Mobile Programs for Mobile Users on Malleable Platforms

Jean-Patrick Giacometti

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

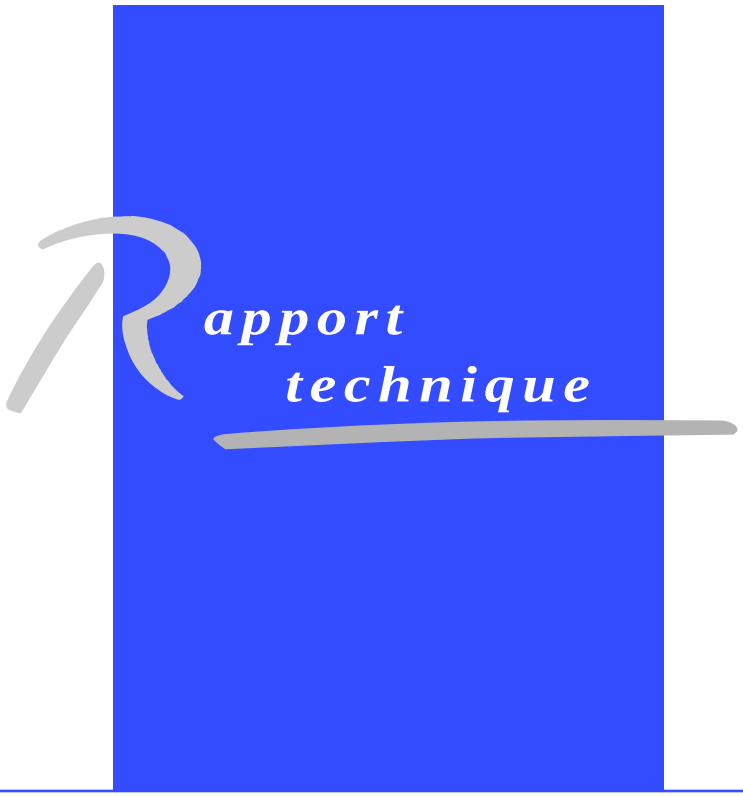
Mobile Programs for Mobile Users on Malleable Platforms

Jean-Patrick Giacometti

N° 0213

Décembre 1997

THÈME 1



ISSN 0249-0803

Mobile Programs for Mobile Users on Malleable Platforms

Jean-Patrick Giacometti *

Thème 1 — Réseaux et systèmes
Projet Semir

Rapport technique n° 0213 — Décembre 1997 — 98 pages

Abstract: Sun's Java popularized the technique of mobile programs, that is, which travel on an internetwork. Network communications have enable to extend the subroutine call mechanism up to use, in addition to local, remote ancillary processing power through an internetwork. Now they enable local execution of programs upon their downloading from some place on the network web. We examine several aspects of this evolution from the point of view of practical design of distributed applications, from which we extract some basic characteristics. With these characteristics in mind, we examine a two main alternative propositions for designing mobile applications. The conclusion of this study points out six landmarks in the whirling landscape of networked software technology.

Key-words: distributed computing, network programming, software technology

This work was supported by Semir.

The author acknowledges the suggestion from Luc Ottavj to write this report and the provision of computing facilities.

* Email :jpgsophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 93 65 77 77 - International : +33 4 93 65 77 77 — Fax : 04 93 65 77 65 - International : +33 4 93 65 77 65
à partir du 05/01/1998
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Mobilité des utilisateurs et des programmes, malléabilité des plateformes

Résumé : Sun, avec Java, a rendu célèbre la technique des programmes mobiles, « mobiles » car ils voyagent sur un réseau hétérogène. Les communications par réseau ont permis d'étendre le mécanisme de l'appel de sous-programme, au point d'utiliser des ressources de puissance de traitement non seulement locales, mais aussi à distance, au travers d'un réseau d'interconnexion. Aujourd'hui ces techniques permettent d'exécuter des programmes dès la fin (et éventuellement le début) de leur téléchargement depuis quelque part dans les toiles. Nous examinons plusieurs aspects de cette évolution, en nous plaçant du point de vue de la conception pratique des applications distribuées. De là, nous tirons des caractéristiques de base. Avec celles-ci à l'esprit, nous examinons les deux propositions alternatives les plus importantes pour la conception des applications mobiles. La conclusion de cette étude met le doigt sur six repères dans le paysage tourbillonnant de la technique du logiciel de réseau.

Mots-clés : informatique distribuée, programmation de réseau, technique du logiciel

At the request of Luc Ottavj, who wondered about our interest for Java and highly distributed systems, we filled this report with thoughts and observations from experiences and concrete measures while working as a software designer at Inria's Rodeo project. Our intention was to enlight the way before our recent position change to Semir.

Contents

1 Foreword **17**

2 Reviewing our Programming Abilities. **18**
2.1 What was Wrong with Chimaera? 18
2.2 What was Wrong with the Object-Orientation in Ivs4? 18
2.3 The Lessons of a Wild OO Design Experience 18
2.3.1 Micro-Kernels 19
2.3.2 Location Transparency 19
2.3.3 Security 19
2.3.4 Open-Minded Systems 19
2.4 The Object Paradigm 19
2.5 The Distribution Model 19
2.6 Objects and Distribution 20

3 Terminology **20**
3.1 Basics 20
3.1.1 Network 20
3.1.2 Host 20
3.1.3 Process 21
3.1.4 Remote 21
3.1.5 Ressource 21
3.1.6 Operation 21
3.1.7 Remote Procedure Call 21
3.1.8 Event 21
3.1.9 Request 21
3.1.10 Response 22
3.1.11 Object 22
3.1.12 Client 22
3.1.13 Type 23
3.1.14 Signature 23
3.1.15 Interface 23
3.1.16 POSIX 23
3.1.17 Interface Repository 23
3.1.18 Interface Object 24
3.1.19 Interface Satisfaction 24
3.1.20 Interface Inheritance 24

3.1.21 Server 24
3.1.22 Broker 24
3.1.23 Object Request Broker 24
3.1.24 Object Adapter 24
3.1.25 ORB Core 24
3.1.26 Location Broker 25
3.1.27 Local Location Broker 25
3.1.28 Internet 25
3.1.29 Global Location Broker 25
3.1.30 Location Broker Client Agent 25
3.1.31 Interoperability 25
3.1.32 To Implement an Interface 25
3.1.33 Implementation 25
3.1.34 Manager 26
3.1.35 Skeleton 26
3.1.36 Method 26
3.1.37 Marshall 26
3.1.38 Unmarshalling 26
3.1.39 Module 26
3.1.40 Interface Definition 26
3.1.41 Stub 27
3.1.42 ASN1 Stubs 27
3.1.43 IDL Stubs 27
3.1.44 Interface Compiler 27
3.1.45 Object/Type Uuid 27
3.1.46 Uuid 27
3.1.47 DCE-like Implementation of Uuid 27
3.1.48 Object Reference 28
3.1.49 Encapsulation 28
3.1.50 Implementation 28
3.1.51 Object System 28
3.1.52 Value 28
3.1.53 Object Name 28
3.1.54 Object Reference 28
3.1.55 Parameter 28
3.1.56 Exception 29
3.1.57 Execution Semantics 29

3.1.58 Attribute 29
3.1.59 Execution Engine 29
3.1.60 Parallelism 29
3.1.61 Array Reference 29
3.1.62 Memory Allocation 29
3.1.63 Concurrent Programming Support 29
3.1.64 Embedded System 30

4 Network Computing Concepts **30**
4.1 Non-Objects 30
4.2 Replicated Objects Against Failures 30
4.3 Replica Consistent Behavior 31
4.4 RPC Commmunications are Above Protocols 31
4.5 Sockets are the Endpoints 31
4.6 Socket Addresses Identify Sockets 31
4.7 Socket Addresses Depends on Family 31
4.8 Sockets Divide to Ports 31
4.9 Socket Address Structure 32
4.10 Well-known Ports Simplify The Network Life 32
4.11 Should We Memorize All the Existing Connected Ports? 32
4.12 Port Numbers are Somewhat Arbitrary 32
4.13 Fundamental Service Ports 32
4.14 The RPC Paradigm 32
4.15 Clients and Servers Exploit Managers 33
4.16 Handles Label Running Requests 33
4.17 Handle Semantics 33
4.18 Handle Binding 33
4.19 Binding States for RPC Handles 33
4.20 Handle Representations 34
4.21 Binding Modes of Communication Handles 34
4.21.1 Manual binding 34
4.21.2 Automatic binding 34
4.22 Stubs Mechanism 35
4.22.1 Client 35
4.22.2 Client Stub 35
4.22.3 Server 36
4.22.4 Server Stub 36
4.23 Interface Definition 36

4.24 Compiling Network Interfaces 37
4.25 The Header File 38
4.26 The Interface Specification 38
4.27 Access to Operations 38
4.28 Simulating the Call 38
4.29 Receiving the Call 38
4.30 Replicated Object Distributed Base 38
4.31 The Replica Server Interface Conflict 39
4.32 The Origin of the Name Conflict 39
4.33 Switch Concept 39
4.34 Making a Pure Client 39
4.35 Making a Replicated Server 39
4.36 The Client Module 40
4.37 The Server 42
4.38 The Manager Module 44
4.39 The Location Broker 44
4.40 The Forwarding Service 45
4.41 Broker Database Entry 45
4.42 Interface definition 45
4.43 the Client Module Revisited 46
4.44 The Server Revisited 48
4.45 The Manager Module Revisited 51
4.46 Network System Software 51
4.47 rpc__ Client Calls 52
4.47.1 rpc__alloc_handle 52
4.47.2 rpc__set_binding 52
4.47.3 rpc__bind 53
4.47.4 rpc__clear_server_binding 53
4.47.5 rpc__clear_binding 53
4.47.6 rpc__dup_handle 53
4.47.7 rpc__free_handle 53
4.47.8 rpc__set_async_ack 53
4.47.9 rpc__set_short_timeout 53
4.47.10 rpc__sar 53
4.48 rpc__ Server Calls 53
4.48.1 rpc__use_family 53
4.48.2 rpc__use_family_wk 54

4.48.3 rpc__register . . . 54
4.48.4 rpc__register_mgr . . . 54
4.48.5 rpc__register_object . . . 54
4.48.6 rpc__unregister . . . 54
4.48.7 rpc__listen . . . 54
4.48.8 rpc__inq_object . . . 54
4.48.9 rpc__shutdown . . . 54
4.48.10 rpc__allow_remote_shutdown . . . 54
4.48.11 rpc__set_fault_mode . . . 55
4.49 rpc__ common calls . . . 55
4.49.1 rpc__inq_binding . . . 55
4.49.2 pc__inq_object . . . 55
4.49.3 rpc__name_to_sockaddr . . . 55
4.49.4 rpc__sockaddr_to_name . . . 55
4.50 The rrpc__ Calls (Remote) . . . 55
4.50.1 rrpc__are_you_there . . . 55
4.50.2 rrpc__inq_status . . . 55
4.50.3 rrpc__inq_interfaces . . . 55
4.50.4 rrpc__shutdown . . . 55
4.51 The socket__ call . . . 55
4.51.1 socket__equal . . . 55
4.51.2 socket__to_name . . . 55
4.51.3 socket__to_numeric_name . . . 56
4.51.4 socket__from_name . . . 56
4.51.5 socket__family_to_name . . . 56
4.51.6 socket__family_from_name . . . 56
4.51.7 socket__valid_family . . . 56
4.51.8 socket__valid_families . . . 56
4.51.9 socket__inq_hostid . . . 56
4.51.10 socket__set_hostid . . . 56
4.51.11 socket__inq_port . . . 56
4.51.12 socket__set_port . . . 56
4.51.13 socket__set_wk_port . . . 56
4.51.14 socket__inq_my_netadd . . . 56
4.51.15 socket__inq_netaddr . . . 56
4.51.16 socket__set_netaddr . . . 56
4.51.17 socket__inq_broad_addrs . . . 56

4.51.18 socket__max_pkt_size 57
4.51.19 socket__to_local_rep 57
4.51.20 socket__from_local_rep 57
4.52 The lb__ Calls 57
4.52.1 lb__lookup_object 57
4.52.2 lb__lookup_type 57
4.52.3 lb__lookup_interface 57
4.52.4 lb__lookup_object_local 57
4.52.5 lb__lookup_range 57
4.52.6 lb__register 57
4.52.7 lb__unregister 57
4.53 The uuid__ Calls 58
4.53.1 uuid__gen 58
4.53.2 uuid__decode 58
4.53.3 uuid__encode 58
4.53.4 uuid__from_uid 58
4.53.5 uuid__to_uid 58
4.53.6 uuid__equal 58
4.54 The error__ Calls 58
4.54.1 error__c_get_text 58
4.54.2 error__c_text 58
4.55 The fm__ Calls 58
4.55.1 fm__cleanup 58
4.55.2 fm__enable 59
4.55.3 fm__enable_faults 59
4.55.4 Fm__inhibit 59
4.55.5 fm__inhibit_faults 59
4.55.6 fm__init 59
4.55.7 fm__reset_cleanup 59
4.55.8 fm__rls_cleanup 59
4.55.9 fm__signal 59
4.56 The main__exit Call 59
4.56.1 main__exit 59
4.57 The System Idl Directory 59
4.58 Writing Interface Definitions 60
4.59 Header Section 61
4.60 Import Declaration Section 61

4.61 Constant Declaration Section 61
4.62 Type Declaration Section 61
4.63 Operation Declaration Section 62
4.64 Developing Distributed Applications 63
4.64.1 Client Structure 63
4.64.2 Managing RPC Handles to Communicate 63
4.64.3 Binding States 63
4.64.4 Obtaining Socket Addresses 64
4.64.5 Using RPC Binding States 65
4.64.6 With Bound-to-Host Handles 65
4.64.7 Using Bound-to-Host Mode upon Server Failure 65
4.64.8 Bound-to-Host Asset 66
4.64.9 Using Unbound Handles 66
4.64.10 Identifying Servers in Practice 66
4.64.11 Server Crash 66
4.64.12 Interface Mismatches 66
4.64.13 Using Cleanup Handlers 66
4.64.14 Portability Considerations 67
4.64.15 A Macro for Portability 67
4.64.16 A Utilitarian Module 68
4.64.17 Server Structure 68
4.64.18 Initialization Code 68
4.64.19 Manager Code 70

5 Standardizing Object Management **70**
5.1 OMG's Vision of the Future of Computing 71
5.2 Current problems 71
5.2.1 The Object Paradigm as Encapsulation 72
5.2.2 Complexity 72
5.2.3 The Illusion of the Standard 73
5.3 Overall Technical Goal of Object Management 73
5.4 Object Distribution Transparency 73
5.4.1 Location 74
5.4.2 Access Path 74
5.4.3 Relocation 74
5.4.4 Representation 74
5.4.5 Communication Mechanism 74
5.4.6 Invocation Mechanisms 74

