an object is the union of the locations of the actions the object may participate [92, clause 8.10].

The nature of interaction point is essential for exploiting design level objects in the open system architecture. It leads towards separation of computational interfaces and engineering interfaces in such a way that computational interfaces would not manifest distribution but allow a set of corresponding engineering interfaces to be allocated conveniently. The distribution and the mobility of interaction points have no fundamental uses in distributed systems, although many applications can be more easily modelled this way.

In object specifications it is often convenient to concentrate on the object behaviour. The ODP object behaviour is defined through the concept of activity, a sequence of internal actions or interactions with other objects:

Definition 3.2 *"Behaviour (of an object): A collection of actions with a set of constraints on when they may occur" [92, clause 8.6].*

Definition 3.3 *"Activity: A single-headed directed acyclic graph of actions, where occurrence of each action in the graph is made possible by the occurrence of all immediately preceding actions (i.e. by all adjacent actions which are closer to the head)" [92, clause 8.5].*

An object is said to be behaviourally compatible with another object, if the first object can be replaced with the second object without the environment to be able to notice the difference on the basis of type criteria. The criteria may allow the replacement object to be derived by modification of an object. If such a modification is necessary, the behavioural compatibility is called coerced behavioural compatibility, otherwise it is natural behavioural compatibility [92, clause 9.4].

Objects can be classified using the concepts of types and templates, that we have already informally used in Section 2.4.2.

Type is defined by RM-ODP as follows:

Definition 3.4 *"Type (of an <X>): A predicate characterising a collection of <X>s. An <X> is of the type, or satisfies the type, if the predicate holds for that <X>. A specification defines which of the terms it uses to have types, i.e. are <X>s. In RM-ODP, types are needed for, at least, objects, interfaces and actions.*

The notion of type classifies the entities into categories, some of which may be of interest to the specifier" [92, clause 9.7].

The concept of type is very general and can be specialised in many ways to form various type hierarchies. Types are used for reasoning and verifying properties of objects, i.e., in the trading and binding processes.

Templates are more constructive by nature and are used for instantiation. The concepts of templates and types are independent of each other: having a common template does not induce being of the same type, and especially, being of the same type does not induce a shared template.

Template is defined by RM-ODP as follows:

Definition 3.5 *"<X> template: The specification of the common features of a collection of <X>s in sufficient detail that an <X> can be instantiated using it. <X> can be anything that has a type.*

An <X> template is an abstraction of a collection of <X>s.

The definition given here is generic; the precise form of a template will depend on the specification technique used. The parameter types (where applicable) will also depend on the specification technique used.

Templates may be combined according to some calculus. The precise form of template combination will depend on the specification language used" [92, clause 9.11].

The relationship between objects, types, and templates has been interpreted slightly differently in different contexts [97, 174, 111]. Therefore, we study this relationship more closely.

In ODP, there is a clear distinction between the instantiation mechanism of objects through templates, and the similarities in appearance of existing objects through types. The ODP framework also specifies an abstract computing engine that supports communication and service invocation between objects in the global system. This engine is primarily concerned with types, not by templates. However, the construction of an operating environment requires templates. The two aspects are reflected in the separation of templates and types in Figure 3.1.

Figure 3.1 shows three central concepts – object, object type and object template – and relationships between them. First, an object can be composed of other objects. Second, an object has a number of properties that characterise it in technical terms or express the function of the object. The object itself does not need to present these properties, but the property values form separate meta-information related to the object. The properties can be used for expressing predicates over a set of objects, i.e., defining object types. Third, an object can be instantiated from a template. The template may include variables (i.e., attributes) that determine some properties of the new object. These attributes are not necessarily the same properties as used as meta-information properties.

The object concept can be applied also to services, behaviour, and interfaces.

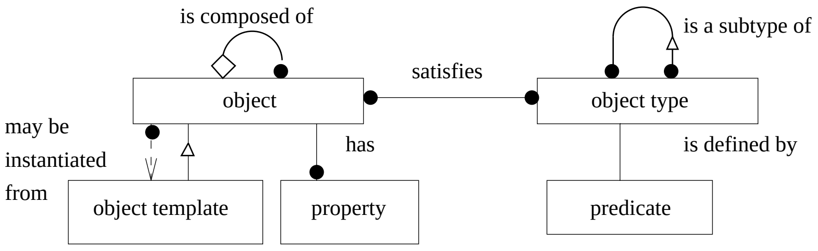


Figure 3.1: Relationships of objects, types and templates.

ODP communication model

The ODP communication model defines terminology for object interfaces and interface references. In Section 2.4.2 we have informally referred to these concepts.

An object can offer services through several interfaces that describe the actions in which the object can be involved. An interface is either an operational interface or a stream interface. An operational interface involves a set of operations, rather like remote procedure calls. However, operational interfaces are enhanced with notions on distribution transparency and quality of service attributes. A stream interface describes a set of continuous data flows, like audio or video flows. The activity associated with the stream interface is supported by a continuous data transfer protocol. Naturally, transparency and QoS aspects are relevant for stream interfaces as well.

The interfaces are abstract. An interface description has in principle two parts [93, clause 7.2.3]:

1. A structure for invoking a functionality at the interface and a set of structures for terminating it. For example, sending an abstract request message to start an operation and a set of potential replies to it. In other words, the interface structure defines input and output information of the operation. The interface structure is often expressed in languages like CORBA IDL (Interface Definition Language).
2. A service description. Currently only a name for an external and abstract behaviour can be reasonably used for this purpose. The behaviour descriptions are compared in the binding process, but the comparison is not computable in a non-restricted form. Problems arise from the large number of commonly used languages and the variety of algorithms suitable for implementing an abstract service. Therefore, for example, specifications expressed as programs are not practical.

The RM-ODP defines interfaces as follows:

Definition 3.6 *"Interface: An abstraction of the behaviour of an object that consists of a subset of the interactions of that object together with a set of constraints on when they may occur.*

Each interaction of an object belongs to a unique interface. Thus the interfaces of an object form a partition of the interactions of that object" [92, clause 8.4].

Operation interface is defined by RM-ODP as follows:

Definition 3.7 *"Operation interface: An interface in which all the interactions are operations" [93, clause 7.1.7].*

Definition 3.8 *"Operation: An interaction between a client object and a server object which is either an interrogation or an announcement" [93, clause 7.1.2].*

Definition 3.9 *"Interrogation: An interaction consisting of*

- *one interaction – the invocation - initiated by a client object, resulting in the conveyance of information from that client object to a server object, requesting a function to be performed by that server object, followed by*
- *a second interaction – the termination – initiated by the server object, resulting in the conveyance of information from the server object to the client object in response to the invocation" [93, clause 7.1.4].*

Definition 3.10 *"Announcement: An invocation initiated by a client object resulting in the conveyance of information from that client object to a server object, requesting a function to be performed by that server object" [93, clause 7.1.3].*

Interrogations are allowed to have multiple terminations. This is not only a consequence of symmetrical definitions. Allowing multiple terminations allow legal terminations and exceptions to be handled in a similar way. In addition, in a complex system, a third category may arise: the termination may be exceptional but still the results of the operation are usable or the computing can otherwise be reasonably continued. Moreover, in a heterogeneous environment, the interfaces may be constructed to accept multiple correct terminations with equal status.

Stream interface is defined by RM-ODP as follows:

Definition 3.11 *"Stream interface: An interface in which all the interactions are flows" [93, clause 7.1.8].*
Where

"Flow: An abstraction of a sequence of interactions, resulting in conveyance of information from a producer object to a consumer object" [93, clause 7.1.5].

The streams are not yet defined in more detail within the RM-ODP or the component standards. However, work on quality of service aspects and binding framework shall address stream interfaces. The TINA-C and ANSA work are most probably used for building a basis for stream definitions [182, 37, 241].

The ODP reference model does not differentiate between interface instances, interface types, and interface templates, which causes differences of interpretations. In the federated system model, we have consistently preferred interface types where multiple interpretations have been possible.

Interface instances are identified by interface references. Interface references are described in RM-ODP [102, clause 8.4.2] as follows: *"Interface references are unambiguous identifiers for the interfaces they reference. The property of unambiguity arises from the complete collection of information in the interface reference structure, rather than from any particular field or fields within it. In general, the representation of an interface reference is specific to an engineering domain, in which there is one appropriate naming authority and binding is managed in a uniform way."*

The description continues by enumerating and defining the following data structure components [102, clause 8.4.2]:

1. Interface type is denoted either as a direct or indirect type name;
2. Channel class identifies a channel template either with a direct or indirect channel name;
3. Location information provides the necessary information for the construction of a binding to the interaction point selected when the interface was created, either by direct or indirect addressing information;
4. Relocation information identifies a relocator object that can be queried if a binding involving the interface reference fails, either during creation or subsequently;
5. Security information is not yet specified in detail;
6. Additional information is manipulated by functions not directly related to binding;
7. Flow information contains the name, the direction, the type, and the quality of service characteristics of a flow;
8. Group information has not yet been specified in detail;
9. Causality information denotes the role that the interface plays in the interaction, e.g., client or server;
10. Interface quality of service type is not yet specified in detail;
11. Non-interpreted reference is used for transporting the interface reference and it includes the following two fields;
    - Interpreter reference identifies an object able to replace the opaque-info part of a non-interpreted-reference with a new reference.
    - Opaque information does not have a standardised format, but it should contain such information that the above listed items can be extracted from it.

The description of interface references used for federated systems, Definition 2.18, is consistent with the ODP interface references, although different aspects of the system environment are stressed.

ODP binding model

The ODP binding model defines concepts that are required to support the communication model: interface binding, contracts, and liaisons. Additional terminology arises from the distribution transparency required from communication and the quality of service contracts involved. The distribution transparency aspects are discussed in the following subsection; the quality of service aspects were already discussed in Section 2.4.4.

For communication, the interfaces must be bound together and a channel has to be constructed to support the information exchange. The ODP reference model allows static binding but it especially introduces the notion of late binding. Late binding means that the communication partner (server) can be selected even after the service request initiation.

Binding is defined by RM-ODP as follows:

Definition 3.12 *"Binding: A contractual context, resulting from a given establishing behaviour.*

Establishing behaviour, contractual context and enabled behaviour may involve just two object interfaces or more than two.

An object which initiates an establishing behaviour may or may not take part in the subsequent enabled behaviour.

Enabled behaviour (and, by analogy, contractual context) may be uniform (i.e. each participating object can do the same as every other) or non-uniform (i.e. one participating object has a different role from another, as in client and server).

There is no necessary correspondence between an object which initiates establishing behaviour and a particular role in non-uniform enabled behaviours (e.g. in a client-server contractual context, either object could validly have initiated the establishing behaviour)" [92, clause 13.4.2].

The definition of binding actually requires only information about the shared contract to be available to all object interfaces involved in communication. The shared contract is further defined:

Definition 3.13 *"Contract: An agreement governing part of the collective behaviour of a set of objects. A contract specifies obligations, permissions and prohibitions for the objects involved.*

The specification of a contract may include

a) a specification of the different roles that objects involved in the contract may assume, and the interfaces associated with the roles;

b) quality of service attributes;

c) indications of duration or periods of validity;

d) indications of behaviour which invalidates the contract;

e) liveness and safety conditions" [92, clause 11.2.1].

The contractual context can be seen dually: the contract information represents the state, and the potential behaviour is represented by a liaison. Liaison is defined by RM-ODP:

Definition 3.14 *"Liaison: The relationship between a set of objects which results from the performance of some establishing behaviour; the state of having a contractual context in common.*

A liaison is characterised by the corresponding enabled behaviour" [92, clause 13.2.4].

Member objects of a liaison may have distinct roles in the joint community. Thus, the federated system concept for client and server role interfaces (recall Definitions 2.14 and 2.15) can be derived.

The ODP reference model differentiates computational and engineering interfaces, but it does not specify whether the binding establishment should take place at the computational or the engineering interfaces. The separation of these interface abstractions help in hiding interface heterogeneity: Two objects can communicate through an interface pair, if the interfaces are conformant. Conformance requires that the service descriptions at client and server interfaces are compatible and the information passed across the interfaces is sufficient for the receiving partner. If the information representation as data form or data grouping differs at client and server interfaces, then interceptors can transform one data representation to another.

The conceptual binding must eventually be realized by a concrete channel between the communicating interfaces. The channel structure is rather traditional:

Definition 3.15 *"Channel: A configuration of stubs, binders, protocol objects and interceptors providing a binding between a set of interfaces to basic engineering objects, through which interaction can occur" [93, clause 8.1.8].*

The responsibilities of the channel components are familiar: Stubs marshal and unmarshal the exchanged messages, binders monitor that the end-to-end connection is not lost, and the protocol objects transport the exchanged messages. However, the concept of interceptors is explicitly separated:

Definition 3.16 *"<X> interceptor: An engineering object in a channel, placed at a boundary between <X> domains. An <X> interceptor*

- *performs checks to enforce or monitor policies on permitted interactions between basic engineering objects in different domains;*
- *performs transformations to mask differences in interpretation of data by basic engineering objects in different domains" [93, clause 8.1.11].*

A special case of contracts is environment contract that defines how an object is supported by its environment, i.e., the supporting platform:

Definition 3.17 *"Environment contract: A contract between an object and its environment, including quality of service constraints, usage and management constraints.*

Quality of service constrains include

- *temporal constraints (e.g. deadlines),*
- *volume constraints (e.g. throughput),*
- *dependability constrains covering aspects of availability, reliability, maintainability, security and safety (e.g. mean time between failures).*

Usage and management constraints include:

- *location constraints (i.e. selected locations in space and time),*
- *distribution transparency constraints (i.e. selected distribution transparencies)" [92, clause 13.2.4].*

The relationship between binding contracts and environment contracts is interesting: once an object establishes a binding to another object, the binding contract becomes part of the environment contract of that object. This means, that all previously established liaisons restrict the liaisons into which an object can later join.

ODP transparency model

The communication model aspires more powerful communication primitives for programmers by including concepts for the management of heterogeneity and distribution. The concept of selective distribution transparency should be made available for programmers, together with automatic tools for managing the component concepts. Programmers should be able to select a set of distribution transparency services for each communication primitive; platforms should implement the corresponding support functions.

Distribution transparency is defined by RM-ODP as follows:

Definition 3.18 *"Distribution transparency: The property of hiding from a particular user the potential behaviour of some parts of a distributed system" [92, clause 11.1.1].*

Distribution transparency is selective in ODP systems [93, clause 16]. The RM-ODP describes the following distribution transparencies [93, clause 16]:

- access transparency – masks differences in data representation and invocation mechanisms to enable interworking between objects [93, clause 16.1];
- failure transparency – masks, from an object, the failure and possible recovery of other objects or itself, to enable fault tolerance [93, clause 16.2];
- location transparency – masks the use of information about location in space when identifying and binding to interfaces [93, clause 16.3];
- migration transparency – masks, from an object, the ability of a system to change the location of that object [93, clause 16.4];
- persistence transparency – masks, from an object, the deactivation and reactivation of other objects (or itself) [93, clause 16.5];
- relocation transparency – masks relocation of an interface from other interfaces bound to it [93, clause 16.6];
- replication transparency – masks the use of a group mutually behaviourally compatible objects to support an interface [93, clause 16.7];
- transaction transparency – masks coordination of activities among a configuration of objects, to achieve consistency [93, clause 16.8].

The ODP reference model does not yet define how these transparencies should be implemented (Chapter 9 discusses the positioning of transparency support objects in federated channels). However, the reference model describes a hierarchy of transparency concepts so that a group of transparencies can be implemented together. The users cannot select transparent services individually, but groups of transparencies. The hierarchy is illustrated in Figure 3.2. The implementation techniques of the transparency concepts include channel stubs, indirectness of references, and binding based of types instead of interface identifiers.

ODP structuring concepts

The ODP reference model introduces the structuring concepts of community, domain, and federation. These concepts can be used for organising objects for producing and exploiting services. The structuring concepts are characteristic to ODP reference model and are from there being adopted to some platform architectures, like CORBA. These are also the essential concepts on which the federated system model is founded.

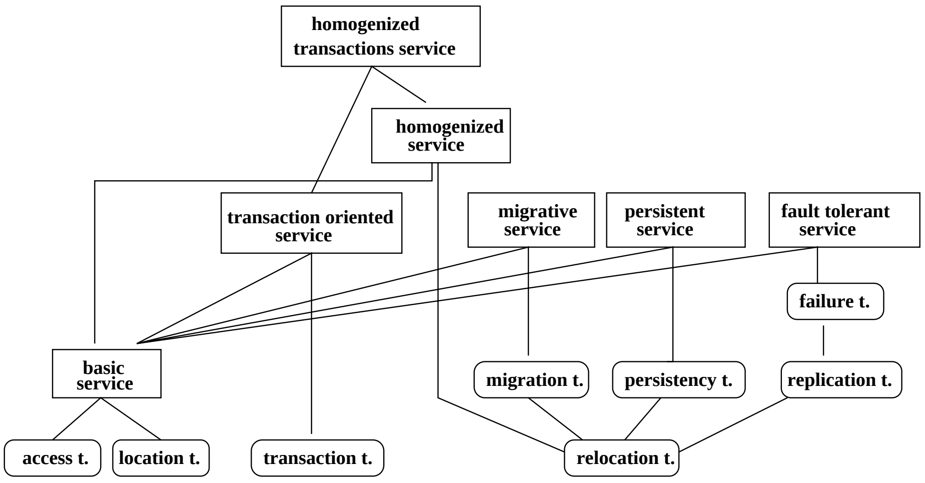


Figure 3.2: Hierarchy model of transparency concepts.

Definition 3.19 *"Community: A configuration of objects formed to meet an objective. The objective is expressed as a contract which specifies how the objective can be met"* [93, clause 5.1.1].

Definition 3.20 *"<X> federation: A community of <X> domains [93, clause 5.1.2]."*

Definition 3.21 *"<X> domain: A set of objects, each of which is related by a characterising relationship <X> to a controlling object"* [92, clause 10.3].

The structuring concepts can be considered to be either static, design time concepts, or dynamic, operation time concepts. For example, we can consider a community of exporters and importers around a trader to be a dynamic concept, because exporters and importers can join and leave the community at will. The trading community can also be considered as a trading domain, because the trader controls the other objects by manipulating meta-information about their interfaces. In some cases the federation between two trading domains can be static, because the traders are coupled at design time.

In case of a federated system architecture, the federation is preferably created at operation time, when the traders find out each other's trading interfaces. The structuring concepts are exploited in Part III.

ODP approach to conformance

The ODP reference model aims at specifications that allow software portability and interoperability. The specification must be very strict in order to guarantee these properties. The implemented systems must conform to the specification and be testable against the standard specification. On the other hand, the specifications should allow heterogeneity of the system even at very high abstraction level.

In order to accommodate both of these requirements, the ODP reference model defines four classes of reference points – points where the conformance to the standard specification is both required and allowed to be tested. Other behaviour than that visible at these reference points is not considered. The reference points of an object reflect such interactions that change the object's environment.

Definition 3.22 *"Programmatic reference point: A reference point at which a programmatic interface can be established to allow access to a function. A programmatic conformance requirement is stated in terms of behavioural compatibility with the intent that one object be replaced by another. A programmatic interface is an interface which is realized through a programming language binding" [92, clause 15.3.1].*

Definition 3.23 *"Perceptual reference point: A reference point at which there is some interaction between the system and the physical world" [92, clause 15.3.2].*

Definition 3.24 *"Interworking reference point: A reference point at which an interface can be established to allow communication between two or more systems. An interworking conformance requirement is stated in terms of the exchange of information between two or more systems. Interworking conformance involves interconnection of reference points" [92, clause 15.3.3].*

Definition 3.25 *"Interchange reference point: A reference point at which an external physical storage medium can be introduced into the system. An interchange conformance requirement is stated in terms of the behaviour (access methods and formats) of some physical medium so that information can be recorded on one system and then physically transferred, directly or indirectly, to be used on another system" [92, clause 15.3.4].*

These conformance concepts are applied for the specification of a functionality of a system, not a whole system. Specifications of several interrelated functionalities can be allocated to same implementation objects. The development of ODP component standards gives a good example of this convention: trading function and type repository function are specified in different standards, although the same implementation objects can well support interfaces for both functions. The ODP work plan also includes a set of standards called 'component composition standards'. They shall combine two or more function standards at engineering viewpoint level.

3.3 ODP viewpoints

An important component in the ODP reference model is the definition of viewpoint languages. In this dissertation, the viewpoints are not directly used, but instead, most discussions are mapped to a single viewpoint (computational viewpoint). However, it is impossible to avoid some references to other viewpoints as well. So it should be helpful for the reader to understand the reason for the existence of the five viewpoints. Therefore, and also because the viewpoints are often considered as the most essential part of the model, we give a short overview.

The ODP viewpoints allow object systems to be specified in an organised, guided manner. However, the viewpoint rules do not instruct on the level of detail or completeness of the specifications. Those aspects must arise from the software engineering process in which the viewpoint specifications are involved. Because of this multi-purpose nature, the specification rules are very general and should have their specific interpretations in each use-case. We can still claim that the goal of the ODP reference model is not to cover the full software engineering process, but only to support system specification and specification conformance analysis. For implementation specifications the ODP viewpoints would be too hefty.

The ODP reference model defines five viewpoints – enterprise, information, computational, engineering and technology viewpoints [94, 93]. A system must be specified from each of these viewpoints. Each viewpoint specification is a consistent and complete specification on its own, but it only considers those aspects of the system that are valid on its point of view. The viewpoint specifications do not overlap totally, but they may show different level of detail in the areas where they discuss same or related features. The engineering viewpoint specifications are tightly related with the ODP infrastructure model that is specified as part of the engineering viewpoint specification rules. The viewpoint specifications can be considered as projects of a system.

The enterprise viewpoint description of a system specifies the activities and the responsibilities of the system [93, clause 6]. Activity means any information exchange sequence and it is a high-level abstraction of the operations within the system. The system itself can have any granularity that is interesting. The system can be as wide as a global information network with all applications or as small as a memory cache in a processor. The enterprise specification identifies the system, its environment, and the required communication of the system and its environment. The specification answers to the questions "What is the purpose of the system?" and "What services the system is responsible to provide?" and "Who needs the services?".

The information viewpoint description of a system identifies logical information entities, their logical contents, their repositories and the objects that are responsible of the information flow in the systems [93, clause 7]. Questions for information viewpoint specification are "What information is needed to support the system's services?", "Where does the information come from and go to?", and "Is it necessary to store the information somewhere?". The information viewpoint specifications should not describe data structures, but only the semantics of the information. Also, the technique of storing information is irrelevant in this viewpoint (as the logical infrastructure supports storage services).

The computational viewpoint specification captures the behaviour of the system [93, clause 8]. Behaviour is an abstraction of how things are done, in contrast to the notion of what things are characteristic in the enterprise viewpoint activities. An activity identified in enterprise viewpoint may involve several objects to perform a sequence of operations in the computational viewpoint. The computational viewpoint shows the system as a composition of logical objects. For each object its interfaces are described. If the interface involves operations, each operation gets logical parameter descriptions (logical information components, not technical data structures) – if the interface involves streams, each data flow component of a stream gets logical protocol descriptions instead. This is the viewpoint that usually explicitly shows potential for distribution. Neither the enterprise viewpoint nor the information viewpoint specifications need to express any distribution concerns. The computational viewpoint answers to questions like "Which operations are available?", and "Who (which logical entity) performs the operation?".

The engineering viewpoint specification identifies the infrastructure services needed for the system to operate [93, clause 9]. The RM-ODP engineering viewpoint defines the set of available infrastructure services, and all other engineering viewpoint specifications should show how the specified system utilises these services. The engineering specification, therefore, answers the question "By which services are the computational objects supported?". The ODP infrastructure model identifies a set of global, distributed basic services that should be available at each node in the global system. These include invocation of operations, transfer of continuous data as streams, trading, type repository functions, etc. [91, 96, 97]. These services facilitate selective transparency of communication between objects.

The technology viewpoint specification shows in a concrete hardware and software configuration how the system services and other required components are realized [93, clause 10]. The specification answers the question "How are the infrastructure services realized?".

The system specification includes five complete specifications, that all can be analysed as separate. Each viewpoint reveals a different aspect of the system, and therefore, the full functionality can only be seen by looking at all specifications together. As the viewpoint specifications are all complete alone, the abstraction levels of objects in the specifications can differ. Still, the specifier must show how the specifications are mapped together. The usage of viewpoints helps the specification of large systems by separating concerns to separate specifications.

The ODP reference model introduces a vocabulary and a set of languages to discuss (distributed) systems, but it does not prescribe any special techniques to do this. Any specification language or technique that supports the same concepts can be utilised for ODP-style specifications. The problem with current formalisms that are close to the ODP concepts is that they do not support the level of dynamicity required in ODP specifications. However, development of tools and languages supporting the reference model would be very important and beneficial. When suitable tools are available, an ODP development technique could be formulated. (A study in that area is presented in [22].)

The relationship between the viewpoints and the software engineering process phases have been widely discussed and various interpretations have arisen.

It has been suggested [197] that the viewpoints follow the waterfall model [193] steps from requirement analysis with enterprise viewpoint to the implementation with technology viewpoint, as illustrated on the left side of Figure 3.3. This interpretation brings information viewpoint and computational viewpoint specifications unnecessarily close to each other to actually use the full power of the viewpoint concepts. The resulting specifications are very programming oriented.

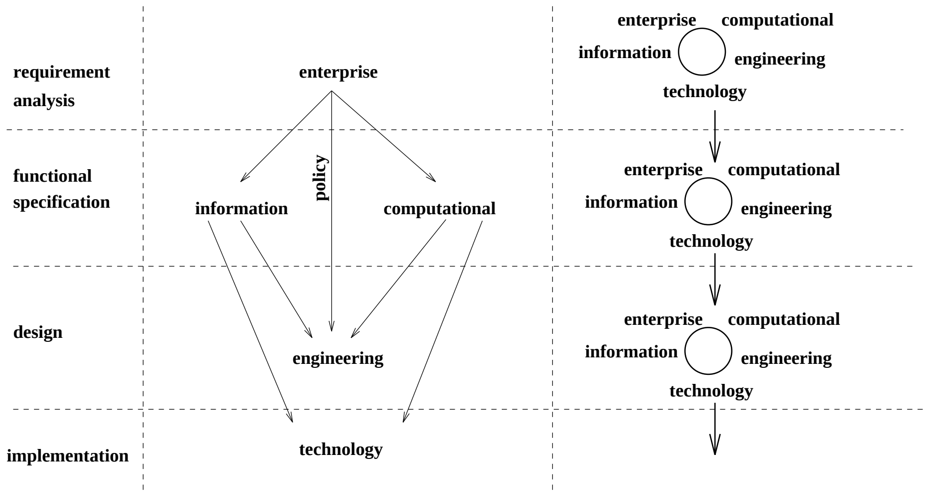


Figure 3.3: ODP viewpoints and software engineering.

It has also been suggested [228, Glossary] that the viewpoints match the levels in the abstract computing platform hierarchy. An often heard but not published interpretation suggests that each viewpoint specification should be aimed at different audience and that each audience should get sufficient information from a single specification. Both of these interpretations lead either to insufficient information at a given specification, or more commonly, the specifications are extended to include aspects of other viewpoints as well.

In spite of the above suggestions, we consider that in each development phase (requirement analysis, specification, design, etc.) all five viewpoints are needed to describe the system in all aspects from the necessary level of detail (right side of Figure 3.3). This interpretation seems to be widely accepted within the ODP standardisation group (see for example [140]). Naturally, the focus of each development phase is reflected on the extent of each viewpoint specification.

3.4 ODP infrastructure

The ODP reference model defines an abstract infrastructure to offer basic services like distribution transparent communication primitives. The infrastructure is described using some supporting concepts, that give an internal engineering view of the middleware. However, these concepts are only used for description purposes, not as technology requirements. In the following, we consider the abstract infrastructure from two points of view, functions supported and the organisation of the supporting, hypothetical objects.

The infrastructure model is an abstract computing engine and does not intend to prescribe the implementation technique of the platform involved. Therefore, the infrastructure model allows implementation independent discussions on various services, which property we exploit in Section 5.3.3 when discussing the distribution of a trader object.

Functions

The ODP reference model supports distribution transparent communication with the four fundamental function classes: management, coordination, repository, and security functions [93, clause 12].

The management functions include

- node management function that controls processing, storage and communication within a node, i.e., nodes support time services, creation of channels between objects, location of interfaces, and management of processing threads;
- object management function that checkpoints and deletes objects;
- capsule management function that instantiates, recovers, reactivates, and deactivates clusters and deletes capsules; and
- cluster management function that checkpoints, recovers, migrates, deactivates or deletes clusters.

The coordination functions include

- engineering interface reference tracking function that monitors the transfer of engineering interface references between engineering objects in different clusters
- event notification function that records and makes available event histories (logs),
- checkpointing and recovery function that coordinates creation of cluster checkpoints (time, storage), and coordinates the use of the stored checkpoints in recovery of failed clusters,
- deactivation and reactivation function that coordinates cluster deactivation and reactivation using checkpointing for other reasons than failures,

- migration function that coordinates the migration of a cluster from one capsule to another,
- transaction function that coordinates and controls a set of transactions to achieve a specified level of visibility, recoverability and permanence.

The repository functions include

- trading function that mediates advertisement and discovery of interfaces, and
- type repository function that manages a repository of type specifications and type relationships.

The security functions include conventional security related services, i.e., functions for access control, security audit, authentication, integrity, confidentiality, non-repudiation and key-management.

Objects

The management and coordination functions are supported by clusters, capsules, nodes, nuclei, channels, and basic engineering objects.

Definition 3.26 *"Cluster: A configuration of basic engineering objects forming a single unit for the purposes of deactivation, checkpointing, reactivation, recovery and migration" [93, clause 8.1.1].*

Definition 3.27 *"Capsule: A configuration of engineering objects forming a single unit for the purpose of encapsulation of processing and storage" [93, clause 8.1.4].*

Definition 3.28 *"Node: A configuration of engineering objects forming a single unit for the purpose of location is space, and which embodies a set of processing, storage and communication functions" [93, clause 8.1.7].*

Definition 3.29 *"Nucleus: An engineering object which coordinates processing, storage and communication functions for use by other engineering objects within the node to which it belongs" [93, clause 8.1.6].*

Definition 3.30 *"Basic engineering object: An engineering object that requires the support of a distributed infrastructure" [93, clause 8.1.1].*

We have used the term application object for basic engineering objects in the context of open systems, in Section 2.4.

The concept of channel bridges between the ODP binding model and the actual infrastructure objects. Therefore, we have already introduced channels in Definition 3.15.

Configuration

The configuration of these objects is illustrated in Figure 3.4. A node contains a set of capsules. One of the capsules has a special role of being a nucleus of that node. All the other capsules are bound to the nucleus capsule to get the basic services. The nucleus is responsible of the node management function. Each capsule contains a set of clusters. One of the clusters has a special role of being a capsule manager. It is responsible of the capsule management function. For these purposes, it may request the nucleus for some services. The capsule manager, or another cluster, may perform the tasks of one or more cluster managers, with the corresponding functions. All objects are instantiated in such a way that they have prebound connections to the objects that offer the fundamental functions for them. This means that all objects are encapsulated into the clusters. Each cluster is bound to a cluster manager, which in turn is able to request the services of the capsule manager. The situation can be made more concrete by using an example: a segment of virtual memory containing data items can be considered to form a cluster, and a process represents a capsule; a computer with operating system and applications can be considered to form a node, and an operating system kernel represents a nucleus.

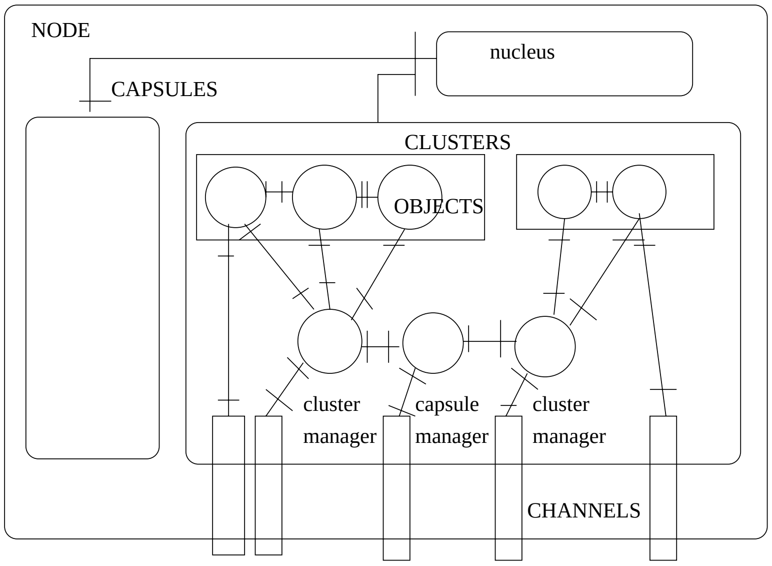


Figure 3.4: RM-ODP infrastructure objects [93].

As each cluster is indivisible, inter-object communication between objects in the same cluster may take place in any suitable method. The cluster is the smallest possible unit of migration and activation, so those actions cannot cause problems. Between objects in different capsules, either in same or different nodes, a channel is required for communication.

The granularity of these structures is arbitrary. When a system is described using these terms, the components that use non-ODP communication methods between each other are encapsulated within a cluster. A node can represent one computer or a set of them.

For the computational behaviour of objects, the channel creation is the most fundamental infrastructure service. Channels are created by nucleus objects in cooperation with each other. The channel structure has been illustrated in Figure 2.5. The channel supports distribution transparent interaction between engineering objects. This includes, for example,

- operation execution between a client object and a server object,
- a group of objects multi-casting to another group of objects, and
- stream interaction involving multiple producer objects and multiple consumer objects.

Some of the functions listed above are also considered to be computational functions, i.e. visible and exploitable by programmers. These include trading and type repositories. The detailed specification of these functions is not within the scope of the basic reference model of ODP.

Instead, a series of component functions has been started. First of the component functions is the trading function that has been recently completed [96]. We will discuss the trading function standard in Chapter 8. The second component function will be the type repository function. It currently has a committee draft status and it is scheduled to be finished within two years. Also component framework standards are under development, for example, the ODP binding framework [102]. The interface reference structure and representation as well as the general model for federated binding processes will be included. Also this work has a committee draft status and will be finished in a couple of years. Other work started in 1997 includes quality of service negotiation processes within ODP and concrete binding protocols.

Conclusion

The ODP model defines an abstract computing platform that comprises a set of coordination, management and repository functions. Each open system should independently support these functions. These fundamental functions include

- binding of an local interface to an remote interface,
- creation of a communication channel for bound objects in cooperation with an other ODP system,
- exchange of necessary information for interface binding process, such as interface properties, types, available protocols, quality of service, etc.,
- exchange of necessary information for channel creation, such as interface identity, location, and
- exchange of channel reconfiguration information.

The implementors of these services should respect the following assumptions:

- Computing entities are modelled as objects. An object has an implementation part and an interface, that represents the requests it is able to response to. This information must be present for all object realizations, although the information would not be an automatic product of the programming process.
- A globally accepted interface description language cannot be expected to exist, because a variety of languages is already used and new languages are still wanted to cover additional interface aspects.
- The abstract service type of an object can be realized as various object and interface implementations.
- The templates from which objects and object interface instances are created from can be defined in various languages.
- The templates and the abstract types can be mapped together at run-time. This is called late binding.
- The object instances or service types cannot in general inherit knowledge of each other's properties. The inheritance is restricted by the independent administrative domains. Other mechanisms must be used, such as trading.

These requirements and implementation assumptions differentiate the ODP reference model from other architectures that include open system features. The ODP reference model offers most general services, and makes least assumptions of the implementations. (The open features are collected in a comparison table in Chapter 4.)

Chapter 4

Related work: Some distributed platform architectures

This chapter compares three well-known open platform architectures against the ODP reference model. The purpose of this comparison is to investigate how open architecture models are supported by the current platforms. We first discuss each architecture separately, also noting the role of trading function in each case. We summarise with a comparison that also relates the platforms with the distributed and the federated system architectures.

4.1 Relationship of open architectures

The discussed architectures are not independent from each other. The RM-ODP work is done in interaction with the consortia of vendors and service providers, like APM (Architecture Projects Management Ltd) [4], TINA-C (Telecommunications Information Networking Architecture Consortium) [6], and OMG (Object Management Group) [210]. These consortia have each produced their own concrete architectures and tool-sets. However, each architecture is in some aspect more restricted than the ODP reference model. The RM-ODP offers them a common framework that allows evolution, and even competition, and shared guidelines for long term development.

APM manages the development of ANSA architecture (Advanced Networked Systems Architecture) [242] that has a very strong influence in this research area. ANSA was the first and very successful project with a set of software development tools for open systems (ANSAware [3]). The current focus of the ANSA research is in interoperability between open platforms [199], real-time multi-media distributed systems [137], and security in distributed systems [25].

OMG is working on the definition and selection of services for a distributed platform, OMG/CORBA [176, 177]. System vendors work on their corresponding proprietary services. These supporting services include the 'traditional operating system services', but not the application services trusting on them. The consortia identify the required services and their interfaces, and recommend selections of current commercial products from which a distributed, homogeneous platform can be configured. However, not all required basic services are yet available as commercial products. Research for identifying these requirements is in progress.

The TINA Consortium works on an architecture, TINA (Telecommunications Information Networking Architecture) [6, 232, 231], for telecommunication services. The consortium ties together standard frameworks such as the ODP model, the OSI management model, the IN (Intelligent Network) framework [59], the TMN (Telecommunication Management Network) model [106], and network technologies, such as ATM (Asynchronous Transfer Mode) [13]. The importance of the TINA architecture lies in the way it ties together the results of open distributed computing research and traditions of telecommunication.

The approaches taken in each of these architectures differ is some major aspects – the contract related concepts, distribution of interface concepts, and exploitation of trading service. Several steps are visible that can be made to join the ANSA, TINA, and CORBA approaches with the ODP model. On the conceptual side, the necessary set of concepts, and the fundamental infrastructure functions can be adopted to all platform architectures. The consortia can distinguish a few central business areas and develop an interoperability contract framework for each of these areas. For instance, OMG is developing a business object framework that includes interface type definitions for accounting services. Such frameworks have two audiences. First, they help programmers by giving them predefined services. Second, they contribute to system interoperability by specifying common interface types, on which federation can be based.

The architectures discussed in this chapter are by no means the only ones that include distributed or federated system features. We could have included systems like DCOM [154], DCE [180], IBM Blueprint [81], ICL OPENframework [24]. However, we restrict the comparison to a set of system architectures that suitably present the essence of current industrial interests. (Other comparisons are available, e.g. [12].)

The goals of the architectures presented are shared, but the designs differ in some aspects. The differences can be derived from the basic assumptions on which the architectures are based.

4.2 TINA

The Telecommunications Information Networking Architecture Consortium (TINA-C) is formed by network operators, and telecommunication and computer equipment suppliers. The consortium aims at defining and validating an architecture for telecommunication services. The results of the consortium work will be used as input in standardisation work in the area.

Architecture

The TINA architecture [231, 224, 228] has adopted many concepts from the ODP work. However, the TINA work area is broader than the area of ODP reference model: TINA work captures also system management and software engineering aspects. On the other hand, the TINA work area is more specific: it captures a specific application area – telecommunication services – which gives a more concrete semantics for many of the concepts. The peculiarity of the distributed telecommunication applications is that they directly access and manipulate components of the underlying transport network. In contrast to the ODP goals, the TINA architecture aims at a concrete methodology in the area of telecommunications. The technical goal of the TINA architecture is to consider telephone services, future interactive multi-media services, information services, and (telephone) network management as software-based applications that operate on a distributed computing platform.

The TINA architecture is partitioned to four sub-architectures:

- Computing architecture defines methods for creating distributed software. An important part of the computing architecture is the TINA-DPE (TINA Distributed Processing Envir-

onment) [225]. The scope of this sub-architecture is rather similar to the ODP reference model. Special research activity has been focused on the stream interface concepts and bindings that involve quality of service negotiation [58].
- Service architecture defines principles for the telecommunication specific business services.
- Network architecture defines generic components for networking. In the telecommunication area, information transport and voice transport will be combined via new, open protocols [227]. In the ODP model this level of communication is thought to take place with already existing protocols.
- Management architecture defines structures and principles for the management of the objects defined by the sub-architectures [226].

For the purposes of this dissertation, the focus of discussion is in the TINA Distributed Processing Environment, DPE. The TINA DPE infrastructure provides an abstract execution environment (a homogeneous middleware) for computational objects with services like

- trading service, for supporting late binding between objects (late binding: a client can be configured or installed earlier than a server),
- transaction service;
- repository service, for persistent storage of objects;
- notification service, for emit and receive notifications without being aware of their sources or sinks;
- security service;
- performance monitoring service;
- object life-cycle service (creation, deletion, activation, deactivation and moving of objects);
- installation service (object installation and removal at nodes); and
- configuration service (location, activity status and binding of objects).

The TINA DPE architecture also relates the DPE to the application objects, and to the kernel transport network (kTN) that provides end-to-end information transfer between them (Figure 4.1).

The TINA architecture offers a homogeneous middleware service layer but it trusts to a collection of heterogeneous telecommunication systems. The TINA interworking model describes how the DPE services are supported by autonomous computing nodes and autonomous transport networks. Autonomy means that a node is capable of independent operation. Examples of DPE nodes are a single processor system capable of independent operation, a multi-processor system, and a system distributed over a LAN with a distributed operating system. Each autonomous node supports a local part of the DPE infrastructure (called Native Computing and Communication Environment (NCCE)). Transport networks may include IN systems or broadband systems like ATM and broadband ISDN [229].

A subsystem can be joined to a TINA-system only if it is conformant in all its conformance points, i.e. interfaces through which interaction with other domains or other layers is allowed [230]. Conformance points are covered by testable reference points. Each reference point is defined by specifying all potential interactions. In order to consider potential conformance points, we study the three TINA layers: DPE layer, network resource layer and service layer (Figure 4.2). The DPE layer implements the distribution transparencies and provides various DPE services to the applications. The network resource layer controls and manages the transport network. The service layer includes the applications built on the TINA system, like video conferencing and tele-shopping, electronic mail, and basic calls.

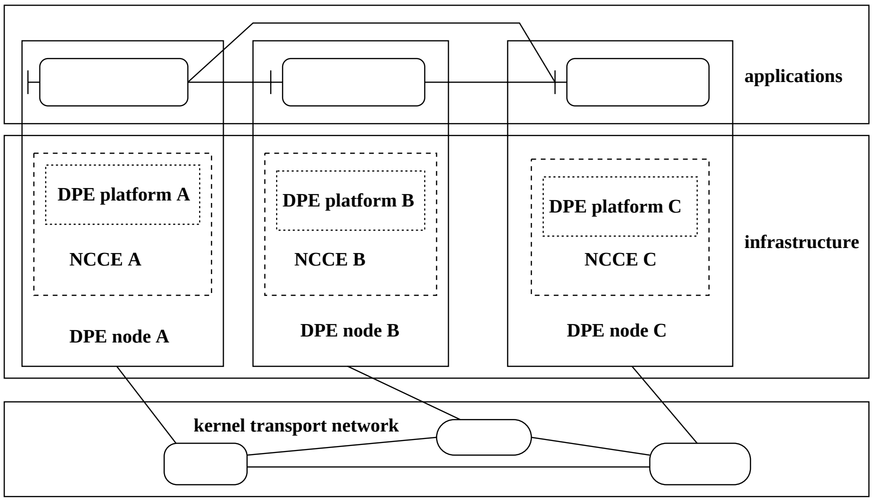


Figure 4.1: Model of the infrastructure and its environment in TINA [224].

Objects and interfaces

Objects in the TINA computing architecture have similar features as the ODP objects. However, TINA objects are more programming language oriented. The TINA application behaviour is specified by using TINA Object Definition Language, ODL [233]. ODL includes the OMG IDL language and extends it with object definition and some additions in interface specifications. The object communication model defines interfaces as the only means that provide access to services. Interaction occurs on operational interfaces and stream interfaces. Operations include interrogations with two variations - blocking and non-blocking interrogations, and announcements. Stream interfaces include multiple, simultaneous, uni-directional bit flows [60].

The specification of an operational interface consists of the specification of

- the syntactic structure (signature) of the operations (the name of the operation, the names and types of the arguments, and the names and types of the result parameters);
- an indication for each operation specifying whether it is an interrogation or announcement;
- service attributes, like availability or dependability; and
- the behaviour of the interface specifying the semantics of the operations. The behaviour description shows timing constraints, operation sequencing constraints, and concurrency constraints.

A stream interface is an abstraction that represents a communication end-point, either the source or the sink for some information flows. The flows between stream interfaces are unidirectional bit sequences with a certain frame structure (data format and coding) and quality of service (QoS) parameters. The QoS parameters include timing requirements for different frames, and synchronisation requirements between flows [228].

In addition to the object concept, the TINA model also uses the concept of package. A package is a group of objects communicating with each other and potentially also with objects within other packages. A package may be, for instance, a module of an application. For openness, and

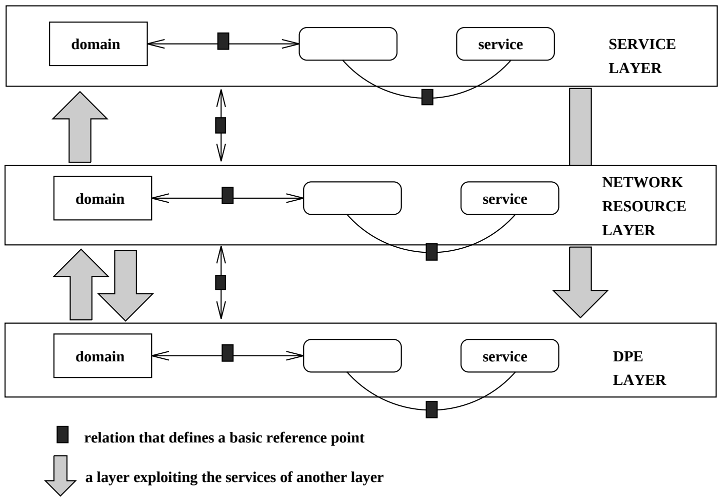

domain
service
SERVICE LAYER
domain
service
NETWORK RESOURCE LAYER
domain
service
DPE LAYER
relation that defines a basic reference point
a layer exploiting the services of another layer

Figure 4.2: The TINA layers and reference points between them.

portability, only those interfaces are critical that are accessed from other packages. Therefore, the TINA object model considers only those external interfaces to form a contract [228].

Interactions within a package within a node can be done using memory sharing, local procedure calls, etc. When the interactions cross node boundary or package boundary, a standard communication channel must be used [228]. The channel structure resembles the structure defined in RM-ODP. In communication, the notion of distribution transparency is used. Transparency includes access, location, migration, concurrency, failure, and replication transparencies which can be used selectively - only access transparency is compulsory.

Binding model

In TINA, objects can be bound together only when they are type conformant. Type conformance requirement is based on template hierarchy. Only interfaces that are defined by the same template are of the same type [228]. Templates can however have an inheritance hierarchy, and through that hierarchy, subtyping can be achieved.

Two object interfaces can be bound together when their interface references are known. Interface references contain information about

- the remote address of the interface instance, including communication ports and server host machine IDs for the selected protocols;
- the object configurations on which a channel can be based, together with suitable instantiation information, e.g.,
  - the addresses of objects which realize accepted protocol adapter functions,
  - the selected stubs and binder objects,
  - the objects necessary to provide additional transparency support, and
  - the QoS parameters for the objects involved in the binding;

- the interface references of objects (such as relocation service) which can be contacted in the case of a communication failure) [224].

Communication over multi-point bindings is possible, but the binding must always be constructed through a binding management interface. The basic binding action only considers bindings between two interfaces.

The TINA model allows application objects to make QoS contracts with the infrastructure [225]. Each offered interface declares its potential by describing the range of QoS values the service (transport or operating system service) is expected to be able to support. Likewise, the service requester (application) declares requirements for the quality of service, called a requirement contract. The DPE supports a negotiation service at binding time to match these contracts. For example, trading can be used to match QoS contracts. Furthermore, the QoS contract is not necessarily static. The actual QoS values can be monitored during the service usage and if the QoS contract is not met, the service user may request run-time re-negotiation. Using QoS monitoring, for instance, multi-media applications can do reasonable resource re-allocations when necessary.

Conclusion

As a summary of TINA architecture we can make the following classification:

- The TINA architecture supports a homogeneous middleware layer on top of heterogeneous operating systems and transport networks. The identification of transport networks in the model is fundamental, because of the telecommunication application area.
- The TINA ODL is used for describing system components.
- Interface structures are either inherited or traded. Interfaces carry place-holders for their current QoS attribute values.
- Binding requires an interface reference to be know. As a result a channel is created with transparency support when necessary. Explicit bindings can be managed. The QoS attributes have an essential role in selecting a server.
- Liaisons are recognised between application objects, platform objects, and network objects as conformance points. An object or a system must conform to these in order to be able to join the TINA system.
- Applications have the possibility to be aware of distribution.

For trading this model gives a minor responsibility. Only the responsibility of passing on QoS information is given to the traders. Type conformance and interface reference communication can be easily arranged by the other DPE services. Most current TINA DPE implementations are based on CORBA platforms, with interface and implementation repositories. In such an environment, trading supports easier management by saving administrators from extra software installation and configuration work. The TINA DPE services also includes QoS monitoring tools, that can be used to feed the traders with up-to-date QoS information. For the stream interface binding in TINA systems this is essential.

When the TINA middleware services are built the following assumptions are made in contrast to the ODP model:

- The TINA ODL is assumed to be a globally accepted interface description language.
- The object instances or service types can inherit knowledge of each other's properties in the whole system.

Most current TINA DPE implementations are based on CORBA, and therefore at the programming level the differences are minor. However, TINA has adopted streams and QoS negotiation, thus extending the model.

4.3 ANSA

ANSA is a program for computer companies, telecommunication companies, system integrators and end-users. The program is managed by APM and participated by companies like Bellcore, BT, HP, and ICL.

The ANSA program has produced an architecture for distributed systems and a package of software and software tools (ANSAware) to support the architecture and software development in that environment. In the following, we will consider the main features of the ANSA architecture model (read [242, 74] as structuring guides or [183, 78, 75, 50, 44, 243, 198] to give a more detailed view) and study the properties of ANSAware trading service [44, 76, 3]. The current focus of the work in ANSA is in

- interoperability between some current open platforms [199],
- real time multi-media distributed systems (DIMMA) [137], and
- security in distributed systems.

Architecture

The ANSA program has concentrated on defining an architecture, and because of this goal, the architecture specification mostly deal with conformance requirements, instead of giving design specifications for the system components. The validity of ANSA architecture has been demonstrated by successful system development projects. Examples of systems are the Astrophysics Data Systems (ADS) of the US National Aeronautics and Space Administration (NASA), and a hospital information system in Rome [242].

The ANSA architecture emphasises that systems are integrated of autonomous, heterogeneous domains [10]. Within a domain, technology and administrative policies reflect the needs of the users served by that domain. The architecture specification defines in which way each autonomous domain should support portability and interoperability. Interoperability is considered separately at application and network interfaces. The requirements for this kind of support can be expressed as conformance points, as illustrated in Figure 4.3. The conformance points for portability assert that an object is able to work properly on the services of the underlying abstract machine, independent on the abstract machine implementation. The services of the abstract machine include object creation, operation invocation and execution, operation termination delivery, and type safe binding of identified objects. The conformance points for interoperability at application level assert that the objects are able to communicate with each other, because they share a common view of potential interactions (service types) and general interaction semantics (such as synchrony and need to reply). The conformance points for interoperability at network level assert that transportation of data from one object to another can be successfully done as the objects are able to agree on a communication protocol, data representation, and transportation quality.

The interoperability of application objects requires that the objects have a shared view on several aspects, including

- the authority of controlling and monitoring resources,
- the method and policy of accounting and billing for resource usage,
- the use of shared data transport mechanisms on the infrastructure level,
- the used features of application interfaces, such as interface signatures and quality of service, and
- the shared semantics and naming of objects and services.

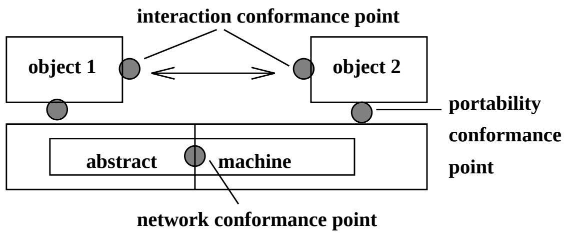

interaction conformance point
object 1
object 2
portability conformance point
abstract
machine
network conformance point

Figure 4.3: Conformance points in the ANSA architecture.

