

# A CORBA-based Interface-centric Approach to Signaling for IP-based Telephony Services

submitted by

**Tian Lu, M.Eng.**

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

**Master of Engineering,  
Telecommunication Technology Management**

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, K1S 5B6, Canada

February 28, 2001

© 2001, Tian Lu

The undersigned hereby recommend to  
The Faculty of Graduate Studies and Research  
acceptance of the thesis,

# A CORBA-based Interface-centric Approach to Signaling for IP-based Telephony Services

submitted by

Tian Lu, M.Eng.

In partial fulfillment of the requirements

For the degree of Master of Engineering, Telecommunication Technology Management

---

Dr. Bernard Pagurek, Thesis Supervisor

---

Chairman, Department of Systems and Computer Engineering

Carleton University

February 28, 2001

## **Abstract**

Convergence between the existing telephone networks and the emerging IP Telephony over the Internet not only demands software applications that span both networks, but also offers opportunities for innovative development approaches that satisfy the new requirements like fast product delivery, diversified customer services and decentralized network intelligence. In the distributed computing world, the maturing of Common Object Request Broker Architecture (CORBA) has offered built-in solutions for integrating legacy systems, as well as becoming an increasingly common element in telecommunication systems due to its ability to leverage emerging technologies. Recent research shows that CORBA is used to provide internetworking between various message-based protocols and management architectures [Berg1998, Fischbeck1999]. Moving forward from that, our research challenges the current heterogeneous message-centric approach for signaling, which generates tremendous tasks for software developers in terms of system interoperability, with a fresh approach for future IP-based telephony services.

In this thesis, we explore a new interface-centric approach using CORBA/Internet Inter-ORB Protocol (IIOP) as the signaling mechanism for IP Telephony as an alternative to the message-centric ITU-T H.323/H.245 Multimedia Control Protocol. We have converted Abstract Syntax Notation One (ASN.1) constructed H.245 messages into CORBA Interface Definition Language (IDL) data types, defined the IDL interfaces, implemented three sets of protocol procedures to achieve the basic functionality of H.245 messaging, followed with the integration of H.225 call set up and media transmission procedures. Finally, we address performance aspects of this approach to show the suitability of using CORBA-based signaling for enterprise applications.

## **Acknowledgements**

I would like to express my thanks to my supervisor, Professor Bernard Pagurek, for his guidance and encouragement in this research. I would like to give special thanks to Dr. Nilo Mitra, who is the principal system engineer from Ericsson Research Canada, for his profound supporting knowledge and rich industrial experience.

I am thankful to my parents and family members for their love, support and understanding during the course of my studies. Thanks to Jiang Tao for the understanding and love she gives me.

My colleagues at the Network Management Lab have been very helpful and supportive for which I am thankful. At the same time, I would also like to express my gratitude to all those who have helped me in this research. Thanks to Audrey Bufton for proofreading. Without her editing, the thesis would be unreadable.

This research work was supported by grants from Communication Information Technology Ontario (CITO) and Ericsson Research Canada.

# Table of Contents

<b>ABSTRACT.....</b>	<b>III</b>
<b>ACKNOWLEDGEMENTS.....</b>	<b>IV</b>
<b>TABLE OF CONTENTS .....</b>	<b>V</b>
<b>LIST OF FIGURES.....</b>	<b>VIII</b>
<b>LIST OF TABLES.....</b>	<b>IX</b>
<b>LIST OF LISTINGS.....</b>	<b>IX</b>
<b>LIST OF ACRONYMS .....</b>	<b>X</b>
<b>CHAPTER 1 INTRODUCTION.....</b>	<b>1</b>
1.1 THESIS MOTIVATION.....	2
<i>1.1.1 Problem Statement.....</i>	<i>2</i>
<i>1.1.2 Industrial Concerns and Trends .....</i>	<i>3</i>
1.2 THESIS OBJECTIVE.....	5
1.3 THESIS CONTRIBUTION .....	5
1.4 THESIS ORGANIZATION .....	7
<b>CHAPTER 2 BACKGROUND (ASN.1-BASED PROTOCOLS AND CORBA).....</b>	<b>8</b>
2.1 ASN.1-BASED PROTOCOLS.....	8
2.1.1 ASN.1.....	9
2.1.2 Encoding Rules.....	12
2.1.3 Widely-used Communication Protocols.....	16
2.1.4 Protocol Development Process.....	17
2.2 COMMON OBJECT REQUEST BROKER ARCHITECTURE .....	20
2.2.1 Object Management Architecture (OMA) .....	20
2.2.2 The CORBA Architecture Reference Model .....	22
2.2.3 General Inter-ORB Protocol (GIOP)/IIOP .....	24
2.2.4 Common Data Representation.....	28
2.2.5 Passing Object by Value (OBV) .....	29
2.2.6 CORBA Services.....	31
2.2.7 Extensions in CORBA 3 .....	32
2.3 CORBA IN THE TELECOMMUNICATIONS DOMAIN.....	33

2.3.1 Internetworking Gateway with IN.....	34
2.3.2 Internetworking Gateway with TMN.....	36
<b>CHAPTER 3 CONTROL PROTOCOLS FOR MULTIMEDIA COMMUNICATIONS .....</b>	<b>39</b>
3.1 ITU-T RECOMMENDATION H.323/H.245 .....	39
3.2 OMG CONTROL AND MANAGEMENT OF AUDIO/VIDEO STREAMS .....	43
3.3 DSM-CC AND DAVIC 1.4 SPECIFICATIONS .....	45
3.4 THE COMPARISON SUMMARY.....	47
<b>CHAPTER 4 THE INTERFACE CENTRIC APPROACH.....</b>	<b>49</b>
4.1 A COMPARISON OF INTEROPERABILITY REFERENCE POINTS .....	49
4.2 REQUIREMENT ANALYSIS.....	50
4.2.1 Transparency Requirements.....	50
4.2.2 Component Requirements .....	52
4.3 SELECTION OF TECHNIQUES .....	53
4.3.1 The Selection of CORBA versus DCOM and RMI .....	53
4.3.2 The Selection of Java in Telecommunications.....	54
4.4 OTHER ACTIVITIES TOWARDS OPEN INTERFACES .....	55
4.4.1 PARLAY.....	56
4.4.2 Java APIs for Integrated Networks (JAIN).....	58
4.4.3 Open Programming Interfaces for Networks (PIN).....	59
4.4.4 Summary on Future Control Infrastructures.....	61
<b>CHAPTER 5 DESIGN AND IMPLEMENTATION .....</b>	<b>63</b>
5.1 ASN.1 TO CORBA IDL TRANSLATION .....	63
5.2 MAPPING COMPLEX DATA TYPES.....	66
5.3 OBJECT MODELING AND PROGRAM DEVELOPMENT.....	68
5.4 CORBA REQUEST INVOCATIONS .....	72
5.4.1 Current Support for Requests.....	72
5.4.2 Asynchronous Messaging.....	72
5.5 USEFUL CORBA SERVICES.....	73
5.5.1 CORBA Naming Service .....	73
5.5.2 CORBA Trading Service .....	74
5.5.3 CORBA Event Service.....	76
5.6 VISIBROKER DEVELOPMENT ENVIRONMENT .....	77
5.6.1 Object Activation Service (Visibroker).....	78
5.6.2 Uniform Resource Locator (URL) Naming .....	79
5.6.3 Multithreading and Connection Management.....	81

5.7 INTEGRATION WITH OTHER H.323 CONTROL PROTOCOLS .....	83
<b>CHAPTER 6 CORBA PERFORMANCE EVALUATION.....</b>	<b>87</b>
6.1 PERFORMANCE OVERVIEW .....	88
6.1.1 IIOP Performance Limitations .....	88
6.1.2 GIOP Performance Implications.....	89
6.1.3 Setting Interceptors in Message Sequence .....	91
6.1.4 ORB Benchmarks.....	93
6.2 EXPERIMENTAL STRATEGY AND TEST ENVIRONMENT .....	94
6.3 PERFORMANCE RESULTS .....	96
6.3.1 Benchmark Test Results .....	96
6.3.2 H.245 Signaling Test Results.....	105
6.4 PERFORMANCE CONCLUSIONS.....	107
<b>CHAPTER 7 CONCLUSIONS AND FUTURE WORK .....</b>	<b>108</b>
7.1 CONCLUSIONS .....	108
7.2 FUTURE WORK .....	109
<b>REFERENCES.....</b>	<b>111</b>
<b>APPENDIX A: CONVERSION OF H.245 MESSAGE SYNTAX (ASN.1 TO IDL) .....</b>	<b>114</b>
<b>APPENDIX B: CORBA-BASED INTERFACE-CENTRIC APPROACH IMPLEMENTATION FOR H.323 SIGNALING (SCREEN SHOT).....</b>	<b>122</b>

## List of Figures

Figure 2.1 An example of Syntax Relationship (Abstract, Concrete and Transfer).....	11
Figure 2.2 BER and PER in ISO Object Registration Tree.....	13
Figure 2.3 BER Transfer Syntax.....	14
Figure 2.4 Example BER Format of the Tag Octets .....	14
Figure 2.5 PER Encoded H.245 Terminal Capability Set Request with Headers .....	16
Figure 2.6 Example of Protocol Development Process with ASN.1/C Compiler .....	19
Figure 2.7 Object Management Architecture Reference Model.....	21
Figure 2.8 CORBA 2.x Architecture Reference Model.....	22
Figure 2.9 Interworking between CORBA-based IN Application and Traditional IN Application (IN/CORBA Gateway) .....	35
Figure 2.10 Internetworking between CORBA-based IN Applications using SIOP (SS7 as Kernel Transport Network) .....	36
Figure 2.11 CORBA TMN Integrated Architecture .....	37
Figure 3.1 Messages in the Capability Exchange Signaling Entity.....	42
Figure 3.2 Messages in the Master/Slave Determination Signaling Entity .....	42
Figure 3.3 Messages in the Logical Channel Signaling Entity .....	43
Figure 3.4 A Basic Stream Architecture from OMG .....	45
Figure 3.5 DSM-CC Functional Reference Model.....	46
Figure 3.6 Application Portability and Service Interoperability Interface .....	47
Figure 4.1 Comparison of the Interoperability Reference Points.....	50
Figure 4.2 Tradeoffs between the Different Distributed Computing Technologies.....	54
Figure 4.3 The Architecture of Parlay 1.2 API.....	57
Figure 4.4 The JAIN Layered Approaches .....	58
Figure 4.5 The P1520 Reference Model: Open Programming Interfaces for Networks .....	60
Figure 5.1 Generation for H.245 Signaling Interfaces (ASN.1 to IDL) .....	64
Figure 5.2 Example Diagram for Development of CORBA Applications .....	71
Figure 5.3 Naming Services for Object Access (Call Registration Example).....	74
Figure 5.4 Trading Services for Object Access .....	75
Figure 5.5 Factory Methods and Interfaces in the Event Service API .....	77
Figure 5.6 oadj Communication Sequence Model .....	79
Figure 5.7 Visibroker Thread-per-session Model and Thread Pool Model.....	81
Figure 5.8 Visibroker Connection Management.....	83
Figure 5.9 Example of Signaling Diagram for H.323 Control Protocols .....	85
Figure 5.10 System Architecture of CORBA-based Interface Approach for H.323 Signaling .....	86
Figure 6.1 Points of Interceptors in Two-way CORBA Message Sequence .....	922



Figure 6.2 Performance Comparison for Byte Array Transfer Bandwidth.....	99
Figure 6.3 Performance Comparison for Int Array Transfer Bandwidth .....	100
Figure 6.4 Performance Comparison for Double Array Transfer Bandwidth.....	101
Figure 6.5 Dependency between Data Throughput and Message Size .....	104
Figure 6.6 Average Distribution of Time in a Sample CORBA Invocation.....	105

## List of Tables

Table 2.1 Some Universal-Class Tags and Corresponding Types .....	12
Table 2.2 GIOP Message Types, Originators and Values (GIOP1.2).....	27
Table 6.1 Remote Method Call Time for 3 Arguments.....	97
Table 6.2 Byte Array Transfer Bandwidth.....	98
Table 6.3 Int Array Transfer Bandwidth.....	99
Table 6.4 Double Array Transfer Bandwidth.....	100
Table 6.5 Marshaling Test for Round Trip Times.....	103
Table 6.6 PER and CDR Marshaling Tests for H.245 TCS Message Sample .....	107

## List of Listings

Listing 2.1 The Basic Structure of A GIOP Message.....	28
Listing 5.1 ASN.1 Message Syntax Example for H.245 CESE TerminalCapabilitySet.....	67
Listing 5.2 IDL Data Type Example for H.245 CESE TerminalCapabilitySetType.....	68
Listing 5.3 IDL Example for H.245 CESE Signaling .....	69
Listing 5.4 IDL Example for CESE Interface with One-way Invocations.....	72
Listing 5.5 The State Object Class in Visibroker oadj.....	78
Listing 5.6 Get the Initial Reference to the URLNaming Service (Server Object).....	80
Listing 6.1 Sample Output for Tracking Two-way CORBA Message Sequence though Visibroker Interceptor.....	92
Listing 6.2 Remote Method Call Test IDL .....	97
Listing 6.3 Numerical Data Transfer Test IDL.....	98
Listing 6.4 Marshaling Test IDL Example.....	103

## List of Acronyms

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ATM	Asynchronous Transfer Mode
BER	Basic Encoding Rules
CCITT	Consultative Committee on International Telephony and Telegraph
CDMA	Code Division Multiple Access
CDR	Common Data Representation
CESE	Capability Exchange Signaling Entity
CMIP	Common Management Information Protocol
CMIS	Common Management Information Services
CMOT	CMIP over TCP/IP
CORBA	Common Object Request Broker Architecture
DAVIC	Digital Audio-Video Council
DCE	Distributed Computing Environment
DER	Distinguished Encoding Rules
DII	Dynamic Invocation Interface
DPA	Document Printing Application
DPE	Distributed Processing Environment
DSI	Dynamic Skeleton Interface
DSM-CC	Digital Storage Media – Command and Control
DTF	Domain Task Force
EDIFACT	Electronic Data Interchange for Fiance, Administration, Commerce, and Transport
ESIOP	Environment Specific Inter-ORB Protocol
GDMO	Guidelines for the Definition of Managed Objects
GIOP	General Inter-ORB Protocol
GSM	Global System for Mobile Communications
HTTP-NG	Hyper Text Transfer Protocol Next Generation
IDL	Interface Definition Language

IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IN	Intelligent Networks
INAP	Intelligent Network Application Part
IOR	Interoperable Object Reference
IP	Internet Protocol
IR	Implementation Repository
ISDN	Integrated Service Digital Networks
ISO	International Standards Organization
ISP	Internet Service Provider
ISUP	ISDN User Part
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
JAIN	Java APIs for Integrated Networks
JCAT	JAIN Coordination and Transaction
JCC	JAIN Call Control
JIDM	Joint Inter-Domain Management
JMF	Java Media Framework
JTAPI	Java Telephony API
LAN	Local Area Network
LCSE	Logical Channel Signaling Entity
MAP	Mobile Application for GSM & IS41
MC	Multipoint Controller
MCU	Multipoint Control Unit
MHS	Message Handling Systems
MP	Multipoint Processor
MSDSE	Master/Slave Determination Signaling Entity
NMF	Network Management Forum
OBV	Object by Value
OMA	Object Management Architecture
OMAP	Operations, Maintenance and Administration Part

OMG	Object Management Group
ORB	Object Request Broker
OSF	Open Software Foundation
OSI	Open System Interconnection
OSS	Operation Support System
PC	Personal Computer
PDU	Protocol Data Unit
PER	Packed Encoding Rules
PIN	Programming Interfaces for Networks
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RAS	Registration Admission and Status
RFI	Request for Information
RFP	Request for Proposals
RMI	Remote Method Invocation
RM-ODP	Reference Model of Open Distributed Processing
ROS	Remote Operations Service
RPC	Remote Procedure Call
RTP	Real-Time Transport Protocol
SCCP	Signaling Connection Control Part
SCP	Service Control Point
SDL	Specification and Definition Language
SE	Signaling Entity
SET	Secure Electronic Transaction
SII	Static Invocation Interface
SIOP	SCCP Inter-ORB Protocol
SIP	Session Initiation Protocol
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
SRM	Session and Resource Manager
SS7	Signaling System No.7

SSP	Service Switching Point
TC	Transaction Capability
TCAP	Transaction Capability Application Part
TCP	Transmission Control Protocol
TKPT	Transport PDU in Discrete Units
TINA	Telecommunications Information Networking Architecture
TLV	Type (Tag) Length Value
TMN	Telecommunications Management Network
UDP	User Datagram Protocol
UML	Unified Modeling Language
UNO	Universal Networked Objects
URL	Uniform Resource Locator
U-N	User-to-Network
U-U	User-to-User
WAP	Wireless Application Protocol
XDR	eXternal Data Representation
XML	eXtensible Markup Language

## Chapter 1

### Introduction

---

Enabled by technological advances in packet switching, signal processing and explosive growth of the Internet economy, Internet Protocol (IP) Telephony is becoming a very successful alternative to the traditional circuit-switched technology. On the other hand, the Public Switched Telephony Network (PSTN) deployed with Signaling System No.7 (SS7) networks has made impressive achievements in terms of coverage, reliability and being feature-rich. Matching the SS7 features with a fully IP-based network is a major engineering challenge that might take a long period of time [Mitra1999a]. SS7 was originally designed for a closed community of telephone companies, although deregulation has changed the operational environment and created opportunities for insider attacks against the system. In addition, the Internet will provide more open interfaces to encompass a far greater range of rapid time-to-market services than the traditional voice network could offer. Various standards bodies and consortia are developing signaling solutions in IP networks to support telephony services, the two principal contenders being International Telecommunication Union – Telecommunication Standardization Sector (ITU-T) H.323-series of Recommendations [ITU1996a] and the Internet Engineering Task Force (IETF) Session Initiation Protocol (SIP) [Schulzrinne1998]. However, at this stage of the game, neither of them has gained overwhelming acceptance in terms of deployment. This situation leaves us the opportunity to explore new approaches.

## **1.1 Thesis Motivation**

### **1.1.1 Problem Statement**

The problem that our research addresses can be illustrated from following three viewpoints, i.e., the service provider, the network vendor and operator, and the software program developer.

In order to build and retain a strong growing customer base, Internet Service Providers (ISP) have to meet, if not exceed, the customer expectations set by today's traditional voice services. Acceptance of IP telephony will depend on the quality and efficiency with which service providers offer, deliver, and manage IP services. Most likely, future services over the Internet will encompass a range far greater than voice-based telephony services.

In today's telecommunication systems, most current approaches for signaling are based on the exchange of messages between particular hardware equipment, most likely from different vendors. To overcome interoperability requirements and get the systems to work together often cost a fortune for network vendors. Due to its dramatically enhanced usage, the current trend in the Internet economy is to move towards service outsourcing to third parties for fast product delivery, leaving a significantly diminished role for central network operators. Furthermore, because the richness of functional capabilities has moved to end user terminals like the Personal Computers (PCs), the service intelligence is not likely to be concentrated within the network. Rather, open interfaces to various network capabilities are essential if intelligence is to stay in the network.

The ITU, IETF and regional standards on signaling and transmission have made such interworking possible. However, these standards show that a tremendous amount of effort

has been placed in describing messages, including syntax and semantic definitions, encoding/decoding rules, procedures for message generation and reception, and how to react to abnormal situations. The software development process based on such standards separates the phases of analysis and design in traditional telecommunication products. It defines the message-based external interfaces to other systems in the analysis and specification phase, while the software design phase comes later and has to focus on the proper generation and reaction to those predefined messages. The software developed from this methodology comes from vendor-proprietary requirements and software design documents. Such an approach naturally leads to monolithic software or vertically integrated software by pieces from the same vendor. There are no open or standardized programming interfaces, because the base standards do not identify or require any, and it is not in the interest of the vendors to expose their internal interfaces to third party software development. Therefore, it is not reusable by software developers, who usually develop the distributed systems shielded from knowing details of the infrastructure, i.e., the exchanged messages, communication protocol, operating system, or hardware [Mitra1999a].

### **1.1.2 Industrial Concerns and Trends**

While the above mentioned message-centric approach may have worked in the past, the following concerns and trends from the industry make it unsuitable as the way to develop services over the Internet:

?? Leveraging intelligence between endpoints and network services: A traditional phone can only generate a small set of signaling events and tones. It cannot receive or process signaling of any sophistication. Signaling is received in the same voice channel as the



phone call, and processed by the human using the phone. These phones are considered “dumb” devices, and service intelligence is kept within the network. In contrast, IP phones can receive and process signaling message directly. Signals are sent as a separate set of IP packets. An endpoint’s ability to receive and act on signaling is the fundamental property that makes it intelligent. This enables the service functionality’s complete separation from voice bit transport. However, if the intelligence totally moves to the endpoint, the application may become costly and complex. The leverage of intelligence between the endpoints and network services may depend on well-accepted distributed infrastructure.

?? Separating services from networks: The message-based approach ties the services too closely to the underlying network, because the messages have been defined for a particular network by a particular standards body with a specific charter. For example, the H.323-based services are currently available only on the Local Area Network (LAN), while the Intelligent Network (IN) services are available only from the operators that have a SS7 network. However, such standards do not define a generally distributed mechanism for the service software independent of the network.

?? Offering feature transparency across networks: Consumers may become frustrated when services to which they have grown accustomed are not available due to technological constraints. For example, when you browse the web site for particular goods, you may expect to talk with the sales person before you make the order. So, instead of picking up your phone set, you may wish to click the button to place the call. For wireless subscribers, they may wish to have the same features that they have on their wired phones, as well as the features offered through their PC, like email or web browsing.

## **1.2 Thesis Objective**

The use of software technologies, in particular distributed object technologies like CORBA, is one way to address the above concerns. It shows that the design provides the infrastructure and may become independent of both network and access. The objective of this thesis is to provide credible evidence to show a vision of signaling for future IP networks that emphasizes an interface-centric approach based on CORBA rather than the current heterogeneous message-centric approaches [Mitra1999b].

This research seeks to “reverse engineer” the functions embedded in the H.323 series of protocols into Object Management Group (OMG) IDL interfaces. Such an effort would enable a distributed implementation of the logical H.323 architecture, which is based on terminals, gatekeepers and gateways, using CORBA as the signaling mechanism. This would have the advantage not only of all the distribution transparencies and the programming language/platform independence inherent in such an approach, but also the simplicity that comes from using a single messaging protocol, IIOP.

The performance of such an implementation demonstrates valuable insights on the suitability of using CORBA-based signaling for enterprise applications. In particular, it will determine the impact from message size and type complexity of CORBA requests as compared to text-based messages or other binary formats such as Packed Encoding Rules (PER) encoded H.323 messages. The result has to show whether or not it has a significant impact on IP telephony call set-up delays.

## **1.3 Thesis Contribution**

This thesis identifies the problems of signaling for current heterogeneous networks and

explores an interface-centric approach for targeting these problems. The thesis investigates the use of CORBA in the area of signaling for IP telephony, where the industry has not yet settled definitely on one approach. The primary focus of this software-centric approach is on defining a control infrastructure based on a distributed computing architecture where common capabilities, e.g., access control, usage recording, service logic and data, network events, etc., are accessible through language/platform-neutral interfaces while communicating through a common message set. The thesis makes a number of contributions as follows:

- ?? Reviewed the current application protocol development process and concepts, like ASN.1, encoding rules; reviewed the distributed architecture and key concepts of CORBA, as well as various CORBA-based internetworking activities with service and management networks in the telecommunication domain.
- ?? Compared ITU-T H.245 protocol specification with the OMG standard on “Control of Audio-Visual Streams”, and the standard of Digital Storage Media – Command and Control (DSM-CC) from Digital Audio Visual Council (DAVIC).
- ?? Explored the CORBA-based interface centric approach.
- ?? Fully converted ASN.1 defined H.245 messages to CORBA IDL using ASN-to-IDL compiler based on the translation specification from Joint Inter-Domain Management (JIDM), defined the IDL interfaces for signaling entities.
- ?? Partially implemented the selected procedures in H.245 to achieve its basic functionality, followed with the integration of H.225 Registration Admission and Status (RAS)/Q.931 signaling and Real-time Transport Protocol (RTP) media transmission procedures. The

H.323/H.225 part is undertaken in parallel by Christian Gosselin from UQAM. The project is well integrated through our common efforts.

?? Determined the factors based on the latency test for CORBA-based H.245 messaging performance.

## **1.4 Thesis Organization**

The structure of the thesis is as follows. In chapter 2, background information is given for the ASN.1-based protocols, CORBA, and the recent efforts for CORBA internetworking in the telecommunication domain. A knowledgeable reader, who has experience in ASN.1, encoding rules or CORBA could skip part of this chapter. In chapter 3, the H.323/H245 standards are briefly introduced. This is followed by a comparison of other two specifications for multimedia streaming control. In chapter 4, the interface-centric approach is illustrated with the emphasis on the design requirements and technique selections as well as industrial activities towards open interfaces. In chapter 5, the implementation is presented with the overall design and techniques used in our implementation. Chapter 6 gives an overview on CORBA performance issues, such as GIOP/IIOP implication and limitation on performance, CORBA performance monitoring technique and benchmarks. Experimental environment and performance results are illustrated, along with the discussion of performance concerns for selecting CORBA in the design. Chapter 7 provides a summary of the thesis's key messages and a number of conclusions addressing our project's objectives, as well as future work of the research.

## **Chapter 2**

### **Background (ASN.1-based Protocols and CORBA)**

---

Background information is provided at this chapter in three parts, and knowledgeable readers might skip the sections with which they are familiar. First of all, we introduce the concepts and development process for ASN.1-based application protocols. Second, we focus on one of the most dominated distribute computing technology, i.e., CORBA, explaining the key concepts, followed by several related OMG specifications. In the third part, we investigate several on-going efforts for integrating CORBA to the telecommunication service and management architectures, such as Intelligent Networks, Telecommunications Management Network (TMN), Telecommunications Information Networking Architecture (TINA).

#### **2.1 ASN.1-based Protocols**

In today's global communications infrastructure, computer systems have collaborated to perform a wider range of activity than ever before. Applications require increasingly complex exchanges of information between computer systems and between appliances with embedded computer chips. There is a requirement for the detailed specification of the exchanges the computers are to perform, and for the implementation of software to support those exchanges. For communication to be possible between applications and devices produced by different vendors, standards are needed for these application protocols. In a number of industrial sectors, but particularly in the telecommunications sector, in multimedia exchanges and in security-related exchanges, ASN.1 is the dominant means of specifying

application protocols.