5.4.7 Storage Mechanisms 74
5.4.8 Machine Type 74
5.4.9 OS 75
5.4.10 Programming Language 75
5.4.11 Security Mechanism 75
5.4.12 Performance 75
5.4.13 Extensible and Dynamic Nature 75
5.4.14 More than One Name Space 76
5.4.15 Name Spaces 76
5.4.16 Object Handles 76
5.4.17 Efficient Reference to Objects 76
5.4.18 Queries 76
5.4.19 Access Control 76
5.4.20 Concurrency Control 77
5.4.21 Parallelism Policy 77
5.4.22 Open-Ended-ness 77
5.4.23 Errors 77
5.4.24 Versioning 77
5.4.25 Events 77
5.4.26 Handling of objects 77
5.4.27 Supervision 78
5.4.28 Backups 78
5.4.29 Restore 78
5.4.30 Locale 78
5.5 What is Missing 78
5.6 The Object Model for Architecture Compliance 79
5.6.1 A Common Semantic 79
5.6.2 The Basic Semantics 79
5.6.3 Servers are Execution Semantics 79
5.6.4 Sophisticated Systems are not Considered 80
5.6.5 Object Management Plays Classics 80
5.6.6 Execution and Construction 80
5.6.7 Activation is a Necessary Step 80
5.6.8 Operation Signatures are the Interface 80
5.7 The Core Object Model 81
5.7.1 Non-Object Types 81
5.7.2 Exception Handling 81

5.7.3 Things that are Not Core Objects 81
5.7.4 Confusion Between Subtyping and Inheritance 81
5.7.5 Subtyping is a Modus Vivendi Rule 81
5.7.6 Inheritance Saves Work 81
5.7.7 Inheritance is Mono and Constant 82
5.7.8 One Inherit Two Things 82
5.7.9 Subtyping Implies Inheritance 82
5.7.10 Operation Dispatching 82
5.7.11 Arguments 82
5.7.12 A Unique Face 82
5.8 Implementation 82
5.8.1 A Class is a Combination 83
5.9 CORBA Principles 83
5.9.1 Is a Request Always First Bound to an Object? 83
5.9.2 The Connection of the Implementation 83
5.9.3 The Travelof the Request 84
5.9.4 Skeletons 84
5.9.5 The Basic Object Adapter 84
5.9.6 The Library Object Adapter 85
5.9.7 The Database Adapter 85
5.9.8 The Interface Repository 85
5.9.9 The Implementation Repository 85
5.9.10 Models of Implementation Design 85
5.10 The Failure of DCE as a Global Solution 86
5.11 OMA Object Management Architecture 86
5.11.1 The ORB Component 86
5.11.2 The Object Services Components 87
5.11.3 The Common Facilities 87
5.11.4 The Application Objects 87
5.12 The ORB Specific Duties 87
5.13 An Example of ORB Function 88

6 Basic Characteristics **88**
6.1 The Need For a Standard Network OS 88
6.2 Interaction of Network Tools with Local Ressources 89
6.3 What Should a Net OS Be? 89
6.4 The Object Orientation 90
6.5 The System Language Choice 90

6.6 The Abstract Machinery 90
6.7 The Adaptation to Multimedia 90
6.8 The Integration of Previous DCE 91
6.9 Adaptation to Embedded Systems 91
6.9.1 Implementation Architectures 92
6.9.2 Virtual Machine Encapsulation 92
6.10 The OS should be Ubiquitous and Free 93
6.11 An International Standard Language 93
6.12 An International Standard Object Management 93

7 Java **93**
7.1 The Java Language 94
7.2 The Java Components 94
7.3 JAVA and CORBA May Complement 94

8 Inferno **94**
8.1 The Language of Inferno 94
8.2 The Limbo Virtual Machine 95

9 Landmarks **95**
9.1 The Commercial Success of Linux 95

We use from here some acronyms :

JAVA	Jet Another Vague Acronym
CL/AF	Class Libray/Application Framework
ORB	Object Request Broker
GUI	Generated User Interface
IDL	Interface Definition Language
Micro$oft	a software company
CORBA	Common Object Request Broker Architecture
DCOM	a widely spread solution to distribution
RMI	Remote Method Invocation
OO	adj. Object Oriented
OS	Operating System
CSMF	Conceptual Schema Modeling Facilities

A software designer's life is not an easy one. Software is by nature abstract. Thus, building good quality and organizing its production is difficult.
Besides, platforms are quickly evolving. The want for malleable software increases as a competitive asset.
The performances of computing and communication systems are widely limited by an unsufficient account of the basic structures of thought and behaviors.
Besides, the communication modes of machines between themselves or with the environment are diversifying. Let's have a look the tendancies to uniformization or diversification: the client/server model (DCE), the transport mode (TCP/IP), the services (e-mail, Web), the graphic interface, Java ISO, the applet Java protocol, The Java virtual machine, the publishing protocol of web pages, postscript, pdf, html, abstract windowing platform independant toolkits, research robots, distributed databases, cheap home minimal computers (NC), multimedia, supervision, observation.
Uniformization may hide a shameful diversification. The stubs are not at the same level. Listen to the press :

> ... On the Suns in the Sun lab you can point 'Netscape' directly at a test page:
>
> netscape test.HelloWorld.html

> but this does not work with Orion.
>
> ...Each ".class" file encodes [bytecode] a single "class" from the file. These classes can be executed by the "java" program or transmitted via the world wide web to a browser that runs them. It is said that if the page is in directory with path name D on the server and your operating system uses S in path names
>
> (S.DOS=‘\’, S.UNIX=‘/’, S.Mac=‘:’) then the applet code for class C must be in directory DSclasses and have a name C.class. However Netscape 2.01 for Suns looks in directory D, unless you add the attribute CODEBASE=classes to the Applet tag.
>
> ...Javascript is a mark up language invented by Netscape to satisfy the same need as Sun's Java. Namely to have a way of sending code accross the WWW and have it run on the client's system. (Unlike a CGI [CGI in comp.html.syntax] which is executed on the server and sends the results to the client). It was originally called "livescript". It looks like Java. Javascript is probably easier to implement in a browser.

There are many reasons to do nothing and wait that the market yields the best of all. The tendancy is to Object-Oriented components and Framework-based or so technology (CL/AF). The latter starts from the idea that every application can be reduced to a simple completion, with parameters and functional specifications (IDL), of articulation zones (hot-spots), within a template application which is used as a domain skeleton, and which is called *framework*.
Industrial standards exist (CORBA, DCOM, Java RMI, etc) for the *middleware*, or customized standard software; they point at get rid of the complexity of distributed softwares. Below, at the implementation level, ORB and message passing technologies are themselves practically industrial standards.
Besides, we have some reasons to doubt that there is nothing to do. Emerging standards have still a vague semantics; they are far from being totally specified; they lack functions; their interoperability is erroneous. They are not yet fit to real-time performance and match but a few fiability requisites.
People hope so promising competitivity gains that the CL/AF remain internal products among the groups that develop them. On this horizon, no sign of change is apparent.
These technologies and standards are mainly focused on some points, the GUI. But we cannot go further with products which are designed to match a unique function-

ality; new systems will have to account for constraints specified from a larger crafts and know-how spectrum that the traditional engineer one.
We cannot foresee all future possible applications. Design will need to be revised iteratively.
Even with such dominant technologies, the making of an application remains black magic. If domain focused AF can cooperate, this is possible only via a very small common ground ((root Class Object), which brings the fundamentals. A communication problem remains.
If we consider a universal AF, a sum of all functionalities which are not only tied to a particular domain, as an extensive OS of which a sole component is currently active, we have got rid of the communication problems. But we have a permanent overhead.
Emerging technologies make projects as well as teams more dependant than in the classical case. The goal is now to build a class hierarchy; it brings an additional dimension and a new difficulty : the agreement on the interfaces. The least design decision has more extent than in the conventional case while the OO way does not compensate by itself for the poor quality of the design.
Thus a science of the communication between cognitive agents is still to develop. This implies an enlarged vision of perception, of reasoning conceptualization, and of representations in singular contexts.
Artificial cognition, that of robot browsers, can no longer be dissociated from our natural cognition.
At that point, can we put forward a method of action? It should ne convenient to recognize and understand to what extent existing data bases and applications are interoperable and *to measure* their shortsightedness level. Software shortsightedness measures how a software piece is blind to its environment, as well as by its design as by its execution.
Most ORB badly if not tolerate dynamic invocation. This deserves a glance at.
CORBA does not submit to the typing paradigm. If typing is to remain a useful verification technique, that's a problem. We must also have a look at this.
Iteration and incremental increase in the designer-programmer-user trilogy must be encouraged. This trilogy must be seen as a concrete system.
No problem must be ignored or over-estimated; all of them interact between themselves. The intervention of man in the use, the maintenance and the programming process is to be seen as an actor of the process, or as a know-how source.
From here, can we clearly define some goals? If we do not reject beforehand any model type, the modeling of the whole will be a complex matter. So we must

think design systems which account for all the aspects of modeling, simulation, and knowledge management. Full accounting for wants and knowledges should lead to design systems much better fit to current needs of everyday life.
Is it too early to carve a design solution for communication systems which could help us to access intelligent communication? We coild start with rules of thumb : get in each domain a browser and a configuration tool; schedule to discard every version of every software; this will be done imperatively sooner or later. So schedule it! Give a good position to work on distributed forms of cognition, particularly if oriented towards the analysis, interpretation, decision or action processes. To get there, the domains of artificial intelligence, non-standard logic, and interactive multimedia communications are major challenges. They lead to building around the interfaces of cognition sciences and ergonomy.

1 Foreword

Initially our basic purpose was to plan to issue a mathematical model for graph drawing in a network context. We intended to develop an automatic drawing tool which would be capable of handling first-order CGs. Some people seem to be willing to pay for.
Our first question was : isn't it too technically complex? We planned to examine the database and the connectivity complexity of the project, versus our own present technology, to issue something like a bronze architecture. In addition to a full review of our technical abilities in the network context, we wondered about possible extensions of the model and further about the relation between languages such as CGs, KIF, Prolog, SQL, Predicate Calculus, and HTML. Are they intra-translatable? Is this question sensefull? Then we realized that such interesting questions will have out of reach solutions as long as our network environment is uncertain. Thus we faced a more basic question. What is the trend of the network context? Will structured document become the ultimate IHM, over the client/server model? What kind of cooperative objects will invade the internet? Are cooperative processes among the bricks of the network context?
We begin here a reflexion about interactive cooperative processes (objects?) on a network, starting with our concrete knowledge of the matter. We intend to push each cluster of ideas to its limits. An asset should be to get a well-founded Java developer ability. This seems a good short term payback.

2 Reviewing our Programming Abilities.

Chimaera is a big application we wrote a good part of. Although the RPC is part of the application code, Chimaera is a client/server application. Chimaera implements its own model inspired from the OSF's DCE model.

2.1 What was Wrong with Chimaera?

Using a OSF's DCE model with RPC makes coding easy but operation suffers from the creation of orphans objects due to network or host failures. The handling of failure is left to the application, which reveals to be a tough problem in practice.
What was wrong with Chimaera? Having no object-mind in Chimaera, no object interface, made testing and debugging a sheer hassle.
Including the object technique in Chimaera leads to CORBA Although Chimaera has nothing to do with object methodology, we withdraw the OMG's CORBA parlance to describe the architecture.
Object-orientation is not a panacea. We developed a modest part of Ivs4 (also known as "rendez-vous"). From this experience we demonstrated that being "object oriented" does not by itslef solve any of the (hard) problems we were facing.
Which problems multimedia programming brings? Our experience of a multimedia application convinced usthat were reinventing the wheel, if not even the barrow, as there are mimicking distributed OSs.

2.2 What was Wrong with the Object-Orientation in Ivs4?

Ivs4 was an object-oriented version of iv3. Two things were confused then. One was to merely build something using OO technique. The other was building something interesting, using an object metaphor as a construction tool. It is well-known evidence that any big system can benefit from OO technique and can be written in an OO language, just like it can be written in French.

2.3 The Lessons of a Wild OO Design Experience

From this, what are the real goals to achieve adequate OS? Some say that " the computer is the network. ", bringing in a major challenge, braking the old monolithic single machine OS to pieces as a collection of distributed services. A bunch of topics derive from this look : micro-kernels, location transparency, scattered security, open-mindedness.

2.3.1 Micro-Kernels

Delocalizing an OS yields micro-kernels. What remains local? or to what extent can we distribute things? These questions lead to the micro-kernel idea. Note that micro-kernels may exists in well-designed old monolithic OSs, say VMS.

2.3.2 Location Transparency

Simple semantics for taming network complexity and proliferations. From the network metaphor we derive a leading principle. Things (services) should be accessed the same way regardless of the address space they are sited, on the same or on another machine. This implies that service semantics be based on network semantics. The best optimization is supposed to bewhen most things are local.

2.3.3 Security

Keep your traveling secret secret? Security should not be lowered by distribution.

2.3.4 Open-Minded Systems

An obsolete-as-soon-as-paid Babel Tower of black-box bricks? User should be able to easily introduce services among the "standard" ones. This implies well-designed interfaces between systems services and support for dynamic instal. More, users should be able to write their own system services without threatening the overall system. different languages, different paradigms, deal with diversity and unforeseen evolution.

2.4 The Object Paradigm

What to retain from the object paradigm ? A mini kit from object paradigm may retain a few (demonstrated) useful points : the interface rules above all; the implementation is separate; multiple implementations may coexist without contradiction. That's all for the sake of simplicity and coherence.

2.5 The Distribution Model

Which models for distribution? a few specific paradigms. CSP, RPC and distributed objects.
CSP model for distribution. (Communicating Sequential Processes). Mainly messages for IPC (Inter Process Control).

RPC model for distribution (Remote Procedure Call) classical call paradigm copied, simple. pass control to different machines.
RPC model advantage simple to program, since preserves the semantics of ordinary procedure call, serial mode, but for servers.
RPC model drawback no concurrent operation between caller and callee. it is difficult for the semantics of the call to be exactly the same. ex: call by name parameter management: the recipient of the RPC will need to execute in an environment that is similar to the one in which the call was made. Thus problems of formats. It is not usually possible to create the dynamic stack in the called procedure's address space. The parameter and return mechanisms should be the same as for a local implementation. ASN1 and IDL compilers help to handle the uniformity.

2.6 Objects and Distribution

Are objects really useful for distribution? We can argue that objects naturally embrace the message-passing paradigm.
What about Distributed objects? Are objects natively distributed? No at first sight. The orientation is : a single name space on a single machine. So the tendancy is to make the distribution transparent, thinking in serial mode. The system manages the distribution, message-passing, syncro and load-balancing of objects on the network.

3 Terminology

To that point, we believe useful to fix our semantics and introduce concepts and definitions in derivative order, using as few known words as possible.

3.1 Basics

3.1.1 Network

Data transmission media thru which hosts can be connected.

3.1.2 Host

Also 'host processor'; exchanging node of at least one network; a processor where telecommunications access methods reside.

3.1.3 Process

A semantically identified program run on a host.

3.1.4 Remote

In another process, on the same or a different host.

3.1.5 Ressource

Also called a 'service', say in CORBA. Unix and Limbo presents most system ressources as files.