Objects and interfaces

The ANSA architecture denotes a system as a collection of objects related by their interactions. An object is a distinct element that manipulates information. An interaction is an exchange of symbols and requires that the interacting parties have a compatible interpretation for the symbols. If there are any differences in the symbol interpretation, they should be transformed. The transformed (intercepted) symbols can then be inherited by further communications between the same objects [198, 78].

The (original) ANSA architecture introduces only operational interaction models, interrogations and announcements. Operations are atomic and the operation types are classified only by their signatures, not by their behaviour. Interrogations are considered to be synchronous and announcements asynchronous communications between peer objects.

The interaction model is supported by the abstract machine that is able to set up a binding between objects. The binding process includes the following steps:

- reach agreement between the authorities of the service provider and the service consumer (willingness of cooperation between human organisations),
- advertise and search in appropriate repositories to store and locate information about existing and creatable objects (trading),
- compare specifications of offers and requirements (trading),
- negotiate between the involved parties where options are available or modifications are possible to the offered or required services (trading, peer object negotiations),
- select objects for co-operation (trading), and
- resolve any resolvable barriers to cooperation (create interceptors and servers) [78].

The result of the binding process is an established channel between objects. As we can deduce from the above list, type checking is not part of the binding process. The binding process is a run-time action of an object, whereas the ANSA model considers type checking to be most profitable in the compilation phase. The static type checking only covers operation signatures, not the operation behaviour.

The end-points of channels are associated with interface references. The interface references include

- the name of the end-point to which any messages should be sent in those terms managed by the local naming authority (for example, the name can consist of a network port and a protocol name);
- alternative route for reaching the same end-point, i.e., another end-point name;
- method for end-to-end integrity checking against errors in the underlying transport mechanism;
- relocation information, for example, an interface reference to a relocator that is able to give a new route;
- replication information, for example, a list of alternative interfaces for the same kind of service;
- information about interception strategies, for example, responsibility of data transformation is assigned to a certain component in a channel located either at client or server system; and
- interception information for interface references, to be used in cases when a remote interface reference is expressed in a language not native to the system where the reference is stored or used (it may be necessary to retrieve the relocator interface reference from the interface reference; this requires understanding of the foreign reference structure).

The major difference to the interface reference described in Section 2.4 are the issues of interception. The ANSA interface reference identifies a couple of variations in the information transfer process and therefore must flag which strategy is used.

Binding model

In the ANSA model the binding process can be either implicit or explicit. In implicit binding there are no methods for controlling the channel properties, such as transport quality, timing of binding action, or number of peer objects bound. Therefore, implicit binding is not suitable for applications where predictable performance, multi-object communication, or changes in the channel properties are required. Such requirements are common for all multi-media applications. When an explicit binding is created, a management interface to the channel object is available for the peer objects. Through this interface the channel properties can be manipulated.

When we study the explicit binding process in more detail, we see that the ANSA specifications give several options for establishment and management of a channel. The ANSA binding model emphasises that the application programs, especially client objects, have to control timing, quality of service, and other contractual information themselves. Special scenarios can be implemented when reasonable, but the model does not force them. However, the ANSA group believes that some scenarios become popular and therefore turn into de-facto-standards with readily available support tools.

In the simplest scenario suggested in the ANSA binding framework [183], a server has a private binding manager and all potential clients receive (for example through trading) a reference to the binding manager's interface. The binding manager instantiates a new service endpoint at the server for each client request. Another scenario is that the clients use a library that offers them a private binding manager. In such case both the client and the server have binding managers, as illustrated in Figure 4.4. The client's binding manager can be extended by abilities to interact with traders or by abilities to configure interceptors into the channel. A third scenario uses an external binding manager for multi-party bindings.

The ANSA model suggests that the binding managers are private to each server, and moreover, that they are specific to the interface type. The model used in this dissertation has placed the role of binding managers on the infrastructure. Instead of being specific to a given interface type, the binding managers can be constructed to be more general and to exploit type repository. As the ANSA model utilises only static type checking, there would be no other uses for the type repositories. Therefore, type specific binding managers are a very sensible solution.

The ANSA model differentiates between a logical binding and a physical channel. The channel is created and discarded based on a resource allocation policy associated to the binding. The policy can vary: A channel can be created and terminated separately for each invocation request and termination transmission. Alternatively, the channel is created when a first operation is invoked and left open, trusting that a garbage collection mechanism shall remove it after the last operation has terminated.

The ANSA applications must be aware of the existence of potential distribution. The ANSA system supports selective transparency with programmer tools. Research on distribution-aware applications can be found for example in [56].

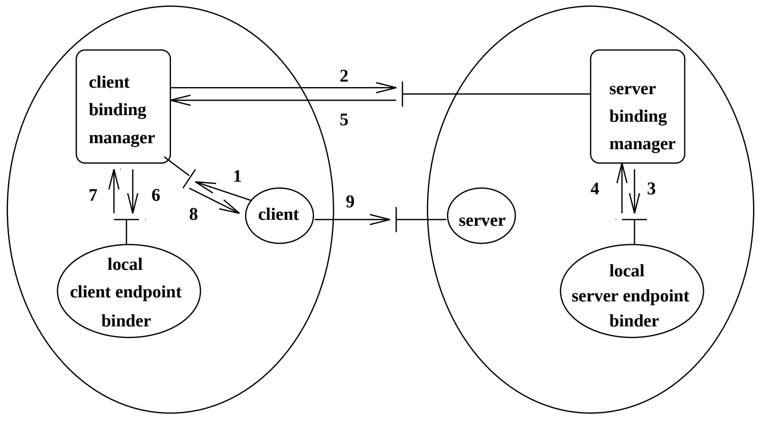
Conclusion

As a summary of this short introduction of ANSA architecture we can do the following characterisation:

- The ANSA architecture recognises autonomous domains within the global system.
- The ANSA architecture assumes a homogeneous platform interface, but still allows a heterogeneous engineering of the platform itself. In addition, the ANSA architecture offers tools for building homogeneous middleware services on several operating and communication systems.
- The ANSA binding model is designed for already identified objects, and it does not consider mechanisms for finding the suitable objects.
- The ANSA middleware includes a trading service for dynamic recognition of objects.
- In the ANSA architecture, liaisons are formed directly between the application layer objects.
- Originally the ANSA model only discussed operational interfaces, currently also stream interfaces.
- ANSA applications use selective transparency and are aware of distribution.

Additional work on the ANSA architecture adds QoS management into the binding process with the help of a trader [4].

The ANSA architecture and tools are an example of the potential open computing environments described by the ODP reference model. The ANSA architecture emphasises that applications and computing environments are integrated from autonomous, heterogeneous domains.



1. The client generates a client endpoint binder and invokes its own binding manager. The binding manager receives references to the client's binder and to the server's binding manager.
2. The client's binding manager invokes the server's binding manager and passes a reference to the client binder as an argument.
3. The server's binding manager creates a new server endpoint binder. It also requests the new binder to bind itself to the client's endpoint.
4. Reply for 3.
5. Reply for 2.
6. The client's binding manager invokes the client endpoint binder to bind it to the server endpoint.
7. Reply for 6.
8. Reply for 1.
9. Binding is ready for use and the client transmits to the service. The transmission usually means invocation.

Figure 4.4: Using client and server binding managers for explicit binding [183].

Therefore, the concepts of communication between objects and federation between organisational computing systems are central. When the ANSA middleware services are implemented, the same assumptions as with ODP model are used. However, the ANSAware implementation only gives one technical domain.

4.4 CORBA

The Object Management Group, OMG, is a consortium of software vendors, computer manufacturers, and computer software users. The goals of OMG include the reduction of computing system complexity and the lowering the cost of producing new software applications. OMG has introduced an architectural framework, Object Management Architecture (OMA) [211], and a set of related platform service recommendations [176, 177]. The platform recommendations are known with the name CORBA (Common Object Request Broker Architecture). For commercial reasons, the name CORBA is almost solely used to mean both the architecture framework and the recommendations proposing suitable realisations of it. In this section, we study OMG recommendations mainly on the architectural level.

Architecture

The architecture problem area covered by OMG [176, 177] can be partitioned into three components:

1. Application-oriented aspects that include solution-specific interfaces and common facilities. Application objects are not subject to OMG standardisation, but they are critical for a global business architecture. Common facilities provide a set of generic application functions, such as printing, document management, database services and electronic mail.
2. System-oriented aspects that include infrastructure specific issues of distributed object computing and management, such as object request brokers (ORB) and object services. ORB is the middleware that establishes the client-server relationship between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be either in the same computer or elsewhere in the network. Object services standardise, for example, the life-cycle management of objects. These services build a middleware layer for object manipulation. A small group of predefined object implementation styles are supported. These object models include individual programs and libraries.
3. Vertical market oriented aspects that include domain interfaces between applications and operating systems. Domain interfaces are specific to a certain application domain, instead of being generic like common facilities. Domain interfaces may combine common facilities and object services.

In this context, only the CORBA middleware and the domain interfaces are of interest.

The OMA framework and the CORBA specifications are focused on intra-organisational interoperability. This means that the problems to be solved focus on masking the heterogeneity of computing hardware and networking software. The model originally trusted on centralised control [176] but a second release [177] has defined concepts for autonomous domains and their co-operation. Heterogeneity between domains can occur as differences in object references, types, security, transactions, data representation, transport network protocols and addressing.

Processes, objects and interfaces

In the CORBA model, client processes are isolated from servers (called objects) by a well-defined encapsulating interface. The interface is implemented by ORB. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object's interface. The model is illustrated by Figure 4.5.

An interface is a description of a set of possible operations that a client may request of an object. An operation has a signature that describes the legitimate values of request parameters and returned results. In particular, an operation signature consists of

1. a specification of the parameters required in the requests,
2. a specification of the result of the operation,
3. a specification of the exceptions that may be raised by a request for the operation and the types of the parameters accompanying them,
4. a specification of additional contextual information that may affect the request (no other contextual information is transported to the object implementation), and

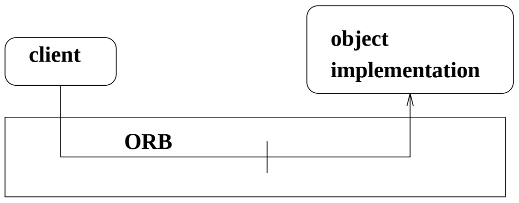


Figure 4.5: Object model in CORBA [176].

5. an indication of the execution semantics the client should expect from a request for the operation (at-most-once, best-effort). Operations can be synchronous operations (interrogations), deferred synchronous operations, one-way operations (announcements), or sequences of operations. In related areas, operation sequences are often referred as streams, but they do not support continuous data flows as we defined earlier.

Objects are created as instances of types and there is no difference between the concepts of template and type. The subtyping facility allows an object of one type to replace another object without being noticed.

OMG only uses one language for expressing operational interface templates. The standard language for this is OMG IDL [176]. The operational interfaces are matched on equality basis, or the interface template implementations must be in direct inheritance hierarchy which each other. This means that the knowledge about the peer object's properties can be achieved by inheriting it from the environment. This also means that dynamically applied interceptors are not exploited in the architecture. The types of interceptors (object adaptors and bridges between different ORB realisations) are statically positioned to the system constellations. Object adaptors are normally used to support CORBA conformant objects in the system. In addition, vendors can provide adaptors for integration to systems supported by other technologies, such as object databases or ODP systems.

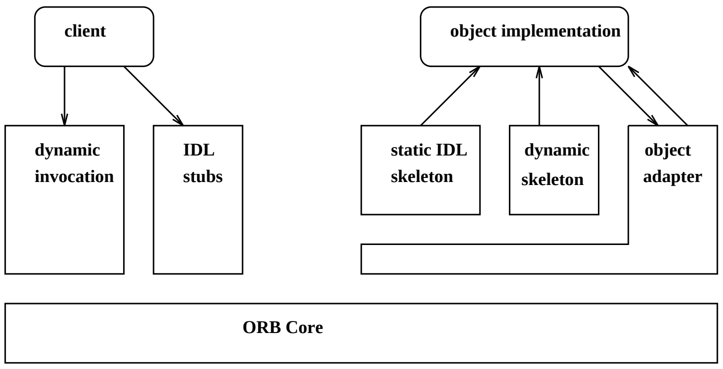


Figure 4.6: Structure of the ORB [176].

ORB services for binding

In CORBA systems, the binding actions are implicit and are performed by ORB services [176]. The essential components of the ORB can be seen in Figure 4.6. The ORB Core includes interface repository, implementation repository, and inter-ORB communication facilities. To make a request, the client can use the dynamic invocation interface (DII) or an OMG IDL stub. The dynamic invocation interface is an interface that is independent of the target object's interface. The stub interface includes specific stubs that are dependent on the interface of the target object. The object implementation receives a request as an up-call either through the OMG IDL generated skeleton or through a dynamic skeleton.

An object adapter is the primary way that an object implementation accesses services provided by the ORB. Services provided by the ORB through an object adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations. There are a variety of possible object adapters: for example basic object adapters for separate programs, and library object adapters. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects.

The interface repository is a service that provides persistent objects that represent the IDL information in a form available at runtime. The interface repository information may be used by the ORB to response operation invocations.

The implementation repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the implementation repository is specific to an ORB or operating environment, the implementation repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the implementation repository.

ORB interoperability services for binding

The ORB interoperability architecture introduces bridging of ORB domains [177]. When information in an invocation must leave its domain, the invocation must traverse a bridge. The role of a bridge is to ensure that the contents and the semantics of a message are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

The general inter-ORB protocol GIOP element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs (e.g., 'request', 'reply', 'cancelrequest', 'locaterequest', 'locatereply', 'closeconnection', 'messageerror'). The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection oriented transport protocol.

In CORBA systems, liaison related information is represented as interface references. Therefore, we will next look at the CORBA interoperable object references [177]. Each ORB has autonomy on deciding the structure of local object references. The ORB interoperability scheme uses domain relative addressing. (In this context, domain is specified by technical and administrative aspects: the organisation administering an ORB has authority over names in the domain, the ORB vendor selects the mechanisms available.)

Interoperable object references are only used when two ORBs communicate across a domain boundary. There is a need to allow any transport protocol to be used for object reference passing. The interoperable object references should also leave it to the ORBs to decide what kind of translation or proxy mechanism they use for adding foreign addresses to their own address spaces. Potential methods are creation of a proxy process, encapsulating the address to a wrapper that fulfils the local address structuring rules, etc.

Interoperable object reference includes a type identifier for the interface, type of the ORB, and notion of the security mechanism used at that interface, followed by a sequence of object-specific protocol profiles. Each profile supports one or more protocols and encapsulates all the basic information that the protocol needs for identifying an object. Any single profile holds enough information to drive a complete invocation. This means that the GIOP profiles shall include

- version number of the transport-specific protocol that the server supports,
- the address of an endpoint for the transport protocol being used
- an opaque datum (object key) used exclusively by the agent at the specified endpoint address to identify the object

Within OMG, security services [167, 168, 171] have also been developed. These services are even considered for inclusion to ODP reference model. However, there are concerns about the basic assumptions under the CORBA security services. For example, it is not clear that organisations can trust each other in the manner suggested in the CORBA specifications.

Conclusion

As a summary of the OMA architecture and the CORBA design, we can do the following characterisation:

- The OMA model identifies autonomous domains within the global system, but allows only a variety of ORB implementations to be used. Some other platforms can be integrated through encapsulation with standardised, static bridges. The middleware and applications are homogeneous, but the operating system and technology layers can be heterogeneous.
- Inheritance of object interface information is used as a basis for interoperability. However, the introduction of autonomous domains has made dynamic knowledge acquisition necessary. This is reflected by recent request for proposals, especially for a trading service [169]. The current tools already include an interface repository that corresponds to a type repository function. However, this allows only one description language, IDL, and covers only access structure, not the behaviour. The traders could be used in selection of object implementations (locator service) as a match-making service, and the service offers could be stored in the interface repository [169].
- Binding of objects takes place based on object references. Early and late binding has separate tools in the infrastructure. The resulting channel is always static and the channel properties are not selectable or modifiable.
- Liaisons are not recognised in the model. Only interoperation relationships between ORBs can be considered as infrastructure liaisons.
- The applications are not expected to be aware of the distributed nature of the system.

When CORBA middleware services are built the following assumptions are made in contrast to the ODP model:

- The CORBA IDL is a globally accepted interface description language.
- The interfaces and the object implementations can be mapped together at run-time. This is called late binding and it can be supported by trading.
- The object instances or service types inherit knowledge of each other's properties. The IDL specifications heavily trust inheritance as a method for ensuring type safe communication between objects.

CORBA platforms are currently leading the development and the dissemination of open middleware services.

4.5 Comparison

The goal of this section is twofold: First, we compare the features of TINA, ANSA, and CORBA architectures using the ODP reference model as a taxonomy. Second, we analyse the extent in which these architectures support features required by distributed and federated systems. The discussions are structured to four aspects: object modelling (see Table 4.1), meta-information (see Table 4.2), federation mechanisms (see Table 4.3), and content of interface references (see Table 4.4).

Object modelling

Comparison of object models involves investigation of the set of separate concepts available. In ODP reference model, the following concepts are separated: object implementation, multiple interfaces on an object, and client and server role variants on the interfaces. Furthermore, the kind of interfaces supported is classified to operation and stream interfaces. Against this background, we can study the TINA, ANSA, and CORBA models.

All three models distinguish object implementations and object interfaces, although only ANSA separates client and server sides of an interface. Multiple interfaces per object are currently accepted by all three architectures, as CORBA model recently changed the interpretation of IDL specifications [172]. Operations are in all three cases supported by various RPC mechanisms, while streams are only being adopted. The stream interfaces have been of special interest in TINA architecture.

In addition to differences in object structure, there are also differences in what is modelled as an object. In CORBA and ANSA, only programming language objects are relevant. In TINA, also sovereign applications are of interest, but even then, the name package is used. The ODP reference model would allow packages to be considered as objects as well.

In relation to object modelling, the three platform architectures support more features that would be necessary for traditional distributed systems. For example, object implementations and interfaces are clearly separated, and stream interfaces are in most cases supported, at least as an extension.

The object model of federated systems is a superset of the object models in the three platform architectures. Especially, we should notice that the operations have more powerful semantics than RPC. In addition, the responsibility of meta-information and federations is placed on the infrastructure instead of applications. The RM-ODP includes all these features, except it does not allocate responsibilities between different objects.

Meta-information

We have concluded, that fundamental meta-information can be captured to binding liaisons and interface references. As components in these information objects we can typically find quality of service statements and policy statements. We have also indicated, that the meta-information exchange between objects or systems can takes place via functions like trading, type repository and binding. As part of the binding function, a separate liaison negotiation phase can be required.

ANSA model separates the concepts of binding liaison and interface reference explicitly, whereas CORBA uses an interface reference for both purposes. In TINA, the CORBA model is extended by explicitly introducing QoS contracts. In each case, the binding liaisons are considered to be static, i.e., constructed at system design or installation time, or managed by installing separate bridges between systems.

QoS contracts are present in the TINA model, and have been added to ANSA model as well. Policy frameworks are considered only in the ANSA architecture.

From the meta-information services, only the trading service has been adopted to all three platform architectures.

Traditional distributed systems require only implicit and static binding liaisons to be formed between objects or systems. Therefore, no other concept than interface reference is required. All three architectures thus have additional features to offer.

The requirements for meta-information model of federated systems exceed the features of the three platform architectures. They lack the ability for dynamic liaison negotiation.

Federation mechanisms

The federation model is strongly related to the liaison negotiation process. Therefore, the TINA, ANSA and CORBA models support the required features of traditional distributed systems, but do not reach the requirements of federated systems.

All three platform architectures present a static configuration of the systems involved, offering a unified set of platform service interfaces. The interworking relationships between objects are established based on engineering interface references. Therefore, there are no facilities for service liaisons between computational interfaces.

For federated systems, liaisons on the computational level should be supported separately. This would mean the use of computational interface references or separate objects for capturing service liaison agreements. The engineering level bindings would be governed by the computational liaisons.

Interface references

The technical content of interface references varies among the three platform architectures, but also among the various implementations of these architectures. Therefore, the variety of information components selected to interface references seems not to be fundamental. More importantly, the interface references are in each case allowed to carry extra information that is not manipulated by the native platform services. This allows use of foreign interface references in each platform environment.

Federated systems are able to negotiate all technical details of bindings. All features seem to be negotiable in at least one of the platform architectures. This means that the features are feasible and implementable.

Feature	**Federated system**	**Distributed system**	**ODP**	**TINA**	**ANSA**	**CORBA**
Separation of object and interface	YES	NO	YES	YES	YES	YES
Separation of client and server role interfaces	YES	NO	YES	NO	YES	NO
Level of object abstraction	programming language object (process) or sovereign application object	programming language object (process)	programming language object (process) or sovereign application object	programming language object (process)	programming language object (process)	programming language object (process)
Objects with multiple interfaces	YES	NO	YES	YES	YES	YES
Operations	interrogations and announcements	RPC	interrogations and announcements	RPC	RPC	RPC
Operations with multiple terminations	YES	NO	YES	NO	NO	NO
Streams	YES	NO	YES	YES	extension	NO
Responsibility of meta-information	on platform	on application	not decided yet	undefined	both choises allowed	on application

Table 4.1: Object modeling.

Feature	**Federated system**	**Distributed system**	**ODP**	**TINA**	**ANSA**	**CORBA**
Interface references	separate	joint with liaison	separate	joint with liaison	separate	joint with liaison
Liaisons	explicit, dynamic	implicit, static	explicit, dynamicíty undecided	implicit, static; for QoS explicit	explicit, static	implicit, static
QoS	YES	NO	YES	YES	extention	NO
Policy framework	YES	NO	YES	NO	YES	NO
Trading	YES	NO	YES	YES	YES	YES
Type repository	YES	NO	YES	NO	NO	NO
Liaison negotiation	YES	NO	not decided yet	NO	NO	NO

Table 4.2: Meta-information and meta-information services.

Feature	**Federated system**	**Distributed system**	**ODP**	**TINA**	**ANSA**	**CORBA**
Federation establishment	dynamic / static	static	not decided yet	static	static	static
Separation of liaison from channel	YES	NO	YES	NO	NO	NO
Platform interface	unified / personalised	unified	unified	unified	unified	unified
Liaison establishment at computational (C) or engineering (E) level	C, E	E	E, C not yet decided	E	E	E

Table 4.3: Federation mechanism.

Feature	**Federated system**	**Distributed system**	**ODP**	**TINA**	**ANSA**	**CORBA**
Interface type	YES	YES	YES	YES	YES	YES
Channel class	negotiatable	fixed	static choise	static choise	static choise	fixed
Location	YES	YES	YES	YES	YES	YES
Relocation	YES	NO	YES	YES	YES	NO
Security	YES	NO	YES	NO	YES	NO
Flow definition	YES	NO	YES	YES	extension	NO
Group	YES	NO	YES	NO	YES	NO
Causality	YES	NO	YES	NO	YES	NO
QoS	YES	NO	YES	YES	YES	NO
Standardized transport form	YES	NO	YES	YES	YES	YES

Table 4.4: Content of interface references.

Conclusion

TINA DPE is a traditional distributed system instead of a federable system, because it offers a homogeneous set of middleware services that is supported by a heterogeneous telecommunication network. However, TINA architecture includes federation features, e.g., it supports dynamic QoS contracts and static binding liaisons between software packages.

ANSA model is based on distributed system traditions, but in addition, it includes concepts of autonomy and federation. However, federation is considered to be a static agreement between systems. The model is basically a federated system model, but does not support run-time negotiation features.

Among the three platform architectures, the CORBA model represents the most traditional distributed system. However, functions that could support federation are being incorporated into the model. Still, CORBA does not include sufficient concepts for description of sovereign systems, nor sufficient middleware services. The essential restrictions are related to the interfaces, type, and subtyping concepts.

In this part, we have seen that a dynamic federation mechanism is an essential factor of open system architecture. We have also seen that federation mechanism trusts in meta-information embedded to contract schemata. Trading exchanges meta-information and thus forms foundation for system federations.

Part Three

TRADING

In this part, we study the logic of trading functionality and its realisations. The purpose of trading-based information services is to give a fast response to a request specifying requirements on a target whose identity and location are unknown. Trading is not a mapping mechanism like a name service giving a technology-specific address for a human-oriented name. Instead, the purpose of trading is to provide any answer that satisfies the query.

Chapter 5 discusses the various forms of trading mechanism. The discussion is opened with a characterisation of trading functionality as a global repository service. Then, the design problems arising in any trading community are discussed. Various forms of providing service offers to traders and retrieving offers from traders are analysed. As a separate theme, the solutions for a group of cooperating traders are studied. Within this study we rise a conceptual separation between distributed traders under a single source of controlling rules and trader federation where multiple controllers are involved.

Chapter 6 discusses services in federated systems required to support trading. Especially we are interested in type repository services and federated naming services.

Chapter 7 exploits the supporting services for two trader organisation models: one for a traditional distributed environment, and another for a federated environment.

Chapter 8 discusses some trader specifications and implementations, including ODP trading function standard and DRYAD trader software. The chapter is concluded with a comparison of trader realisations.

Chapter 5

Trading mechanism

In this chapter, we study the trading services as an independent service, disregard of its usage scenarios in various environments. The representation uses the terminology of federated systems and ODP reference model, and even specifies implementation requirements with the help of ODP engineering viewpoint concepts. Exploitation schemes for trading service are studied in Part IV.

We start the study of trading service by covering the design problems relevant for trading systems in Sections 5.1 and 5.2. The subsequent sections discuss potential solution strategies for these design problems. Concrete designs for trading services with differing strategic choices are discussed in Chapter 8.

5.1 Characteristics of trading

Trading presents a global information repository. The global repository can be updated by independent information producers throughout the world-wide network, and the information users can create effective and large information searches.

Trading is an additional technique for constructing information repositories, as are e.g. the group of database systems, object database systems, global name services, and directory services. The special features of trading include its global nature, the wide variance of application areas it can support simultaneously, the expected characteristics of load, and its ability to hide the complexity of heterogeneous, federated environment in which it operates.

Trading is expected to be a global and generic service that is able to give an equal service on various platforms, and moreover, across various platforms. However, the users of trading services have not only expectations on the semantical contents of the service but also expect certain technical characteristics. In a federated system environment, the global trading service may need to simultaneously support different interface characteristics.

Trading mechanism can be used to support various information services depending on the interpretation given to the mediated information. The trading mechanism defines available operations and syntactic structure for information that is manipulated by the operations. For the trading mechanism, the mediated information is just a list of properties of some entity. Only when the information semantics is further specified and the information manipulation is supported with other services along with trading, a meaningful information service is created. The trading mechanism can also support various application areas simultaneously and is easily adjusted to changes in the information schemata. The trading mechanism can be used for example in the areas of electronic commerce and software reuse. A simple example is a general search tools for browsing WWW-pages in Internet. Furthermore, the trading mechanism can be exploited by federated system middleware services, especially binding of computational interfaces and protocols that support computational interactions. This use is the intended application area of the ODP trading function [96]. The concepts of binding and computational interactions were introduced in Chapter 3, and are elaborated later in Chapter 9.

Trading also differs from repository services – like name servers, directory servers, and databases – in the assumptions made of the typical load profile. Traders are assumed to manage frequent updates and frequent queries. The set of trader clients that are allowed to update the repository is large, unknown and constantly evolving. Also the structure of stored information evolves. The actions in which traders participate are not necessarily transactions and they are intended to be small in respect to required memory space and required processing time. However, keeping the agreed response time is critical, even more critical than being able to do a full search in the repository.

The functionality is independent of selected platform technology. The mechanism is specified on a logical level that can be implemented on a variety of platforms, programming languages, storage techniques and communication protocols. Due to this design choice, trading can be widened to a world-wide service: all trader realisations make their technical decisions independently, only considering the logical design and preparing to intercept technical differences. In some technology oriented aspects, general guidelines are required. However, such guidelines are highly flexible, as they offer multiple alternative implementation techniques. For example, traders may exchange information in object format or as lists of named values.

Realisations of the trading functionality have to solve various problems. Trading makes information available to a large but still controllable set of users. The qualities of the available information should fulfil the user expectations: information consistency, freshness, and accuracy may be required, access times may be essential, and high probability for information availability may be crucial. However, the selected techniques may not affect the autonomy of the information producing systems nor the information user systems. Excess of network traffic must be avoided, as well as creating new security threats to the systems. Furthermore, the trading mechanism should be able to adopt to the evolution of the mediated information contents. Moreover, the topology of trader network that cooperate for the global trading service is constantly changing. Finally, federated environments create a special area of problems. The federable systems have autonomous administrations and different technologies, and evolution schemes.

5.2 Trading concepts

This section studies problems that are general to all trader constellations. Traditionally, a service or a mechanism is represented as a concrete algorithm, optimised for a selected platform. In this case, the description continues listing alternative ways of solving the various problems arising from the functionality. This is an implication of the properties of the environment in which the trading services are used. First of all, the trading system is required to deal with the numerous platforms to the benefit of application software. The purpose of trading is to hide that complexity. Secondly, the required characteristics for traded information vary from administrative domain to another, but still the traders are required to interoperate. In the case of a single trader, a single technique is selected for the use of the whole community.

5.2.1 Trading community and trading domains

Trading activities are described through a trading community that represent the roles of 'importer', 'exporter' and 'trader' [96], as illustrated in Figure 5.1. The object in trader role supports a repository of 'offers'. Each offer describes properties of an entity. The offers are produced by objects in exporter roles. When an exporter sends an offer for a trader to be stored, it is said to 'export'. When an exporter requests an offer to be deleted, it is said to 'withdraw' an offer. The objects in importer roles make queries to the offer repository. The query and the response to the query form an interaction called 'import'. The import request has a basic form that is similar to a database query: the request specifies criteria for selecting the offers to be included to the response. The objects that use the traded information are not necessarily the importers or exporters themselves. The 'clients' and 'servers' that use the information are therefore drawn as separate roles in Figure 5.1.

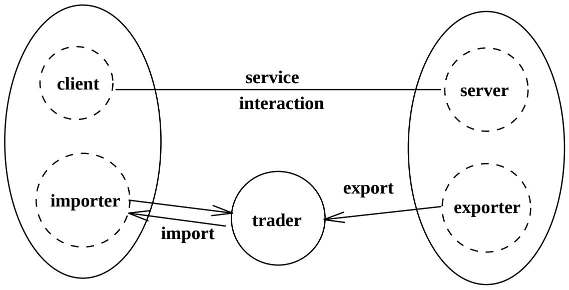


Figure 5.1: The trading community.

Community must be understood as a design pattern that specifies roles for objects that participate in the activity. The use of the term community is consistent with Definition 3.19. The community is therefore a dynamic construct, as the roles can be fulfilled by different set of objects at different times. Also, a role can be filled in by a set of objects.

The trader role can be distributed to a set of autonomous trader objects. Each trader object controls a private trading domain. This arrangement has two benefits: first of all, each federable system in the federated system environment has a private agent in their possession. Secondly, the potentially huge trading community is divided into smaller partitions and thus the load on each trader object becomes reasonable. The organisation of trading domains is illustrated by Figure 5.2.

For the trading community, it is essential that the property set specified for the interesting service type is equally understood by all objects. Furthermore, it is essential that the property names and property value sets have similar definitions and the semantical interpretation does not differ. In the case of a single centralised trader object this seems trivial. However, when we study cooperation between traders in Section 5.3, this requirement is no more trivially granted. A consistent set of service types and properties within service types can be standardised, which will enlarge the set of potential members in any trading community. However, standardisation does not provide a solution that would be sufficient alone, because of the long evolution delays involved. An additional mechanism for dealing with the semantic consistency of service types and properties is shown in Section 6.2. The same mechanism can be used between an importer and a trader that reside at different administrative domains.

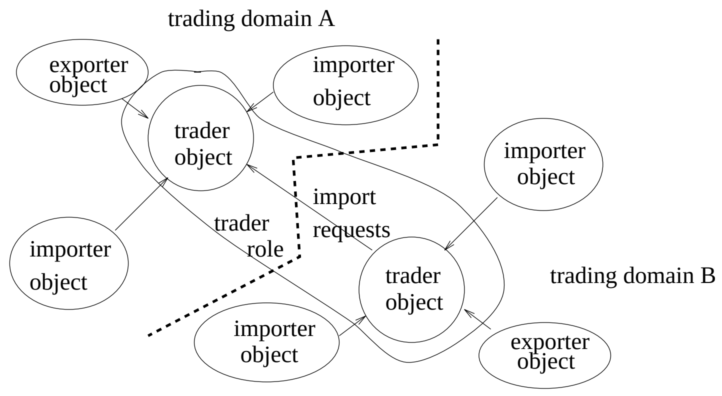


Figure 5.2: The trading domains.

The traders act as controlling objects within the trading domains, but only in relation to trading activities. Other aspects related to the exporter objects and importer objects are not restricted by the trader objects. Only the communication rules are specified. As a consequence, having a binding, i.e. communication channel, between importers and trader or exporters and trader does not necessarily mean that all the objects would share their technology choices. For example, data representation formats may be different. In such cases, the channel between the objects must contain interceptors to transform between the formats involved. Trader objects manage many of communication technology problems. The purpose is to hide such complexity from the users of trading services. However, dealing with communication technologies is not part of the trading logic, but is further embedded to the binding establishment process.

Each importer or exporter object is bound to a trader object in order to have access to the service interfaces. An object can be bound to multiple traders as well, but then it is considered to enter the same role in multiple domains.

Each offer is stored primarily to one trader object only. If an exporter object plays a role in multiple trading domains, the created offers are considered to be individual instances and managed accordingly. Therefore, it is reasonable to claim that an offer is owned by the trader object to which it is exported. The offers may be copied to other traders in order to provide fast access to information that it frequently used. However, the consistency of such copies is not necessarily supported.

Traders are not responsible for the information quality they mediate [96]. It is the responsibility of the exporters to provide trustworthy information. As the traders have no method for controlling the information quality, the importers must evaluate that themselves. Most importantly, the traders do not invalidate any imported offers later on. Traders are not responsible of tracking who uses the offers. Some users import a set of offers, store them, and use the information later. At the time the offers are used, the information may be outdated. The users are themselves responsible of requesting new offers frequently enough. However, fresh offers do not always guarantee that the property values would represent current system state. For example, property values expressing remote system load are never 'current' because of network delay.

We refine the community activities and object roles in the following sections. First, we analyse the management of information within the trader. Then, we analyse the exchange of information between the exporters and the trader, and the importers and the trader. Finally, we analyse how importers and exporters are allowed or denied access to the trader.

5.2.2 Offers

Exporters, importers and traders pass logical information objects, offers, to each other. An offer contains property values for a named set of properties. The set of properties is specified separately for each type of service. For example, for a service type 'compiler' the interesting properties are 'sourcelanguage' with data type 'set of strings', 'host' with data type 'string'. Another service type, 'travel' has properties 'destination' with data type 'set of strings', 'duration' with data type 'integer' together (implicitly interpreted as number of days), and 'cost' with data type 'integer' (implicitly interpreted as number of Finnish marks). Example offers of these types are

```
servicetype: compiler, sourcelanguage: "C", host: "hydra",
servicetype: travel, destination: "Hawaii",  duration: 7, cost: 8000.
```

Type of offer

It is also important that the objects participating the trading community have equal expectations on the set of properties used. A slight variance on the property set can be however allowed. The properties can be defined to be mandatory, optional, or additional.

Mandatory property values must be always present in an exported offer. Retrieving the property value may still fail under exceptional environment conditions, for example when executing an evaluation is delayed too much.

Optional properties must be defined in a consistent way through the trading community. They are not necessarily present in all offers, but if so, the semantics must be the same. The optional properties support transition to new sets of mandatory properties.

Additional properties have no restrictions. The exporter may define semantics for an additional property and pass such property values to the trader in the hope that an importer has knowledge about this kind of 'private' property values. This can be considered as a method for exploiting vendor dependent, de-facto standardised properties prior to actual standardisation efforts.

Quality of information

The information users have expectations on the quality of information traded. Depending on the application area, these expectations focus on different aspects. For example, in a telephone switch, the phone number register does not change very often, but the numbers are requested frequently. The characteristics required for the phone numbers focus on the fast access, and high availability. However, change processing is not as critical. Even data consistency between different parts of the world-wide system may not be a critical requirement. In the telephone switch example, a non-consistency might be allowed in case of a person moving from an address to another and having a phone connected to both addresses simultaneously for some days.

The traders are not themselves responsible of the information quality, but provide several management techniques for properties. These techniques support different qualities of information. For each property, the technique is separately selected by the exporter, based on assumptions of the change and request frequencies on the information.

Management techniques for property values

The forms of property values are static, dynamic, and modifiable [96]. These forms allow two aspects. First, the information user may be aware whether the value is prone to changes over a period of time. Second, the information producer may either push new values to the offer itself, or the trader requests a new value when needed.

Static property values suit for non-changing properties of an entity. For example, a printer is able to use certain paper sizes only. Static property values are also adequate in situations where the offer is exported and withdrawn frequently, thus refreshing the property values. For example, a meeting room may be offered for each period of available meeting time separately.

Modifiable properties are like static properties, but the exporter can overwrite an existing property value without withdrawing and re-exporting the whole offer. This technique gives an opportunity to refresh values with an atomic operation. This technique also allows other sources for the new values. For example, a trader administrator can change the list of available paper sizes at a printer. In a single trader case, this potential does not seem to be of interest. However, in certain cooperation constellations between traders, the information sharing technique may be used in a more optimal way. These problems are further discussed in Section 5.3.

Dynamic properties do not have actual values stored within an offer. Instead, the offer includes an interface reference that can be used to invoke an operation to evaluate a new value for the property. This technique is usually used together with a caching mechanism. A printing queue length is an example of a dynamic property.

The possibility of using dynamic properties is one of the key features that distinguish traders from directory services or name services. Many trading applications exploit this feature. For example, within the process of binding a client to a server interface, it is possible to select an interface that is available at the time, and not only available in principle.

Dynamic properties can also be used to allow the servers themselves to participate the selection process. The processing involved in the dynamic property evaluation can include, for example, rules for submitting the client identification information to the server for approval. In this way, the server can approve the import, before the offer is revealed to the client.

Static properties are secure to use. Therefore, when security is especially important, either importers or traders may restrict the offers to static property values only. The dangers of dynamic properties are related to the evaluation mechanism. The evaluation of a new value may cause some information about the importer to be spread into the system that supports the trader. The evaluation of a new value also causes execution of program code that may originate from some other system than the one that supports the trader.

Offer management information

Offers contain not only property values, but also information that is used by the trader in order to manage the offers themselves. This information is collectively called as offer properties in

```
* an offer identifier,
* a service type identifier,
* exporter policy, including access rules or interface for
  executing access rules, and time-to-live values,
* a flag  differentiating between normal and proxy offers,
* in case of proxy offer:
        - an interface reference,
        - instructions for creating a request to be sent to the interface,
* in case of normal offer a set of items containing:
        - a property name,
        - either a property value, or
          an interface reference and instructions for invocation.
```

Figure 5.3: General offer structure.

distinction to properties. To the management information we classify here proxy offers, exporter policy information, and offer identifiers.

Sometimes the property values for an offer are too dependent on each other so that dynamic evaluation of the individual property values would be sensible. Therefore, the trading mechanism allows to choose between normal offers and proxy offers. A proxy offer includes an interface reference that can be used for creating a set of offers. The proxy offer also includes instructions on how to build the creation request. The request is intended to carry as much information about the importer and the matching criteria as possible. The benefits of proxy offers are found in environments where individual offers change often, they have inherently changing property values, but for the benefit of import performance the property values have to be static. Of course, this mechanism has all the drawbacks of the dynamic properties.

The exporter is allowed to express restrictions on the users to whom the offers are revealed. This can be done in two different ways. The first method is that of using proxy offers. The process of creating new offers to correspond the proxy offer can use imported identity for deciding whether offers are made available. The second method uses a property that is aimed, not at the importers, but at the trader to use. This offer property expresses instructions to evaluate whether the offer is available for the identified importer or not. The instructions may involve communication with an external interface, or the format may be that of an access control list. A simple way of restricting importers is to define an offer to be valid only for a period of time. The trader is expected to automatically withdraw the offer after the offer has expired.

For the organisation of the offer storage, and for the communication between exporters and trader, we use an additional attribute for identifying the offer within the trader.

5.2.3 Offer storage

Logically, the offer storage consists of a set of offers and proxy offers [96]. The content of both types of offers is summarised in Figure 5.3.

The semantic structure of the offer storage is determined by the set of service types represented. Each offer includes the set of properties specified by a service type and is therefore understandable only within the context of that type. These structures can be separately standardised for each application area of trading. However, changes to the structures and introduction of new application areas would be expensive. Therefore, a separate type repository service is exploited for supporting the structuring information.

The physical structure of the offer storage is not required to follow the semantic structure.

The offers can be stored separately depending on their service type, or the offers can be further subdivided into groups that fulfil the most frequent matching criteria in import requests. Also, the offers may have identification keys as additional attributes for indexing purposes within the underlying database management system.

The offer storage can use any suitable technique. Databases are frequently used for storing objects but file servers, name servers, and memory servers can be used as well. The technique must be in conformance with the quality aspect of the information stored.

Database management systems are robust and they efficiently protect the information consistency, but they have high overhead for these properties. Therefore, database solutions may be too slow. In most cases we feel that a 'light-weight database' solution is required, because the mediated information is typically changing rapidly. We discuss the storage methods further in Section 6.3.

Technically, the offer storage contains also a cache. The cache contains recently manipulated offers and recent values of offer properties. The cache has an important role in the design of search algorithms that span multiple traders. However, the cached offers are not logial copies of the stored offers. Thus, no update nor invalidation mechanisms for the cached offers is needed.

5.2.4 Exporting, withdrawing and modifying offers

The offer storage can be modified by several interactions that do not involve exporters and traders alone, but can be invoked by other authorised objects as well. We start by studying offer creation, continue by offer modifications, and close by offer disposal techniques. This set of interactions is complete only if we assume that the service types are not modified. Modification of service types would enforce modifications on the offer storage, but we postpone the discussion of schemata changes to Chapter 7.

Creation of offers

The trader creates an offer at the request of an exporter [96]. Typically, the exporters are bound to a trader when instantiated, so there is no problems for the exporters to find a suitable trader. In the same way as normal offers, an exporter can create a proxy offer.

The exporter must give all property values or evaluation rules in the export operation. The logical information object may be represented as an actual object in systems where objects can be communicated, or a set of name-value pairs from which the trader can create the offer. Although the offer is logically an object, it is not compulsory for the trader to implement the offer as an actual object. Important is that the exporters and the trader find a common protocol for transporting either the objects or the corresponding data.

The trader may restrict the set of exporters and offers it accepts. The trader may refuse to store offers from exporters it has no record of, or it may serve only exporters owned by the same company as itself. The trader may also protect its storage space by accepting only offers with a certain service type, or certain expected values. For instance, a trader may restrict its offers to cover printers only, and require that the printers must be owned by a certain department. The latter property can be demonstrated by including a property with an encrypted password that is known only to the department administrators.

When an export operation is accepted and an offer is created, an offer identifier is created by the trader. That identifier is returned to the exporter. The identifier can be used for referring to the offer later by withdraw and modify operations. The exporter object can pass the identity to other objects, so the identifier may not serve as any kind of credential.

When the exporter creates the values for the offer properties, it is important that the language or the value set of the properties is commonly known within the trading community. For certain property types some languages have already become natural. For example, if we trade for server interfaces, one of the properties expresses the interface type. This is often expressed in IDL. Furthermore, the interface references expressed within the offer for evaluating new property values must be understandable by the trader. The interface references are specific to the platform on which the objects operate.

Proxy offers can be used to produce a set of offers with static properties. The set of created offers is primarily produced for the use of initiating importer. The set of created offers can be as small as a single offer that is immediately consumed by the importer and therefore the offer is not even temporarily stored to the offer storage. However, the exporter policy produced for the new offers may allow access for other importers as well.

Modification of offers

Facilities for modifying the stored offers requires that clients are able to identify each offer for the trader. Therefore, the trader returns an offer identifier for the exporter as a result of an export operation. That identifier can be used as a key for all interactions that modify the offer later on. The identifier can be passed on to other object by the exporter object and therefore the set of modifying objects is undefined. Passing the offer identifier has no meaning as capability passing technique. The offers are protected against unauthorised access even when a correct identifier is presented to the trader.

The technique for offer modifications is dependent on whether the property values under modification are static, dynamic, or modifiable.

For changing static properties in an offer, the only technique is to replace the offer. Traders may encapsulate a pair of withdraw and export operations to an atomic interaction 'replace'. 'Replace' overwrites all property values of an identified offer but maintains offer identity [96].

For dynamic properties, modifications are automatic. The property value itself is specified indirectly, as an evaluation rule. Therefore, new values can be retrieved by invoking the evaluation.

For modifiable properties, a new property value is explicitly pushed to an identified offer by a 'modify' operation. 'Modify' differs from 'replace' by changing only individual property values instead of the whole contents. 'Modify' can also modify the number of optional or additional properties present in the offer [96].

None of the modification operations is able to change the property category, e.g., a static property cannot be changed to a dynamic property [96]. The modification operations are neither able to change the data type of the property value. These restrictions are necessary for the consistency of the mechanism itself, and for the security of the trading services. Changes between different property categories affect the information quality that must be controllable by the exporter object. Changes between data types would easily cause semantic confusion between exporters and importers, and moreover, would allow a bypass of the restriction of non-modifiable property categories.

Proxy offers cannot be modified except by a 'replace' operation.

Disposal of offers

The disposal of offers can be initiated by either an object holding the offer identifier or by the trader. The trader can act based on offer properties, and other objects can invoke a 'withdraw' operation. 'Withdraw' is protected by access restrictions similarly to offer modification operations.

The goal of withdrawing an offer from a trader is to stop importers using the information revealed in the offer. For example, if a service interface is offered, the destruction of that interface may induce the withdrawal of the corresponding offer. However, the importers may already hold copies of the offer, or clients may already be using the offered interface. Trading does not solve these problems. Withdrawing an offer only guarantees that no further imports can reveal the offer.

Proxy offers may produce a set of normal offers that are protected by exporter policies. The offers created in this way cannot be withdrawn by any object but the trader. Therefore, if the produced offers are stored, they must include a time-to-live value to allow the automatic disposal of the offer. If the object referenced in the proxy offer does not provide this value, the trader must create a suitable value itself.

5.2.5 Import

In this section, we discuss the interaction between importers and traders. The 'import' operation allows importers to retrieve offers from the offer storage and also to guide the search. The purpose of 'import' operation is to find fast some offers that satisfy the query, without a hard requirement of finding the ultimately best choices.

Characteristic to the import interaction is that the importer guides the search simultaneously on two levels: On the logical level the importer specifies which offers are interesting; on the technical level the importer specifies how extensive search is acceptable, and how much space the result of the search is allowed to take. The inclusion of the technical level guidelines may appear irrelevant, but when a search is forwarded around in a world-wide network, such considerations become relevant. However, most importers can trust that the trader to which they have been bound to adopt a reasonable technical behaviour without additional specifications. This assumption can be based on the fact that the administration of the local trader for an importer is most probably from the same organisation.

The import is parameterised with service type, matching criteria, preference criteria, ordering criteria, search scope and cardinality restrictions, and specification of the contents and cardinality of the information to be returned. These parameters are defined shortly. The service type, the matching criteria and the preference criteria guide the logical level of the search. Other parameters guide the technical aspects of the search. Import is not parametrised by the trader address: the importer does not need to search for a suitable trader. An 'import' interaction can be initiated by any object bound to a trader via a communication channel.

Logical specification of the search

The logical offer selection rules fall into two groups, compulsory and prioritising. The compulsory rules, i.e. matching criteria, specify a set of acceptable offers. The prioritising rules, i.e. preference criteria, express that some of the acceptable offers are more interesting than the others. The selection rules are expressed in criteria languages that use, for example, arithmetic and Boolean operators over property values in the offers.

The only necessary information for initiating the import is the name of the service type of the searched offers. The service type specifies the properties present in the offers, and therefore also specifies what property names can be used as part of the criteria.

The matching criteria are formulated of expressions considering property values within a single offer while preference criteria compare property values of a set of offers. The returned offers must fulfil the preference criteria if enough offers can be found that way – if there is lack of preferred offers any matching offer is acceptable.

Criteria languages

The matching criteria specify which offers are acceptable for the response to the importer based on the properties within each offer alone. Suitable expression components for a criteria language are

- property names and values,
- literals within several data types,
- existence tests for properties (exists),
- comparisons involving properties, literals or both (equal, almost similar string, all ordering relationships),
- Boolean expressions (and, or, not),
- basic arithmetic operations (add, subtract, multiply, modulus, divide), and
- tests for membership in sets (in).

The preference criteria specify the priority of some offers over others. The criteria language applied for preference criteria must be able to express relationships between offers based on one or more property values present in each offer.

A set of ordering functions can be used to compare offers against each other. Suitable expressions are

- ordering functions 'min' and 'max' parametrised with the property name used as ordering key,
- 'random' for any order,
- 'first' that means first of a sequence ordered by an implementation, and
- 'with' parametrised with a boolean-valued expression to give priority to those offers producing a true value for the expression.

The function 'with' actually allows using the full criteria language as part of the ordering language.

The actual languages used for the matching and preference criteria may vary from trading domain to another. However, standardised languages are recommendable for enabling interoperation between domains, although any language that has the same expressive power can in theory be intercepted to the standard version at execution-time.

Examples of acceptable import criteria are:

```
(servicetype=compiler, sourcelanguage in ["C", "C++"], host="hydra"),
min(servicetype=travel, destination="Hawaii", 1 <= duration <= 3, cost < 10000)
```

Technical specification of the search

The problem in import activities is that in a world-wide environment we cannot predict the extent of the search. Still, in many cases, trading is required to have properties of a real-time system. Therefore, we must at least ensure that the import operation terminates with a reasonable result in a guaranteed time. We refer back to interrogations with QoS contracts in Section 2.4.

The technical search specifications also fall in two categories, those of restricting the resources spent to the search by the trader, and those of restricting the resources consumed by the importer when receiving the resulting set of offers. The term used to refer these restrictions jointly is 'import policy'.

The resources spent by the trader can be limited by restricting the number of offers considered, the time spent in search, the number of suitable offers found, and the amount of money spent for the search. Reaching any of the given limits on these items terminates the search.

The resources spent by the importer in receiving the results mean memory space. The size of the response must be small enough for the importer to store, but still the essential parts of the search result must be captured. Therefore, it is not enough to just restrict the number of offers returned. Instead, the offers must be first ordered for minimising the number of interesting offers dropped out just because of memory restrictions. In addition to restricting the number of offers, the importer can also explicate which property values in the offers are of interest. This gives an opportunity for minimising the size of returned offers.

The technical search restrictions are expressed in ordering criteria, search scope and cardinality parameters, and specification of the contents and cardinality of the information to be returned.

Expressions for technical restrictions

The technical search restrictions are expressed in ordering criteria, search scope and cardinality parameters, and specification of the contents and cardinality of the information to be returned.

For the ordering criteria the goal of the expressions is similar to that of preference criteria, and therefore, the same language can be used. This often leads to unification of the two parameters as we will see in Section 8.1.

The scope parameter is in the case of a single trader a technical matter dependent on the storage technology selected. Such restrictions are not generally applicable and there are no generic rules for them. However, scoping in a network of traders is more important, and we discuss the problem in Section 5.3.

The search cardinality parameters and return cardinality parameters are simple expressions with a standardised parameter name and a limiting value. For example, a search that should only walk through 10000 offers and use at most 20 seconds for doing so can be restricted by the following expression:

```
maxofferstosearch = 10000; maxsearchtime = 20
```

For specifying the contents returned for each offer a list of property names can be used. A special term can be reserved for denoting 'no properties' and 'all properties'.

Search from the repository

The search from the trader's storage is guided not only by the importer via the parameters in the import request. For each trader, there are internal rules that restrict the searches in respect of resources spent for a single search. In addition to these restrictions that protect the trader itself from overload, the trader can support default restrictions for importers that are not willing to consider the technical parameters of their import requests. Another area of restrictions arises from the visibility of offers for separate importer groups. The restrictions made by the trader itself can be expressed with the same languages as the import parameters.