### **2.1.1 ASN.1**

Abstract Syntax Notation One is an international standard, which aims at specifying the data used in application protocols. It provides a high level description of messages that frees protocol designers from having to focus on the bits and bytes layout of messages. As a computing language that is both powerful and complex, ASN.1 was designed for modeling efficiently the communication between heterogeneous systems. For the time being, ASN.1 has been adopted for use by a wide range of applications, such as network management, secure email, cellular telephony, air traffic control, and voice and video over the Internet.

In 1982, four years after the appearance of Open System Interconnection (OSI), many people who worked on the development of standards on Application Layer had encountered the same problem: the data structures had become too complex to allow procedures for encoding and decoding in bits or bytes. In 1984, ASN.1 was originally proposed as a notation and an algorithm that could define the format of encoding bits for the email Message Handling Systems (MHS) protocols by the Consultative Committee on International Telephony and Telegraph (CCITT, X.208) and joint work with the International Standards Organization (ISO, ISO 8824). This recently evolved to X.680. Though the standards are very thorough and precise in their definitions, they are not very easy to read and practice for application protocol designers [Dubuisson2000].

In a given programming language like C, the data structure to be transferred is represented in “Concrete Syntax”, which respects the lexical and grammatical rules of a language. In contrast, the concept of “Abstract Syntax” describes the generic structure of data

independent of any encoding technique used to represent the data. The syntax allows data types to be defined and values of those types to be specified. ASN.1 is one kind of the abstract syntax, and is being used to define the following types of data:

1. the abstract syntaxes of application data
2. the structure of application and presentation protocol data unit (PDU)
3. the management information base for both Simple Network Management Protocol (SNMP, RFC1157) and OSI systems management, like the Common Information Services and Protocols for the Internet, Common Management Information Protocol (CMIP, RFC1189) and CMIP over TCP/IP (CMOT)

There are other abstract notations that can be compared with ASN.1 and even compete with it in some respects. Some examples are OMG IDL for CORBA, Sun Microsystems' eXternal Data Representation (XDR, RFC1832), Electronic Data Interchange for Finance, Administration, Commerce, and Transport (EDIFACT, ISO9735).

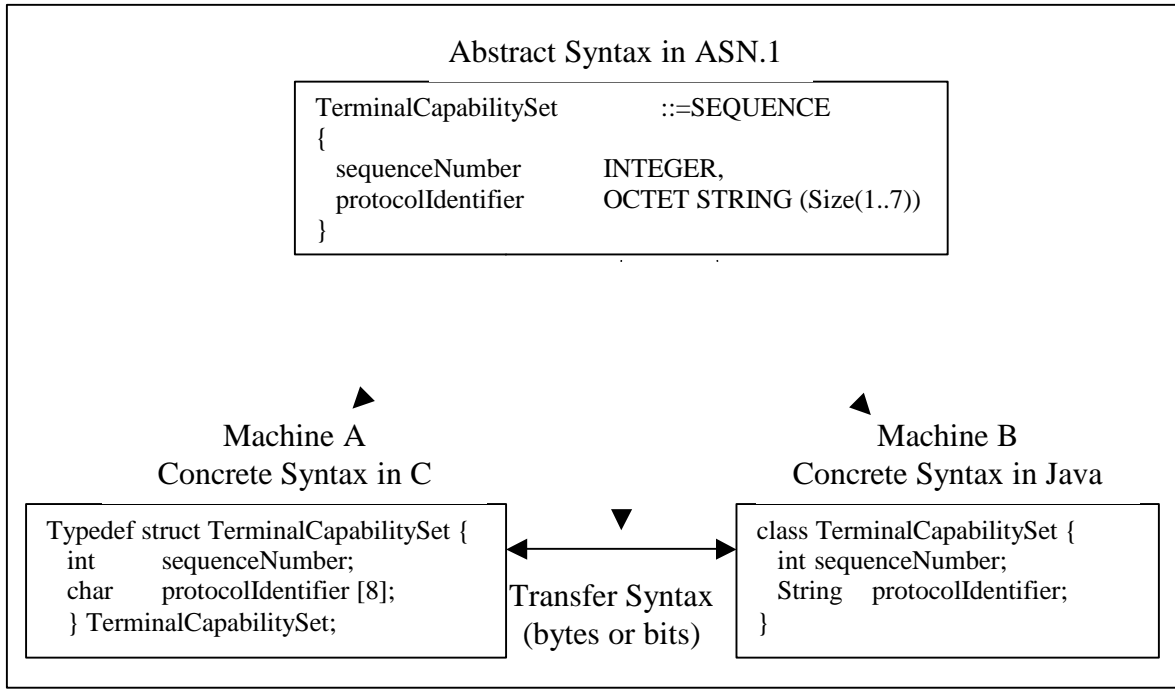
The third concept of "Transfer Syntax" defines the representation of data to be exchanged between data transfer components. The translation from abstract syntax to the transfer syntax is accomplished by means of encoding rules that specify the representation of each data value of each data type.

This approach to exchange application data solves two problems that relate to data representation in a distributed, heterogeneous environment.

1. a common representation for the exchange of data between different systems
2. internal to a system, an application uses some particular representation of data. The abstract/transfer syntax scheme resolves differences in representation between co-

operating application entities.

Figure 2.1 gives an example to show the relationship among the three kinds of syntax.



**Figure 2.1 An example of Syntax Relationship (Abstract, Concrete and Transfer)**

From a single ASN.1 data description, we can derive as many concrete syntaxes in as many programming languages, and as many procedures implementing the transfer syntax in the encoders/decoders.

In ASN.1, a type is a set of values. For some types, there are a finite number of values, and for other types there are an infinite number. ASN.1 has four kinds of type: simple type, structured type, tagged type and other type. Every ASN.1 type other than “Choice” and “Any” has a tag, which consists of a class and a non-negative tag number. ASN.1 types are abstractly the same if their tag numbers are the same. There are four classes of tag:

1. Universal: for types whose meaning is the same in all applications as defined in



X.208. Table 2.1 lists some ASN.1 types and their universal-class tags.

**Table 2.1 Some Universal-Class Tags and Corresponding Types**

Tag Number (decimal)	Type
UNIVERSAL 1	BOOLEAN
UNIVERSAL 2	INTEGER
UNIVERSAL 3	BIT STRING
UNIVERSAL 4	OCTET STRING
UNIVERSAL 5	NULL
UNIVERSAL 6	OBJECT IDENTIFIER
UNIVERSAL 16	SEQUENCE and SEQUENCE OF
UNIVERSAL 17	SET and SET OF
UNIVERSAL 19	PrintableString
UNIVERSAL 22	IA5String
UNIVERSAL 27	GeneralString
UNIVERSAL 31 ...	Reserved for future use

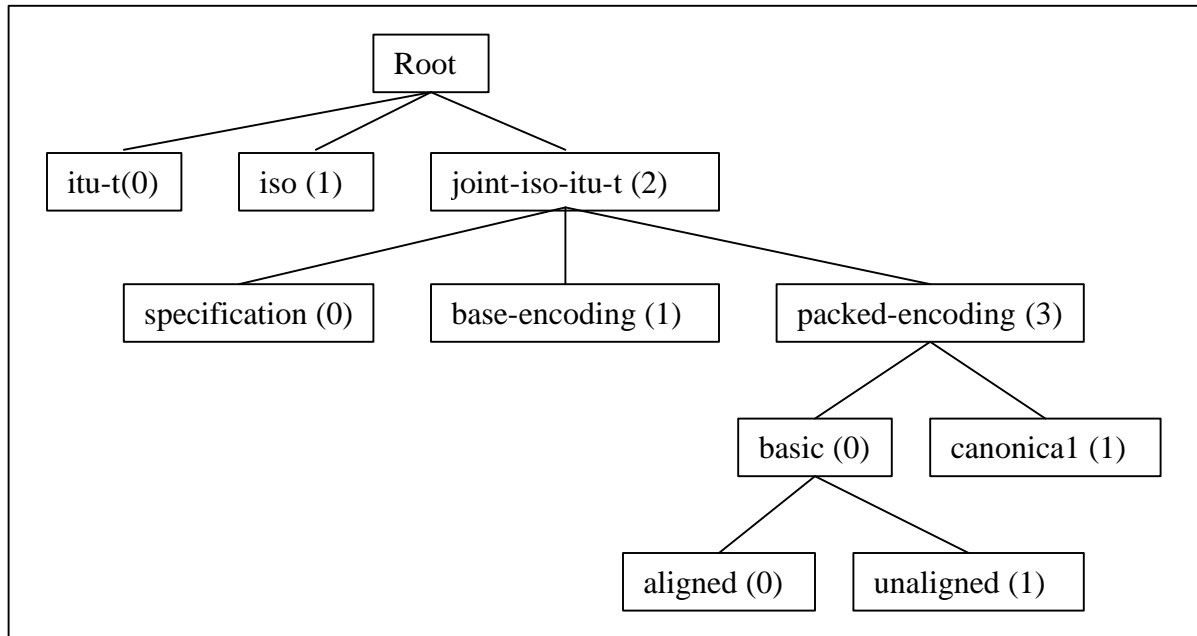
2. Application: for types whose meaning is specific to an application, such as X.500 directory services. Types in two different applications may have the same application-specific tag and different meaning.
3. Private: for types whose meaning is specific to a given enterprise.
4. Context-specific: for types whose meaning is specific to a given structured type.

Other features of ASN.1, such as information object classes and information objects, modules and specifications, can be found in various background material [Kaliski1993].

### 2.1.2 Encoding Rules

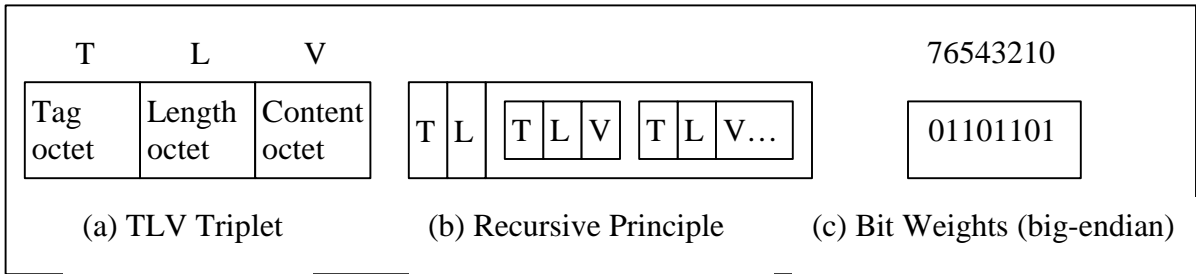
Closely associated with ASN.1 are sets of standard encoding rules that describe the bits

and bytes layout of messages as they are in transit between communicating application programs. Like ASN.1, the encoding rules are also not tied to any particular computer architecture, operating system, language or application program structure, and are used in a range of programming languages, including C, C++, or Java. Some of these encoding rules are registered with the ISO object registration tree such as the one shown in Figure 2.2.



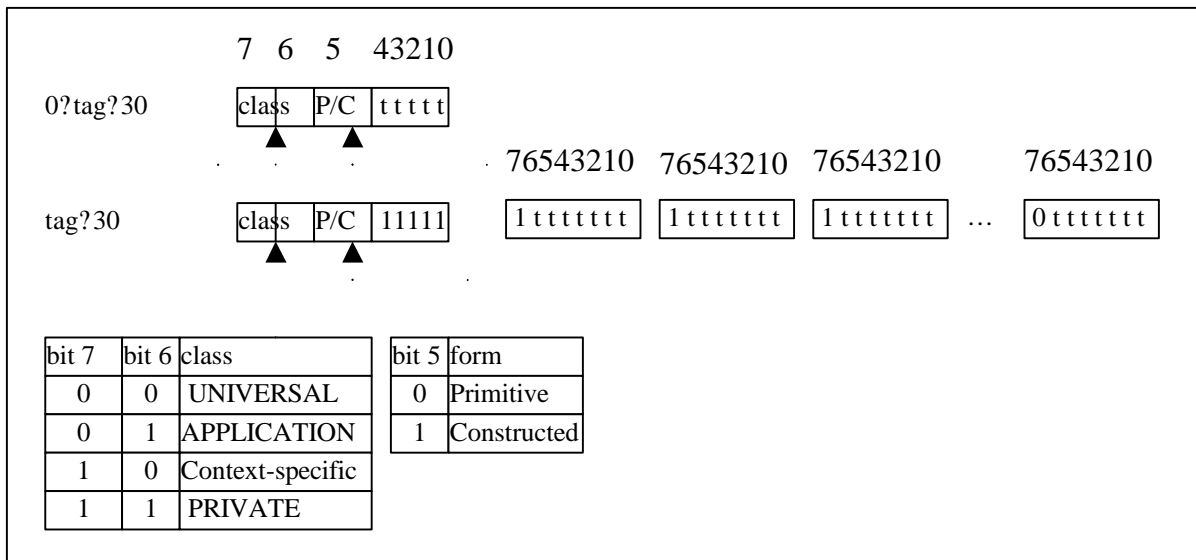
**Figure 2.2 BER and PER in ISO Object Registration Tree**

The Basic Encoding Rules (BER) are the original encoding rules of ASN.1 since they were part of X.409 standard in 1984. The BER transfer syntax always has the format of a triplet “TLV”, i.e., Type (or Tag), Length, Value as shown in Figure 2.3 (a). All the fields of T, L, and V are series of octets. The value V can, itself, be a triplet TLV if it is constructed. The most complex of the ASN.1 values is no more than a stack of less and less complex values as shown in Figure 2.3 (b). The transfer syntax is octet-based and self-delimited since the field L provides a means of determining the length of each TLV triplet. The BER follows big-endian principle, the high-order bit is at the left-hand side as shown in Figure 2.3 (c).



**Figure 2.3 BER Transfer Syntax**

To get a detailed understanding of how octets are constructed in BER, Figure 2.4 shows the constructed format of the tag octets. The tag octets correspond to the encoding of the value's type. If the tag number is smaller than or equal to 30, the tag class and number are encoded on a single octet. If the tag number is greater than 30, the number consists of the concatenation of the bits from no.6 down to no.0 for all octets but the first one, whose five lower-order bits equal 11111.



**Figure 2.4 Example BER Format of the Tag Octets**

The format of length also follows the specific rules to construct the octets, which can be

in either definite or indefinite form. The value octets are constructed based on the information given in tag and length octets.

A criticism expressed towards the BER is regarding their cost in terms of size, with 50% extra cost on average compared to the actual data to encode. This drawback led to the development of the much more efficient PER, but they are not self-defining and less flexible. PER follow the rule: “obtain the most compact encoding using encoding rules as simple as possible”, and are particularly appropriate for protocols that need to transfer data at a high rate in domains like telephony over the Internet, video conferencing and multimedia in general.

Instead of using a systematic recursive format in triplets TLV like the BER, the PER format could be interpreted as '[P][L][V]' (optional preamble, optional length, optional value) where the fields P, L and V are no longer series of octets but series of bits. PER can provide a more compact representation of the values that are actually sent in an instance of communications. This approach is popularly used when both the transmitter and the receiver expect data to adhere to a known structure.

The PER break down into two categories: basic and canonical, and either can be of the aligned or unaligned variant. In aligned variant, padding 0 bits are inserted when needed to restore the octet alignment. The unaligned variant is far more compact but requires much more processing time for encoding and decoding. H.245 is implemented using PER. Since both sides of a message exchange know that the syntax of the messages will conform to the H.245 specification, it is not necessary to encode the specification into the message. For decoding simplicity, the aligned variant of PER is used. This forces fields that require eight or more bits to be aligned on octet boundaries and to consume an integral number of octets.

Tags are not encoded in PER. A length field L is encoded only if the size has not been fixed by a SIZE subtype constraint in the ASN.1 specification or if the data size is important. The encoding of values of type SEQUENCE or SET is preceded by a bit-map, which indicates the presence or absence of optional components. Similarly, an index indicates the alternative retained in a CHOICE type before encoding the value associated with this alternative.

Figure 2.5 gives an example for the PER encoded H.245 Terminal Capability Set request with the IP/TCP/TPKT (Transport PDU in Discrete Units) headers in hexadecimal strings. IP header and TCP header are formatted as *italic* and **bold** respectively. The next 4 octets are a TPKT header that is underlined, followed with the H.245 messages (74 octets).

<i>4500</i>	<i>0081</i>	<i>e14d</i>	<i>0000</i>	<i>4006</i>	<i>05b2</i>	<i>c0a8</i>	<i>8915</i>
<i>c0a8</i>	<i>8911</i>	<b>3aa1</b>	<b>3a9c</b>	<b>c3ba</b>	<b>2276</b>	<b>028b</b>	<b>29e9</b>
<b>5018</b>	<b>111c</b>	<b>20d7</b>	<b>0000</b>	<u>0300</u>	<u>004e</u>	0270	0106
0008	8175	0002	800d	<u>0000</u>	<u>3c00</u>	0100	0001
0000	0100	0003	8000	0020	c03b	8000	0108
a817	6f40	0002	2200	0740	0003	09f8	0def
404a	3700	5040	0100	0080	0001	0100	0000
0201	0001	0003					

**Figure 2.5 PER Encoded H.245 Terminal Capability Set Request with Headers**

### 2.1.3 Widely-used Communication Protocols

Although ASN.1 seems to be obscure, it is actually being wisely used. Every time we place a 1-800-number call, ASN.1 defined messages are exchanged between the switching machine and the network database to route the call to the correct common carrier and local phone number to which the 1-800-number maps. Whenever routing data is changed within SS7, the central nervous system of the telephone network, Operations, Maintenance and Administration Part (OMAP) messages that are described in ASN.1 are utilized in carrying out the change.

Every call placed on a cellular telephone in North America, Europe, and Japan results in Transaction Capability Application Part (TCAP) protocol messages. These messages, described using ASN.1 and encoded using one of its predefined encoding rules, go flying through the air to establish the call. When we walk along talking on the cellular telephone and go from one cell to another, ASN.1 helps transfer control of the call between cells.

Companies such as Federal Express use ASN.1 and its encoding rules heavily to track their packages. ASN.1 is also used by the electric and gas utilities to control the latest generation of substations and transformers. ASN.1 is the choice of companies such as Hewlett Packard, IBM, Sun and Xerox for defining the Document Printing Application (DPA) standard interface for printer job management. To list a few, ASN.1 specified communication protocols could be categorized as follows [ASNResource2000]:

- ?? High-level layers of the OSI model, the Application and Presentation layer protocols
- ?? X.400 electronic mail system
- ?? X.500 directory
- ?? Multimedia environment, such as Multimedia and Hypermedia information coding Expert Group (MPEG) and ITU-T H.323, H.225, H.245 recommendations
- ?? The Internet, like SNMP, CMIP, etc.
- ?? Electronic Data Interchange (EDI) protocols
- ?? Business and electronic transactions, like Secure Electronic Transaction (SET), etc.

#### **2.1.4 Protocol Development Process**

It is the ASN.1 compiler that carries out the generation of language specific concrete

syntax. The compiler should be implemented with some encoding rules, which describe the links between the abstract syntax and transfer syntax. A working example for protocol development process using ASN.1/C compiler is shown as Figure 2.6. First of all, all files that constitute the ASN.1 specification are collected, including those referenced in the IMPORTS clauses. All these files are then given to the ASN.1/C compiler as input files. The major functions of the compiler cover lexical analysis, parsing, semantic analysis and target language code generation. The generated codes normally have two parts:

- ?? A file with the concrete syntax, which is the translation of the data types defined in the ASN.1 specification into the target language (for example the .h file in C language);
- ?? One or several files including one encoding procedure and decoding procedure, like BER, PER, for each type of the ASN.1 specification (for example the .c files in C language).

Both commercial and public ASN.1 compilers are available for C/C++/Java following various encoding rules, like BER, PER, Distinguished Encoding Rules (DER) [ASNHome1997]. Without further effort, the designers of a communication application have data transfer procedures at their disposal. What remains to be done is to program the complete local behavior of the protocol, which is usually described in Specification and Definition Language (SDL). However, the task of encoding complex data structures for network transmission is still more expensive in terms of processor time and memory usage than most other components of the protocol stack. This is so even after the optimization for encoding rules, and this may lead to the development of non-standard data representations tuned for a particular application, which is not portable across different environments [Sample1993].

The files generated by the ASN.1 compiler and those specific to the communicating

application are then given to a compiler of the computing language used for programming the communication application (like a C compiler). This produces an executable for the machine architecture using libraries provided with the ASN.1 compiler, which contain the encoding and decoding procedures of all ASN.1 primitive types. The executable can send and receive a binary stream on a telephone line or a computer network.

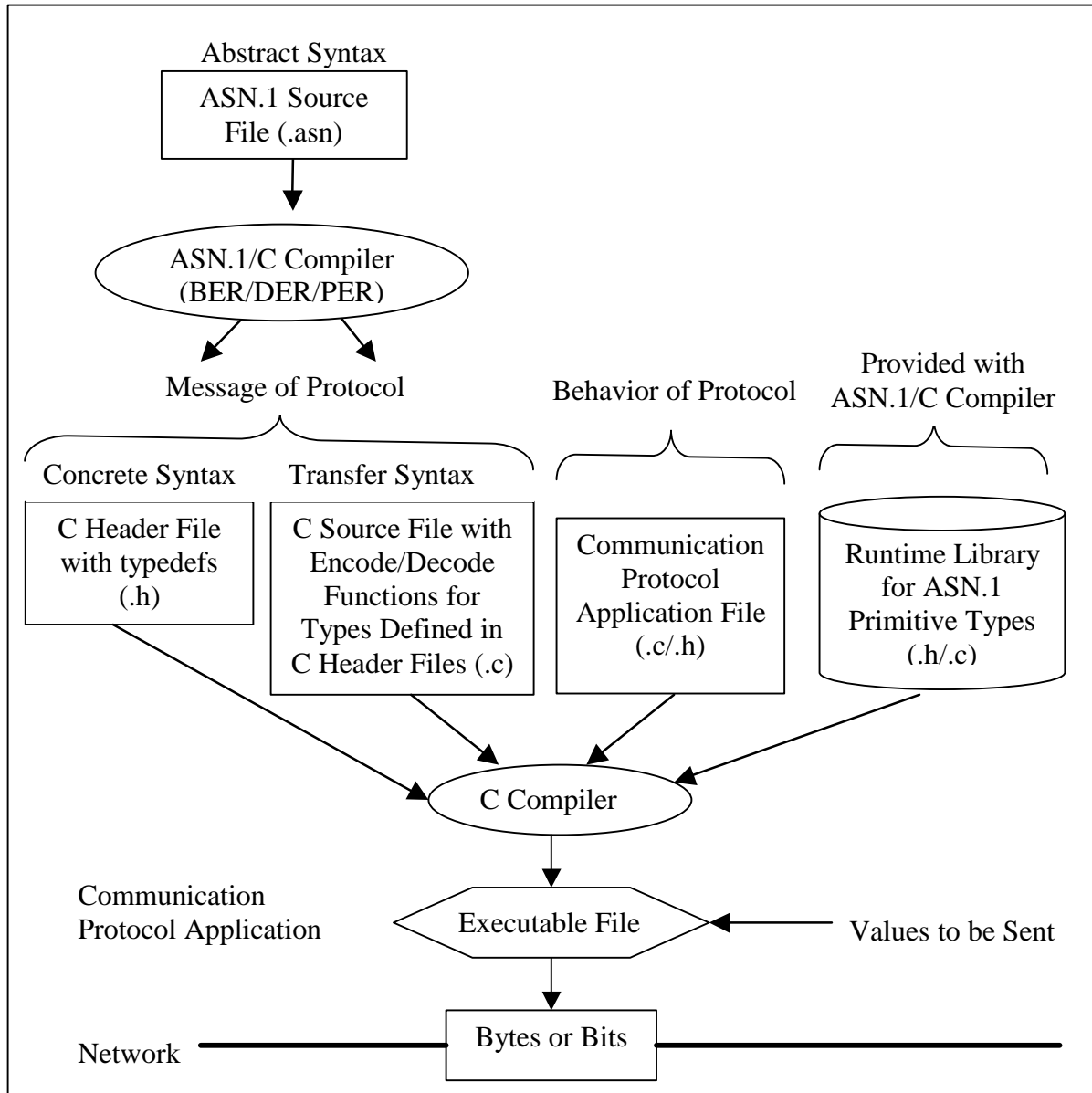


Figure 2.6 Example of Protocol Development Process with ASN.1/C Compiler



The above protocol development process indicates that one of the major concerns in service or feature upgrade is that the protocol messages may be expanded over the time. This may happen either through new messages, or new parameters in existing messages, or new parameter values for existing parameters, or a combination of all three. While the functions embodied by these messages are separate, the actual binary format of the messages does not permit an easy separation of the content. Thus, a change in any of these functions requires the software upgrade of intermediate switches in communication systems.

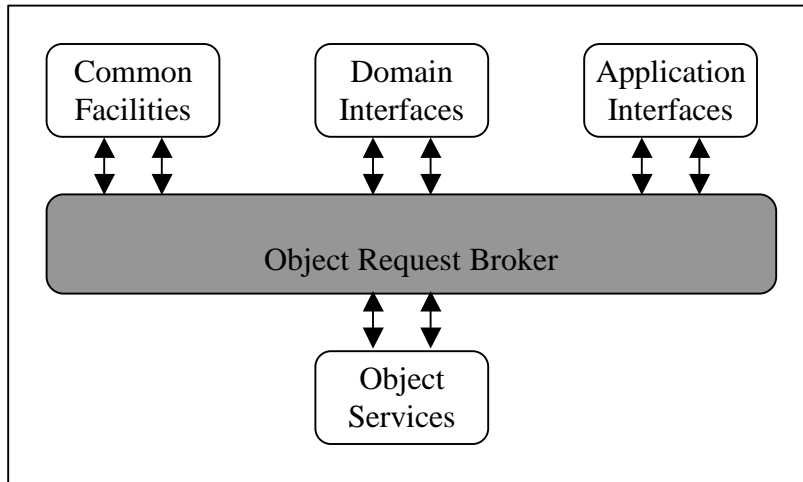
## **2.2 Common Object Request Broker Architecture**

With advances in communication technologies and development of powerful network stations, computer systems are rapidly changing from a centralized model to a distributed environment. In order to support distributed application development and to provide connectivity and interoperability among heterogeneous computing systems, a number of distributed environments, called “middleware”, have been developed. Examples are the Distributed Computing Environment (DCE) offered by the Open Software Foundation (OSF), CORBA by the OMG. The objectives of middleware environments are to provide the services that distributed applications need and to facilitate the development of distributed applications, which is independent of underlying platforms. To accomplish this, middleware provides runtime services supporting various forms of transparency, such as distribution and location transparency. In the following sections, we will explain the key concepts of CORBA.