3.1.6 Operation

A function or procedure thru which an object is accessed; a service that can be requested; a definition of a signature; an identifiable entity that denotes a service that can be requested; an operation is not a value; an operation has a signature that describes specifications of the parameters required, the results, the exceptions, the contextual info and an indication of the execution semantics the client should expect from a request for th eoperation. The genericity of operation is obtained by the total decoupling of implementation from interface specification.

3.1.7 Remote Procedure Call

An invocation of a remote operation. A remote procedure call request that a particular operation be performed on a particular object.

3.1.8 Event

Something that occurs at a particular time.

3.1.9 Request

A client issues a request to cause a service to be performed; a request consists of an operation and zero or more parameters; an operation invocation; clients request services by issuing requests; a request is an event; th einfo associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context.

3.1.10 Response

Also called ' results', the information returned to the client, which may include values as well as status info indicating that exceptional conditions were raised in attempting to perform the requested service; a distinguished OUT parameter.

3.1.11 Object

An entity that is manipulated by well-defined operations; a combination of state and a set of methods (operations) that explicitely embodies an abstraction characterized by the behavior of relevant requests; in computer architecture, an object can be anything that exists in and occupies space in storage (programs, lib) and on which operation can be performed; OMG defines the object management paradigm as the ability to encapsulate data and methods for software development; an object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client; every object has a type; programs can access any object of a given type thru one or more interfaces each interface is a set of operations that can be applied to any of those objects.
ex : classify several printer queues as objects of one type ; any of these objects can be accessed thru a directory interface that includes operations to add, delete, and list jobs in the queues.
ex : consider databases as objects : programs can access databases thru two interfaces : an update interface with operations add and delete; and a lookup interface with ops byname and bynumber.

3.1.12 Client

A process that uses ressources. In a basic context, namely RPC, a program that makes remote procedure calls to request operations. The code or process that invokes an operation on an object; a client of a service is any entity capable of requesting the service.
RPC Remote Procedure Call Runtime library Contains the calls that enable local programs to execute procedure on remote hosts. These call transfer requests and responses between clients and servers. The clients are the programs calling the procedures. The servers are the programs executing the procedures.

3.1.13 Type

Basically an operative class of objects : all objects of a specific type can be accessed thru the same interface(s); a categorization of values operation arguments, typically covering both behavior and representation; objects are grouped into types and individual objects are instances of theirrespective types; a type determines what operations can be applied to ist instances; a type is an identifiable entity with an associated predicate defined over values; a value satisfies a type if the predicate is true for that value; a value that satisfies a type is called a member of the type; the extension of a type is the set of values that satisfy the type at any particular time

3.1.14 Signature

Defines the parameters of a given operation including their number ordre, data types, and passing mode; the results if any; and the possible outcomes that might occur.

3.1.15 Interface

A set of operations; vague; a listing of the operations and attributes that an object provides. This includes the signatures of the operations and the types of the attributes; a description of a set of possible operations that a client may request of an object; an object satisfies an interface if it can be specified as the target object of any request that satisfies that interface [in each potential request described by the interface]; interfaces are specified in IDL. The definition of an interface determines the calling syntax – the signature – for each of its operations; ex: C++'s .h . A server that implements the operations in an interface is said to export that interface. A client that request the operations is said to import that interface. The interface is independent of the mechanism that conveys request between client and server. The interface is also independent of the way the operations are implemented.

3.1.16 POSIX

Portable operating system interface; a standard.

3.1.17 Interface Repository

A storage place for interface information.

3.1.18 Interface Object

An object that serves to describe an interface; resides in an interface depository.

3.1.19 Interface Satisfaction

An object satisfies an interface if it can specified as the target object in each potential request described by the interface.

3.1.20 Interface Inheritance

A composition mechanism for permitting an object to support multiple interfaces.

3.1.21 Server

A process that provides ressources. In a basic context, a program containing routines that can be invoked from remote hosts. A server is a program that implements one or more interfaces and provides access to one or more objects. A server exports one or more interfaces. A server accepts request for operations in any of its interfaces. A process implementing one or more operations on one or more objects.

3.1.22 Broker

Basically a server that provides information about ressources.

3.1.23 Object Request Broker

A CORBA concept; provides the means by which clients make and receive request and responses.

3.1.24 Object Adapter

The ORB component which provides object reference, activation, and state related services to an object implementation.

3.1.25 ORB Core

The ORB component which moves a request from a client to the appropriate adapter for the target object.

3.1.26 Location Broker

A broker is a server that provides information about resources. The location broker enables clients to locate specific objects (databases) or specific interfaces (data retrieval interfaces). a set of LLB, GLB and the LB client agent, maintaining information about the location of objects and interfaces.

3.1.27 Local Location Broker

A server; maintains info about objects on the local host, plus the LB forwarding facility.

3.1.28 Internet

A set of connected networks, which not necessarily use the same communication protocol.

3.1.29 Global Location Broker

A server; maintains global info about objects on a network or internet.

3.1.30 Location Broker Client Agent

The way (run time lib) client programs communicate with LLB and GLB servers.

3.1.31 Interoperability

The ability for ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.

3.1.32 To Implement an Interface

To provide the routines that execute the operations in an interface.

3.1.33 Implementation

A definition that provides the information needed to create an object and allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of the data structure used to represent the core state associated with the object, as well as definitions of the methods that access the data

structure. It will also typically includes information about the intended interface of the object.

3.1.34 Manager

A manager implements one interface for one type.

3.1.35 Skeleton

The object-interface-specific ORB component which assists an object adapter in passing requests to particular methods.

3.1.36 Method

An inplementation of an operation. code that is executed to perform a service; an immutable description of a computation that can be interpreted by an execution engine; has an immutable attribute, the method format, that defines the set of execution engines that can interpret it.

3.1.37 Marshall

To copy data into an RPC packet. Stubs perform marshalling.

3.1.38 Unmarshalling

To copy data from an RPC packet.

3.1.39 Module

To cope with the namesapce and dynamic loading needs of large programs.

3.1.40 Interface Definition

An interface definition specifies the interface between a user of a service and the provider of the service; it defines how a client sees" a remote service and how a server "sees" request for its service.

3.1.41 Stub

Basically a program module that transfers rpcall and responses between a client and a server. Stubs perform marshalling/unmarshalling and data format conversion. Both clients and servers have stubs. Generated from an interface definition.

3.1.42 ASN1 Stubs

The stubs produced by an ASN1 compiler contain nearly part of the "remoteness" in a distributed application; they perform : data conversions; they do not perform : assemble and dissassemble packets, interact with the RPC RTL Runtime Library.

3.1.43 IDL Stubs

The stubs produced by an IDL compiler contain nearly all of the "remoteness" in a distributed application; they perform : data conversions, assemble and dissassemble packets, interact with the RPC RTL Runtime Library

3.1.44 Interface Compiler

ASN1/IDL(Interface Definition Language) Compiler takes as input an interface definition (written in ASN1/IDL); from this generates source code (in portable C) for clients and server stub modules.

3.1.45 Object/Type Uuid

A uuid that identifies a particular object/type. Both the RPC RT lib and the LB use object/type uuids to identify objects/types.

3.1.46 Uuid

Universal Unique Identifier used by core software to identify interfaces, types and objects; basically a combination of a network address and a time stamp.

3.1.47 DCE-like Implementation of Uuid

Identifies every object, type and interface. Say, – DCE-like – 16-bytes indicating the host on which UUID is created, and the time. Say 6 byte for time, say two reserved, and say 8 the host; uuid__gen as text strings (28 characters) or structure. ex : 3a24f883c400.0d.00.00.fb.40.00.00.00

3.1.48 Object Reference

Called also 'objref'. A value that unambiguously identifies an object. Objref are never reused to identify another object.

3.1.49 Encapsulation

Encapsulation of object data is making data accessible only in a way controlled by the software that implements the object.

3.1.50 Implementation

Types have implementations; a set of data structures that constitues a stored representation, and a set of methods or procedures that provide the code to implement each of the operations whose signature is defined by the type.

3.1.51 Object System

A collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code.

3.1.52 Value

Anything that may be a legitimate actual parameter in a request.

3.1.53 Object Name

A value that identifies an object.

3.1.54 Object Reference

An object name that reliably denotes a particular object.

3.1.55 Parameter

Characterized by its mode (IN, OUT, both) and type.

3.1.56 Exception

An indication that an operation request was not performed successfully; all signatures include the standard exceptions:

3.1.57 Execution Semantics

Two in OMG : at-most-once and best-effort (a request-only operation, cannot return any result, and the requester never synchronizes with the completion, if any, of the request.

3.1.58 Attribute

Declaring a pair[may be read-only] of accessor functions. [what difference with pure memory?]

3.1.59 Execution Engine

An abstract machine that can interpret methods of certain formats, causing the described computations to be performed.

3.1.60 Parallelism

Coarse-grain parallelism. For instance, Limbo has a channel facility for convenient (coarse-grain) parallel programming.

3.1.61 Array Reference

A source of bugs. So we appreciate languages which bounds-check them.

3.1.62 Memory Allocation

Another source of bugs. We wish to have a memory which is garbage collected, and throw away famous ugly debugging expensive tools.

3.1.63 Concurrent Programming Support

A set of routines creates and manages a multitasking environment within a process. Several threads of execution run within one process and address space. Each thread is called a task. This allows to divide a program into pieces that run in parallel ; especially useful in servers that simultaneously manage several remote requests.

3.1.64 Embedded System

Maybe it would be fair to say that an embedded system is simply any system dedicated to either a fixed function or a very limited range of functions, provided those functions involve interacting with some form of application specific hardware. And perhaps adding... that it is not generally perceived as a computer but as a device performing some useful role or task.
A microwave oven is perceived as a microwave oven, the embedded computer aspect is probably not realized by the average consumer (until they have to reset the clock!). Just ask the question, can you run your own software on that 8048 MCU? Normally, a computer that you can't load and run different programs on, and that is dedicated to just one task is considered "embedded". Another thing to note is that the 8048 is called a "Micro*controller*", not "microprocessor". But a microcontroller could be either the central component of an embedded system or a general purpose computer. Take the 8052-BASIC as an example.

4 Network Computing Concepts

Let's have our first step to exploring network softwares interactions. A Network Computing architecture can benefit from object-orientation.: programs access objects thru interfaces, not directly : more interchangeability. Programs are cast in terms of the objects they manipulate rather than the machines with which they communicate : more universality.

4.1 Non-Objects

Some applications do not involve specific objects and types. if an application operates on only one object, one can specify uuid_nil, the nil UUID, as its type. if an application does not operate on any object, specify uuid_nil for both the type and the object.

4.2 Replicated Objects Against Failures

Replicas are copies of an object; all replicas of an object have the same UUID. Replicating an object can ensure availability despite hard or network failure.

4.3 Replica Consistent Behavior

Replicas can be weakly or strongly consistent; weak consistency allows access to replicates even at times when they are not identical.

4.4 RPC Commmunications are Above Protocols

The RPC RT is independant of any underlying communication protocol. The destination address of a message automatically determines the protocol to be used. Both the sending host and the receiving host must support the protocol. ex : IP (Darpa Internet Protocols) or DDS (Domains Protocols).

4.5 Sockets are the Endpoints

A socket is an endpoint for communication, in the form of a message queue. NCS RPC uses the Berkeley Unix socket abstraction for interprocess communicatins. An RPC server "listens" on one or more sockets; it receives every message delivered to a socket on which it is listening.
ex : DDS socket or IP socket.

```
dds:464a.590f  port 67
ip:192.5.7.9   port 40
```

4.6 Socket Addresses Identify Sockets

A socket is uniquely identified by a socket address (sockaddr) :
address family (also called protocol family)
network address (uniquely identifies a host)
port number

4.7 Socket Addresses Depends on Family

In the IP family, the network address and the host ID are identical. In the DDs family, a network address = network ID + host ID.

4.8 Sockets Divide to Ports

The port number specifies a communication endpoint within a host; "port" and "socket" are synonymous; "port number" and "socket address" are not.

4.9 Socket Address Structure

A socket address can be represented textually by a string of the form family:host[port] ex :

```
ip:tuc[57]
ip:#102.5.7.9[53]
dds://allium[101]
dds:#87f99.114a[120]
```

4.10 Well-known Ports Simplify The Network Life

For example, the administrators of the Internet Protocols have assigned the port number 23 to the "telnet" remote login facility.

4.11 Should We Memorize All the Existing Connected Ports?

The Location Broker concept solves the conflict problem by enabling clients to locate servers easily without direct use of well-known ports.

4.12 Port Numbers are Somewhat Arbitrary

A server can use ports that the RPC RT library assigns dynamically. The server registrates its socket address, including the assigned port, with the location broker. The client uses location broker lookup calls to obtain the socket address of the server. The dynamically assigned port is said to be opaque. Neither the client nor the server can know the port number. Thus an application can always coexist with other servers.

4.13 Fundamental Service Ports

Are usually well-known ones The location broker itself uses one well-known port to listen for requests. Clients and servers find Global Location Brokers by broadcasting to this port.

4.14 The RPC Paradigm

Solves real non-locality to virtual locality A client stub acts as the local representative of a (non-local) procedure; it uses RPC RT library calls to communicate with the server.

4.15 Clients and Servers Exploit Managers

An RPC client is a program that makes remote procedure calls to request operations. An RPC sever is a program that performs the operation using one or more interfaces. A manager is a set of procedures that implements the operations in one interface for objects of one type. Each combination of interface and type has its own manager.

4.16 Handles Label Running Requests

The clients makes a rpcall, requesting that a particular operation be performed on a particular object; the RPC RT lib needs the following info to transmit the call : the object on which the operation is to be performed the location of a server that can perform the operation. The client process represents this info in a "handle".

4.17 Handle Semantics

A Network Architecture provides calls to create and manage handles. Once created, a handle represents the same object; however it may represent different servers at different times; or no server at all. The server location represented in a handle is called the "binding".

4.18 Handle Binding

To bind a handle is to set its server location.
ex : PRC handle (type handle_t) -> struct object ID, socket address = address family, network address, port

4.19 Binding States for RPC Handles

Unbound (also called "allocated") : identifies an object but does not identify a location. Thus the RPC RT lib broadcasts the request to all hosts on the localnetwork. Any server that exports the requested interface and supports the requested object can respond; the client accepts the first response it receives.
Bbound-to-host : does not identify the port number of the server that exports the requested interface. If the requested interface specifies a well-known port, the RPC RT send the request to it; otherwise, the request goes to the Local Location Broker forwarding port (that provides access via a single addr to all of the objects and interfaces at the local host), and the LLB forwards the request to the server.

Fully-bound (also "bound-to-server") : identifies an object and the complete socket address of a server.
Therefore, RPC handles are always fully bound when a rpcall returns, and the client does not need to use the broadcasting or forwarding mechanism for subsequent calls to the server.

4.20 Handle Representations

Explicit Handles Each operation in the interface may have a handle variable as its first parameter, or the handle identifier may be a global variable in the client. The explicit handle passes from the client to the server, thru the client stub, the client and server RPC RT libs, and the server stub. The server can thereby identify its client.
Implicit handles When the handle is global, the client stub delivers a rpcall using the implicit handle variable to supply the handle info needed by the client RPC RT lib. No need to pass "special" info in each call. But the client can access only one object at a time, unless it explicitely passes another parameter?, and only one server at any time.