The search process invoked by the import request can be split to multiple, simultaneously running sub-searches in different parts of the storage. The searches terminate either because of reaching the final result of a complete search, or because some of the technical restrictions specified for the search deny continuation of the search. The termination reasons are of equal importance – termination because of reaching technical limits is not considered as a failure. In the following, we first study the search algorithm and then study the semantics of search results in case of different termination reasons. Figure 5.4 represents the propagation of a search within a trader working in isolation of other traders. The search algorithm is based on multiple simultaneous sub-searches, each of which is responsible of a separate partition of the offer storage. The storage partitions may include various indexes for organising the access of offers. Even partial results of searches can be indexed as well as final search results. Such indexes can be exploited for reusing previous search results to speed up new import activities, given that the results are not used too long. If the offers contain dynamic or modifiable values, re-evaluation or modification of properties should invalidate the indexes as well. The sub-searches are monitored by an external process that controls the total resource consumption of the search and terminates the sub-searches when any of the resource limits is exceeded.

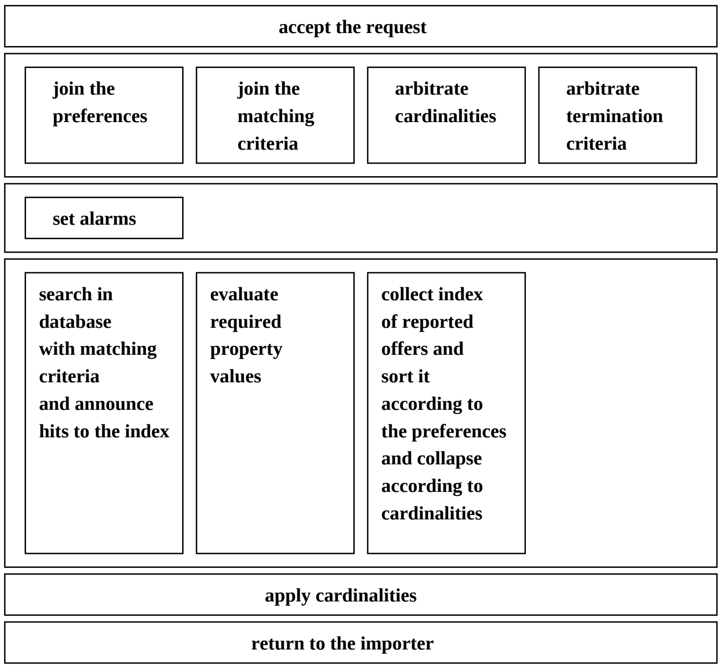

accept the request
join the preferences
join the matching criteria
arbitrate cardinalities
arbitrate termination criteria
set alarms
search in database with matching criteria and announce hits to the index
evaluate required property values
collect index of reported offers and sort it according to the preferences and collapse according to cardinalities
apply cardinalities
return to the importer

Figure 5.4: Search algorithm used within a single trader.

During the search, it is not necessary that all property values in the offers are known. The dynamic properties can be evaluated at reference. The search process can invoke the evaluation asynchronously and let the offer to be studied again later. A simple mechanism uses synchronous evaluations, but the evaluation time may be long and especially if the number of sub-searches is low (even just one) the number of offers considered becomes very low.

In some failure cases, it is possible that the referenced property value is not available and cannot be evaluated. In the evaluation of matching criteria and ordering criteria, such values are treaded so that the offer is disfavoured. In preference criteria, the missing value has no critical role.

In the search algorithm, proxy offers create a special case. When a proxy offer is met during the search, there are two choices depending on the mission of the proxy offer evaluator. We will see in Section 8.1 that the proxy offer evaluator can be used for creating new servers and reserving resources for the offered service. In this case, the proxy offer may not be expanded unless the resulting single offer is actually taken as the only result of the import operation. If the proxy offer evaluator just produces a set of normal offers without any resource consuming side effects, the created new offers can be considered as an additional partition of the offer storage. In this case, the offers are searched in the same way as normal offers.

Interpretation of the import result

The search can be terminated either because the whole offer space has been searched or because a resource limit is reached. In both cases, the trader has collected a set of offers, and the termination of the import operation is considered to be normal. Termination due to scope limits is flagged separately, but such termination is not interpreted as a failure. Similarly, missing attribute values are flagged, but the trader should not discard incomplete offers if they fulfil the matching criteria. It is the privilege of the importer to decide whether such offers are useful or not. The usefulness of incomplete offers is dependent of the application area – for example, a missing author name in an offer that describes a WWW page is not a problem, as long the address of the page is available.

The search fails only if the search process is terminated by a programming failure, configuration error, computer crash or any other data independent reason. The trader probably is unable to give that kind of exception termination, but other parts of the distributed infrastructure, especially components of the channel between importer and trader, should notice and report the problem.

5.2.6 Trader administration

The administrator can decide trader behaviour related to export actions and import actions, and also define the set of service types used.

The rules for the trader behaviour in import actions are collectively named 'import action policy' [96]. They can be either stored or hard coded into the implementation. Import action policy includes rules for accepting import requests. This is different from the access control on the interface: the rules can operate on import parameters instead of importer identity only. In addition, the import action policy includes counterparts for all import parameters either as maximum values or as pairs of default values and acceptable maximum values.

Similarly, the rules for the trader behaviour in export actions are collectively named 'export action policy' [96]. The main rule restricts the accepted exporters and offers. Additional rules can guide the internal organisation of offer storage.

For import actions, there are two sources of criteria, the trader executing the search and the importer. Arbitration policy defines the merging rules from these two sources. Depending on the arbitration policy the trader assumes a different role in the computing system.

Several trader roles have been presented: a match maker, a dispatcher, a trustee, a consultant, a coordinator [254], or a secretary. The different roles of a trader are related to the different sources of matching and preference criteria given for an import operation. The match makers are the simplest trader systems: They work on static offer properties, while traders of other roles can utilise also dynamic values. For example directory servers can be utilised as match makers. Dispatchers and trustees take import criteria only from one source. The dispatchers accept only administrative arguments for import operations, while the trustees depend on importer's arguments. Consultants and coordinators use both administrator's and importer's arguments, but their arbitration policies differ. The consultants run their own selection arguments first and complete the selection by importer's arguments. The coordinators allow importer's arguments to start the selection, but restrict the selection further with their own rules. The final role, secretary, is a combination of the other roles. Secretaries allow the trader's role to be dependent on a set of aspects: the importer, the service type, and the selection and preference criteria. The combinations can be so rich that the trader's role is effectively different in each import operation.

The set of service types in use can be defined indirectly by a reference to a type repository. Another aspect of service types is the support for advanced subtyping relations. Some type repositories relate two service types as logically equal, even if they have minor technical differences. For importers and exporters the knowledge of such facility is important. For example, when trading is used as part of the binding process between object interfaces, it is essential to know whether interceptor should be supported.

The administrator can also manipulate trader's knowledge about other traders. A trader needs information about other traders for negotiating federations. The traders must be able to obtain this information explicitly, as the traders themselves should offer this service to other objects in the system. The information can be inserted by administrators, or communicated by other traders. In the following section we discuss trader federations.

5.3 Cooperation of traders

As the world-wide federated system potentially creates an enormous number of offers, the trading service must be partitioned to domains, in order to scale the service appropriately. Moreover, the importers and exporters are supported by sovereign federable systems. Therefore, there is at least one trader for each sovereign system involved in the world-wide federation.

For scaling reasons, a trading domain can be further divided to sub-domains, or the traders can be replicated to share the load of import requests. The trader can also be replicated for availability. However, these reasons for creating multiple traders do not result into trader federations, but into distributed trader solutions. The difference between these concepts is in their administration: federated traders do not share their administration while distributed traders do.

This section studies problems specific to the cooperation of multiple traders, i.e., how interworking can be reached in cases where different techniques have been selected at the trading domains.

5.3.1 Trader federation vs. distributed trader

In order to further clarify the distinction between distributed traders and trader federations, we study several aspects of the trading administration and communication technology.

In the previous section, we concluded that the trader administrator can restrict access for trading services, and specify the set of service types understood within the trading domain. The service type specification further declares the properties in each offer, property names and property value sets. The administrator can also define criteria for matching, preferences, ordering and cardinalities to be obeyed in each import action. Moreover, the platform selected to support the trader determines the technical representation of offers – whether they are represented as objects or just name-value pairs – and the transport protocols used for data communication.

In case of distributed traders, all trader objects involved obey the same rules in all of the above mentioned aspects: they use the same data communication protocol, they use the same service types, same property names, and they govern import actions with the same criteria.

In case of trader federation, all trader objects involved may independently make decisions on these aspects. Such trader objects that share some of the service types and are able to transfer information among themselves can forward import requests and results among themselves. A major problem is to find suitable traders to cooperate with and establish the federation. Furthermore, the techniques for delegating imports and collecting import results are nontrivial because of the heterogeneity.

A distributed trader constellation is rather static in nature. The trader objects are assigned to a group of computers and expected to work together. Dynamic behaviour occurs because of failures and failure recovery. In contrast, a trader federation is very dynamic in nature. In an extreme case, each import operation initiates a new graph of traders.

The purpose of trading federations is to allow a world-wide process for import actions starting from any trader object. The trader federations do not actively propagate exported offers. The purpose of distributed trading is to create an available, trustworthy trading service. The traders can export and import from each other and also have a logically centralised administration. They can also have a decentralised mechanism for establishing a cooperation constellation.

The level of autonomy and heterogeneity of federated and distributed traders differ. A single administration provides a distributed trader constellation. In such a constellation, the traders can have separate failure recovery mechanisms, separate access control mechanisms, concurrency control and storage systems. However, the policies controlling these things are equal in all traders. For example, although the access control mechanisms are separate, the permitted access group is the identical set of clients. The existence of several administrations forces the use of a federated trading constellation or a federated infrastructure that supports a distributed trading service. The traders can use, for example, separate type systems. Chapter 7 studies two methods of constructing a trading service in the presence of federation requirements.

5.3.2 Federation establishment

Establishment of trader federation starts from finding out suitable cooperation partners. This can be achieved by active search or by using previously stored information. During the federation establishment, the suitability of the partners is considered in terms of type systems, name systems, security systems, and information transfer capabilities. Problems created by differing import action policies are handled at import processing time.

Acquiring information on other traders

Traders can learn about each other by three different techniques: introduction by administrator, search from a global repository of well-known traders, and active exporting of trading offers to other traders.

The first technique, introduction by administrators, is simple to implement. The administrator creates a list of known traders including their names, addresses, type repositories, and other details. This technique suffers from the number of potential traders and the burden of maintaining the addresses as they change. The more information is included about each trader, the more human work is involved; the more information needs to be retrieved from the remote trader at run-time, the more time each federation establishment takes. In a world-wide environment, this technique does not scale enough. The development of Internet name services can be used as an analogy for predicting the potential behaviour.

The second technique, search from a global repository, is an improvement of the first technique. The Internet name services were initially supported by locally administered lists of all hosts. Later, a hierarchically administered domain-based system was introduced. In a similar way, traders can be organised to a hierarchy. However, trading is designed to help in searches where the name and location of the search target are unknown. Forcing a strict name-based or location-based identification of the search engines involved would contradict this goal.

The third technique, active exporting of trading offers, utilises the trading mechanism itself to spread information about traders. In this technique, a new trader gets a small set of trader addresses from its administrator. The new trader then exports an offer of its own interfaces to these traders. The traders holding that offer may be allowed to reveal it either by further exports or by giving it as a response to an import for trading services. The trading offer is under the control of the new trader object through the exporter policy that may restrict the offer visibility to importers. The technique resembles probabilistic sparse diffusion multicast [206]. The problem of this technique is the difficulty of finding out the full set of traders – in some respect, for example, all traders including offers of a certain service type.

Techniques for storing knowledge about potential federations

Once information about potential cooperation partners is found, it must be stored for use. Again, three different techniques are presented: a separate trading offer storage [123], links [96], and proxy offers [44, 96].

The trivial technique is to store trading service offers as they are received from other traders or from the administrator. However, establishing a communication relationship based on the unprocessed information may be too slow.

Links are pointers between traders. Links can be created by trader administrator, or by the trader object using trading offers. A link includes the remote trader's name and address, and information about the service types understood by the remote trader. The link may also include translation rules for mapping between different type systems. In addition, private information about the quality of the trading service available can be stored. The use of the link can as well be restricted by a set of criteria. The criteria can contain statements of the access rights and acceptable import requests to be delegated through the link. We will study the effect of link criteria when we study the propagation of an import action through a graph of traders.

Proxy offers can serve as simple links. Proxy offers allow offers to be retrieved from any external object in a similar technique independent whether the object is a trader or not. Proxy offer has an interception mechanism embedded, as it includes a recipe for constructing the service invocation message from the available import parameters. Proxy offers can be used, for example, for integrating directory services or database services to traders.

Federation constellations

As a constellation, trader federation is a trading community where trader objects assume the importer roles and trader roles. One trader object resolves part of an import request coming from a local importer object by delegating the import to another trader. The protocol between the two traders is the same protocol used between any importer and trader.

Another community is created when the traders search for suitable cooperation partners. The trader objects assume the roles of 'exporters' and 'traders' when a new trader joins the network of traders. The trader objects assume the roles of 'importers' and 'traders' when they search for new traders from which they could import.

These two communities should not be mistaken as being both part of a trader federation. In this section, we consider only the trader federation community that handles import actions.

In the trader federation, each trader object processes import actions in isolation. Although the importer may be a trader object, the requested import is still processed using the same rules. Only two additional problems are induced by the federated import processing. First of all, the access control mechanism must be aware of the federation scheme. We discuss access control in more detail in Chapter 7. Second, the importer is able to guide the search through a set of traders by using a scope criteria parameter in the import request.

The trader federation has a very complicated and very dynamic topology. Starting from each trader, an edge can be drawn to all those traders that are reachable, i.e., from which the starting trader could directly import offers of the correct abstract type. From these target traders, a continuation edge can be drawn to all those traders to which the import could be forwarded. Continuation of this process would result to a graph characterising a single trader with imports of a single service type. The graph would be different started from another trader, or based on another service type.

The trading graphs are based on the knowledge the traders possess about each other. The edges have no technical meaning as reserved communication channels. Channels can be established and maintained depending on traffic.

5.3.3 Distributed traders

In this section, we map the trading service components to the ODP infrastructure (recall Chapter 3). For the construction of a trading service, we use some storage (offer storage, link storage, offer cache), a trader object, and the trader interfaces. For the communication within a single trader community or between federated traders we use channels. The engineering language concepts used include capsule, cluster, and node. The study is summarised in Figure 5.5.

The trader object is a composition of capsules for offer storage and link storage. Each offer storage capsule must also implement all of the offer space modification operations. Furthermore, as offer cache is created by the same operations, also the offer cache must be part of the same capsule. Also the link storage must be implemented by a single capsule that contains link management operations. A single capsule can support both kinds of storage.

Requirement of a capsule to contain all trading functionality does not restrict the logical (computational) trader object to a single capsule. The storages of a trader can be partitioned to a set of capsules as long as all these capsules also support the operations.

A trader object can contain a set of capsules within a node that is controlled by a single nucleus. This means that the trader is fully under the control of a single administration. If the capsules reside in separate computers, the single nucleus is also distributed to all these computers and thus it controls the consistency of the operating environment.

The trader object is accessible through a set of interfaces. From Definition 3.6 we recall that an interface implies the location of the access point and the access technique for all communication. Thus, the structure of the communication channel between the processing capsule and the external access point becomes specified. As the trader object consists of a set of equal capsules, also the trader's channels must be connected to the same set of capsules. There are two choices: the access point itself can be expressed as a set of capsules, or a single access point is represented externally but the channel structure forks separate paths to the capsules. The ODP reference model definition of object location ([92, clauses 8.9, 8.10, and 8.11]) allows multiple access points for a logical interface. In both cases, a partial channel must reach from the capsule supporting a trader partition to an external access point. This partial channel is composed of channel objects encapsulated by a cluster within the same capsule as the trader partition.

The channel structure contains security stubs that consult security services. The incoming requests must be checked for access rights. The trader interfaces may each have a different protection scheme that is also dependent on the general security policies of the system. In addition to this general security scheme, the trader operations use separate policies for testing the request further before accepting it. However, these tests are part of the operation logic and implemented within the operations themselves.

This study shows the partitioning of a logical trader to physical processing objects within a node. A node may be a complex, distributed system if it only has a single operating system to control it. This partitioning to physical processing entities is necessary because of resource management reasons within the supporting node. These reasons are separate of those reasons leading to trader federations. Still, confusion arises in designs that suggest a single model to describe simultaneously both technical partitioning and semantical isolation. However, the algorithms applied to both cases are indeed similar, as shown in the following section.

5.3.4 Distribution and federation of import

Import in a federated or distributed constellation does not differ much from the single trader case. The import is delegated from one trader object to another using the rules for importer and trader roles. We study first, how the importer can guide the search within a group of traders. Then, we consider how import requests and import responses are manipulated when traders pass them between each other. We summarise the extensions to the trading behaviour by giving an updated version of the import operation algorithm.

Guiding the propagation of import in the trading graph

When an importer initiates an import, the import may be propagated to the full trading graph or part of the graph. The order in which the graph is traversed depends on two aspects. First, the trader can initiate delegated imports in parallel to the searches it initiates within its private

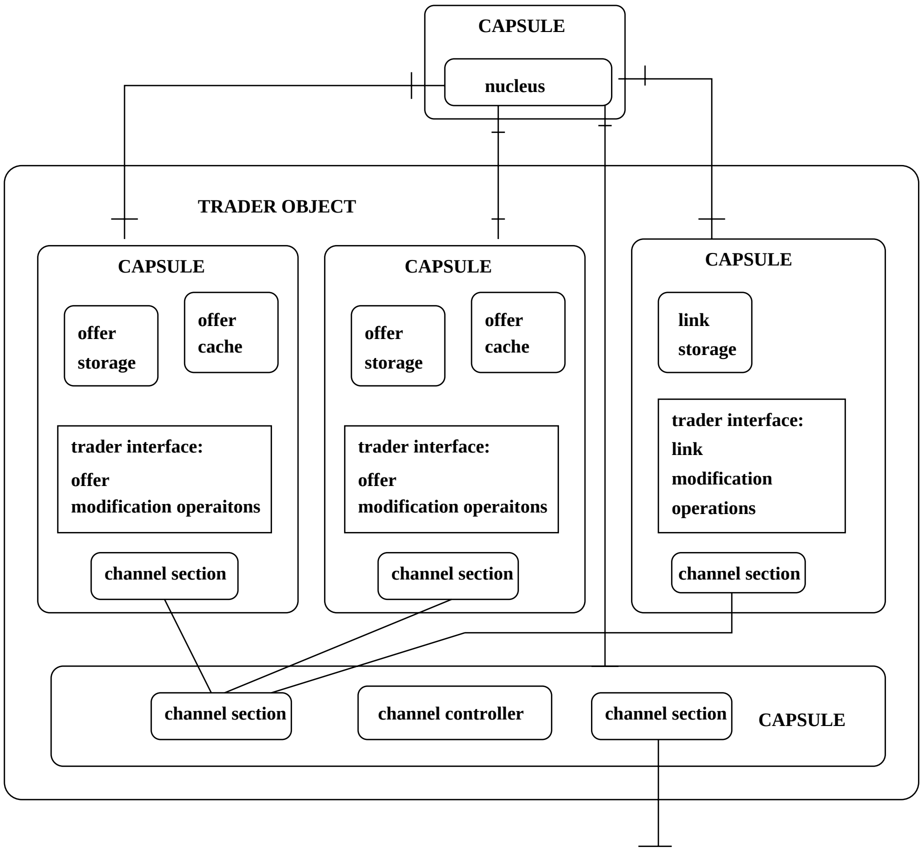


Figure 5.5: An example mapping of trader object components to an ODP node.

repository, or the trader can first search its private repository and only after that initiate delegated imports. Second, when the delegation phase is started, the initial trader can issue subsequent imports either as asynchronous, parallel requests, or as a sequence of synchronous requests.

The propagation of the import along the trading graph is restricted primarily by the importer constraints on the resources to be spent on the search. Therefore, the constraints must be passed along with the import request through the graph. In addition, each trader that is passed through, must predict or measure the resources it spends for the search, and must also update the resource constraints correspondingly.

The number of hops, that can be reasonably taken for resolving an import request, is naturally restricted by the application area. Each hop increases the resulting response time remarkably. Figure 5.6 illustrates the situation with assumptions of trader processing times of 4 ms (fast) and 100 ms (slow), and round-trip network delays of 50 ms for LAN and 250 ms for WAN. In order to decide reasonable number of hops for different situations, we consider these linears in context of response times of commonly used applications. Typical response time acceptable to a human user is within the range of 1-10 seconds. Response times for name services are within the range of 10-

1500 ms [205]. Some telecommunication applications may produce 2800 requests per second, so we gather that the response times should be close to 4 microseconds [185]. Thus, in cases where the trading service is used for browsing only, the depth of the trading graph is practically not limited. However, for the convenience of the user, some progress reports should be available at all times. In cases where trading is embedded to other services, for example replacing traditional name services, the trading graph depth is limited practically to three hops.

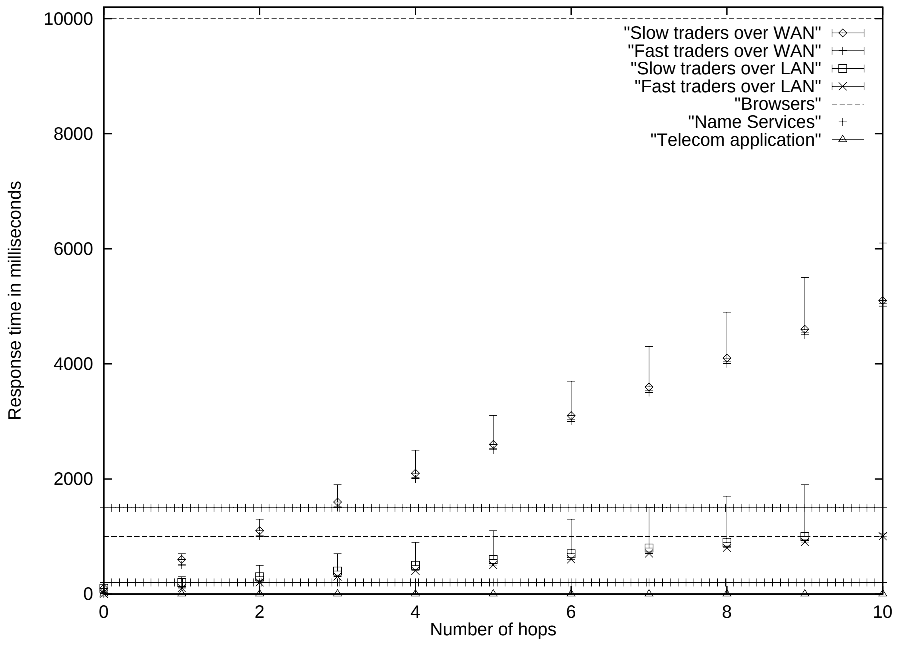


Figure 5.6: Analysis of the number of hops in the trading graph acceptable for different type of applications.

The propagation of the import can also be restricted by explicitly naming the traders in which the import should be processed. This possibility is actually a source based addressing scheme and therefore requires good knowledge of the topology of the trader federation. A special case of this source routing is found in some traders: a special 'resolve' operation for starting an import from a named, remote trader.

The propagation can be restricted also by restricting the number of edges traversed in the graph. This can be done either by counting all edges, or just counting subsequent edges and treating parallel searches as one unit. A special case of hop counts can be found in some traders that use flags for 'allowed', 'denied', or 'forced' delegation of an import (Section 8.2).

Delegating imports

In a trader federation, or even, in a distributed trader constellation, an import is resolved by a chain of trader objects. In the chain, each trader object acts as an importer towards the next trader object. The subsequent imports can be initiated either in parallel or serially starting a new import only after receiving offers from the previous trader object.

For the subsequent import, each trader must create an import policy that fulfils the import policies of the original importer and the trader object itself. This is necessary, because the import resolution process may visit several sovereign domains, and each trader object takes the responsibility of, for example, the import expenses.

The import resolution chain may fork at any trader. In such case, the import policy of that trader may be different for each outgoing chain of traders. Therefore, the import criteria are associated with each outgoing link, instead of the trader itself.

The import policy limits the resources spent for the search. One of the problems here is how the resources can be split among the subsequent imports. Some solutions can be found in traders introduced in Chapter 8.

One of the resource limits is the time allowed for an import operation to return a set of offers. The suitability of the selected time limit is of high importance. If the times are not in coherence with each other an importing trader can proceed in providing offers without waiting results of the subsequent import requests. In this way a lot of processing power and network bandwidth can be wasted. The time limit cannot therefore be the only criteria for terminating an import. We discuss various levels of asynchronous or synchronous, and QoS contracted interrogations in Chapter 9.

Joining the import responses

In a federated (or distributed) import, the import responses are passed towards the original importer through the chain of trader objects involved trough the import delegation. In each forking point of the chain, the trader object collects import responses from several other traders. However, the trader cannot propagate all the offers through the chain. Instead, it must apply those import policy rules that specify response cardinality.

The response cardinality and ordering rules must be applied to the set of offers selected either by the trader itself or the traders it imported from. As each importing trader can itself restrict the cardinality by import policy, they can also reserve enough resources for managing the flow of import responses. However, the result of this merging operation is rather unpredictable. All offer sets are selected in independent processes with an arbitrary terminating rule. In addition, while merging the resulting sets, only some offers are picked.

Extensions to the import algorithm

The processing of an import appears to have similar structure in both distributed traders and in federated case. The difference is actually shown in the federation establishment process and the complexity of communication during import. In distributed traders, the constellation of trader capsules is static and established at the installation time of the trader. The communication channels between the trader capsules are also continuously present. In trader federations, the federation establishment is invoked by import requests or by administrative operations on the running trader object. The communication channels between trader objects are created and destroyed at run-time, depending on the traffic between traders.

When a federation is frequently exploited, the involved traders can optimise the imports by techniques suitable for distributed traders. However, even in such federations, the possibility of independent changes for example in type systems may cause termination of the cooperation. Therefore, the essential feature of federated traders is the negotiation capability.

As a summary of the above discussions about federated and distributed traders we give in Figure 5.7 an extended algorithm for processing an import. The algorithm allows high parallelism, but it does not yet offer an answer to the problem of how to control the communication between two traders and how to restrict the amount of unnecessary processing and data transmission.

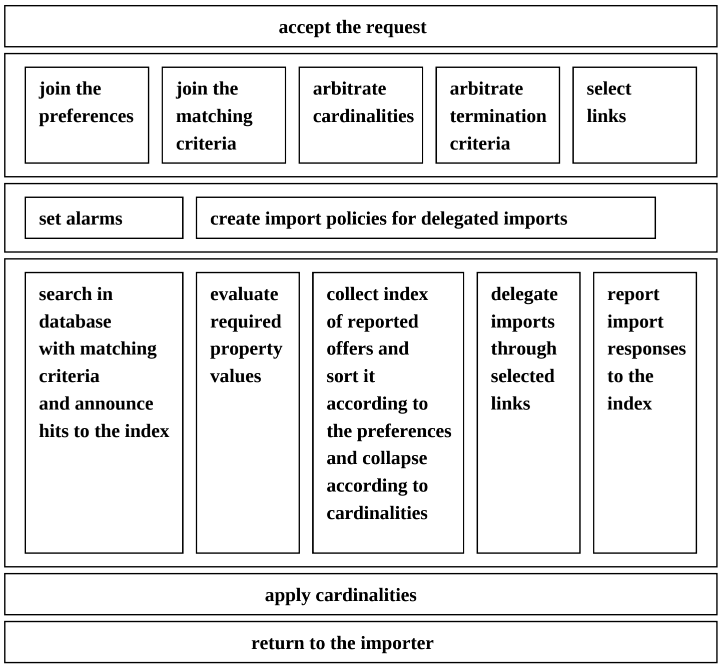

accept the request
join the preferences
join the matching criteria
arbitrate cardinalities
arbitrate termination criteria
select links
set alarms
create import policies for delegated imports
search in database with matching criteria and announce hits to the index
evaluate required property values
collect index of reported offers and sort it according to the preferences and collapse according to cardinalities
delegate imports through selected links
report import responses to the index
apply cardinalities
return to the importer

Figure 5.7: Search algorithm used within a cooperative trader.

The communication problem can be tackled by studying the possibilities of optimising the import performance, see Section 5.3.6.

5.3.5 Administration of trader cooperation

Administrative actions that can affect any trader federation can take effect only on one trader object. Administrative actions that affect potential trader federations include the creation and modification of links and trader's import policies stored per link.

A trader federation is not a static constellation of traders. Instead, each import can lead to a different trading graph. In cases, where a group of traders communicates frequently, better performance is reached if a permanent communication channel is maintained between them. We will see later in Chapter 9 that the federated system binding mechanism supports such functionality. Therefore, it is not a question of administrative actions at all.

The establishment of a federation between traders requires that some basic assumptions about the communication match: service types understood, properties, information exchange protocols, and security systems involved. One of the characteristics for federated systems is the support of constant and independent evolution of the systems. Therefore, any of these assumptions may change. This may either increase the number of potential federations between traders, or decrease it.

Traders can store information about potential federations into links. If the type system used by a neighbouring trader is changed, all links referring to that trader should be reconsidered and modified. Even if the type information is stored to type repositories, the link information may still include type and interceptor names within link criteria.

Basically, the same administration rules are suitable for distributed traders as well. However, the constellation tends to be more static, and the trader capsule properties tend to be similar to each other. Still, the service types used by each trader capsule may be different because of partitioning of offers can be based on their type.

5.3.6 Optimising the trader cooperation

Essential for trading applications is the response time of imports. In order to optimise import performance, the interesting offers should be made available as close to the importer as possible. However, the optimisation should not overload the communication network.

Properties of the basic import functionality

The basic import behaviour already gathers some cached offers to the traders that reside on the route from the importer to the trader providing the offer. A simple optimisation solution is to store these temporary copies for further use. The solution creates no extra network load, but a deletion mechanism for outdated offer copies is needed. The time-to-live offer property can be exploited for this purpose.

The simple caching solution also automatically creates and maintains a set of actively used, cached offers. Users typically use a rather small set of applications that are relevant to their work, and each application may induce a set of imports, repeating their matching criteria and import policies.

Caching allows independent users, importing from the same trader, to reuse cached offers. The effect experienced by the users depends on the import policies used in the import. If search time is not expressed in import policy, the users experience a shortened response time. If response cardinality limits the import, the result is produced with fewer resources. Finally, if no resource limits are given in addition to search time, the response is a result of considering more offers or importing from a greater number of traders.

In a general case, any trader can occasionally import some offers from another trader. For each service type there tends to be a separate network of frequently used traders. Between traders, that frequently import from each other, a communication channel can be established and maintained permanently or based on communication frequency thresholds.

The response times for imports are affected by hit ratios of offer caches in individual traders. Figure 5.8 illustrates how the increasing hit ratio improves the performance of imports. We assume that the imported remote offers come from the distance of one hop only.

There is not enough practical experience on trading applications for well-founded claims on the hit ratios. Some commercial WWW based applications are in use, but the trading interface is mainly used as an object database access interface. As the offer spaces are centralised by nature, no characteristic features of trading are actually shown. Therefore, we would need more concrete experience in world-wide environments.

We can however relate the caches within traders to two other types of caches with better known characteristics: caches for virtual memory and for WWW servers. The characteristic load of virtual memory contains memory references generated by some program code, with locality properties. A typical virtual memory hit ratio is above 95 % [220]. The characteristic load of WWW servers contains references to Web pages generated by a group of users with no strong locality properties. However, the theme of the pages visited causes some grouping effect. A typical Web-cache hit ratio is reported to be in the range of 24 - 45 % [49, 250]. The characteristic load of a trader would be generated by the personel of an organisation using applications related to the organisation goals. The locality properties of the load are not as high as in the case of virtual memory, but higher than with WWW. Therefore, we assume, that a trader cache would have a hit ratio in the range of 60 - 80 %. According to Figure 5.8, we can assume an average response time in the range of 10 – 70 ms, which appears reasonable for most applications.

Performance measures on imports also depend on the scope of the search: whether the search is assumed to produce any single suitable result or to do a more thorough search. Cached offers can be held relatively long times in order to improve the cache hit ratio. On the other hand, too much memory should not be consumed.

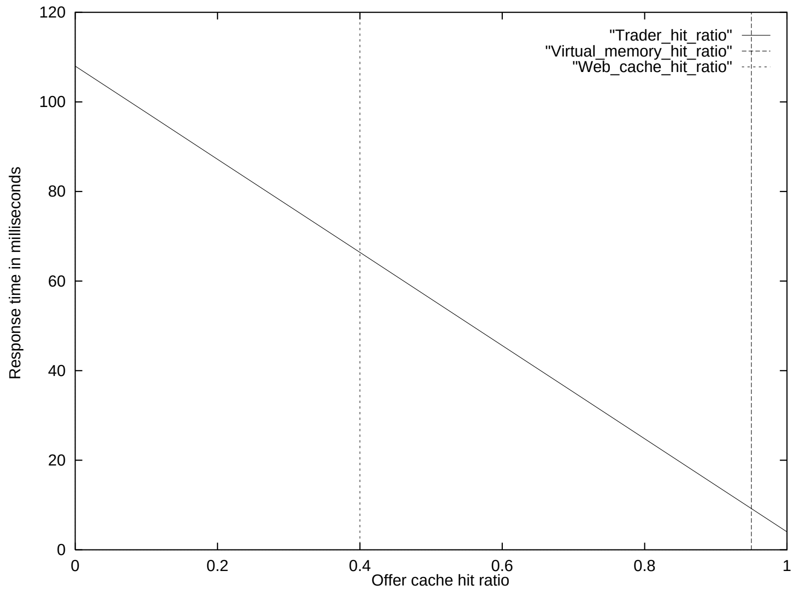


Figure 5.8: Analysis of the hit ratio in the offer cache to the expected response time.

Active alternative: working set analogy

The performance of imports can potentially be improved by an aggressive working set update. Such an update must be based on predictions on future imports. We assume that imports created by a group of users follow a pattern characteristic to the applications used. Therefore, we can assume that the offers requested also follow a characteristic pattern, in analogy to the working sets [43] appearing in virtual memory systems. We can characterise a working set as a group of memory pages typically used within a limited time period in a computer system. The same phenomenon is present here, although in a wider-spread environment.

The working set analogy can be employed by creating typical sets of offers that are communicated whenever one of the offers matches an import. Successful use of the analogy requires, that all traders would have a similar import profile. However, this is seldom true. In fact, there would appear to be a set of working sets, one for each importing trader. Each offer would be flagged by the working set it belongs to and protocols would be required for duplicating the working set offers efficiently, without too heavy network traffic. Especially any modifications should be propagated.

The working set analogy is problematic, because it involves more than one cache for each trader to update. The potential number of involved traders is large, and it would in general case be impractical to have all importers listed. In a limited case, like in distributed trader constellations, such protocols would be efficient. Also, in federated trading cases, federations with long duration can be supported, because the number of traders involved would be controllable.

The aggressive update propagation may produce too heavy traffic, and thus we analyse the number of messages required. We assume a single trader providing updates for its client traders. In addition to the number of clients, we consider also the number of offers involved, and the frequency of update propagation. Figure 5.9 illustrates the increase of messaging in the system, in cases where the trader repository includes 100, 1000, and 10000 offers, and the updates are performed at intervals of 10 seconds, 1 minute and 1 hour. We should expect the trading service to create at most equal network load with DNS services. In September 1991, the DNS contribution to NSFnet backbone was about 270 messages per second [39]. It would only take two aggressively updating traders with ten clients to reach the same level of load.

Sharing and replicating offers

In some cases, the trader cooperation may be so frequent, that it is reasonable to replicate a set of offers in several traders. For instance, when a computing centre provides some basic service for all departments in a company, the computing centre offers might be replicated to all departmental traders.

The technique for maintaining the replicated offers depends on the trader implementation environment. In a distributed system environment, for example a virtual shared memory service can be used. Thus the trading mechanism itself does not need to be involved in the replication functionality. In a federated environment such service does not appear, and therefore the offer sets should themselves include corresponding facilities, in the manner that is shortly described.

Active propagation of all changed offers would in general create too much traffic to the network – the number of offers was considered large enough to justify development of an additional information retrieval mechanism. However, it is natural for the importing trader to create predictions on the required set of offers, and for instance, to activate prefetches from remote traders. The

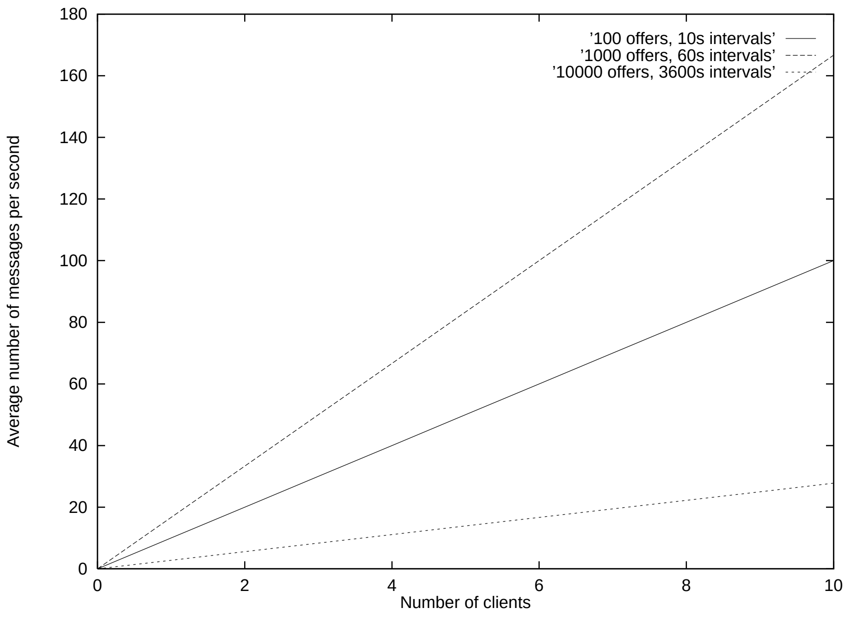


Figure 5.9: Analysis of the number of messages required for aggressive update propagation.

active partner is the one that supports the clients and thus has knowledge about the behaviour patterns.

Active propagation of changed offers is reasonable in cases where the imports are much more frequent than the changes. For example, in a telephone switch, the route information for a phone is static in relation to the frequency of phone calls.

The techniques of pulling and pushing offers to another trader can be combined. The combined technique requires that each offer includes, as offer properties, some additional management information. In offers that are locally created by export, we must add a modification action policy, to indicate what actions should be taken when the offer is modified. The choices include (i) 'none' to indicate that the simple import method for propagating the offer is sufficient, and (ii) a protocol name with a list of target traders to indicate with which protocol and to which traders a further update is needed. In cached offers, that are either imported or actively copied from other traders, a storage policy is required to indicate when the offer can be destroyed or how the offer can be updated. The information required for requesting active update is similar to that of proxy offer. The resulting offer structure is summarised in Figure 5.10.

Because the traders may represent sovereign administrations, neither the push nor pull side of the technique can be omitted. Each side can independently decide how long the offers are available, considered relevant, and considered to contain up-to-date information. For example, when two organisations buy services, the intervals for checking prizing information may be different. The selling organisation bills according to its own prizing information, regardless whether the buyer based the decision on old or new advertisements.

```
* an offer identifier,
* a service type identifier,
* exporter policy, including access rules or interface for
  executing access rules, and time-to-live values,
* modification policy, including active update propagation information,
* a flag  differentiating between normal offers,
  proxy offers and cached offers,
* in case of proxy offer:
       - an interface reference,
       - instructions for creating a request to be sent to the interface,
* in case of normal offer a set of items containing:
       - a property name,
       - either a property value, or
         an interface reference and instructions for invocation,
* in case of cached offer:
       - items of a normal offer,
       - items of a proxy offer referencing to another trader.
```

Figure 5.10: General offer structure for cooperating traders.

Figure 5.11 illustrates the effect of splitting the offer space to several traders. The replicated offer space can be exploited by partitioning the clients so that the number of requests to the trader is equally distributed. However, at the same time, the change propagation requires more messages. In a federated case, the offer space is partitioned, but the caching effect allows some of the offers to be picked up from traders that have earlier imported the offer. Each client is assigned to a certain trader similarly to the scenario with replicated traders. However, the propagation of the request is only hindered by hits in the offer cache, if the request is about a foreign offer. We assume that each trader receives requests on the same frequency, denoted by 1. In the federated case, each trader is able to supply the requested offer by probability of 0.5 or 0.75. The offer cache hit ratio is 0.1 or 0.2

However, although the number of queries seems not to scale, the situation is not so bad after all. The trading graphs are established separately for each service type, and thus the load can be partitioned based on the service type. If the traders support only a few service types, the number of requests is small in comparison of the total number of requests.

Long-lived imports

The frequency of imports can be decreased by allowing the importers to keep an imported offer for a defined time with a guarantee that any changes to the offer are notified. For instance, if the interface referred by an offer is deleted, the importer receives a notification of the withdrawal of the offer. This kind of behaviour is called long-lived import [87, 166].

The technique requires that a list of notification addresses is available for each offer. Depending on the computing platform, different techniques can be used. If the computing platform supports objects and remote object references, the importer can obtain references to offer objects by importing. When the offer object within the trader is modified, the modifications are also visible to the importer keeping the reference. When the offer object is removed from the trader, the reference held by the importer becomes invalid. However, the importer may still be unaware of the changes and try to misuse either the reference or the offer contents. If the computer platform does not support objects or object references, the import response lists a set of property names and values. The offer stored within the trader must include a list of importers in order to be able to send change messages.

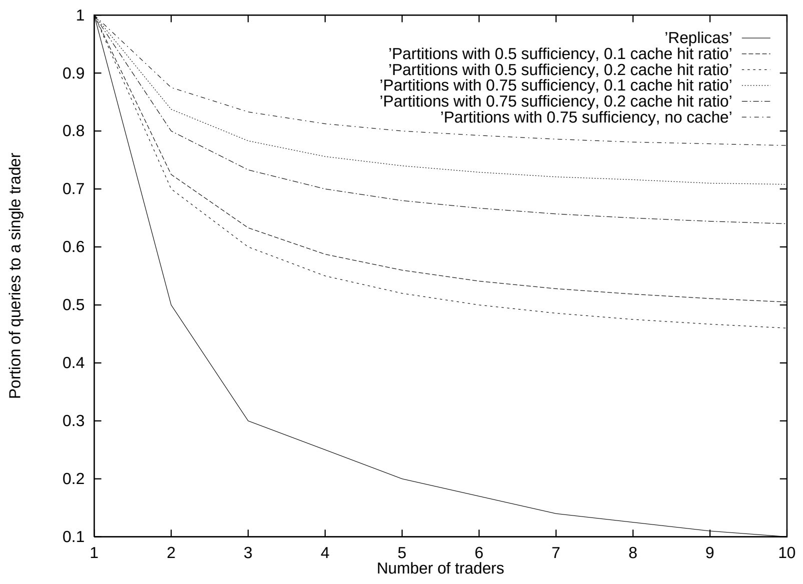


Figure 5.11: Analysis of the distribution of requests to traders.

Long-lived imports do not guarantee anything of the importer behaviour. The importer may keep using the information from an outdated offer, although the trader has invalidated it. Still, the trader is forced to use resources for maintaining address lists and sending notifications. The mechanism also trusts in a shared time service to guarantee that the importer and the trader know when the import deceases.

Cooperation contracts

We have earlier discussed links between traders as single-sided references from an importing trader to another. Instead of links, cooperation contracts can be used for similar purpose.

The cooperation contracts (federation contracts [8], also implemented in [42]) differ from links by guaranteeing that the actual communication between the traders succeeds. The guarantee covers matching service types, access rights, and communication protocols. However, there are no availability guarantees involved.

The cooperation contract is established by a mutual negotiation protocol. The initiating trader creates a catalogue of its functional capabilities and sends it to another trader. This other trader can response to appropriate requests by sending matching entries of its own catalogue and the received catalogue. When both traders have a copy of the negotiated catalogue of capabilities, a cooperation contract becomes established and is stored to both traders.

In addition to import actions, the cooperation contract can cover export, modification, and withdraw actions as well.

Chapter 6

Support for trading mechanism

The users of trading services assume that service types and property specifications are similarly understood by importers, exporters and traders. The users also assume that federation or distribution of the trading service itself is masked, i.e., the users expect a federation (and distribution) transparent trading service. The traders can meet the transparency requirements either by themselves performing necessary mappings and transformations or by exploiting federation transparent services.

In this chapter, we study services for maintaining type information, name information, and other supporting services required by trader objects. Alternative ways of constructing a trading service, by exploiting the services described in this chapter, will be shown in Chapter 7.

6.1 Naming

The object name management functions have a key role in the open architectures. Names are used for addressing all kinds of objects, for example interfaces, behaviours, type descriptions, and type relationships. Traders use name services for various reasons: for identifying offers, links, importer and exporter objects, and for example for type names, property names, protocol names, and policy names. Most of these names are used only within the trading domain and never passed to other domains. However, type names and property names are passed to other traders during federated imports. Section 6.2 will show how these special cases are supported.

To support other infrastructure functions, the object name management must offer operations for assigning and reassigning names, deleting names, communicating names across a transport network, making synonyms to locally and remotely managed names, and comparing names disregarding their location.

6.1.1 Naming systems

A well-established set of distributed naming systems is currently in use. The interesting areas with different kind of names include

- addresses for computers in various data communication protocols,
- identities for resources controlled by various operating systems and resource managers,
- identities for objects and interfaces controlled by various distributed system services,

- names for service types and properties including QoS properties, and
- names for service behaviour.

This list refers to 'addresses', 'identities' and 'names'. All these variants refer to a unique concept of some kind, but the properties of these variants differ.

An address is a location description for a communication partner, expressed in terms suitable for a communication protocol. Each communication protocol, for example ISO/TP or TCP/IP, have different conventions for structuring addresses. An address is a more convenient way of expressing the target of a message than a route description that defines which signalling resources should be used for reaching the target.

An identity is a globally unique reference to a resource, interface, object, or other entity – even a user. Usually, for communicating with the identified entity or for accessing the identified entity, the identity is mapped to an address. As the communication protocols for separate naming systems, the identity management systems create an additional unifying layer on top of them. The identities of addressable entities are collected to a single name space with a single naming convention. This allows easy use of identities and still allows access to the heterogeneous addressing. An example identity system is X.500 Directory service [105].

A name also allows separation of name spaces, however, not based on differences in underlying technique, but based on separate administration. Separate administration means that the name spaces can evolve independently: they can have different conventions, they may use the same names for different entities freely, they may use different names for the same entities freely, etc. Actually, having discrepancies in naming the same entities is not a real threat in this case: separate administration also means that the controlled entity sets are separate. However, names are also used for concepts like behaviour patterns and service types, that are logically shared, but not concrete resources. Still, each open system must have concrete expressions, names, for the concepts.

6.1.2 Scope of naming systems

In all naming systems, the names are required to be 'globally unique', which must be interpreted as 'unique within the scope of the naming system'. For different naming systems, the natural scope differs. In each computing node, a set of naming systems is present, each having an independent scope. For example, in Figure 6.1 the computing node X uses separate naming systems for behaviour patterns, interface identities, and TCP/IP addresses. The behaviour patterns are controlled in node X, and mapped to behaviour patterns in other nodes when necessary. The node X is part of a distributed computing environment that forces interface identities to be under a centralised control in node Z. The control of TCP/IP addresses is world-wide.

The importance of integrating the name systems of individual computing systems varies depending on the kind of name system used. In addressing systems, a set of world-wide systems is necessary. The systems must be joint together to a federation through a gateway system that also routes the communication traffic to the network with different protocols. In identification systems, the need is similar. However, the scoping rule is not any more just technical but also organisational, administrative. Still, integration requires that the name spaces are joint. There is no theoretical reason for this, instead a practical reason: current distributed platforms already include such naming services. In systems, where plainly ideal concepts are named, other mechanisms can be used. The naming domains are rather large and extra overhead is created only when a domain boundary is crossed. There has not yet been a practical integration step for these

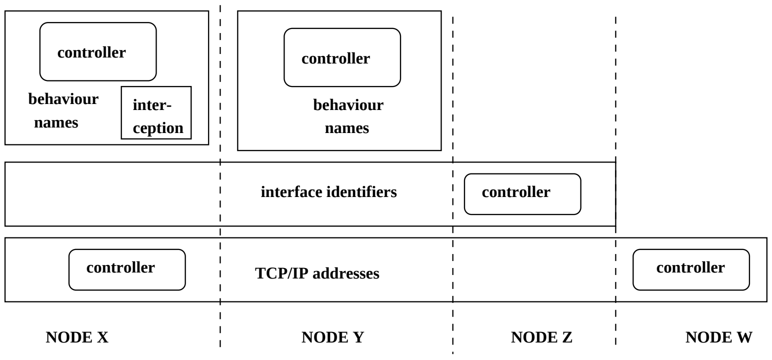


Figure 6.1: Scopes for naming systems.

names, so the naming systems are very different. Integrating existing systems to a single name space would be impractical and would lead to non-evolving system design.

In the following, we discuss naming concepts for distributed and federated naming. A federated naming service allows

- multiple roots for the naming hierarchy,
- multiple naming conventions used simultaneously,
- an open set of protocols for communicating between name servers, and
- resolution of potential name overlaps at communication time.

The difference between a distributed and a federated name system can be clarified by an example, the distributed name service DNS used in Internet. Although DNS covers, in many respects, multiple administrative domains, we still claim it to be an example of distributed naming. The reasons for this classification include the following: First, the naming domains share a single naming convention. The naming convention ensures that each domain produces only such names that no overlap is possible. Second, communication between name servers is performed with a closed set of protocols. Third, the resolution is based on a single, world-widely accepted root for the naming hierarchy. All names are relative to this root.

6.1.3 Naming concepts

Context relative naming is a commonly accepted design for naming systems in distributed environments [177, 243]. Especially, federated naming framework is included as a component to the ODP reference model [98].

Naming context is a set of relations that bind together names and entities. Formally, we give the following definition [110]:

Definition 6.1 *Context is a 3-tuple (T, R, E),* **where**

- *T is a set of possible terms, fulfilling the requirements of the naming convention in use by a naming authority;*
- *E is a set of nameable entities under the control of the name authority; and*
- *R is the set of relations binding together terms and entities.*

In this definition, we deliberately avoid using the word 'name' in order to avoid the usual confusion between potential names that are still unbound and the already bound names referring to an entity. To match Definition 6.1 we define also 'name':

Definition 6.2 *Name is a relation (t,e) that belongs to the set R so that t belongs to T and e belongs to E.*

This definition slightly differs from the usual meaning. We require that the whole relation (t,e) is considered as a name, instead of t alone.

In a federated environment, multiple naming systems must cooperate. Therefore, naming conventions and authorities over nameable entities must be considered. In a distributed environment, there can be a single authority for naming conventions, and in that case, the relation (t,e) can be in practice replaced by t alone. However, in federated environment, such simplification causes semantical difficulties.

Contexts can be organised into a global hierarchy that has one or more roots for starting globally unique names. Contexts can also be organised to a free-structured network where each context can use itself as a starting point for the globally unique names it is able to use. In this design, the clients of two contexts cannot directly adopt names from each other, as they are able to do with the hierarchical, single-rooted model. The two models are illustrated in Figure 6.2.

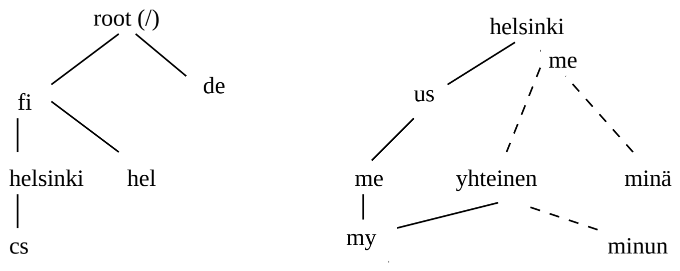


Figure 6.2: Single-rooted naming graph and a free-structured naming graph.

User name context can be separated from the infrastructure name contexts.

Naming domain is controlled by a single naming authority, a name server. A naming context can be supported by multiple naming domains. In that case, the term space and the entity space of the context must be partitioned so that each name server can operate independently from the others.