### **2.2.1 Object Management Architecture (OMA)**

After the OMG was formed in 1989, it had defined Object Management Architecture,

which provides a common language for applications and enables the interoperability at the application level by defining standard services and interfaces. The core element of OMA is CORBA Object Request Broker (ORB), which will be addressed in following sections. Besides that, Figure 2.7 shows the OMA Reference Model (1992) with the following components.



**Figure 2.7 Object Management Architecture Reference Model**

?? Object Services: These components provides a standardized functionality, which is defined in the form of object interfaces, e.g. for class and instance management, storage, integrity, security, query, and versioning.

?? Common Facilities: These are horizontal facilities that vendors may use to provide a set of generic applications that can be configured to the specific requirements of a particular configuration, such as email. In the current version of OMA Reference Model, the Common Facilities are suppressed.

?? Domain Interfaces: These are the standard interfaces that are defined by particular industrial groups towards specific application domains, such as telecommunications and e-commerce.

?? Application Interfaces: These are not standardized object interfaces, which are specifically developed for an application.

### 2.2.2 The CORBA Architecture Reference Model

CORBA ORBs are infrastructure components that allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols, and hardware. Figure 2.8 illustrates the key components in the CORBA reference model that collaborate to provide this degree of portability, interoperability, and transparency [Schmidt2000].

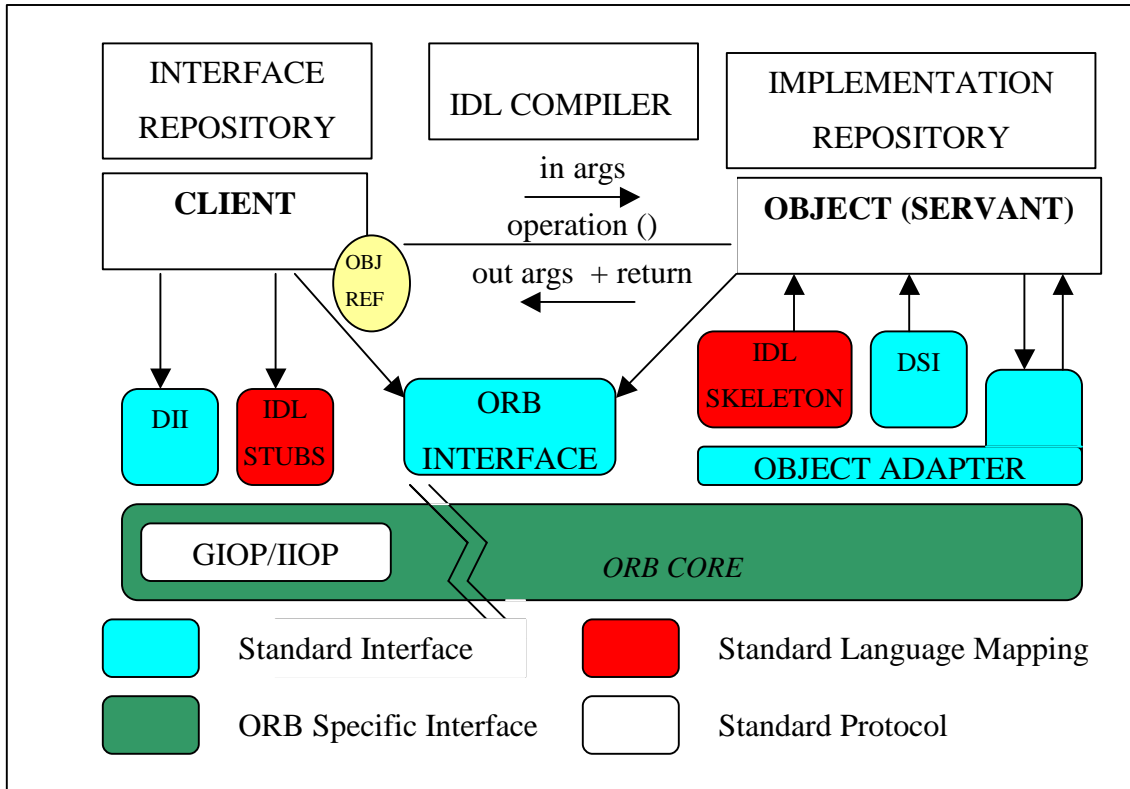


Figure 2.8 CORBA 2.x Architecture Reference Model

In this model, a client obtains references to objects and invokes operations on them to perform application tasks. Ideally, a client can access a remote object just like a local object,

i.e., object → operation (args). An object is an instance of an OMG IDL interface. Each object is identified by an object reference, which associates one or more paths through which a client can access an object on a server. When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response to the client. An ORB Core is implemented as a run-time library, which includes ORB communication protocol, like IIOP, linked into client and server applications. IDL stubs and skeletons serve as “glue” between the client, servants and ORBs. The stubs implement the proxy pattern and provide a strongly typed static invocation interface (SII) that marshals application parameters into a Common Data Representation (CDR) format. Conversely, skeletons implement the Adapter pattern and demarshal the data from CDR back into typed parameters.

Major components in the CORBA reference model are outlined below:

- ?? Client: A client is a computational context that makes requests on an object through one of its references.
- ?? Server: A server is a computational context in which the implementation of an object exists.
- ?? Object: A CORBA object in an abstract sense is a programming entity with an identity, an interface, and an implementation. From a client’s perspective, the object’s identity is encapsulated in the object’s reference. From a server’s view, it is explicitly managed by object implementations through the object adapter interfaces.
- ?? Servant: A servant is a programming language object or entity that implements requests on one or more objects. A servant generally exists within the context of the server

process. Request made on an object's references are mediated by the ORB and transformed into invocations on a particular servant.

?? Interoperable Object Reference (IOR): An IOR of an object implementation is the unique identifier of an object implementation, providing all the information necessary for another CORBA process to locate and communicate with it.

?? IDL Compiler: An IDL compiler automatically transforms OMG IDL definitions into an application programming language.

?? Object Adapter: An object adapter associates a servant with objects, demultiplexes incoming requests to the servant and collaborates with the IDL skeleton to dispatch the appropriate operation up-call on that servant. It is the essential component for portability among different object systems.

?? Interface Repository: An interface repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was unknown when the program was compiled, and be able to determine what operations are valid on the object and make invocations on it.

?? Implementation Repository (IR): An implementation repository provides a common location to store information associated with servers, such as administrative control, resource allocation and activation modes.

### **2.2.3 General Inter-ORB Protocol (GIOP)/IIOP**

As we mentioned earlier, the GIOP and IIOP support protocol-level ORB interoperability in a general, low cost and simple manner. With only seven message formats

in version 1.0, the GIOP messages are exchanged between agents to facilitate object requests, locate object implementation, and manage communication channels. GIOP semantics require no format or binding negotiations. These factors allow clients to send requests to objects immediately upon opening a connection. As a concrete realization of GIOP, IIOP describes how agents open TCP/IP connections and use them to transfer GIOP messages [OMG2000].

GIOP makes the following assumptions about the underlying transport that is used to carry messages. The list of assumptions matches the guarantees provided by TCP/IP, as well as other transport protocols, including Systems Network Architecture (SNA), Asynchronous Transfer Mode (ATM), Hyper Text Transfer Protocol Next Generation (HTTP-NG).

?? The transport is connection-oriented: A connection-oriented transport allows the originator of a message to open a connection by specifying the address of the receiver. After a connection is established, the transport returns a handle to the originator that identifies the connection. The originator sends a message via the connection without specifying the destination address with each message; instead, the destination address is implicit in the handle that is used to send each message.

?? Connections are full-duplex: The receiving end of a connection is notified when an originator requests a connection. The receiver can either accept or reject the connection. If the receiver accepts the connection, the transport returns a handle to the receiver. The receiver not only uses the handle to receive messages but can reply to the requests sent by the originator via the same single connection and does not need to know the address of the originator in order to send replies.

?? The transport is reliable: The transport guarantees that messages sent via a connection are delivered no more than once in the order in which they were sent. If a message is not

delivered, the transport returns an error indication to the sender.

?? The transport provides a byte-stream abstraction: The transport does not impose limits on the size of a message and does not require or preserve message boundaries. In other words, the receiver views a connection as a continuous byte stream. Neither receiver nor sender need be concerned about issues such as message fragmentation, duplication, retransmission, or alignment.

?? The transport indicates disorderly loss of a connection: If a network connection breaks down, both ends of the connection receive an error indication.

Based on the above transport assumptions, GIOP defines the following rules for transferring messages.

?? Asymmetric connection: GIOP defines client and server as two distinct roles with respect to connections. The client side of a connection originates the connection, and sends object requests. The server side accepts requests and sends replies. The server side of a connection may not send object requests. This restriction allows the GIOP specification to be much simpler and avoids certain race conditions.

?? Request multiplexing: Multiple clients within an ORB may share a connection to send requests to a particular ORB or server. Each request uniquely identifies its target object. Multiple independent requests for different objects, or single objects, may be sent on the same connection.

?? Connection management: GIOP defines messages for request cancellation and orderly connection shutdown. Therefore, the CORBA specification does not require any particular connection management strategy for ORBs.

GIOP basic message types are summarized in Table 2.2, which lists the message type names, whether the message is originated from client, server or both, and the value used to identify the message type in GIOP message headers. GIOP 1.0 supports seven different types of messages. GIOP 1.1 and 1.2 also support Fragment message type. Detailed descriptions for each message are defined in up-to-date CORBA specification [OMG2000]. Listing 2.1 shows the basic structure of a GIOP message in pseudo-IDL.

**Table 2.2 GIOP Message Types, Originators and Values (GIOP1.2)**

Message Type	Originator	Value in GIOP Header
Request	Client	0
Reply	Server	1
CancelRequest	Client	2
LocateRequest	Client	3
LocateReply	Server	4
CloseConnection	Server	5
MessageError	Both	6
Fragment	Both	7

```

Module GIOP {
    Struct Version {
        Octet major;
        Octet minor;
    };
    enum MsgType {
        Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError
    };
    struct MessageHeader {
        char magic [4];
        Version GIOP_version;
        octet flags;
        octet message_type;
        unsigned long message_size;
    };
}
    
```



```
};  
};
```

### Listing 2.1 The Basic Structure of A GIOP Message

#### 2.2.4 Common Data Representation

GIOP defines a Common Data Representation that determines the binary layout of IDL types for transmission. CDR is a transfer syntax. It maps data types defined in OMG IDL to a bicononical, low level representation for transfer between agents. CDR has the following main characteristics.

?? CDR supports both big-endian and little-endian representation: CDR-encoded data is tagged to indicate the byte ordering of the data. This means that both big-endian and little-endian machines can send data in their native format. If the sender and receiver use different byte ordering, the receiver is responsible for byte-swapping. This model, called receiver makes it right, has the same endianness, they can communicate using the native data representation of their respective machines. This is preferable to encodings such as XDR, which require big-endian encoding on the wire and therefore penalize communication if both sender and receiver use little-endian machines.

?? CDR aligns primitive types on natural boundaries: CDR aligns primitive data types on byte boundaries that are natural for most machine architectures. For example, short values are aligned on a 2-byte boundary, long values are aligned on a 4-byte boundary, and double values are aligned on an 8-byte boundary. Encoding data according to these alignments wastes some bandwidth because part of a CDR-encoded byte stream consists of padding bytes. However, despite the padding, CDR is more efficient than a more

compact encoding because, in many cases, data can be marshaled and demarshaled simply by pointing at a value that is stored in memory in its natural binary representation.

This approach avoids expensive data copying during marshaling.

?? CDR-encoded data is not self-identifying: CDR is a binary encoding that is not self-identifying. This means that CDR encoding requires an agreement between the sender and receiver about the types of data that are to be exchanged. This agreement is established by the IDL definitions that are used to define the interface between sender and receiver. The receiver has no way to prevent misinterpretation of data if the agreement is violated.

CDR encoding is a compromise that favors efficiency. Because CDR supports both little-endian and big-endian representations and aligns data on natural boundaries, marshaling is both simple and efficient. The downside of CDR is that certain type mismatches cannot be detected at run time in the case of using Dynamic Invocation Interface (DII) or Dynamic Skeleton Interface (DSI). Other encodings do not suffer from this problem. For example, as mentioned earlier, the Basic Encoding Rules (BER) used by ASN.1 use a Tag-Length-Value (TLV) encoding, which tags each primitive data item with both its type and its length.

### **2.2.5 Passing Object by Value (OBV)**

In CORBA, the client and server are generally executing in two different machines, and the invocation of a remote object is accomplished by passing object reference. When the object is passed by reference, if the receiver intends to access any data or operation within the object, it can do so using the reference passed to it. But every such access would end up in the wire traffic because the object is still within the sender's domain. This can be slow.

Furthermore, simply having a reference in the receiver's space does not guarantee the object still exists in the sender's domain. To address these requirements, the OMG has added an extension to the CORBA 2.4 specifications to enable the passing of objects by value [OMG2000].

This extension introduces a new IDL type "Value", which is used to pass state data over the wire. A value is best thought of as "Struct" with inheritance and methods. Value types differ from normal interfaces in that they contain properties to describe the state of value type, and contain implementation details beyond that of an interface. Value types are always local. They cannot be called remotely, which means only the data part of a value object is transferred, not the implementation. There are two kinds of value types.

?? Concrete value types: concrete value types contain state data. They extend the expressive power of IDL structs by allowing: single concrete value type derivation and multiple abstract value type derivation, arbitrary recursive value type definitions, null value semantics and sharing semantics, etc.

?? Abstract value types: abstract value types contain only methods and do not have state. They may not be instantiated. Abstract value types are a bundle of operation signatures with a purely local implementation.

OBV provides a chance to increase location transparency by minimizing remote access. In cases of OBV, when the receiving party instantiates a copy of an object, it implies that the receiver knows how to implement the object (instantiate it, initialize it, and provide implementations of the operations). More importantly, this also implies the receiver knows something about the semantics of the object and can utilize those semantics locally. The new instance created by the receiving side has a separate identity from the original object, and

once the parameter passing operation is complete, there is no relationship between the two instances. Obviously, this approach violates the fundamental CORBA concept of encapsulation, which normally hides an object's encapsulation from its clients. Meanwhile, there are several very complex edge efforts of the OBV specification, such as when interface references and value types are intermixed. Therefore, this may complicate the development work of ORB vendors, IDL designers and programmers. The design of our interface approach does not use OBV, though OBV is going to get more attention in developing CORBA applications in the future.

### **2.2.6 CORBA Services**

CORBA services are individual software components designed to promote a greater amount of software reuse. In defining the services, the OMG took an in-depth look at the software development process and tried to focus on common steps or pieces of functionality most programs need to implement. These services are building blocks from which the CORBA objects can inherit functionality or standalone components with which the objects interact. Each service has been defined and engineered with two main underlying concepts: 1) the service must be generic, meaning it should be domain-independent; 2) the service should do one specific task in a thorough manner. CORBA service specifications describe 16 common services [OMG1998]. Once all services are available, the development life cycle of a CORBA application will be substantially shortened. Here, we list a few that will be addressed later in chapter 5.

?? The naming service allows names relative to a name context to be bound to objects, and names to be resolved into object references, therefore locating objects in a network.

?? The trading service allows services to be offered, whereby they are registered with a broker object, and services to be located, whereby the broker object is queried about services with certain properties.

?? The event service supports the communication of objects using asynchronous message, i.e., messages that have not been directly requested.

### **2.2.7 Extensions in CORBA 3**

OMG technical task force is always making improvements on various CORBA specifications addressing the requirements from the industry. The coming version 3.0 of CORBA will have following extensions as announced by OMG [Siegel1999]:

?? Distributed components support

1. The CORBA component model specifies a framework for the development of plug-and-play CORBA objects. It encapsulates the creation, lifecycle, and events for a single object and allows clients to dynamically explore an object's capabilities, methods, and events.
2. The CORBA scripting language specification makes composition of CORBA components easier.

?? Java and Internet integration and legacy support

1. A Java to IDL mapping allows developers to implement applications completely in Java and to generate the IDL from Java classes. This enables other applications to access Java applications using Remote Method Invocation (RMI) over IIOP.
2. DCE/CORBA interworking specifications provide a road map for integrating DCE

applications into CORBA environments.

?? Quality of service specifications

1. Minimum CORBA addresses the need for CORBA-compliant systems that can be operate in embedded environments.
2. Real-time CORBA introduces real-time ORBs in the CORBA specification that give developers a more direct control over resource allocation.

### **2.3 CORBA in the Telecommunications Domain**

Distributed object technologies such as CORBA are important to the telecommunications domain, especially when they are applied for the management of telecommunications networks and for the delivery of telecommunications services. Traditionally, telecommunications systems consist of dedicated switching systems with embedded intelligence to provide telecommunications services. Recently, technologies such as IN, TMN and TINA aim at moving the intelligence out of the switching systems into generic computer systems. Adopting CORBA for large-scale application development provides major benefits like increased software reuse, improved system scalability, ease of distribution, implementation language independence and object orientation. For IN and TMN, CORBA can only be used as an internetworking gateway to connect legacy systems, because the main middleware is not based on CORBA but is based on messaging systems such as SS7 and CMIP. The IN/CORBA gateway can translate between CORBA and SS7 networks. The TMN/CORBA gateway can translate between CORBA on one side and SNMP or CMIP systems on the other side. For TINA, it explicitly states all intelligence to be implemented in a Distributed Processing Environment (DPE). This concept of DPE explicitly

allows CORBA to become the main middleware for the delivery of telecommunications services. In this section, we show various impacts of CORBA in the telecommunications domain.

### **2.3.1 Internetworking Gateway with IN**

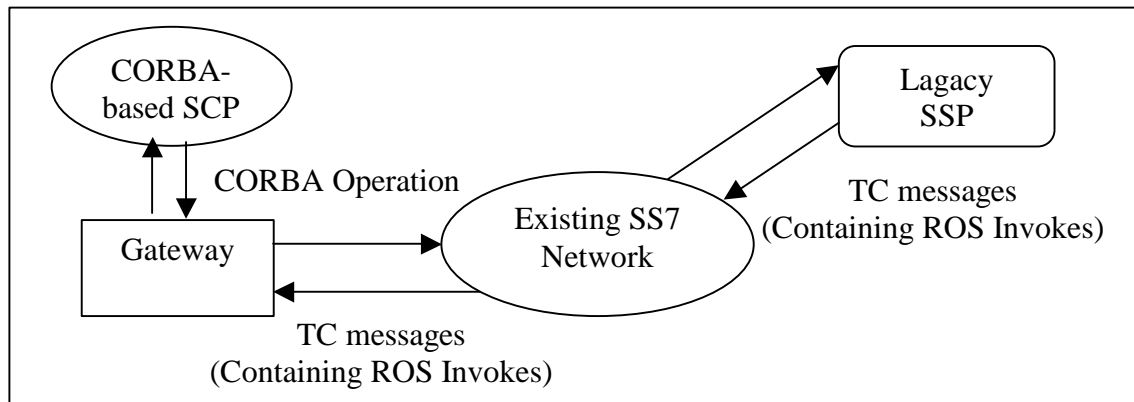
IN are developed based on the plain old telephony networks, in which computational infrastructure is used to process calls and provide services without the need of human intervention. The advent of IN infrastructure has led to the development of a range of services, which add value to the products of both the network and service provider. The infrastructure eases the introduction of new services by centralizing the service logic in a few dedicated service nodes, which allow services to be added without costly upgrading of the switching hardware and software infrastructure.

The OMG's IN/CORBA internetworking specifications [OMGTelecom1998b] enable CORBA-based systems to internetwork with existing IN infrastructure which uses Transaction Capabilities (TCs) for communication. With CORBA-based service objects, which use IIOP for communication, the specification promotes the adoption of CORBA for the realization of IN functional entities.

There are two proposed scenarios for the use of CORBA in IN signaling:

1. The interworking of CORBA-based IN Application Entities (e.g., a Service Control Point (SCP)) with legacy IN Application Entities (e.g. a Service Switching Point (SSP)) through a gateway mechanism, which provides a CORBA view of a legacy target and a legacy view of a CORBA target. As illustrated in Figure 2.9, the CORBA-based SCP has IDL interfaces created through Specification Translation of

the ASN.1 specifications of Intelligent Network Application Part (INAP).



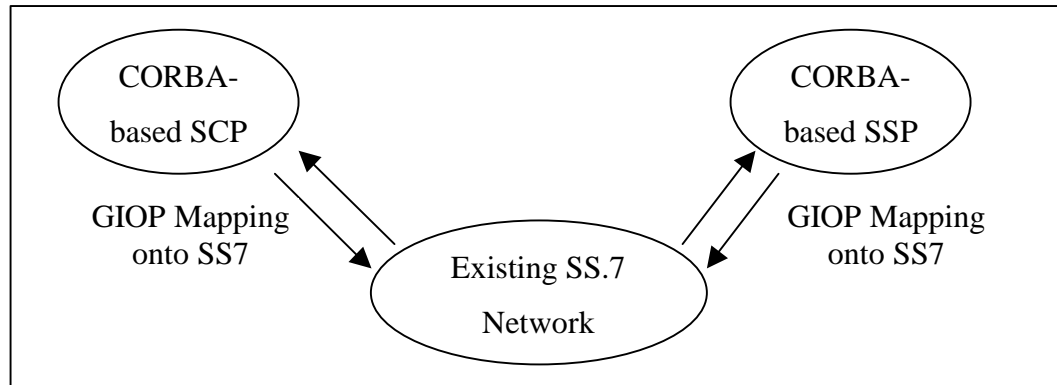
**Figure 2.9 Interworking between CORBA-based IN Application and Traditional IN Application (IN/CORBA Gateway)**

A tutorial on the design of a TC/CORBA inter-working gateway is given as a complement to the formal specifications [Mitra1999c]. The tutorial provides descriptions for the specification and interaction translation process. Examples are given for translating a TC/Remote Operations Service (ROS) specification to corresponding IDL interfaces, and for the dynamic behavior at the gateway when exchanging messages representing interactions between the CORBA and the TC/SS7 domain. This approach is similar to that used in our design of signaling for IP telephony services.

2. As shown in Figure 2.10 for the second scenario, the internetworking of CORBA-based IN Application Entities uses the existing SS7 infrastructure as a transport network for GIOP messages. The ORB hides the use of SS7 as a transport mechanism from the interacting CORBA objects. The GIOP mapping onto the connectionless Signaling Connection Control Part (SCCP) protocol of the SS7 protocol suite, the so called SCCP Inter-ORB Protocol (SIOP), which allows inter-ORB communication



over SS7, is defined [Fischbeck1999]. This approach brings the advantages of CORBA to the telecommunication domain without requiring the exchange of large parts of an operational network.



**Figure 2.10 Internetworking between CORBA-based IN Applications using SIOP (SS7 as Kernel Transport Network)**

The CORBA interfaces in the IN/CORBA specification provide standardized interfaces, which allow more open and distributed implementations of IN services. The common CORBA approaches to management and service provisioning produce a more integrated network and less cumbersome service management. These approaches have the added advantage of providing a homogeneous interface for any SS7 protocol stack implementation, reducing technology lock-ins, and allowing service creation that is independent of proprietary SS7 protocol stack implementations.

### 2.3.2 Internetworking Gateway with TMN

The work of CORBA/TMN Internetworking has been undertaken both within the Network Management Forum (NMF) in the JIDM and within OMG in the Telecom Domain Task Force (DTF). The OMG Telecom DTF issued a Request for Proposals (RFP) for a CORBA/TMN internetworking specification. In response, NMF JIDM advocated an

“adapter” approach, which includes a GDMO (Guidelines for the Definition of Managed Objects) to IDL compiler and a CORBA/CMIP gateway kernel. These gateways can be built as an adapter service on the top of CORBA through static or dynamic translations, both from the JIDM working group.

An example of CORBA/TMN integrated architecture is shown in Figure 2.11. The translation data, which is produced from the translation of GDMO and ASN.1 definitions, provides the mapping between IDL methods and parameter types and the corresponding Common Management Information Services (CMIS) requests and ASN.1 types. The translation data is stored in the gateway as a set of managed objects. The basic gateway kernel is in itself an agent offering both a CORBA and a CMIP based management interface, and translation data may, consequently, be updated from either CORBA and/or CMIP based management application at runtime [Rasmussen1998].

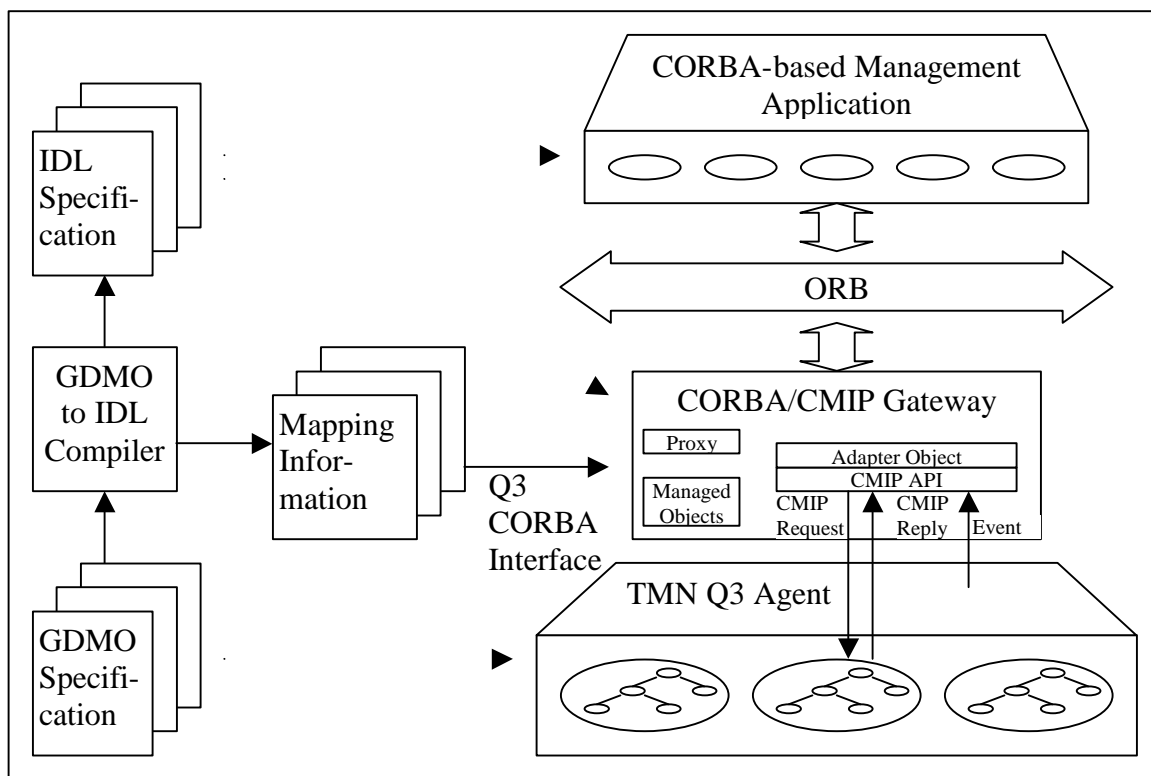


Figure 2.11 CORBA TMN Integrated Architecture

An advantage with this approach is that there is only one notation (i.e., IDL) to specify managed and managing systems. Using a general-purpose CORBA interaction model implies an important change with respect to the traditional way of developing network management applications, which strongly relies in the use of precise protocol stacks (for instance Q3). CORBA applications are independent of specific communication protocols, which helps to integrate network management with other telecommunication software (e.g., service control and management), and eases the tasks of the programmer, who can work with more familiar general-purpose development toolkits.