4.21 Binding Modes of Communication Handles

Between manual and automatic bindings, the difference of performance is smallest in applications where each call is likely to require rebinding of the handle.

4.21.1 Manual binding

1. client
   generates RPC handle; binds handle as necessary; makes procedure call to stub.

2. client stub
   sends request to server; receives response from server; returns to client.

3. client
   receives call return from stub; frees RP handle as necessary.

4.21.2 Automatic binding

1. client
   using generic handle makes procedure calls to stub.

2. client stub
   call autobinding routine.

3. autobinding routine
   generates RPC handle from generic handle; binds RPC handle as necessary; returns RPC handle to stub.

4. client stub
   sends request to server; receives response from server; call autounbinding routine.

5. autounbounding routine
   frees handles as necessary; returns to stub.

6. client stub
   returns to client.

7. client
   receives call return from stub.

4.22 Stubs Mechanism

Both clients and servers are linked (in the sense of combining object modules to form executable files) with stubs. The client stub takes the place of the remote procedures in the client process. The server stub take the place of the client in the server process. Stubs that make remote procedure calls resemble local calls, enabling clients and servers to use the RPC facilities almost transparently.
These stub modules facilitate remote procedure calls by encapsulating them : copying arguments to and from RPC packets, converting data representation as necessary, and calling the RPC RT lib.

4.22.1 Client

When a client calls an operation in an interface, it invokes a routine in the client stub.

4.22.2 Client Stub

1. marshalls the input parameters into an RPC packet.

2. calls rpc__sar, an RPC RT lib routine called only by stubs, to send the request to the server stub and await a reply.
3. receives the reply packet.
4. unmarshalls the output parameters from the reply packet into the data types expected by the client (specified in the interface definition).
5. converts the output data to the client's native representation, if the server's native representation is different.
6. returns to the client.

4.22.3 Server

1. When a server receives a request, the RPC RT lib calls a server stub routine.

4.22.4 Server Stub

1. unmarshalls the input parameters from the request packet into the data types expected by the server (specified in the interface definition).
2. converts the input data to the server's native representation, if different.
3. calls the manager procedure that implements the operation.
4. marshalls the output parameter vakues into an RPC packet.
5. returns the packet to the RPC RT lib for transmission to the client stub.

Each side uses its native format when it marshalls parameters. Stub procedures in both side check the data repersentation format in incoming packets. A label in the header of each transmitted packet indicates the sender's data representation format for integers, characters and floating-point numbers. When different, the receiving stub converts the data type when it unmarshalls values.

4.23 Interface Definition

```
# file alu/alu.idl
%c // use c syntax

[uuid(4448ecb460000.0d.00.00.fe.da.00.00.00),
```

```
port(dds:[19], ip:[6677]), version(1)]
interface alu
{
[pure] // can safely be called more  than once
void alu\_add( // signature of operation add
handle\_t [in] h,// RPC handle
long [in] a,
long [in] b,
long [out] *c
);
}
# eof alu/alu.idl
Interface definition
# file alu/alu.idl
%c // use c syntax

[uuid(4448ecb460000.0d.00.00.fe.da.00.00.00),
port(dds:[19], ip:[6677], version(1)]
interface alu
{
[pure] // can safely be called more  than once
void alu\_add( // signature of operation add
handle\_t [in] h,// RPC handle
long [in] a,
long [in] b,
long [out] *c
);
}
# eof alu/alu.idl
```

The pure attribute enables more efficient calling semantics. [unclear]

4.24 Compiling Network Interfaces

A review of the files generated by the IDL compiler which translates an IDL interface definition into stub modules linkable with clients and servers.
Flow of files .idl -> .h (header file) _cstub.c or c.c, _cswtch.c or w.c (client files, stub and switch) _sstub.c or s.c

4.25 The Header File

The header file declares the add procedure, initializes the interface specifier alu_v1_if_spec, defines the data type alu_v1_epv_t, contains directives to include the standard NCS header files that define basic data types, and declare RPC RT lib calls.

4.26 The Interface Specification

An if_spec is a data structure that servers pass to the RPC RT lib when they register an interface.

4.27 Access to Operations

An epv_t is a type for an entry point vector, a record of function pointers to the operations in an interface. An IDL compiler appends the version number, in this case _v1, to the interface name when it generates identifiers for interface specifiers and epv types.

4.28 Simulating the Call

The files c.c and w.c contain a procedure add, which marshalls its two inputs arguments, a and b, into an RPC packet and calls rpc__sar to send a rpcall; when rpc__sar returns, the result is unmarshalled from the returned packet into the output argument c.

4.29 Receiving the Call

The code s.c unmarshalls a and b from the packets sent by the clients, then passes those values to the manager procedure add. It marshalls the result, c, into an RPC packet and returns control to the RPC runtime lib, which sends the packet back to the waiting client.

4.30 Replicated Object Distributed Base

A replicated server provides access to a replicated object and works with other servers to maintain consistency among all replica of an object.

4.31 The Replica Server Interface Conflict

The switch file is for the creation of replicated servers. A replicated server functions as both a client and a server of the interface that performs replication. This potentially produce naming conflicts between client stub routines and corresponding manager routines.

4.32 The Origin of the Name Conflict

Servers handling replica must pass copies between them. Thus each of them needs to ask for and yield back a copy. They become client and server for each operation handling the replica. The client and the server libs being melt, one can no more have the same entry point name for the call and the treatment operations.

4.33 Switch Concept

To remain coherent with ordinary client/server design, the server program must contain a say repdb_add manager procedure to execute the operation and a repdb_add client stub procedure to request the operation. The client switch contains "public"procedures – repdb_add –, while the client stub only "private", whose name are not visible outside of the c.c. The client stub defines an epv containing function pointers to thes private procedures, and the client switch invokes these procedures thru the epv – by calling repdb_v1_client_epv.repdb_add.

4.34 Making a Pure Client

To build an ordinary client, link both the client switch and the client stub with client. The client calls the procedures by they ordinary public names;these procedures are contained in the client switch, which then calls the client stub procedures thru the client epv.

4.35 Making a Replicated Server

To build a replicated server, link it with the client stub, but not with the switch. The client code in the server invokes operations not by their public names, but directly thru the epv; conversely, when the server stub receives a rpcall, it invokes the requested operations by the its public name, and the local manager executes the call.
Here follows ordinary client code :

```
normal client code
if\_op(h, a, b, c);

client switch
if\_op(h, a, b, c)
{
(*if\_v1\_client\_epv.if\_op)(...);

}
client stub
if\_v1\_epv\_t if\_v1\_client\_epv = { if\_op };
static if\_op(h, a, b, c)
{
rpc\_\_sar(...);
}
```

Replicated server

```
client code
(*if\_v1\_client\_epv.if\_op)(...);

client stub
if\_v1\_epv\_t if\_v1\_client\_epv = { if\_op };
static if\_op(h, a, b, c)
{
rpc\_\_sar(...);
}
```

4.36 The Client Module

```
#include <stdio.h>
#include "alu.h"
#include "socket.h"
#include <fm.h>

#define CALLS\_PER\_PASS 100

globalref uuid\_\_t uuid\_\_nil;  /* globalref == extern */
extern long time();
```

```
main(argc, argv)
int argc;
char *argv[];
{
handle\_t h;
status\_\_t st;
socket\_\_addr\_t loc;
unsigned long llen;
ndr\_\_long\_int i, n;  /* NDR = network data representation
                                protocol */
int k, passes;
long start\_time, stop\_time;

if (argc != 3) {
fprintf(stderr, "usage: client hostname passes\n");
exit(1);
}
passes = atoi(argv[2]);

fm\_\_init((long) fm\_\_init\_signal\_handlers);
             //  fault manager

/* convert network address of the server host onto a
            socket address */

socket\_\_from\_name((long)socket\_\_unspec,
                  (ndr\_\_char*)argv[1],
(long)strlen(argv[1]), (long)socket\_\_unspec\_port,
&loc, &llen, \_st);

/* create an RPC handle and bind it to the socket address */

h = rpc\_\_bind(&uuid\_\_nil, &loc, llen, &st);

for (k=1, k <= passes; k++) {
start\_time = time(NULL);
```

```
for (i=1; i <+ CALLS\_PER\_PASS; i++) {
alu\_add(h, i, i, &n);
if (n ! = i + i)
printf("Two times %ld is \
                                        NOT %ld\n", i, n);
}
stop\_time = time(NULL);
printf("pass %3d; real/call: %2ld ms\n",
k, ((stop\_time - start\_time) * 1000)
                        / CALLS\_PER\_PASS);
}
}
```

At first call alud_add, the RPC RT lib at the client host extracts the well-known port number in the server from the alu_v1_if_spec interface specifier; so that the handle is fully bound when the RT lib sends the request, and remains so for all subsequent calls.

4.37 The Server

```
#include <stdio.h>
#include "alu.h"
#include "socket.h"
#include <fm.h>

globalref uuid\_\_t uuid\_\_nil;  /* globalref == extern */
globalref alu\_v1\_epv\_t alu\_v1\_manager\_epv;


main(argc, argv)
int argc;
char *argv[];
{
status\_\_t st;
socket\_\_addr\_t loc;
unsigned long llen;
unsigned long family;
socket\_\_string\_t name;
```

```
unsigned long namelen = sizeof(name);
unsigned long port;

if (argc != 2) {
fprintf(stderr, "usage: server family\n");
exit(1);
}

fm\_\_init((long) fm\_\_init\_signal\_handlers);

/* convert name into integer representation of the
           family */

family = socket\_\_family\_from\_name(
(ndr\_\_char *)argv[1], (long)strlen(argv[1]),
                 &st);

/* create a socket for the server at its well-known
           port */

rpc\_\_use\_family\_wk(
family, add\_v1\_if\_spec, &loc, &llen, &st);

/* register the  managers  types the server manages */
rpc\_\_register\_mgr(
&uuid\_\_nil,
&alu\_v1\_if\_spec,
alu\_v1\_server\_epv,
(rpc\_\_mgr\_epv\_t)&alu\_v1\_manager\_epv,
&st);

/* same for managed objects */
rpc\_\_register\_object(...);

/* extract a textual network addr and port nb form
           my socket address */
```

```
socket\_\_to\_name(&loc, llen, name, &namelen, &port,
                           &st);
name[namelen] = 0;
printf{"Registered: name='%s', port=%ld\n", name,
                port);

/* handle 5 calls concurrently */

rpc\_\_listen((long)5, &st);
}
```

4.38 The Manager Module

```
#include "alu.h"

globalref alu\_\_epvt alu\_\_manager\_epv = { alu\_add };

void alu\_add(h, a, b, c)
handle\_t h;
ndr\_\_long\_int a, b, *c;
{
*c = a + b;
}
```

4.39 The Location Broker

Provides clients with info about location of objects and interfaces. Servers register with a location broker their socket addresses and the objects and interfaces to which they provide access. The broker returns database entries that match an object, type, interface or combination of these, as specified in the request.
The location broker architecture is :
LB=LLB+GLB+LBCA
where LB = location broker, LLB = local location broker (runs as daemon llbd),
GLB = global location broker (daemon glbd and nrglbd ,not replicable),
LBCA = location broker client agent set of lib routines.
The local location broker architecture is :
application programs call the LBCA routines to access LLB and GLB databases. When a program issues any location broker call, that call goes to the Client Agent

at the local host. The Client Agent performs the actual remote lookup or update in the appropriate LB databases.

4.40 The Forwarding Service

LLB also provides the LB forwarding service; The forwarding facility of the LLB eliminates the need for a client to know the specific port that a server uses, and thereby helps to conserve well-known ports.

4.41 Broker Database Entry

A LB database entry contains only one object UUID, one interface UUID, and one socket addr.

```
LB database entry = {
object UUID ;
type UUID  ;  //  the unique id that specifies the
                  //  type of the object
interface UUID ;
flag ;        //   global object, so should be
                  //   registered in the GLB database
annotation ;  //   64 chars for user-defined info
socket addr length ;
socket address ; // the location of the server that
                     // exports the interface to the object
```

4.42 Interface definition

```
# file alu/alulu.idl
%c // use c syntax

[uuid(4448ecb460000.0d.00.00.fe.da.00.00.00),
version(1)]                    // no more well-known port
interface alulu
{
[pure] // can safely be called
                                          more  than once
void alulu\_add( // signature of
                                                  operation add
```

```
handle\_t [in] h,// RPC handle
long [in] a,
long [in] b,
long [out] *c
);
}
# eof alu/alulu.idl
```

4.43 the Client Module Revisited

```
#include <stdio.h>
#include "alulu.h"
#include "lb.h"                 // new
#include "socket.h"
#include <fm.h>

#define CALLS\_PER\_PASS 100

globalref uuid\_\_t uuid\_\_nil;  /* globalref == extern */
extern long time();

main(argc, argv)
int argc;
char *argv[];
{
handle\_t h;
status\_\_t st;
lb\_\_entry\_t entry;      // new  GLB entry
lb\_\_lookup\_handle\_t ehandle =
           lb\_\_default\_lookup\_handle; // new GLB entry handle
unsigned long nresults;                  // new
socket\_\_addr\_t loc;
unsigned long llen;
socket\_\_string\_t name;   // new
unsigned long namelen = sizeof(name);
unsigned long port;      // new
ndr\_\_long\_int i, n;  /* NDR = network data
                                    representation protocol */
```

```
int k, passes;
long start\_time, stop\_time;

// new : no need to specify a host
if (argc != 2) {
fprintf(stderr, "usage: client passes\n");
                                        // no more hostname
exit(1);
}
passes = atoi(argv[1]);

fm\_\_init((long) fm\_\_init\_signal\_handlers);

/* no need to convert network address of the server
                                host onto a socket address */

do {                     // new lookup a server addr
                                          in the LB database
lb\_\_lookup\_interface(
&alulu\_v1\_if\_spec.id, &ehandle, 1L,
&nresults, &entry, &st(;
if ( nresults < 1) {
fprintf(stderr,
"interface on valid family not
                                        found in lb lookup\n");
exit(1);
}
) while (!socket\_\_valid\_family(
                (long)entry.saddr.family, &st));


/* create an RPC handle and bind it to the socket
           address */

//      h = rpc\_\_bind(&uuid\_\_nil, &loc,        llen, &st);
h = rpc\_\_bind(&uuid\_\_nil, &entry.addr, llen, &st);
```

```
//      from now fully bound
rpc\_\_inq\_binding(h, &loc, llen, &st);

/* extract a textual network addr and port nb form
           my socket address */

socket\_\_to\_name(&loc, llen, name, &namelen, &port,
                            &st);
name[namelen] = 0;
printf{"Bound to port %ld at host %s\n", port, name);

// from now unchanged

for (k=1, k <= passes; k++) {
start\_time = time(NULL);
for (i=1; i <+ CALLS\_PER\_PASS; i++) {
alulu\_add(h, i, i, &n);
if (n ! = i + i)
printf("Two times %ld is NOT \
                                        %ld\n", i, n);
}
stop\_time = time(NULL);
printf("pass %3d; real/call: %2ld ms\n",
k, ((stop\_time - start\_time) * 1000)
                                          / CALLS\_PER\_PASS);
}
}
```