Formally, we can define [110]:

Definition 6.3 *Domain is a 3-tuple (T, Dom_D(R), E),* **where**

- *T is a set of possible terms, fulfilling the requirements of the naming convention in the naming context;*
- *E is a set of nameable entities under the control of the naming context;*
- *R is the set of names in the context; and*
- *Dom_D(R) is the subset of R that is maintained and controlled by the naming authority at the domain D.*

Naming federation is a group of naming domains that are able to resolve names in cooperation. Federation occurs at name resolution time. The name servers do not enter any fixed constellation, or join their name spaces. Instead, the name resolution process propagates through the name graph.

6.1.4 Federated name service

The requirements for a federable name service differ slightly from the requirements for a distributed name service, because holding a term only cannot be assumed to carry enough information about the name binding in a federated environment. Therefore, the distinction between bound names and unbound terms is essential. In addition, it is important that multiple naming systems do not directly bind their terms to entities, but only a single naming system has direct control over the entity itself. In ANSA architecture, the concept of administrative naming system is used [243]. Other naming systems bind their terms to the administrative names.

The federation model affects both the structure of name objects and the naming related activities.

Name objects

In a federated name service, the names are objects that consist of

- name-server interface reference,
- term to be interpreted by that name server,
- name class (interface name, behaviour name, etc. to help the name server to choose a syntax for interpretation),
- immutability policy,
- relationships to other names (such as 'synonym'), and
- cache, time-to-live for cache, and re-evaluation instructions.

In comparison to DNS name objects, the above specification adds to a single name the properties that are defined for the group of names maintained by a domain. In DNS, all names within a name domain are transferred to secondary name servers at the same time. The group of names has a time-to-live property, they can be accessed through a given IP address, and they all use the same syntactic format. This information is either explicated for each domain, or is implicitly known because of the DNS protocol specification. This design for name objects follows the same rules as we used to optimise the federated trader imports.

We need to have a small set of immutable names that are stored to all computing systems for facilitating name management. A set of name servers must be globally named, also the name management operations, and the attribute names and allowed values for them. In addition, the name for operation invocation behaviour must be globally understood. The rest of the names can be constructed based on the initial set. Such initiating names must be standardised in order to facilitate basic infrastructure services in federated systems.

Activities

Name binding, and name unbinding activities change the contents of a naming context. The activities are performed in the naming domain that controls the name objects involved.

Name resolution process searches for a given term t in a local naming context and finds out the names (t,e). The result is used for referencing the entity e in a technology dependent way. If the initial name to be resolved has some structure, each component in the structure is resolved separately in a sequence where the found entity e is used to resolve the remainder of the name structure.

Name communication transmits names (t,e) from sender to receiver through the infrastructure. During communication, the infrastructure may create an additional name (t2, e) or (t2, (t,e)), at the target naming domain.

Name comparison resolves two names up to a point where it is possible to determine, whether they denote the same entity or not. Even if the resolution is successful, the comparison result may be undetermined, for example because of access restrictions to the names or entities. The comparison operation may thus terminate with outcome of 'same', 'different', 'comparison undetermined', or 'comparison operation failed technically'.

6.2 Type repository services

The type repository service mediates type information that is available at run-time in federable systems. Traders use type repository services for checking conformance between object interfaces or for transforming other descriptive information about objects to locally acceptable representation formats. Traders could include the management of type information themselves, but separating type management aspects to an independent service provides flexibility in configuration and supports evolution of the systems.

6.2.1 Federation of type systems

In a federated system, each participating federable system may use a separate type system. Each type system expresses a set of logical, abstract type concepts and corresponding technical solutions that are dependent on the computing platform in that system. For interoperation between the technical solutions, a logical equality of abstract types is necessary. At technical level, interoperation requires a relationship from a technical type representation in one type space to another technical representation in another type space. This relationship is associated with the technical conversion tools (i.e., interceptors) needed for the replacement of one technical type with another. Concrete type mapping problems between different distributed platforms have been studied in [23, 83]. The type repository function can also be used for mapping between different type abstraction layers, for instance, between the end-user view to service types and the system designer view [194].

The federated system that is able to match different representations for a shared abstract type, and to find suitable templates for the concrete type is illustrated in Figure 6.3. The topmost layer represents the idea of an abstract type. It is a logical entity with no single representation. The second layer represents a group of relevant representations for the abstract type. Each of these representations is possibly used in a different federable system. The mapping between the representations is done by introducing their relationship to the 'ideal type'. The lowest layer represents the implementation layer, the templates. For each representation of the abstract type there is a set of templates. At this layer also interceptors are identified for implementing the necessary conversions between the objects fulfilling similar types. The interceptors bridge between the types and the templates. An example: the ideal type is billing, the representation of the type denotes an interface with operations '`SendBill(addressee, amount)`' and '`ReceiveMoney(from, amount)`', and finally, a template supports an operation like '`Bill(ToWhom, HowMuch, Currency)`'. (The three layers of Figure 6.3 can be compared with the layers of abstract schemata in federated databases, as described in Section 2.3.4.)

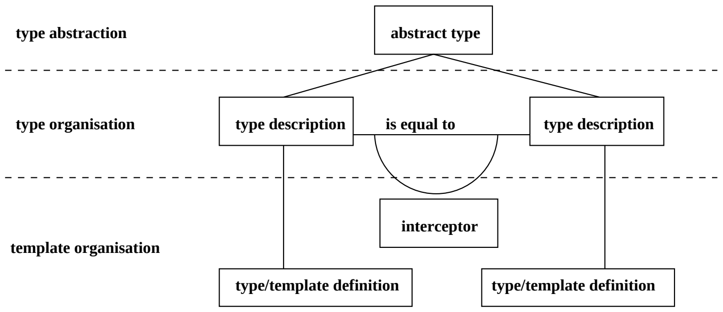


Figure 6.3: Three layers of type abstractions.

The conceptual requirement for federation is that the type systems include the same abstract types [112]. Potential for federations is increased by standardising type systems. However, also nonstandard types can be included. This supports system evolution and allows, for example, interworking between the products of the same vendor although more generic cooperation is not possible.

The practical requirement for a type federation is that the technical solutions at each involved system can cooperate at least through an interceptor.

6.2.2 Type related activities

The type repository service offers operations for

- publishing realisations of abstract types,
- checking whether two type realisations are conformant and interchangeable,
- retrieving subtypes or supertypes of a type realisation,
- retrieving templates for a given abstract type,
- translating one type realisation to another,
- retrieving names for abstract types and type realisations in other type domains.

These operations have an essential role in federated systems. In the federated binding process, the type repositories map together type and template descriptions across type system boundaries. In other applications, traders use type repositories for an external source of offer structuring and transformation rules.

In the following, we study the management of type definitions and relationships.

6.2.3 Type definitions

A type system consists of type descriptions representing for example interface types and related templates. Each type description is structured depending on the rules for the abstract type it represents. An abstract type concept hierarchy was already introduced in Figure 2.3. In the draft type repository function [97], a set of abstract types with the same role in the system is called as target concepts. For example, service types, interface types, and behaviour types are target concepts.

A type definition represents an instance of an abstract type. A type definition includes a collection of type descriptions, possibly in different languages [97]. A type description is a concrete expression for an abstract type. A type definition can also contain maintenance information for the type repository system to be able to handle the definition and description objects.

Relationships can be defined between type descriptions based on their representing of the same abstract type. The relationship can be equality, or replaceablity via interception. If interception is required, the interceptor interface references are stored to the properties of the relationship [112].

6.2.4 Service type definitions

For trading, service types are especially interesting. A service type definition is uniquely named within the type domain. Its realisations are defined as a set of interface type definitions with associated attributes. The attributes reflect the contract schema specific for the described service type. Suitable attributes include quality of service issues and communication protocols.

Interface types can be expressed in various languages. For operational interfaces, a commonly used description language is the CORBA IDL. The IDL supports expression of operational interfaces only, and it does not include any aspects of behaviour at interfaces. For other attributes, there seems not to exist a commonly used description language. Instead, for example for available protocols, name enumerations are used. A study of type expressions in different system environments is presented by [83]. Most type definitions actually represent templates.

In principle, it would be possible to deduce interface conformance based on the stored interface type descriptions [97]. However, the descriptions may vary in both their language and their presentation style, and therefore the deduction process is not reasonable during the binding process. In a general case, the process is probably intractable [187]. Still, deductions can be done independently and the deduction results can be stored to the type repository [111]. The deduced relationships are called ‘asserted relationships’ [97].

The interface types can have three kinds of basic relationships. First, interface types can be separate representations of technically same interface. Second, they can be separate representations of interfaces with a subtyping relationship. This means, that one interface can be replaced with another without other considerations: the requested operability is supported, but some extra functionality is present on the subtype interface. Third, the represented interfaces can support the same abstract service but differ technically so that interceptors are required.

Because the relationships between interface types may require interception information to be stored, we store type relationship objects to the type repository along with the type descriptions. A type relationship refers to two type descriptions and includes information for interceptor instantiation. In addition to service types and its component types, such as interface types, the type repository can store also data type names, behaviour type names, and protocol names. Different representation domains can appear also within a single system, in order to allow different type information users to use expressions suitable for their tasks.

6.2.5 Type definition manipulation

Modification of a type definition causes mismatches between the type system and the objects exploiting it. Therefore, a type definition is never instantly removed from the type system, but marked as deprecated. New offers or links cannot be created based on that type, but existing objects can still be interpreted based on the definition [97]. Various mechanisms can be exploited for finding out when the type is actually not used any more. Construction of such a mechanism is easy in case of a full cooperation contract protocol, because each link pointing to a type system also has information stored to the system pointed at. In other designs, such information is not available.

Deletion of types is eventually necessary, because the type repository is finite in size and the search times in the repository grow as the repository size grows. The type definitions can be assigned with a property indicating their intended permanence [97]. Types can be classified as immodifiable, modifiable and temporary. Immodifiable are actually never removed from the type system, and links using such type definition can be safely used. Modifiable types are managed by some kind of announcement protocol for changes or deletions, or the exploiters are expected to check the definition often enough for their own purposes. A type definition, that has been flagged to be temporary, discourages all external use.

6.2.6 Suggested realisation of service type repository

A type repository includes a set of type descriptors. A type descriptor consists of

- type repository interface reference,
- type name to be interpreted via that interface,
- classification for the abstract service type (service type, interface type, behaviour type, etc. to help the type repository server to choose a structure for interpretation),
- names of component types (such as signature and behaviour type names),
- immutability policy (such as 'immutable', 'mutable using protocol', 'temporary'),
- notification protocol for modified definitions,
- relationships to other types either within the same type domain or in other domains (such as 'equal', 'subtype', 'interceptable'),
- interceptor interface reference, and
- cache for type descriptors from other type repositories, time-to-live for cache, and re-evaluation instructions.

Other suggestions (like [83, 11] and [162]) differ from this in four aspects.

First, interceptors are usually not included to the responsibilities of type repositories. Instead, traders are assumed to instantiate the necessary interceptors [79]. This task can however be assigned to the binding process as well (see Chapter 9).

Second, it is usual that a single protocol is used for all exchange of type information, including the distribution of changes in type definitions. In this model, we suggest that each type carries a protocol name in case active propagation is required. This design repeats the design suggested for offers stored to traders, and name objects in federating name servers.

Third, type descriptions from other type domains are usually not cached within the type repository. There is no need for caching, because the type repositories are used separately. For example, a trader requests the target system types separately and caches the responses if optimisation is considered. However, we believe that all type management aspects should be embedded to the type repository service itself.

Fourth, an additional abstraction level of abstract service type is used here. This is reasonable, because various business area service types are under study in consortia like OMG. They are going to provide a framework of services for general use in business domains. Such a framework is on the correct abstraction level to be used as a grouping concept. A similar approach has been presented where artificial intelligence techniques are applied in the context of type repository [194, 61]. The work suggests that different user groups describe a type with different concepts. The type repository matches these descriptions to a single type name, which in turn can be interpreted to a computing system specific description.

6.3 Storage

Trading function, as well as naming and type repositories, require support of information storage. It is not necessary to use a distributed storage service, such as a distributed database. Instead, the distribution aspects are governed by function specific behaviour. Furthermore, in a federated environment, distribution of functionality at the level of storage service is not applicable. Still, if trading is exploited in a distributed system only, solutions like shared virtual memory can be used for improving the system performance.

The characteristics of trading function set special requirements for the storage system. For example, because the goal of trading is to provide some suitable offers fast, high consistency of the stored information is not a primary goal.

We set the following requirements for the storage system [130]:

- The access and update of information items (offers, properties) must be very fast.
- The storage must be able to handle virtual data. Virtual data values can be evaluated by demand at the time of access. The evaluation of virtual values becomes part of the search process within the storage, and it should allow reasonable amount of concurrency but still remain in given resource boundaries. The mechanism for accessing virtual data can also be used for predicting new values, based on the history of previous values.
- The access time and the cost of searches from the storage should be minimised by caching the virtual data values after evaluation. Yet, the caching mechanisms should be sensitive to the class of data and the cost of its evaluation.
- The storage should be able to give answers to queries even when some of the data is missing, or the evaluation of virtual data is unsuccessful (within given time limits). Incomplete results shall be handled separately within the actual services themselves (the trading function, the type repository function), because of the different semantics.
- There is a need to update the schemata of the stored information objects without disturbing the running searches. Changes are due to new service types and new cooperation policies that are constantly introduced within the global system.

In a federated environment, each subsystem has a private storage. The subsystems can independently select the storage technology. They can also independently decide about the schemata of the stored information objects. However, when cooperation with other subsystems is required, the object schemata must be negotiated by the system administrators. The requirement of autonomy does not necessarily lead to a distributed database problem: the system function standards will eventually describe, how agents at each subsystem should act as clients for each other.

6.4 Operation invocation

We can take two different approaches to the invocation of operations at the target trader interface.

In a distributed environment, the client can use an engineering level addressing mechanism for the trader and invoke the operation without any transparency or quality of service requirements. This can be done for instance on any straightforward RPC implementation that trusts in a single administration on both the client and server side. However, if we wish to exploit trading in a federated environment, also the operation invocation mechanism for trading operations must follow the federated design.

In a federated environment, a major problem is that the platforms may have different invocation techniques. The problem is solved in a generic way as described in Chapter 9. Although the generic model exploits trading as part of the mechanism, the traders can still use a simplified version. Instead of trading, they use naming when addressing each other. The trader names are found by passing trading service offers or by establishing links between traders, as described in Chapter 5. The federated binding mechanism creates a channel between the client-trader interface and the server-trader interface. The components within the trader are independently selected by the federating systems so that the channel sections meet at the border, at a communication port with some protocol agreed. For the binding requests, there is an initial network of systems that accept requests from each other using a predefined protocol.

6.5 Security and trading service availability

In this dissertation, we concentrate on the mechanisms that allow new cooperation relationships to be established automatically. However, the mechanism must be restricted in order to ensure security and supplemented in order to improve dependability of the trading service. Although the security aspects are not within the scope of this dissertation, this section includes some observations about dependability and trust of trading systems.

Combining secured services and trading

Traders can be used for isolating systems form each other, when they include both secured and public services. Public services can be announced through the trader, and meanwhile, only authenticated clients have knowledge about the secured services.

Problems arise if secured services are traded. In that case, all offers must be protected as carefully as the services themselves, i.e., the service offers may not be available for clients unauthorised to use the services. If several security groups use the same trader, also each offer must be further tested for access control before revealed to the importer. This may cause decreased performance. Therefore, it is recommendable that each separate security group uses an isolated trader. Federation is not recommended for secure services.

Threats for trading service

In addition to the traditional threats of message passing (like eavesdropping, replaying messages, changing messages) trading services have some special concerns.

Trading service availability can be compromised by flooding a trader with offers or computationally intensive import requests. The potential can be decreased by including import policies to the trader. The policies can effectively restrict the resources used for a single request. Also, the policies can restrict the frequency of requests from a single object.

Exporters can mislead a trader to execute unwanted operations by selecting improper interfaces to be used for dynamic property value evaluators. The trader administrator can prevent this in two alternative ways. First, the trader's export action policy may restrict the interface types within each offer. Second, the administrator can deny the use of dynamic properties totally.

Trading service availability

The trading service is in many cases a potential single point of failure in a system. Therefore, special care must be taken to ensure the dependability of the traders.

The trading service availability can be improved by traditional database replication mechanisms with log-based recovery [32]. However in some systems, the recovery phase may be too long in comparison to the frequency on which offers or property values are changed.

Another approach for improving trading service availability uses active replicas and a proxy agent that multiplexes the requests to a selected object [259, 126]. The multiplexing decisions are done based on the response times of the replica group members. The mechanisms also allow load balancing within the replica group.

Federated security services

Security of a federated system requires that the peer objects are trusted and that there exists a trusted third party for authentication [165]. The main goal in current security services is to protect the communication that is assumed to tunnel via a hostile environment.

In principle, federable systems are free to join federations, but the community of federated systems only accepts new members if it has reasons to trust the new member. In current systems, like Internet, a system is trusted, if an existing member agrees to join it to a well-administered naming domain. However, this kind of mechanism is not sufficient for secure federated systems. All addressable systems are not equally trustworthy [237].

The OMG consortium is working on a secure interworking model [171]. The model trusts in a shared OMG platform architecture. The mechanism is not sufficient for federated systems.

A solution to the trust problem may be found from other areas of human life: law of contract and notary. When two organisations make a legal contract on something they can ask the services of a notary. The notary may have records about the trustworthiness of the organisations, and the notary usually stores a copy of the contract so that it cannot be repudiated. In computer networks national notary can be used for logically 'closing' the network. Anybody can connect a computer to the network, but everybody is considered suspicious in the beginning. If a trusted binding is required, a suspicious system may not be involved in the establishment process. Everybody is however free to request that a notary gives a ranking for the system trustworthiness. In the binding process between objects, an object may be reachable or not depending on the rank of its supporting system. The notaries can be used for building a set of semi-trusted systems. The PGP system (pretty good privacy) [260] already presents a system with this kind of features. However, the problem of trust between systems cannot be solely solved by the computer science methods, instead international laws must be introduced on the field.

Chapter 7

Realising trading functionality

Trading functionality can be implemented on top of the platform services discussed in the previous chapter. However, the appropriate design it is not straightforward. The design must consider, beside the local organisation, also the cooperation facilities of the trader. In this chapter, we discuss the effects of two different approaches. The first approach trusts in a set of integrated middleware services and resolves all trading and type system related problems within the trader. The second approach exploits the federated system model at various layers: it builds trading services on transparent naming federations and on transparent type system federations.

7.1 Interworking model

In the interworking model, the trader designer trusts that the cooperating traders work on platforms with a shared architecture. Therefore, assumptions can be made on communication system, naming system, subtyping system and description languages used to describe types. In this environment, the only problem related to administrative domains is the potential use of different type systems. Therefore, separate type repositories are used for each administrative domain, and the traders explicitly make type queries to these type repositories. The design is illustrated in Figure 7.1.

In this interworking model, the administrative domain is equivalent to the type system domain. This interpretation is commonly used in projects working with CORBA platforms.

The design problems to be solved in this model include the following questions:

- How a trader propagates a request to another trader?
- How modification or deletion of a type definition affects trading?
- How modification in a type system is propagated from one trader to others?

When a trader intends to propagate a request (import or export) to another trader, it must first check that the type definition involved in the operation is similar on each type system domain or whether some translation is necessary. The mechanisms for doing this were already discussed in Section 6.2: searching based on type name, asserted relationship, or comparison of interface signatures.

Modification of a type definition causes new versions of the types to be created. The existing service offers and links are preserved by the trader, but new offers are expected to refer the new

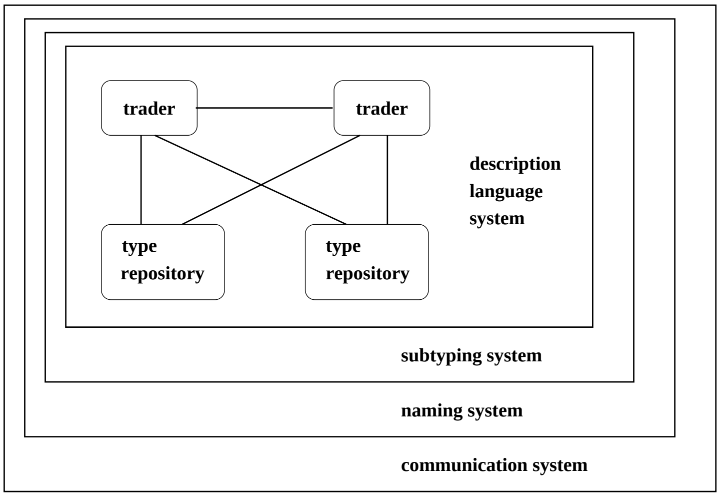


Figure 7.1: Interworking model.

type version. Similarly, type deletion takes full effect only after the final offer has been withdrawn from the trader.

If cooperation contract protocol (recall Section 5.3.6) is used, the trader whose type system is changed, can start a renegotiation of the cooperation contract each time a service type is introduced, modified, or deprecated. In other cases, traders must actively browse each other's type systems before each interaction. The amount of queries can be decreased by caching immutable type definitions of other type systems within the trader.

7.2 Federated model

In the federated model, the trader designer assumes that the cooperating traders work on separate platforms without a shared architecture and have separate designs for type systems. Even name systems are not necessarily under shared control. Therefore, assumptions can be made only on communication system.

In this environment, it would not be reasonable to expect the traders to explicitly manage all administrative domains of all the middleware services. Instead, all middleware services including the type repositories are exploited in a federation transparent way. The traders make only local queries to the local type repositories, and local name servers, which in turn take care of cooperation with the corresponding services at the other trader's domain. The design is illustrated in Figure 7.2. It should be noted, that there is no requirement for the naming system domains and type repository domains to coincide with trading domains. Each trader has one interface for each of these services to support it, but for example a type repository may support many trading domains.

The design problems to be solved in this model include the same questions as in the interworking model. In the designs, the responsibilities for actions have been placed differently. The

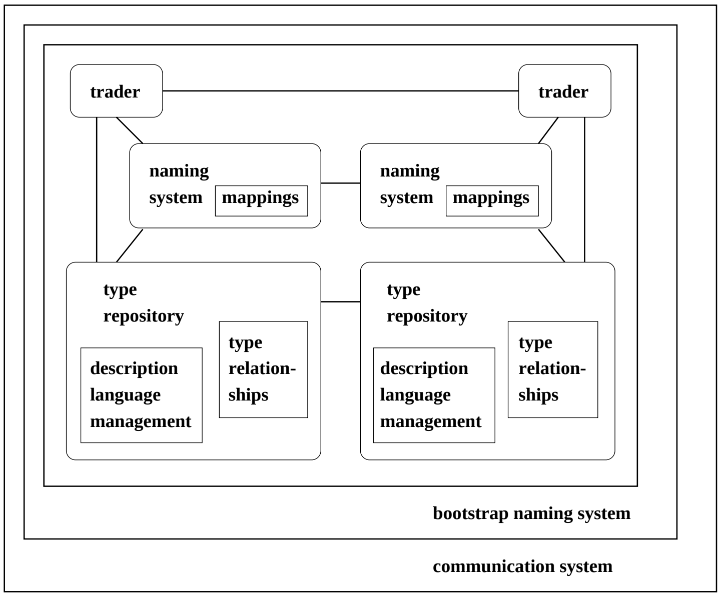


Figure 7.2: Federation model.

supporting services must be federated and offer a federation transparent communication environment for the traders. The situation is briefly studied in Section 7.3.

The federation model introduces also a new problem. The model cannot assume any more that all service types are defined in a single language, not even within a single type system. This problem is solved by including several language expressions into each type definition, as described in Section 6.2.

7.3 Channels between traders

In order to understand how a trader propagates a request to another trader in the federated model, we study the channel between two traders. The example illustrated in Figure 7.3 is an extreme case. Most federations are expected to occur between domains that have some resemblance with each other. In this section, we only consider the structure of the channel, in Chapter 9 we study how the channel is created. For the federation model, only the two topmost components are interesting.

In the federated model, the trader constructs the request in the format it would require from its own clients. This is in contrast to the interworking model, where the propagating trader must format the request in such a form that is required by the clients of the target trader. The channel between the two traders is, in the federated model, able to do all necessary transformations for the request.

The client stub (Figure 7.3) represents the client role interface. It is not a generic RPC client stub, but an entity that is responsible of checking and translating application type specific aspects

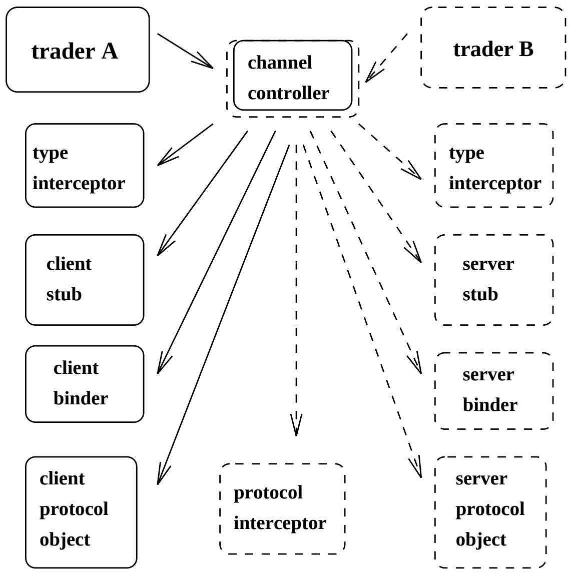


Figure 7.3: Channel between two traders.

of messages that pass through the channel. In this case, the client stub must be aware of the import and export operation parameters, like service type and associated property types. The client stub interrogates the local type repository for a suitable transformation routine for the message based on the local service type name and the target trader interface reference.

The type repository server is able to identify the target type repository by the information it can find based on the target trader interface reference. The type repository server may trust in the locally asserted relationships to the target system types and return a reference to a suitable interceptor. On the other hand, the type repository server may invoke a search of conformant types at the target type repository server. The result of this search is then cached as asserted relationships.

In comparison of the interworking model, the federated model moves two tasks from the trader design to other system components. First, the transformation of trading related requests is moved from the trader object to the client stub. This move allows the trader implementer to exploit generic binding services for federated systems. Generic services are more critical for the system behaviour and performance, and therefore the developers make more effort for their correctness and optimisation. In addition, usage of a generic service also reduces the amount of heterogeneity in the systems. Second, all type conformance related aspects – decisions related to propagation of type changes and caches of relationships – are embedded to the type repository service.

7.4 Performance considerations

The models can be characterised by the amount of network traffic they generate, the amount of storage they need at each trading domain, and the expected response times for import operations.

We can study the behaviour of the trader and type repository combination in three different scenarios. In the first scenario, traders hold standard links to each other. In the second scenario, traders negotiate cooperation contracts. In the final scenario, traders hold type specific links to each other. These scenarios do not differentiate whether the stored information is within the trader or within the type repository and thus the scenarios apply equally both to the interworking and to the federative model.

In the first scenario, three queries are involved each time a cooperative trading request is processed. First, the importing trader queries the target trader for the type repository interface. A standard link contains an interface reference to target trader interfaces for importing and exporting. Either of the interface references can be used for interrogating the target trader for its type repository interface. Then, a query for conformant type specifications is performed. For each trading request, only the service type requested is checked. Finally, the actual trading request is performed.

This scenario implies that each import request requires six messages to be passed across the network. However, cached information does not exist and thus no additional memory requirements are present. The drawback is however the slow response times because of the high traffic.

As an optimisation, the responses for conformant type specifications can be stored when queried for the first time. As well, the type repository interface of each trader can be stored for further use at the initial query. However, the information is updated at some intervals. This optimisation behaves like the third scenario, when implemented efficiently.

In the second scenario with cooperation contracts, queries about the type repository and conformant types are performed before entering a cooperation phase. The contract is stored on both the requesting trader and the target trader. It contains all type relationship information applicable between the two traders. The negotiation is performed each time the type space of either trader changes. However, during a single trading operation after the negotiation, only a single query for the actual import is needed.

The drawbacks of this scenario are high memory requirements and high deviation of expected response times. The memory requirement at each trader increases linearly when either the number of cooperating traders increases, or the number of service types increases. The response times of import operations fall in two groups. Fast responses are achieved under normal operation when the cooperation contract is in effect. Each import operation only involves a request and a reply message. Very slow responses are given when an import is started during a cooperation contract negotiation. The import operation must wait until the negotiation is finished, which requires an interrogation for each involved service type.

In the third scenario, the previous two approaches are combined. A link is negotiated when a change takes place, but the amount of information stored or cached is restricted. Also the amount of messages is restricted, because only one service type is involved. The requesting trader has grouped the links based on its local type system. Each of the type specific links carry both the target trader and the target type repository references, and also cache for applicable interceptors.

7.5 Global trading system organisation

In a global environment, the performance of the trading service is dominated by network delays in communication. We have learned, that trading graphs should be restricted in depth depending on the network delay (LAN or WAN) to 3-4 hops (recall from Figure 5.6). We have also learned that the best method for decreasing the load of a single trader in the global system is to isolate different service types to separate traders (recall from Figure 5.11).

The study of cooperation between traders and type repositories have further shown, that each type system involved creates an additional negotiation level. Instead of using 3-4 hops solely for the cooperation between traders, we have to allow 1-2 cycles for type repository negotiations. The type repositories can work on 1-2 interrogations only if the trader links are either optimised standard links or type specific links. This gives further support for the organisation of offers based on their service type.

Similarly to trading, type repositories also benefit from caching and prefetching strategies.

Chapter 8

Discussion on some traders

In this chapter, we discuss some trader realisations. Two trader designs are discussed in length, the ODP trading function standard [96, 108, 7] and the DRYAD trader prototype [126]. From some other traders we point out their contributions. The chapter is concluded with a comparison of the designs.

8.1 ODP trader

In the ODP reference model standard the role of trading is expressed in two ways. Firstly, the trading function is defined to offer means to advertise and retrieve information about available services in the system, i.e., to mediate information about object interface instances. Secondly, each object in the system is statically bound to a trading service interface. Only the trading service is guaranteed to be available for all objects, although for example a binding service would be more essential as we will see in Chapter 9.

The ODP trading function is not dependent on any specific platform, instead the assumed environment is expressed in terms of ODP engineering viewpoint language.

OMG has adopted the ODP trading function standard computational specification as its object trader service [169, 173]. The OMG submission is a joint submission of AT&T/Lucent Technologies, CRC for Distributed Systems Technology, Digital Equipment Corporation, Hewlett-Packard Company, International Computers Limited, Nortel Limited and Novel, Inc.

The TINA architecture also includes a trading service in the DPE architecture [213, 224]. The trading service sketched resembles the 1992 working document on ODP trading [87]. However, we expect the current ODP trading function standard to be adopted and implemented in TINA auxiliary projects.

8.1.1 Trader object composition

All trader objects contain

- a set of offers available for import,
- a set of attributes that provide information about policies affecting trader behaviour at run-time, or affecting the import policy of a propagated import,
- a set of interfaces.

Traders can contain different sets of interfaces, thus providing different levels of functionality. The functionality levels are expressed as standardised conformance classes.

Depending on the conformance class of the trader, it can also contain

- a set of links representing paths for propagating queries to other traders, and
- a set of proxy offers that are an intermediate form between offers and links.

We will first review the offer space and trader attributes. Then we will review the activities related to each conformance class separately. The two cooperation constellation forms, exploiting links or proxy offers, will be discussed in the context of the appropriate conformance classes.

Offer space

The ODP trader can include normal or proxy offers. We discuss proxy offers later in Section 8.1.6. The normal offers include a property name, service type name and a sequence of properties (Figure 8.1).

The trader can exploit the service type name for querying the type repository for an interface type name for the service described, and a list of properties. Each property is defined by its name and data type, and it can in addition be flagged to be 'mandatory' and 'readonly'. A non-flagged property is considered to be 'changeable' and 'non-mandatory'. A 'mandatory' property must always be present in all offers of that service type. A 'readonly' property is considered immutable.

```
typedef string Istring;
typedef Istring PropertyName;
typedef sequence<PropertyName> PropertyNameSeq;
typedef any PropertyValue;
struct Property {
       PropertyName name;
       PropertyValue value;
};
typedef sequence<Property> PropertySeq;
struct Offer {
       Object reference;
       PropertySeq properties;
};
struct OfferInfo {
       Object reference;
       ServiceTypeName type;
       PropertySeq properties;
};
```

Figure 8.1: IDL specification of offers in ODP traders [96].

Attributes to describe trader policies

The trader policies specify which optional capabilities are supported by the trader. The trader attributes also express what kind of offers are accepted from exporters and what facilities are available for importers. These policies are made available to trader attributes (Figure 8.2) so that external objects, like other traders, can query the attribute values.

The ODP trader includes three methods for managing changes of service offers: dynamic properties, modifiable offers, and proxy offers. For dynamic properties, normal service offers include a reference to a private service offer evaluator interface for each dynamic property. A

```
interface SupportAttributes {
        readonly attribute boolean supports_modifiable_properties;
        readonly attribute boolean supports_dynamic_properties;
        readonly attribute boolean supports_proxy_offers;
        readonly attribute TypeRepository type_repos;
};
```

Figure 8.2: IDL specification for trader policy attributes [96].

proxy offer includes only an evaluator interface that can be used to interrogate for a suitable service offer. The benefit of proxy offers is that the external evaluator may do other tasks in addition to the offer selection. For instance, the evaluator may reserve resources for the service, log usage of the service, or the server can be instantiated as the service offer becomes selected. The third method of making changes to service offers is to use the modify-service-offer operation that is specified for exporters to use.

The trader policy attributes also specify which type repository interface it uses to obtain type descriptions.

The standard also specifies another set of trader attributes. These attributes set technical limitations for the import operation execution (Figure 8.3). These policies restrict the number of offers to be searched, the number of matching offers to be sorted, the number of matching offers to be returned, and the number of links to be traversed in sequence in a path leading from trader to trader. The scoping policies also require the trader to state, when it federates with other traders: always, only when no local offers are suitable, or never.

```
interface ImportAttributes {
        readonly attribute unsigned long def_search_card;
        readonly attribute unsigned long max_search_card;
        readonly attribute unsigned long def_match_card;
        readonly attribute unsigned long max_match_card;
        readonly attribute unsigned long def_return_card;
        readonly attribute unsigned long max_return_card;
        readonly attribute unsigned long max_list;
        readonly attribute unsigned long def_hop_count;
        readonly attribute unsigned long max_hop_count;
        readonly attribute FollowOption def_follow_policy;
        readonly attribute FollowOption max_follow_policy;
};
```

Figure 8.3: IDL specification for trader import policies [96].

These policies can be nominated by the importer too, but during an import the actual value is limited by the corresponding trader attribute. 'Search_cardinality' is a policy for nominating upper bound for the number of offers to be searched. 'Match_cardinality' is a policy for nominating upper bound for the offers to be selected in the search process. 'Return_cardinality' is a policy for nominating upper bound for the offers to be returned to the importer. 'Hop_count' is a policy for nominating upper bound for the depth in the trading graph to be traversed. 'Max_list' determines the upper bound for the number of items in any list returned by the trader. The rest of the items to be returned must be included to the corresponding iterator object.

Both attribute sets are packed to interface structures in the IDL specifications (Figures 8.2 and 8.3). This is the idiosyncrasy of IDL for expressing that other objects – like other traders – are allowed to query the attribute values.

ODP trader interfaces and conformance classes

The ODP trading function standards define five interfaces: Lookup, Register, Link, Proxy and Admin. Lookup interface supports operations for importing offers from the trader. Register interface supports operations for modifying the set of normal offers stored within the trader. Proxy interface is analogous with Register interface, but supports operations for modifying the set of proxy offers instead. Link interface supports operations for creating and modifying links to other traders. Admin interface supports operations for modifying the trader attribute values. Not all traders are required to include all these interfaces.

The ODP trading function standard specifies six conformance classes for implementations: query trader, simple trader, stand-alone trader, linked trader, proxy trader, and full-service trader. The trading function features are linked in a set of interfaces so that the conformance classes can be constructed by composing the interfaces, as shown in Figure 8.4.

```
interface TraderComponents {
          readonly attribute Lookup lookup_if;
          readonly attribute Register register_if;
          readonly attribute Link link_if;
          readonly attribute Proxy proxy_if;
          readonly attribute Admin admin_if;
};
```

Figure 8.4: IDL specification for a full-service trader object [96].

8.1.2 Query trader

A query trader can serve a Lookup interface, that implements only a 'query' operation. The query operation interface is specified in IDL in Figure 8.5. The input parameters represent matching criteria ('constr'), preference criteria ('pref'), and technical selection criteria ('policies'). The potential content of the 'policies' parameter is represented in Figure 8.6. Most of the import policies restrict the same aspects as the trader import policies. However, importer can also restrict type conformance to exact match instead of allowing subtyping rules to be used. In addition, the importer can force the query to be started at another trader by specifying a starting trader. Further parameters of the query operation define what are the desired property values to be returned.

The output parameter set of the query operation introduces a combination of techniques for transporting offer contents. The first output parameter includes a list of offers represented as pairs of field name and value. The second output parameter includes an offer iterator[1] interface, that can be used for retrieving offer object interface references. Depending on the platform, either of these parameters is natural to use. The trader attribute 'max_list' defines the length of the list in the first parameter, and thus the list may be empty. The offer iterator must always be present, although the number of offers within the iterator may be zero. The last output parameter expresses which import policies restricted the query processing. If the list is empty, the search space was exhausted.

The doubling of the actual output parameter, for allowing two technical solutions, is clearly a committee compromise. However, this compromise is unnecessary: The prescription is given

[1] Iterators are commonly used in IDL specifications. An iterator is an object that contains a set of objects of the requested type. The iterator interfaces supports an operation for querying how many objects are still available, an operation for accessing the next object in line, and finally an operation to destroy the iterator object itself. In other words, an iterator is a small repository object that implements a finite, typed list of objects and can be destroyed on request.

```
void query (
     in ServiceTypeName type,
     in Constraint constr,
     in Preference pref,
     in PolicySeq policies,
     in SpecifiedProps desired_props,
     in unsigned long how_many,
     out OfferSeq offers,
     out OfferIterator offer_itr,
     out PolicyNameSeq limits_applied
) raises (
     IllegalServiceType, UnknownServiceType,  IllegalConstraint,
     IllegalPreference,  IllegalPolicyName,   PolicyTypeMismatch,
     InvalidPolicyValue, IllegalPropertyName, DuplicatePropertyName,
     DuplicatePolicyName
);
```

Figure 8.5: Query operation [96].

```
struct LookupPolicies {
       unsigned long search_card;
       unsigned long match_card;
       unsigned long return_card;
       boolean use_modifiable_properties;
       boolean use_dynamic_properties;
       boolean use_proxy_offers;
       TraderName starting_trader;
       FollowOption link_follow_rule;
       unsigned long hop_count;
       boolean exact_type_match;
       OctetSeq request_id;
};
```

Figure 8.6: Policies for governing the query operation [96].

at the computational viewpoint, which deals only with information exchange patterns between logical objects, not actual data structures. Therefore, the two solutions would be equally valid engineering viewpoint refinements for the computational interface with a sequence of offers. The use of IDL as a computational description language misleads the reader to consider the specification more concrete as it should be. The actual realisations would implement either solutions, and cooperation across the technical boundary would only need an interceptor. With the current solution, each implementation is practically forced to include both techniques.

In the various development phases of the ODP trading function standard, a sequence of slightly differing language specifications has been presented. The major problem has been that, as we recall from Chapter 5, there are both a logical and a technical reason for ordering the matching offers to be returned to the importer. Different ways of combining the criteria to one or more criteria parameters require a slightly different syntax.

In the final standard, there is only one criteria language. Only the expressive power of the language is enforced by the standard but also gives a universal syntax. Unfortunately, the final version of the language is the most restrictive among the proposed languages. The language includes features that were already presented in Section 5.2.5. The constraint language BNF is included to the standard and presented in Figure 8.7.

Both the trader policies and the importer policies may independently inhibit the usage of dynamic properties in normal offers, inhibit the usage of proxy offers, or inhibit the usage of the modify operation. Reasons given for such inhibition are normally security related, sometimes

```
<constraint>    := /* empty */ | <bool>
<preference>    := /* empty */ | min <bool> | max <bool>
                 | with <bool> | random  | first
<bool>          := <bool_or>
<bool_or>       := <bool_or> or <bool_and> | <bool_and>
<bool_and>      := <bool_and> and <bool_compare> | <bool_compare>
<bool_compare>  := <expr_in> == <expr_in> | <expr_in> != <expr_in>
                 | <expr_in> <  <expr_in> | <expr_in> >  <expr_in>
                 | <expr_in> <= <expr_in> | <expr_in> >= <expr_in>
                 | <expr_in>
<expr_in>       := <expr_twiddle> in <Ident>| <expr_twiddle>
<expr_twiddle>  := <expr> ~ <expr> | <expr>
<expr>          := <expr> + <term> | <expr> - <term> | <term>
<term>          := <term> * <factor_not> | <term> / <factor_not> | <factor_not>
<factor_not>    := not <factor> | <factor>
<factor>        := ( <bool_or> ) | exist <Ident> | <Ident>
                 | <Number> | - <Number> | <String> | TRUE | FALSE
```

Figure 8.7: BNF definition for the ODP trader criteria language [96].

also performance related.

If the trader supports dynamic values, it must do so by using DynamicPropEval interface signature (Figure 8.8). The evaluation behaviour is not defined but left for further study. Committee conflicts, that were caused by defining the dynamic property evaluation interface, were masked by defining the input argument 'additional info' to carry all those extra arguments suggested.

```
module CosTradingDynamic {
exception DPEvalFailure {
          CosTrading::PropertyName name;
          CORBA::TypeCode returned_type;
          any extra_info;
};
interface DynamicPropEval {
any evalDP (
          in CosTrading::PropertyName name,
          in CORBA::TypeCode returned_type,
          in any extra_info
) raises (
           DPEvalFailure
);
};
struct DynamicProp {
           DynamicPropEval eval_if;
           CORBA::TypeCode returned_type;
           any extra_info;
};}; /* end module CosTradingDynamic */
```

Figure 8.8: Interface signature for dynamic property evaluator [96].

8.1.3 Simple trader

A simple trader can serve Lookup and Register interfaces. The register interface (Figure 8.9) must support operations 'export', 'withdraw', 'describe', 'modify', 'withdraw_using_constraint', and 'resolve'. A trader that accepts offers must specify what restrictions it has to the offer structure.

The export operation takes an interface reference and a list of property values together with a service type name. It returns an object identifier, that can be further used as a key to the offer space in 'withdraw', and 'modify' operations.

```
OfferId export (
          in Object reference,
          in ServiceTypeName type,
          in PropertySeq properties
) raises (InvalidObjectRef,          IllegalServiceType,
          UnknownServiceType,        InterfaceTypeMismatch,
          IllegalPropertyName,       PropertyTypeMismatch,
          ReadonlyDynamicProperty,   MissingMandatoryProperty,
          DuplicatePropertyName
);
void withdraw (
          in OfferId id
) raises (IllegalOfferId, UnknownOfferId, ProxyOfferId
);
OfferInfo describe (
          in OfferId id
) raises (
          IllegalOfferId,
          UnknownOfferId,
          ProxyOfferId
);
void modify (
          in OfferId id,
          in PropertyNameSeq del_list,
          in PropertySeq modify_list
) raises (
          NotImplemented,      IllegalOfferId,
          UnknownOfferId,      ProxyOfferId,
          IllegalPropertyName, UnknownPropertyName,
          PropertyTypeMismatch,ReadonlyDynamicProperty,
          MandatoryProperty,   ReadonlyProperty,
          DuplicatePropertyName
);
void withdraw_using_constraint (
           in ServiceTypeName type,
           in Constraint constr
) raises ( IllegalServiceType,  UnknownServiceType,
           IllegalConstraint,   NoMatchingOffers
);
Register resolve (
           in TraderName name
) raises ( IllegalTraderName,   UnknownTraderName,
           RegisterNotSupported
);
```

Figure 8.9: IDL for the register interface operations [96].

The other operations at the interface – 'describe', 'withdraw_using_constraint', and 'resolve' – are redundant.

'Describe' is a subset of query operation at the Lookup interface. The motivation of including a simple query operation at the Register interface was related to conformance requirements. The simple query operation allows exporter objects to check whether their offers are correct and up-to-date and still be bound to the Register interface only. This is claimed to decrease the implementation cost of exporters because they are not required to include full query operation just for being able to have a type conformant interface with the trader they use.

'Withdraw_using_constraint' replicates the 'withdraw' operation. Instead of the offer identifier the operation gets a matching constraint as an input parameter. The constraint language is capable of presenting a constraint that simply identifies an object. The operation 'withdraw_using_constraint' is result of the different design goals. On one hand, a single operation that is suitable for both end-user and administrative needs was expected. On the other hand, a simple operation was expected that can be required in all implementations. However, in the conformance requirements the whole interface is required. Thus, there is no acceptable way of supporting only the simple withdraw operation.

The 'resolve' operation translates a trader name to an interface reference, based on the naming system supported by the trading graph within a single administrative domain. If the trading graph exceeds the limits of an administrative domain, no trustworthy knowledge about the link names can be available.

8.1.4 Standalone trader

A stand-alone trader has the properties of a simple trader and includes in addition an Admin interface. The Admin interface includes operations for setting the various policy related attributes in the trader. It also has browsing operations for the offers and proxy offers.

8.1.5 Linked trader

A linked trader is similar to a stand-alone trader but has an additional Link interface for manipulating links. A linked trader must also be conformant to a Lookup interface in an importer role.

Links can be used for operations on Lookup, Register, and Proxy interfaces. In earlier versions of the ODP trading function standard, several cooperation scenarios were attempted. In some phases, the cooperation between traders took place only by allowing a trader to import from another trader. In still earlier versions of the standard, a very detailed protocol was defined for maintaining a channel between the cooperating traders. The protocol required additional functionalities of traders, especially a private binding mechanism. The protocol also required a great amount of information to be cached for each potential cooperation partner. Finally, need for different levels of cooperation relationships was identified, and only the basic version with import capabilities was expected to be included into the standard.

A link

- refers to a Lookup interface of a target trader,
- refers to a Register interface of a target trader,
- includes the link's default follow policy, that is used when an importer does not specify a 'link_follow_rule' policy, and
- includes the link's limiting follow policy, that overrides the importer's 'link_follow_rule' in cases where the limiting follow policy is exceeded.

'Follow_policy' specifies under which conditions the link can be traversed. The policy values are 'local_only', 'if_no_local' and always, in an ascending order. The value 'local_only' indicates that the link is never followed unless explicitly named in an operation. The value 'if_no_local' indicates that the link is followed when the local search could not provide any offers. The value 'always' indicates that the link is always followed unless some policy denies the propagation. Each link has private follow policies included. They are used when an import is traversed through the link. The corresponding trader properties give values to be used when a link is created.

The Link interface signature is given in Figure 8.10. The interface supports operations 'add_link', 'remove_link', 'modify_link', 'describe_link' and 'list_links'. Operation 'add_link' creates a new link structure, and therefore, the target trader Lookup interface is given. In addition, the follow policies default and maximum values are defined. However, the link structure includes also a reference to the Register interface of the target trader, which cannot be initiated to the link structure with ‘add_link’ operation. Neither is operation ‘modify_link’ able to manipulate this information item. Operation ‘modify_link’ can only manipulate the follow policy values. Link name is used as an identifier for all link manipulation operations. ‘Describe_link’ shows the values stored to a single link structure, while ‘list_links’ produces a list of link names only.

```
void add_link (
in LinkName name,
           in Lookup target,
           in FollowOption def_pass_on_follow_rule,
           in FollowOption limiting_follow_rule
) raises (
           IllegalLinkName,            DuplicateLinkName,
           InvalidLookupRef,
           DefaultFollowTooPermissive, LimitingFollowTooPermissive
);
void remove_link (
          in LinkName name
) raises (
          IllegalLinkName,  UnknownLinkName
);
LinkInfo describe_link (
          in LinkName name
) raises (IllegalLinkName,  UnknownLinkName
);
LinkNameSeq list_links ( );
void modify_link (
          in LinkName name,
          in FollowOption def_pass_on_follow_rule,
          in FollowOption limiting_follow_rule
) raises (IllegalLinkName,              UnknownLinkName,
          DefaultFollowTooPermissive,    LimitingFollowTooPermissive
);
```

Figure 8.10: IDL specification for Link interface [96].

For interworking between traders, an arbitrary network of traders can be established, and therefore, loops can occur during imports. In the ODP trading function standard, using a hop count that terminates the process when the hop count reaches zero, was not considered effective enough. More active loop detection can be performed in a distributed manner so that each request carries a request identifier and each trader remembers all recent queries received from other traders. The ‘request_id_stem’ is used in the mechanism for loop control to create a separate value set for request identifiers at each trader. The solution contradicts the federated architecture model, but some reasoning behind the solution was given: most loops involve only two traders and the amount of unproductive processing can be huge. However, we have already concluded in Chapter 7 that a reasonable hop count in most cases is 1-3, which restricts the loops efficiently.

8.1.6 Proxy trader

A proxy trader is similar to a stand-alone trader but has in addition a Proxy interface (Figure 8.11)for manipulating proxy offers. A proxy trader must also be conformant to a Lookup interface in an importer role.

The proxy offer contains service type and properties of a service offer. It is matched in the same way as normal offers. However, the proxy offer refers to a Lookup interface instead of

```
struct ProxyInfo {
        ServiceTypeName type;
        Lookup target;
        PropertySeq properties;
        boolean if_match_all;
        ConstraintRecipe recipe;
        PolicySeq policies_to_pass_on;
};
OfferId export_proxy (
        in Lookup target,
        in ServiceTypeName type,
        in PropertySeq properties,
        in boolean if_match_all,
        in ConstraintRecipe recipe,
        in PolicySeq policies_to_pass_on
) raises (IllegalServiceType,      UnknownServiceType,
        InvalidLookupRef,          IllegalPropertyName,
        PropertyTypeMismatch,      ReadonlyDynamicProperty,
        MissingMandatoryProperty,  IllegalRecipe,
        DuplicatePropertyName,     DuplicatePolicyName
);
void withdraw_proxy (
        in OfferId id
) raises ( IllegalOfferId, UnknownOfferId,
          NotProxyOfferId
);
ProxyInfo describe_proxy (
        in OfferId id
) raises (IllegalOfferId, UnknownOfferId,
         NotProxyOfferId
);
```

Figure 8.11: IDL specification for proxy offer [96].

the actual offered service. Therefore, the request is modified using the 'ConstraintRecipe' rules and forwarded to the Lookup interface. The original type parameter, preference parameters, and desired properties are passed on unchanged, but constraint parameters, policies and cardinalities are modified to suit the trader needs. With the parameter 'if_match_all' the proxy offer exporter can notify the trader that the proxy offer is a valid match for all imports of the specified service type irrespective of the importer criteria on property values. This approach would make the proxy offer similar to a type specific link, without any control mechanism for import propagation.

The proxy offer works like a simplified link. The type system in the object referred to and in the calling trader must be equal, but the query interface signature in the referred object can differ. Therefore proxy offers can be used to encapsulate other objects than traders to the trading system. The same functionality could be expressed by using interceptors together with links.

8.1.7 Full-service trader

A full-service trader includes Lookup, Register, Admin, Link and Proxy interfaces. It also has to be conformant with Lookup and Register interfaces in importer and exporter roles correspondingly.

8.1.8 Exploitation relationship to other services

The ODP trading function is expected to exploit the ODP type repository service. However, the schedules of the standards are such that the ODP trader function had to have a private, simple type system specified. It is going to be replaced by the type repository function standard [97] when it reaches international standard status. However, there is a discrepancy, because OMG assumes the type repository documented in the trading function standard to be sufficient and thus final. For CORBA platform purposes the trading type repository is sufficient, as we can recall from Sections 4.4 and 6.2. However, for open system and for federation between open systems the solution is not sufficient, because it does not include any mechanisms for defining relationships between type descriptions.

The ODP trading function specification also mentions that a federated naming service is required. However, the relationship between naming and trading is not clarified, except for the 'resolve' operation that indicates a number of private, global naming systems to exist.

An additional standard gives requirements for systems that implement ODP trading function on top of X.500 Directory service [100]. The standard has difficulties in exploiting the ODP reference model, because X.500 does not follow ODP guidelines. However, the mapping can be done. For X.500 technology dependent reasons, the standard prescribes also some aspects not included in the trading function specification itself [96]. These aspects include some security considerations, data formats, and policies.

8.1.9 Conformance testing

For each standard, a set of conformance tests should be defined, to allow vendors to assure that their implementations behave according to the standard requirements. The test cases for trading service were supposed to be created on the basis of a formal specification of the functionality. For earlier trading standard versions, such a specification existed [88].