## Chapter 3

### Control Protocols for Multimedia Communications

---

At the early stage of the research, we did some studies on three multimedia control specifications, i.e., ITU-T H.323/H245, OMG Control and Management of Audio/Video Stream, and DSM-CC from DAVIC. In this chapter, we review the specification principles for each of them and give the comparison summary at the end of the chapter.

#### 3.1 ITU-T Recommendation H.323/H.245

The ITU-T Recommendation H.323, “Visual Telephone Systems and Terminal Equipment for Local Area Networks which Provide a Non-Guaranteed Quality of Service”, serves as the "umbrella" for a set of standards defining real-time multimedia communications for packet-based networks [ITU1996a]. Much of the excitement surrounding the H.323 standards is due to the ability of H.323 entities to communicate over the Internet or managed IP networks. The standards under the H.323 umbrella define how components that are built in compliance with H.323 can set up calls, exchange audio and/or video, participate in conferences, and inter-operate with non-H.323 endpoints.

As one of the H.323 subordinate specifications, H.245, “Control Protocol for Multimedia Communication – Line Transmission of Non-Telephone Signals” [ITU1996c], specifies the in-band signaling protocol necessary to actually establish the media requested for a call, negotiate the media capabilities, and issue the commands necessary to open/close the media channels. Here, in-band messages are those that are transported within the channel or logical

channel to which they refer. The H.245 signaling entities are required for media control functions in multimedia communications. H.245 message syntax is fully defined using the ASN.1, while the protocol procedures with state changes are separately described. The Recommendation covers a wide range of applications, including storage/retrieval, messaging/conversational and distribution services. The protocol itself does not cover quality of service (QoS), so it is intended to be used with a reliable transport layer protocol, like TCP.

H.245 signaling is established between two endpoints, an endpoint and a Multipoint Controller (MC), or an endpoint and a Gatekeeper. H.245 specifies a number of independent protocol entities, which support endpoint to endpoint signaling. A protocol entity is specified by its syntax (messages), semantics, and a set of procedures, which specify the exchange of messages and the interaction with the user. H.323 endpoints support the syntax, semantics, and procedures of the following protocol entities:

- ?? Master/slave Determination
- ?? Capability Exchange
- ?? Logical Channel Signaling
- ?? Bi-directional Logical Channel Signaling
- ?? Close Logical Channel Signaling
- ?? Mode Request
- ?? Round Trip Delay Determination
- ?? Maintenance Loop Signaling

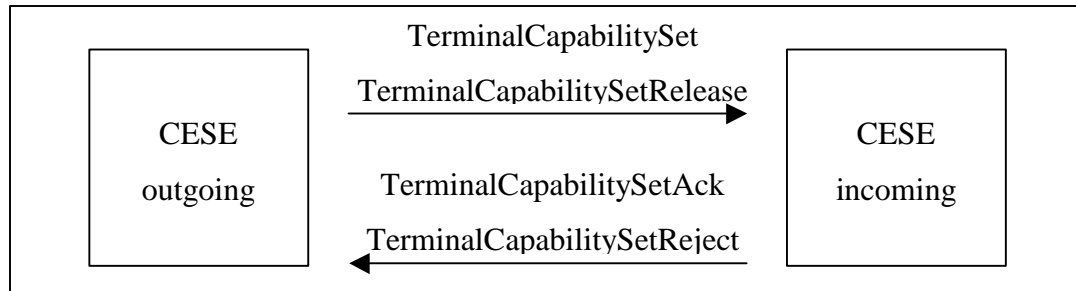
H.245 messages fall into four categories: Request, Response, Command, and Indication. Request and Response messages are used by the protocol entities. Request messages require a specific action by the receiver, including an immediate response. Response messages respond to a corresponding request. Command messages require a specific action, but do not require a response. Indication messages are informative only, and do not require any action or response. H.323 terminals shall respond to all H.245 commands and requests, and shall transmit indications reflecting the state of the terminals.

The H.245 Control Channel is a reliable channel used to carry the H.245 control information messages between two H.323 endpoints. The H.245 Logical Channel is the channel (either reliable or unreliable) used to carry the information streams between two H.323 endpoints. These channels are established following the H.245 OpenLogicalChannel procedures. An unreliable channel is used for audio, audio control, video, and video control information streams. A reliable channel is used for data and H.245 control information streams. There is no relationship between a logical channel and a physical channel.

The H.245 procedures that are mimicked in our implementation (chapter 5) are listed as follows. The signaling entities and corresponding H.245 messages are described in Figure 3.1, 3.2 and 3.3. Communication between the signaling entity and its local user, including the primitive, parameter and state transition, is not implemented, only the peer-to-peer signaling entity communication between incoming and outgoing signaling entity is concerned.

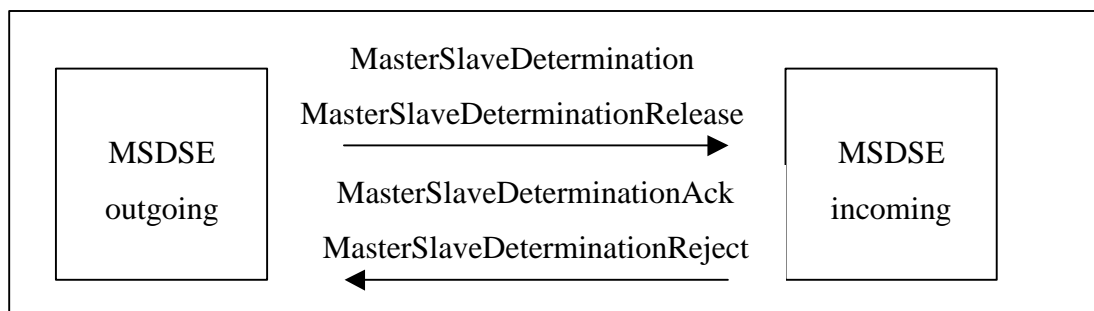
?? Capability Exchange: The capability exchange procedures are intended to ensure that only the multimedia signals to be transmitted are those that can be received and treated appropriately by the receiving terminal, i.e., ensuring compatible real-time bi-directional multimedia communication. These procedures require that the capabilities of each

terminal to receive and decode be known to the other terminal. The total capability of a terminal to receive and decode various signals is made known to the other terminal by transmission of its capability set. Terminals may reissue capability sets at any time. “CESE” stands for “Capability Exchange Signaling Entity”.



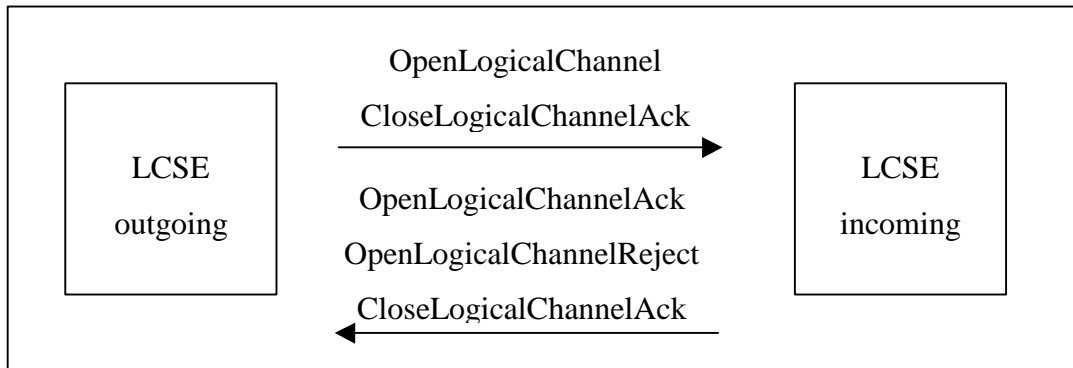
**Figure 3.1 Messages in the Capability Exchange Signaling Entity**

?? Master/Slave Determination: Conflicts may arise when two or more terminals involved in a call initiate similar events simultaneously and only one such event is possible or desired. To resolve such conflicts, one terminal may act as a master terminal and the other terminal(s) may act as slave terminal(s), according to predefined rules. The master/slave determination procedures allow terminals in a call to determine which terminal is the master and which terminal is the slave. “MSDSE” stands for “Master/Slave Determination Signaling Entity”.



**Figure 3.2 Messages in the Master/Slave Determination Signaling Entity**

?? Logical Channel Signaling: This acknowledgement protocol is defined for the opening and closing of logical channels, which carry the audiovisual and data information. The aim of these procedures is to ensure that a terminal is capable of receiving and decoding the data that will be transmitted on a logical channel at the time the logical channel is opened. “LSCE” stands for “Logical Channel Signaling Entity”.



**Figure 3.3 Messages in the Logical Channel Signaling Entity**

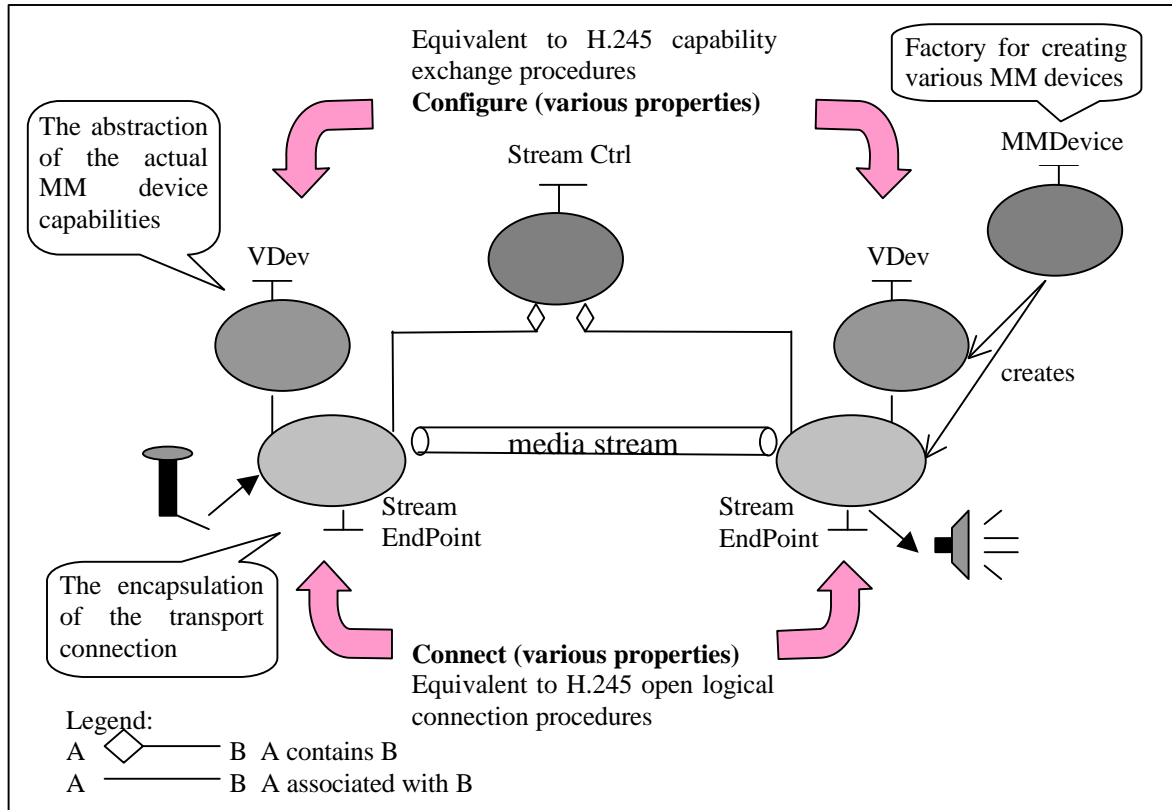
### 3.2 OMG Control and Management of Audio/Video Streams

In June 1998, the OMG put out a formal Telecommunications Domain Specification on the subject of Control and Management of Audio/Video Streams [OMGTelecom1998a]. The objective of this Streams specification is to extend the Object Management Architecture (OMA) to provide a basic support allowing OMA conformant applications to setup and manage streams between objects; a requirement that is becoming increasingly evident in telecommunication, multimedia and online markets. The specification shows that the stream and control management support should provide the necessary abstractions for multimedia communication streams, enabling the application programmer to develop distributed multimedia application without concern of the intricacies of the underlying communication mechanisms.



The emphasis of the specification is not on defining new protocols for the actual data stream transfer. Rather, the specification focuses on providing an administrative framework for dealing with streams. The specification defines interfaces for streams and flows, operations to set up, modify, and release streams, and functions for dealing with quality of service, flow synchronization and interoperability. The intention is to have a generic framework for stream management that can be used with a variety of lower-level network protocols.

The Streams architecture is based upon terminology defined in the ITU RM-ODP [ITU1995]. In this Model, a Stream represents continuous media transfer, usually between two or more virtual multimedia devices (devices are described as virtual because they are objects, which abstract upon the underlying physical multimedia device). Streams originate and terminate at a StreamEndPoint as shown as Figure 3.4. These end-point objects communicate over some forms of agreed communication channel and exchange continuous media according to a previously agreed negotiation process. Overall coordination of stream control is managed through a central StreamCtrl object. Each Stream Endpoint may contain multiple flows in either direction. The Stream Endpoint components are created, together with a Virtual Device (VDev), on receipt of a connection request by a Multimedia Device (MMDevice). Once created, the endpoint components are associated with the StreamCtrl object. The specification also defines two basic profiles for streaming services: a 'full' profile, in which endpoints and flow connections have accessible IDL interfaces, which maximizes system flexibility; and a 'light' profile, which is a subset of a full profile where flow endpoints and flow connections do not expose IDL interfaces.



**Figure 3.4 A Basic Stream Architecture from OMG**

Several research efforts and commercial products are developed around the CORBA Streams specification. A research group from Washington University – St. Louis has implemented the OMG A/V streaming model based on their real time ORB TAO [Munjee1999]. However, the commercial product OrbixStreams from IONA does not appear to be very successful, owing to a lack of customer interest, and we could not find any supporting information on its implementation details.

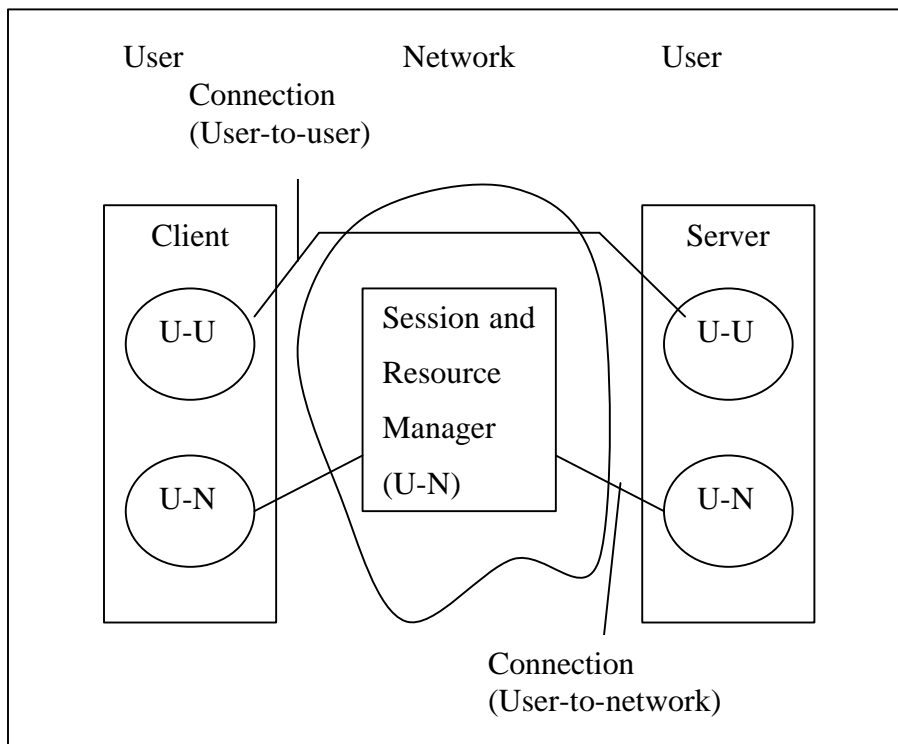
### 3.3 DSM-CC and DAVIC 1.4 Specifications

Digital Storage Media – Command and Control is an ISO/IEC standard developed as open protocols for the delivery of multimedia broadband service [Balabanian1996]. It allows various devices to access multiple services from multiple service providers. The key to DSM-

CC is in its flexibility; i.e., each protocol area can be used standalone, or in concert with other protocol areas, depending on the application(s) being addressed. The Digital Audio-Video Council has adopted DSM-CC as the protocol for control of multimedia interactive sessions, the resources within the sessions, and for service-level interactions [DAVIC1998].

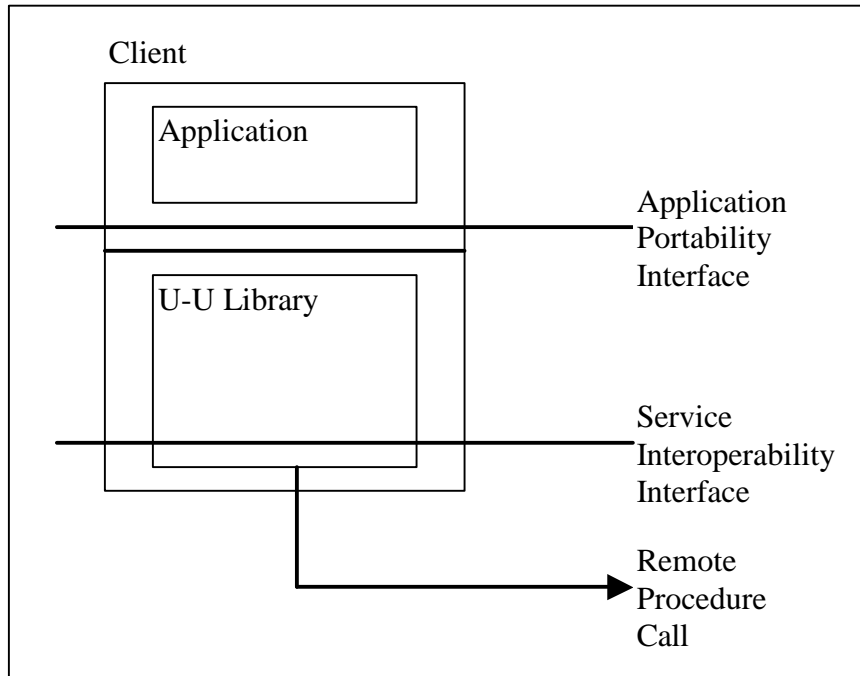
The functional reference model for DSM-CC is shown as Figure 3.5. A User-to-User (U-U) information flow is used between the network and the client or the server. User-to-Network (U-N) messages are exchanged over U-N connections and their purpose is to control sessions and network resources. The Session and Resource Manager (SRM) entity, which could be distributed over a geographical area spanning multiple network providers, terminates the U-N connection from a user. The U-N part defines a U-N interface protocol.

The DSM-CC U-U part provides a generic set of multimedia user-to-user interfaces, which enable a wide range of multimedia applications.



**Figure 3.5 DSM-CC Functional Reference Model**

As shown in Figure 3.6, two distinct interfaces are defined in OMG IDL, i.e., the Application Portability Interface for programmers writing applications that run on clients, and the Service Interoperability Interface to allow clients and servers from different manufacturers to inter-operate. The IDL of Service Interoperability Interface leads to a fixed bit pattern on the wire once the Remote Procedure Call (RPC) encoding scheme and message set have been chosen. DAVIC has chosen Universal Networked Objects (UNO, the CORBA RPC) and CDR encoding for its specification of DSM-CC.



**Figure 3.6 Application Portability and Service Interoperability Interface**

### 3.4 The Comparison Summary

The detailed comparison results are submitted in our first stage project report [Lu1999]. Briefly, the CORBA Streams specification addresses overall stream establishment and management. The DAVIC stream can be managed as a flow under the CORBA architecture.

Some of the procedures in H.245 can be mapped to the CORBA Streams specification. For example, as previously shown in Figure 3.4, the capability exchange and the open logical channel connection procedures in H245 are similar to the device configuration and connection negotiation process in the CORBA Streams specification. However, each specification has a different approach in realizing the communication. In the CORBA Streams specification, binding between devices can be created through invocation on the MMDevice interface's bind() method. In H.245, a signaling entity defined in each procedure issues messages to its remote peer signaling entity based on predefined message primitives. In the following chapters, we will demonstrate our CORBA-based interface centric approach on implementing H.245, a formal message-based protocol.

## Chapter 4

### The Interface Centric Approach

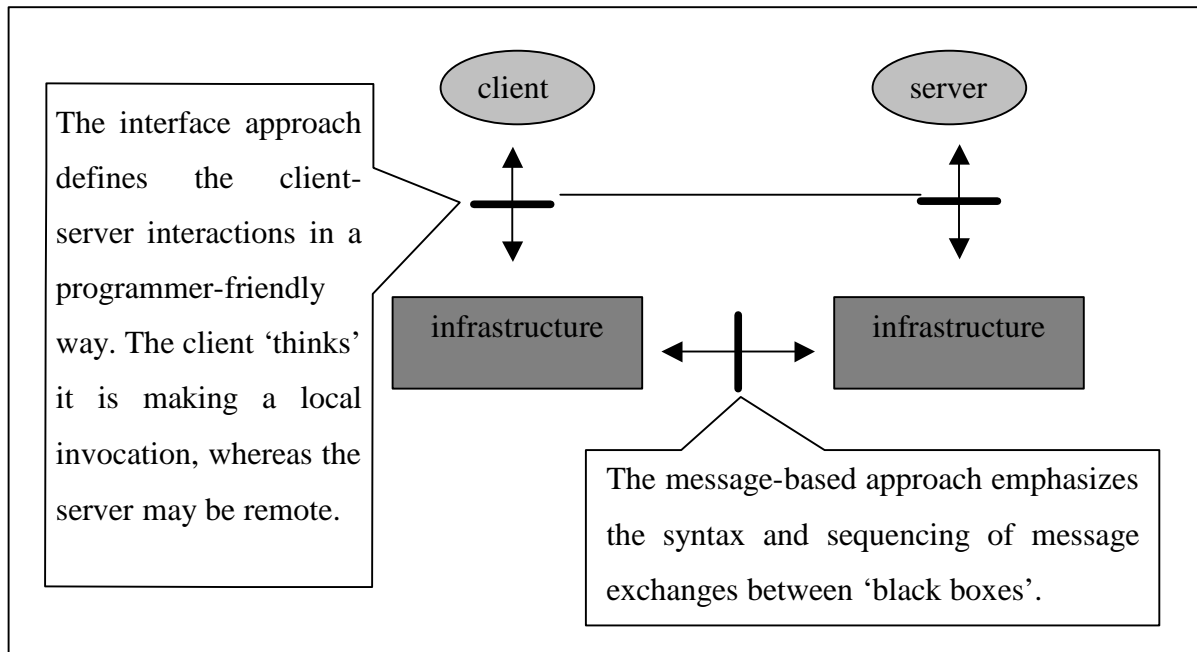
---

To respond to the problems, concerns and industrial trends overviewed in Section 1.1, a new model of signaling for IP-based telephony services is needed which permits the flexible distribution of control capabilities between endpoints, the network and third party service providers. The design should be independent of both network and access. Rather than design a control infrastructure from the bottom-up using low-level, message-based interfaces, which is the current approach as we described in section 2.1, the future model should be designed on a distributed computing platform onto which common capabilities are pushed, and are accessed by applications using standardized or open interfaces. Distributed software technologies, such as OMG's CORBA, provide the underlying foundation, removing the need for each new protocol design to specify data representation and transport reliability [Schulzrinne1997]. They make the communications between distributed objects transparent to the programmer by defining a standard way, i.e., the interface provided by a server to a client.

#### 4.1 A Comparison of Interoperability Reference Points

The interface-based specification style defines what needs to be done by the server at the request of the client, while the message-based specification emphasizes at least one aspect of how it should be done, i.e., the messages exchanged. In the interface-based approach, a server provides a client the interfaces with the operations that may be invoked, and all associated data types that make up requests and responses. Communications is simply one

sub-system among many that are required to provide distribution of software. Like everything else in this approach, details of the networking are abstracted through an interface definition so that, at least in principle, any networking technology can be used. The message-based approach is purely an engineering approach, i.e., a solution is provided for a particular networking technology, and any generality is only accidental. Figure 4.1 indicates the reference points where each approach defines conformance to provide interoperability.



**Figure 4.1 Comparison of the Interoperability Reference Points**

## 4.2 Requirement Analysis

This section illustrates the high level requirements of the interface-based approach.

### 4.2.1 Transparency Requirements

The interface-centric approach provides the foundation for application portability. First of all, interfaces raise the abstraction level so that application developers deal only with the semantics of the interactions within whatever programming environment they are familiar,

while the complexities of the distribution are shifted to the infrastructure. This approach has the advantage of distribution transparency, which consists of:

- ?? Access transparency: hiding the differences in data representation
- ?? Location transparency: masking the actual physical addresses of servers/objects
- ?? Failure transparency: masking the failure and possible recovery of servers/objects
- ?? Migration transparency: masking the relocation of servers/objects
- ?? Persistence transparency: hiding the deactivation and reactivation of servers/objects
- ?? Replication transparency: maintaining the consistency of a group of replica objects
- ?? Transaction transparency: hiding the co-ordination required to satisfy the transaction properties of a set of operations

Middle infrastructures such as the CORBA provide all these transparencies, either through ORB Core or Services. The CORBA object model identifies various distribution transparencies that must be supported within a single ORB environment, such as location transparencies, while the CORBA interoperability architecture extends the transparency to span multiple ORBs. An object request may have implicit attributes that affect the way in which it is communicated. These attributes range from fundamental mechanism such as reference resolution and message encoding to advanced features such as support for security, transactional capabilities, recovery, and replication. The attributes are provided by ORB Services, which in some ORBs are layered as internal services over the core, or in other cases are incorporated directly into an ORB's core. Note that the application developer does not have to "know" CORBA to be able to use it. In fact, the infrastructure is developed by third parties, either as a commodity or imbedded in some OS, and it is reusable as well. Therefore,



application developers do not have to develop their own, and incompatible, mechanisms for effecting some or all these transparencies.