4.44 The Server Revisited

```
#include <stdio.h>
#include "alulu.h"
#include "lb.h"     // new
#include "socket.h"
#include <fm.h>
```

```
globalref uuid\_\_t uuid\_\_nil;  /* globalref == extern */
globalref alulu\_v1\_epv\_t alulu\_v1\_manager\_epv;
extern char *errir\_text();

main(argc, argv)
int argc;
char *argv[];
{
status\_\_t st;
socket\_\_addr\_t loc;
unsigned long llen;
unsigned long family;
boolean validfamily;        // new
socket\_\_string\_t name;
unsigned long namelen = sizeof(name);
unsigned long port;
lb\_\_entry\_t lb\_entry;       // new
fm\_\_cleanup\_rec crec;     // new


if (argc != 2) {
fprintf(stderr, "usage: server family\n");
exit(1);
}

fm\_\_init((long) fm\_\_init\_signal\_handlers);

/* convert name into integer representation of the
           family */

family = socket\_\_family\_from\_name(
(ndr\_\_char *)argv[1], (long)strlen(argv[1]),
                 &st);

/* create a socket for me,  server, at a dynamically
            assigned port */
```

```
rpc\_\_use\_family(family, &loc, &llen, &st);
                // no more  rpc\_\_use\_family\_wk

/* register the  managers  types the server manages */
rpc\_\_register\_mgr(
&uuid\_\_nil,
&alulu\_v1\_if\_spec,
alulu\_v1\_server\_epv,
(rpc\_\_mgr\_epv\_t)&alulu\_v1\_manager\_epv,
&st);

/* same for managed objects */
rpc\_\_register\_object(...);

// new : register my interface and socket addr with the GLB

lb\_\_register(
&uuid\_\_nil, &uuid\_\_nil,
                &alulu\_v1\_if\_spec.id,
0L, (ndr\_\_char *)"alulu example",
&loc, &llen, &lb\_entry, &st);
);
/* extract a textual network addr and port nb form
           my socket address */

socket\_\_to\_name(&loc, llen, name, &namelen, &port,
                           &st);
name[namelen] = 0;
printf{"Registered: name='%s', port=%ld\n", name, port);

st = fm\_\_cleanup(&crec);  // new : set a cleanup handler
if (st.all != fm\_\_cleanup\_set) {
status\_\_t stat;
fprintf(stderr,
  "Server received signal - %s\n",
                  error\_text(st));
lb\_\_unregister(&lb\_entry), &stat);
```

```
rpc\_\_unregister(&alulu\_v1\_if\_spec, &stat);
fm\_\_signal(st);
}

/* handle 5 calls concurrently */

rpc\_\_listen((long)5, &st);
}
```

4.45 The Manager Module Revisited

// same except for name changes

```
#include "alulu.h"

globalref alulu\_\_epvt alulu\_\_manager\_epv = { alulu\_add };

void alulu\_add(h, a, b, c)
handle\_t h;
ndr\_\_long\_int a, b, *c;
{
*c = a + b;
}
```

4.46 Network System Software

The main pieces of software are :

software	description	package
idl	interface definition language compiler	idl
uuid_gen		NC Kernel
libd	local location broker daemon	_
gibd	global location broker replicable daemon	_
nrgibd	non-replicable form	_
drm_admin	data replication manager admin tool	_
lb_admin	local location admin tool	_
stcode	status code translator	_
conf files	glb_obj.txt, alternate object uuid for the glb	_
	glb_site.txt, in case broadcast fails	_
	uuidname.txt assoc uuid with textual names	_
.idl files	interface definitions	_
.h	C header files	_
calls	rpc__,	_
	rrpc__,	_
	socket__,	_
	lb__,	_
	uuid__,	_
	error__,	_
	fm__,	_
	main__	_

We list the interface to the RPC RT lib.

4.47 rpc__ Client Calls

4.47.1 rpc__alloc_handle

that call identifies a specific object but not a specific server.

4.47.2 rpc__set_binding

in an allocated handle so that it specifies a socket address.

4.47.3 rpc__bind

allocate an RPC handle and sets its binding; same as rpc__alloc_handle ; rpc__set_binding

4.47.4 rpc__clear_server_binding

removes the assoc of an RPC handle with a server but retains the assoc with a host. Making a rpcall with it => the call is sent either to a well-known (wk) port or to the LLB forwarding port on the remote host.

4.47.5 rpc__clear_binding

removes association with server and host, saves the handle for reuse in accessing the same object, possibly via a different server. On a rpcall, the call is broadcast.

4.47.6 rpc__dup_handle

for threads.

4.47.7 rpc__free_handle

4.47.8 rpc__set_async_ack

set or clear asynchronous ack mode in client : receipts of replies are acknowledged asynchronously.

4.47.9 rpc__set_short_timeout

set or clear mode on a handle, so the call fails quickly.

4.47.10 rpc__sar

send a rpcall an Await a Replay from the server; appears only in client stubs.

4.48 rpc__ Server Calls

4.48.1 rpc__use_family

create a socket to communicate with clients, specifying the address family. The RPC RT lib assigns an available port number for the socket.

4.48.2 rpc__use_family_wk

create a socket that uses a well-known port.

4.48.3 rpc__register

an interface with the RPC RT lib. obs.

4.48.4 rpc__register_mgr

specify an interface, a type for which the server exports the interface, and a set of manager procedures that implement the interface for the type.

4.48.5 rpc__register_object

object and its type for which server exports interfaces.

4.48.6 rpc__unregister

an interface. Server will not respond to requests for the unregistered interface.

4.48.7 rpc__listen

for requests, then call the manager proedure for the requested operation and send the result in a reply to the client.

4.48.8 rpc__inq_object

returns the uuid of the object represented by an RPC handle. Used by manager procedures to determine the specific object that they must access.

4.48.9 rpc__shutdown

rpc__listen returns.

4.48.10 rpc__allow_remote_shutdown

allow or disable remote shuts via rrpc__shutdown.

4.48.11 rpc__set_fault_mode

control handling of faults that occur in server routines. By default, a server reflects faults back to the client and continues. Now sends a "comm failure" fault to the client and exits.

4.49 rpc__ common calls

4.49.1 rpc__inq_binding

returns the socket address identified by an RPC handle. Used to identify the server that responded to a rpcall.

4.49.2 pc__inq_object

returns uuid of object represented by an RPC handle

4.49.3 rpc__name_to_sockaddr

given a host name and port number, returns the equivalent socket address; obsoleted by socket__from_name.

4.49.4 rpc__sockaddr_to_name

vice versa. obsoleted by socket__to_name.

4.50 The rrpc__ Calls (Remote)

4.50.1 rrpc__are_you_there

4.50.2 rrpc__inq_status

4.50.3 rrpc__inq_interfaces

4.50.4 rrpc__shutdown

4.51 The socket__ call

4.51.1 socket__equal

4.51.2 socket__to_name

convert a socket address to a textual host name and a port number.

4.51.3 socket__to_numeric_name

4.51.4 socket__from_name

4.51.5 socket__family_to_name

convert the integer representation of a protocol family to its textual name.

4.51.6 socket__family_from_name

4.51.7 socket__valid_family

4.51.8 socket__valid_families

list the address families that are ok to use.

4.51.9 socket__inq_hostid

returns the host id part of a socket address.

4.51.10 socket__set_hostid

4.51.11 socket__inq_port

4.51.12 socket__set_port

4.51.13 socket__set_wk_port

4.51.14 socket__inq_my_netadd

r returns the primary network address of the local host for the specified protocol family.

4.51.15 socket__inq_netaddr

returns the network address part of a socket address.

4.51.16 socket__set_netaddr

4.51.17 socket__inq_broad_addrs

returns a list of broadcast addresses that the local host can use.

4.51.18 socket__max_pkt_size

for the specified protocol family.

4.51.19 socket__to_local_rep

convert a socket of type socket__addr_t to a socket address of a type specific to the local system.

4.51.20 socket__from_local_rep

4.52 The lb__ Calls

Interface to the location broker local agent.

4.52.1 lb__lookup_object

find entries in the glb database that match the specified object identifier.

4.52.2 lb__lookup_type

4.52.3 lb__lookup_interface

4.52.4 lb__lookup_object_local

find entries in the specified llb database.

4.52.5 lb__lookup_range

find entries in the specified database (llb or glb) that match the specified combination of object, type and interface uuids.

4.52.6 lb__register

a specific object and interface. The entry is local or global.

4.52.7 lb__unregister

a specific object and interface.

4.53 The uuid__ Calls

4.53.1 uuid__gen

4.53.2 uuid__decode

convert a character-string representation oof a uuid as generated bu uuid__gen into a uuid__t value.

4.53.3 uuid__encode

4.53.4 uuid__from_uid

convert a domain unique id (uid) into the equivalent uuid. Useful for objects, like files, having already domain uids.

4.53.5 uuid__to_uid

not all uuid can be converted into equivalent uid.

4.53.6 uuid__equal

4.54 The error__ Calls

convert status codes into textual error messages.

4.54.1 error__c_get_text

returns subsystem, module, and error texts for a status code.

4.54.2 error__c_text

returns an error message for a status code.

4.55 The fm__ Calls

allow programs to manage signals, faults and exceptions by establishing cleanup handlers.

4.55.1 fm__cleanup

establish a cleanup handler. Clearly wrong term.

4.55.2 fm__enable

enable async faults

4.55.3 fm__enable_faults

4.55.4 Fm__inhibit

4.55.5 fm__inhibit_faults

inhibit async faults but allow time-sliced task switching.

4.55.6 fm__init

4.55.7 fm__reset_cleanup

4.55.8 fm__rls_cleanup

release a cleanup handler.

4.55.9 fm__signal

signal the calling process.

4.56 The main__exit Call

used at end of a cleanup handler to terminate the program.

4.56.1 main__exit

exit from the calling program.

4.57 The System Idl Directory

It may be located at /usr/include/idl. It contains :

base.idl	basic types and constants
nbase.idl	used in network interface
ncastat.idl	status codes in network computing architecture
	interface definition files for local interfaces (not remotely callable)
lb.idl	interface to the location broker client agent
rpc.idl	interface of the RPC RT lib.
socket.idl	types, constants, and operations pertaining to socket addresses and protocol families.
uuid.idl	types, constants, and operations
uuid_uid.idl	types, constants, and operations for conversion to and from other domain uid.
	interface definition files for remote interfaces
conv.idl	operations that manage client-server conversations.
glb.idl	interface to the glb
llb.idl	
rrpc.idl	operationd that a client can use to request info about a server or to shut it down.
The rrpc_	interface is exported by every RPC server. Its operations are implemented by the runtime support for the server.
idl.base.h	
	defines primitives present in nidl but lacking in C, such as the boolean type. declares or defines data types, external functions and macros used by stubs.
fm.h	
	portable interfacee to the Proces Fault Manager package.

4.58 Writing Interface Definitions

`generating interface uuids`

```
\_ uuid\_gen -c >toto.idl
\_ cat toto.idl
%c
[
uuid(....),
version(1)
]
interface REPLACEME {

}
```

4.59 Header Section

interface attibutes :

- uuid.
- version.
- port; the well-known port or ports on which servers exporting this interface will listen. Avoid it. Prefer to allow the RPC RT assign port dynamically.
- implicit_handle; the global var containing handle information if not set, the handle must be passed as an explicit param to each operation.
- local. do not generate stubs.

4.60 Import Declaration Section

rpc.idl is implicit.

4.61 Constant Declaration Section

4.62 Type Declaration Section

typedef [type_attribute_list] type_specifier type_declarator_list ;
type attr : handle, transmit_as,

handle : a type can serve as a generic handle. You supply an autobinding routine to convert the generic handle type to the RPC handle type.
transmit_as : associates a transmitted type that stubs pass over the network with a presented type that clients and servers manipulate. You should supply routines to perform conversions between the presented and transmitted types.
One use is to help applications pass complex data types as say trees, linked lists, and records that contains pointers. The idl compiler cannot generate code to marshall and unmarshall these data types; but you can supply routines to convert the complex types into simpler types that can be marshalled and unmarshalled.
Or to pass data more efficiently : stubs transmits packed arrays over the network but present sparse arrays to the client and server programs.
filed attr to pass open arrays, last_is and max_is
ex :

```
typedef [ handle] socket\_\_string\_t sockhandle\_t;

typedef struct {
int last;
int [last\_is(last)] data[CARRAY\_SIZE];
} compress\_t;

typedef [ transmit\_as(compress\_t)] int compress\_array[ARRAY\_SIZE];

typedef int nocompress\_array[ARRAY\_SIZE];
```

4.63 Operation Declaration Section

```
[ operation\_attribute\_list ] type\_specifier
operation\_declarator ( paramete\_list );
```

Operation attributes may be :

- pure
- broadcast
- noblock ;no need for confirmation; usable only if no output param nor return value.
- comm_status ; returns a completion status.

paremater attributes may be : in, out, comm_status;
both in and out are possible.
ex :

```
[pure]
void toto\_tata (
handle\_t [i] h,
int      [in, out] *last;
int      [in] max,
status\_\_t [comm\_status, out] *st,
int      [in, out, last\_is(last), max\_is(max)]
values []
);
```

4.64 Developing Distributed Applications

4.64.1 Client Structure

For each interface,
- the header file .h, generated from the interface definition.
- the client application, user-written code calling the remote procedures.
- the client switch, generated
- the client stub, generated
- user-written code that performs automatic binding or data type conversion.

4.64.2 Managing RPC Handles to Communicate

The client uses a rpc handle to represent the object it is trying to access and the location of a server that can execute the call; two binding techniques : manual and automatic;
- manual : the client creates and uses rpc handles directly;
- automatic : the client uses generic handles ; whenever it makes a rpcall, the stub calls a user-written routine that converts the generic handle into a rpc handle.

4.64.3 Binding States

RPC handles may be in three states :
unbound handle : only an object ref – used for broadcasting;
bound to host handle : object + host refs – used for forwarding;
fully bound handle : host, object, port refs.

4.64.4 Obtaining Socket Addresses

Use a location broker lookup, lb__lookup_interface, or a socket__from_name call. A client usually specifies lb__default_lookup_handle as the value in its first location broker lookup call. The returned lookup_handle value works as a location pointer, for subsequent calls if the parameter max_results is reached before the end of the database. If a client knwos the name and the address family of the host it wishes to access, it can call socket__from_name to obtain a socket address withou using the lb.

```
status\_\_t st;
lb\_\_entry\_t entry;
lb\_\_lookup\_handle\_t ehandle = lb\_\_default\_lookup\_handle;
unsigned long nresults;
...

do {  // new lookup a server addr in the LB database
lb\_\_lookup\_interface(
&alulu\_v1\_if\_spec.id, &ehandle, 1L,
&nresults, &entry, &st(;
if ( nresults < 1) {
fprintf(stderr,
"interface on valid family no found
                          in lb lookup\n");
exit(1);
}
) while (!socket\_\_valid\_family(
                (long)entry.saddr.family, &st));

h = rpc\_\_bind(&uuid\_\_nil, &entry.saddr, entry.saddr\_len,
                 &st);
if (st.all != status\_\_ok) {
fprintf(stderr, "Can't bind - %s\n", error\_text(st));
exit(1);
}


socket\_\_from\_name((long)socket\_\_unspec,
```

```
                    (ndr\_\_char *)argv[1],
(long)strlen(argv[1]), (long)socket\_\_unspec\_port,
                    &loc, &llen, &st);
h = rpc\_\_bind(&uuid\_\_nil, &loc, llen, &st);
```

4.64.5 Using RPC Binding States

With fully bound handles The RPC RT lib sends the call directly to the host and port identified in the handle. Any socket address obtained from a lb will be fully specified. A socket address converted from a host name will not be.