Currently there is no conformance testing arranged for trader implementations, because the conformance testing specification for traders is missing [101]. The major problem seems to be the lack of general framework for the ODP conformance testing methodology.

8.2 DRYAD trader

The DRYAD trader is a prototype service developed during the research leading to this dissertation.

The DRYAD trader is designed to be part of the federated binding process, but there are no restrictions to use the mechanism for other purposes, too. The DRYAD trader design has initially been influenced by the very early ODP trader drafts, but later been used to initiate contributions to the ODP trading function and other component standards. In the following, we first review the DRYAD trader design and discuss then some performance measurements.

8.2.1 Trader object composition

The DRYAD trader object (Figure 8.12) [125] consists of

- a trader repository object, that has two parts: the offer space for information about offered services, and the trader address storage for finding other traders from the network,
- a type repository object,
- a policy repository object, that includes trader's import and export action policies,
- a template repository object for creation of new processing entities, and
- a concierge object to control the processing.

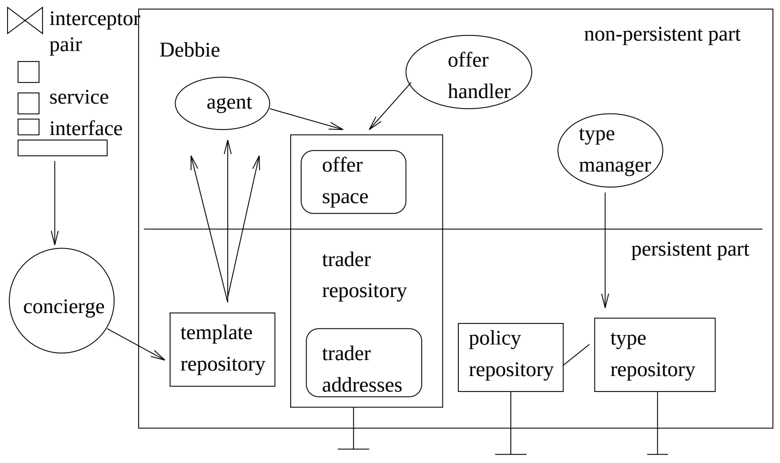


Figure 8.12: The DRYAD trading system components.

The repositories are aggregated into one object. The trader interface is controlled by a concierge object, and the repositories rely on Debbie database system [130] services.

Initially only one active object, a concierge, is running together with the Debbie system. The concierge watches the incoming communication ports (import, export and administration) and requests the Debbie engine to create a new active object to handle each incoming message. The concierge was also designed to support a replica group of trader component objects [259] for fault tolerance. The active objects are instantiated by letting the Debbie system to execute methods from object templates that are stored in the template repository.

Each repository object has one or more active processing objects working on it: The trader repository has agents for handling import requests and offer handlers for handling export and withdraw requests. Each import is processed by a private agent instance. The type repository has a type manager to response to queries.

The repository objects themselves are passive. The Debbie database can store persistent and non-persistent information. The offer space of the trader repository is by a policy decision stored non-persistently. The other repositories must be persistent, because they are used at the start-up of the trader.

The trader object is accompanied by interface interceptors. The interceptors are exploited when import requests are directed from a DRYAD trader to another trader implementation, or a foreign trader imports from a DRYAD trader. A DRYAD trader offers two types of interfaces, a native interface and an IWT interface for importers [178]. The IWT interface is specified by the IWT project (Interworking of Traders) [246].

In the following sections, the type and policy repository structures are introduced. As the components coexist in the same database, also an allocation scheme is shown: an implementation object may contain both a type object and a set of policy objects. This introduction is followed by a description of offer space objects.

8.2.2 Internal type repository

The type repository includes a type description object for each service type. The object is a compound object and it has the following structure:

- the abstract type name,
- a verbal, natural language description of the service type semantics,
- information about the service type's visibility to other trading domains,
- a set of attribute objects, and
- a set of interceptor objects.

The attribute objects in turn consist of the attribute name, verbal description, data value type, unit of value, and default virtual value of the attribute. Technically it also includes caching time for the evaluated values. Examples of attribute specifications are

```
{name = InterfaceSignature,
data type = XDR description (corresponds to IDL descriptions),
unit = NIL (not applicable with this data type), default value = NIL,}

{name = Location, data type = set of addresses, unit = NIL,
default value = NIL}

{name = QueueLength, data type = Integer, unit = Bytes,
default type = "lpq -P$ID"}
```

Each type interceptor object defines the relationship between a locally known type and a type known in the remote domain. As import requests can span several type domains, translation between logically similar but technically distinct types is required. To carry the required mapping information, we introduce an interceptor object that consists of

- local name for the remote type management domain,
- name for the wanted service type within the remote type management domain, and
- for each local attribute a rule for transforming the values retrieved from the remote trader.

The transformation rules are constructed from arithmetic and string manipulation operations over the attribute values. Also some functions are available for accessing the attribute type information from the remote type domain: data type, high and low limits of potential values, and unit of the attribute value. The transformation rules can also include conditional statements and comparisons.

The type repository of the DRYAD trading system assumes that service types are matched by names. It also assumes that service types are abstract and therefore can be represented by several interface types. The importers request for offers of a certain interface signature type. Subtyping of service types or interface signature types are currently not supported.

The example in Figure 8.13 shows what kind of attributes could be used in the DRYAD trading system. The example service type is an operation to sell a book. Books can be purchased by naming the type of book, for example a detective story, the author, the range of price, or the language. This helps to define with which application specific parameters the offered operation is able to work. Furthermore, we can include technical attributes, such as time intervals that are allowed for computing, and technical details about the interface location and access technique. Especially, the actual parameter list and operation name can be included as attribute values, that mediate between application concepts and the implementation technique. These attributes are based on the binding liaison.

```
stringset book-type
string author
interval prize
stringset language
choice [interrogation | announcement ] communicationstyle
interval timetocompute
signature invocationtechnique
interface identifier  (or addressstruct location)
```

Figure 8.13: Example of a service type "purchase-book".

8.2.3 Internal policy repository

The trader policies used in the DRYAD software are defined separately for each service type. The policies are

- import action policy, that consists of import acceptance policy, federation policy, selection policy, and resource consumption policy, and
- export action policy that includes only export acceptance policy.

The acceptance policies restrict the set of potential importers or exporters per service type. The policies are expressed as evaluation rules and each request is tested against these expressions. We represent the acceptance policy as regular expressions on addresses. An address consists of a host name, a port number, a protocol name, a file name, and a process number.

The federation policy defines which other traders are requested for import and in which order. We represent the federation policy as a partially ordered list of trader names.

The selection policy includes system matching and system preference criteria for selecting offers. The criteria are represented with the same language constructs as the importer's selection criteria (see Figure 8.14). The trading system can be configured to have different roles over different service types, e.g., a consultant or a trustee (recall Section 5.2.6). The arbitration policy is realized by rule priorities incorporated in all selection criteria and policy rules. The criteria priorities are the following (in descending order): (i) the system requirements, (ii) the user requirements, (iii) the user preferences, and (iv) the system preferences.

In the DRYAD trader implementation, many policies can be changed at run time through the management interface. Only the export action policy is fixed: the offers are stored non-persistently (i.e., a new trader instance does not adopt the offers exported into the previous one) and an offer may be deleted by the trader itself if that trader cannot access the offered interface. If persistence of offers is required, there must be an additional active object that forces the trader to store the offers in a stable storage. Installation of such an object would mean a change in the trader's export action policy. The system structure is suitable for such a change.

8.2.4 Structure and contents of offer space

The offer space contains the essential information of a trading system: the exported service offers. Offers appear in two forms. One form represents a local offer from the trader's own trading domain. The other form, a proxy offer[2], is an offer that has been imported from another trading domain, and exists only temporarily.

[2]The ODP trading function standard uses the term proxy offer for an object that is used for propagating an import out from a trader. The proxy offer in DRYAD trader is a cached copy of an offer.

```
%token identifier, string, integer, floating_point_number, REQUIRE, PREFER,
   OR, AND, IN, CAT, function_identifier
restriction: clause |  restriction ';' clause;
clause: disjunction |  priority ':' disjunction;
priority: integer |  REQUIRE |  PREFER;
disjunction: conjunction | disjunction OR conjunction;
conjunction: predicate | conjunction AND predicate;
predicate: server_identifier | expression  comparison_operator expression
  | expression  IN '{' expression_list '}'
  | expression  '!' IN  '{' expression_list  '}' | '(' disjunction ')';
comparison_operator:  '<' |  '<' '=' |  '=' |  '!' '='
  |  '>' |  '>' '=' |  '~' |  '!'  '~';
expression: term |  expression addition_operator term;
addition_operator:  '+' |  '-' | CAT;
term: factor | term multiplication_operator factor;
multiplication_operator:  '*' |  '/';
factor: primary | function_identifier '(' expression_list ')' |  '(' expression ')';
primary: attribute_identifier |  string |  integer |  floating_point_number;
expression_list: expression | expression_list ',' expression;
attribute_identifier: identifier;
server_identifier: string;
```

Figure 8.14: Yacc definition of the criteria and policy rule language.

A service offer logically includes

- an identifier (includes service type and trading contract template names),
- methods for evaluating the attribute values, a method per attribute, and
- value cache, where each value is tagged with maximum time-to-live.

The service offer object is created by the offer handler object as a result of an export operation. The offer is stored until it is deleted by the corresponding withdraw operation.

The proxy offer has the same structure as the real service offer. It is created by the trader agent when it is executing an import operation. The agent requests a remote trader to immediately return a set of offers without evaluating any missing values. The values that the remote trader was able to produce immediately are transformed to the local representation form using the interceptor knowledge. The results are stored into the proxy offers as static attribute values. If any attribute values are missing, they are replaced with an evaluation rule, that imports the value from the remote trader and transforms it in to the local form. If large amounts of dynamic attributes are used in interworking, this may cause a very slow response and a high load for all involved traders. It would be possible to combine the remaining attributes to one import request. However, the current implementation relies on the observation that most requests do not use very many attributes [244].

New values for dynamic information are always evaluated on the trader's initiative. The trader has no means to collect information that is sent to it without request. However, trading could be used for example for predictive load sharing which requires statistical load information. To support that kind of information, we can create a separate information server, that automatically collects reports from system sensors and stores them [245]. The trader can then request for predictions from that information server. The trader performance would not be hindered by constant information change requests.

For the creation of service offers we aggregate one type description object and a set of policy objects. This object includes all the necessary information for the creation of service offer objects and for management of the import requests. This offer template can be partitioned to a local partition and a set of federation related partitions, one for each cooperating remote trader.

The local part of the offer template includes

- a service type description object,
- virtual values (evaluation rules for dynamic values or static values as such),
- acceptance policy as lists of allowed servers and clients,
- import action policy object, and
- the federation setting that defines whether the local trader is allowed to show the service type in question to other traders importing from it.

The federation related parts of the offer template contain

- the remote trader name and the service type name that should be used when importing from that remote trader, and
- reference to the interceptor object to be used for translations of attribute values between the local and the remote presentation forms.

8.2.5 Operational behaviour

The trading service interface offers operations for import, export and withdraw. In this section, we consider only the import and export behaviours.

Import operation behaviour

The import request contains

- service level, which defines what level of detail the importer wants to see the service offers (e.g., server identities, all attribute values, named attributes only),
- federation mode, which defines whether the importer denies, allows or forces interworking with other traders,
- federation search method, which defines whether search is done in parallel at all traders, or sequentially from trader to another until a service offer is found,
- the service type name,
- the matching criteria, the preference criteria and the importer policy rules for selecting service offers – all packed to a shared criteria field using the language described in Figure 8.14, and
- a list of attribute names, if the used service level requires so.

After receiving the request message, the agent checks whether this client can be accepted. If not, it exits with no response. Otherwise, it continues by retrieving and merging the importer's matching criteria, preference criteria and scope criteria, and the trader's selection policy.

The process of retrieving rules and merging them produces a database query sequence for the Debbie engine to execute. The query consists of steps with distinct levels of priority: requirements and a sequence of preferences. The first part of the query sequence handles all requirement level criteria and it is executed over the service offer space. If a requirement cannot be satisfied by an offer, that offer can not be visible for the importer under any circumstances. Further steps can be taken with the preference criteria and executed over the intermediate result set of offers. A step is committed only when it does not create an empty result. If an offer cannot satisfy preference criteria, it may be shown to the importer, if there is an inadequate number of services that would fulfil also this preference rule. This interpretation is also used when attribute values cannot be evaluated for some reason. A requirement cannot be satisfied if the value is not known, whereas a preference is not violated by a missing value. After the agent has traversed the query sequence, only those attributes that were used in the selection criteria have been evaluated. If the importer wished to have a single offer as result, then all but one random offer are deleted. Then the rest of the attributes required for the response are evaluated. Finally, the response message is sent to the importer, and the agent exits.

The reply message includes the name of the requested service type, and a set of items where each item consists of: (i) a joint name for a set of offered servers which together are able to produce the requested service (set size is currently always one), and (ii) a set of service offer descriptions which consist of server identity and/or address, a natural language description, and a set of attribute values, as [name, value, unit] triples. Especially interface locations and signature types are included in the attributes.

If the import request has a syntax error, a denial report is sent as response. Otherwise, if there occurs a minor failure during the evaluation of attribute values, the message is only tagged as 'qualified' and all the available information is returned to the importer. At the place of an attribute value, there may appear a notification of the evaluation failure and also an explanation of the type and location of failure.

In cases where federation occurs, the remote import operations can be done either concurrently with the local processing or after the local trader has failed to find any suitable offers. The modes are chosen by the administrative federation policy. The policy also states which other traders to consult. Depending whether the federation is concurrent or sequential, the processing may split to threads. For concurrent processing, the processing splits after request has been accepted. For sequential processing, remote imports are invoked after the requirement processing phase. All concurrent processing joins together before the preferences processing phase can be started.

Proclaim operation behaviour

Instead of an export operation, the DRYAD trader implementation uses a proclaim operation that only identifies the offered interface and its address. The reason is simple. The export operation includes virtual values, i.e, evaluation rules for properties. By the evaluation rules the exporter can force the trader object to invoke whatever operations the exporter wishes. Currently, no standard security functions are available. The DRYAD implementation does not include any security considerations except those inherited from the runtime environment. In this situation, the standard export operation would cause an easily observable way to intrude into the system.

In order to avoid this vulnerability in the DRYAD implementation, we force the trader administrator to define for each service type the operations that are allowed to be evaluated. The suitable evaluation rule is then identified by the server identity at the time of proclaim operation. As the static properties of the offers are usually similar for each offer within the service type, we used the same method for the static values also.

The application-programmer interface for trading

The DRYAD trader's application-programmer interface has two components, an application-programmer interface library for importing purposes, and a general exporter object that can be used together with arbitrary server objects and server object templates [126].

On the importer side, the trading library offers C functions for building up together an import request, sending it to a selected trader object, and breaking down the reply given by the trader. On the exporter side, the general exporter object offers exporting and withdrawing facilities under the control of trader policies. In addition to the trading operation, the interface library also allows requests about the type system used behind the trader mechanism.

The application programmer interface has been used, for example, to create a graphical browser, Artemis [113, 19]. Artemis allows users to browse service types known in the system, currently offered services, and property values of the currently offered services.

The library is also included into the trader agent itself for federation purposes. Therefore, the library allows connections to multiple traders simultaneously.

8.2.6 Administrative facilities

The trading system administration involves type, policy and trader interworking management. None of these tasks are supported by the trader object itself, but a separate graphical management tool is provided [71, 19]. The Nereid management operations interface directly with the Debbie database engine and allow interactive updates for the contract template components.

The type management operations include addition, registration, and deregistration of types, and addition of attribute definitions. Also, interceptor descriptors can be added by naming the federation contract involved and translation rule.

The policy management operations include addition and deletion of rules in named policy tables.

The trader interworking management operations include addition of name-address pairs for remote traders, and addition, registration and deregistration of federation contracts.

8.2.7 Exploitation relationship to other services

The DRYAD trader includes a simple type repository service for its internal needs.

The DRYAD trader uses a special Debbie database system for data storage. Debbie is a very compact database management system that has been developed specially to support software for distributed systems [130].

Debbie is a relational, memory based system. Besides the usual relational operations (projection, selection, join) it offers operations for partial and approximate string matching [196], and recognises basic data types (integers, reals, strings, methods) automatically. The database schema can be dynamically manipulated. Debbie supports concurrency and optimistic locking, but does not include heavy logging mechanisms. The applications of Debbie system do not require rollback facilities.

8.2.8 Performance aspects

In this section, we study the DRYAD trader performance mainly in terms of import operation response time. In addition, we compare the load caused by imports and exports, because it effects the choice of reasonable federation policies, and view examples of memory requirements.

We studied the DRYAD trader's performance by direct measurements of the DRYAD trader performance [126] and measurements of the Debbie system performance [130]. The measurement environment consisted of PCs based on 90 MHz Pentium processors running Linux version 1.2.13. The DRYAD trader or the Debbie server runs on a machine with 32 MBytes memory, and the clients had 16 Mbytes. The PCs were connected using 10base2 Ethernet. The PCs had no other load during the measurements, but a background load on the Ethernet was unavoidable.

In trader measurements, the results were obtained by using an instrumented trader client program which generated the required synthetic workloads. The workloads consisted purely of import operations. The trader's offer space contained synthetic offers that had unique identifiers and either 9 static or 9 dynamic attributes, depending which of the two available service types they presented. In most tests, the offer space size was 100 offers. Most measurement results present averages for 100 processed queries, and all measurement series were repeated at least three times.

The Debbie measurement results were obtained by using an instrumented database client program which was also modified to generate the required synthetic workloads. The workload consisted purely of read operations. The database contained synthetic data that had a uniform key distribution. Also, the key values in the synthetic queries were evenly distributed. All measurement results represent the averages for at least 2000 processed queries, and all measurement series were repeated at least three times.

Size of offer space

The database search facilities form the basis for import operation performance, and therefore we started the study of database size effects with measurements on the Debbie query times. Then we continued by measuring DRYAD trader response times with varied database sizes.

In the Debbie query time measurements, we used a database table that consisted of two columns, a key and a value, all keys being non-numeric and unique. The number of rows varied from 100 to 100 000. The query consisted of the selection of one row based on a random existing key. We used two scenarios. In the first scenario the client and the server were located on the same host and communicating through UNIX domain sockets. In the second scenario, the client and the server were located on distinct hosts and communicating over TCP/IP. The measurement results are shown in Figure 8.15.

In the single host scenario we can see the expected logarithmically increasing behaviour of the average response time. However, even there we can see a comparatively large constant factor which is due to the communication overhead. We can achieve response time of less than 0.5 ms even with rather large tables. That corresponds to a query processing rate of over 2000 queries per second. The remote communication scenario shows a rather stable behaviour in the 3 ms range. The change from logarithmic behaviour to an almost constant value is due to the dominating factor of network delay.

For the measurements of import response times with the DRYAD trader, we varied the number of exported service offers and monitored the number of Debbie commands called. Import operations invoke 50-76 Debbie commands. Therefore, we expected import operation response times to be in the range of 25-38 ms. The trader import behaviour was also expected to repeat the logarithmic behaviour, with a constant factor of overhead.

The results showed two modes, a low an a high response time, as shown in Figures 8.16 and 8.17. The difference between these response time levels was too large to be explained by

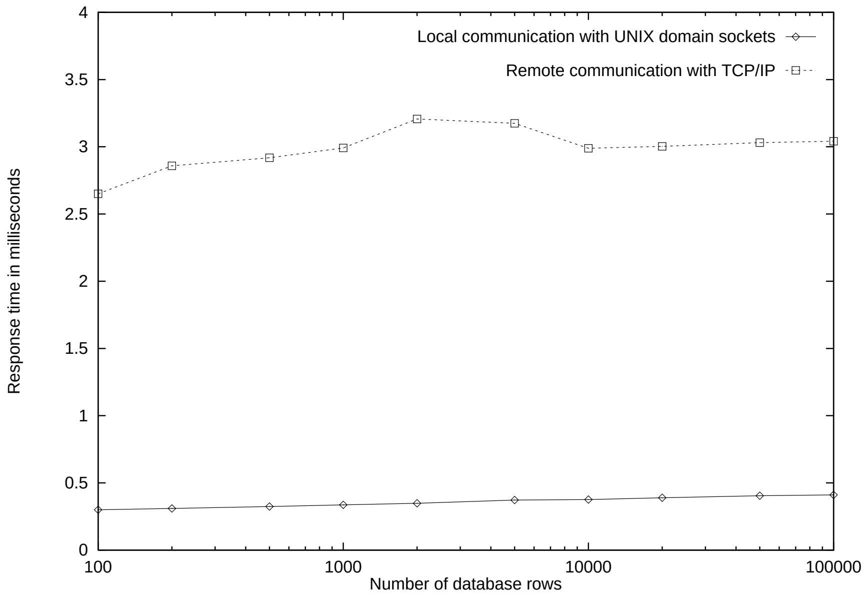


Figure 8.15: Response times related to database size.

random fluctuation, although both of the curves independently show random components. In Figure 8.16 the general level of response time stays around 30 ms. As the offers include only static properties, the variance is rather low. However, Figure 8.17 shows considerably high response times. Also the variance of the response times is high. For the importers, the variability in response times appears to be random. Technically, the difference is explained by the lazy resource reservation and indexing scheme used in the trader agent. During the first import operation after trader initialisation or export operation, a set of indexes is rebuilt. The difference between the two response time levels represents the overhead of these actions.

For a more robust service, a more aggressive rebuild of indexes should be done both at trader initialisation time and after a block of export operations. We assume however, that exports are relatively rare in comparison to the frequency of import operations, and thus the overall import performance is adequate. Still, in the test environment, an aggressive update of indexes would not have affected the performance of normal imports, because the server host had plenty of unused capacity that could be used for the background updates.

Concurrency of clients

For analysing the effect of concurrent queries to the response times, we first arranged a series of tests for the Debbie database engine where the number of clients were increased from one to ten[3].

[3]The measurements were rerun in May 1998. The hardware used was identical with the original measurements, but Linux 2.0.32 was used. The enhanced operating system performance improves the results, but does not affect the general trends.

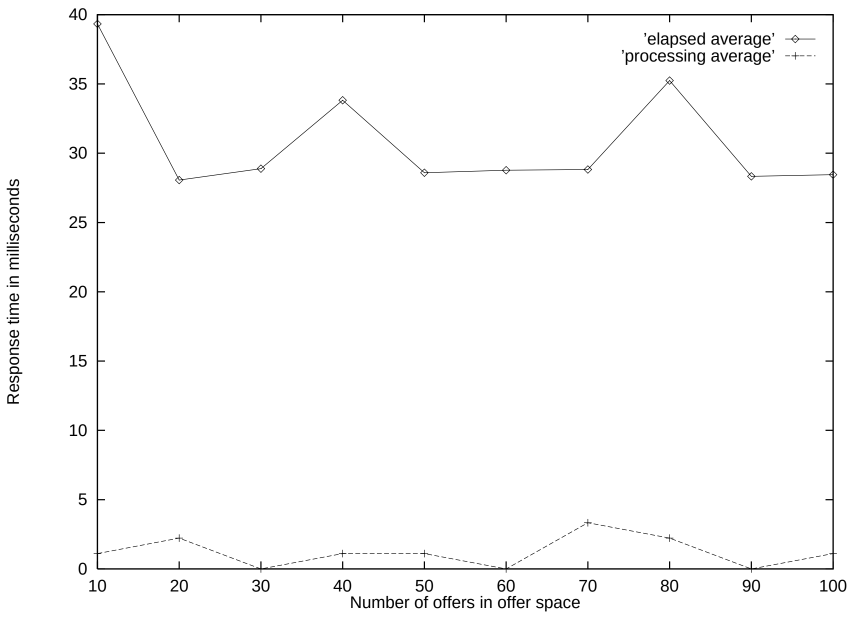


Figure 8.16: Response times as function of repository size, normal case.

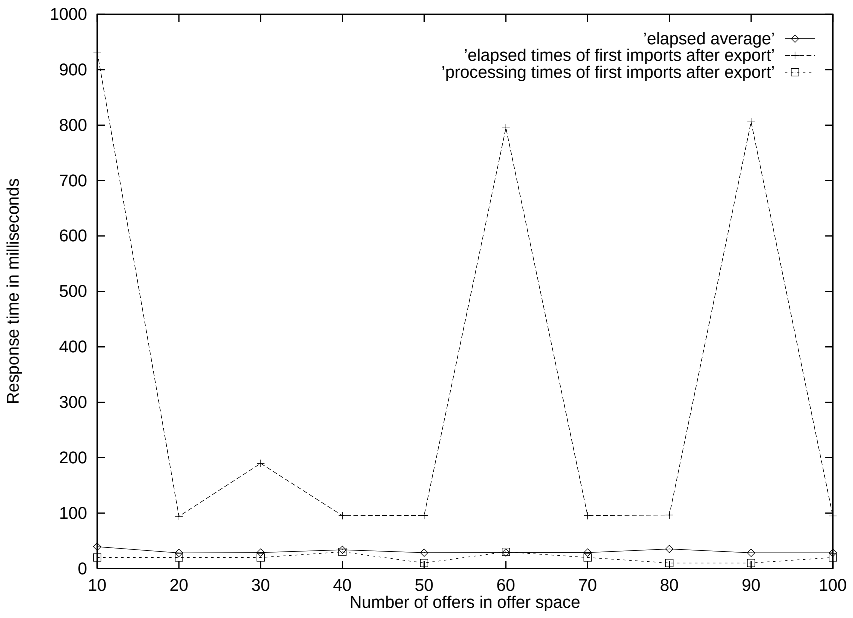


Figure 8.17: Response time in milliseconds as function of repository size, first cases after exports.

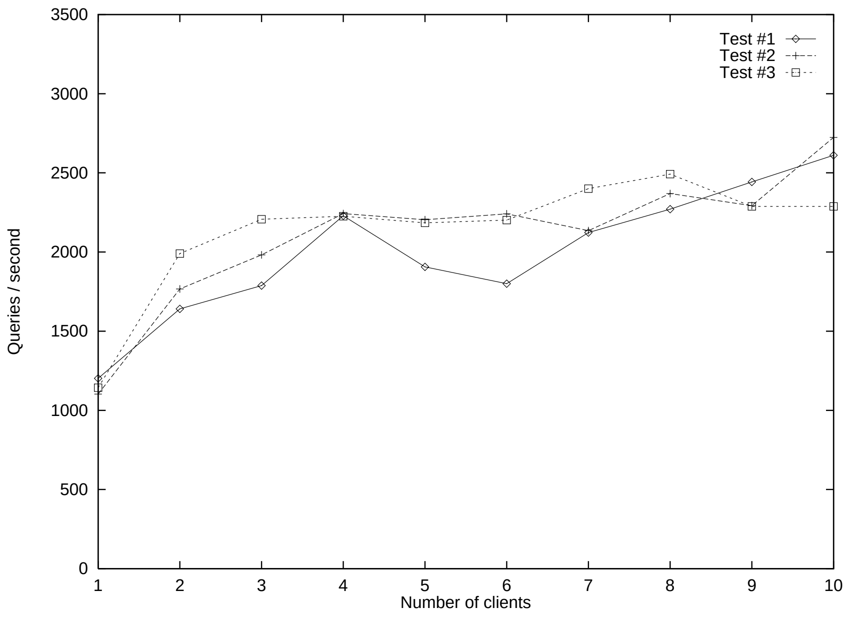


Figure 8.18: Query completion rate vs. number of concurrent clients.

In each test, the clients generated bursty queries as quickly as they could, however, the clients had to wait for the responses to previous queries before issuing a new burst of 10 queries. A burst of queries simulates the load generated by a single import request for a trader. The clients were independent of each other and all communication took place over TCP/IP.

The performance is presented as query completion rate at the server. We expect to see behaviour where increasing the number of clients, i.e. increasing the load, causes increase in the number of completed queries until a saturation level is reached. The measurements presented in Figure 8.18 confirm this hypothesis. The query completion rate approaches a limit in the range of 2500 queries (in the original measurements, in the range of 1300 queries) per second. The clients form a negative feedback loop in the system, so that the offered load does not necessarily increase linearly with the number of clients. Due to the somewhat unstable behaviour of the query completion rates, we considered in this case three individual test runs instead of averages.

As a further study of concurrency effects on the DRYAD trader itself, we varied the number of clients from one to ten and measured the trader response times. All imports were based on static properties only and no attributes were included to the responses. In order to minimise the processing requirements, a randomly selected, identified offer was requested. Each client had to wait for a reply before sending the next request, but otherwise the requests we issued as fast as possible. Figure 8.19 illustrates the measurement results.

The results are in some extent unexpected. From the measurements shown for example in Figures 8.16 and 8.23 we see that the average elapsed time of 33 milliseconds is normal. In the curve shown in Figure 8.19, measurement points with 1-5 and 7-9 clients are rather much expected: no remarkable queueing delay can be seen yet. The peaks at 6 and 10 simultaneous clients seem to

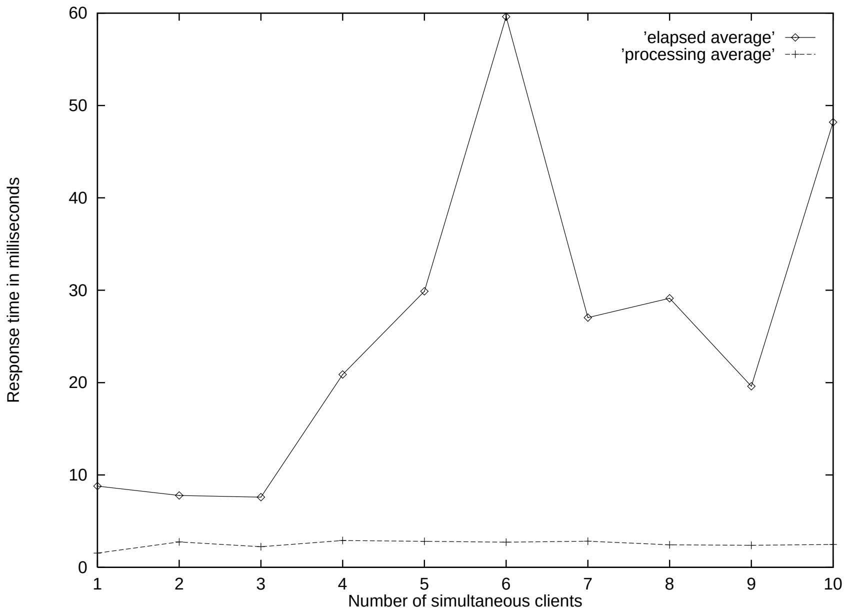


Figure 8.19: Response times as function of concurrency degree.

be caused by memory management problems. The Linux operating system has a good paged memory management system with LRU algorithm for releasing pages. As the Debbie tables ('indexes' and 'data slots') grow, the data page working set of the simultaneous clients are driven out of synchrony. In the peaks the effect of traditional LRU anomaly is experienced. Figure 8.20 shows similar effect on Debbie system only. The behaviour changes dramatically at the point where the database size exceeds the physical memory size available.

The memory management problem could be diminished or removed using techniques developed in object database systems. For example, ObjectStore system uses virtual memory addresses as object identifiers and forces data slots that are used together to the same or adjacent pages [132]. With improved memory management techniques, the peaks shown in Figure 8.19 could be lowered. The positions of the peaks are inherent to the amount of available memory in the system during the measurements.

Complexity of queries

Complexity of queries can be expressed in terms of import criteria complexity, computational complexity and response complexity. To describe the import criteria complexity, we measure the effect caused by varying the number of static attributes referenced in the import criteria. As a computational complexity measure, we use the number of dynamic attributes referenced in the import criteria. For the evaluation of attributes a syntetic evaluator was used to provide equal evaluation times of 1 second to all attributes. As response complexity measures, we use both the number of returned attributes and the number of returned offers.

Import criteria complexity effects were measured in a configuration where one client imports

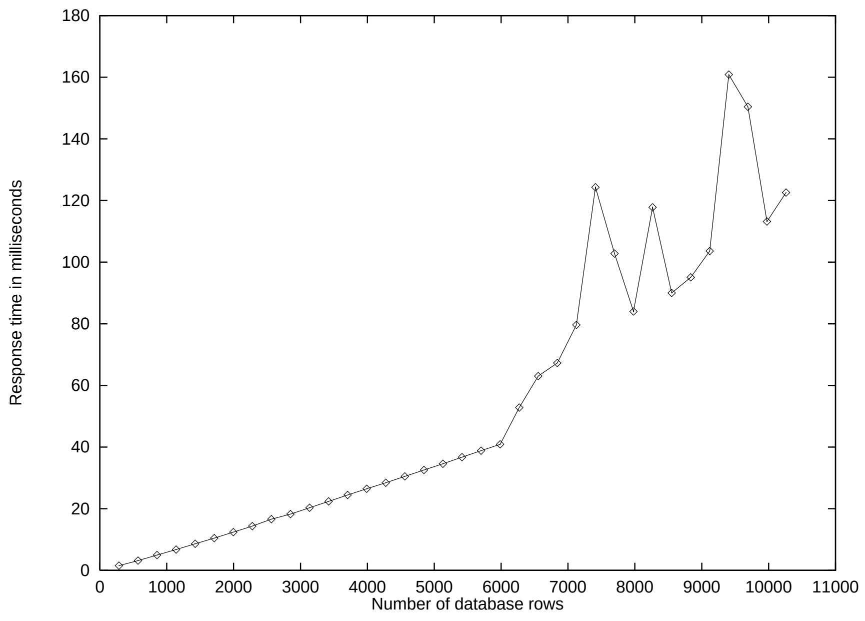


Figure 8.20: Trashing behaviour of Debbie: response time as function of database size.

a single offer. The import criteria includes a variable number of static property names. In order to avoid cache effects in the database system, also the offer identifier is used in the criteria to ensure that a different offer is returned every time. In the response, no attributes were returned.

The measurement (Figure 8.21) shows no clear increase or decrease in response time when the number of attributes in import criteria increases.

Effects of computational complexity were measured with a single client importing offers. The import criteria referred to a varying number of dynamic properties, but the returned offers contained no property values.

In Figure 8.22 there are two curves that reveal the effects of caching and reusing evaluation results. In the first case the evaluated values (100) all had the same evaluation rules (similar attributes), in the second case there were 10 different rules (separate attributes) and each rule was used for 10 values. (The semantic contents of the rules was equal, only the organisation of entries through the type management interface was different.)

The curve 'elapsed average with similar attributes' is very similar to the curve in Figure 8.21 (import complexity). The behaviour is explained by the caching and reusing scheme in Debbie. Actually, only one value was evaluated and all the other values were picked up from the cache. In this test case all 100 values were collapsed to one evaluation only.

The upper curve, called 'elapsed average with separate attributes', is produced with different offers. In these offers, the 100 attributes were differentiated in to 10 groups. Instead of letting all the evaluations collapse in to one, we force at least 10 evaluations to take place. Others were again obtained from the cache, but also cache timeouts occurred. The caching time was 20 milliseconds, and there was a maximum of 8 for simultaneous evaluations. These two things together cause the

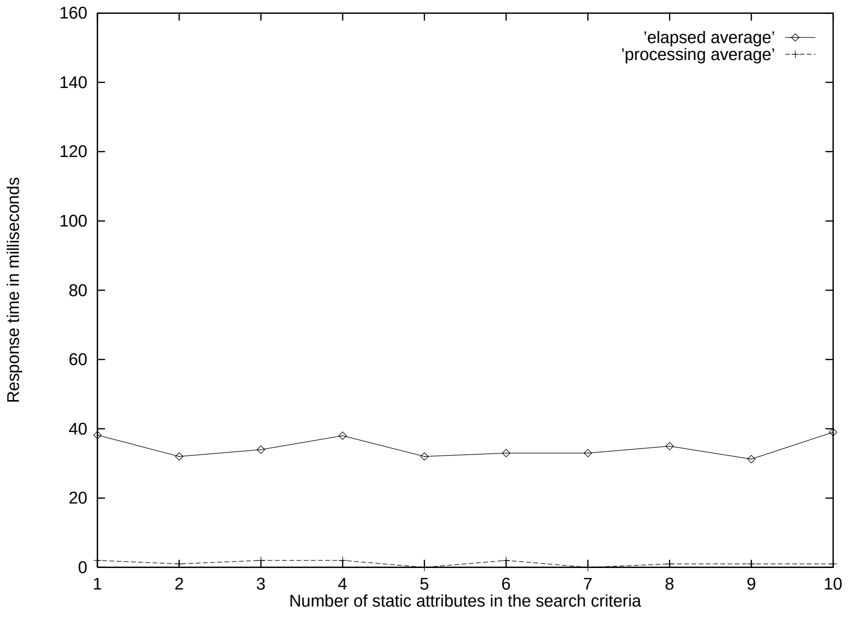


Figure 8.21: Response times as function of import complexity.

cyclic behaviour and the peaks in the curve. The simultaneously running evaluations can benefit from the evaluation results of each other through the shared cache. Also, when the cached values are invalidated, (in this case) all threads are affected by the re-evaluation.

To describe the response complexity we consider separately the amount of attributes returned and the number of offers returned. The sizes of offers were equal. Again, the configuration was a single client importing offers with static properties.

The measurement produced similar curves, as expected. Therefore, we only show the results of varying the response length, in Figure 8.23. The length of messages is normally expected to affect the communication times. However, the network delay is the dominant factor and no urgent need for minimising the length of messages between traders is visible.

Improvement techniques for dynamic property evaluation

The management of dynamic values is important in the implementation of a trader. There is no exact requirement when the virtual values have to be computed, and therefore the database system can exploit caching techniques. It can store old values and reuse them when responding to subsequent queries.

We studied the effects of caching and prefetching of dynamic property values with measurements on the Debbie database system response times in scenarios that resemble load generated by a trader.

First, we measured the average response time as the function of the number of independent dynamic values in a query. We measured the response time first without and then with a cache, using a 5, 10 and 20 second timeout. The client and server were located at the same host, in

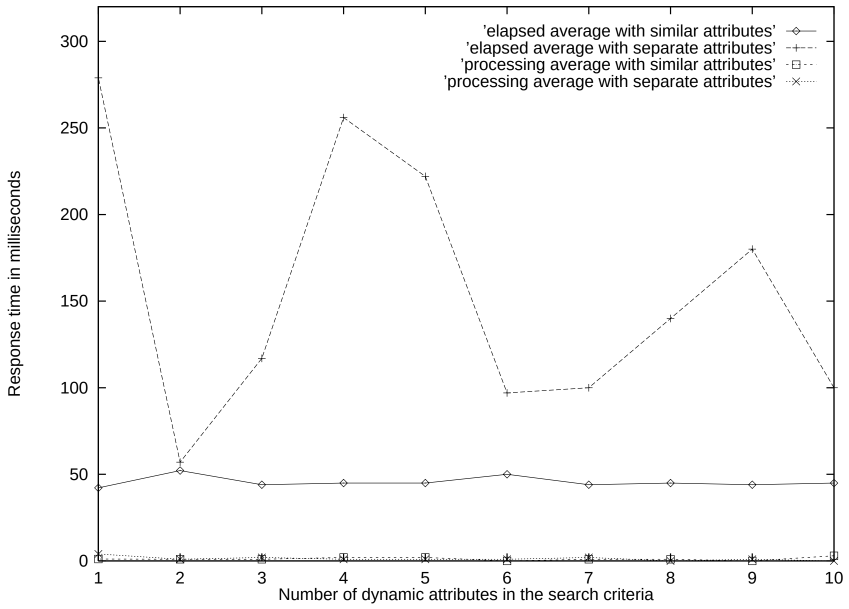


Figure 8.22: Response times as function of computation complexity.

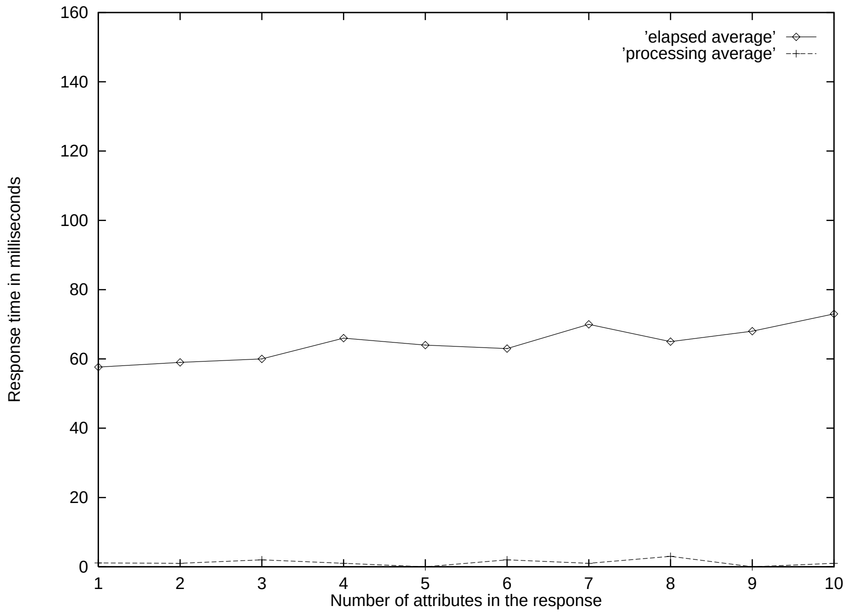


Figure 8.23: Response times as function of response length.

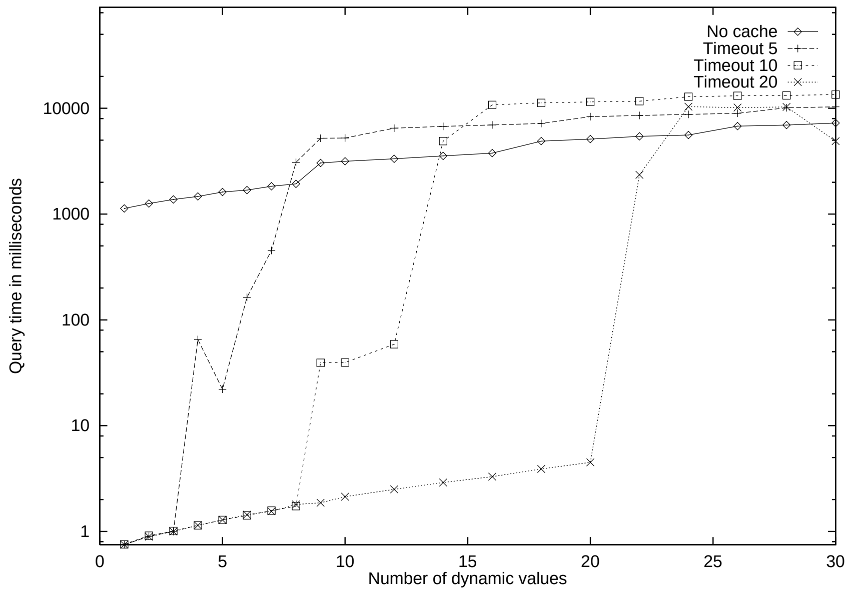


Figure 8.24: Effects on query time caused by caching dynamic virtual values.

the same way as a trader would be located at the same host with the database server. We used a synthetic external process to accomplish the evaluation of the dynamic values. The with 1 second elapsed time imitates the effect of acquiring a parameter over a network. The results are illustrated in Figure 8.24.

The results without a cache show the expected behaviour where the external evaluation time dominates. The concurrency of evaluations makes the behaviour sublinear. We can see small steps in the performance at the multiples of the chosen concurrency limit (in this case, 8 simultaneous evaluation processes).

With the cache, the majority of the requests are satisfied from the cache. If the virtual value evaluation time exceeds the caching time, the performance decreases drastically. The choice of an appropriate cache expiry timeout is of great importance. With a careful selection of this parameter for each class of data, we can reach a performance of several hundred processed queries per second.

Second, we studied the effect of prefetching. Prefetching is a well-established technique to improve performance of memory and disk caches. Prefetching means that the cache tries to predict what data is needed next and starts a fetch before the values are actually requested. Ideally, the prefetch should start so that the evaluation always finishes just in time before cache timeout. We can exploit the same technique for dynamic property values. For the Dryad trader we use the following prefetch strategy supported by Debbie: We specify a prefetch threshold that is shorter than the cache expire timeout. The first reference to the value after the threshold returns the cached values but also invokes pre-evaluation of the values in the background.

In this experiment, we used a 20 second cache timeout and 10, 15 and 18 second prefetch

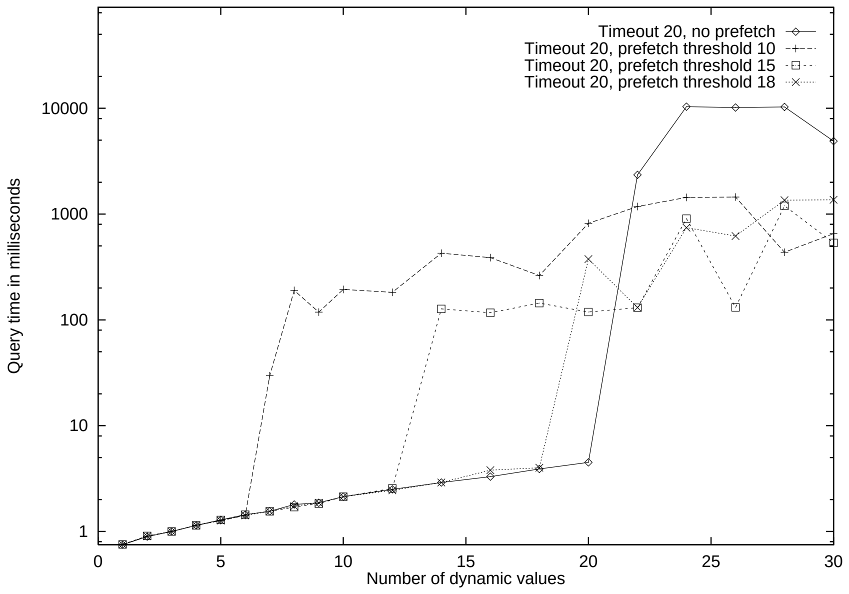


Figure 8.25: Effects on query time by prefetching dynamic virtual values.

timeouts. The results of the measurements are shown in Figure 8.25.

The measurement results show that prefetching may hamper performance if the difference of the expire timeout and the prefetching threshold is large compared to the actual evaluation time. Therefore, the prefetch should start so that the evaluation always finishes just before the cache timeout. Pre-evaluation is effective in decreasing the unavoidable loss of performance with a large number of dynamic values.

We have good reason to use the simple system for determining cache timeout values and prefetch threshold values for dynamic properties. We allow the Debbie system to use previous property value evaluation time as a prefetch threshold value, and double the time for cache timeout.

Export behaviour

In the above described measurements, we also monitored the number of Debbie commands invoked. Each import operation uses 50-76 Debbie commands, while export operations require about 230 Debbie commands. Also measurements of export response times show that an average export operation is completed in around 115 ms.

Memory requirements

Both import and export operations use private memory structures of 5000-10000 bytes, the average size being 7500 bytes. In these test cases, each offer contains about 1000 bytes of information (which is very small). In our test system this means that the maximum number of offers supported is around 30 000 offers, when we allow space for the operating system, and parallel threads of importers.

8.3 Other traders

In this section, we take a brief look at some traders that have introduced interesting features. The description here is only for pointing out some special features of these traders, not to give any deep insight in their properties.

8.3.1 ANSA

In the ANSA architecture model, trading is one of the potential ways for application objects to learn about each other's properties. ANSA trader can also be seen to extend the naming facilities of a system. The initial ANSA trader was part of the ANSAware testbench [3] running on various platforms including UNIX. The ANSA trader is often considered to be a direct predecessor of the ODP trading function.

Purpose

The ANSA model has several naming systems for object identification. Invocation name system supports remote operation invocation. An invocation name is either a local name of a memory address where some executable code resides or an interface reference [243, 183]. Attributive name system allows object selection by its properties. Finally, administration name system guarantees uniqueness of a name and exact knowledge about the object's location. The relationships between the name systems are shown in Figure 8.26. The task of the trading service is to mediate between these naming systems, i.e., the goal of ANSA trading is to give an invocation name that matches a name given in any of the other naming systems [243].

As the ANSA trader always returns invocation names, the trader is obliged to instantiate any servers that it offers as response to an import request. The trader is also responsible of creating required proxy interfaces, gateways, and interceptors so that the potentially remote client is able to contact the server through a standard interface. A successful import operation guarantees that the importer is able to communicate with the server.

Structure and contents of offer space

The trader repository of offers is organised and subdivided to contexts. Contexts are organised as a directed graph, so that each context includes a set of service offers and a set of references to other contexts. Referred contexts can be remote if that is agreed by the remote administration. Some contexts can have a special role in the cooperation of domains: any exported offers are also exported to a context in a remote trading domain.

The registration operation carries the following information:

- an interface reference,
- a description of the interface type of the offered service: the names of the operations and their parameter types and result types (note, parameter names are not included, but parameters are identified based on their location instead),

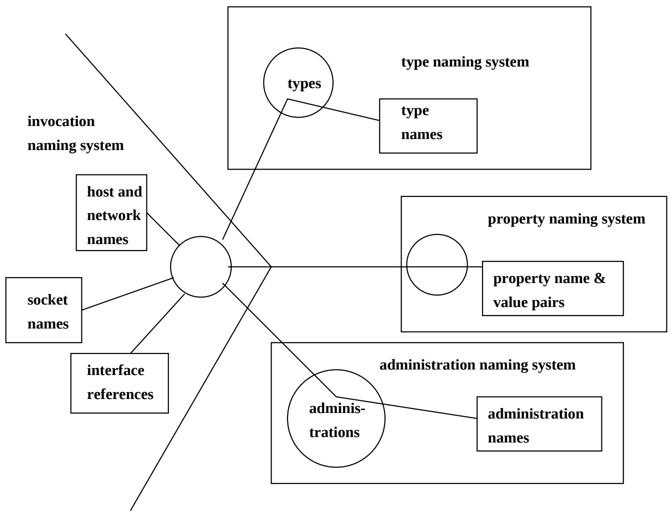


Figure 8.26: Relationships between naming systems in ANSA [243].

- information on the distinguishing properties of the offered service, such as the different protocols that are supported by the server or its infrastructure, the type of infrastructure that supports the service, the location of the service, the use of distribution transparencies, and quality of service information,
- name of the trader context in which the offer should be registered [44].

The import operation is simpler than the one discussed in Chapter 5. The arguments can only be originated from the importer, and they do not include any termination rules for the processing. The scope of the search is totally determined by the configured context relations. The import arguments include in particular

- a type description that tells the trader what operations the application expects to be able to invoke,
- constraints on the acceptable values of particular properties, and
- name of the context in which to search.

The criteria language [3] has the same expressive power as the one described in Chapter 5.

The ANSA trader model introduces a monitor concept in order to allow more efficient use of resources. There may be substantial delay between service export, service import and service invocation. During this time the service is idle, but according to the ANSA model, it must be instantiated and thus consumes resources. The role of the monitor is to delay service instantiation. Instead of exporting a service interface reference to the trader, a monitor reference is provided. When the service is being imported, the trader invokes the monitor to obtain the actual reference to the service. The monitor may create the server by demand. The monitor can also act as a trader for service templates – it receives all import operation arguments, so it can instantiate and return the most suitable server it is aware of.

Cooperation organisation

For trader cooperation the initial ANSA trader provides two mechanisms. The first mechanism is to bind together the trader contexts of two traders in such a way that for the clients, the combination looks like one trader. The other mechanism is to add into the context of one trader a proxy offer to represent the other trader.

Later ANSA trader design reports [76] lists various trading configurations, including surrogate trading and peer trading. In surrogate trading, a trading process manages a trading information base, providing a way to access the information. Surrogate trading is considered to be the likely way for most large trading systems, because trading repositories will emerge for different purposes and this design would allow flexible combination. In peer trading, no trust exists between the traders. Therefore, the traders, for example, need to authenticate all offered information. The surrogate and peer trader configurations fulfil the needs of federated trader constellations.

Exploiting relationship to other services

Type management interface allows addition and deletion of types known by the trader. Initially, only type names were registered with the trader and the type management was directly handled by the trader itself. Except the very first versions of the trader, subtype polymorphism is supported.

The user explicitly creates a directed acyclic graph (DAG) of the names of related types. At run-time, a sanity check is performed to ensure that the required operations can actually be performed on the selected subtype.

The ANSA trader has also been extended to mediate multi-media services. The Procrustean trader [141] proposes service property and type coercions and flexible subtyping to support multimedia services. The additional components, coercion management and multi-media type management, actually include the functionality of a type repository service. The concept of type coercion is adopted from type theory [158, 27].