#### **4.2.2 Component Requirements**

The major components of the interface-based approach can be classified as the distributed computing infrastructure, the interfaces, the server, the client (end point) and the network. The requirement for each of these components can be listed as follows:

- ?? The distributed computing infrastructure, such as CORBA, can provide the full set of distribution transparencies as defined in the previous section, as well as reliability, scalability, redundancy included in the design from the start.
- ?? The interfaces can be exposed to end users and third party service providers through appropriate access control mechanisms; the interfaces should hide implementation for easy functionality and software update. Therefore, if a new routing algorithm is developed for connection control, it can be invoked without affecting clients of connection control because the interface remains unchanged.
- ?? The server locations can be based on performance considerations to achieve minimized network communication latency, while their number can be based on capacity requirements.
- ?? The network and the client (endpoint) are not confined to a single distributed computing infrastructure, such as DCOM/ActiveX, or Java RMI/Java Beans/Enterprise Java Beans.

## 4.3 Selection of Techniques

### 4.3.1 The Selection of CORBA versus DCOM and RMI

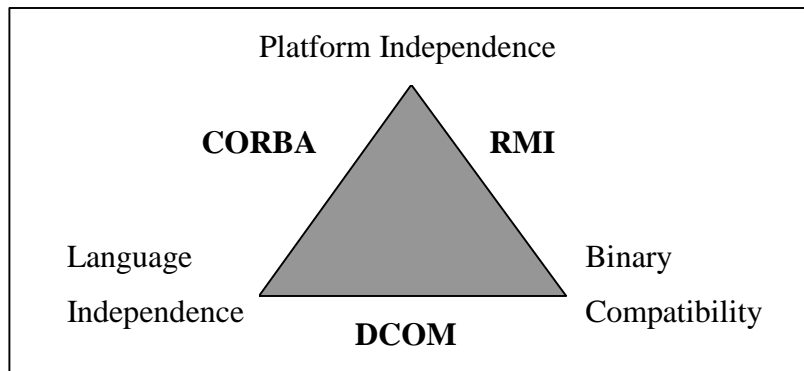
Besides the OMG CORBA, the IT industry has developed several alternative distribution infrastructures to make the communications between distributed objects transparent to the programmer, such as Microsoft's DCOM and Java RMI. All these seek to easily create distributed application software taking advantage of object-oriented programming concepts and practices by defining in a standard way the interface. Among those techniques, CORBA provides the greatest degree of software interoperability by using the language and platform neutral notation, i.e., the OMG IDL and with standardized mappings from IDL to the major programming languages. Microsoft has its own IDL, the so-called Microsoft IDL (MIDL), while RMI requires Java to program both clients and servers.

In all cases, communication takes place through a standard set of messages: IIOP in the case of CORBA and RMI, and DCE-RPC in the case of DCOM. The contents of the messages depend on the server interface/operation but the basic envelope and encoding scheme remains the same. Thus, one never has to learn the different binary formats for messages of different applications, as one would for ITU-T or IETF standards where the message set in each standard is different. One also does not have to define myriad interworking "gateways" which map one message set into another by brute force. Indeed, one may program distributed applications without even knowing what the actual messages look like, while at the same time being guaranteed "out-of-the-box" interoperability.

Interoperability is guaranteed through the use of a single network message representation, i.e., GIOP in the case of CORBA, which also takes care of byte ordering and

memory alignments, for example. GIOP messages exchanged over the TCP/IP, i.e., IIOP, is the OMG standard for interoperability through which ORBs implemented by different vendors can internetwork. Mappings of GIOP onto other transport mechanisms are also available, like the mapping onto SS7 transport as was mentioned earlier in section 2.3.1.

When we decide which technique is appropriate for IP signaling, the simple way is to consider the tradeoffs in terms of platform independence, language independence and binary compatibility as shown in Figure 4.2. CORBA is regard as an integration technology that offers both platform and language independence, which is the most likely requirement in a heterogeneous environment, while DCOM is platform-specific and RMI is language specific.



**Figure 4.2 Tradeoffs between the Different Distributed Computing Technologies**

### 4.3.2 The Selection of Java in Telecommunications

Java's platform independence and mobility have led to its increasing use in telecommunications applications. Extending from its initial usage in the development of graphical user interfaces for network and element management systems, Java is now found in end-user applications, downloadable protocol support, call control applications, integrated network management solutions, and open service creation environments [Jepsen2000]. Java has been particularly used to address the following problems and challenges in telecom:

- ?? The close coupling between intelligent network applications and the call state machine in the present PSTN result in complicated interactions among call processing features, and increase the complexity of new feature development. By providing a generic “middleware” layer for call/session control, Java allows decoupling of applications and state machines, and reduces feature interdependency.
- ?? Java-based component software can provide the necessary platform-independent protocol support for the convergence of circuit (PSTN) and packet networks.
- ?? Java standardized APIs (as will be mentioned later in section 4.4.2) and built-in security features allow new service providers to create new and innovative service without proprietary restrictions.

Java was originally released to develop network centric applications. Java’s native support for threads, garbage collection, exceptions and built-in complex data types orientation made the definition of CORBA’s Java language mappings a relatively painless process. Together CORBA and Java have dramatically raised the functionality and adaptability of the developed applications. By incorporating Java into the CORBA architecture, the result is a farther-reaching CORBA infrastructure and a more robust Java.

#### **4.4 Other Activities towards Open Interfaces**

There is increasing pressure to split apart the vertically integrated software and hardware solution for traditional telecommunication equipment. Third parties are trying to make use of much of the currently deployed signaling infrastructure to provide value-added services built on top of the generic services provided by networks. Examples for such services are connection control, consumer authentication and usage recording. This provides for quicker

and cheaper service delivery. It also reassures network operators that they can retain the control and intelligence in the network. These cross-industry initiatives are described as in the following sub-sections.

#### **4.4.1 PARLAY**

The Parley Group, an open multi-vendor forum founded in 1998, has been formed to create an explosion in the number of communications applications by specifying and promoting open network Application Programming Interfaces (APIs) on service functions such as call control, messaging and security. These functions intimately link IT applications with the secure network resources of the telecommunications world. The Parlay API describes two sets of interfaces: 1) Framework Interfaces, which provide for the common functions that are required to enable services to work together in a coherent fashion; and 2) Service Interfaces, which provide for the common functions that deliver complex services or sub-components of services. The Phase 1 of the API was focused on authentication, event notification, integrity management, discovery, etc. The Phase 2 extended the scope of the APIs to include IP network control, mobility, performance management, etc. [Parlay2000].

The Parlay APIs abstract from the network-specific details. Thus, an operation invocation from a third party to the network to connect a call to a given address does not require the application to know that the underlying signaling is via ISDN User Part (ISUP) on a SS7 network or H.323-based on a LAN. The APIs are not language specific and use the Unified Modeling Language (UML) to specify the interface classes. In this case, CORBA would make the ideal means to inter-work language and platform dependencies between the third party and the network. The inter-working is taken at the message passing level via IIOP,

DCOM or RMI, and achieved at the software level via the interface definitions, along with standard programming language mappings and the rules governing the order of operation invocations.

The Parlay APIs enable both third parties (external companies, operating outside the security domain) and network operators to build new applications that rely on real-time control of network resources. Figure 4.3 shows the architecture of Parlay 1.2 API, which defines object-oriented interfaces on both the network and client application sides of the API in the form of network interfaces (e.g., IparlayCall) and client application callback interfaces (e.g., IparlayAppCall). The third-party application vendor implements callback as part of their application to handle remote methods that are called from the network to the client application during a Parlay session.

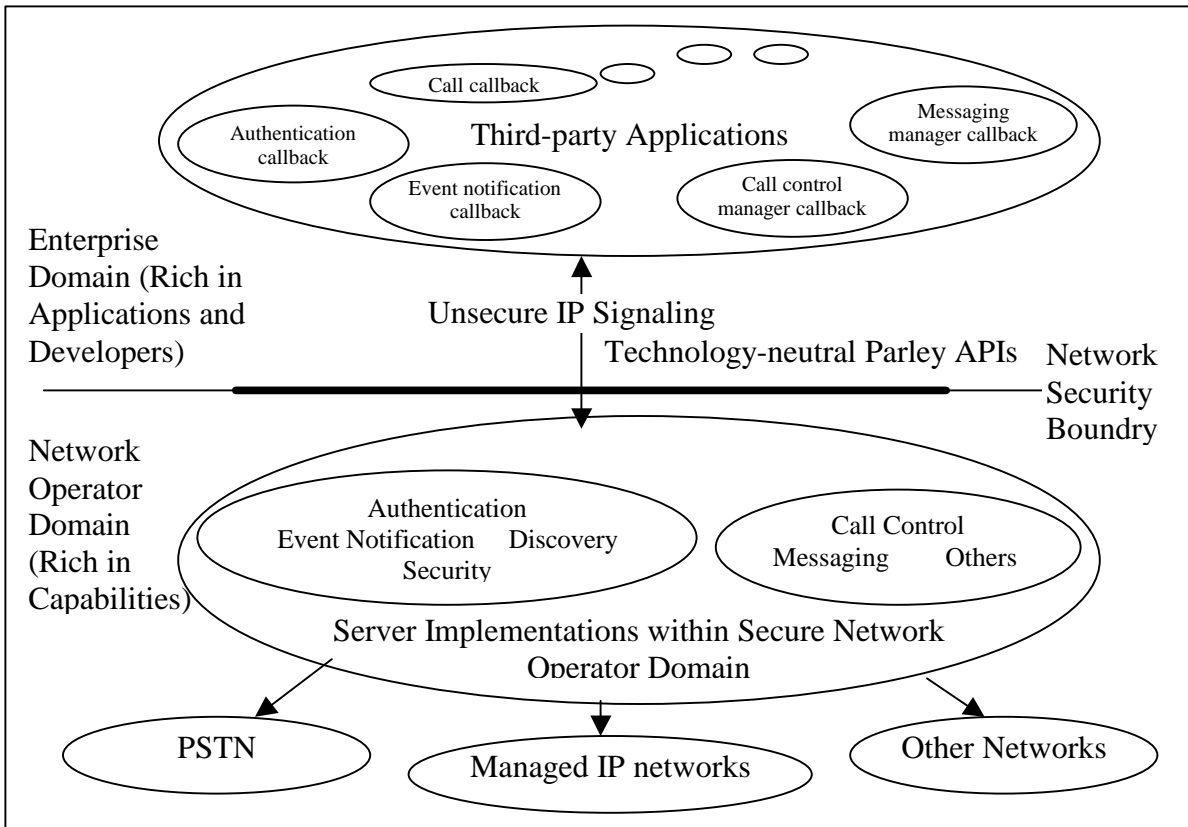
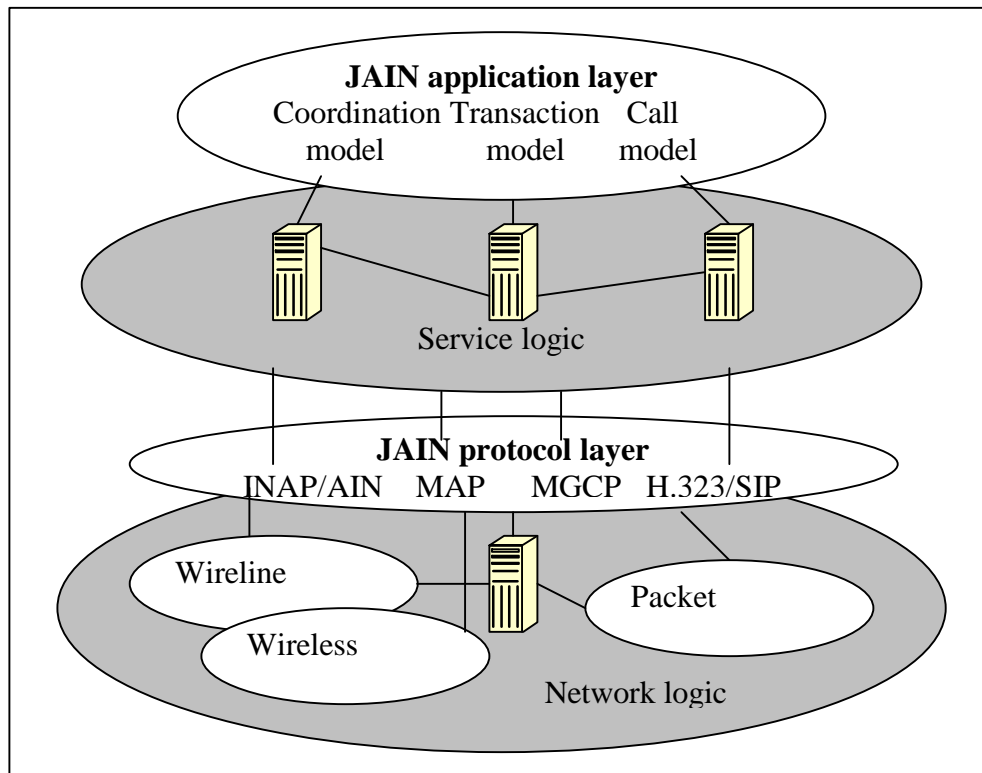


Figure 4.3 The Architecture of Parlay 1.2 API

#### 4.4.2 Java APIs for Integrated Networks (JAIN)

The JAIN APIs provide service portability, convergence, and secure network access to telephony and data networks for rapid development of next generation telecom products and services on the Java platform [SunJAIN2000]. Because JAIN technology provides a new level of abstraction and associated Java interfaces for service creation across PSTN, packet (e.g. IP or ATM) and wireless networks, it enables the integration of Internet and IN protocols, such as INAP, Mobile Application for GSM & IS41 (MAP), and it breaks the tight coupling of signaling applications to the SS7 protocol stack. As shown in Figure 4.4, the JAIN specification effort is divided into two areas (layers) of development:



**Figure 4.4 The JAIN Layered Approaches**

?? The Protocol layer API specifications specify interfaces to wireline, wireless and IP signaling protocols. By providing standardized protocol interfaces in a Java object model,

applications and protocol stacks can be dynamically interchanged and, at the same time, provide a high degree of portability to the applications in the application layer using protocol stacks from different vendors.

?? The Application layer API specifications address the APIs required for service creation within a Java framework spanning across all protocols covered by the Protocol layer APIs. The application layer provides a single call model, which can be viewed as a single state machine for multiparty, multimedia, and multi-protocol sessions for service components in the application layer. This state machine is accessible by trusted applications that execute in the application layer through the JAIN Call Control (JCC)/JAIN Coordination and Transaction (JCAT) API. The current proposal is to use the core part of the Java Telephony API (JTAPI) as JCC, further augmented with JCAT for a richer signaling model.

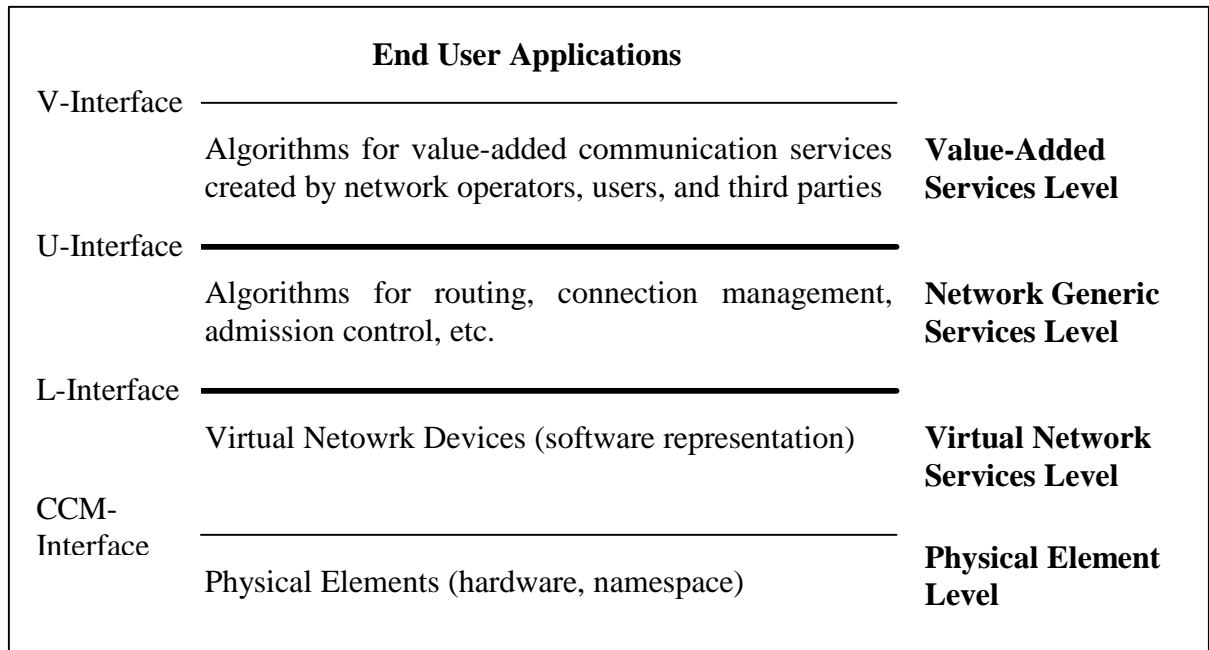
The JAIN Community defined a Java version of the Parlay API to bring the benefits of the Java language to Parlay API, and to promote an industry-wide adoption of the Parlay API. The JAIN Parlay Edit Group enhanced the JAIN architecture to support the Parlay API as its external API. The pure Java client side definition of the Parlay API focused on providing Parlay API features while removing the complexity of distributed computing technology and enabling application portability. Further information regarding JAIN Parlay API can be found in the IEEE Communications Magazine's special issue in April 2000 [Beddus2000].

#### **4.4.3 Open Programming Interfaces for Networks (PIN)**

IEEE P1520 – Reference Model for Open Programming Interfaces for Networks is an



ongoing standards development project. This project presents the basic principles that will enable the deployment of innovative and dynamic services on large open distributed systems that comprise both telecommunication resources and distributed software [Biswas1999]. The approach in developing programming interfaces for networks focuses on horizontal interfaces. These horizontal interfaces are essentially high level in nature, and deal with abstractions of network devices and corresponding states. The objective is to open the control/signaling interface to network nodes, such as switching or fabric control interfaces, by providing a set of standardized APIs, to allow software developers to write different control/signaling software running over the network without having to standardize signaling or control protocols. Figure 4.5 shows the reference model, which is composed of separated levels and the interfaces between these levels. The standards are specified in CORBA IDLs.



**Figure 4.5 The P1520 Reference Model: Open Programming Interfaces for Networks**

#### **4.4.4 Summary on Future Control Infrastructures**

Current message-based signaling protocols such as ISUP and Q.931 convey information for three separate functions: connection control, call control and service control. The original motivation for ISUP was to provide a simple and efficient link-by-link connection-control mechanism between switches. However, the messages were expanded over time with call and service control information. Such information was just not encapsulated very well, and led to the problems described in section 2.1.4 for message-based protocols. This situation closely connects service or feature upgrade to switch software upgrade.

A monolithic protocol such as ISUP or Q.931 to provide connection, call and service control is adequate for current needs. This is so because there is an intimate connection between all three functions, given the nature of the current telephony service which essentially resolves around the theme of a point-to-point circuit-switched connection. However, future multi-media, multi-point services over an IP infrastructure require a clearer separation if their signaling is to be more easily specified, standardized, implemented and deployed.

A software-centric control infrastructure would provide the three functions separately and differently. Such an infrastructure would be a collection of servers, for connection control services, for call control services and service control. A server is a definition of an interface to a set of functions specified in IDL (to ensure programming language and platform neutrality), thereby allowing the control to be fully distributed.

For example, connection servers would include operations to route a connection request, or modify a QOS parameter for an existing connection. They might be queried to find the best route through the network for a connection type of a given QOS prior to setting up the

connection. Call server interfaces would provide operations to trigger service invocations, determine end-to-end compatibility and availability of terminal equipment, etc. Application servers would offer service control interfaces specific to applications. The Parlay APIs are a good example of such application server interfaces.

There would also be additional servers, for example those that provide switching or fabric control interfaces, such as those being standardized by the IEEE PIN, to underlying physical devices. Security servers would provide access control interfaces. Location servers would provide an interface to mobility management functions. Billing servers would provide interfaces to record or retrieve charging related events.

## Chapter 5

### Design and Implementation

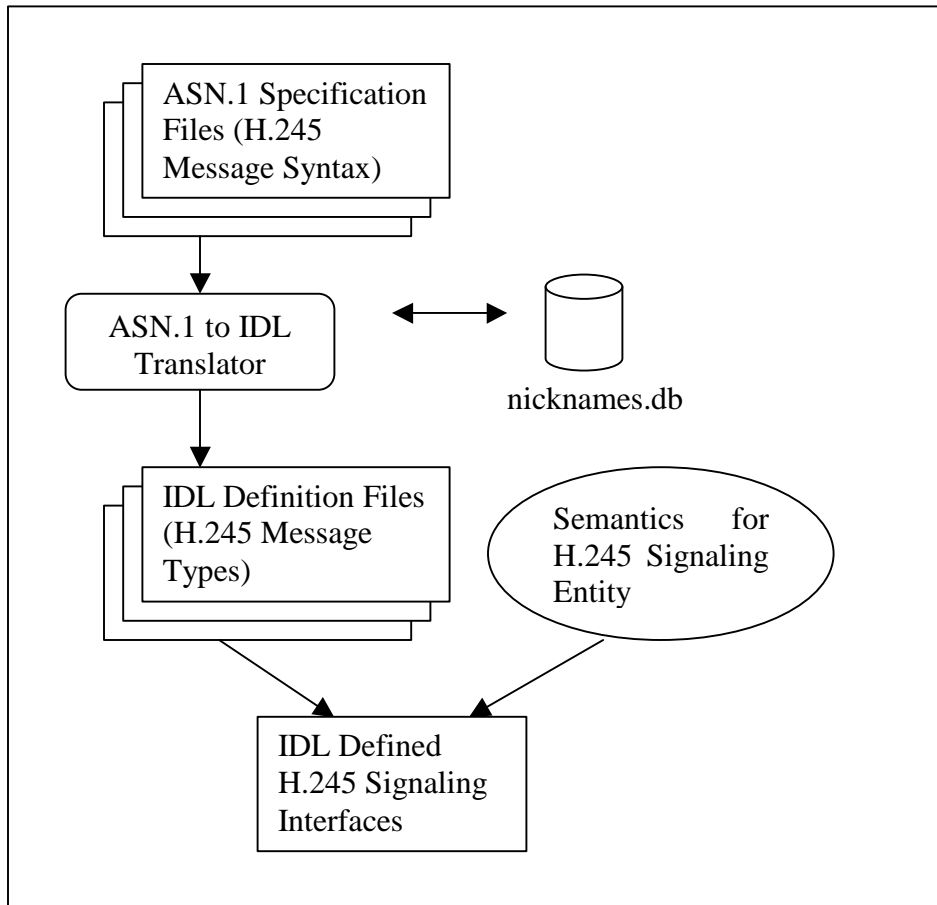
---

The CORBA-based interface-centric approach for services in IP telephony, which is simulated in the following sections, starts with describing the process for the conversion of H.245 ASN.1 messages. It is followed by the mapping of complex data types in section 5.2 and the object model in section 5.3. In section 5.4, various CORBA request invocation methods are illustrated. The use of CORBA Naming, Trading and Event Services are in section 5.5. The Visibroker specific development APIs are shown in section 5.6. The integration for the implementation to other H.323 control protocols is demonstrated in section 5.7.

#### 5.1 ASN.1 to CORBA IDL Translation

The Open Group and Open-Network Management Forum Joint Inter-domain Management group developed a technology that defines how network management components based on OSI and SNMP can inter-operate with CORBA-based components. The JIDM Technical Standard is a set of two specifications, i.e., Specification Translation and Interaction Translation [JIDM1997]. The Specification Translation for ASN.1 to/from IDL first provided the definition of model equivalencies between the domains of CORBA and OSI, enriched with specification for GDMO or SNMP SMI to/from IDL. The Interaction Translation defines how to perform OSI-like (and SNMP-like) services in CORBA, mainly through standard CORBA Naming Service, Event Service, etc. Messages in H.323 protocol stack are defined by ASN.1. The conversion of H.245 ASN.1 messages to IDL follows the

standardized ASN.1 to IDL rules. As mentioned earlier, a similar approach is used in the design of the CORBA/TC Internet-working Gateway. Commercial or public free translation toolkits are available from research institutes and universities [Nexus1999, BellLabs1998, Orbycom2000]. Most of the toolkits are designed for CMIP/GDMO or SNMP/MIB to CORBA IDL translation, which covers both specification and interaction translations. To speed up the translation process, based on the software program provided by Xenus Labs from Korea, we modified the ASN2IDL/Solaris translation compiler to statically translate all H.245 message syntax to IDL types. The generation process is illustrated as Figure 5.1. For H.245 interaction procedures, we have manually defined them as CORBA invocations in the IDL interfaces.



**Figure 5.1 Generation for H.245 Signaling Interfaces (ASN.1 to IDL)**

The translation process that is used to map ASN.1 modules includes the following steps:

1. Use as input the original published document, i.e., the H.245 message syntax.
2. Map each ASN.1 module to an IDL module in a separate IDL file, such as cE.idl for ASN.1 module of Capability Exchange Definitions, cEMC.idl for ASN.1 module of Capability Exchange Definition: Multiplex Capabilities.
3. Prior to mapping each of the clauses contained in an ASN.1 module, transform it into a canonical form.
4. Traverse the contents of the canonical ASN.1 module in order, and map each of the clauses.

~~///~~ EXPORT clauses are ignored.

~~///~~ IMPORTS clauses are mapped as a list of #include directives for the file corresponding to the imported module inside the ASN.1 module and a list of typedefs and constants.

~~///~~ Type assignments are mapped to typedefs in IDL.

~~///~~ Value assignments are mapped into either OMG IDL constants or operations in a constant interface at the end of the generated IDL module.

5. Re-order the generated IDL code to obtain valid OMG IDL code by eliminating forward reference.

All generated IDL interfaces for H.245 message types are submitted as Appendix 1 to the project report [Lu2000]. In this paper, a list to illustrate the message hierarchy mapping to IDL modules is attached in Appendix A. We have chosen to implement three H.245

protocol procedures described in section 3.1, follow the general steps to develop CORBA applications.