4.64.6 With Bound-to-Host Handles

The RPC RT sends the call to the host identified in the handle. If a well-know port was specified in the definition of the requested interface, the call is deleivered to that port. Otherwise, the call is delivered to the llb forwarding port. The llb, provide a server for the requested object and interface has registered with it, forward the call to the port on which the server is listening. When the call returns, the RPC RT lib at the client host binds the handle to that port, and any subsequent calls are sent directly to the server.

```
socket\_\_from\_name (socket\_\_internet, hostname, hlen,
socket\_\_unspec\_port, &saddr, slen, &st);
h = rpc\_\_bind (&matrix\_id, &saddr, slen, &st);
matrix\_multiply (h, a, b, result, &st);
```

4.64.7 Using Bound-to-Host Mode upon Server Failure

A client typically uses the second method, bount-to-host handle, invoking

rpc__clear_server_binding

after it has received an

rpc__wrong_boot_time

error. If a client is fully bound to a server that exits and then restarts, listening to a new port, the client can reset the binding to the new port by calling

rpc__clear_server_binding

on the existing handle.

4.64.8 Bound-to-Host Asset

Bound to host handles are most efficient when a client already knows the name or address of a host that is running the server it needs. The client does not need to do a lb lookup. The server needs to register with the llb on its hosts, but not with the glb.

4.64.9 Using Unbound Handles

The RPC RT lib broadcasts the call to all hosts on the local network. If a well-known port was specified in the definition of the requested interface, the call is broadcast to that port. Otherwise to the llb forwarding port.

4.64.10 Identifying Servers in Practice

```
rpc\_\_inq\_binding(h, &loc, &llen, &st);
socket\_\_to\_name(&loc, llen, name, &namelen, &port, &st);
name[namelen] = 0;
prinf("Bound to port %ld at host %s\n", port, name);
```

4.64.11 Server Crash

While handling a rpcall, an rpc__comm_failure is signaled to the client. If crash and restart between rpcalls, a rpc__wrong_boot_time. Applications are connectionless or maintain some state between calls, later case it has to unwind to the point at which it bound to the server.

4.64.12 Interface Mismatches

rpc__unk_if requesting a new op from an old server gives rpc__op_rng_error, so add a cleanup handler.

4.64.13 Using Cleanup Handlers

```
fm\_\_cleanup\_rec clrec;  // set the cleanup handler
st = fm\_\_cleanup(&clrec);  // test the return value

// if error clean up
if (st.all != fm\_cleanup\_set) {
if (st.all == rpc\_\_op\_rng\_error) {
```

```
//find an out-of-date server; find another one and rebind
fm\_\_reset\_cleanup(&clrec, &st);
} else {
// report the error and exit
fm\_\_signal(st);
}
}
// otherwise proceed normally
if\_newop(h, input, &output); // call the operation
fm\_\_rls\_cleanup(&clrec, &st); // release the cleanup handler
```

4.64.14 Portability Considerations

The fm package uses the C routines setjmp and longjmp to implement cleanup handlers. If one have local variables in fault handling code, some optimizing compilers may generate errant object code. To ensure that modifications made to the local variable in the normal code path are visible to the fault handling code, the variable should be declared with the ANSI C volatile qualifier.

4.64.15 A Macro for Portability

Define a portable Volatile macro since volatile is not yet supported by all C compilers. Translates to volatile on systems whose compilers support the qualifier; on other systems it is null.

```
Volatile boolean flag;
flag = false;
st = fm\_\_cleanup(&crec);
if (st.all != fm\_\_cleanup\_set) {
if (flag)
release\_pkt(pkt);
fm\_\_signal(st);
}
pkt = allocate\_pkt();
flag = true;
// more code
// if a fault occurs here, the value of flag is indeterminate
//  more code
```

```
fm\_\_rls\_cleanup(&crec, &st);
```

The comm_status parameter attribute sets an exception handler If you specify comm_status for an operation parameter, the idl compiler puts a cleanup handler in the client stub routine for the operation. The cleanup handler catches any error with the rpc__mod module code and passes the error to the client in the status parameter.

4.64.16 A Utilitarian Module

```
#include appli.h

char *error\_text(st)
status\_\_t st;
{
static char buff[200];
extern char *error\_\_c\_text();

return (error\_\_c\_text(st, buff, sizeof buff));
}
```

4.64.17 Server Structure

Contains :
- header file .h form idl compiler
- server initialization code, which registers the interface with the RPC RT lib and the lb.
- the manager code, which implements the operations in the interface.
- the server stub generated
- any user-written code that performs data type conversion.

4.64.18 Initialization Code

- process any argument supplied in the command line.
- creates the sockets on which it will listen.
- register the server's objects and managers with the RPC RT lib.
- register the server's objects and interfaces with the lb

- establish termination and fault handling conditions
- begins listening for requests.

```
// processing arguments
family = socket\_\_family\_from\_name((ndr\_\_char *) argv[1],
(long)strlen(argv[1], &st);

validfamily = socket\_\_valid\_family(family, &st);
if (!validfamily) {
printf("Family %s is not valid\n", argv[1]);
exit(1);
}



// creating sockets
rpc\_\_use\_family(family, &loc, &llen, &st);

// registering with the RPC RT lib
rpc\_\_register\_mgr(
&uuid\_\_nil,
&appli\_v1\_if\_spec,
appli\_v1\_server\_epv,
(rpc\_\_mgr\_epv\_t)&appli\_v1\_manager\_epv,
&st);

rpc\_\_register\_object // if needed

// registering with the location broker
lb\_\_register(&uuid\_\_nil, &uuid\_\_nil,
        &appli\_v1\_if\_spec.id,
(long)lb\_\_server\_flag\_local,
         (ndr\_\_char *) "example",
&loc, llen, &lb\_entry, &st);

// unregistering and fault handling
st = fm\_\_cleanup(&crec);
```

```
if (st.all != fm\_\_cleanup\_set) {
status\_\_t stat;
fprintf(stderr, "Server receives signal - %s\n",
error\_text(st));
lb\_\_unregister(&lb\_entry, &stat);
rpc\_\_unregister(&appli\_v1\_if\_spec, &stat);
fm\_\_signal(st);
}

// listening for requests
rpc\_\_listen((long) 5, &st);
```

4.64.19 Manager Code

A manager epv names the routines that implement the operations in an interface.
// defining manager epvs global appli_v1_epv_t appli_v1_manager_epv = appli_add;
// identifying objects // if manually bound rpc__inq_object // auto bind // the generic handle must be either the object uuid itself // or some other data type from which the manager can // determine the uuid.
// identifying clients

```
rpc\_\_inq\_binding(h, &loc, &llen, &st);
socket\_\_to\_name(&loc, llen, name, &namelen, &port, &st);
name[namelen] = 0;
printf("Request from port %ld at host %s\n", port, name);
```

// register and unregister transient objects as needed
// initializing status parameters // if an operation has a status parameter (comm_statts // attribute), the manager routine that implements the // operation should set the status parameter to status__ok // before it returns.

5 Standardizing Object Management

Here is our second step to exploring network softwares interactions. The challenge is to have interoperable programs from everywhere. The client/server RPC model is mute about the glb. Its ambition is intranet application scoped. URL are uuid

without replica and without reality. How to FREELY exchange objects of any kind (multimedia) on a wide scale (internet)?
The OMG moto is to exchange computer ressources only on an intranet basis, and compatible software packages on the internet. Too short. But with an emphasis on the object approach.
On the road to Java, we leave now these implementation issues. Looking at CORBA, I will develop showing that the object-oriented approach has no real pay-off on the back-end implementation side, but may have on the front-end side, as a standard.
Where does Java come from? What is it, a language, a standard, a technology? Will other similar technology arise? What about the elder OMG technology? I devote some lines to examine it before entering Java world.

5.1 OMG's Vision of the Future of Computing

> The CPU as in island, contained and valuable in itself, is dying in the nineties. [the end of autonomous systems? but the rise of networked systems?]
>
> The next paradigm of computing is distributed. [and next ? the extinction of computing and the reign of (embedded) products?]
>
> This is driven by the very real demand of corporations recognizing information as an asset, perhaps their most important asset. [isn'it the very definition of "industry"?].
>
> To make use of information effectively, it must be accurate and accessible across the departement [intranet] , even across the world [internet]. [why is there no extranet?].
>
> This means that CPUs must be intimately linked to the networks of the world [thus becaming a "host"]
>
> and be capable of freely [the key word here] passing and receiving information, not hidden behind glass and cooling ducts or the complexities of the software that drives them [say : pretty print a Word file in Mac format on a Unix xterm].

5.2 Current problems

> The major hurdles in entering this new world are provided by software:
>
> - the time to develop it [once one said all has been already written; is there still anything to develop?],

> - the ability to maintain and enhance it [justification of many departements all around],
>
> - the limits on how complex a given problem can be in order to be profitably [the key word] produced and sold [embedded technology limit problem?],
>
> - and the time it takes to learn to use it.
>
> This leads to the major issue facing corporate information systems today : the quality, cost, and lack of interoperability of software.
>
> While the hardware costs are plummetting [true? why?],
>
> software expenses are rising. [true? why?] [where is the difference?]] ...
>
> As systems departments require information among a diversity of in-house, brought-in, supplier, customer, and commercial applications, those applications become increasingly difficult and complex . [why not simply buy the last desk-suite from Micro$oft?].

The OMG solution to stay alive life in the networked jungle OMG = Object Management Group

5.2.1 The Object Paradigm as Encapsulation

The OMG defines the object management paradigm as the ability to encapsulate data and methods for software development.

5.2.2 Complexity

The idea is to reduce complexity by standardization and mimicking the "real world", lower costs from these, hasten the introduction of new software applications with an object-oriented approach to software construction.

> computing should be viewed by the users as "my" world, with no artificial barriers of os, hardware architecture, network compatibility or application incompatibility. Procedural construction has never really modeled the "real" world. The real world approach is quite simply a commodity approach where software is designed and built from logical components, snapped together with "requests" where the sum of the parts is equal to, and many times greater than, the whole.

> The reality of distributed network computing and object-oriented products is here. The job now is to develop standardization for tools and applications that reduce the effort it takes to build the applications of tomorrow.

The idea that procedural productions are more complex than the "real world" is to be proved. Does the object paradigm better model the "reality"? Systems showing the desk metaphor can be built without object approach. ex: the Amiga DOS.

5.2.3 The Illusion of the Standard

The fact that standards reduce costs and management complexity is a truism, but the cost of bad adaptation is not evident. A standard, but which one? Current industrial standards first remove unnecessary details and unaccurracy from the grid. [to have everywhere access to a standard object to be used in a standard way whatever be the model of the main investment] [is it a mere problem of bolts and knots standard?]

5.3 Overall Technical Goal of Object Management

> To adopt interface and protocol specifications that define an object management architecture supporting interoperable applications based on distributed interoperating objects.

Two steps from DCE are a methodological one, the object approach to software design, and a practical one, the interoperability of applications.
Conformance to the OMG Object Model :
- inheritance of interface, i.e. conformance of interface types.
- inheritance of implementation
- a program or process should be able to define and execute one or more methods [same as DCE]
- the methods of an object may be defined and executed in one or more processes. [same as DCE]

5.4 Object Distribution Transparency

A client object using a server object thru its interface should be independent of the server object's:

5.4.1 Location

Physical location of the objects's methods and associated state. [DCE can]

5.4.2 Access Path

- The path between the client and the server objects

5.4.3 Relocation

The movement of the data or methods of the objects to new locations. It should not require that a client object be modified in order to use the relocated server object.

5.4.4 Representation

The format of the data and methods associated with the object; purely implementation matter; if the internal representation of an object is changed, this should not affect its interface or client behavior.

5.4.5 Communication Mechanism

How communication with the process that is performaing a request is affected. Which interprocess comm mechanisms are being used at any given time to communicate object requests may depend of the relative loactions. A client object should be able to send requests to a server object without needing to be aware of the specificundelying comm mechanisms or protocols used.

5.4.6 Invocation Mechanisms

How methods associated with the object are executed; different os supply different model and mechanisms for controlling computer processes.

5.4.7 Storage Mechanisms

How the data and methods of the object ares stored and managed.

5.4.8 Machine Type

Nothing is said.

5.4.9 OS

Nothing is said.

5.4.10 Programming Language

Nothing is said.

5.4.11 Security Mechanism

Inforce to control access to the object; should not affect the interface of the object. Changes in any of the above should not require a client to be recompiled, relinked, or reloaded in ordre to maintain the distribution transaprency of the server object. Thus OMG makes most of the DCE client/server code migrate to an underlying layer which encapsulate all the details of networked distribution.

5.4.12 Performance

For local and remote server objects requests. Perormance should concern :
- scalability as the number of objects accessible to a client increases and the number of requests handled increases.
- method invocation time
- storage overhead
- resource consumption
- parallelism
- throughput

5.4.13 Extensible and Dynamic Nature

Such changes should not require other objects to be recompiled, relinked, or reloaded, or the shutdown of the environment.
- adding new w implementations (classes)

- replacing implementations where the interface is not changed, adding, replacing and deleting methods or data structures, changing the type of an attribute or method, renaming classes, attributes , and methods.

- replacing implementations where the interface is changed (secondary goal); a mechanism should be provided, based on versioning?, such that existing objects can continue to use older implementations. Alternatively, a dynamic upgrade facility might

allow existing objects to correctly use the new interface.
- changing the location of implementation (secondary) any part of the implementation of an object (methods, attributes, or storage of object relationships) should not affect other objects.

5.4.14 More than One Name Space

The DCE name space is unique. OMG find necessary to provide a naming system architecture that supports multiple naming contexts. A naming system allows names to be mapped to objects in a context.
The architecture should facilitate :

5.4.15 Name Spaces

In which objects can be named unambiguously; an object can have more e than one name; can change over time; be considered attributes of an object; the values (objects) to which the names are mapped can also change.

5.4.16 Object Handles

The notion of an object handle which is unique for an object is supported.

5.4.17 Efficient Reference to Objects

5.4.18 Queries

Based on object names, attr, and relationships with other objects; support for : - implementation info including class and method - type info - interface info for any given object - naming system data

5.4.19 Access Control

Support discretionary access control of objects means that access control can be set by an object's owner and cannot normally be circumvented.
- AC should be able to be set on objects controlled by servers; an object's data should not normally be accessible by any means other than the methods of its class.
- ac provided for metaobject data including implementation and interface info, and naming system data. - mandatory ac.