Performance aspects

The import response time has been reported to grow linearly with the size of the trader database and with the number of clients. The time scale for import operations seem to be very similar to DRYAD trader, varying from 26-90 ms with light load and only a few service offers containing static property values [150].

The effect of evaluating dynamic property values, caching and prefetching has also been studied, assumingly with the same constellation [122]. The basic behaviour appears when the trader synchronously evaluates each dynamic value during the selection process – a linearly rising graph is given starting from around 100 ms with a minimum offer space and reaching slightly over 600 ms with a trader base of 20 service offers. The expected reduction of response times is reached by using asynchronous evaluation and multiple threads.

8.3.2 RHODOS

The RHODOS project is a distributed operating system project at the Deakin University in Australia. The goal of RHODOS trader is to support user autonomy and cooperation by allowing users themselves to export and import their own objects [65, 166].

It should be noted, that the RHODOS trader mediates objects instead of meta-information objects. Therefore, a shared object space is created among the cooperating trading domains.

Purpose and environment

The RHODOS system uses the trading service as part of the naming facilities of an operating system. It also denotes that there are two classes of sharable objects, those owned by the system and those owned by the individual users. For the users, an object is known by its attributes. Objects like files, users, group of users, ports, processors, and software based services are considered. Within the operating system the objects are known by their system names. The trading service is required to map user attribute names to corresponding system names.

One of the basic concepts in RHODOS is a domain. The smallest domain is a user domain. The users have their own domains that contain those objects they wish to retain exclusive access. They can also export those objects to other users. All system objects form a system domain, that should be accessible by all users. The domains can be arranged into a hierarchy. For instance, there can be a set of user domains; there can be a departmental domain that captures a set of objects accessible for all employees of that department but no other users; and finally, there can be a company level domain that includes objects available for all company employees. The hierarchy is used for defining the correct nodes in which service offers are held, potentially as replicas.

To support users in this environment, three different naming services are offered:

- a conventional service which maps evaluable names onto system names,
- an enquiry serve which allows the user to identify objects using imprecise descriptions, and
- a selection service which maps an imprecisely-described object onto a system name or refers to a suitable mapping service.

Cooperation organisation

In the RHODOS system, the trading service is implemented as a set of distributed traders, each trader running on a local distributed system. The cooperation between traders takes place when an import operation is forwarded to another trader because the request cannot be satisfied locally. As results of earlier imports from other traders, imported remote objects may be found already in a local search [166].

RHODOS traders exploit long-lived imports (recall Section 5.3.6), which feature is also visible in the database entries. Each trading database entry has three logical sections. The first section describes the object itself by listing attributes, access rights, owner domain, and a reference to a system name. The second section lists to which domains the object is exported and for what period of time, the cost of usage, and the withdraw conditions. The third section identifies the domains from where the object is imported. The information held is similar to exported objects.

Exploiting relationship to other services

Also technically, the RHODOS naming and trading facility are interlinked. The name server provides the conventional name related operations – such as object creation and deletion, and changing object name – thus maintaining a naming database. The trader provides facilities for making objects visible and addressable by their attributes. Import operation attaches a visible object to the importer's naming database [166].

8.3.3 MELODY

MELODY (Management Environment for Large Open Distributed sYstems) is a project at the University of Stuttgart working on the field of open service market. They develop a market infrastructure where the two main components are a trading service and a management service that consists of configuration, monitoring, and control services for the market and the marketed services [120].

The MELODY trader is supported by X.500 Directory services, and both DCE and CORBA platforms. It is implemented in C++ [26, 119].

Structure and contents

The trader consists of a trader user agent that supports importers and exporters, and a trader server agent. The trader user agents are embedded into user programs. The trader server agent supports

- the trader's type system that manages service and interface types and their relationships,
- management module for type system manipulation,
- offer storage as a hierarchy of contexts,
- operation module with export, withdraw, list and import operations, and
- cooperation module that handles forwarding of requests to other traders [26].

The service type system dictates what attributes are present in each service offer. In general, each offer must include interface identifier, location information, and list of service properties. In addition, the type can have a special function for defining ordering of values within that type (to be used when preference criteria includes the notion 'min'). Furthermore, the special function can be a reference to an intelligent agent [26].

The static properties are stored in the trader itself, but the dynamic properties are queried from MELODY management system. Because the property values are retrieved from different storages, access location objects are used. Instead of the property value itself, the service offer contains an access location object that can be instantiated to evaluate the value [118].

The offer space is organised to a context hierarchy [118]. The offers are exported to a context that represents the services of an organisational unit. Imports directed to a high level context reaches all sub-contexts, but an import started from a sub-context does not proceed to the containing contexts.

Features for trading activities

The MELODY trader is implemented according to an early ISO draft document [87] [26]. This means that the assumptions on the trading behaviour are wider than discussed in Chapters 5. For example, the trader is expected to guarantee that a selected offer describes a service that is actually available, i.e., the offered service has not failed since the offer was issued. In addition, the context model for organising the offer space was described in the draft. Later working drafts for the standard included similar cooperation establishment protocols as MELODY trader represents.

An interesting aspect of the trading activities of MELODY trader is the way of acquiring service offers. The offer space is build based on the information retrieved from

- direct export operations,
- static configuration information held in a separate configuration database,
- importing from other traders, and
- active search through configuration management services, resource discovery system or a web-worm [120].

Also the import operation parameter list shows some differences: The import can be guided by a strategy parameter ('best choice', 'cyclic choice', 'random', 'conservative choice', 'first choice' (see Section 8.3.5 for definitions)). Additional information that can be passed to the selected strategy include (i) the time limit allowed for the import, (ii) number of offers required in the response, and (iii) the maximum number of offers to be involved in the search. In the matching criteria, the importer can use predefined criteria names. The named criteria, explicit criteria, are specified by the trader administrator separately for each service type [120].

Cooperation organisation

For cooperation in a homogeneous group of traders the MELODY trader supports three modes: light-weight cooperation, simple negotiation, and federation. The light-weight cooperation mode allows a trader to act as a normal importer towards another trader. The simple negotiation mode involves long-lived imports of single offers. In this mode, the frequency of import requests is higher than with light-weight cooperation. Both light-weight and simple cooperation modes require that the type systems of the involved traders are equal. The federation model involves either heterogeneity of type systems used by the trader or long-lived imports of large amounts of offers.

The cooperation scheme of the MELODY trader supports also an automated method for determining the suitable mode of cooperation and transition mechanisms between the modes. The need for mode changes is determined by comparing measured import frequencies to given threshold values [26].

Exploiting relationship to other services

The MELODY trader is rather tightly integrated to the MELODY management system. The trader can, for instance, receive event notifications (failure reports) directly from the management system. The trader can also reserve resources for each imported service offer through the management system [118].

Administrative facilities

The administrative facilities include the management of the type space, the context hierarchy, and cooperation contracts between traders.

In addition, the administrator can guide import query processing in the trader by defining implicit matching criteria and default criteria for each service type. Implicit matching criteria are used for every import request in addition of those criteria defined by the importer. These criteria helps the administrator to guide load balancing, resource consumption, etc. Default criteria are used when the importer defines no matching criteria [120].

Performance aspects

Illustrations by Kovacs and Wirag [121] imply that the import response times for the MELODY trader are within the range of 0.2-8 seconds. The response times were improved to 0.2-4 seconds by using asynchronous evaluation for dynamic properties or by using parallel processing (3 threads for a query). The measurement with consistency predicates implies that checking property value from the management system takes an additional 0.2-4.2 seconds. In each case, the measurements show nearly a linear increase in response time when the number of offers in offer space increases from 1 to 6.

8.3.4 TRADE

The COSM (Common Object Service Market) project, at the University of Hamburg, designs and implements a generic system infrastructure for common object service market [162, 133]. The COSM project applies trading in the area of electronic commerce, where service providers such as travel agencies and newspapers offer their services to clients who wish to access information through a generic interface.

The TRADE (Trading and coordination environment) trader contains as main components the service offer manager, the service selection manager, the trader interworking manager, and the type manager. TRADE is implemented on DCE platform [162].

The TRADE type manager supports explicit subtyping [162]. The classification of services can be based on names and explicitly defined relations between these names. In addition, attributes can be attached to names for additional details about the type.

The service offer manager is based on the DCE Cell Directory and Global directory services. The service offers are stored in a special format. The offer schema includes

- the service type of the offer and the corresponding interface types,
- the current values of the static service attributes,
- the interface reference for server binding, and
- optionally, a service type description for programmers [162].

The TRADE project has participated in the IWT project (Interworking of Traders) [246]. The main goal of IWT project has been to demonstrate a trader interworking scenario between independently developed traders.

The interworking scenario was established between Brisbane and Hamburg, each location supporting a separate trading domain. Both traders were supported by DCE platform, and the domains were connected via DCE Global Directory Service. Both traders are administered by a specific, shared GUI-based trader administration tool, that is able to create cooperation links between the trading domains [161]. According to the classification given in Chapter 5, the IWT trading scenario is a distributed trading scenario.

8.3.5 Y

The GMD FOKUS in Berlin works actively as a research center and producer of distributed system services. From their work we study Y in this section and TOI traders in the following section.

The GMD FOKUS has created a distributed service environment based on broadband ISDN [66]. The project is called BERKOM (Berliner Kommunikationssystem). The environment includes a set of heterogeneous services, resources, users and organisations. As a platform for

BERKOM application projects, a system called Y is used [189]. Its services cover trading, binding, monitoring of quality of service, authentication and authorisation, and transportation services.

The Y trader is one of the early implementations. Therefore, the scope of the trading functionality is wider: The trader in this system offers to its users search, selection, and invocation of servers.

Features for trading activities

The trader can select the service offers to be returned based on a selection strategy. The following strategies have been defined [253]:

- First choice strategy accepts the first service offer found.
- Random choice strategy utilises probability measures assigned for each service offer. When a set of offers are otherwise equally good matches for the request, the offer with highest probability value is selected.
- Cyclic choice strategy keeps the equally well matching service offers in a list so that the following offer can be returned for the next similar request.
- Conservative choice strategy returns the same service offer that it returned to the previous similar request.
- Probing choice strategy first applies the search to a subset of the service offers. This strategy is designed for better import performance.
- Best choice strategy is used when the importer requires that only very up-to-date information is used for the selection process, although the selection takes more time and resources. Best choice strategy involves for example re-evaluation of dynamic property values.

The Y system introduces a role concept for traders in the process of selecting the most suitable server. The trader can assume different roles: a dispatcher, a trustee, a consultant, or a coordinator [254] (recall from Chapter 5). The importance of these roles is evident, when trading is used in a distributed system for load sharing and load balancing. The different roles of a trader are related to the different sources of matching rules given for an import. If the trader is not allowed to guide imports, the clients can use a selfish optimisation policy and thus harm the overall system performance. The roles define which arbitration rule is used between the importer and trader matching criteria.

Exploiting relationship to other services

The Y trader utilises a combination of type repository and application management repository. The repository is based on X.500 Directory and includes abstract service specifications. Each abstract service specification consists of a service name, domains on which the service is offered to, attributes of the service, and knowledge about the behaviour and invocation of the service [66].

8.3.6 TOI

TOI is a product (of GMD FOKUS) that implements the OMG trading object service [239]. OMG trading object service specification is technically equivalent with the ODP trading function standard computational specification. The differences in the specifications are due to the different scope goals and different representation requirements of the involved organisations. The OMG trading object service specifies that a trader mediates instances of services of particular types [82].

TOI version 1.0 supports the Lookup, Register, Admin and Link interfaces (see Section 8.1) in separate modules. By combining the interface modules, a trader that is conformant to query trader, simple trader, stand-alone trader or linked trader can be configured [82].

TOI exploits the services of ODP type repository function and WWW-based graphical user interface. TOI interfaces are specified in CORBA IDL and implemented on CORBA (Orbix2.0) platform. The offer space is supported by Postgres95 database management system. Implementation language is C++ [82].

During the presentation of TOI in the ICODP conference [239] Volker Tschammer explained some performance measurements: TOI running with an offer space of 10-10000 offers was giving a general response time of 200-500 ms. The tested imports used return cardinalities of 1-10 offers, and import complexity of 1-10 properties. Because of object database support for the offer space we would expect the increase of response times to be logarithmic when the offer space size grows.

8.3.7 DSTC Object Trader

DSTC (Distributed Systems Technology Centre) of Australia has been actively involved in the development of ODP trading function standard, and has also submitted a trading object service implementation to the OMG consortium.

The DSTC object trader [47] conforms to the OMG trading object service specification [173]. The DSTC trader conforms to the level of linked trader and supports dynamic properties.

The trader implementation works on Solaris, and Windows NT platforms, It uses either flat files or selected object database management systems as storage structures.

Before this commercial trader, DSTC has been involved in a series of trader implementations reflecting the various versions of the ODP trading function standard. This work has had a major role in the standard development. The group studied also cooperation contract negotiation protocol [8, 7] (see cooperation contracts in Section 5.3.6), and a rule based context structure for trader offer spaces [9]. However, these features were finally not included to the standard. In addition, the group produced most of the material [248] for the standard that prescribes how X.500 Directory service is used as a trading repository [100].

8.4 Comparison and conclusions

In the previous sections we have briefly surveyed some trader specifications and implementations. In this section, we summarise their features.

Functionality

The traders presented differ in their functionality because the overall systems into which they are embedded have different goals. We can separate two ways of exploiting traders: first, traders are exploited by open infrastructure software in the late binding process; second, human end-users exploit traders for finding objects and services they need either in computing related activities or in social-life activities. We study examples of these exploitation modes in Part IV. Table 8.1 shows the role of each trader in their system environment. The ODP, DRYAD, ANSA, RHODOS, MELODY and Y traders are clearly focusing on the infrastructure needs, while TRADE focuses on human browsing needs. Naturally, most traders are suited to both uses. The expected mode of exploitation is naturally reflected to the results of import operations. From Table 8.1 we may notice, that DRYAD trader makes an exception from the generic model by allowing template mediation in addition to instances. Mediating templates avoids unnecessary resource reservation for unused services, although service invocation is more expensive.

The traders presented are of different ages, and therefore they have their conformance goals adjusted to different versions of the ODP trading function standard. As the standard was accepted late in comparison to the development cycle of the presented traders, their functionalities and features, like operation signatures, are dissimilar. However, we can generally say that all traders have some form of query, simple, and stand-alone traders (Table 8.1). The Y trader seems not to support linked behaviour. The ANSA trader cooperates with other ANSA traders via proxy offers. Other traders use some form of links, having rather different implementations for them. For example, DRYAD and MELODY use links that are specific to a certain service type only.

There are also a variety of designs in regard to changing property values in service offers. All traders, except the two oldest, support dynamic property values. Both MELODY and TRADE have during their development changed from using modifiable properties to dynamic properties. In addition to this, Table 8.1. shows also that all presented traders support some form of subtyping. However, the subtyping concept is very different in each case as explained earlier when each trader was separately discussed. Here we will only pick up the DRYAD trader that works on an abstract service type level instead of the engineering types and therefore differs from the others.

Trader cooperation

The cooperation concepts of the traders presented are practically restricted to distributed trader constellations. Even in cases, where two separate implementations have been connected for interworking, the relationship has been static in nature and strongly based on joint administrative decisions, on selection of joint type systems and middleware platform [246, 257].

One of the obstacles for federation is that the concept of trading domain is not yet shared by the trader implementations. In Table 8.2 we have listed the views of trading domain in each case. The views of ODP, DRYAD, and ANSA are joint, and there seems to be nothing in TOI and DSTC traders to prohibit similar interpretation. TRADE trader could be included to the same group by extending its type repository concepts. However, RHODOS, MELODY and Y by nature are forced to an organisational or an administrative view to a hierarchy of domains. The hierarchy effectively restricts the potential cooperation relationships of a trader. The relationship between a trading domain, an administrative domain, and expected infrastructure services is vaguely agreed. Table 8.2 shows this by viewing the varied level of joint service expectations.

In this dissertation, we have considered the ability of importing to be sufficient for a federation relationship. However, this view is not necessarily shared by other projects. In most cases, it is required that also exports are supported, as shown in Table 8.2. An object can however be bound directly to other traders beside its local trader. Each object can be a member of more than one trading domain, unless trading is directly involved in resource management as in MELODY, RHODOS and Y systems. In ANSA architecture the relationship between management and trading has been solved by using separate naming schemes for resources and offers (see Figure 8.26).

The cooperation of traders is also hindered by the differences in expressing how the import request should traverse the trading graph. As shown in Table 8.2, only those traders, that were implemented after the ODP trading function standard was accepted, have the similar graph traversal notion. The import is propagated through a link, if (i) the importer's import policy, (ii) the trader's import policy, and (iii) the link policy suggest so. Either the importer or the trader may restrict a search to local offers only (local-only). The link may also be traversed only if no local offers were found – in this case all three policies must agree on propagation. Finally, if all three policies require it, an import can be propagated irrespective of the local search result. The propagated import is further restricted by the local policies in each trader involved, and the hop count (number of traversed links). In RHODOS and MELODY traders the trading graph traversal follows always the same strategy: links are followed only when local offers are not found. A local offer may as well be a previously imported offer. In DRYAD, the import propagation can be forced by a trader or by an importer, if the other partner does not deny propagation at the same time. In principle, propagation can be started simultaneously with the local search, and therefore, there is no counterpart for local-only mode. However, the import can be limited in time. A sequential search that has no time to traverse the trading graph further is a good approximation for a local-only mode.

Governing import activities

The trading interfaces supported by the presented traders differ. Table 8.3 captures some aspects of import operation signature and import operation behaviour.

In the import operation signature, each trader requires that the service type name is present. However, DRYAD trader uses an abstract service name instead of actual, engineering related type name. All engineering types under the same abstract type name are interchangeable, but may require an interceptor. Also matching constraint is required by all traders to specify what parameter values the importer considers necessary. Older implementations support only matching constraint, but newer implementations have also included preference criteria to do further selection.

The import operation signature can also include parameters for specifying technical restrictions. These include cooperation related restrictions, and cardinalities, as well as restrictions on using proxy offers, and dynamic or modifiable properties. MELODY and Y traders also allow the importer to specify what search strategy should be used. Available strategies are listed in Table 8.3.

The operation signature contains a variable number of criteria that are expressed in a criteria language. During the development of ODP trading function standard, a sequence of languages has been suggested. In principle, the expressive power of these languages has continuously been the same as suggested already in the ANSA trader. However, the syntactic structures have varied and it cannot be expected that two trader implementations could work together without a language interpreter in-between. The DRYAD trader suggested a priority flagged language that could be used for matching and preference criteria, and exploited for arbitration as well. The DRYAD trader uses the criteria language for exploiting the trader in different system roles. This is achieved by interleaving importer policies with trader policies based on criteria priorities.

Import operation behaviour can be governed by either importer alone, or jointly by the importer and the trader. Again, older implementations differ from the newer in this respect. The role of traders as controllers of the overall system performance was first introduced in the ANSA system. The two sources for import criteria, importer and trader, may cause conflict situations. For resolving the conflicts, an arbitration policy must be specified. The details of the arbitration policy is private to the trader, and therefore TOI and DSTC traders do not fail even if they have not published their arbitration policies. The arbitration policy does not create any additional level of nondeterminism – the import activity is already so complex and contains so many independent decision points that it appears nondeterministic to users.

Implementations and performance

Some general facts about the trader implementations have been collected to Table 8.4. The presented trader implementations are based either on an object-oriented database system, on X.500 Directory, or on a special purpose database.

The database solutions provide a normal import response time in some hundreds of milliseconds, while X.500 based solutions are considerably slower. In the database solutions, the storage technique also guarantees that the response times grow only logarithmically when the database size increases. Furthermore, the majority of the response delay is used for network communication.

The techniques used for minimising the search times within the traders and for minimising the message passing between traders include long-lived imports, caching and prefetching. Caching and prefetching are also used for dynamic property values. Probing and limited search time both decrease the search cardinality without explicit request from the importer. In both cases, a random part of the offer space is not searched. All these techniques offer good results, if same offers are used frequently enough. However, there are not yet enough practical information about actual load in any of the environments.

Feature	ODP	DRYAD	ANSA	RHODOS	MELODY	TRADE	Y	TOI	DSTC
Role in system	mediating interface references	mediating meta-information objects	mapping other names to invocation names	facilitating a shared object space together with the name server	mapping attribute names to interface references made available by the management system	mediating interface information at open service market	trader controlled, attribute based interface binding	repository of object keys	repository of object keys
Import result	references to interface instances, property values	references to (computational) interface instances or (computational) templates, property values	references to (engineering) interface instances, property values	objects	references to (engineering) interface instances	references to (engineering) interface instances, property values	references to reserved (engineering) interface instances, property values	references to (engineering) interface instances, property values	references to (engineering) interface instances, property values
Classification based on ODP trader conformance class	query, simple, standalone, linked, proxy, full-service	query, simple, standalone, linked	query, simple, standalone, proxy	query, simple, standalone, linked	query, simple, standalone, linked	query, simple, standalone, linked	query, simple, standalone	query, simple, standalone, linked	query, simple, standalone, linked
Additional policies (features to support)	dynamic properties, modifiable properties, subtyping	dynamic properties, substitutability groups subtyping	explicit subtyping	modifiable properties, subtyping	dynamic properties, modifiable properties, subtyping	dynamic properties, modifiable properties, explicit subtyping	dynamic properties, subtyping	dynamic properties, modifiable properties, subtyping	dynamic properties, modifiable properties, subtyping

Table 8.1: Trading purposes and functionality goals.

Feature	**ODP**	**DRYAD**	**ANSA**	**RHODOS**	**MELODY**	**TRADE**	**Y**	**TOI**	**DSTC**
Concepts for multiple administrative domains	<X> domains	<X> domains, usually coincident borders because of practical administrative needs	layer model of isolated domains	hierarchy of domains	hierarchy of domains	multiple traders	hierarchy of domains	multiple traders	multiple traders
Expected shared support	none on the selected abstraction level of discussion	transport protocol	name service, type repository service, transport protocol	name service, transport protocol	MELODY management system	name service, type repository service, DCE platform	BERKOM platform	ORB	ORB
Cooperation establishment	link interface	link operations	link operations for trading contexts	defined by domain hierarchy	link creation	link creation	defined by domain hierarchy	link interface	link interface
Cooperation	import, resolve, export	import, export denied by the model	import, export	import, export	simple imports, long-lived import of single offer, channel with interceptors	import	not discussed	import, export	import, export
Traversing trading graph	follow policy on import requests and links, hop count	allow / deny / force propagation on links	proxy offers	links followed when no local offers are found	links followed when no local offers are found	links	not discussed	links and follow policy (local-only, if-no-local, always)	links and follow policy (local-only, if-no-local, always)

Table 8.2: Trader cooperation mechanisms.

Feature	**ODP**	**DRYAD**	**ANSA**	**RHODOS**	**MELODY**	**TRADE**	**Y**	**TOI**	**DSTC**
Logical specification parameters	Service type name, matching constraints, preference criteria	service type group name, criteria	service type name, matching constrains	service type name, matching constrains	service type name, matching constrains	service type name, matching constraints, preference criteria	Service type name, matching constraints	service type name, matching constraints, preference criteria	service type name, matching constraints, preference criteria
Technical specification parameters	search and return cardinalities; hop count; use of modifiable and proxy offers and dynamic properties	search time, request/ deny /allow federation in parallel /sequentially	none	none	first, random, cyclic, conservative, best choice	none	first, random, cyclic, conservative,probing, best choice	search and return cardinalities; hop count; use of modifiable and proxy offers and dynamic properties	search and return cardinalities; hop count; use of modifiable and proxy offers and dynamic properties
Languages	joint for matching and preference criteria, name-value pairs for technical aspects	a single criteria language for all aspects	similar to ODP criteria language	similar to ODP criteria language	similar to ODP criteria language	similar to ODP criteria language	similar to ODP criteria language	similar to ODP criteria language	similar to ODP criteria language
Sources of specification	importer, trader	importer, trader	importer	importer	importer	importer	importer, trader	importer, trader	importer, trader
Arbitration	policy specification required	arbitration policies and selectable roles built in to the language interpretation	unnecessary	unnecessary	unnecessary	unnecessary	role selectable (consultant etc.)	not specified (not required)	not specified (not required)

Table 8.3: Governing import activities.

Feature	**ODP**	**DRYAD**	**ANSA**	**RHODOS**	**MELODY**	**TRADE**	**Y**	**TOI**	**DSTC**
Normal import response time	specification	30-300 ms	20-600 ms	not available	200 - 8000 ms	not available	not available	200-500 ms	not available
Effect of increasing size of offer space	specification	logarithmic	linear	not available	linear	not available	not available	logarithmic	not available
Effect of increasing number of importers	specification	dependent on number of threads and amount of memory	not available	not available	not available	not available	not available	dependent on number of threads and amount of memory	not available
Facilities for limiting import times	search cardinalities	importer can limit the response time directly	none	none	long-lived imports as a form of caching	none	none	search cardinalities	search cardinalities
Techniques for improving response times	not applicable	caching, prefetching	none	long-lived imports	long-lived imports, automatic cooperation modes	none	probing, caching	not specified	not specified
Implementation platform	specification	UNIX, Debbie database system	ANSAware system	RHODOS and UNIX	DCE, CORBA, X.500, Melody management system	DCE, DCE directory service	BERKOM environment with administrative data repository	CORBA (Orbix), Postgres	Solaris, Windows NT, object database

Table 8.4: Performance aspects.

Part Four

EXPLOITING TRADING SERVICES

In this part, we study exploitation scenarios of trading service. Trading is commonly considered to suit two areas, open infrastructure software that supports late binding, and open service market for end-users browsing for suitable products for purchase or for information specific to a certain topic.

Chapter 9 investigates a federation transparent interoperation between sovereign systems. Many of the techniques for federation transparent interoperation are equivalent to those required for distribution transparent interworking within a single administration domain. However, fewer assumptions about shared environment properties can be made in the federated environment. Establishment of a federation transparent communication is based on a federated binding process where trading services have an essential role. The traders mediate potential contracts and partially implement the negotiation process that results to binding liaisons.

The federated binding process supports provision of higher level communication primitives that

- *support distribution transparencies,*
- *support multi-party communication patterns,*
- *are able to mask fluctuation in communication quality of service,*
- *allow user-initiated changes in quality of service requirements,*
- *allow heterogeneous communication software and middleware services to be used, and*
- *support failure recovery.*

Chapter 10 reviews how traders can be used in the area of electronic commerce. The exploitation environment is set by a framework for electronic commerce that is formulated by combining the initial models of relevant consortia. Within this framework we identify the locations where distribution transparent communication facilities and trading services can be exploited. The study is not detailed, because there is not yet real shared understanding about where communication networks can reasonably to be exploited for electronic commerce.

Both Chapter 9 and Chapter 10 bring up contracts as an essential concept. The concepts are clearly related to each other, although the contracts have very different nature: Chapter 10 discusses commercial contracts, while Chapter 9 discusses suitable protocols for binding object interfaces.

Chapter 9

Federation transparent interoperation

In this chapter, we construct an interaction model suitable for federated systems. The model supports computational interface binding and provides federation transparent interoperation among sovereign application objects. Conceptually, the computational interface binding implements a service liaison between the communicating objects, in which the objects share an understanding on the communication structure, semantics, and means. In a federated environment, knowledge about such properties cannot be inherited, because of the isolated system development.

The system architecture is discussed in Section 9.1. Then binding liaison representation as a contract object is described in Section 9.2. The requirements for binding liaisons are studied in Section 9.3 and Section 9.4 focuses on the realisation aspects of the federated binding process, exploiting the meta-information services discussed in Part III. Section 9.5 compares the presented model to some related interoperability models. The discussion includes prototypes developed in DRYAD project. As the federated binding model is a refinement of the ODP binding framework [93, 102], we do not present a separate comparison of these models, but comment their relationship during the model presentation.

9.1 System model

In federated environments, interoperation must be established at run-time, as part of the interface binding process.

Broker architecture

The federated architecture is a broker architecture, where server selection is an internal matter of the infrastructure (Figure 9.1). This approach is usually avoided in ODP related work, because insufficient trust in other administrative domains in the global information processing network. Other architectures, like ANSA, exploit a matchmaker model, where the client first uses server selection tools (trading) and identifies the selected server in service invocation. (The terminology of broker and matchmaker model is adopted from [219].) The unwillingness for adopting a broker architecture is understandable in systems based on networks like the Internet, where no requirements on trustworthiness of the service vendors are enforced. However, we ideally expect a more sophisticated network to evolve, where communication is not only concerned with technical connectivity, but is governed also with laws and commercial contracts.

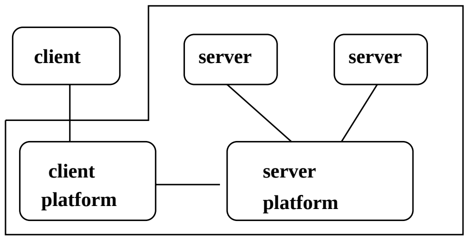


Figure 9.1: Broker architecture.

In the federated architecture, the matchmaker model is not applicable, because the initiating object has no direct control over the objects of remote sovereign systems. Furthermore, because there are no shared resources (like data files or equipment) there is no need to identify objects. Instead, shared identifiers can be attached to service (behaviour) and information contents. The aspiration for interoperation of objects in a client-service model, instead of client-server model, was already proposed by Cygnus architecture [30].

In this kind of environment, designs that integrate trading and system management are popular (e.g., [127, 114, 121, 191, 192, 240]). However, in this dissertation, we do not address that area.

Client-service architecture

The federated architecture is also a client-service architecture, where abstract service types can be used as the basis for the communication between objects. A client may request from its environment a named service, a behaviour pattern, instead of addressing a server object to perform an operation. The communication requests within one domain can use client-server style addressing as well.

In an ODP-based system, instances of objects can appear in the system in two ways, either through instantiation process or through introduction. For binding purposes, the origin of the object is not interesting. Figure 9.2 illustrates the situation. Only the object type is interesting. The objects may however differ in their properties. Therefore, the object that is selected or instantiated must be further selected based on criteria that specify the service type and also QoS.

For binding purposes, the time of instance creation not is interesting either. Therefore, a guarantee of future behaviour of correct type is as good as an already instantiated object. Therefore, a lazy instantiation process can be used: the object is instantiated from a template only when the service request is processed, instead at the time when the service is first offered.

The object configurations that participate in the cooperative behaviour can be decided independently at each sovereign system.

Binding process overview

The result of the federated binding process is a binding liaison, that ensures that a communication channel can be created between the federating objects. The information object that captures the contents of a binding liaison is called a 'binding contract' [102]. The binding contract is copied to

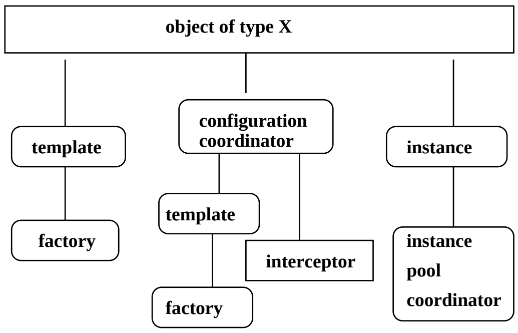


Figure 9.2: Various sources for object instances.

each involved system, in order to allow instantiation of separate sections of the shared channel (recall Section 2.4.6).

In order to understand the phases of the federated binding process, we need to consider the different views to the federated system: The architecture is essentially divided into two layers, application layer and infrastructure layer. For an application programmer, only the application interface aspects should be visible. The infrastructure layer services enhance these aspects to a full engineering view.

At the application layer, the federated binding process can be seen as follows (see Figure 9.3). An object is composed of a set of interface objects and private state information. The object can be asked to change the encapsulated information by sending it a signal that matches an interface signature. The signature and behaviour associated to an interface are described by object property values. The property values can be stored within the object itself and retrieved for example by property service [176]. In our federated system model we store the property values in a trader, as service offers. Each service offer is structured according to an abstract service type definition, that specifies a contract schema. When the binding negotiator in the infrastructure has received a service request, it requests the trading service for matching service offers. When a suitable set of offers is found, a binding liaison is formed. The liaison establishment phases are further studied in Section 9.4. As a result of a successful binding process, the application object can be expected to have a communication channel to its peers.

At the infrastructure layer, the federated binding process is more complicated, as shown in Figure 9.4. The negotiation task is split to two parts, by separating service liaisons and infrastructure liaisons (recall Figure 2.1 in Section 2.4). A service liaison captures the application interface involved and an infrastructure liaison captures the facilities of the platforms. Consequently, a binding liaison between application objects consists of two parts, the application specific part, and the infrastructure specific part. The first guarantees application interoperability, the latter guarantees interworking properties of the platform.

Federations between sovereign systems do not form a single static network of nodes. Instead, the systems are able to join various communities at run-time, depending on their needs. Separate federations emerge for different types of application services. The scope of the federation network is restricted by the interworking capabilities of the platform services. The relationship between the service liaisons and infrastructure liaisons is separately determined at each sovereign system.

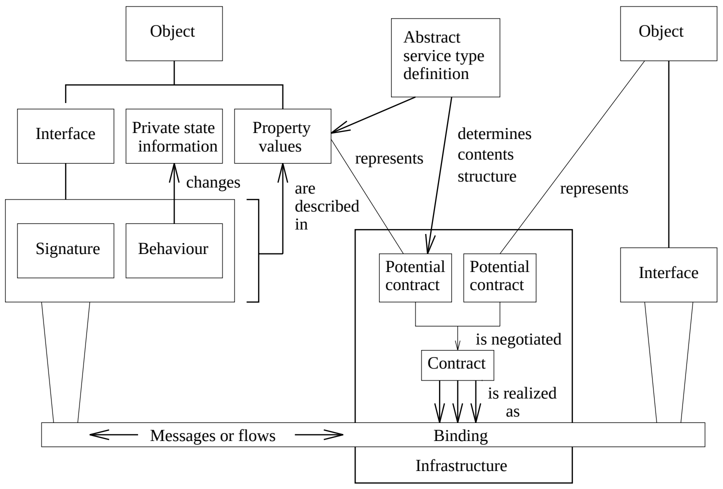


Figure 9.3: Object structure, interface references and binding.

The two-layer model of federated systems gives us reason to define two levels of interface references: a computational interface reference that only carries application related information, and an engineering interface reference that expresses the full binding contract with all platform details. A computational interface reference is indirect in the sense that it refers to an interface template and an interface factory, together with some limitations for the interface properties. An engineering interface reference refers an existing interface and denotes the actual properties of that interface.

The computational interface reference captures the information that is determined by the application objects; the additional information captured by the engineering interface reference is inherent to the platform environment that supports the application object.

9.2 Binding liaison representation

The binding liaison can technically have two states: a binding state, that is a state where no communication resources are reserved, and a channel state, that is a state where resources are in use.

The binding liaison is implemented by a channel controller, that includes the binding contract object, and depending on the liaison state, potentially also some of the channel components. The channel controller is able to modify the binding liaison between binding and channel states. The state is not necessarily always consistent, because of its distributed nature. Even the channel controller is a federated object.

The binding contract information is replicated to each of the object interfaces involved in the liaison. Because the interfaces can reside at separate domains, the data representing the contract information may have different local formats and coding. Each object can use the local copy of

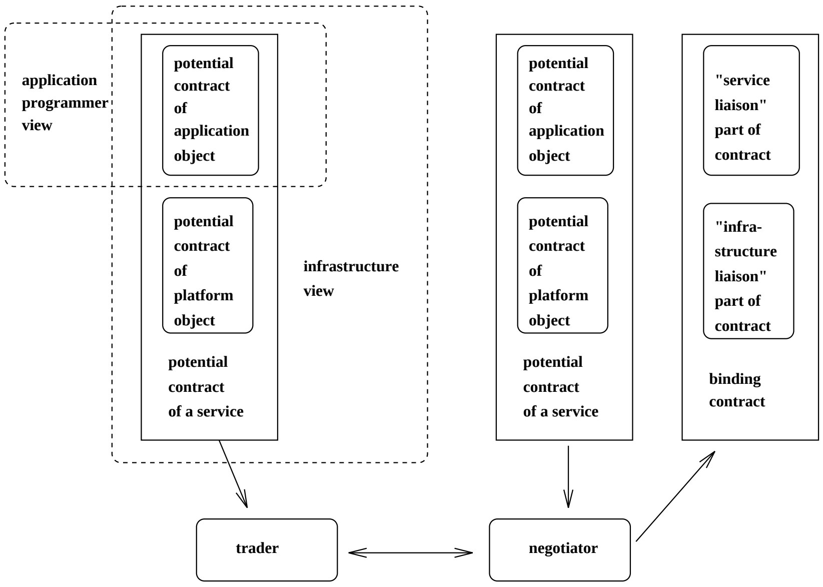

application programmer view
potential contract of application object
potential contract of platform object
potential contract of a service
infrastructure view
potential contract of application object
potential contract of platform object
potential contract of a service
"service liaison" part of contract
"infra-structure liaison" part of contract
binding contract
trader
negotiator

Figure 9.4: Infrastructure view to the federated binding process.

the binding contract as policy information or parameters to its internal activities. This mechanism can be used to provisioning of the binding contract as part of the object behaviour (see [151] for an example).

The binding contract supports the following information and interfaces:

- abstract service type identifier, that allows each administrative domain to access the related parts of the local type system,
- collectively selected behaviour as a behaviour pattern name and choices made from the associated policy framework,
- technical descriptions for interface signatures, which can be differently selected at each administrative domain,
- communication protocol,
- channel class that expresses the type of channel to be instantiated, allowing each administrative domain to access local details for stub, binder and interceptor objects,
- service type specific QoS agreement,
- failure detection and recovery protocols, failure defined as not being able to meet the QoS agreement,
- remuneration protocol, and
- channel control interface reference separately for each administrative domain.

9.3 Requirements for channels

In order to specify further details of the binding contract contents, we need to analyse the requirements for channel structure. In the following, we study federation transparency, a set of distribution transparencies and quality of service management. The dynamic nature of the liaisons causes additional requirements for channel structures.

9.3.1 Federation transparency

The requirements for a channel in a federated environment include support for federation transparency that hides heterogeneity and asynchrony of system evolution caused by sovereign systems. This involves management of service and interface types, selection of servers, translation of message contents, etc. However, the channels in federated environments contain the same components as expected in traditional distributed systems, i.e. the selective transparencies identified in RM-ODP. Naturally, achieving these transparencies has become more difficult, and even in some cases restricted by federation transparencies. Furthermore, federation transparent interoperation must be applicable to all modern communication needs, for instance, multi-media applications and mobile systems. An interesting aspect for these areas is quality of service management.

Federation transparency differs from the transparencies identified in RM-ODP because it is not selective. Instead of parametrising communication operations for selected transparency services, a more primitive set of communication operations must be used if federation transparency is not exploited. We have used the term interworking for the communication primitives where the cooperation partners themselves manage problems caused by evolution and heterogeneity. Federation transparency masks the problems caused by sovereign administrations from the communicating partners.

9.3.2 Selective transparencies

The general interoperability requirements for distribution transparent communication include

- support for server replication (replication transparency),
- support for multiple representation syntaxes for data (component for access transparency),
- support for object migration (migration and relocation transparencies),
- support for failure recovery (failure and persistence transparencies),
- support for multiple transport protocols (component for access transparency),
- support for multiple platform architectures (component for access transparency),
- support for other distribution transparencies, like transaction transparency,
- secure communication, and
- multi-point communication.

Interoperability requires a group of simultaneous interworking liaisons formed by the channel components. The interface references must support the information requirements for these interworking liaisons. The interworking issues that reflect the above list of distribution transparent communication include: functionality representation and information transfer, migration, group communication, security, quality of service predictability, and failure recovery.

Functionality representation and information transfer are successful only if the communicating objects are able to interpret the same description language. Because the communicating partners cannot trust in inherited knowledge about these things, they need an explicit representation for the negotiation and conformance checking. The task is rather simple, if only one representation language is allowed. For example, IDLs are commonly used for this purpose. Marshalling stubs can be directly generated from the information structuring rules for several platforms [63].

Migration of objects generally invalidates interface location information. Several migration protocols are available. For instance, the migrated object may leave a proxy interface behind to relay any messages, or the migrated object may trust all senders to relocate it via a re-locator service when communication otherwise fails [48]. The method used by an interface must be recorded to interface references pointing to it, so that the user of the interface reference can use the same protocol. Migration of objects may also change the object's communication domain. This means that the protocols in use and the address format will change. Of course, the quality of service information must also change, because for example communication delays are relative to distances and protocols.

Group communication services require that the membership in the liaison is dynamic, that the members may join and leave at will. Groups have several ways of organising themselves, depending on their purpose. For instance redundancy groups, backup groups and cooperation groups have different protocols for communication. The multi-point communication services have several alternative methods for distributing the end-point information - ranging from notifications to polling. In some cases it is essential that only one of the objects receives messages, while in some other cases the communication fails if not everybody receives all the messages in the same order. The failure recovery protocols differ between group organisations [36].

For secure communication we must choose between several alternative protocols. Most security protocols are available at transport level [55, 5]. Application level authentication can be supported by Kerberos style solutions [165].

Quality of service predictability and timeliness of service are fundamental to modern applications. There are several monitoring protocols for communication services and also several recovery protocols for situations where the service fails to reach its promises. Most of failure recovery techniques are defined in multi-media application environment. Common choices are to neglect the failure and continue, terminate the application or to cut down some of the communication lines for releasing the resources for other, more important communication lines within the same application [38].

Failure detection and recovery protocols are often dependent on the transport protocol in use. However, many choices can be made at application level. For interoperability, it is essential that each communicating partner knows the recovery mechanism. Different methods have been studied, for instance in [36].

9.3.3 Quality of service management

In modern applications, like multi-media services, the capability of controlling quality of service aspects is essential. The capability has two modes: static and dynamic [20]. In the static mode, the service invocation is parametrised with QoS requirements, in order to derive the required resources for the service. In the dynamic mode, the QoS requirements can be modified during the service execution. For example, in a video-phone application, the rate of picture frames can be modified depending on the available bandwidth. Overload at the network connection affects all applications that require bandwidth, unless a reservation protocol is exploited.

Current work on quality of service concepts for ODP reference model [103] aspires definitions of more powerful failure and recovery models for QoS aspects. The model defines a QoS negotiation method that uses conditional expressions. For example, a quality of service level is reached on the condition that the server receives acceptable level of service from other components. In such a case, the interpretation of the QoS offer requires access to some system status information, which cannot be available across sovereign system boundaries.

In federated environment, the expressions used for QoS attributes in the liaison negotiation phase must be self-explaining. For example, an expression that a service is available at a given probability is acceptable. The system status can be monitored and captured in suitable, probabilistic statements in the QoS offers.

9.3.4 Dynamicity of channels

A binding liaison is implemented by a channel controller, that also supports a management interface, that allows changes to the liaison during its life-time. Additional requirements for modifications in the channel structures are caused by failures of channel components. Even the traffic load on the channel can cause changes: the channel resources can be released for periods with no traffic.

With careful design, applications can be prepared for changes in their liaisons. Liaisons are not necessarily affected by changes in group membership and migration within agreed communication domains. Members that join or leave the liaison are allowed to do so as long the group organisation is not violated and the new members are able to meet the requirements of the adopted liaison agreements. Furthermore, we have already defined, that liaisons have two modes: a plain liaison without resources and an established channel. Changes between these two modes can occur without having any effect on the liaison itself. The mode change is activated by the channel section automatically, based on a guiding policy. Re-instantiation of a channel can also be a consequence of transport failure recovery or violation of quality of service agreements.

Liaison can also be negotiated to cover, for instance, several alternative transport protocols. If a protocol change is later required, any agreed protocol is available without liaison re-negotiation, and can be done on channel reconfiguration. At application level, this kind of design would be visible as different communication modes, for example, as fixed network LAN communication mode and mobile network communication mode.

Re-negotiation of a liaison allow the original liaison to stay in effect while potential for a new, replacing liaison is tested. The new liaison is established, if the requirements for transparency support, communication domain, quality of service, timeliness, or communication domain should be changed. Such a need may arise, for example, in mobile systems (migration between communication domains and dramatic change of transport delays) or in multi-media applications (transport load varies and non-availability of resources causes violations for quality of service promises). If the re-negotiation is successful, the interface references are replaced with new ones and the channels are re-configured. If the re-negotiation fails, the interface references must stay as they were before the re-negotiation. It is up to the initiator of the re-negotiation if it wants to terminate the liaison or to continue it as it is.

To support these changes in channel structure and interface references as well as liaison changes the channel sections must support services like

- query of liaison properties (items in interface references),
- initiation of the liaison re-negotiation,
- initiation of the channel instantiation,
- destruction of the channel,
- initiation of the channel reconfiguration, i.e., destroyal of the old channel and initiation of new instantiation,
- monitoring the channel status and notifications of failures,
- monitoring of transport integrity or security and notifications of violations,

- re-evaluation of the peer location, and
- joining or deleting a member of the liaison.

These services should be offered at the generic service-management interface of all federable systems.

9.4 Federated binding process

In this section, we can finally study the federated binding process from the middleware point of view. Especially, we review the sources of information during the binding liaison negotiation.

A binding contract identifies three levels of types: identifiers for abstract service type, descriptions for interface signatures and behaviour names, and channel class name (references to template and configuration instructions). These information items are collected from

- service invocation request,
- federated type repository,
- application service offers via federated trading, and
- platform service offers.

9.4.1 Creation of a potential contract

The negotiation process is invoked when an object initiates an explicit binding action or an operation invocation. In the programmer view, the initiating request contains

- expression of the client view to the exploited interface,
- potential set of behaviours, expressed within a standardised policy framework,
- QoS requirements for the platform, such as selected transparencies (transparencies for access and location, relocation, replication and failure, persistence, migration, transaction) and timeliness,
- service type specific QoS requirements for the server, such as presentation of video stream in colours instead of black and white, expressed as a set of acceptable values, and
- failure detection and recovery protocols, failure defined as not being able to meet the QoS agreement.

This request forms a stem for the application level part of a service request.

In the programming environment, the above interaction parameters may partially be predefined by the programming tools. For example, inheritance of binding properties is reasonable within a programming language environment. At the same time, the liaison is restricted by the facilities available in the supporting platform.

The service request stem is supplemented with an abstract service type name. This can be retrieved from the local type repository. The abstract service type is required for the federated trading action that follows. In addition, the potential contract captures the platform service offer.

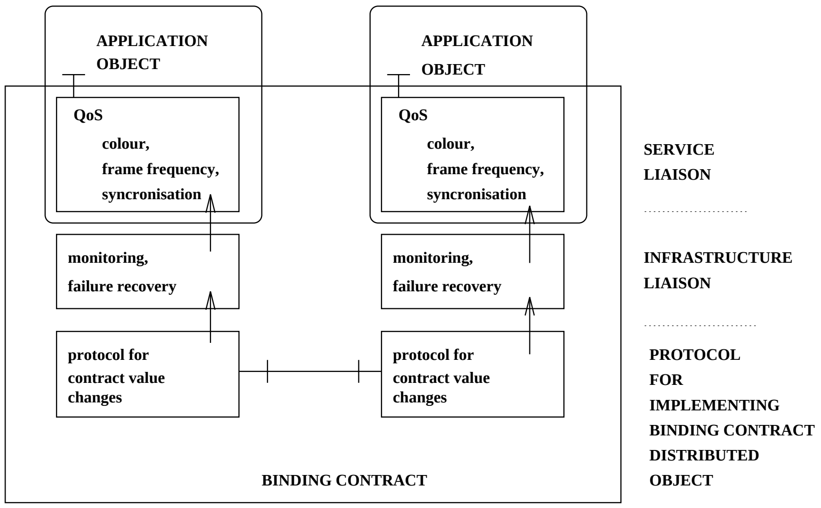

APPLICATION OBJECT
APPLICATION OBJECT
QoS
colour,
frame frequency,
syncronisation
QoS
colour,
frame frequency,
syncronisation
SERVICE LIAISON
monitoring, failure recovery
monitoring, failure recovery
INFRASTRUCTURE LIAISON
protocol for contract value changes
protocol for contract value changes
PROTOCOL FOR IMPLEMENTING BINDING CONTRACT DISTRIBUTED OBJECT
BINDING CONTRACT

Figure 9.5: Example of federating a binding contract.

9.4.2 Contract negotiation

The second step is the search for suitable servers. This is performed by traders. Each component of the potential contract of the client can be mapped onto a service offer property.

Each reachable trader that supports the requested abstract service may be queried. In the communication between traders, the property values and import criteria are transformed by interceptors when needed. The interceptor information can be retrieved from the local type repository.

When the selection is done, the negotiation service creates a binding contract object that includes such values that are acceptable for all communicating partners. All potential contracts involved – that of the client and those represented by the selected service offers – include a range of acceptable property values. The binding contract is structured according to the service offer structure (equals with contract schema) and the property values are inserted. For each value, a range is selected so that the values are present in the potential contracts of client and server.

9.4.3 Contract establishment

Finally, the binding contract information is replicated for each of the applications in the federation. Because the interfaces can reside at separate domains, the replicas may have different data representation formats and coding. The conversions required for the binding contract items are supported by type repositories. However, the transformation is performed by the binding factories receiving the copies.

Figure 9.5 shows with an example, how a binding contract can be supported in a federated environment. The binding contract structure includes first the service liaison and the infrastructure liaison related aspects, then agreements related to the maintenance of the binding contract itself.

Also internal event may cause channel management actions to occur. For example, if a breach of the binding liaison occurs, the channel may be reconfigured. It should be noticed that the breach may be related either to the federated application or to the virtual platform supporting the application federation. In the latter case, the application federation can be retained and reconfigure only the lower service layer. This means that the channel type is retained but the instances for it are replaced.

9.4.4 Binding factories

Each domain has a separate factory for channel sections. Each of these factories receives a copy of the binding contract and tailors it, for example, by including a channel template.

The binding factories exploit type repositories at various abstraction levels. The concepts exploited are binding types, channel types and channel templates (Figure 9.6):

- Binding types are used when negotiating the abstract purpose of the channel. Binding type expresses the rules for the liaison in regard to interface types and roles of the communicating objects, and the interface type of the binding control interface.
- The factories retrieve channel type definitions, when they have to interpret, what is the expected channel functionality (distribution transparency and QoS requirements, security support) and required behaviour in case of channel failures. Channel types represent the type descriptions available at each type domain.
- Channel templates specify the required configuration of stubs, binders, protocol objects, and interceptors for the instantiation of a channel section. The required configuration may be different on each platform or on each channel administration domain.

The terms of binding type, channel type and channel template are adopted from the binding framework of the ODP reference model [102].

Finally, based on the channel template information, each binding factory separately instantiates a section of communication channel defined to locate at their domain [127, 102]. For local bindings, separate channel factories can be optimised.

A channel section can be constructed, for example, as illustrated in Figure 9.7. The actual channel structure varies depending on which distribution transparencies are selected by the programmer, and which communication protocols are in use. The actual channel structure varies also depending on the platform architecture and administrative rules on the systems that support the partial channel. Also, several stubs can be active concurrently, using the same protocol link. For instance, the stubs can participate in protocols for group management, or QoS management. In a general case, the stubs are not self-sufficient, but require services from management functions like authentication services.

9.5 Discussion on some binding models

In this section, we discuss some related binding models. We have chosen to study binding in the ANSA models, because the model has affected the ODP binding model, and distributed binding factories in Sumo, because they deal with stream interfaces. In addition, we briefly discuss the operation invocation model with which the DRYAD project experimented.

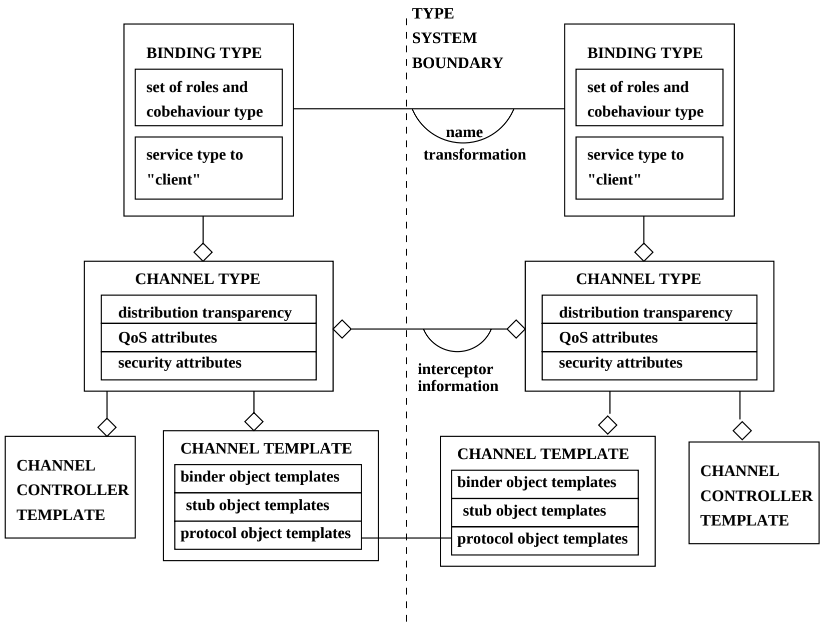


Figure 9.6: View to a federated type system.