## 5.2 Mapping Complex Data Types

As mentioned earlier in section 2.1.2, the ASN.1 messages of H.245 use PER for binary encoding of data structures. Similarly, CORBA uses CDR, which like PER are not self-describing. The IDL generated stubs and skeletons will promise the correct matching of data types, or even interoperability between ORBs. Both rules use padding for data alignments. Several aggregated ASN.1 data types are used in H.245 messages, i.e., CHOICE (select exactly one), SEQUENCE (a grouping of dissimilar data types) and SEQUENCE OF (SEQUENCE significant in order). IDL uses constructed data types like the Enum/Union pair, Struct, Sequence to match with them. For example, the simplified H.245 CESE TerminalCapabilitySet is shown as Listing 5.1, the matching IDL data type definition is shown as Listing 5.2.

```

TerminalCapabilitySet ::=SEQUENCE
{
  sequenceNumber      SequenceNumber,
  protocolIdentifier  OBJECT IDENTIFIER,
                    -- shall be set to the value
                    -- {itu-t (0) recommendation (0) h (8) 245 version (0) 2}
  capabilityDescriptor      CapabilityDescriptor,
  multiplexCapability        MultiplexCapability,
  capabilityTableEntry      CapabilityTableEntry OPTIONAL
}
SequenceNumber      ::=INTEGER (0..255)
CapabilityDescriptor ::=SEQUENCE
{
  notStandard          BOOLEAN,
  capabilityDescriptorNumber  INTEGER(0..255)
}

```

```

}
MultiplexCapability ::=CHOICE
{
  nonStandard      INTEGER(0..65535),
  h222Capability   OCTET STRING
}
CapabilityTableEntry ::=SEQUENCE
{
  capabilityTableEntryNumber INTEGER(0..255)
  capability        OCTET STRING
}

```

**Listing 5.1 ASN.1 Message Syntax Example for H.245 CESE TerminalCapabilitySet**

```

struct CapabilityDescriptorType {
  ASN1_Boolean notStandard;
  ASN1_Unsigned capabilityDescriptorNumber;
};

enum MultiplexCapabilityTypeChoice {
  nonStandardChoice ,
  h222CapabilityChoice
};

union MultiplexCapabilityType switch (MultiplexCapabilityTypeChoice) {
  case nonStandardChoice : ASN1_Unsigned nonStandard;
  case h222CapabilityChoice : ASN1_OctetString h222Capability;
};

struct CapabilityTableEntryType {
  ASN1_Unsigned capabilityTableEntryNumber;
  ASN1_OctetString capability;
};

typedef sequence<CapabilityTableEntryType> CapabilityTableEntryTypeOpt;
struct TerminalCapabilitySetType {
  ASN1_Unsigned sequenceNumber;
  ASN1_OctetString protocolIdentifier;
  CapabilityDescriptorType capabilityDescriptor;
  MultiplexCapabilityType multiplexCapability;
  CapabilityTableEntryTypeOpt capabilityTableEntry;
};

```



};

### **Listing 5.2 IDL Data Type Example for H.245 CESE TerminalCapabilitySetType**

The IDL data description for messages needs to include `ASN1Types.idl` for mapping of ASN.1 primitive types, such as `ASN1_Boolean`, `ASN1_Unsigned`, etc. The content for `ASN1Types.idl` is given in Appendix A.

## **5.3 Object Modeling and Program Development**

Before a signaling interface can be specified, the functionality supported over the interface must be clearly defined. The starting point for the CORBA-based object model is an H.245 approved object model, mostly specified with signaling entities (SEs) communicating with messages defined in ASN.1. The scope of CORBA is the transfer of information in a distributed object environment, and not what is done with that information. Thus, we do not cover most of the local procedures within the SE in our design. In our simplified object model, the major component is the SE, which should compose both the Outgoing SE and the Incoming SE. In most cases, the Outgoing SE is the entity that generates most requests, like a CORBA Client; the Incoming SE implements the IDL with the object classes, like CORBA Server Objects.

The CORBA IDL specification was developed as the basis for interface's implementation. CORBA IDL has been designed to be independent of any programming language, and it only describes the operation signature, not its semantics. Shown as Listing 5.3, there are two kinds of interfaces for SEs, i.e., the SE factory interface and the actual SE interfaces.

```
#include <ASN1Types.idl>
```

```

module CESE {
    // converted ASN.1 types are first, followed with interface, exceptions and operations
    ...
    struct TerminalCapabilitySetType {...};
    struct TerminalCapabilitySetReplyType {...}; //for Ack or Reject
    struct TerminalCapabilitySetReleaseType {...};
    enum CauseType {localTimeout, remoteTimeout};
    ...
    interface H245SEFactory {
        CESE create_CESEOutgoing();
        CESE create_CESEIncoming(in CESE outgoingCESE);
        ...
    }
    // actual SE interfaces
    interface CESE {
        exception noResponse {CauseType cause;};
        //operations in two-way invocations
        TerminalCapabilitySetReplyType transferIndication (
            in TerminalCapabilitySetType tcs)
            raises (noResponse);
        void releaseIndication (in TerminalCapabilitySetReleaseType tcs_release);
        ...
    }
}

```

### Listing 5.3 IDL Example for H.245 CESE Signaling

The SEs are created from H245SEFactory Interface, which contains operations like “CESE create\_CESEOutgoing();” and “CESE create\_CESEIncoming(in CESE outgoingCESE);” to create the Outgoing CESE and Incoming CESE CORBA objects. The object reference of the Outgoing CESE is passed as parameter in creating the Incoming CESE. Both operations return the object reference to the interface CESE, which represents the actual CESE interactions. The interface for Outgoing SE and Incoming SE is identical, but the implementations are different based on the operations that each SE uses. The

interface may become different for complex cases. Each interface supports several operations that allow the SE to interact with the peer SE. These operations are similar in concept to H.245 signaling procedures. For example, the CESE interface supports an operation called “TerminalCapabilitySetReplyType transferIndication (in TerminalCapabilitySetType tcs) raises (noResponse);”, which instructs the Incoming SE to get the message. The message in complex data type, i.e. TerminalCapabilitySet, is passed as the parameter when the operation is invoked. The operation returns the result in TerminalCapabilitySetReplyType to indicate either “Ack” or “Reject”.

We followed generic ways to develop CORBA applications. Figure 5.2 shows an example diagram for the CESE development process. Compiling the CESE IDL interface generates Java source files which are compliant with CORBA’s Java language mapping. These files provide the implementation of client stub classes and server skeleton classes. The IDL compiler also generates helper and holder classes that allow the manipulation of IDL user-defined types. We develop the client code considering that the operations on a server are those declared in the IDL specification. In addition, we use the support from CORBA tools/services to be able to program how the client can find and bind to the server.

Most of Java ORBs support two ways of implementation for the operations declared in the IDL interface: inheritance and delegation. The inheritance-based approach requires that the class implementing an IDL interface extend a base class generated by the IDL compiler. This base class allows the ORB to forward incoming calls to the implementation object and provides marshalling and demarshalling functions. In contrast, the delegation-based approach eliminates the need for the implementation class to extend an ORB-generated class and allows it to implement a Java interface generated by the IDL compiler due to Java’s lack of

support for multiple inheritance. This approach is also called the “tie” approach because an implementation object is tied to a skeleton object at runtime by passing its reference to the tie object’s constructor. Passing an implementation object to its tie object is necessary so the ORB can, via the tie object, forward incoming calls to the implementation object.

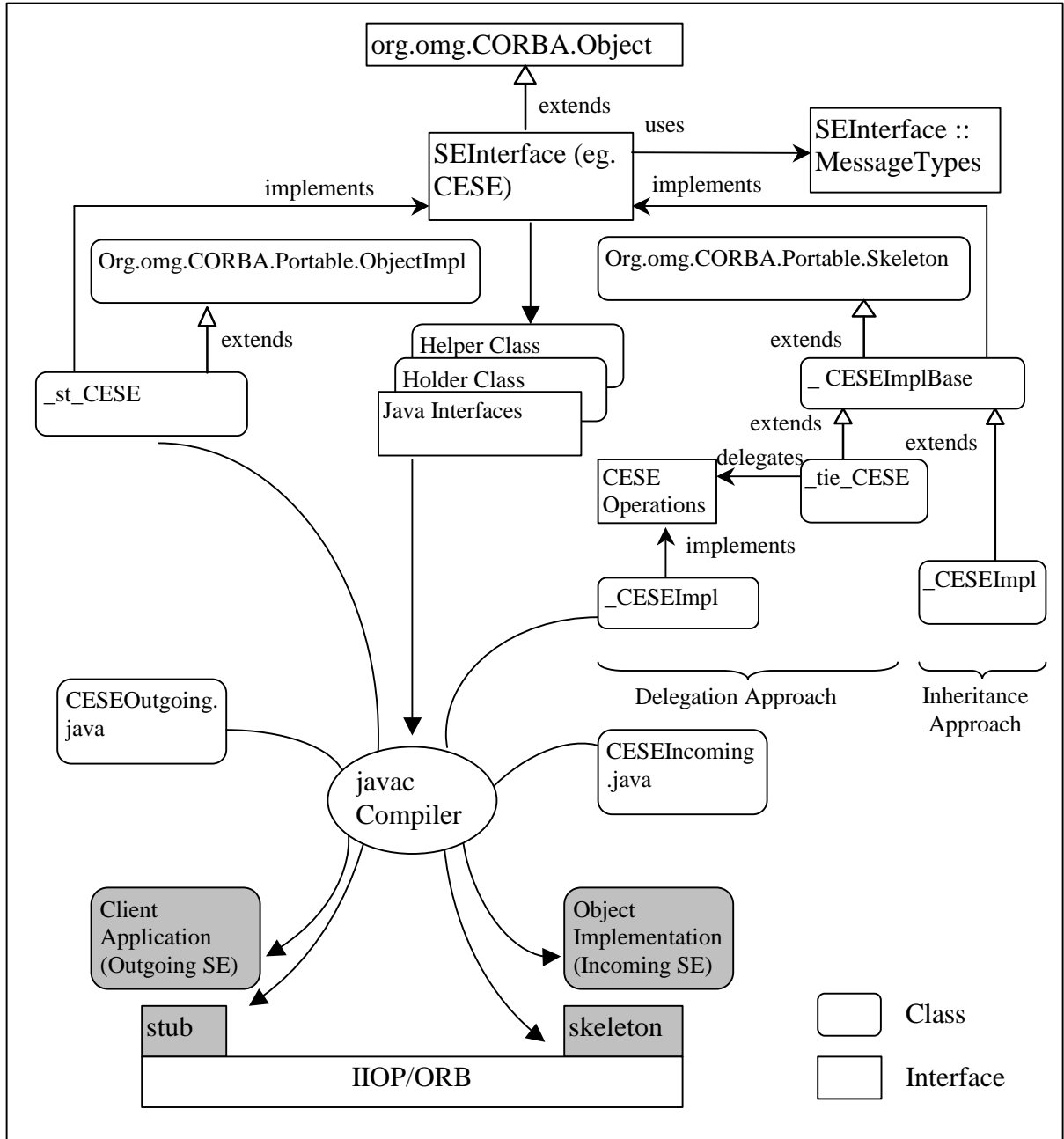


Figure 5.2 Example Diagram for Development of CORBA Applications

## 5.4 CORBA Request Invocations

In this section, we list various CORBA request invocation methods, including the current support and on-going OMG specification efforts.

### 5.4.1 Current Support for Requests

Standard CORBA remote method invocations are executed synchronously, i.e., the client is blocked while waiting for a reply to an invocation as shown previously in Listing 5.3. CORBA also supports asynchronous one-way invocations as defined in IDL without blocking the calling thread, and deferred synchronous requests used in dynamic invocation interface (DII) with later polling for the response. An example of one-way invocations in CESE interface is shown in Listing 5.4. Following the direction of message flow, the Incoming CESE provides the implementation of `transferIndication()`, and the Outgoing CESE provides the implementation of `transferAccept()` and `transferReject()`.

```
interface CESE {
    ...
    //operation in one-way invocation
    oneway void transferIndication (in TerminalCapabilitySetType tcs) raises (noResponse);
    oneway void transferAccept (in TerminalCapabilitySetAckType tcsa);
    oneway void transferReject (in TerminalCapabilitySetRejectType tcsr);
    oneway void releaseIndication (in TerminalCapabilitySetReleaseType tcs_release);
}
```

#### Listing 5.4 IDL Example for CESE Interface with One-way Invocations

### 5.4.2 Asynchronous Messaging

In May 1998, OMG published the Messaging specification [OMGTC1998a], which provides two asynchronous invocation models: Callback and Polling. In the Callback model,

a Callback object is registered at the time of the invocation. When the reply is available, that Callback object is invoked with the data of the reply. In the Polling model, the invocation returns an object, which can be queried at any time to obtain the status of the outstanding request. Asynchronous messaging permits applications to queue messages without blocking. Persistent message storage provides a way for messages to be delivered even if the sender and receiver applications are not running at the same time. However, there is no commercially available ORB so far supporting the asynchronous messaging.

## **5.5 Useful CORBA Services**

This interface-centric approach benefits heavily from CORBA's object services. As pinpointed in section 2.2.6, this section highlights only those services that provide a key functionality to the approach. CORBA Naming and Trading Services are intended being used for object discovery and location, the process happens during H.225 signaling period on behave of the H.323 Gatekeeper. The Event Service is used as one way to achieve asynchronous messaging in H.245 capability exchanges.

### **5.5.1 CORBA Naming Service**

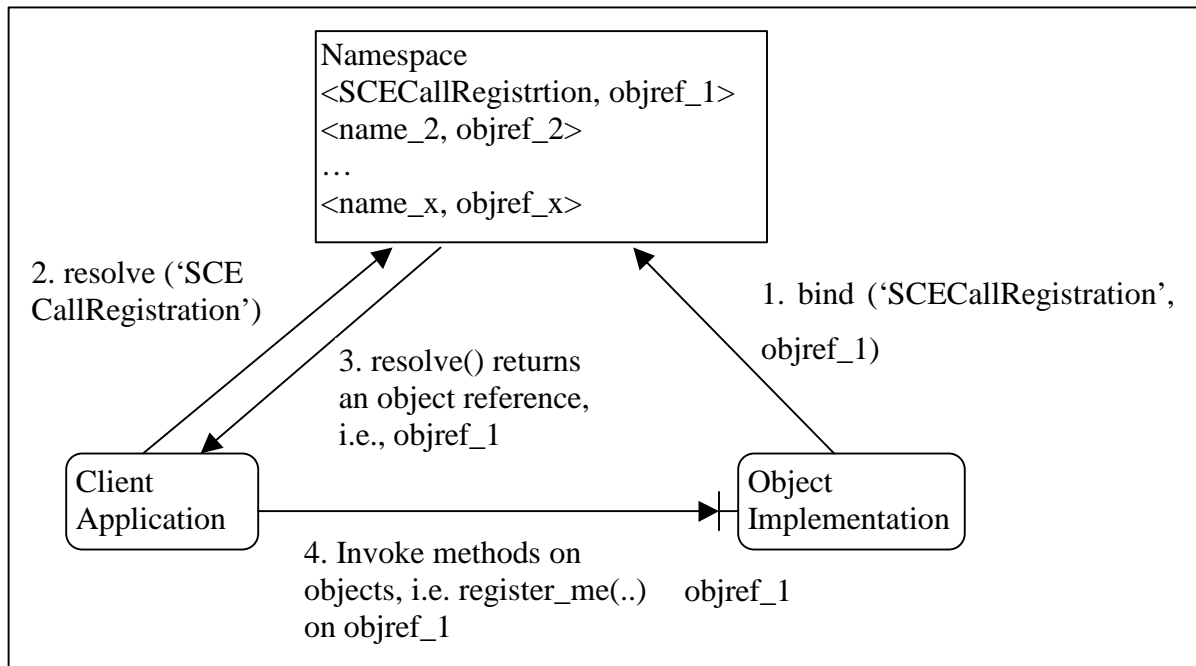
The Naming Service provides the capability for CORBA objects to find other CORBA objects using an easily distinguished naming convention. It provides a mechanism for associating remote objects in the network with a logical name within a searchable structure. It defines interfaces to describe names and interfaces to represent the contexts associated with each node. The same as other COS services, the Naming Service is just another object defined by its IDL. The operations of the Naming Service fall into the following three steps:

?? Obtaining an initial Naming Service Context

?? Binding and Resolving: operations that change the Naming Service

?? Navigating the Naming Service

Figure 5.3 gives an example of the call registration process. A Gatekeeper provides the object implementation for SCE call registration. The Naming Service is used to bind object references for implementations with hierarchical names. A resolve() query on the Naming Service returns the object references associated with a name to the Endpoint client application. Then the Endpoint communicates with the Gatekeeper for SCE call registration.



**Figure 5.3 Naming Services for Object Access (Call Registration Example)**

### 5.5.2 CORBA Trading Service

The OMG Trading Service facilitates the offering and the discovery of instances of services of particular types. A trader is an object that supports the trading object service in a

distributed environment. It can be viewed as an object through which other objects can advertise their capabilities and match their needs against advertised capabilities. Advertising a capability or discovering services is called “export.” Matching against needs or discovering services is called “import.”

To export, an object gives the trader a description of a service and the location of an interface where that service is available. To import, an object asks the trader for a service having certain characteristics. The trader checks against the service descriptions it holds and responds to the importer with the location of the selected service’s interface. The importer is then able to interact with the service. These interactions are shown in Figure 5.4 as an example for Gatekeeper behaviors. A Gatekeeper registers (exports) its service with the properties it can offer to the trader. An Endpoint client application searches the trader for all Gatekeepers that can serve it based on various criteria, such as the alias address. The trader returns the object reference to object implementation in the Gatekeeper that matches input criteria. The client communicates with the Gatekeeper directly.

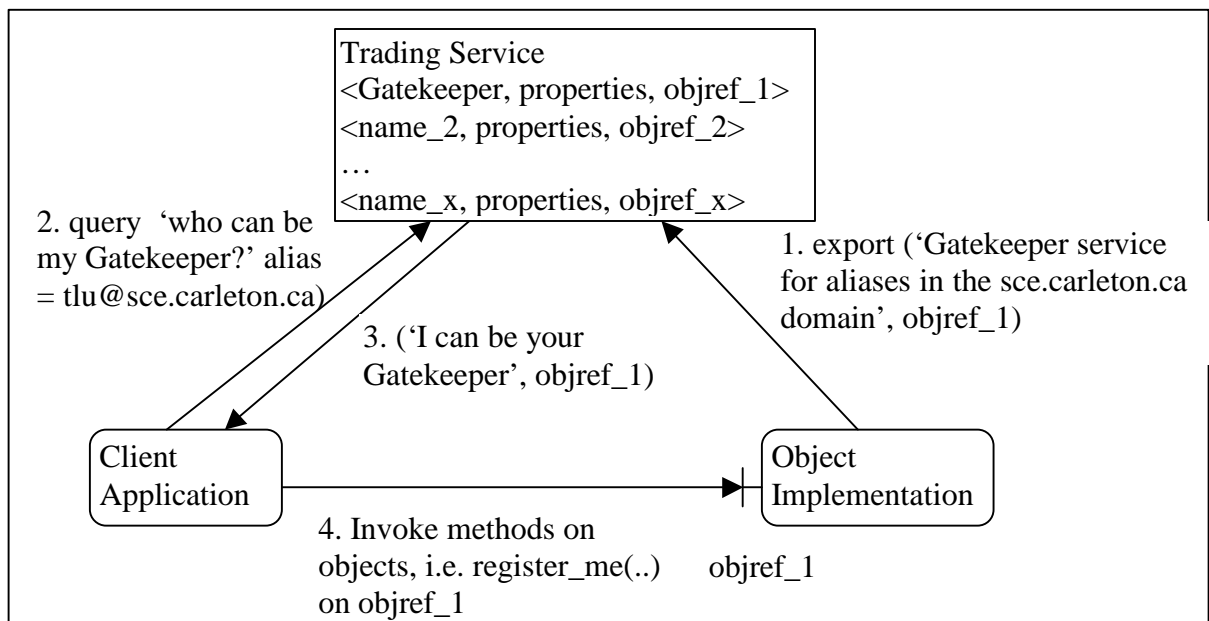


Figure 5.4 Trading Services for Object Access

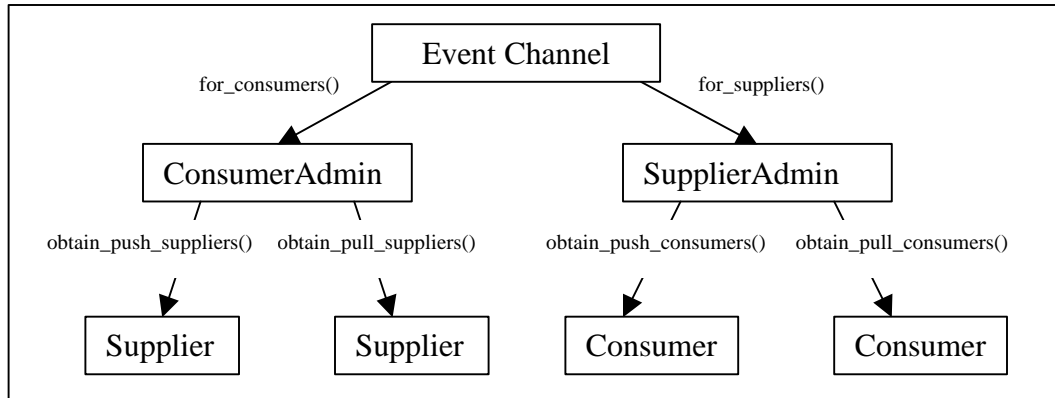


Visibroker does not provide free CORBA Trading Service. Some other ORBs like JacORB from MICO support free Trading Service [JacORB2000].

### **5.5.3 CORBA Event Service**

Large distributed environments may have issues like tracking callback objects in persistent storage, handling possible failures of connections, and tight coupling on callback interface. These issues require a different system to handle event delivery. CORBA Event Service provides solutions for these concerns. It supports asynchronous, disconnected communications between CORBA clients and servers. There are three primary participants in the Event Service: the Consumer that generates and transmits event messages, the Supplier that receives and further processes the messages, and the Channel that is used for communicating. Two general approaches are defined for initiating event communication: The Push and the Pull Model.

The COS Event API is split between two IDL modules. The first is the CosEventComm module, which contains the interfaces for the application developer to implement, i.e. the PushSupplier and PushConsumer interfaces. The second is the CosEventChannelAdmin module, which is implemented by the ORB vendor. The second module allows the application to join an event network and use an event channel. The relationship between objects in the CosEventChannelAdmin module is shown as Figure 5.5. At the top level is the EventChannel. From the EventChannel, two factory objects can be accessed: the SupplierAdmin and the ConsumerAdmin. From these two factories, we obtain one of four interfaces. The interfaces are for the proxies for consumers, suppliers, push and pull. The four Proxy interfaces are then used for runtime registration and notification.



**Figure 5.5 Factory Methods and Interfaces in the Event Service API**

We implemented the Push Model to mimic the procedure that H.245 CESE uses to notify its peer CESE of capability changes. Although the basic functionality has been easily achieved, we found the Event Service was relatively costly due to its way of creating event channels, delivering events and filtering data. The contents of events are of type Any, which provides a loosely typed interface between consumers and suppliers, and allows the consumer to extract the event data for intended information. However, it is a waste of resources for the event channel to send all events to all consumers; some of them have to discard the event data after the type-safe extraction. The Event Service still has a limitation on delivering events with a guarantee, or within a specific time period. Fortunately, OMG has addressed these concerns by adopting the Notification [OMGTC1998b] and Messaging Services.

## 5.6 Visibroker Development Environment

Our CORBA development environment is based on Visibroker for Java [Inprise1999] from Inprise/Borland, which is a leading provider of Internet access infrastructure and application development environment. Visibroker has been selected as a key part of

Ericsson's Operation Support System (OSS) for managing both the Global System for Mobile Communications (GSM) and the broadband Code Division Multiple Access (CDMA) networks, where the integration reference points specified for various systems are defined in IDL [InprisePress2000]. In this section, we demonstrate several Visibroker specific APIs.

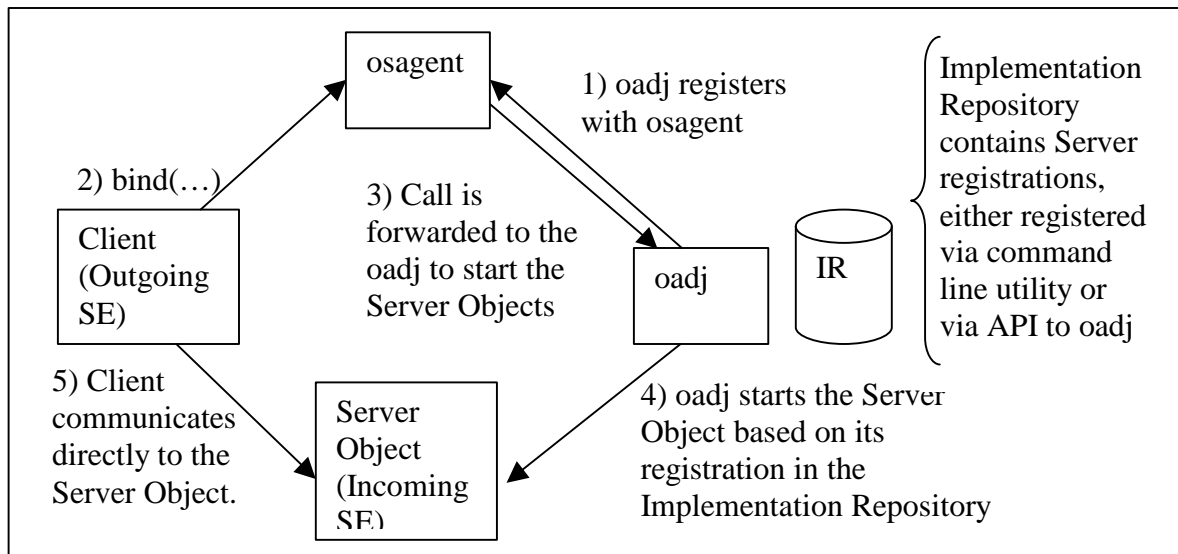
### 5.6.1 Object Activation Service (Visibroker)

As one of the value-added features, Visibroker provides Object Activation Daemon (OAD) for Java (oadj), which offers automatic activation for Server Objects. Oadj is ideally used for large systems with thousands of object implementation. The OAD works in conjunction with the CORBA Implementation Repository to start up object implementation on demand. After a Server Object has been registered with the oadj, its status of activation can be queried through `get_Status()` method, and the State object. As shown in Listing 5.5, the State object indicates whether the Server Object is active, inactive, or waiting for activation, which can be matched to the outgoing/incoming, idle, awaiting states defined for H.245 signaling entities.

```
Public final synchronized class com.visigenic.Activation.State extends java.lang.Object {
    public static final int_ACTIVE;
    public static final int_INACTIVE;
    public static final int_WAITING_FOR_ACTIVATION;
    public static final com.visigenic.vbroker.Activation.State ACTIVE;
    public static final com.visigenic.vbroker.Activation.State INACTIVE;
    public static final com.visigenic.vbroker.Activation.State WAITING_FOR_ACTIVATION;
    public int value();
    public static com.visigenic.vbroker.Activation.State from_int(int);
    public java.lang.String toString();
    static {}
}
```

**Listing 5.5 The State Object Class in Visibroker oadj**

The sequence to initiate a Server Object through oadj, and the communication model is shown in Figure 5.6. Once an osagent starts, the oadj, which represents the list of server objects registers with the osagent. When the Client starts, it binds with the osagent first. The Client then attempts to locate the Server Object via the osagent. Because the Server Object itself is not running, there is no registration for it in the osagent's memory table. There is, however, an entry for the oadj. Thus, when the request comes to the osagent, the osagent returns the reference of the oadj to the Client. The Client makes an invocation to the Server Object. The call actually goes to the oadj, which then starts the Server Object based of one of the four activation policies (Shared Server by default), and forwards the call to the spawned Server Object. The Client communicates directly to the Server object.