Provide concurrency control for objects support

5.4.20 Concurrency Control

The concurrency control should be implicit; server objects are responsible for mediating access to themselves. Stated conversely, client objects should not need to perform explicit actions to insure the consistency of the objects they access.
-

5.4.21 Parallelism Policy

It should minimize the availability of a server object due to its use by another client object; ex: could queue requests, or support parallel threads of execution, or support locking of only affected sub-objects or sub-components.

5.4.22 Open-Ended-ness

It should admit different type of use, such as read/write

5.4.23 Errors

Recovery from abnormal conditions should be consistent. Provide robust operation and a high level of availability - notification to client objects of abnormal terminations - recovery back to a consistent state.

5.4.24 Versioning

Provide support for the versioning of objects - versioning of classes with existing instances

5.4.25 Events

Provide support for the notification of events to interested objects - register interest in an event - define and trigger event - receive notification of events in the same way as normal requests are received.

5.4.26 Handling of objects

Provide support for inter-object semantic relationships by holding object references within other objects.
Provide an application programming interface for all object management functions and capabilities with, as a minimum, a C binding.

Require minimal administration of the environment and facilitates management of the object base

5.4.27 Supervision

The environment should be self-administrating as far as possible. or support distributed admin.

5.4.28 Backups

Backups of all part of the object base should be able to be incremental with only short interruptions in the availability of objects.

5.4.29 Restore

Restore of objects from backups in such a way that non-restored objects continue to work (they may contain references to the restored object).

5.4.30 Locale

Support application internationalization :
- language
- culturally defined conventions
- dynamic switching

5.5 What is Missing

What is missing is only a set of standard interfaces for interoperable software components.
Introduction of an architectural framework with support of detailed interface specifications (ex : ORB specification, object storage, class structure, peripheral interface, user interface, object services, API for spellers, mailers, time managers).
This models the "real world" thru representations of program components called "objects". Results in faster application development, [wrong if interfaces are not intuitive ones and you have to read an inexistant doc] easier maintenance, reduced program complexity, and reusable components, and central benefit to grow in functionality thru an extension of existing components and the addition of new objects to the system. [is this unique to object management paradigm?]

> Object-oriented architectures will allow appications acquired from different sources and installed on different systems to freely exchange information
>
> OMG envisions a day where users of software start up applications as they start up theirs cars, with no more concern about the underlying structure of the objects they manipulate than the driver has about the molecular construction of gasoline. [bad metaphor, particularly with multi-type gas presently in Europe, euro95, euro98, super97, normal gas]

Application objects (computer simulated representations of real world objects) [this is true but for computer science and mathematics] are presented to end users as objects that can be manipulated in a way that is similar [key question : similar, same, or other?] to the manipulation of the real world objects. [who does not agree around?]

> It is easier to see and point than to remember and type.

[This is not established and not certain at all.]

5.6 The Object Model for Architecture Compliance

5.6.1 A Common Semantic

For common objects. A common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In an OMA-compliant system, they perform operations and maintain state for objects having a common semantics.

5.6.2 The Basic Semantics

It is the client viewpoint. The object model first describes concepts that are meaningful to clients, including such concepts as object creation and identity, requests and operations, types and signatures.

5.6.3 Servers are Execution Semantics

It then describes concepts related to object implementations (in a more suggestive than specific way), including such concepts as methods, execution engines, and activation.

5.6.4 Sophisticated Systems are not Considered

There are some other characteristics of object systems that are outside the scope of the object model: compounds objects, links, copying of objects, change management, transactions, model of control, model of execution.

5.6.5 Object Management Plays Classics

OMA uses a classical object model, where a client sends a message to an object. Conceptually, the object interprets the message to decide what service to perform; as in classical object models, a distinguished first parameter is required, which identifies the operation to be performed; the interpretation of the message by the object involves selectioning a method based on the specified operation. In practice, method selection could be performed either by the object [the chimaera way] or the ORB [in the DCE way].

5.6.6 Execution and Construction

The object implementation model has two parts: the construction model, how services are defined and the execution model, describes how services are performed. The execution model determines the construction model. A computational object system must provide mechanisms for realizing behavior of request in concrete actions, the construction model.

5.6.7 Activation is a Necessary Step

Performing a requested service causes a method to execute. This may operate upon an objects's persistent state. If the persistent form of the method or state is not accessible to the execution engine, it may be necessary to first copy the method or state into an execution context. This process is called activation.

5.6.8 Operation Signatures are the Interface

The externally visible characteristics of objects are described by an interface which consists of operation signatures. [true of C++ and DCE] The external view of both object behavior and object state (info needed to alter the outcome of a subsequent operation) are modeled in terms of operation signatures. [true of DCE].
The OMA Object Model define a "core" set of requirements to comply; called the Core Object Model. Core components are not-required extensions.

5.7 The Core Object Model

Basic concepts are : objects, operations, types and subtyping.

5.7.1 Non-Object Types

The core does not specify a set of non-object types.
Objects and non-objects collectively represent the set of denotable values in the Object Model. In a pure object system, such a Smalltalk, all denotable values are expressed as objects and so the set of non-object types is empty; non-object types are defined in a Component and chosen for inclusion in a Profile.
It does not exist a supertype of Objects and the types of non-objects. Therefore, one cannot specify an operation parameter that may be either an object or a non-object. This eliminates the difficulty of systems having to provide run-time discriminations of objects and non-objects.

5.7.2 Exception Handling

It is not included in the Core Object model, intended to be introduced as a component and can therefore be included in profiles.

5.7.3 Things that are Not Core Objects

In the Core, operations (definitions of signatures) are not objects. Requests (operation invocations) are not defined to be objects.

5.7.4 Confusion Between Subtyping and Inheritance

Many objects systems do not distinguish between subtyping and inheritance.

5.7.5 Subtyping is a Modus Vivendi Rule

Subtyping is a relationship between types based on their interfaces, and a relation between interfaces (types). It defines the rules by which objects of say one type are determined to be acceptable in contexts expecting another type.

5.7.6 Inheritance Saves Work

Inheritance is a mechanism for reuse. It allows a type to be defined in terms of another type. It is a notational mechanism.

5.7.7 Inheritance is Mono and Constant

The Core Object Model restricts objects to be direct instances of exactly one type, (its "immediate" type). The Core has no mechanism for an object to change its immediate type.

5.7.8 One Inherit Two Things

Inheritance can apply to both interfaces and implementations. The Core is concerned with inheritance of interfaces only.

5.7.9 Subtyping Implies Inheritance

The Core relates subtyping and inheritance. If S is declared to be a subtype of T, then S also inherits from T. The core supports multiple inheritance, but does not provide a name conflict resolution mechanism nor does it allow subtypes to redefine inherited operation signatures. These two constraints are relaxed in a component.

5.7.10 Operation Dispatching

In the Core, the process of selecting an implementation to invoke is based on the type of the object supplied as the controlling argument of the actual call. That can be relaxed in a component, generalized to multiple argument dispatching. The operation of the given name defined on the immediate type of the controlling argument is chosen for invocation.

5.7.11 Arguments

The argument passing semantics is unique. The Core defines a pass-by-value argument passing semantics.

5.7.12 A Unique Face

The interfaces of a type are all same mode The Core does not support private, public, subtype_visible and friends interfaces. These may be defined by components.

5.8 Implementation

The Core has nothing to say about implementation.

5.8.1 A Class is a Combination

The combination of a type specification and one of the implementations defined for that type is termed a class. An individual object, at any given point in time, is an instance of one class.

5.9 CORBA Principles

Principles of the Common Object Request Broker Architecture. The client emits a request to an object. This means it wishes to perform some operation on the object. The Object Implementation is the code and data that actually implements the object. The ORB is the mechanisms required to find the object implemantation 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 any aspect which is not reflected in the object's interface.

5.9.1 Is a Request Always First Bound to an Object?

No, a request may refer to a class of object – a printer –, without knowing any particular occurence or to no object at all. A client can use the stub or the dynamic invocation interface. The Core does not require a request to be delivered to any particular object.
The client of an object has an object reference that refers to that object. An object reference can be converted to a string that can be stored in files or preserved or communicated by different means and subsequently turned back into an object reference by the ORB that produced the string.

5.9.2 The Connection of the Implementation

The OI is not directly connected to the ORB The object adapter is the primary way that an object implementation accesses services provided by the ORB. ex: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, registration of implementations.
Conversely, an object adapter provides an interface to ORB services that is convenient for a particular style of object implementation.

5.9.3 The Travelof the Request

The request is not deposited directly to the OI An object implementation receives a request from the ORB core thru an IDL skeleton, build on a private ORB-dependent interface. Skeletons are specific to the interface and the object adapter. When processing the request, the OI may obtain services from the ORB thru an adapter it can choose.

5.9.4 Skeletons

For each particular language mapping there is an interface to the methods that implements each type of object. Generally an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them thru the skeleton.
It is possible to write an object adapter that does not use skeletons to invoke implementations methods.
An adapter exports a public interface to the object implementation and a private interface to the skeleton. It is build on a private ORB-dependent interface. With Object Adapters, it is possible for an Object Implementation to have access to a service whether or not it is implemented in the ORB Core. If not provided, the adapter must implement it on top of the ORB Core.
ex: it is common fo Object Implementation to want to store certain values in the object reference for easy identification of the object on an invocation. If the OA allows the implementation to specify such values when a new object is created, it may be able to store them in the object reference for those ORBs that permit it. If the ORB Core does not provide this feature, the Adapter would record the value in its own storage and provide it to the implementation on an invocation.

idl definitions	-> interface repository	-> client
idl definitions	-> stubs	-> client
idl definitions	-> skeletons	-> object implementation
impl. installation	-> impl.pository	-> object implementation

5.9.5 The Basic Object Adapter

Conventional implementations are generally separate programs. The basic adapter allows to have a program started per method, a separate program for each object, or a shared program for all instances of the object type. It provides a small amount of

persistent storage for each object, which can be used as a name or identifier for other storage, for access control lists or other object properties. If the implementation is not active when an invocation is performed, this adapter will start one.

5.9.6 The Library Object Adapter

An adapter for objects that have library implementations. It accesses persistent storage in files, does not support activation or authentification, since the objects are assumed to be in the clients program.

5.9.7 The Database Adapter

This adapter uses a connection to an o.-o. database to provide access to the objects stored in it. Since the OODB provides the methods and persistent storage, objects may be registered implicily and no state is required in the object adapter.

5.9.8 The Interface Repository

A service that provides persistent objects that represent the IDL information in a form available at runtime. So it is possible for a program to use an object whose interface was not known at compiletime. May contain more: debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects, etc.

5.9.9 The Implementation Repository

Adds debugging info, administrative control, ressource allocation, security etc.

5.9.10 Models of Implementation Design

- Client and implementation-resident ORB.

- Server-based ORB.

- System-based ORB :
CORBA can be provided as a basic service of the underlying OS. This avoids marshalling when on the same machine, and unforgeable objrefs.

- Library-based ORB :
for objects that are light-weight and whose implementation can be shared, the implementation might actually be in a library. The stubs are then the actual methods. This assumes that it is possible for a client program to get access to the data for the objects and that the implementation trusts the client not to damage the data.

5.10 The Failure of DCE as a Global Solution

Let us look why DCE is not a solution as a set of standard interfaces for interoperable and portable software components DCE does not provide a reference architecture with terms and definitions upon which all specifications are based, focusing on distributed applications, distributed services, and common facilities.
It is envisonned that

> ...common shared objects lead to a uniform and consistent user interface... allows for increased focus on application creation rather than application education.

It is also put forth that

> ...the use of object-oriented concepts is the gateway thru which even the most compuphobic person can pass. It has the potential to enable users to control their computing environment, rather than being controlled by its limitations.

5.11 OMA Object Management Architecture

5.11.1 The ORB Component

The ORB component of the OMA (commercially referred to as CORBA) is the communication heart of the standard. It provides an infrastructure allowing objects to communicate, independent of the specific platform, a DCE. But techniques used to implement the addressed objets are from a global consensus, which DCE lacks.
The ORB component will by construction guarantee portability and interoperability of objects over a nework of heterogeneous systems.
Other components of the architecture

5.11.2 The Object Services Components

They standardize the life cycle management of objects, creation ofe objects (The Object Factory), controlling access to objects, keep track of relocated objects, controlling the relationship between species of objects (Class Management).

5.11.3 The Common Facilities

This is a set of generic application functions such as rinting, database, e-mail.

5.11.4 The Application Objects

They perform specific task for users. They are derived from Object Services objects and Common facilities objects (by generalization and specialization).

5.12 The ORB Specific Duties

The ORB is expected to - name services object location services use the object names in the request to locate the method to perform the requested operation. This may be simple attribute lookups on objects.
- request dispatch This function determines which method to invoke. The Core does not require a request to be delivered to any particular object.
- do parameter encoding This is to convey the local representation of parameters alues in the requester's environment to equivalent representations in the recipients's environment. de facto standards : OSF/DCE, ASN.1, ONC/NFS/XDR, NCA/NCS/NDR
- do delivery Requests and results must be delivered to the proper location as node, address, space, thread or entry point. may use standard protocols TCP/UDP/IP, ISO/TPn
- do synchronization This manages the parallelism of the objects making and processing a request and the rendezvousing of the requester with the response to the request, among three modes, namely aynchronous (request with no response), synchronous (request; await reply) anddeferred synchronous (proceed after sending request; claim reply later).
- do activation the housekeeping preocessing necessary before a method can be invoked.
- handle exceptions
- run security mechanisms

5.13 An Example of ORB Function

Let us consider request = "print cg_layout666 laser_printer". This could be send to the object "cg_layout666" whose "print" method would then print it on "laser_printer" or could be send to the object "laser_printer" whose "print" method would access "cg_layout666" or could be send to the object "print" etc.
Or, instead of relying on such a generalized "print" routine, the Name Service inside the ORB could determine an appropriate method jointly owned by the classes of "cg_layout666" and "laser_printer".
The most important feature of the CORBA is its interface definition language, used by applications to specify the various interfaces they intend to offer to other applications via the ORB layer.
The operations provided by Object Services are made available thru the ORB, or other interfaces. They are class management, instance management, storage, integrity, security, query, versions.
The Common Facilities are optional A service becomes a CF when it communicate using the ORB, implements a facility that OMA chooses to adopt, and has an OMA-compliant object interface
Examples of common facilities cataloging and browsing of classes and objects, link management, printing and spooling, error reporting, help facility, e-mail facility, tutorials and computer-based training, common acces to remote information repositories, intelligent macro agent, interface to external systems, object querying facilities, user preferences and profiles.
example of possible requests: edit_ascii_text(editor_object, text_string)
query_database(query_object, sql_string) to a SQL-compliant query processor.
Application objects An OMA-compliant application consists of a collection of interworking objects.