The federated binding model presented above differs from the related models in one essential feature: it introduces a negotiation step to a point where other models trust in shared architectural assumptions. As a consequence, the binding types exploited also have to manage the more complicated contract management scheme.

9.5.1 ANSA model for cross-domain interactions

In the ANSA model (recall Section 4.3), domains are related to organisational boundaries. Therefore, managing inter-domain interactions is focused interaction management in regard to security, authentication and tractability, instead of the existence and location of suitable communication facilities [77].

Domains are autonomous in regard to control of interaction, allocation of their resource, and selection of platform architecture. In this respect, the ANSA model and the above federation model are equal.

The difference of the models is in the liaison negotiation process. In the ANSA model, the domain authorities negotiate shared policies, that are enforced on domain administrators separately at each domain. The shared properties are further inherited by the service provision and service consumption relationships. The situation is illustrated in Figure 9.8. In the above described federation model, the liaison negotiation takes place only within the service liaison.

In the ANSA model, inter-domain interactions are implemented using gateways that ensure that the communication is managed according to the contract agreed among the domain authorities.

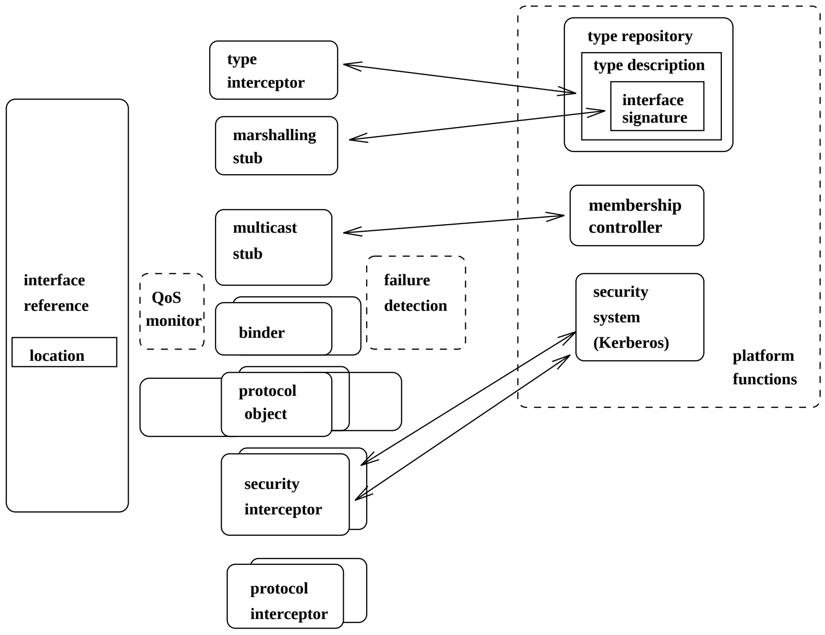


Figure 9.7: Channel section.

To achieve such a configuration, the trader gateways must detect interface references to pass through and insert gateways in the invocation path of the potential link before or when it is actually used. In order to manage interaction between domains, gateways have access to the policies that constraint acceptable interactions. Gateways also have the ability to infer message types and recipients.

The design suggests, that gateways should be located at object stubs. For performance reasons, it would be preferable to use generic gateways, or at least to colocate several gateways to the same stub object. As an implementation tool for gateway stubs, CORBA DII (Dynamic Invocation Interface) is mentioned.

9.5.2 Sumo model for stream interface bindings

Sumo project (collaborated by CNET, France Telecom and Lancaster University) has two implementations (Sumo-CORE and Sumo-ORB) for a microkernel-based, ODP compliant platform supporting distributed multi-media applications [20]. The major concern in the system model is on stream interfaces, stream bindings and flow protocols.

The communication system design is based on the engineering model of RM-ODP (recall Chapter 3). The corresponding computational view shows binding object as an intermediate object between two or more computational objects. This modelling technique is recursively repeated between the binding object components: there are no separate engineering level and no separate protocol objects. This differs from our model where binding object is only a property of the computational binding, and a set of object configurations at the engineering view.

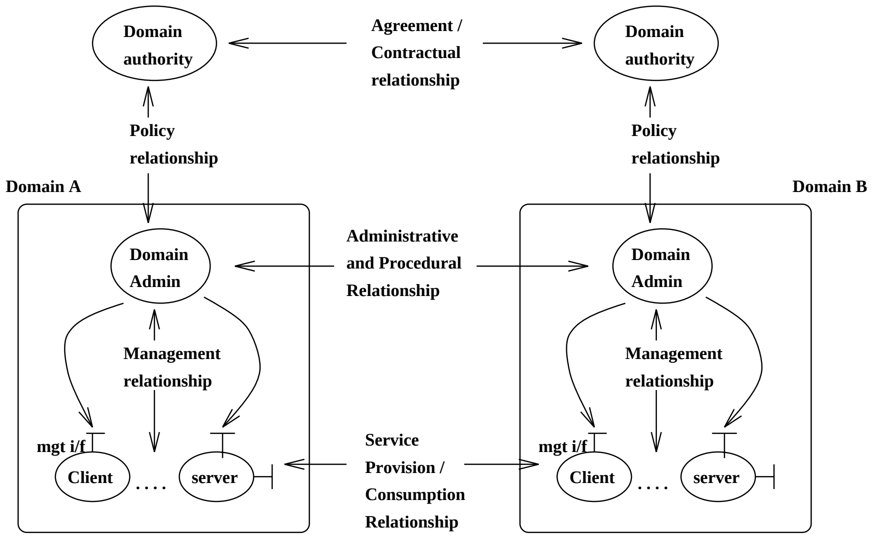


Figure 9.8: Relationships between domains and their agents in the ANSA model [77].

System model

The Sumo system exploits a matchmaking model: the communication partners are identified and located before the binding process is started. The binding process is parametrised with the interface references of the objects to be bound. Because there is no separation between computational and engineering aspects of the binding, there is also only one kind of interface references, i.e., only the engineering interface references are present.

The binding object is an active object. This means, that the binding object is not a passive repository of messages from which the receiver must pull the messages to be received. Instead, the active binding object is allowed to invoke the receive action by an up-call on the receiving objects.

The binding object is governed by a QoS contract, either static or dynamic. For static QoS management the QoS contract is negotiated before the binding is established and the agreed QoS values are used as parameters for the binding process. For static QoS management, a controlling object with a management interface is created. The controlling object can monitor the fulfilment of the QoS contract and offer renegotiation of the contract if the binding is not able to act according to the contract.

Binding factories

Because bindings are considered as normal objects in the system, the binding process is simply a normal instantiation process, expressed in detail by an abstract binding protocol.

Each binding factory supports the creation of bindings with a given functionality and a given quality of service. A generic behaviour pattern of binding factories is described below. The concrete implementation is supported by functions provided by the micro-ORB (interface reference management and local binding) and the appropriate binding library (stub and protocol channel creation). The implementation of a binding factory in Sumo-ORB is normally distributed. In particular, a particular binding factory normally consists of a number of cooperating local binding factories.

Operation interfaces are bound by a binding factory that uses the IIOP (Internet Inter-ORB Protocol specification) implementation of the RPC protocol. Continuous media traffic is supported by a connection-oriented, real-time transport service that offers fragmentation and flow control.

Abstract binding protocol

The abstract binding protocol relies on the existence of one or more binding factories that create particular classes of binding object. The speciality of these factories is that they generate distributed objects that span multiple nodes. To support this process, binding factories themselves probably are distributed objects with representatives on each node.

The abstract binding protocol requires that each node supports a primitive binding operation for linking two interfaces. The primitive binding operation, 'localBind', binds the communicating objects to the interfaces of the binding object. The 'localBind' operation is supported by a stub object and parameterised with a reference to the interface to which a primitive binding should be established. The local bindings make delivered information instantly visible to both participants of the local binding, and information cannot be lost or corrupted. Thus, the semantics is similar to local message exchange via shared memory.

The abstract binding protocol consists of four steps:

1. Client C requests a binding object to be created by invoking the binding factory for appropriate binding type. The invocation is parametrised with the set of references to interfaces, QoS annotations, and potentially other management related items depending on the factory interface.
2. The binding factory instantiates the binding object with type conformant binding interfaces at the right nodes. In order to do so, the binding factory has to do two things. First, the binding factory resolves the precise locations of the application object interfaces. Second, stub objects of the appropriate type are created for each interface. The stubs are in turn bound together by a recursive call of this protocol, unless the stubs are in the same node, in which case no further recursion cycles are needed. This step fails if required types or QoS cannot be achieved.
3. If the previous step was successful, the binding factory establishes a primitive binding between the application object interface and the corresponding binding object interface supported by the newly created stub.
4. If all the previous steps have been successful, the binding factory returns an interface reference to the control interface of the created binding object.

Conclusion

In Sumo, the ANSA binding model applies, but it is refined for QoS negotiation and distributed factories. For each channel class there is a separate, distributed factory. However, the protocol within the distributed binding factory is not open.

The abstraction level of the federated binding process above and the Sumo binding factory are different. The Sumo binding factory performs actions that are not visible in the federated model, but are considered as internal to the selected protocol layer.

9.5.3 DRYAD prototype for operation invocations

DRYAD project developed, in addition to the trader prototype, also some experimental tools for computational operation invocations.

The experiments aimed to a service type based binding factory that exploits trading. The Naiad prototype is described below. In contrast to the Sumo approach, the resulting binding factory is independent of the channel type. In a federated system, there are no commonly accepted channel types that would form a well-established and inherited set of services. Instead, type repository must store the known types.

One of the goals in the DRYAD project was to identify a smooth propagation path from current networked systems to the future federated systems [129]. For this purpose, a special purpose interceptor, Daphne, was created. Also Daphne is briefly illustrated below.

Naiad

The Naiad package offers programming interface that supports interrogations (and announcements as a special case) as a method of invoking operations at remote objects. Characteristic to the supported operation invocations is that

- communicating objects are connected by late, dynamic binding,
- instead of direct addresses or names of the target objects, the required behaviour can be named,
- failures are considered as normal behaviour in the distributed system and therefore retries, recovery and replication have no special position – their existence and occurrence is usually not even visible to the application programmer,
- each communication action is associated with an explicit task-performing semantics that describes potential terminations, cancellations, delays, protocols, etc., and
- each communication action is associated with an explicit environment contract that describes how the object communicates with the underlying infrastructure services when requesting interrogations through it.

The Naiad package is implemented as a library for C programmers [68, 126].

The Naiad package functionality is supported by a service invocation controller (i.e., a binding factory). The service invocation controller uses the services of the trader, the type repository and the policy repository, and when a server object or a server template is selected, it starts the binding establishment. In the binding establishment, the first task is to create a channel controller. The channel controller is then instructed to instantiate binder objects at both client and server objects. If the trader selects a server template instead of a server instance, the invocation controller instantiates the server. Under the binder objects, RPC is used as a protocol layer. Operations at this level include message exchange, connection status testing and opening, closing, suspending and resuming a connection. The only properties that are required from the RPC system are message transport without loss of packets and delivery of messages in the correct order at the correct port. Figure 9.9 illustrates the system.

Clients and servers are represented by stub objects. A client stub uses the invocation controller to set up a channel (binder to binder) and then transmits the input data required for the operation. The request for the invocation controller includes

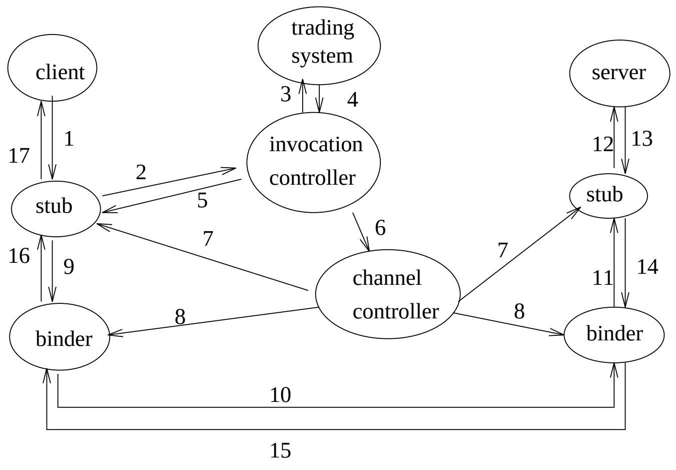


Figure 9.9: The Naiad system structure.

- abstract service type,
- operation signature type,
- operation signature, i.e., the actual operation parameter values,
- federation mode and method,
- import criteria (to be used when trading the service), and
- QoS attributes (i.e., the environment and service liaisons).

The QoS attributes define the environment and service liaisons of the client. The environment contract includes client's reservation to cancel the operation later (rollback features required, not full support for transactions), and virtual execution environment (does the operation change the client's environment or the server's environment). The service access contract includes indicators for communication synchrony, resource consumption limits, and failure recovery protocol [124]. The attributes that are actually supported by the current implementation are

- implication of interrogation or announcement,
- request for synchronous or asynchronous execution of the service, and
- time-out value for the request.

The concept of communication synchrony is differentiated from the separation between announcements and interrogations. Synchrony between application objects means that the client is going to wait until the server has completed its task. However, announcement does not return any data. After release from a synchronous announcement the client knows that the service has been accomplished if no failure report is given.

The Naiad system allows the client and infrastructure to communicate either synchronously or asynchronously. Asynchronous announcement from the client to the infrastructure does not give any guarantee on the service invocation. In contrast, synchronous announcement to the infrastructure guarantees that the target computing system has received the request and promises to invoke the service there. For announcements, no guarantee is given about completing the service.

In the student project [68] that implemented the Naiad component, we used the ODP concepts and viewpoint languages as a specification guideline. The concepts were considered useful and the resulting implementation was far more compact than it would have been without this approach.

The Daphne virtual server interface support

Daphne is an interceptor, that allows legacy software to transparently access the DRYAD infrastructure services. Figure 9.10 shows how two architectures can coexist in the same system: In a legacy system, a client refers to a server object by its name in order to invoke an operation. We can replace such server objects by a virtual object controlled by Daphne. The virtual objects play as proxy clients for the invocation controller of the Naiad system.

As objects are mostly represented as files in their operating system environments (like UNIX program files), we chose to implement Daphne as a special file system. Each virtual object is associated with personalisable criteria that are automatically used as import criteria in trading. At the implementation, these criteria are stored to the Daphne system files.

Because of the technology selections done in the implementation of Daphne, it is possible to offer virtual server interfaces to personal computers that are able to run NFS protocol.

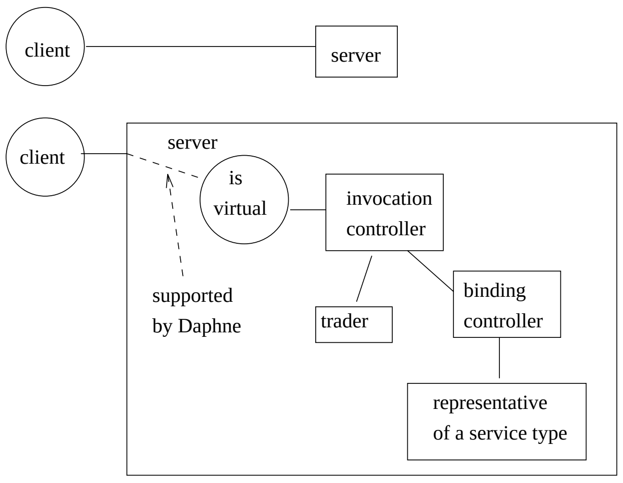


Figure 9.10: Coexistence of two architectures in the same system.

9.6 Conclusion

The presented federation architecture minimises the shared assumptions required in the construction of middleware platforms. Although several questions arise on the feasibility of this model the benefits are fundamental. In the following, we first study the feasibility questions on issues like performance, management, techniques for expressing contract details like managed object properties ('policies'), and security. After that we consider the benefits of federation capabilities.

Feasibility

In comparison to interoperation that takes place within a community with a shared administration, the federated model surely causes overhead at run-time. However, comparison of the two models based on run-time performance is actually not fair: The flexibility and evolution support of the federated model effectively decrease the system maintenance effort required for arranging interworking facilities between systems. Both the system engineering time and the run-time performance time should be evaluated. Eventually, our belief is that the overhead of the federated model at run-time is bearable. This requires three restrictions. First, federations must be established on large application package interfaces instead on process ports for scalability reasons. Second, propagation of trading requests should be restricted to a small number of hops (1-3) because network delay is the dominant factor in trader response times. Finally, type conformance should in most cases be based on assertions, because in a generic form the problem is too hard to be solved [126].

Languages for expressing contract items form a problem if an elaborated facility is expected. However, as an initial phase, identifiers and names, number values with standardised semantics, and IDL specifications are sufficient. Object management does not create a need for expressing detailed policies in the contracts. The federated architecture model divides management aspects to traditional management activities, such as resource control, and federated behaviour control that is based on a policy framework [127].

Security of a federated architecture is our greatest concern. Within OMG, security studies are in progress [171], and it has also been suggested, that the recommendation should be adapted as part of the ODP set of standards as well. However, the secure ORB design does not give sufficient answers to the questions arising from system sovereignity.

Benefits

The described federation model allows sovereignity of systems and thus tolerates a high level of heterogeneity. The founding principle for the design is to deny direct control of remote objects. Instead, each domain is managed by a private controller and this controller makes all management related decisions. The federated management methods must feed these controllers with contract and monitor information in order to persuade them to make suitable management decisions. The federation model also supports controlled access to otherwise closed systems.

The federation architecture and the programming-in-the-large approach of megaprograms have shared goals. Both approaches attach system size, persistence of the systems over evolution steps in system components, and diversity of various aspects, like software production concepts and tools, communication facilities, and interfaces. An application in a federated environment can be seen as a megaprogram [251]. However, the evolution of megacomponents is not controlled by the megaprogram programmer, but the federation mechanism is used for creating a temporary commitment between the components. The megaprogramming tools trust in a component database and a shared ontology. In the federated environment, the database is replaced by traders, and the shared ontology is selected at run-time, instead of compilation time.

Traders can be used as trusted entities to guarantee a fair scheduling of tasks among users of a large network. An analytical study for a generic trading environment has been published [255]. The study concentrates on the effects of accessing a service when the service provider is found by trading. The study points out that the delay in server state information acquisition is significant in comparison to service time. Example services in the environment under study included trading and name servers. Rather good results were achieved by combining the use of actual state information with random selection. The study also shows that if clients can optimise the selection based on their own needs only, the overall system level performance suffers. The study suggests, that the traders should be used as trusted entities to prevent too selfish optimisations.

The federation model can be exploited by programming environments in multiple ways. First, the type repository service can serve as a single point for inserting new services and new service versions. This facility saves from recompilations or re-implementing of the client programs, even if exploited in a single domain environment. Second, the federated binding model elaborates the traditional client-server model by supporting multi-partner communication, and distribution transparencies. Especially, failure transparency of communication can be build on the liaison concepts. The channel controller within the federated binding introduces tools for implementing self-healing channels. Such a channel is able to reconfigure itself whenever a channel component fails.

Chapter 10

Trading at open markets

In this chapter, we discuss commercial systems among people and organisations, supported by computer networks. Traditional solutions for exploiting computer networks for commercial needs are based on EDI [85] (electronic document interchange). These solutions are closed networks: the clients and service providers have to negotiate and sign a valid legal contract before starting document interchange in electronic form. Open markets differ from this design, by denying the need of preceding contracts. The goal is to allow as many people as possible to gain access to information facilities connected to the Internet. These solutions trust in dynamic loading of software over a shared language platform (e.g. Java RMI [217]) or shared access tools for equally structured data (WWW browsers).

The area of electronic commerce is only now starting to consider using distributed system platforms. Naturally, the progress plans show high expectations on the distributed platform (or standardisation) abilities in respect to security and interoperability.

Currently, ideas of EDI and object oriented programming frameworks lead the design of commercial network services, because the focus of design, standardisation and research is on suitable user interfaces. However, a major design requirement has emerged from the application area: three tier configurations match the commercial world model better than client-server models. Besides product or service consumers and producers the three tier architecture always introduces some form of broker, merchant, trader or agency. A vast number of broker cantered business models are currently evolving (a few examples: [258, 2]).

The current state of actual electronic commerce software is limited to cataloguing and payment facilities. By cataloguing we mean facilities required for advertising products and services to potential clients. The market is supported either by brokers that mediate information about available products or by commercial traders that mediate products. The market needs also support of banks for money exchange and notaries for registering contracts.

Section 10.1 sketches an electronic commerce framework, while Section 10.2 discusses evolution strategy for electronic commerce software as agreed by consortia on the field. Section 10.3 describes a realisation approach for the electronic commerce framework governed by the strategy. Section 10.4 considers the open problems of the approach, and Section 10.5 suggests some solutions. The overall solution strategy is followed by examples on exploiting traders in the construction of individual software components: a component can be build around a trader, or trading can be used in the production process of a component.

10.1 Electronic commerce framework

Electronic commerce frameworks try to capture the essential commercial processes and formulate – even modify – them in such a way that computers and networks can be exploited. The commercial processes are illustrated in Figure 10.1 and discussed below.

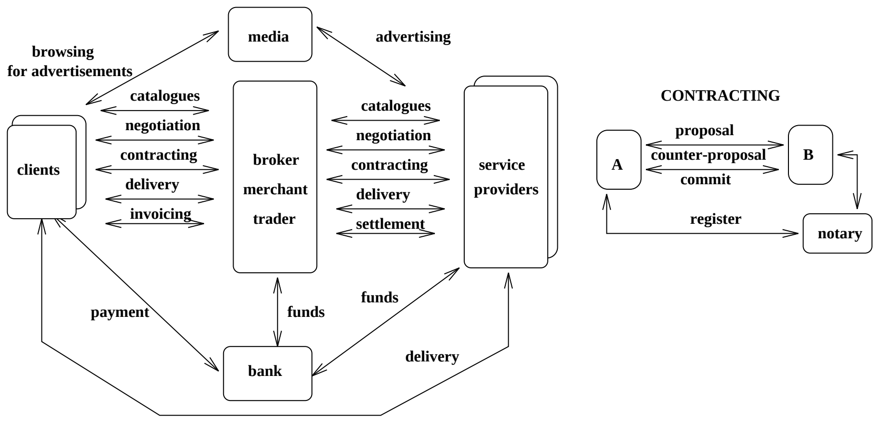


Figure 10.1: Commercial processes.

Basically, commerce is about buying and selling products of various kinds, both physical entities (like books, network cards, and disk drives) and information related services (like addresses of book shops, files containing the book text, authority to exploit disk space at someone's computer). A producer creates an entity, either a concrete artifact or a piece of information, that is interesting enough for some clients to pay in order to obtain or access it. In order to get the product sold, the producer must make the product and its availability known via advertising in some media. Besides the traditional advertising in newspapers, TV and such media, also the network facilities are now available, for example WWW.

If dealing with advertising and negotiation about prices is too cumbersome, the provider can out-source these activities to a broker. A broker makes a living by buying cheap bulks from a number of providers and selling with raised prices to the clients. A value-added provider buys products of other providers and combines them to another, more interesting product. From the point of view of a client, even a broker is a value-added provider: the broker has a variety of similar products to choose from. Search engines and link collections in WWW can be considered as non-commercial brokers.

In case of simple products, the purchase is a simple process. The client selects a product, pays for it, and takes the artifact or accesses the information. However, the process can be complicated by involving invoicing, settlement, and separate delivery of products. Electronic commerce systems currently are involved only in invoicing and payment tools. A developing system is, for example, the secure electronic transaction (SET) specification [145]. Delivery systems are not commonly considered as part of the electronic commerce area, although the products often are information based. Still, delivery management systems should be integrated.

In case of complex services, the interaction of interest here is contract negotiation and establishment. After the contract is established, the performance of the service (writing a book, shovelling snow, cleaning the house every week) is not an electronically controllable process. However, the contracts can be electronically maintained in a notary service for later use. In addition, monitoring of the service can create electronic reports on the quality of the service. These reports can be compared to the contract itself, and corrective actions can be taken depending on the contract contents. In principle, all interactions between the provider and the client are governed by a contract. In simple cases the notary is omitted or replaced by a register.

10.2 Evolution aspects

The EDI standards still have a surprisingly strong standing as the primary solution on the electronic markets, in spite of their aged technology. The EDI standards view electronic commerce as exchange of documents. At each commercial process, a different set of documents is required. Each document set is standardised separately and there is no enforced relation across the various documents. Three distinct areas have their separate standard sets: the semantic information contents of a document, the data representation format of information, and finally, the data transfer protocols. The EDI networks are by necessity closed, and the establishment of a connection requires the negotiation of messages, formats and protocols. The messages have been separately standardised or agreed for each type of merchandise. It is often the case that the same format of data is not used if the transfer protocol changes.

The modern electronic commerce designs still include the three aspects: documents, information representation and transfer protocols. However, the techniques for these areas are changing. Furthermore, the closed EDI networks are expected to be replaced by open networks where clients, providers and brokers can join without a preceding off-line negotiation. The open market model also discards the bilateral EDI contracts and replaces them with multi-party interactions. The openness of the market is assured by public interface specifications for documents.

Currently, major vendors independently offer electronic commerce platforms. For instance, the following platforms are available: IBM Commerce Point, Microsoft Internet Commerce Framework, Netscape ONE (Open Network Environment), Oracle NCA (Network Computing Architecture) and Sun/Javasoft JECF (Java Electronic Commerce Framework) [234]. These vendors are also members of CommerceNet consortium, that intends to exploit CORBA as an interoperability support and shared object bus for economical objects [234]. CommerceNet also wants to agree on a minimum set of standard business messages in a Common Business Language, CBL [69]. The purpose of CBL is to replace text based EDI messages by object-oriented CBL information exchange.

10.3 Realisation strategy

The realisation of electronic commerce software is suggested to be built around an object-oriented programming framework. (A programming framework is almost complete application that the programmer can customise for particular use.) The framework should include set of shared services from which applications can be assembled [234, 147, 146]:

- exchange of semantic, economical objects (profiles and catalogues) between the open market participants;
- selection and negotiation of the contract associated to a product;

- certification services (authorisation, non-repudiation, certified record, anonymity) for clients, providers, or brokers in commercial transaction; and
- payment services.

A profile object describes a service or a product offered or requested in a format that can be interpreted correctly by all basic services at the electronic commerce platform (object browsers, catalogue services, brokers). A profile object must therefore contain contract and related service description.

The contract structure must follow the information content of real commercial contracts [147]. Essentially, a contract involves a life-cycle starting from contract creation, followed by steps of client and server responsibilities, and terminating with the total fulfilment of the contract from both sides. This life-cycle is expressed as a service description. It refers to (names) commercial services, like payment, purchase delivery, or advertising. In addition to commercial aspects, the profile object may also contain technical contract information related to the computing network processes involved.

A catalogue is a container of profile objects, and it allows profiles to be sorted for different views and modification of individual profiles.

The common desktop facilities (for example, object browser, inspection and shopping parts) are expected to be portable and inter-operable in the desktop environment of different electronic commerce systems. The interoperability is ensured by using OMA architecture and openness and connectivity by using Internet [234, 146]. The object framework would serve as a unifying middleware layer that creates an illusion of a homogeneous system with services like authentication, invoicing and payment, and catalogues. The use of a framework also allows exploitation of inheritance hierarchy for maintaining relationships between service requests and business objects and for evolution of new business objects.

In order to make the potentials of the reference model more concrete, we view an example system developed in the COSM project [148].

The design of COSM is organised around trading service that serves as a catalogue server. From the trader the users can obtain catalogues of services, like list of travel agencies or lists of holiday packages. The catalogue browser is a generic client agent that allows a user to browse the catalogue with a graphical user interface. This can be done, because the catalogue structure is the same at each site. When a profile is selected from the catalogue for closer inspection, the agent must be specialised for the corresponding server interface. When a selection is done, the generic client requests for the server interface details. Using the detailed interface information, the generic client is able to load the required user-interface program code. The loading operation turns the generic client to a specialised client, that is able to fully communicate with the server. The design trusts in an object framework that gives the features of the service, but still requires customisation. This customisation is normally done by programmers, but in this case at run-time. For example, button texts for a graphical user interface can be loaded this way.

The loading phase requires that the service interface specific code is represented in a portable way. Therefore a service presentation language has been developed. A service representation includes: operation signatures, specification for user interface components (dialog boxes, buttons), server address and invocation specific information like parameter values. The loading phase is implemented with the help of CORBA dynamic invocation interface.

10.4 Problems in EC framework evolution

The object framework for electronic commerce does not yet give solutions for the problems of administering contract and service information in the open markets. The consortia also assume, that a shared platform architecture will emerge in such extent that all electronic commerce can safely be expected to adopt the object framework based on OMA and Internet. Furthermore, portability of software is assumed to be a primary goal with no drawbacks. We will discuss these three assumptions below.

Agreements on services

The current plan is to create a standardised, hierarchical taxonomy of service types [234]. The standardised tags would then be used in profiles to denote what services are requested, offered or contracted.

The taxonomy would be fixed and very slow to evolve, therefore, use of such taxonomy hinders the development of new service types. The service types should be easily defined in an on-line repository.

It would be very difficult to agree on a global taxonomy of services, because the grouping of service items are traditionally different in different cultures. For example, 'normal insurance package' of a family is very different in different countries, because of underlying differences of social security systems in each country. Therefore, some kind of domain structure should be reflected by the service taxonomy as well. The taxonomy should also capture natural relationships (inheritance or containment) of services.

Finally, the service tags are not able to explicitly denote the contract structures involved. It is not practical to trust users or user interface agents to enforce that all such contract related information is available that is necessary for a given service type. That would require that either users or agents are aware of the commercial rules related to the merchandise, making the evolution of new product types difficult. The contract rules should be explicitly available on-line, so that also they can be easily evolved and exploited. Furthermore, especially the contract structures vary from domain to domain (national laws).

Distributed system platforms

The current plan is to trust OMA and Internet architectures, together with a shared object framework, to solve interoperability problems. The OMA architecture is considered as the de-facto standard of distributed systems and therefore guarantee a reasonable coverage in future years.

However, OMA has some restrictions in regard to federation capabilities [128], that should be overcome by the commercial reference model. These restrictions deal with heterogeneity of server interfaces, service type repositories, and management of interactions between sovereign domains. Fortunately, the CORBA products and OMA architecture are still evolving towards ideas represented for example in RM-ODP. Therefore, commitment to OMA leads to proper designs provided that its shortcomings are identified.

Software architecture

Object-oriented programming frameworks are popular because they offer full portability of software, with an ability to customise important features of the service. In the previous section, we saw that even the customisations can be done at run-time, in a portable way.

However, if this run-time customisation is not sensitive to localisation aspects, an essential aspect for global open markets is lost. At each (geographical, national, legislative) domain, different languages, customs, and laws regulate the special requirements for the user interface. The programmers or users of commercial applications should not be forced to face concepts outside the context of the user community in which the application belongs. For example, a user purchasing a book should not be bothered with tax regulations in a foreign country, although the book may eventually be mailed from a distance. This means, that the representation of merchandises and representation of market interfaces must be specialised for each domain.

10.5 Open solution strategies

For the management of merchandises and commercial actions on the electronic markets, we need three levels of concepts: contract structures, service type descriptions, and representations for individual services. Each of these concept categories must be supported by a separate repository service, in order to allow independent development cycle for each category. Figure 10.2 illustrates how contract repositories, service type repositories, and server repositories can utilise each others services.

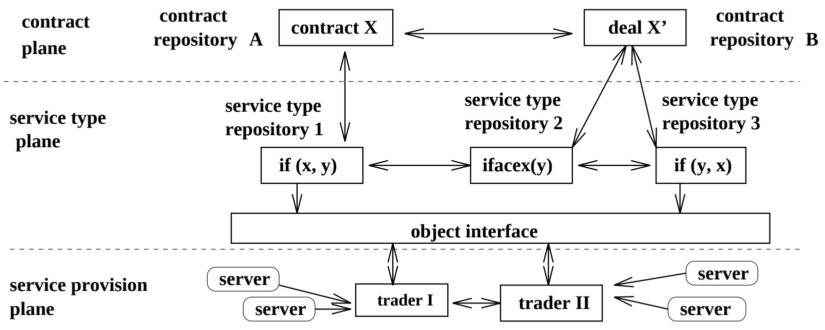


Figure 10.2: Constellation of repositories.

As a general rule, each repository service supports a domain structure, in order to allow federation between autonomously administered domains. The domain structures involved in commerce include

- legal domains, where commerce is regulated by the same law, and
- market domains, where commerce is regulated by the laws of demand and supply.

In addition, the repositories are realised within an open distributed computing environment. The domain structures involved in distributed computing include

- organisational domains, where the owner of the computing system regulates the operation policies, and
- technology domains, where for example the selected programming environments and communication protocols regulate the operation.

These domains do not necessarily share their boundaries. Instead, a legal domain would, for example, exploit several technology domains, but each of these technology domains might as well span multiple legal domains.

10.5.1 Contract repositories

Contract repository (suggested by [155, 156]) provides a common understanding of contract types and contents. A common business language, CBL [234], could be used for defining the required concepts.

A contract includes (slightly modified from [157]):

- identifier and validity domain of the contract (validity domain refers to the legal or professional domain under which contract enforcement can take place);
- roles of contractors;
- period of contract expressed either as times or as defined termination clauses;
- services or products exchanged, and for each exchange the quality and quantity of services or products, including delivery time and cost.

The contract repositories support operation for creating, modifying and deleting contract types. In addition, relationships can be asserted between contracts in order to allow choices of contract forms.

The contract repositories can be independently constructed at domains that are governed by joint jurisdiction. Services can be either more locally known, or even known to a wider domain. The contract repositories can create relationships and mappings between contract types. Only asserted relationships can be used, but otherwise the mechanism resembles the type repository function mechanism.

The services or products the contract involves may be concrete and therefore outside of the computer supported system, or they may be computer supported, in which case the situation is more complicated. In both cases, services and quality of service aspects are managed in further detail by service type management.

10.5.2 Service type repositories

Service types are essential in the electronic commerce framework: The actual merchandises are naturally grouped by their type, and furthermore, the contract structures related to each service should naturally match the merchandises.

The purpose of service type repository is to identify whether the proposed and required services are actually similar enough to be used in a contract. In the case of electronic commerce, the service type repository may carry types for both information processing services and physical merchandises.

For interoperation, a logical similarity of abstract service types is necessary. At the technical level, interoperation requires similarity of interfaces. Major factor for the measure of required similarity is that the information flows received through the bound interface are sufficient to fulfil the expectations of each communicating partner. The data and message structures involved may be converted in some cases with an interceptor.

The service type repository function mediates service type information at run-time among the interoperating systems. Each member system may use a separate type system, each of which expresses a set of logical, abstract type concepts and corresponding technical, platform dependent templates. Type is defined as a predicate that characterises a collection of objects. For instance, a type can be used for expressing the similarity of services, interfaces, or behaviours. Template is defined as a set of features expressed in sufficient detail for instantiation. Clearly, the concept of template is dependent on the instantiation facilities, i.e., dependent on the assumed platform. Therefore, a set of features can be a template at one platform, but not necessarily at other platforms with different architecture.

We recall that a type repository contains a set of service type descriptor. A service type descriptor includes a set of type definitions in different type languages. The service type descriptors include abstract service type name, interface signature, and attribute definitions (i.e., attribute name, and attribute value data type). In this structure, the type concept is represented only by the abstract service type name and the related attributes. Other parts are related to templates. The attributes can carry QoS information required by the contract that governs the domain of service type. The attributes can be stored also in the contract repository and inherited to the type descriptor. Additional attributes can also be defined for expressing service specific aspects, such as jitter of voice transmission or migration transparency support.

In addition to type descriptors, the type repositories may include type relationships, even to type descriptors managed by remote type repositories. The relationships carry also information for technical level transformations on the service interfaces. Via these transformations, the distributed computing environment is able to hide from its clients the technical differences present at individual technology domains. Similarly, more user oriented aspects can be transformed. For example, property names can be presented in the native language of the user.

10.5.3 Traders as server repositories

In the electronic commerce framework, the trading function naturally supports various brokerage and mediation tasks.

Trading, as a standardised mechanism advertising and discovery of service offers, can be used for manipulating individual profiles. The profiles can be represented by service offers, as both are basically structured in name-value pairs. Providers can export their offers to a trader, and clients can browse the offers or request the trader to select a set of offers that match the explicated requirements.

An offer should represent information covering contracts, service types, and service representations, and furthermore quality of service aspects. Although the service offers give a flat sequence of information items, each of the items must be interpreted through the appropriate repository. The repository hierarchy cannot be collapsed to a flat structure, because the domain boundaries of each repository are independent of each other. Furthermore, the evolution cycles of the concepts represented in the repositories are independent from each other: there is no reason to change interface signatures of a printer server because a monetary union is established.

Also a catalogue service can be implemented by a trading function. A broker could join together several catalogues by creating a network of traders, for example exploiting proxy offers or links. The broker could also act as the trader administrator of a provider.

10.5.4 Trader supported merchandises

The open market is naturally involved in computer supported or computing related merchandises. These merchandises can exploit trading functionality either as an embedded component, or as a tool in the production sequence.

Information repositories

Various information repositories will be popular services at the open markets. Examples include listing of publications, recommended movies, and catalogues of experts at different areas like dentists, car repair, and computer scientists.

The information repositories may support three different usage models. The first model, is a normal database that is made openly accessible via a network (Internet) for information retrieval. The second model, is also a localised database, but it can be freely updated by any interested user. The third model supports a federated database system. All three models can exploit traders as implementation components.

In the first model, the only benefit gained is the use of a standardised interface. None of the special features of trading functionality is actually used: The database is local, and collected by ad-hoc methods. In most cases, the user interface requires that a full search is performed. The service could as well be supported by any database management system.

In the second model, individual users can act as exporters. The trader forms a directory service that is freely updatable. However, there is no protection mechanism for the service offers intrinsic for the trading functionality. Nevertheless, the trader storage mechanism may impose a security mechanism, as it has to be done in X.500 implementations [101]. The information service constructed in this way can exploit the standardised interfaces for importing and exporting.

Finally, the information repositories may be linked together in order to distribute the amount of offers, or the responsibility of database administration and update. The benefit of this design over federated database systems is the embedded failure transparency of communication between traders, provided that the ODP communication model is also adopted.

An essential feature for trading functionality is the separation of type system management from the matching mechanism. Traders derive the structure of the manipulated information items, offers, separately for each import and export request. Consequently, the evolution of information structures is strongly supported. In addition, the type systems may express arbitrary subtyping rules, and thus the similarity requirements for the stored and compared information objects, offers, are extremely weak.

Trading has also a simple benefit over search engines currently so popular in Internet. Search engines, like AltaVista [46], support world-wide searches of WWW pages, based on key words. Traders use structured information instead and thus allow more optimised selection to take place. Furthermore, WWW search engines expect the pages themselves to contain necessary meta-information that describes the information content, update time, and page administrator. In contrast, traders allow meta-information to originate from a separate source. In respect to scaling problems in large environments, e.g. the Internet, the trading and the search engine approaches have no essential difference.

Information repositories available through Internet include Fast [84], COBRA [51], Kasbah [31], just to mention a few.

Tailorable software

The merchandises of the open markets also include computer software. Currently, software is very often object-based: programmed with object-oriented languages, or associated with an IDL specification for publication.

The software components are encouraged to be reused within companies or software engineering projects. The reuse is usually supported by component libraries and by inheritance hierarchies. These mechanisms restrict reuse to rather small group of exploiters.

Traders appear a tempting tool for organising reuse of software components in a wider environment. Descriptions of the components are exported to a trader and classified by their intended behaviour. The problem is however, that no commonly acceptable method of expressing object behaviour is yet available. The repository should therefore be a browsable collection of informal, descriptive texts about the components. This arrangement does not encourage to very large scale cooperation. Still, the design would serve as a CSCW tool.

An object component trader requires a working environment where the roles of objects are agreed. Such conditions would appear, for example, within a company, or within a large software project. Examples of this kind exist: A company selects component versions for custom tailored software deliveries from an object storage via a trader [24].

10.6 Conclusion

We suggest that the electronic commerce reference model under development considers tools available in ODP reference model.

The problem of defining a global type system for services can be solved by allowing a dynamic repository to support commerce software. The solution is flexible: new types can be added whenever needed, and if different solutions are accidentally imposed, a mapping can be created between them. New types can be easily adopted by end-users and programmers.

The electronic commerce reference model should also consider the problems of isolated contract domains and federations between them. A contract repository is already suggested elsewhere, but we feel that it complements our approach.

The federation problem arises at other areas as well, type systems and distributed system platforms. The federation problems between distributed systems should not concern the developers of electronic commerce software too much, as the platform designers work towards distribution and federation transparent communication. In this work, services like federated type repositories and ODP trading function have a key role.

Although type repositories and trading are designed to support distributed transparent ODP platforms, they have further applications. The electronic commerce software requires classification tools and browsing tools for services other than computer based information services. These can be supported by type repository and trading mechanisms, provided that a federation mechanism is included.

Part Five

CONCLUSION

In this dissertation, we have inspected aspects of openness in world-wide information-processing networks and the role of trading functionality in open systems. In the design, we have aspired to solutions for the topical problems of interoperation: communication across fixed and mobile communication networks, QoS management, and multi-point communication. However, it is not possible to discuss all of these problems in detail in this dissertation. Chapter 11 summarises the results of this dissertation and discusses the consequences of the results for further work on distributed and federated systems. Chapter 12 briefly comments on the future prospects on the needs of modern distributed and federated systems. As this dissertation is just a single representative of the huge number of projects in the area, the commentary takes a broad view on the needs of standardisation and consortia recommendations.

Chapter 11

Conclusions and consequences

This dissertation presents results on the characteristics of trading functionality, and on the open system architecture design. These results are being used as contributions to the family of ODP standards. We consider the ODP model to be a practical route to provoke parallel changes in the commercial platform architectures, like OMG/CORBA.

In the following, we first summarise the trading characteristics presented. Then, we summarise separately the architecture related results on the federated system architecture, and on the use of ODP object concepts. Finally, we discuss the consequences the presented ideas have on software engineering processes and software architectures.

Trading

We introduced trading as a special kind of global information repository, where a large set of exporters can update the repository and where importers can retrieve information by resource-restricted, non-exhaustive queries. We also investigated separately the realisation of trading service in a distributed and in a federated environment.

We studied the principles for establishing trading domains. Substantial aspects for domain division include autonomy of organisations to evolve their computing systems independently, standardisation of service interfaces and behaviour, trading scalability, and variety of platform architectures involved. We concluded that trading domains should be divided according to service type domains and platform architecture domains, whenever appropriate. This principle helps to keep the load at each trader as low as possible. In practice, the amount of interoperability among federating systems is based on the number of applicable offers in the trading domains and the ratio on which those offers are selected.

The exploitation methods of trading cause different needs. Traders can be used for browsing services by end-users, in which case, the response times can be rather long and a rather thorough search through the trading graph is expected. However, the primary exploitation scenario in this dissertation was federated binding protocol, where an agent plays the importer role. In this case, the trading graph must be rather shallow, limited to 1-3 hops over the network. The dominant factor in import operation times is the network delay.

The offer repositories of traders can be efficiently supported by current database systems or object database systems. Moreover, caching and prefetching techniques – analogous to virtual memory management or web caches – are applicable.

Trader federation or trader interoperation is supported by a policy framework. The trader interface signatures have been standardised, and even the de-jure and de-facto standards are equal. The trader behaviour has also been standardised, de-jure. However, within the trading behaviour, some alternative behaviour patterns are accepted. Such alternative behaviour patterns are controlled by a trading policy framework that expresses the trader assumptions as trader property assignments and the importer assumptions as policy parameters of the import operation. The trading policy framework is explicitly manipulated and controlled by the trader objects themselves. Application federations exploit similar concepts in interoperability negotiation, although the negotiation is done on their behalf by meta-information services during the federated binding process.

The use of separate type repository is an essential feature for the trading functionality. Contract schemata and semantics can be inserted to systems via type repositories without disturbing the system operation.

Openness and federation

We have chosen to study the characteristics of trading in a federated system environment. Although the traditional distributed systems benefit from the use of trading functionality, such an environment does not exploit or even require all features.

We consider the federated system as a next evolution phase at the area of distributed computing. Essential change that is required is the adaptation to multi-organisational environments and asynchronous evolution of software at independent domains. The federated system architecture presented in this dissertation fully trusts sovereign systems and their dynamic negotiation facilities, which differentiates the approach from related work.

The essential properties of a federable system include

- capability of joining relevant, service type specific federations at run-time,
- support for federation transparent interoperability,
- support for distribution transparent, multi-partner communication, and
- support for contract based binding management that provides both run-time QoS management and negotiation of communication system properties.

The fundamental concepts in a federable system are those of contract schema and liaison. Contract schema is specified and standardised separately for each service type domain. Liaison negotiation requires mediation of potential contracts to a matching process that resolves all communication related aspects concurrently. The negotiation process can not be recursive as in distributed systems. In distributed systems, the decisions made at application and platform levels do not exclude each other, because only the application level and transport protocol level decisions need to be made. In contrast to this, a recursive decision mechanism in a federated environment would constantly lead to situations where already made decisions would make all further alternatives non-exploitable.

In the presented federation architecture, we collect all interface level information to a potential contract that describes the properties of an interface. This information we consider computational interface reference, because it only indicates the limits of engineering techniques acceptable. The potential contract information can be mediated via traders during the binding process between computational interfaces. In practice, the binding process results to an engineering level binding between engineering level interfaces. The engineering interface references include detailed information about the selected technology for message passing between communicating objects.

We consider that the presented federated system architecture is feasible. We base this opinion on the experiences on trader behaviour and the design of federation transparent interoperation that in most parts collects together individually tested mechanisms. Large scale experiments are still missing, because an interoperation experiment with federated architecture should be established across geographical and architectural distance.

Object model

In the design of federated system architecture, we have trusted the object concepts of ODP reference model. This object model addresses concepts that are inadequately considered in object-oriented programming languages and object design methodologies. The ODP reference model offers a model that essentially differentiates between types and templates, separation of views to interfaces, and causalities of interfaces. The ODP object model is better suited for architectural discussions, system design, and programming tasks in large systems than traditional programming and modelling techniques.

However, even within the ODP object model, consequences of the federated architecture can be identified. As an example, we can mention the concept of a computational interface: A computational interface should be regarded as an engineering object with polymorphic features (interceptors), instead of a data-type independent abstraction of engineering object methods. The interpretation allows interceptors to be used for transforming both representation and behaviour.

Federated software engineering model

The federated system architecture also leads to a federated software engineering model that can be founded on a federated application architecture.

Traditionally, a distributed application has been designed in a single software engineering process (in which we include also maintenance cycles). The implementation may be divided to multiple production groups, but nevertheless, the components are integral parts of the final product. The federated system model promotes a distributed model of application construction. Independent application objects can be designed, implemented, made available in the network, and maintained independently by several software groups. Figure 11.1 illustrates the change: The traditional process requires a single control of the production, while the distributed process allows isolated production lines. The products of the isolated software groups are expected to interoperate, because the meta-information services are used during the software development.

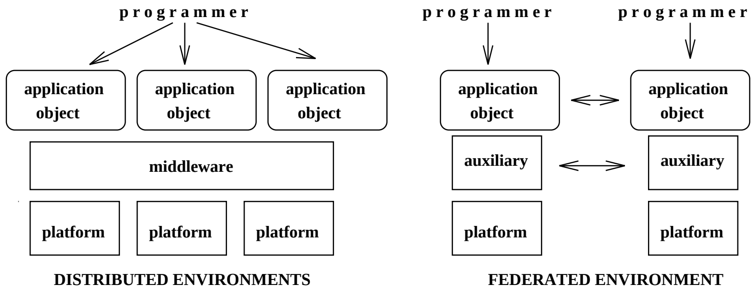


Figure 11.1: Scenarios of the software engineering processes in open distributed environments.

The new application architecture presents a set of application component objects each of which has an independent life-cycle of their own. Each object has its own choice of technology and can evolve independent of the application it is used as part. Applications are built by reusing this kind of resources. The architecture allows combination of different technologies for processing, operating systems, middleware, and communication. For instance, it allows cooperation between a fixed network and a mobile network. In Figure 11.2, we see a typical distributed application architecture on the left side: Each application object has been distributed to several nodes, and when one node fails to support the functionality, the whole application object is obsolete. At the right side of the illustration, we can see the federated application architecture: each object represents a full functionality of an application, and when distributed execution is necessary, the application object can request services from its counterpart in a remote node. If that node fails to support the application, the requesting application object is not deprived of the service – the same service can be requested from somewhere else.

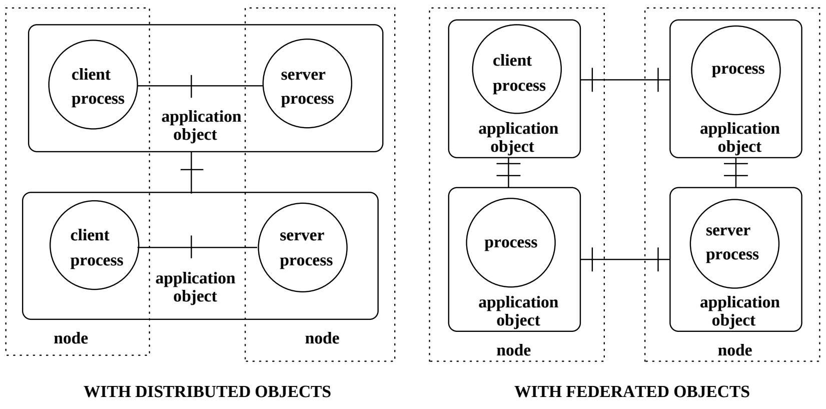


Figure 11.2: Application architecture scenarios for open distributed environments.

The federative application architecture moves responsibilities from applications to platforms. As the interface binding process becomes more demanding, that process is handed over to the infrastructure. As the infrastructure becomes an active player in the communication process, it is natural to exploit it in other functionalities as well. For example, the responsibility of translating data-representations is trusted to the infrastructure.

The distributed software engineering process presents a set of projects each developing a single, independent object that can be effectively reused in several applications. In this model, the application objects are self-contained and can create liaisons with their environment. Therefore, the development process of the objects can be more independent. For example, a teleconferencing service may exploit audio and video recording and replay services (independently) at each conference location, and use several underlying data transmission services between the locations. The teleconferencing packages at each location can be differently structured. For example, one package may include a single object whereas the other separates audio and video. It is by no means necessary to introduce autonomy for each location. Instead, centrally controlled distributed services are encouraged by the architecture (for example, data transport services).

Chapter 12

Further prospects

Three development areas need to be progressed before the open distributed system architectures or even the trading functionality can be efficiently exploited. These areas are related to

- the provision of meta-information services in general,
- evolution of software engineering processes,

and more specifically, to

- the trading function itself.

First, the open distributed system architectures require designs and implementations of standardised meta-information services. Introduction of these services into various commercial platform architectures is intrinsic for their exploitation.

Currently, major distributed platforms do not support the functionalities required for federation or dynamic establishment of distribution transparent interoperation.

Although a trading function has been implemented by some vendors, the type systems supporting the trading activities are still static. The reluctance to implement type repositories is connected to the strong position of inheritance-based interworking model supported by object-oriented programming languages and design methodologies.

The transition from traditional object-languages to an ODP-style object environment can be organised by exploiting reflective object model [21]. The idea is also present in meta-programming systems (e.g., [115]). In these models, the interpretation of a message (operation invocation) can be adapted or modified within the system during the system operation. An analogous situation arises, when an application program is compiled with two compilers that have different semantic rationale for a programming language operator.

Besides type repositories, also federation of name services is desirable. Type repositories and traders create, in fact, specialised naming domains for properties used in trading and in type definitions. We also consider interface references and addresses within communication protocol domains as names, which creates a need for mappings between these naming systems.

Second, the ODP-style object model has impact on the software engineering processes and requirements for programming environments. The designers of modern network programming tools should consider, whether single administration assumptions and inheritance based interworking solutions are sufficient in the long term.