**Figure 5.6 oadj Communication Sequence Model**

### 5.6.2 Uniform Resource Locator (URL) Naming

The Visibroker URLNaming Service allows us to use any commercial Web Server as a Directory Service for retrieving IORs. This provides a convenient way for Server Objects to write their stringified IORs to a central location, i.e., any commercial Web location, and a

standard mechanism for a Client to obtain the stringified IOR via a standard URL address. It provides client applications with an alternative to locate objects without using an osagent or Naming Service, which enables client applications to locate objects from any vendors.

Both the Client and Server Object use the Resolver interface to register and obtain an IOR. After a reference to the Resolver has been obtained by the Server Object, we locate the Web Server through the methods within the Resolver interface, and copy the IOR of the Server Object to a specific directory on the Web Server. The sequence is shown in Listing 5.6. On the other side, the Client application specifies the URL when it calls the bind() method, which accepts the URL as the object name.

```
// Segment of CESEIncoming.java to create the Server Object implementation (URLNaming option)
CESEIncoming ceseIn = new CESEIncoming (?CESEURLName?);
// Export the created object
boa.obj_is_ready (ceseIn);
// Obtain the initial reference to the Resolver through resolve_initial_reference()
org.omg.CORBA.Object rawResolver = orb.resolve_initial_references (?URLNamingResolver?);
// Convert the Object to a Resolver through the use of the Helper call narrow()
com.visigenic.vbroker.URLNaming.Resolver resolver =
    com.visigenic.vbroker.URLNaming.ResolverHelper.narrow (rawResolver);
// Assemble the URL from IP address of the local machine, default port 15000, and CESE interface
java.net.InetAddress localAddress = java.net.InetAddress.getLocalHost();
String thisIP = localAddress.getHostAddress();
String url = ?http:// ? + thisIP + ? :15000/CESE.ior ?;
// Use force_register to overwrite the ior_file
resolver.force_register_url(url, ceseIn);
// Wait for incoming requests
boa.impl_is_ready();
...
```

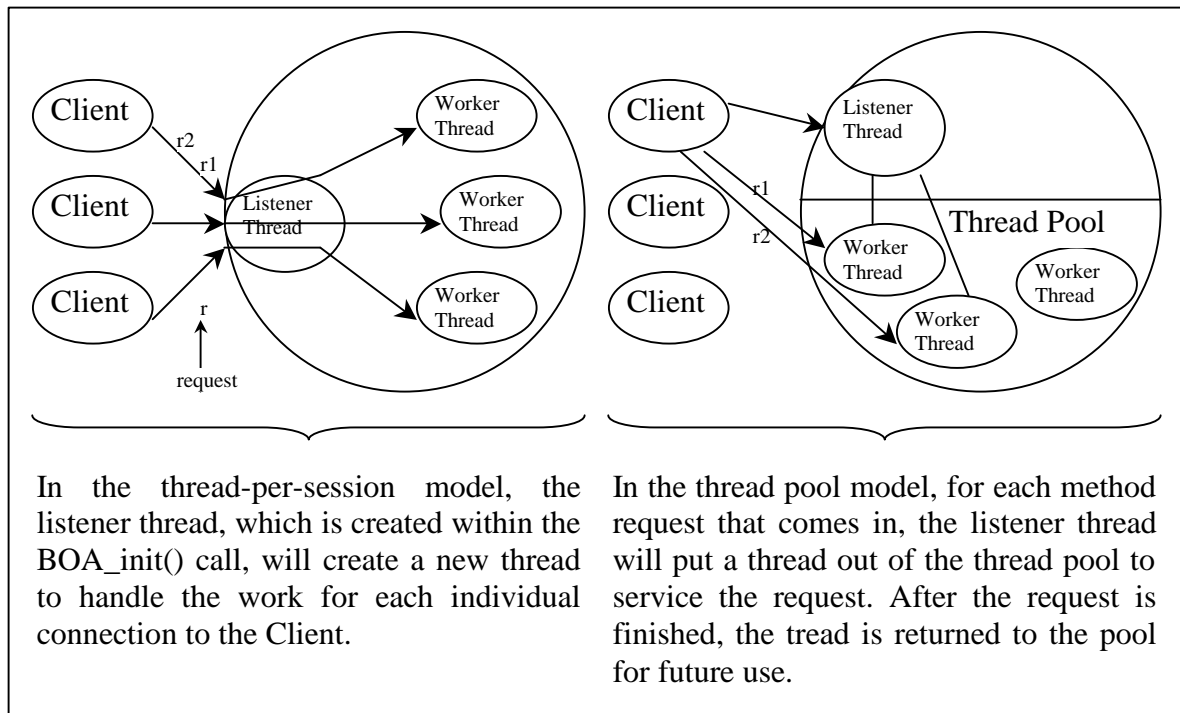
**Listing 5.6 Get the Initial Reference to the URLNaming Service (Server Object)**

In the coming CORBA 3, the interoperable Naming Service defines one URL-format

object reference, `iioploc`, which can be typed into a program to reach defined services at a remote location, including the Naming Service. A second URL format, `iiopname`, actually invokes the remote Naming Service using the name the user appends to the URL, and retrieves the IOR of the named object [Siegel1999]. The integration of CORBA with the Internet applications is an inevitable trend.

### 5.6.3 Multithreading and Connection Management

Java applications, by their very nature, support multiple threads and the Visibroker core automatically uses threads for internal processing, resulting in a more efficient request environment. Using multiple threads provides concurrency within an application and improves performance. Applications can be structured efficiently with threads servicing several independent computations simultaneously. Visibroker for Java offers two different thread models, the Thread-per-Session and the Thread Pooling model as shown in Figure 5.7.

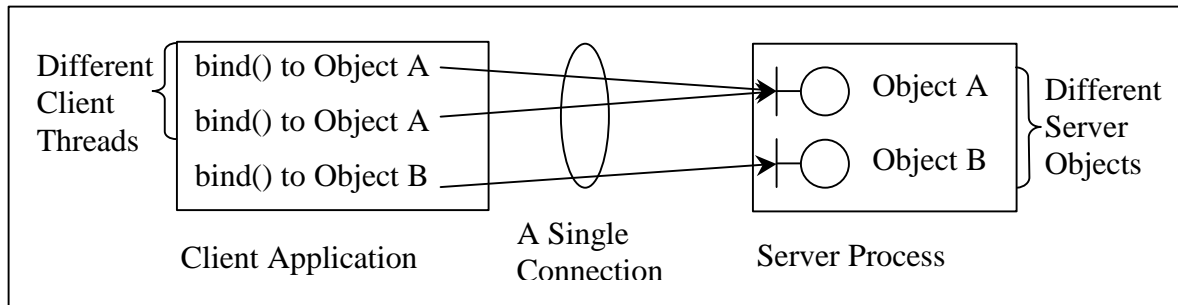


**Figure 5.7 Visibroker Thread-per-session Model and Thread Pool Model**

By default the Thread Polling model is the most efficient way to use resources. The Thread-per-Session model works well for applications in which clients invoke numerous requests on the same server over a lengthy period. However, if the server has many clients, it could result in many threads being created to handle them. When an object adapter is initiated, i.e., BOA\_init(), the object server can choose the thread policy by selecting a particular type of object adapter such as “TPool” or “TSession”, and specifying property parameters such as “OAThreadMax”, “OAThreadMin”, “OAThreadIdleTime”, “OAconnectionMax”.

Connection management is a built-in feature to give the ORB control over how many active connections any given Server Object can have active at any given time. A common problem with CORBA applications is that connections to a Server Object are constantly created and destroyed, but the resources are never cleaned up. This results in an otherwise available Server Object unable to accept any new connections because it does not have any available socket connections. The built-in connection management within Visibroker is designed to provide an automated mechanism for managing server-side socket connections to provide maximum scalability.

Overall, Visibroker selects the most efficient way to manage connections based on the above mentioned thread policies. The connection management minimizes the number of client connections to the server. All client requests are multiplexed over the same connection, even if they are originated from the different threads or bound to different Server Objects as shown in Figure 5.8. Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the overhead for new connections.



**Figure 5.8 Visibroker Connection Management**

## 5.7 Integration with Other H.323 Control Protocols

This section describes the overall scenario of H.323 signaling. The H.323 recommendation defines the following types of components: Gatekeepers, Gateways, Multipoint Control Unit (MCU) and Terminals.

- ?? The gatekeeper provides call control services to the terminals. Examples of such services are address translation, admission control, call authorization and directory services, etc. The RAS (Registration, Admission and Status) protocol defined in H.225 is used to communicate between a terminal and a gatekeeper. Christian Gosselin from UQAM has completed the implementation of this part through CORBA Naming Service.
- ?? The gateway provides real-time, two-way communications between H.323 Terminals on the packet-based network and other ITU Terminals on a switched circuit network or to another H.323 Gateway. It is responsible for providing all translations necessary for transmission formats and control procedures between the IP supported portion and the PSTN/ISDN part of hybrid calls. They are not discussed in this paper.
- ?? The MCU provides the capability for three or more terminals and gateways to participate in a multipoint conference. It may also connect two terminals in a point-to-point



conference, which later develop into a multipoint conference. The MCU consists of two parts: a mandatory Multipoint Controller and an optional Multipoint Processors (MPs).

?? The H.323 terminal components include a system control unit (H.225, H.245), video codec (H.261, H.263), audio codec (G.711), etc. The system control part of a terminal is composed of following three protocols:

1. The RAS signaling function is used for the dialog between a terminal and a gatekeeper. The associated channel, called the RAS channel, uses the UDP/IP protocol stack. A main function of the RAS channel is to allow the terminal to be attached to a gatekeeper by registering itself. Registration basically results in the update of the gatekeeper's address translation table. This allows other terminals to locate the registered terminal and to determine its transport address in order to initiate a call signaling channel.
2. The call signaling between two H.323 terminals is based on Q.931 messages. The call signaling channel uses a TCP/IP protocol stack. The call setup phase consists of sending a Setup message to the destination. The setup phase is considered successful upon reception of the Connect message from the called terminal. The next phase is the establishment of an H.245 channel, and the previously resolved location addresses in RAS are forward to H.245 signaling entities to carry on H.245 control functions.
3. As mentioned earlier, the H.245 protocol defines end-to-end control messages used for capability negotiation (e.g. supported codecs), master/slave status determination, opening and closing of logical channels, flow control messages, and so on.

H.323 defines "Endpoint" as an H.323 terminal, gateway, or MCU. An endpoint can call

and be called, and it generates and/or terminates information streams. Figure 5.9 shows an example of a control protocol diagram between two H.323 endpoints. Both endpoints are registered to the same gatekeeper, and are followed with direct call signaling and media transmission between endpoints. Later phases after two parties are in conversation through Real-time Transport Protocol (RTP) are not indicated. They would be such as procedures for closing down the logical channels through H.245, tearing down the call through Q.931 and releasing the resources used for the call through RAS.

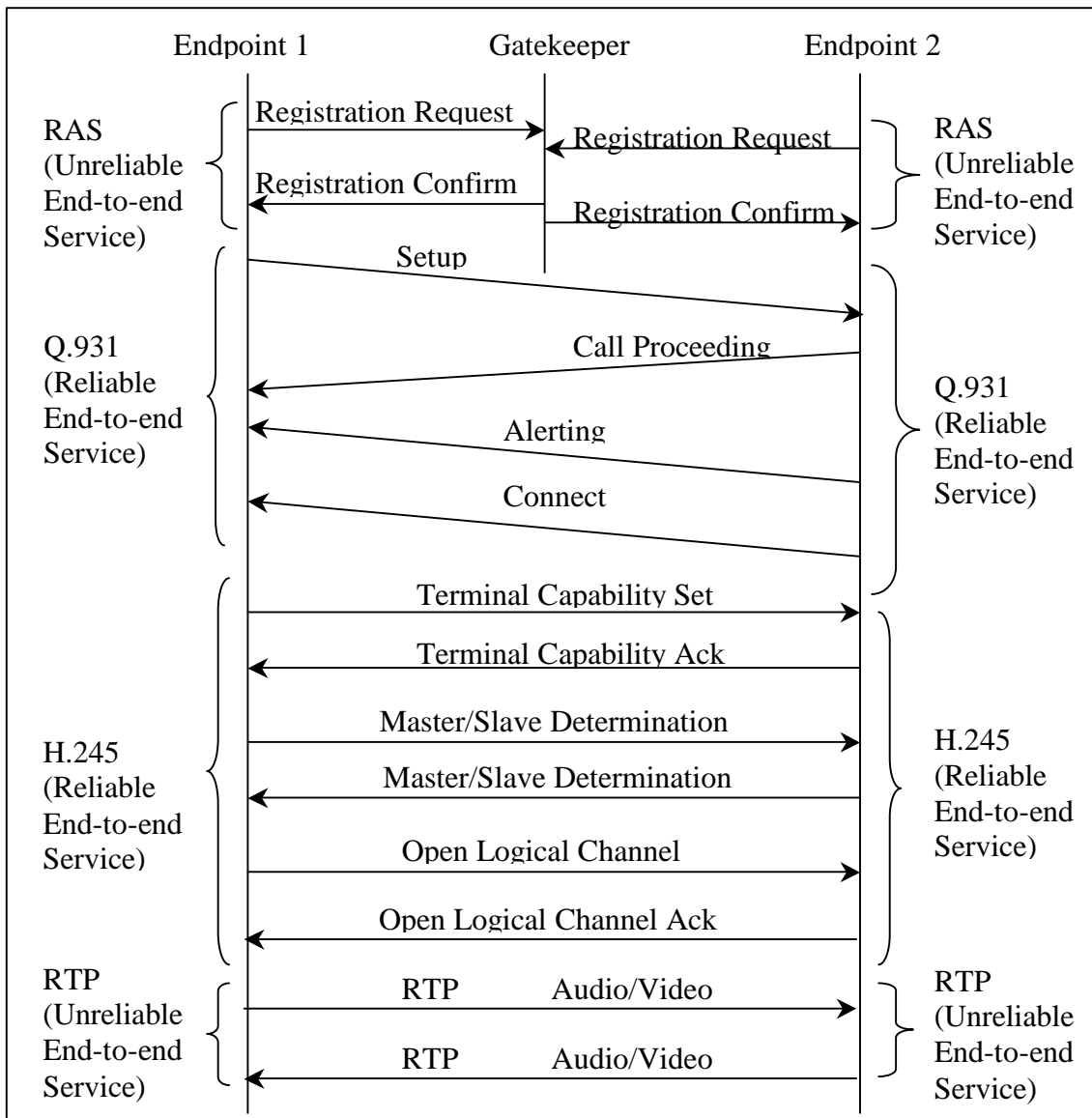
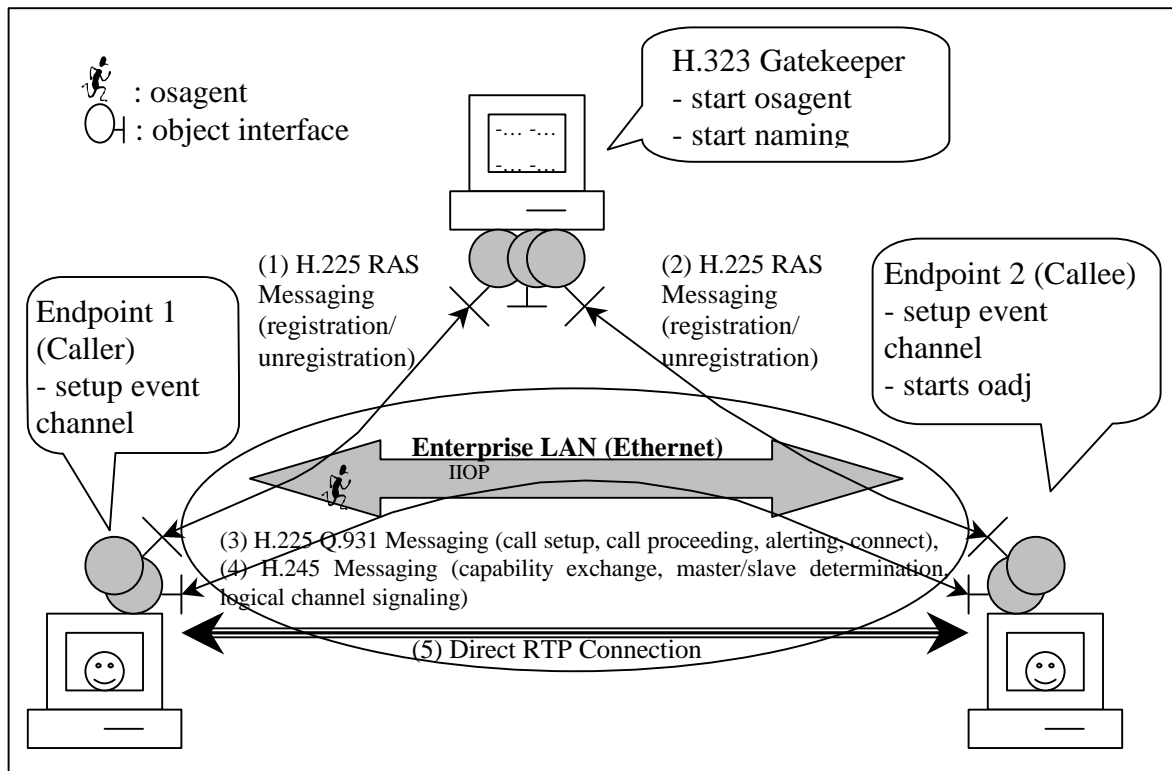


Figure 5.9 Example of Signaling Diagram for H.323 Control Protocols

Christian Gosselin implements the H.225 signaling in a parallel project. For the part of integration with RTP, we choose Sun's Java Media Framework (JMF) 2.0 as the API to incorporate the audio and video together, and convert (encode/decode) the source to packetized RTP data [SunJMF1999]. The RTP implementation will transmit the media using RTP protocol to the destination computer or network in case of unicast or multicast. To complete the demo, CORBA driven H.225/245 procedures were well integrated with the audio/video packet transmission through RTP. The advantage for this kind of combination is mentioned in a later section.

The overall system architecture for our interface approach for H.323 Signaling based on CORBA is illustrated as Figure 5.10.



**Figure 5.10 System Architecture of CORBA-based Interface Approach for H.323 Signaling**

## Chapter 6

### CORBA Performance Evaluation

---

When CORBA is used, the implemented distributed system may have performance and scalability problems, although they are functionally completed. CORBA performance problems normally fall into one of the following categories: Delay problem at the light load or delay problem at the heavy load. Typically, the goals of performance, scalability and maintainability are in conflict. For example, increased scalability often implies a reduction in performance. There are several techniques for gaining high performance in CORBA applications and ensuring that the performance remains acceptable when the number of clients or server objects increases dramatically. They are listed below, and some of them have been addressed in our design: refining the object model, threading models, distributed callbacks, client-side caching, performance monitoring, etc.

This chapter presents the performance results from our experiments with the focus on latency. Section 6.1 gives an overview of CORBA performance related issues, which serves as the underlying guideline for our experiments. Section 6.2 presents our test environment and experimental methods. Section 6.3 gives the results for our tests on ORB's benchmark (Visibroker for Java) and H.245 mimicked signaling procedures. Practical H.323 applications are most likely developed under integrated protocol stacks, using C or Java sockets passing PER messages. However, we did not have a usable H.323/H.245 test environment to compare with, although originally we were supposed to get one from Ericsson. In our benchmark tests, we compared Visibroker to sockets for raw data transferring to get valuable

performance indications between message size and transfer bandwidth. Section 6.4 gives the conclusions on performance experiments.

## 6.1 Performance Overview

### 6.1.1 IIOP Performance Limitations

The basic IIOP performance limitation is determined by two fundamental parameters of ORB sending remote messages: basic call latency, which is the minimum cost of sending any message at all, and marshaling/demarshaling rate, which determines the cost of sending and receiving parameter and return values depending on their size.

For CORBA call latency, the cheapest message is a one-way static invocation that has no parameters and does not return a result, like “oneway void nullCall();”. It sets the design limitation for the number of invocations that the ORB can deliver per time interval. However, the cost of call dispatch varies considerably among environments and depends on a large number of variables, such as the underlying network technology, the CPU speed, the operating system, the programming language, and the efficiency of the ORB run time itself. Developing distributed applications relies more on the performance of the network. Ethernet is the most common local area network. The capacity of Ethernet ranges from 10 Mbps to 100Mbps. Compared to typical CPU speeds, Ethernet is 3-4 orders of magnitude slower; compared to typical disk speeds, it is 2 orders of magnitude slower. Previous research showed that for a test environment such as 10Mbps Ethernet, between two UNIX workstations (SPARC 20, 50MHz) and commercial ORBs, general-purpose ORBs had basic call dispatch times of between 1 msec and 5 msec [Hennings1998]. The call latency time could be reduced through performance optimized ORBs and improved test environment

[Gokhale1998, Ahmad1999].

The marshaling/demarshaling rate depends on the type of data transmitted. Simple types, such as arrays of octet, typically marshaled fastest. Highly structured data, such as nested user-defined types or object references, is usually marshaled more slow because the ORB has to do more work at run time to collect the data from different memory locations and copy it into a transmit buffer. The marshaling rate also depends on the environment variables. For a rough guide from past experiments, marshaling rates were between 200kB/sec and 800kB/sec between two UNIX workstations (SPARC 20, 50MHz) over 10Mbps Ethernet, depending on the type of data and ORBs [Hennings1998].

### **6.1.2 GIOP Implications**

When comparing CORBA performance to TCP sockets, we noticed that most TCP implementations included flow control and congestion avoidance mechanisms, which reduced obtainable bandwidth well below what the underlying network will support. This was particularly true for short connections, which suffered penalties for connection setup, slow-start, and tear-down. Performance-intensive applications on the Internet, such as online games, frequently turned to UDP. One of the H.323 rivals, SIP, could be carried on either TCP or UDP. UDP was capable of achieving latency and throughput values near to those of the underlying network. Unfortunately, the GIOP specification prohibited the use of UDP as an underlying invocation protocol, since UDP is unreliable and connectionless. There had been some discussion about a GIOP mapping to UDP on the OMG's CORBA newsgroup, and some ORBs offered proprietary protocols based on UDP [Mico1999], however no standard implementation or connectionless version of GIOP existed.

CDR and (more significantly) GIOP introduce increased message overhead. If platforms share the same byte-ordering, then byte swapping is not an issue. But CDR may also introduce padding into data structures to maintain alignment. Padding can improve performance for platforms whose memory is aligned on natural boundaries, since messages can be copied directly to memory. But padding can waste bandwidth if it must be included in each member of a long sequence. The GIOP header also add the size of message. The header begins with a 12-byte field providing version and message type information. The rest of the header is variable length and possibly quite long, depending on the type of message. Request headers, for example, include the full name of the operation being invoked. The comparison project found that the length of an operation's name could significantly affect performance [DSRG2000].

CORBA one-way operations could have only input arguments, must be returned void, and could not raise any user-defined exceptions. The CORBA standard even stated that one-way operations are only best-effort, not reliable. Intuitively, one-way operations indicated that the client wished to send a message to the server and then forgot about it. UDP would seem to have been a logical way to implement one-way operations, but because GIOP required a reliable protocol, one-way operations were typically implemented using TCP (just like all other operations). Not only did using TCP slow down the delivery of the message, but it also required the sender to be blocked for the whole TCP process of connection startup, message delivery, and shutdown. Declaring an operation as one-way was, therefore, not likely to improve much of the performance.

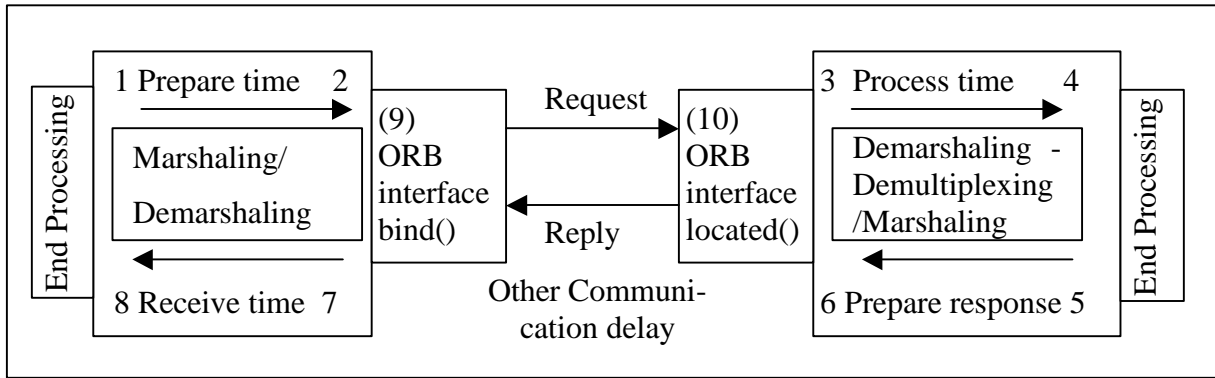
Currently, a potential workaround to enable CORBA applications to use UDP would be to develop a UDP sender/receiver combination. CORBA could be used for all operations

needing reliability (including sender and receiver set up), while the UDP sender/receiver combination could be used for rapid, best-effort communication. While this solution would require porting the UDP connector to each target platform, it would allow even performance-intensive applications to take advantage of the facilities provided by CORBA. This was exactly the scenario we experimented within section 5.7 for protocol integration.