6 Basic Characteristics

6.1 The Need For a Standard Network OS

CORBA defines language bindings which allow, at least theoretically, to build interfaces for other networked programs. This supposes that we gain control on a complex set of carefully made compatible tools to make this work. Alas! We can't easily mix even source code coming from different environments, let alone binaries! Even if CORBA works, objects, once contrived to be portable are mostly isolated

dinosaurs. Because there is no standard network OS associated, they can't use much elaborate services from the host network.
This means a lot of wheels and barrows need to be reinvented from scratch.

6.2 Interaction of Network Tools with Local Ressources

To get access to a local hard drive, one must have installed a server, which enables foreign clients to get in and read or write files (FTP), or to get connected (TELNET) to the host's OS.
If one's host is a mere client, it can only emit requests to read external ressources, but will stay mute to external requests.
Cookies have an other meaning : they enable applications or 'applets', which are loaded in documents to use a very limited set of local storage. The limit comes from the fact that cookies cannot have a size above 4 kbytes. More, they number is also limited. The control is in charge of the browser. Last, they can only be registered in a specific directory.
But this is clearly theory : Netscape browser had a flaw that enabled to go and read files anywhere in the client store. Note that we must keep a watch over ActiveX and Java Micro$oft developments, as these people strive to derogate that security principle.

6.3 What Should a Net OS Be?

A comprehensive offering, one that addresses more of the problems of building networked applications should be : full network operating system including a kernel, programming language, communications protocols, libraries, security and authentication, naming protocols, APIs, and so on.
A programming language, even with its libraries, is not enough solve all the problems of the network. When a network program begins execution, it should have a *fixed interface* to services like security and authentication, naming protocols, directory services, network interfaces, etc. Therefore inherently it is a more portable solution than any language-only solution. The network API should be independent of the hosting operating system or the network itself.
Compression and encryption should be done at the lowest levels of the system. Application should use them uniformly, or best remain unaware of their presence.
The power of the system must extend beyond applications. The system must be network coherent and be usable to to manage the network itself, servicing all possible

network elements. The system shoul dprovide the means to configure the execution of any program , clients as well as servers, from pieces anywhere in the network.

6.4 The Object Orientation

OS functionalities such as file services and network services should be made fromf Customizable Composite Objects. These objects should be especially suitable for building on microkernel-based OS, such that it should possible to implement customized operating system services at user-level.

6.5 The System Language Choice

We do not want to renounce to use C-like syntax. We need garbage collection to manage memory. We need to support the Unicode character set.
We are not C++ addicts; C++ has too complex object-oriented features whose utility is extremely doubtful in pratice. But we recognize the gain of using the object model to provide interfaces to system services. C++ lacks fundamental programming primitives such as threads and simple communication channels, and lacks the list basic type. C++ does not permit to select implementation at run time. An application should configure dynamically, loading only those modules needed at any moment; it should load and even unload modules under environment or flow control. Modules should be run remotely anywhere, without the need to run locally a service interface, say DNS or Rlogin.

6.6 The Abstract Machinery

The virtual machine should be suited to koban (just-in-time) compilation. The machine should have a memory-to-memory architecture, not a stack one.Thus translation to native instruction sets will be easier. The compilers should be kept imperatively small.

6.7 The Adaptation to Multimedia

Existing codecs for playing multimedia stuff should be written as modules, so that the existing C-language or C++-language implementations, once rewritten, which should be easy, will be completely portable. The built-in multithreading, say Modula 3, should simplify the implementation of scheduling in quasi-real-time codecs.

6.8 The Integration of Previous DCE

The system, un like DCE, or OVM, should be designed to provide inherentlythe perfect environment for networked applications. The experience of DCE should help to simplify such bloated OS presently driving most PCs. The networking capabilities should be part the OS structure rather than as a separate library.

6.9 Adaptation to Embedded Systems

Many voices claim that Embedded Systems will be a huge marketplace. Let us review the differences between Implementation of Java in Embedded Systems an embedded system and a general-purpose computer or work-station.

- An embedded system usually lacks secondary storage (e.g. a hard disk) making it difficult to keep a library of application code and to load a particular routine upon demand. Therefore, the size of the application is limited to the size of the existing memory device (such as ROM or Flash).

- In most cases, an embedded system lacks direct user interface such as a monitor. Rather, it communicates with the operator (if at all) through some specialized devices or through a front-end station which is connected to the embedded application via a specialized bus or network.

- The lack of a file-system and traditional user interface devices has a special significance for high-based applications which rely heavily on the presence of these components. Often, an embedded application has limited resources such as memory and CPU bandwidth. Consequently, the scheduling and sharing of these resources become one of the major design challenges when developing such an application. Particularly, in Real-Time applications, the performance requirements and the constraints on resources lead to lean and fine-tuned scheduling disciplines in order to meet the system's performance requirements.

- Typically, an embedded system is very specialized; it is tightly integrated with the surrounding environment and has demanding requirements for robustness. Since the protection mechanisms (such as address-space boundary protection) that are usually provided by a typical operating system are not available in such an environment, the mechanism of introducing new software into the application should be tightly controlled. This fact has special significance in the context of Java where the programming paradigm is based on dynamic downloading of applets from various sources. Java was designed (and initially

implemented) to run on UNIX workstations. It relies on many services that are available on such a system (such as files, processes, Internet naming services) which do not exist in a typical embedded system. Some of these do not make sense in such an environment (for example, a network device might have an IP-address, but it does not necessarily have a domain name).

6.9.1 Implementation Architectures

In order to execute byte code on an embedded system, several key components must be developed or ported to this system. These components have to be tailored to the specific hardware, to the kernel (if one exists) and to some extent to the C compiler being used (since part of the VM and its interface layers are implemented in C). The two main possible approaches are:

- Developing a dedicated language-compatible system, exactly fitted to the embedded environment.
- Porting Sun's original Java to the embedded environment.

The first approach has the advantage of ending with a product which is originally designed to the embedded environment, including all the constraints of this special environment. The second approach has the advantage of compatibility to the de-facto standard which has been already adopted by the industry.

6.9.2 Virtual Machine Encapsulation

As mentioned previously, the ideal technique would be the one which enables the embedded network software to run on top of any commercial Real-Time kernel or be adapted to any application-specific executive. This capability is accomplished by having a well-defined API that encapsulates the services that the VM needs from the underlying system. The system-language supports parallelism in the form of threads (light-weight tasking). It includes methods for thread management, synchronization and communication. There are two approaches for mapping the language threads to the parallel entities of the underlying kernel (typically, they are called tasks): a. Mapping each language thread to a kernel task and utilizing the kernel services to schedule these threads and synchronize between them. b. Dedicating one kernel task to run the VM and then have the langugae threads be implemented internally by the languagerun-time support system. In the first model, the services of the Real-Time kernel are used directly for managing the language threads and can, in

some situations, perform better. In addition, language threads can interact directly with other tasks in the system. However, this approach requires that the overall system be fully tested each time a new langugae thread is introduced or a change to the langugaecode is performed. In the second approach, the VM acts also as an intermediate monitor, which internally manages the language threads and ensures that the language code as a whole performs accurately within its resource boundary. It therefore ensures that in any circumstances the VM does not exceed the amount of system resources allocated to the language application. The practical meaning of this implementation approach is that once the VM has been fully tested in a particular environment, the different applets can be downloaded and executed with no need to re-test.

6.10 The OS should be Ubiquitous and Free

The operating system should be capable of deployment on a variety of host platforms, ranging from personal and embedded computing devices, up to and beyond multi-user workstations and servers.
Work has already begun on prototypes and several proposals are currently under evaluation. An initial working nano kernel should be a first asset.

6.11 An International Standard Language

It should make software easily portable and distributable.

6.12 An International Standard Object Management

Such a standard should make the integration of this software into local and personal applications an easy task.

7 Java

Java is a programming language. Looking at just the language component, however, Java and Limbo have some common ideas and are similar in many ways : both use C-like syntax, both compile to a virtual machine that can be interpreted or compiled on-the-fly for portable execution, both use garbage collection to manage memory, and both support the Unicode character set. It makes sense for Java applications to run under Inferno, where they can gain the advantage of uniform access to system services.

7.1 The Java Language

The Java language is a new(?) object-oriented programming language, developed by Sun Microsystems. It is to the development of portable application pieces of software, called applets. These are dynamically downloaded over the network and executed on the Java-VM-Enabled client computer, equipped for this purpose with a Java-Enabled Web Browser functionality.

7.2 The Java Components

The Java system compiles into Byte Code. Once compiled, the byte code is placed in a file ready to be downloaded through the network to the client computer that requests it. The VM component is responsible for actually running the byte code and it isolates the code from the specifics of the underlying hardware and OS. Consequently, Java applets are developed once but they can run, unchanged, on any client computer with a Java-VM-Enabled Web Browser functionality, without no more concern of the specific platform hardware.

7.3 JAVA and CORBA May Complement

JAVA makes software easily portable and distributable. CORBA is fit to make easy, at least in theory, to integrate this software into your local and personal applications.

8 Inferno

Inferno is a full network operating system. Inferno includes a programming language, called Limbo, it may support others, and Java is an obvious candidate.

8.1 The Language of Inferno

Limbo differs from Java in several important ways. For Limbo, concurrency and communication are an intrinsic part of the language and the virtual machine, and are used extensively in the programming model. Limbo has more general safe pointers. Limbo's garbage collector has constant-time overhead. Limbo programs are built as sets of modules that export interfaces. Limbo has numerous libraries, packaged as modules. The current set covers the same ground as Java's libraries.

8.2 The Limbo Virtual Machine

The machine DIS is has a memory-to-memory architecture. Many Dis instructions translate to a single Pentium instruction. The implementation of the compilers for Intel 386 architectures is only about 1300 lines of C. Code is said to compile with an under-factor of two with compiled C. For the same reason, the Dis interpreter is considerably smaller than the implementation of the Java virtual machine.

9 Landmarks

We found six landmarks to watch over, namely : CORBA/UML, Linux/BSD, Java, Inferno, ActiveX and embedded systems.
Is there an opportunity now and here for any new OS/language combination that does not emulate Linux/BSD and Javas distribution? These softwares are free.
Is Java just a big hype? We are beginning to have doubts about the Pandore's-box-library choice for a distributed, write-one-run-anywhere environment.
Win95 is an impressive product as an OS effort to compete the unix side of the galaxy; it has flaws. We consider taht everyone uses it is because it comes preinstalled with PCs, not for its intrinsic behavior.

9.1 The Commercial Success of Linux

More than 10 millions platforms are supposed to run Linux/BSD at this date. An international hackers'effort gives Linux/BSD the biggest fast development potential. Recent efforts towards ease of installation maintenance leave few advantages to closed-minded Win95.

References

[1] Hamilton G., Khalidi Y., Nelson M., "Why object oriented operating systems are boring", IEEE [WO 190], 1991, pp. 118-119.

[2] Business Engineering With Object Technology, David A. Taylor, Wiley, 1995, ISBN 0-471-04521-7

[3] UML Distilled: Applying the Standard Object Modelling Language, Martin Fowler & K. Scott, Addison-Wesley, 1997, ISBN 0-201-32563-2

[4] Rapid Development, Steve McConnell, Microsoft Press, 1996, ISBN 1-5515-900-5

[5] Designing Object-Oriented Software, Prentice Hall, 1990, ISBN 0213

[6] Mike Kong, Network Computing System Reference Manual, Prentice Hall, 1990.

[7] Nutt G., Open systems, Prentice-Hall, 1990, ISBN 0-136-36234-6.

[8] Analysis Patterns, Martin Fowler, Addison-Wesley, 1997, ISBN 0-201-89542-0

[9] Bal H. & Grune D., Programming Languages Essentials, Addison-Wesley, 1994, ISBN 0-201-63179-2.

[10] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995, ISBN 0-201-63361-2

[11] B. Meyer, "Applying Design By Contract", IEEE Computer, 25,10, 1992 , page 40-51

[12] The Essential Client / Server Survival Guide (2nd Edition), Robert Orfali, Dan Harkey, Jeri Edwards, John Wiley, 1996, ISBN 0-471-15325-7

[13] Object-Oriented Software Engineering, Ivar Jackobson, Addison Wesley, 1992, ISBN 0-201-54435-0

[14] Object-Oriented Analysis and Design With Applications, Grady Booch, Benjamin Cummings, 1993, ISBN: 08053534020

[15] Object-Oriented Modeling and Design, Jim Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen W., Prentice Hall, 1991, ISBN: 0136298419

[16] Migrating to Object Technology, Graham I., Addison-Wesley, Wokingham UK, 199

[17] Migrating Legacy Systems - Gatewaya, interfaces and the incremental approach, Michael L. Brodie, Michael Stonebraker, Morgan Kaufman Publishers, 1995, ISBN: 1558603301

[18] Pattern-Oriented Software Architecture : A System of Patterns, Frank Buschmann, Regine Meunier, Hans Rohnert, Pet Sommerlad, John Wiley & Sons, 1996, ISBN: 0471958697

[19] Software Architecture : Pespectives on an Emerging Discipline, Mary Shaw, David Garlan, Prentice Hall, 1996, ISBN: 0131829572

[20] Succeeding with Objects: Decision Frameworks for Project Management, Ad=E8le Goldberg & K.S. Rubin, Addison-Wesley, 1995,

[21] Object-oriented Methods: a Foundation, James Martin & Jim Odell, Prentice Hall, 1995, ISBN: 0213

[22] Designing Object-Oriented Systems: Object-oriented Modeling with Syntropy, S. Cook & J. Daniels, Prentice Hall, 1994, ISBN: ?

[23] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", IEEE Computer, May 1988, IEEE, pp.61-72

[24] Beck K and Cunningham W "A laboratory for teaching object-oriented thinking" , Procedings of OOPSLA 89.

[25] B. Meyer, "Applying Design By Contract", IEEE Computer, 25,10, 1992 , page 40-51.

[26] Philippe Kruchten, "The 4+1 View Model of Architecture", IEEE Software, 12 (6), November 1995, IEEE, pp.42-50.

[27] ?, "Using goal based use cases", Journal of Object-Oriented Programming, Nov/Dec 1997, Vol. 10, No. 7

[28] Bibliograpy (Implementation), Active Visual Basic 5.0, Guy Eddon, Henry Eddon, Microsoft Press, 1997, ISBN 1-57231-512-1

[29] DCOM Programming, Dr Richard Grimes, WROX Press Ltd., 1997, ISBN 1-8610000-60-X

[30] Inside COM, Dale Rogerson, Microsoft Press, 1997, ISBN 1-57231-349-8

[31] Inside OLE, Second Edition, Kraig Brockschmidt, Microsoft Press, 1995, ISBN 1-55615-843-5

[32] Understanding ActiveX and OLE, David Chapell, Microsoft Press ,1996, ISBN 1-57231-216-5

[33] ourworld.compuserve.com/homepages/Martin_Fowler

[34] hillside.net/patterns/patterns.html

[35] www.rational.com/uml

[36] www.rational.com/support/techpapers/toratobjprcs

[37] www.microsoft.com

[38] www.microsoft.com/cominfo

[39] www.microsoft.com/sitebuilder/dna/default.asp

[40] www.microsoft.com/sitebuilder/dna/tech.asp

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803