The federated system model and the ODP-style object concepts allow division of software engineering process to multiple, independently working projects, as discussed above in Chapter 11.

The model also efficiently supports reuse of interfaces and object implementations, not only within a single organisation but also in a more global perspective. The meta-information facilities can be exploited as project management tools. Type repositories can be exploited as a source of reusable interfaces, and traders can be used as a source of reusable object code. Type repositories and traders can even be used as the compiler information base for megaprogramming based systems.

Third, trading services require further conceptual support, especially service type taxonomies. The work on standardising suitable service types is going to continue. The business models for CORBA and TINA both produce specifications for business objects, including banking services, and brokerage services for various merchandises. Similarly, application areas like CSCW should have a set of common services, like co-authoring tools, and video-conferencing systems. Finally, the trading functionality and the channel structure should be supported by a federable security system.

References

[1] ALANKO, T., KESKINEN, J., KUTVONEN, P., MUTKA, M., AND TIENARI, M. The AHTO Project: Software Technology for Open Distributed Processing. Tech. rep., Department of Computer Science, University of Helsinki, 1989. Report A-1989-4.

[2] ALZON, P., TOTHESAN, P., HUBERT, M., ATHANASSIOU, E., AND HOANG VAN, A. *ABS (Architecture for Information Brokerage Service) Broker Business Model*. ACTS AC206, 1996. http://b5www.deteberkom.de/ABS/D23.htm.

[3] ARCHITECTURE PROJECTS MANAGEMENT LIMITED. *ANSAware 3.0. Implementation Manual*, Feb. 1991. APM Document RM.097.01.

[4] ARCHITECTURE PROJECTS MANAGEMENT LIMITED. *ANSA Phase III 1992-1996*, Feb. 1992.

[5] ATKINSON, R. Security Architecture for the Internet Protocol. Tech. rep., Internet Engineering Task Force, Aug. 1995. RFC1825.

[6] BARR, W., BOYD, T., AND INOUE, Y. The TINA Initiative. *IEEE Communications Magazine* (Mar. 1993), 70 – 76.

[7] BEARMAN, M. ODP-Trader. In *Open Distributed Processing* (Berlin, Germany, 1993), J. de Meer, B. Mahr, and O. Spaniol, Eds., North-Holland, pp. 37 – 51.

[8] BEARMAN, M., AND RAYMOND, K. Federating Traders: An ODP Adventure. In *International IFIP Workshop on Open Distributed Processing* (Oct. 1991), North-Holland, pp. 125 – 141.

[9] BEARMAN, M., AND RAYMOND, K. Contexts, Views and Rules: An Integrated Approach to Trader Contexts. In *Open Distributed Processing* (Berlin, Germany, 1993), J. de Meer, B. Mahr, and O. Spaniol, Eds., North-Holland, pp. 181 – 191.

[10] BEASLEY, M., CAMERON, J., GIRLING, G., HOFFNER, Y., VAN LINDEN, R., AND THOMAS, G. *Establishing co-operation in federated systems*, Oct. 1994. APM.1128.02.

[11] BEITZ, A., AND BEARMAN, M. An ODP Trading Service for DCE. In *First International Workshop on Services in Distributed and Networked Environments (SDNE)* (Prague, Czech Republic, June 1994), IEEE Computer Society Press, pp. 42 – 49.

[12] BEITZ, A., KING, P., AND RAYMOND, K. Is DCE a Support Environment for ODP? In *Open Distributed Processing, II* (1994), J. de Meer, B. Mahr, and S. Storp, Eds., North-Holland, pp. 217 – 231.

[13] BELLCORE. *Asynchronous Transfer Mode (ATM) and ATM Adaptation Layer (AAL) Protocols Generic Requirements*, 1992. Technical Advisory TA-NWT-0001113, Issue 1.

[14] BENFORD, S., AND LEE, O.-K. Collaborative Naming in Distributed Systems. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993), 67 – 74.

[15] BERNSHAD, B. N., ZEKAUSKAS, M. J., AND SAWDON, W. A. The Midway Shared Memory System. In *IEEE COMPCON Conference* (1993), pp. 528 – 537.

[16] BERNSTEIN, P. A. Middleware: A Model for Distributed System Services. *Communications of the ACM 39*, 2 (Jan. 1996), 86 – 98.

[17] BIRMAN, K. P., AND JOSEPH, T. A. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 47 – 76.

[18] BIRRELL, A. D., AND NELSON, G. J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39 – 59.

[19] BJÖRKLUND, T. Integration and Maintenance of DRYAD Trading Software. Software project exercise C-1995-58, University of Helsinki, Department of Computer Science, 1995.

[20] BLAIR, G., AND STEFANI, J.-B. *Open Distributed Processing and Multimedia*. Addison-Wesley Publishing Company, 1997.

[21] BLAIR, G. S., AND PAPATHOMAS, M. The Case for Reflective Middleware. In *3rd Cabernet Plenary Workshop* (Rennes, France, Apr. 1997). Available at http://www.twente.research.ec.org/cabernet/workshops/3rd-plenary.html.

[22] BOWMAN, H., DERRICK, J., LININGTON, P., AND STEEN, M. Cross-viewpoint consistency in Open Distributed Processing. *Software Engineering Journal 11*, 1 (Jan. 1996), 44 – 57. Special issue on Viewpoints in Requirements Engineering.

[23] BROOKES, W., INDULSKA, J., BOND, A., AND YANG, Z. Interoperability of Distributed Platforms: a Compatibility Perspective. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 67 – 78.

[24] BRUNT, R. F. An introduction to OPENframework. *ICL Technical Journal 8*, 3 (1993).

[25] BULL, J., AND REES, O. A Framework for Federating Secure Systems. Tech. rep., APM, Jan. 1994. APM.1006.01. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1006.01.html.

[26] BURGER, C. Cooperation policies for traders. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 208 – 218.

[27] CARDELLI, L., AND WEGNER, P. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys 17*, 4 (Dec. 1985), 471 – 522.

[28] CASTAGNA, G. Covariance and contravariance: conflict without cause. *ACM Transactions on Programming Languages and Systems 17*, 3 (1995), 431 – 447.

[29] CATTELL, R. G. G., Ed. *Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1994.

[30] CHANG, R. N., AND RAVISHANKAR, C. V. A Service Acquisition Mechanism for the Client-Service Model in Cygnus. In *The 11th International Conference on Distributed Computer Systems* (Arlington, Texas, USA, May 1991), pp. 90 – 97.

[31] CHAVEZ, A., AND MAES, P. Kasbah: An agent marketplace for buying and selling goods. In *First International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology* (London, UK, Apr. 1996). Also available as http://agents.www.media.mit.edu/groups/agents/publications/kasbah-paam96.ps.gz.

[32] CHENG, W. C. H., AND JIA, X. A Reliable Trading Service in Open Distributed Systems. In *First International Workshop on Services in Distributed and Networked Environments (SDNE)* (Prague, Czech Republic, June 1994), IEEE Computer Society Press, pp. 122 – 129.

[33] CHIN, R. S., AND CHANSON, T. Distributed Object-Based Programming Systems. *ACM Computing Surveys 23*, 1 (Mar. 1991), 91 – 124.

[34] CORKILL, D. D. Countdown to Success: Dynamic Objects, GBB, and RADARSAT-1. *Communications of the ACM 40*, 5 (May 1997), 49 – 58.

[35] COULOURIS, G. F., DOLLIMORE, J., AND KINDBERG, T. *Distributed Systems, Concepts and Design*, 2nd ed. Addison-Wesley Publishing Company, 1994.

[36] CRISTIAN, F. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing 4*, 4 (1991).

[37] DANG TRAN, F., PERBASKINE, V., AND STEFANI, J.-B. Binding and streams: the ReTINA approach. In *TINA'96 Conference: The Convergence of Telecommunications and Distributed Computing Technologies* (Heidelberg, Germany, Sept. 1996), VDE, pp. 101 – 113.

[38] DANTHINE, A., AND BONAVENTURE, O. From best effort to enhanced QoS. Tech. rep., University of Liege, 1993. CEC Deliverable No. R2060/ULg/CIO/DS/P/004/b1, also ISO/IEC JTC1/SC6/WG4/N827.

[39] DANZIG, P. B., OBRACZKA, K., AND KUMAR, A. An Analysis of Wide-Area Name Server Traffic: A Study of the Internet Domain Name System. In *ACM SIGCOMM Conference* (Aug. 1992), pp. 281 – 292.

[40] DASGUPTA, P. Resource location in Very Large Networks. In *First International Workshop on Services in Distributed and Networked Environments (SDNE)* (Prague, Czech Republic, June 1994), IEEE Computer Society Press, pp. 156 – 163.

[41] DASGUPTA, P., LEBLANC, R. J., AHAMAD, M., AND RAMACHANDRAN, U. The Clouds Distributed Operating System. *IEEE Computer 24*, 11 (Nov. 1991), 34 – 44.

[42] DE PAULA LIMA, L. A., AND MADEIRA, E. R. M. A Model for a Federated Trader. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 173 – 184.

[43] DENNING, P. J. The Working Set Model for Program Behaviour. *Communications of the ACM 11*, 5 (May 1968), 323 – 333.

[44] DESCHREVEL, J.-P. The ANSA Model for Trading and Federation. Tech. Rep. APM.1005.01, APM, July 1993. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1005.01.html.

[45] DETOLEDO, M. B. F., AND BLAIR, G. S. Transaction Support for an ANSA-Based Platform. In *The 14th International Conference on Distributed Computing Systems* (Potznan, Poland, 1994), pp. 209 – 216.

[46] DIGITAL EQUIPMENT CORPORATION. *AltaVista Search Service*. http://www.altavista.digital.com/.

[47] DISTRIBUTED SYSTEMS TECHNOLOGY CENTRE. *DSTC Trading Object Service – An OMG Conformant Trading Service Implementation*, 1996. http://www.dstc.edu.au/AU/projects/corba-trader/fact_sheet.html.

[48] DOUGLIS, F., AND OUSTERHOUT, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software-Practice & Experience 21*, 8 (Aug. 1991), 757 – 785.

[49] DUSKA, B. M., MARWOOD, D., AND FEELEY, M. J. The Measured Access Characteristics of World-Wede-Web Client Proxy Caches. In *USENIX Symposium on Internet Technologies and Systems* (Dec. 1997). Also available as ftp://ftp.cs.ubc.ca/pub/local/techreports/1997/TR-97-16.ps.gz.

[50] EDWARDS, N. An ANSA Analysis of Open Dependable Distributed Computing. Tech. Rep. APM.1149.03, APM, Oct. 1994. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1149.03.html.

[51] EUROPEAN UNION ACTS PROJECT (ONYX INTERNET, COUNTY DURHAM TEC, NEWCASTLE UNIVERSITY, CEDCAMERA, HELSINKI TELEPHONE CORPORATION). *Project COBRA: Common Open Brokerage System*. http://zeus.gmd.de/hci/projects/cobra/.

[52] FAROOQUI, K., LOGRIPPO, L., AND DE MEER, J. The ISO Reference Model for Open Distributed Processing – An Introduction. *Journal of Computer Networks and ISDN Systems 27*, 8 (July 1995), 1215 – 1229.

[53] FAYAD, M. E., AND SCHMIDT, D. Object-Oriented Application Frameworks. *Communications of the ACM 40*, 10 (Oct. 1997), 32 – 38.

[54] FINKELSTEIN, D., ACTON, D., COATTA, T., HUTCHINGSON, N., AND NEUFELD, G. Object properties in Raven System. In *First International Workshop on Services in Distributed and Networked Environments* (1994), IEEE Technical Committee on Distributed Processing, pp. 502 – 509.

[55] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The SSL Protocol Version 3.0. Tech. rep., Internet Engineering Task Force, 1996. Internet Draft, Expires September 1996.

[56] FRIDAY, A., BLAIR, G. S., CHEVERST, K. W. J., AND DAVIES, N. Extensions to ANSAware for advanced mobile applications. In *IFIP/IEEE International Conference on Distributed Platforms* (Feb. 1996), Chapman & Hall, pp. 29 – 43.

[57] GAMMA, E., AND ET.AL. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[58] GARCÍA, F., HUTCHISON, D., MAUTHE, A., AND YEADON, N. QoS support for distributed multimedia communications. In *IFIP/IEEE International Conference on Distributed Platforms* (Dresden, Germany, Feb. 1996), Chapman & Hall, pp. 463 – 477.

[59] GARRAHAN, J., RUSSO, P., KITAMI, K., AND KUNG, R. Intelligent Network Overview. *IEEE Communications Magazine 31*, 3 (Mar. 1993), 30 – 36.

[60] GAY, V., LEYDEKKERS, P., AND HUIS IN 'T VELD, R. Specification of Multiparty Audio and Video Interaction. *Journal of Computer Networks and ISDN Systems 27*, 8 (July 1995), 1247 – 1262.

[61] GEIHS, K., AND FARSI, R. Service Trading in Electronic Markets. *International Journal in Electronic Markets 7*, 3 (1997), 25 – 28.

[62] GELERNTER, G. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan. 1985), 80 – 112.

[63] GIBBONS, P. B. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering 13*, 1 (Jan. 1987), 77 – 87.

[64] GOSCHINSKI, A. *Distributed Operating Systems: The Logical Design*. Addison-Wesley Publishing Company, 1991.

[65] GOSCINSKI, A. Supporting User Autonomy and Object Sharing in Distributed Systems: The RHODOS Trading Service. In *The First International Symposium on Autonomous Decentralized Systems* (Kawasaki, Japan, 1993).

[66] HALL, J., AND TSCHICHHOLZ, M. Management in a heterogeneous broadband environment. In *Integrated Network Management* (1991), I. Krishnan and W. Zimmer, Eds., vol. II, Elsevier Science Publisher.

[67] HAMBERG, M. ODP-viitemallin näkökulmakuvaukset – näkökulmat avoimen järjestelmän mallinnusmenetelmänä (ODP viewpoint specifications as a methodology). Master's thesis, University of Helsinki, Department of Computer Science, Feb. 1997. C-1997-30. In Finnish.

[68] HAMBERG, M., JOKINEN, M., SIVONEN, T., AND YIN, H. Service invocation function. Software project exercise C-1995-38, University of Helsinki, Department of Computer Science, 1995.

[69] HAMILTON, S. E-Commerce for the 21st Century. *IEEE Computer 31*, 2 (May 1997), 44 – 47.

[70] HARRISON, W., AND OSSHER, H. Subject-Oriented Programming (A Critique of Pure Objects). In *Eight Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)* (Washington, DC, USA, Oct. 1993), A. Paepcke, Ed., pp. 411 – 426. ACM SIGPLAN Notices 28(10).

[71] HATAKKA, J., HYVÄTTI, J., KOSKINEN, I., LILJEBERG, M., MOILANEN, M., AND YLITALO, J. Nereid – DRYAD-meklarin ylläpitosovellus (Nereid – A graphical management interface to DRYAD Trader). Software project exercise C-1993-69, University of Helsinki, Department of Computer Science, 1993. In Finnish.

[72] HAYER, R., AND SCHLICHTING, R. D. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering 13*, 12 (Dec. 1987), 1254 – 1264.

[73] HELM, R., HOLLAND, I., AND GANGOPADHYAY, D. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications* (Vancouver, Oct. 1990), ACM, pp. 169 – 180.

[74] HERBERT, A. An Overview of ANSA. *IEEE Network 8*, 1 (Jan. 1994), 18 – 23.

[75] HERBERT, A. Distributing Objects. Technical Report APM.1009.01, APM, May 1994.

[76] HOFFNER, Y. A designers' introduction to trading. Technical Report APM.1387.01, APM, Dec. 1994.

[77] HOFFNER, Y. Interoperability and Distributed Platform Design. In *IFIP/IEEE International Conference on Distributed Platforms* (Dresden, Germany, Feb. 1996), Chapman & Hall, pp. 342 – 356.

[78] HOFFNER, Y., AND CRAWFORD, B. Federation and Interoperability. Tech. Rep. APM.1514.01, ANSA, APM, Oct. 1995. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1514.01.html.

[79] HOFFNER, Y., AND CRAWFORD, B. Using Interception to Create Domains in Distributed Systems. In *IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms* (Toronto, Canada, May 1997), J. Rolia, J. Slonim, and J. Botsford, Eds., Chapmann & Hall, pp. 251 – 263.

[80] IBM CORPORATION. *User's Guide to OS/2 Warp*, 1994. Part No. 19H5128.

[81] IBM CORPORATION. IBM Open Blueprint. WWW information page, http://www.software.ibm.com/openblue/openftp.htm, Oct. 1997.

[82] IKV++ GMBH INFORMATIONS- UND KOMMUNIKATIONSTECHNOLOGIE. *TOI – OMG Trading Object Service Implementation*, Mar. 1997. http://www.ikv.de/products/ftoi.html/.

[83] INDULSKA, J., BEARMAN, M., AND RAYMOND, K. A Type Management System for an ODP Trader. In *Open Distributed Processing* (Berlin, Germany, 1993), J. de Meer, B. Mahr, and O. Spaniol, Eds., North-Holland, pp. 169 – 180.

[84] INFORMATION SCIENCES INSTITUTE, UNIVERSITY OF SOUTHERN CALIFORNIA. *FAST Electronic Broker*. http://www.FASTXchange.com/.

[85] ISO/IEC JTC1. *Electronic data interchange for administration, commerce and transport (EDIFACT) – Application level syntax rules*. International Standard IS9375, 1987.

[86] ISO/IEC JTC1. *Information Processing Systems – Open Systems Interconnection, Systems Management Overview*, 1992. IS10040 | X.701.

[87] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trader*, May 1992. Working document ISO/IEC JTC1/SC21 WG7 N7074.

[88] ISO/IEC JTC1. *A formalised computational Model of the ODP Trader using SDL'92*, July 1994. DIN Input Contribution SC21 WG7 N924.

[89] ISO/IEC JTC1. *Information Technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 1995. IS8824-1.

[90] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection – Systems management – Open Distributed Management Architecture*, 1996. DIS13244.

[91] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing*, 1996. IS10746.

[92] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 2: Foundations*, 1996. IS10746-2.

[93] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 3: Architecture*, 1996. IS10746-3.

[94] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 1: Overview*, 1996. IS10746-1.

[95] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 4: Architectural Semantics*, 1996. IS10746-4.

[96] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trading function. Part 1: Specification*, 1997. IS13235-1.

[97] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Type repository function*, 1997. CD14746.

[98] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – ODP Naming framework*, Jan. 1997. DIS14771.

[99] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – Protocol Support for Computational Interactions*, July 1997. Working draft.

[100] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trading function. Part 3: Provision of Trading Function using OSI Directory Service*, 1997. IS13235-3.

[101] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trading function. Part 2: Implementation conformance statements and test cases*, 1997. Project to be cancelled.

[102] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – ODP Interface References and Binding*, Jan. 1998. FCD14753, editor draft for DIS.

[103] ISO/IEC JTC1. *Working draft for Quality of Service in ODP*, Jan. 1998. SC21/WG7 N10979.

[104] ISWG/AISG OSE EXPERT GROUP. *NATO Open system environment: glossary of terms*, June 1996. Version 3. Available at http://www.nacisa.nato.int/nose/gloss1.htm.

[105] ITU-T. *Rec. X.500 Information Processing – Open Systems Interconnection – The Directory*. 1992.

[106] ITU-T. *Rec. M.3010: Principles for a Telecommunications Management Network*, 1992.

[107] ITU-T. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing*, 1996. Recommendations X.901 - X.904.

[108] ITU-T. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trading function. Part 1: Specification*, 1996. Recommendation X.950.

[109] JONES, A. K. The Object Model: The Conceptual Tool for Structuring Software. In *Operating Systems – An Advanced Course*, Lecture Notes in Computer Science. Springer Verlag, 1978, pp. 8 – 16.

[110] KÄHKIPURO, P., KUTVONEN, L., AND MARTTINEN, L. Federated naming in an ODP environment. In *IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms* (Toronto, Canada, May 1997), J. Rolia, J. Slonim, and J. Botsford, Eds., Chapmann & Hall, pp. 314 – 326.

[111] KÄHKIPURO, P., MARTTINEN, L., AND KUTVONEN, L. Reaching Interoperability through ODP type framework. In *TINA'96 Conference: The Convergence of Telecommunications and Distributed Computing Technologies* (Aug. 1996), VDE Verlag, pp. 283 – 284. Extended abstract.

[112] KÄHKIPURO, P., MARTTINEN, L., AND KUTVONEN, L. Reaching Interoperability through ODP type framework. Tech. Rep. C-1996-96, Department of Computer Science, University of Helsinki, 1996.

[113] KEKKONEN, S., KORKIAKOSKI, J., PARTANEN, R., SIIVOLA, V., AND VEPSÄLÄINEN, H. Arttu – DRYAD-meklarin käyttöliittymä (Artemis – A graphical user interface to DRYAD Trader). Software project exercise C-1993-43, University of Helsinki, Department of Computer Science, 1993. In Finnish.

[114] KELLER, L. Trading of Complex Services in Distributed Systems. In *The Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management* (Toulouse, 1994).

[115] KICZALES, G., AND PAEPCKE, A. *Open Implementations and Metaobject Protocols*, 1996.

[116] KITSON, B. Intercessory Objects within Channels. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 233 – 244.

[117] KONO, K., KATO, K., AND MASUDA, T. Smart Remote Procedure Calls: Transparent treatment of remote pointers. In *The 14th International Conference on Distributed Computing Systems* (Poznan, Poland, June 1994), IEEE Computer Society, pp. 142 – 151.

[118] KOVACS, E. Trading and Management of Distributed Applications: Main Tasks for future Distributed Systems. In *Aachen Workshop New Concepts for Open Distributed Processing* (Aachen, Germany, 1994). In German.

[119] KOVACS, E. Advanced Trading Services Through Mobile Agents. In *International Workshop on Trends in Distributed Systems* (Aachen, Germany, 1996), Springer Verlag. In German.

[120] KOVACS, E., AND BURGER, C. Project description MELODY – Management environment for Large Open Distributed Systems. Tech. Rep. Technical report 8/95, University of Stuttgart, 1995. In German.

[121] KOVACS, E., AND WIRAG, S. Trading and Distributed Application Management: An Integrated Approach. In *The Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'94)* (Toulouse, France, 1994).

[122] KUEPPER, A., POPIEN, C., AND MEYER, B. Service Management using up-to-date quality properties. In *IFIP/IEEE International Conference on Distributed Platforms* (Dresden, Germany, Feb. 1996), A. Schill, C. Mittasch, O. Spaniol, and C. Popien, Eds., Chapmann & Hall, pp. 342 – 356.

[123] KUTVONEN, L. Achieving interoperability through ODP trading function. In *The second international symposium on Autonomous Decentralized Systems* (Phoenix, Arizona, 1995), IEEE Computer Society Press, pp. 63 – 69.

[124] KUTVONEN, L. Supporting transition to open, heterogeneous computing environment. In *TINA'95: International Telecommunications and Distributed Computing – from Concepts to Reality* (Melbourne, Australia, 1995), pp. 55 – 66.

[125] KUTVONEN, L. Overview of the DRYAD trading system implementation. In *IFIP/IEEE International Conference on Distributed Platforms* (Dresden, Germany, Feb. 1996), Chapman & Hall, pp. 314 – 326.

[126] KUTVONEN, L. *The Role of Trading function in Open Distributed Processing Environment*. Department of Computer Science, University of Helsinki, 1996. Licentiate thesis. C-1996-84.

[127] KUTVONEN, L. Management of Application Federations. In *International IFIP Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)* (Cottbus, Germany, Sept. 1997), H. Konig, K. Geihs, and T. Preuss, Eds., Chapmann & Hall, pp. 33 – 46.

[128] KUTVONEN, L. Why CORBA systems cannot federate? In *OMA/ODP Workshop* (Cambridge, UK, Nov. 1997).

[129] KUTVONEN, L., AND KUTVONEN, P. Broadening the User Environment with Implicit Trading. In *Open Distributed Processing* (Berlin, Germany, 1993), J. de Meer, B. Mahr, and O. Spaniol, Eds., North-Holland, pp. 157 – 168.

[130] KUTVONEN, P., AND KUTVONEN, L. Measured Performance of a Special-purpose Database Engine for Distributed System Software. *Computer Communications Journal* (Jan. 1996), 22 – 29. Special Number on High Speed Networks and Open Distributed Platforms Workshop.

[131] LADDAGA, R., AND VEITCH, J. Dynamic Object Technology. *Communications of the ACM 40*, 5 (May 1997), 37 – 38.

[132] LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. The ObjectStore database system. *Communications of the ACM 34*, 10 (Oct. 1991), 50 – 63.

[133] LAMERSDORF, W., MERZ, M., AND MÜLLER-JONES, K. Middleware Support for Open Distributed Applications. In *High Speed Networks and Open Distributed Platforms* (St. Petersburg, Russia, June 1995).

[134] LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (1978), 558 – 565.

[135] LAMPSON, B. W. Designing a Global Name Service. In *Fifth Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Canada, Aug. 1986), pp. 1 – 10.

[136] LEPREAU, J., HIBLER, M., FORD, B., AND LAW, J. In-kernel servers on Mach 3.0: Implementation and performance. In *Third USENIX Mach Symposium* (USA, Apr. 1993), pp. 39 – 55. Also http://www.usenix.org/publications/library/proceedings/mach3/lepreau.html.

[137] LI, G. An Overview of Real-Time ANSAware 1.0. Tech. Rep. APM.1285.01, ANSA, APM, Mar. 1995. For Distributed Systems Engineering Journal. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1285.01.html.

[138] LI, K. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, 1986.

[139] LI, K., AND HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems 7*, 1 (Nov. 1989), 321 – 359.

[140] LININGTON, P. RM-ODP: The architecture. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 15 – 33.

[141] MACARTNEY, A. J., AND BLAIR, G. S. Flexible trading in distributed systems. *Computer Networks and ISDN Systems 25*, 15 (1992), 145 – 157.

[142] MAES, P., AND MARDI, D., Eds. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[143] MÄKELÄ, J. Paperclip stopped trains in Finland. *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems 19*, 10 (Apr. 1997). http://catless.ncl.ac.uk/Risks/19.10.html#subj1.

[144] MANOLA, F., HEILER, S., AND GEORGAKOPOULOUS, D. Distributed Object Management. *International Journal of Intelligent and Cooperative Information Systems 1*, 1 (June 1992), 5 – 42.

[145] MASTERCARD AND VISA. *SET Secure Electronic Transaction Specification, Book 1: Business Description*, May 1997. http://www.visa.com/nt/ecomm/set/set_bk1.pdf.

[146] MCCONNEL, S. The OMG/CommerceNet Joint Electronic Commerce Whitepaper. Tech. rep., OMG and CommerceNet, July 1997. OMG/EC/97-06-09.

[147] MCCONNELL, S., MERZ, M., MAESANO, L., AND WITTHAUT, M. An Open Architecture for Electronic Commerce. Tech. rep., OMG and CommerceNet, Feb. 1997.

[148] MERZ, M. *Elektronishe Märkte im Internet*. Thomson Publishing, 1996.

[149] MERZ, M., MÜLLER, K., AND LAMERSDORF, W. Service Trading and Mediation in Distributed Computing Systems. In *The 14th International Conference on Distributed Computing Systems (ICDCS '94)* (Poznan, Poland, 1994), IEEE Computer Society Press, pp. 440 – 457.

[150] MEYER, B., HEINEKEN, M., AND POPIEN, C. Performance Analysis of Distributed Applications with ANSAmon. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 307 – 319.

[151] MEYER, B., AND POPIEN, C. Flexible management of ANSAware applications. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 271 – 282.

[152] MICROSOFT CORPORATION. *Microsoft Windows User's Guide*, 1992. Document No. PC21669-0492.

[153] MICROSOFT CORPORATION. *Microsoft Windows NT Server System Guide*, 1994. Document No. 58005-0794.

[154] MICROSOFT CORPORATION. *The Distributed Component Object Model. A business overview.* USA, 1996.

[155] MILOSEVIC, Z. *Enterprise Aspects of Open Distributed Systems*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1995.

[156] MILOSEVIC, Z., ARNOLD, D., AND O'CONNOR, L. Inter-enterprise Contract Architecture for Open Distributed Systems: Security Requirements. In *WET ICE'96 Workshop on Enterprise Security* (Stanford, USA, June 1996).

[157] MILOSEVIC, Z., BERRY, A., BOND, A., AND RAYMOND, K. Supporting Business Contracts in Open Distributed Systems. In *Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)* (Whistler, Canada, June 1995), IEEE Computer Society Press, pp. 60 – 67.

[158] MITCHELL, J. C. Coersion and type inference. In *The 11th ACM Symposium on Principles of Programming Languages* (Salt Lake City, USA, Jan. 1984), pp. 175 – 185.

[159] MOCKAPETRIS, P. Domain names - concepts and facilities. Tech. rep., Internet Engineering Task Force, Nov. 1987. RFC1034.

[160] MOCKAPETRIS, P. Domain names - implementation and specification. Tech. rep., Internet Engineering Task Force, Nov. 1987. RFC1035.

[161] MÜLLER, S., MÜLLER-JONES, K., LAMERSDORF, W., AND TU, T. Global Trader Cooperation in Open Service Markets. In *International Workshop on Trends in Distributed Systems* (Aachen, Germany, Oct. 1996), Springer Verlag.

[162] MÜLLER-JONES, K., MERZ, M., AND LAMERSDORF, W. The TRADEr: Integrating Trading Into DCE. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 476 – 487.

[163] NAJM, E., AND STEFANI, J. A formal semantics for the ODP computational model. *Journal of Computer Networks and ISDN Systems 27*, 8 (July 1995), 1305 – 1329.

[164] NEEDHAM, R. M. Names. In *Distributed Systems*, S. Müllender, Ed. ACM Press, Frontier Series, 1989, ch. 5.

[165] NEUMAN, B. C., AND TS'O, T. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications 32*, 9 (Sept. 1994), 33 – 38.

[166] NI, Y., AND GOSCINSKI, A. Trader cooperation to enable object sharing among users of homogeneous systems. *Computer Communications 17*, 3 (Mar. 1994), 218 – 229.

[167] OBJECT MANAGEMENT GROUP. *OMG White Paper on Security*, Apr. 1994. Document 94-04-16. Also http://www.omg.org/docs/1994/94-04-16.ps.

[168] OBJECT MANAGEMENT GROUP. *CORBA Security*, Dec. 1995. Document 95-12-1. Also http://www.omg.org/docs/1995/95-12-01.ps.

[169] OBJECT MANAGEMENT GROUP. *Object Services RPF 5*, July 1995. OMG TC Document 95-6-18.

[170] OBJECT MANAGEMENT GROUP. *OMG Business Application Architecture*, Mar. 1995. http://www.tiac.net/users/jsuth/oopsla/bowp2.html.

[171] OBJECT MANAGEMENT GROUP. *Common Secure Interoperablity*, July 1996. Document orbos/96-06-20. Also http://www.omg.org/docs/orbos/96-06-20.ps.

[172] OBJECT MANAGEMENT GROUP. *Multiple Interfaces and Composition Request for Proposals*, Jan. 1996. Document ORB/96-01-04.

[173] OBJECT MANAGEMENT GROUP. *OMG RFP5 Submission: Trading Object Service*, May 1996. OMG Document orbos/96-05-06. Also http://www.omg.org/docs/orbos/96-05-06.ps.

[174] OBJECT MANAGEMENT GROUP. *Common Facilities RFP-5: Meta-Object Facility*, 1997. OMG TC Document cf/96-05-02.

[175] OBJECT MANAGEMENT GROUP. *UML Semantics*, Sept. 1997. OMG Document No. ad/97-08-04 (Revision 1.1.). Also http://www.omg.org/pub/docs/ad/97-08-04.ps.

[176] OBJECT MANAGEMENT GROUP AND X/OPEN. *The Common Object Request Broker: Architecture and Specification*, Nov. 1993. OMG Document No. 91.12.1. (Revision 1.1.). Also http://www.omg.org/docs/1991/91-12-01.ps.

[177] OBJECT MANAGEMENT GROUP AND X/OPEN. *The Common Object Request Broker: Architecture and Specification*, May 1996. OMG Document No. 91.12.1. (Revision 2.1.). Also http://www.omg.org/docs/interop/96-05-01.ps.

[178] OINONEN, S. Co-operation of Independent Trader Implementations. Master's thesis, University of Helsinki, Department of Computer Science, Aug. 1996. C-1996-73.

[179] OPEN SOFTWARE FOUNDATION. *An Analysis of the OSF/1 Operating System and UNIX System V Release 4*, Jan. 1992. White paper.

[180] OPEN SOFTWARE FOUNDATION. *OSF DCE User's Guide and Reference*, 1994.

[181] OSKIEWICZ, E., AND EDWARDS, N. A Model for Interface Groups. Tech. Rep. APM.1002.01, APM, May 1994. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1002.01.html.

[182] OTWAY, D. Streams and Signals. Tech. Rep. APM.1393.02, ANSA, APM, Jan. 1995. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1393.02.html.

[183] OTWAY, D. The ANSA binding model. Tech. Rep. APM.1392.01, ANSA, APM, Jan. 1995. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1392.01.html.

[184] ÖZSU, M. T., DAYAL, U., AND VALDURIEZ, P. An introduction to distributed object management. In *Distributed Object Management* (1994), M. T. Özsu and P. Dayal, U. Valduriez, Eds., Morgan Kaufmann Publishers, pp. 1 – 25.

[185] PARHAR, A., AND RUMSEWICZ, M. A preliminary investigation of performance issues associated with Freephone service in a TINA consistent network. In *TINA'95: International Telecommunications and Distributed Computing – from Concepts to Reality* (Melbourne, Australia, Feb. 1995), pp. 431 – 444.

[186] PARNAS, D. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM 15* (Dec. 1972), 1053 – 1058.

[187] PIERCE, B. C. Bounded quantificaiton is undecidable. *Information and Computation 112*, 1 (July 1994), 131 – 165.

[188] PLUMMER, D. The Ethernet address resolution protocol. Tech. rep., Internet Engineering Task Force, Nov. 1982. RFC-826.

[189] POPESCU-ZELETIN, R., TSCHAMMER, V., AND TSCHICHHOLZ, M. 'Y' Distributed Application Platform. *Computer Communication 13*, 6 (1991), 366 – 375.

[190] POPIEN, C., SCHÜRMANN, G., AND WEISS, K.-H. *Verteilte Verarbeitung in Offenen Systemen: das ODP-referenzenmodell*, vol. 1 of *Thomson's Aktuelle Tutorien*. Thomson Publishing, 1994. In German.

[191] PRATTEN, A. W., HONG, J. W., BAUER, M. A., BENNETT, J. M., AND LUTFIYYA, H. A Resource Management System Based on the ODP Trader Concepts and X.500. In *The Fourth International Symposium on Integrated Network Management* (Santa Barbara, USA, May 1995), pp. 118 – 130.

[192] PRATTEN, A. W., HONG, J. W., BENNETT, J. M., BAUER, M. A., AND LUTFIYYA, H. Design and Implementation of a Trader-Based Resource Management System. In *The 1994 CAS Conference* (Toronto, Canada, Oct. 1994), pp. 130 – 141.

[193] PRESSMAN, R. S. *Software Engineering – A Practioner's Approach*, 3rd ed. McGraw-Hill, 1992.

[194] PUDER, A., MATKWITZ, S., GUDERMANN, F., AND GEIHS, K. AI-based Trading in Open Distributed Environments. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 157 – 170.

[195] RASHID, R., JULIN, D., ORR, D., SANZI, R., BARON, R., FORIN, A., GOLUB, D., AND JONES, M. Mach: A System Software Kernel. In *The 34th Computer Society International Conference COMPCON 89* (Feb. 1989).

[196] RATCLIFF, J. Pattern Matching by Gestalt. *Doctor Dobb's Journal*, 181 (1988).

[197] RAYMOND, K. Reference Model of Open Distributed Processing (RM-ODP): Introduction. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 3 – 14.

[198] REES, O. The ANSA Computational Model. Technical Report APM.1001.01, APM, Jan. 1995.

[199] REES, O., EDWARDS, N., MADSEN, M., BEASLEY, M., AND MCCLENAGHAN, A. A Web of Distributed Objects. In *Fourth International World Wide Web Conference* (Boston, Dec. 1995).

[200] RICHARDSON, J., AND SCHWARZ, P. Aspects: Extending Objects to Support Multiple Independent Roles. In *The 1991 ACM SIGMOD Conference* (Denver CO, May 1991), pp. 298 – 307.

[201] RITCHIE, D., AND THOMPSON, K. The UNIX Time Sharing System. *Bell Systems Technical Journal 57*, 6 (July 1978).

[202] RUDIN, S. Templates, types and classes in open distributed processing. *British Telecom Technology Journal 11*, 3.

[203] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[204] RYMER, J. R. The Muddle in the Middle. *Byte* (Apr. 1996).

[205] SALAMON, A. *DNS Resources Directory*, 1996. http://www.dns.net/dnsrd/.

[206] SCHWARTZ, M. F. Resource Discovery in the Global Internet. Tech. rep., Department of Computer Science, University of Colorado, Boulder, Nov. 1991. CU-CS-555-91.

[207] SHILLING, J., AND SWEENEY, P. Three steps to views: Extending the object-oriented paradigm. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications* (New Orleans, Oct. 1989), pp. 353 – 361.

[208] SINNOTT, R. *An Architecture Based Approach to Specifying Distributed Systmes in LOTOS and Z*. PhD thesis, University of Stirling, 1997.

[209] SLOMAN, M., MAGEE, J., TWIDLE, K., AND KRAMER, J. An Architecture for Managing Distributed Systems. In *Fourth IEEE Workshop on Future Trends of Distributed Computing Systems* (Lisbon, Portugal, Sept. 1993), IEEE Computer Society Press, pp. 40 – 46.

[210] SOLEY, R. Overview of OMG. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995). Tutorial.

[211] SOLEY, R., AND STONE, C. M., Eds. *Object Management Architecture Guide*. John Wiley & Sons, Inc., June 1995. Revision 3.0.

[212] SPECTOR, A. Z., AND KAZAR, M. L. Wide Area Field Service and the AFS Experimental System. *Unix review 7*, 3 (Mar. 1989).

[213] SPEIRS-BRIDGE, A., RAMESH, S., ROSSITER, P., MEEHAN, A., MUGDAN, C., AND SILMAN, J. MSP – Practical Experiences in the Application of TINA. In *TINA'95: International Telecommunications and Distributed Computing – from Concepts to Reality* (Melbourne, Australia, Feb. 1995), pp. 685 – 702.

[214] SUN MICROSYSTEMS, INC. *RPC: Remote Procedure Call Protocol specification: Version 2*, 1988. RFC1057.

[215] SUN MICROSYSTEMS, INC. NFS: Network File System Protocol specification: Version 2. Tech. rep., Internet Engineering Task Force, 1989. RFC1094.

[216] SUN MICROSYSTEMS, INC. *Solaris 2.5 System Administration Guide*, 1995.

[217] SUN MICROSYSTEMS, INC. *Remote Method Invocation Specification*, Dec. 1996. http://java.sun.com/products/JDK/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

[218] SVOBODOVA, L. File Servers for network-based Distributed Systems. *ACM Computing Surveys 16*, 4 (1984), 353 – 398.

[219] SYCARA, K., DECKER, K., AND WILLIAMSON, M. Matchmaking and Brokering. In *Second International Conference on Multi-Agent Systems (ICMAS-96)* (Dec. 1996).

[220] TANENBAUM, A. S. *Modern Operating Systems*. Prentice Hall, 1992.

[221] TANENBAUM, A. S. *Distributed Operating Systems*. Prentice Hall, 1995.

[222] TANENBAUM, A. S., AND VAN RENESSE, R. Distributed Operating Systems. *ACM Computing Surveys 17*, 4 (Dec. 1985), 419 – 470.

[223] TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G. J., MULLENDER, S. J., JANSEN, A. J., AND VAN ROSSUM, G. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM 33*, 12 (Dec. 1990), 46 – 63.

[224] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Engineering Modelling Concepts (DPE Architecture)*, Dec. 1994. Document TB-NS.005-2.0-94. Also http://www.tinac.com/95/dpe/emc94-public.ps.

[225] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Engineering Modelling Concepts (DPE Kernel Specifications)*, Dec. 1994.

[226] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *TINA Management Architecture*, 1994. TB_GN.010_2.0_94.

[227] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *TINA Network Resource Information Model Specification*, 1994. TB_LR.011_2.0_94.

[228] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Computational Modelling Concepts*, Feb. 1995. Document TB-A2.HC.012-1.2-94. Also Also http://www.tinac.com/95/dpe/cmc94-public.ps.

[229] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Deployment Scenarios for Interworking*, Feb. 1995.

[230] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Domain types and basic Reference Points in TINA*, May 1995.

[231] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Overall concepts and principles of TINA*, Feb. 1995. Document TB-MDC.018-1.0-94. Also http://www.tinac.com/95/overall/public/Overall/overall.ps.

[232] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *Requirements upon TINA-C architecture*, Feb. 1995. Also http://www.tinac.com/95/overall/public/Requirement/requirements.ps.

[233] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORITUM (TINA-C). *TINA object definition language manual. Version 2.3*, July 1996. Document TR_NM.002_2.2_96. Also http://www.tinac.com/96/odl96_public.ps.

[234] TENENBAUM, J. M., CHOWDHRY, T. S., AND HUGHES, K. Eco System: An Internet Commerce Architecture. *IEEE Computer* (May 1997), 48 – 55.

[235] TIENARI, M. *Avoin hajautettu tietojenkäsittely (Open distributed computing)*. Department of Computer Science, University of Helsinki, 1997. Seminar. In Finnish.

[236] TIETOVIIKKO. Klemmari pysäytti junat (Paperclip stops trains). *Tietoviikko*, 14 (Apr. 1997), 3.

[237] TIETOVIIKKO. Nesteestä ja Meritasta tehtiin pornopalveluja: Hyökkäys nimipalvelimeen sotki yhtiöiden kotisivut pornokuvilla (Attack to name server replaced the contents of Nokia and Merita WWW-pages with pornographical material). *Tietoviikko*, 33 (Oct. 1997), 1, 5.

[238] TORVALDS, L. Linux: a Portable Operating System. Master's thesis, Department of Computer Science, University of Helsinki, Jan. 1997. Report C-1997-12.

[239] TSCHAMMER, V., HERZOG, H., AGHOUTANE, N., AND ERCAN, G. Implementing the OMG Trading Object Service – the TOI Product. In *IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms* (Toronto, Canada, May 1997), J. Rolia, J. Slonim, and J. Botsford, Eds., Chapmann & Hall, pp. 303 – 313.

[240] TSCHICHHOLZ, M., TSCHAMMER, V., AND DITTRICH, A. Integrated Approach to Open Distributed Management. *Computer Communications 19*, 1 (Jan. 1996), 76 – 87.

[241] VAN HALTEREN, A., LEYDEKKERS, P., AND KORTE, H. Specification and Realisation of Stream interfaces for the TINA-DPE. In *TINA'95: Integrating Telecommunications and Distributed Computing – from Concepts to Reality* (Melbourne, Australia, Feb. 1995), pp. 299 – 313.

[242] VAN LINDEN, R. An Overview of ANSA. Tech. Rep. APM.1000.01, APM, July 1993.

[243] VAN LINDEN, R. The ANSA Naming Model. Tech. Rep. APM.1003.01, ANSA, APM, July 1993. Also http://www.ansa.co.uk/phase3-doc-root/approved/APM.1003.01.html.

[244] VENETJOKI, T. Meklareiden yhteistoiminta ja sen toteutus DRYAD-projektissa (Cooperation of traders in DRYAD). Master's thesis, University of Helsinki, Department of Computer Science, Apr. 1994. C-1994-25. In Finnish.

[245] VIDMAN, K. Kuorman hajauttamisen liittäminen meklariin ja meklarin tietopalvelija hajautetussa ympäristössä (Load sharing and trading). Master's thesis, University of Helsinki, Department of Computer Science, June 1994. C-1995-55. In Finnish.

[246] VOGEL, A., BEARMAN, M., AND BEITZ, A. Enabling Interworking of Traders. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, Feb. 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 185 – 196.

[247] VOGEL, A., KERHERVÉ, B., VON BOCHMANN, G., AND GECSEI, J. Distributed Multimedia Application and Quality of Service – A Survey. *IEEE Multimedia 2*, 2 (Summer 1995), 10 – 19.

[248] WAUGH, A., AND BEARMAN, M. Designing an ODP Trader Implementation using X.500. In *The Third International Conference on Open Distributed Processing – Experiences with distributed environments* (Brisbane, Australia, 1995), K. Raymond and L. Armstrong, Eds., Chapmann & Hall, pp. 133 – 144.

[249] WEGNER, P. Why Integration is more powerful than Algorithms. *Communications of the ACM 40*, 5 (May 1997), 81 – 91.

[250] WESSELS, D. ICP and the Squid Web Cache. Tech. rep., National Laboratory for Applied Network Research, Aug. 1997. http://www.nlanr.net/~wessels/Papers/.

[251] WIEDERHOLD, G., WEGNER, P., AND CERI, S. Towards Megaprogramming. *Communications of the ACM 33*, 11 (Nov. 1992), 89 – 99.

[252] WOLFF, T., AND LÖHR, K.-P. Transparently programming heterogeneous distributed systems. In *IFIP/IEEE International Conference on Distributed Platforms* (Dresden, Germany, Feb. 1996), Chapman & Hall, pp. 399 – 411.

[253] WOLISZ, A., AND TSCHAMMER, V. Service Provider Selection in an Open Services Environment. In *Second IEEE Workshop on Future Trends of Distributed Computing Systems* (Cairo, Egypt, 1990), pp. 229 – 235.

[254] WOLISZ, A., AND TSCHAMMER, V. Dynamic Binding of Service Users and Service Providers in an Open Services Environment. In *International Computer Networks Conference* (Wroclaw, Poland, June 1991), pp. 91 – 105.

[255] WOLISZ, A., AND TSCHAMMER, V. Performance aspects of trading in open distributed systems. *Computer Communications 16*, 5 (May 1993), 277 – 287.

[256] X3H7. *X3H7 Object Model Features Matrix*. GTE Laboratories Incorporated, Feb. 1995.

[257] YANG, Z., AND VOGEL, A. Achieving Interoperability between CORBA and DCE Applications Using Bridges. In *IFIP/IEEE International Conference on Distributed Platforms* (Dresden, Germany, Feb. 1996), Chapman & Hall, pp. 144 – 155.

[258] YATES, M., TAKITA, W., DEMOUDEM, L., JANSSON, R., AND MULDER, H. *TINA Business Model and Reference Points*. Telecommunications Information Networking Architecture Consoritum (TINA-C), May 1997. Version 4.0. Also http://www.tinac.com/97/bm_rp.ps.

[259] YRJÄNÄINEN, V. M. Portieri DRYAD-meklarin toteutuksessa (Using Concierge in the replication of the DRYAD trader). Master's thesis, University of Helsinki, Department of Computer Science, Apr. 1994. C-1994-22, In Finnish.

[260] ZIMMERMANN, P. *The Official PGP User's Guide*. MIT Press, 1995.

APPENDICIES

Appendix A

English-Finnish ODP Dictionary

A

access transparency	saantituntumattomuus
action	toiminto, tapahtuma
activity	toiminnallisuus, aktiviteetti
announcement	ilmoitus
arbitration	sovittelu
attribute	attribuutti

B

basic engineering object	toteutusobjekti
behaviour	käyttäytyminen
binder	sidosobjekti
binding	sidos, sitominen

C

capsule	kapseli
causality information	suuntatieto
channel	kanava
class	luokka
client	asiakas
cluster	klusteri, rypäs
cluster manager	klusterin hallitsija
compatibility (behavioural compatibility)	yhteensopivuus
communication	kommunikointi, viestintä

C

community	yhteisö
compliance to a standard	standardinmukaisuus
composite object	rakenteinen objekti
computational viewpoint	toimintonäkökulma
compound binding	koosteinen sidonta
configuration	konfiguraatio
conformance	yhteensopivuus
contract	sopimus
contractual context	(yhteinen) sopimustila
cooperation	yhteistoiminta (yleiskielen käsite), vrt. interworking, interoperation
creation	luonti

D

data	data, informaation esitysmuoto
deactivation	passivointi
decomposition	osiin jakaminen, yksityiskohtaisemman rakenteen esittäminen
derived class	johdettu luokka
distribution transparency	hajautustuntumattomuus
domain	alue, hallintoalue
dynamic schema	dynaaminen skeema

E

enabled behaviour	mahdollistetu käyttäytyminen
engineering viewpoint	toteutusnäkökulma, rakennenäkökulma
engineering interface	sidospiste
enterprise viewpoint	tavoitenäkökulma
entity	entiteetti, (reaalimaailman) olio
environment	ympäristö
environment contract	ympäristösopimus
epoch	(ajan)jakso, vaihe
error	virhe
establishing behaviour	aloitus
event	(erityis)tapahtuma
export	(palvelun) tarjoaminen
exporter	tarjoaja

F

failure	epäonnistuminen, vikaantuminen, sopimusrikko
failure transparency	vikatuntumattomuus
fault	häiriö
federation	federaatio
flow	virta

I

identifier	(yksikäsitteinen) tunniste
import	palvelukysely
importer	(palvelun) kysyjä
information	informaatio, data-esityksen semanttinen merkitys
information viewpoint	informaationäkökulma
instance	instanssi, ilmentymä
instantiation	instantiointi
interaction	vuorovaikutus
interceptor	interseptori, muunnin, valvontaobjekti
interchange reference point	siirrettävyyden testauspiste
interface	rajapinta
interface reference	rajapintaviite
interface signature	rajapinnan rakenne
internal action	yksityinen tapahtuma, sisäinen tapahtuma
interoperation	keskinäistoiminta (termi, vrt. cooperation, interworking)
interrogation	kysely
interworking	keskinäinen yhteistyö (termi)
interworking reference point	kommunikoinnin testauspiste
introduction	esittely
invariant schema	invariantti skeema
invocation	käynnistys, aloitus

L

liaison	suhde
location transparency	sijoitustuntumattomuus, paikkatuntumattomuus

M

matching	sovittaminen
migration transparency	siirtymistuntumattomuus

N

name	nimi (tunniste, jolta ei vaadita yksikäsitteisyyttä)
name space	nimijoukko
naming context	nimentäkonteksti
node	solmu, solmukone
notification	tiedonanto
nucleus	ydin

O

object	objekti, olio
oblication	velvoite
open distributed processing	avoin hajautettu tietojenkäsittely
operation	operaatio

P

perceptual reference point	ulkoisten vaikutusten testauspiste
permission	lupa
persistence	jatkuvuus
persistence transparency	keskeytystuntumattomuus
policy	politiikka
portability	siirrettävyys
primitive binding	yksinkertainen sidonta
programmatic reference point	ohjelmallinen testauspiste
prohibition	kielto
property	ominaisuus
proposition	väittämä

Q

quality of service	palvelun ominaisuus, palvelun laatu

R

reactivation	uudelleenaktivointi
reference point	testauspiste
refinement	yksityiskohtaisempi kuvaus, tarkennus
relation	suhdejoukko, relaatio
relationship	suhde
relocation transparency	uudelleensijoitustuntumattomuus
replication transparency	toisinnustuntumattomuus
role	rooli

S

server	palvelija, palvelun tuottaja
service	palvelu
service offer	palvelutarjous
signal	signaali
static schema	staattinen skeema
stub	töpö, tynkä
stream	vuo
subclass	aliluokka
subdomain	osa-alue, osahallintoalue
subtype	alityyppi
superclass	yliluokka
supertype	ylityyppi
system	järjestelmä

T

target concept	ylikäsite
technology viewpoint	teknologianäkökulma, tekniikkanäkökulma
template	kaavain
template class	kaavainluokka
template type	kaavaintyyppi
term	termi
terminating behaviour	lopetus, lopetus-
termination	lopetus, päätös
thread	säie
trace	polku
trader	meklari
trading	meklaus
transaction transparency	transaktiotuntumattomuus
type	tyyppi
type definition	tyyppimäärittely (sisältää tyyppikuvauksia)
type description	tyyppikuvaus
type repository	tyyppitietovarasto
type repository function	tyyppienhallintapalvelu

U

unbinding	sidonnan purku

V

viewpoint	näkökulma
viewpoint language	näkökulmakieli
withdraw	palvelutarjouksen poistaminen

ISSN 1238-8645
ISBN 941-45-8223-3
Helsinki 1998
Helsinki University Printing House