### 6.1.3 Setting Interceptors in Message Sequence

The CORBA interceptor is interposed in the invocation (and response) paths between a client and a target object. Visibroker interceptors work directly at the protocol layer and provide a convenient way to track CORBA message sequences. Interceptors could be defined at two levels: the request level and the message level. Request-level interceptors are given access to a request object corresponding to the current request and are able to access and modify this request before and after it is invoked. Message-level interceptors have access to the actual message buffers before and after the messages are sent across the network. The Visibroker time interceptor is a service application based on three interceptor interfaces, i.e. `BindInterceptor`, `ClientInterceptor`, and `ServerInterceptor`. There are 10 points at which time interceptors could be invoked during the processing of CORBA message, as shown in Figure 6.1. The time periods of Point 1-2 and 5-6 indicate the time for data marshaling. The time period of Point 7-8 on the client side indicates the time for data demarshaling. The time period of Point 3-4 on the server side indicates the time for data marshaling and request demultiplexing. The request is dispatched based on the object reference and the operation name from object adaptors to servants. Phase 9 and 10 are time periods for ORB binding and locating.





**Figure 6.1 Points of Interceptors in Two-way CORBA Message Sequence**

Sample outputs from our test cases are shown as Listing 6.1 for two-way invocation. The other communication delay covers the time costs on delay between point 2 and 3 or between point 6 and 7 without ORB binding/locating; the end processing covers the time costs on client request, server implementation processing. (ms = millisecond)

**(Calling Endpoint)**

bind: 340 ms  
 transferIndication: (prepare time) - 40 ms  
 transferIndication: (send time) - 0 ms  
 transferIndication: (receive time) - 100 ms  
 transferIndication: (total call) - 140 ms      CESE Outgoing: 480ms

**(Called Endpoint)**

locate: 10 ms  
 transferIndication: (process time) - 50 ms  
 transferIndication: (prepare response) - 10 ms  
 transferIndication: (send time) - 0 ms  
 transferIndication: (total call) - 60 ms      CESE Incoming : 70ms  
 Total Process (Including other communication delay, end processing): 760ms

**Listing 6.1 Sample Output for Tracking Two-way CORBA Message Sequence through Visibroker Interceptor**

In our experiments, we intended to use standard CORBA interceptors. However, recent

research indicated that the existence of interceptors could slow down the overall performance of the ORB, and suggested that ORBs be equipped with a set of lightweight measurement upcalls that would signal when a particular request passed a prime point [DSRG2000]. Our further test results also showed those measurements based on interceptors were very inaccurate. Therefore, our tests were conducted through system calls without the involvement of interceptors.

#### 6.1.4 ORB Benchmarks

Over the years, both the CORBA standard and the implementations of CORBA evolved considerably. Various vendors offered a large range of C++ and Java ORBs differed in many aspects. OMG Benchmarking Platform Special Interest Group (PSIG)'s Request for Information (RFI) [OMGBench1998] and its replies such as the one from CORBA comparison project [DSRG2000] provided guidance for appropriate measurements of CORBA distributed systems. The objectives of CORBA benchmarking were to help ORB users to evaluate an ORB implementation, and to evaluate a set of ORBs using the criteria as follows:

?? Standard Functionality, such as adherence to the IDL specification, basic remote invocation functionality, interoperability, etc.

?? Nonstandard Extensions, such as:

- Communication Extensions, like locating objects, binding to objects, instantiating implementations
- Multi-threading Extensions, like multi-threaded servers (single thread, thread per request, pool of threads, thread per client, thread per object), multi-threaded clients

(non-blocking call, call-back receive), multi-thread ORB and concurrency

?? Scalability in three aspects:

- speed with respect to number of objects
- resource consumption with respect to number of objects
- resource consumption with respect to the number of incoming (asynchronous) request

?? Robustness:

- the support for building reliable servers
- the limitation on the data packet size in a single request
- the maximum number of objects with which the server and the client are able to cope

The comparison project developed various test scenarios for the above ORB benchmarking criteria and the evaluation for object services. So far, the on-going project covers following ORBs: omniORB 2.5.0/2.7.1/3.0.0 B2, Orbix 2.2&2.3/3.0, Visibroker 3.0, ORBacus 3.1.2/4.0.1, TAO 1.1.3. Since our CORBA-based interface approach did not bind to specific vendor's ORB, information published on the comparison reports and some vendors' performance tests [OMEX2000, JacORB2000] will have reference value for future product development.

## 6.2 Experimental Strategy and Test Environment

Our experiments focused on two parts: Visibroker for Java benchmark and H.245 signaling performance. The benchmark tests were expected to indicate the some key message effects to the time. The H.245 signaling tests were more specific to H.245 messages. In the

benchmark part, we used the strategies mentioned in the comparison project for the standard functionality and the communication extensions. In cases for large amount data transfer, Java socket and C socket were tested along with Visibroker to show the network or CPU bound. The marshaling tests focused on Visibroker only. The marshaling tests were run 5000 loops without the bootstrapping influences. Since the tests fetched system current time in millisecond, this looping approach provided a simple way to get more accurate and reliable results, to factor out the disruptions from the measurements, hardware interrupts and scheduling interrupts in the Windows environment.

The H.245 latency test strategy was used to analyze the bottleneck causes of the delay, check the impact of message type and size, as well as the number of requests. The layer complexity and the size of an H.245 message varied because of the optional fields. For example, except for the IP/TCP/TPKT header, the layer of TerminalCapabilitySet message varied from 1 to 5 levels of Sequence/Choice, and the sizes ranged from 14 octets to around 100 octets depending on the optional fields. The result was also compared with the signaling latency of PER encoded H.245 sample messages. All tests were run 50 times to get the average time result.

The test environment for both parts is listed as follows:

- ?? CPU: Two Pentium II 350MHz, 512KB cache PC with 128MB RAM each, for client and server program respectively
- ?? Operating System: Windows NT Workstation 4.0 with SP5
- ?? Network: 100Mbps Ethernet using 3Com Fast Etherlink Adapter
- ?? Compiler: Sun JDK 1.2.2

?? Java VM: Just-in-Time (JIT) from Microsoft/Sun

?? Middleware: Visibroker for Java 3.3 (bundled with JBuilder 3 from Inprise)

## 6.3 Performance Results

### 6.3.1 Benchmark Test Results

The following object operations were evaluated by Visibroker for Java. The results highlighted the performance aspects when applications were developed based on Visibroker.

?? Remote Object Connection

A client connected to a remote object that was registered on a server program. The average time was 1.8 ms (milliseconds) for the first trip. (The result will be compared later with marshaling costs.)

?? Remote Object Creation

After a client connected to an existing remote object, the existing remote objects generated a new object, then returned it as a remote object reference to the client. The average time was 6.3 ms. During this process, an osagent searched the remote machine where the remote object existed by using a registry list on the network. This list kept the IOR information about server objects for client lookup.

?? Remote Method Call

A remote method received integers as arguments, and returned an integer.

```
interface ArgTransfer {  
    long methodA1(in long a1);  
    long methodA2(in long a1, in long a2);
```

```

long methodA3(in long a1, in long a2, in long a3);
long methodA4(in long a1, in long a2, in long a3, in long a4);
...
};

```

### Listing 6.2 Remote Method Call Test IDL

Here, we added tests for Java socket, which used the Java socket library and `DataOutputStream` class, and for C socket, which used the Windows C socket library. The buffer size was 32KB. As shown in Table 6.1 for 3 argument tests, the C socket was the fastest, followed with the Java socket, and the Visibroker was the slowest.

**Table 6.1 Remote Method Call Time for 3 Arguments**

Visibroker	Java Socket	C Socket
0.73 ms	0.32 ms	0.28 ms

Our tests also showed that increasing the number of arguments does not affect the delay in Visibroker. The number of arguments can be increased up to 100, without a delay difference.

### ?? Numerical Data Transfer

The following tests compared the performance of Visibroker to that of sockets for numerical data transfer. Different array size of byte, int and double were transferred to test the relation between transfer bandwidth and message size. The results are given in Table 6.2, 6.3, 6.4 with matching Figure 6.2, 6.3, 6.4 to roughly illustrate the corresponding network or CPU bound.

```

interface NumercialTransfer {
    typedef sequence<octet> ByteArray;

```

```

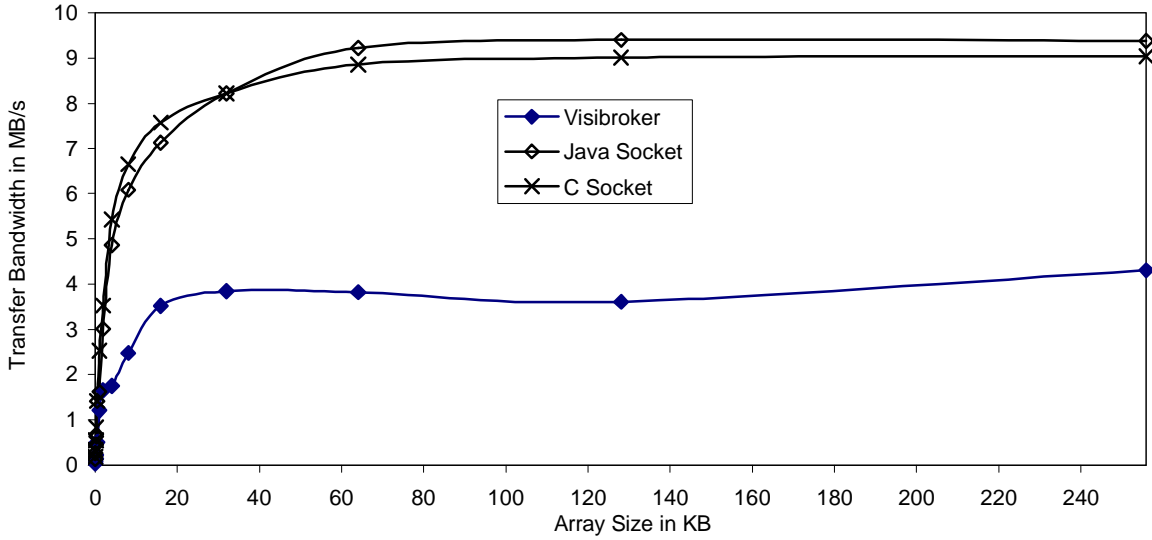
typedef sequence<long> IntArray;
typedef sequence<double> DoubleArray;
void methodByte (in ByteArray ba);
void methodInt (in IntArray ia);
void methodDouble (in DoubleArray da);
};
    
```

**Listing 6.3 Numerical Data Transfer Test IDL**

- byte array transfer, array size from 0.3125KB to 256KB

**Table 6.2 Byte Array Transfer Bandwidth**

	Compare Items	Visibroker	Java Socket	C Socket
Msg. Size (KB)	Transfer Bandwidth ( MB/s)			
0.03125		0.04	0.13	0.18
0.0625		0.08	0.26	0.33
0.125		0.14	0.45	0.54
0.25		0.22	0.63	0.82
0.5		0.51	1.42	1.42
1		1.22	1.62	2.52
2		1.65	3.01	3.53
4		1.75	4.86	5.43
8		2.48	6.08	6.65
16		3.52	7.12	7.57
32		3.85	8.22	8.22
64		3.82	9.22	8.86
128		3.61	9.40	9.01
256		4.31	9.38	9.03
			reaching network bound of 12.5MB/s ideally provided by 100BaseT Ethernet	



**Figure 6.2 Performance Comparison for Byte Array Transfer Bandwidth**

- int array transfer, array size from 0.125KB to 1024KB

**Table 6.3 Int Array Transfer Bandwidth**

	Compare Items	Visibroker	Java Socket	C Socket
Msg. Size (KB)	Transfer Bandwidth (MB/s)			
0.125		0.16	0.41	1.02
0.25		0.32	0.65	1.42
0.5		0.54	0.83	1.84
1		0.98	1.01	2.20
2		1.25	1.45	2.86
4		1.42	1.56	4.36
8		1.49	1.74	5.32
16		1.69	1.79	6.06
32		1.73	1.86	6.76
64		1.79	2.45	7.22
128		2.02	2.78	7.35
256		1.86	3.06	7.25

reaching CPU bound as compared to C Socket which is still increasing



512	1.90		3.23	6.82
1024	1.82		3.32	6.78

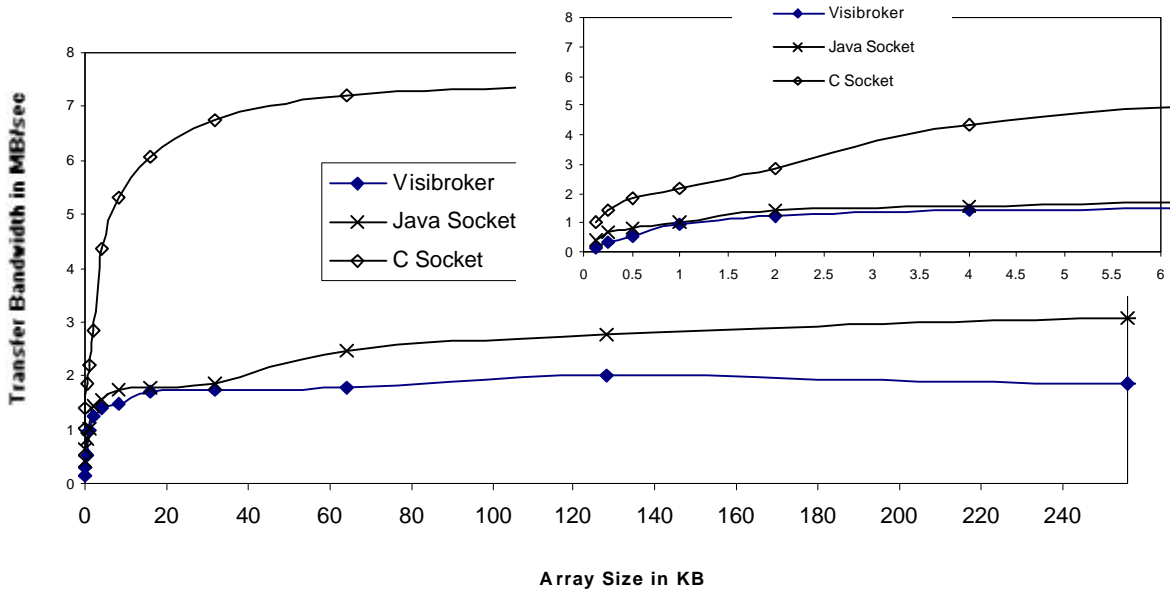


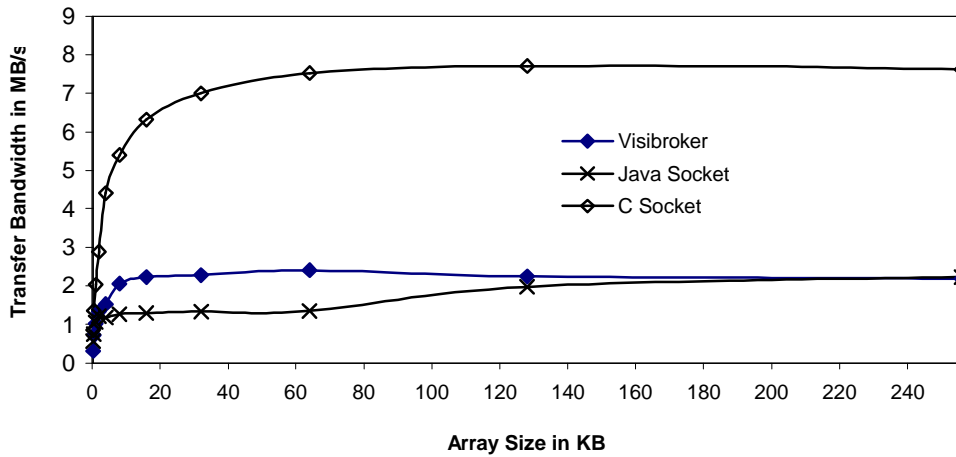
Figure 6.3 Performance Comparison for Int Array Transfer Bandwidth

- double array transfer, array size from 0.25KB to 2048KB

Table 6.4 Double Array Transfer Bandwidth

	Compare Items	Visibroker	Java Socket	C Socket
Msg. Size (KB)	Transfer Bandwidth (MB/s)			
0.25		0.31	0.56	0.85
0.5		0.72	0.75	1.36
1		1.01	1.05	2.03
2		1.34	1.22	2.88
4		1.52	1.18	4.40

8	2.06	reaching CPU bound as compared to C Socket which is still increasing	1.27	5.40
16	2.23		1.29	6.32
32	2.28		1.33	7.00
64	2.40		1.35	7.52
128	2.24		1.98	7.70
256	2.19		2.23	7.61
512	2.06		2.56	7.01
1024	1.96		2.78	6.86
2048	1.93		2.95	6.93



**Figure 6.4 Performance Comparison for Double Array Transfer Bandwidth**

The result showed that byte array transfer had a similar transfer rate for both C and Java socket. When the data amount were added, both can reached the maximum data transfer rate near the 100Mbit/s of the network bound. For int and double array transfer, the Java socket was far worse than that of C socket. The reason was the high cost of byte reordering and data copy in cases for Java, which led to the CPU being bound for medium array size (4KB to 1MB). Visibroker for Java’s byte array transfer had a better performance result compared to its int/double array transfer, which also had the CPU being bound for medium array transfer.

For small amount data transfer that is size less than 1KB, in all cases, the bandwidth followed the increase of data size. The point to be mentioned here is that H.245 messages were normally less than 1KB. So, in terms of message size, the increase of message size followed the increase of the transfer bandwidth. This will lighten the effect of message size on delays.

### ?? Visibroker Marshaling Test for Primitive and Complex Data Types

The following tests were conducted on Visibroker only, with the IDL as Listing 6.4.

```
interface Marshal {
    enum DateEnum {
        Mon, Tue, ...
    };
    union DateUnion switch (DateEnum){
        case Mon: boolean ub;
        case Tue: char uc;
        ...
    };
    typedef sequence<boolean> SeqBoolean;
    typedef sequence<char> SeqChar;
    typedef sequence<double> SeqDouble;
    typedef sequence<float> SeqFloat;
    typedef sequence<long> SeqLong;
    typedef sequence<octet> SeqOctet;
    typedef sequence<short> SeqShort;
    typedef sequence<string> SeqString;
    struct Octboo_T1 {
        octet octetVal;
        boolean booleanVal;
    };
    typedef string<5> BString_T1;
    typedef sequence<float,5> BSeqFloat_T2;
    typedef sequence<Octboo_T1,5> BSeqOctboo_T2;
    typedef sequence<BString_T,5> BSeqBString_T2;
    typedef sequence<BSeqFloat_T2,5> BSeqBSeqFloat_T3;
    typedef sequence<BSeqOctboo_T2,5> BSeqBSeqOctboo_T3;
```

```

typedef sequence<BSeqBString_T2,5> BSeqBSeqBString_T3;
// test 1
void nullCall();
// test 2
void sendPrimitives(in boolean b, in char c, in double d, in float f, in long l, in octet o, in short s);
// test 3
void sendString(in string str);
// test 4
void sendUnion(in DateUnion u);
// test 5
void sendSeqPrimitives(in SeqBoolean sb, in SeqChar sc, in SeqDouble sd,
    in SeqFloat sf, in SeqLong sl, in SeqOctet so, in SeqShort ss);
// test 6
void sendSeqStrings(in SeqString sstr);
// test 7
void sendSeqSeqs (
    in BSeqBSeqBString_T3 ssstr, in BSeqBSeqOctboo_T3 ssoctboo, in BSeqBSeqFloat_T3 ssf);
...
};

```

#### Listing 6.4 Marshaling Test IDL Example

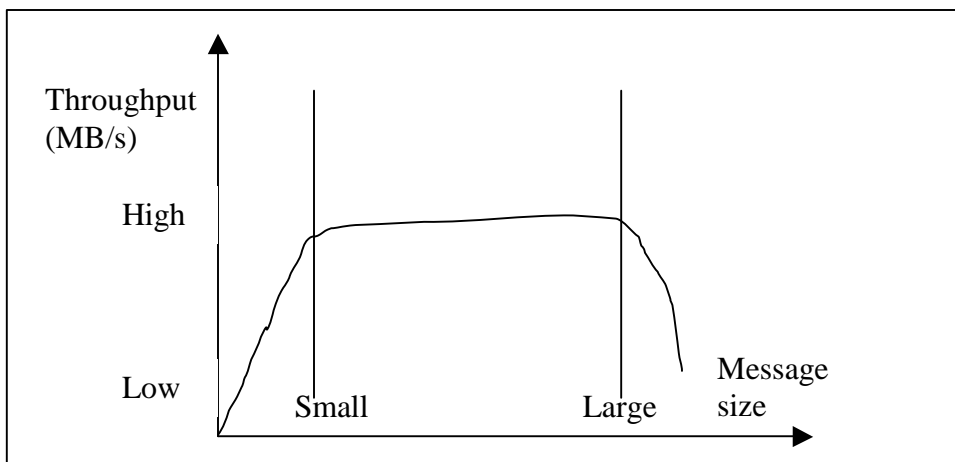
Test case test1 represented a "null call". Test cases test2, test3, and test4 marshaled primitives (arguments being passed together or separately), string (length=80), and union types. Test cases test5 and test6 marshaled sequences of primitives (length=10), strings (length=20). Test case test7 marshaled nested sequences. We also varied the number of sequence size during the test. Table 6.5 indicated the average execution time of the invocation in 5000 loops without the bootstrapping.

**Table 6.5 Marshaling Test for Round Trip Times**

	Test1	Test2	Test3	Test4	Test5	Test6	Test7
Average time in 5000 loops (ms)	0.725	0.738	0.744	0.745	0.822	1.030	2.730

Compared with the previous results for bootstrapping cost, such as opening a connection, locating and creating an object, it was shown that there was a significant difference between the time it took to invoke an operation for the first time and for all subsequent times. The marshaling costs for passing primitive data types were slightly different, and the variation was about 10%. Time to pass a sequence or an array of a basic IDL data type depended mostly on the length of data in octets. Unless a very large number of complex arguments were passed, the constant overhead of an invocation overshadowed the impact of argument sizes and types. Hence, it paid off to call less often with more arguments than vice versa.

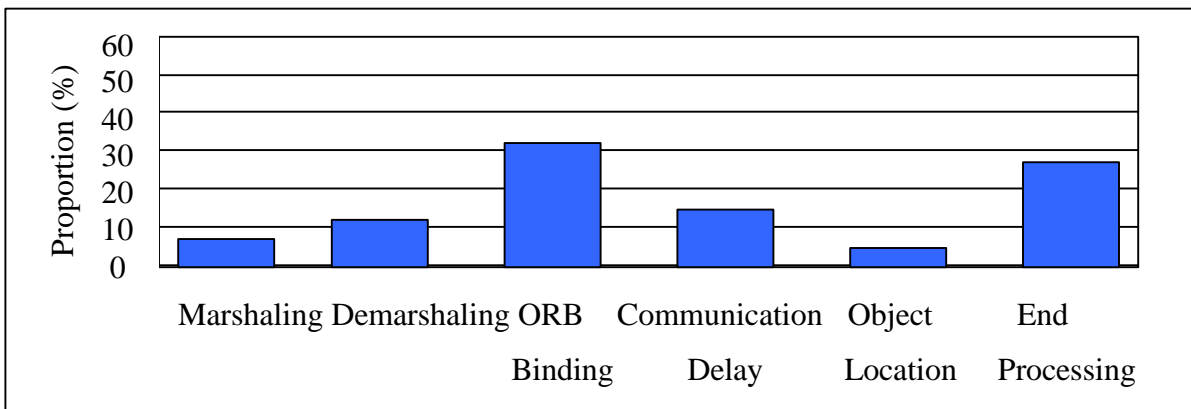
As roughly illustrated in Figure 6.5, a high throughput rate can be usually be achieved by using few requests, each transferring a large amount of data, instead of many requests, each transferring a small amount of data. However, when transferring an extremely large amount of data in a single message, the throughput rate decreased.



**Figure 6.5 Dependency between Data Throughput and Message Size**

### 6.3.2 H.245 Signaling Test Results

The H.245 signaling tests were kept in the same test environment and followed the similar test strategies as the ones used in previous benchmark tests. The sample result of our H.245 call latency test is shown as Figure 6.6, which presents the average distribution of the time in a CORBA invocation under the cases of H.245 signaling procedures, such as capability exchange, master slave determination, logical channel signaling.



**Figure 6.6 Average Distribution of Time in a Sample CORBA Invocation**

#### ?? Bootstrapping Costs

A large portion of time was spent on ORB binding, object locating, communication delay and object implementation processing, totally a count of more than 75%. There were several reasons for this large overhead. First of all, the osagent had to search the registry list to find the right object to invoke. The intra-ORB communications between osagent might also introduce extra time overhead. Reducing the number of osagents in the network could speed up the communication. The binding operation is not part of CORBA standard. It is a Visibroker's proprietary extension allowing dynamic communication to the Visibroker for Java directory service. Different ORB implementations might give significant different

results. Second, starting the server object (like initial `object_to_string()`) took the extra first time cost. Other than those reasons, Object activation daemon, which functioned as an implementation repository, was relatively time-consuming for the first time object activation, but reached a much better performance in later activation.

#### ?? Marshaling/Demarshaling Costs

The marshaling/demarshaling engines in the ORB performed data conversion between the native format used in the client/server implementation and the CORBA CDR format. For data structures, the sending ORB must collect the data from different locations in memory and copy it into a transmit buffer. Then, later on, the receiving ORB must call the traversal routine to parse the structure based on whatever definition for the structure it has. The marshaling/demarshaling costs, which count for about 15% - 25% of the total time costs, weighed as another important factor in call latency. The differences of H.245 message size due to various optional fields were not the critical factors affecting the performance.

#### ?? CDR verses PER

An early report in 1993 showed that PER could provide a relatively small size of encoded message and up to the speed of 12Mbps for data encoding and decoding, as compared to that of hand coded C en/decoder. In their tests, the C en/decoder had a relatively large size of encoded message (more than two times) and an even faster speed of 240 Mbps for en/decoding (Tests run on a MIPS R3260, 48MB RAM, Unix OS 4.51) [Sample1993]. In order to get comparable result for the signaling of H245 messages, we set up a separate experiment based on a commercial ASN.1 compiler and PER encoder/decoder (Java) from OSS Nokalva [Nokalva2000]. The company also provided an ASN.1 PER encode/decode tool for C/C++, and the related open source information was available from the OpenH323

project [OpenH3232000]. The result in Table 6.6 shows the times for transferring message of Terminal Capability Set as Listing 5.1 described. Obviously, PER provided a much faster encoding/decoding speed than CDR, but this just shown what a typical implementation could be achieved when using commercial off the shelf tools. Developers can always do better by doing custom optimizations or by taking advantage of newer state of the art technologies when they become available, such as much faster processors.

**Table 6.6 PER and CDR Marshaling Tests for H.245 TCS Message Sample**

5000 loop average	Mashaling(Encoding)	Demashaling(Decoding)
PER (ms)	0.445	0.654
CDR (ms)	1.256	1.345

## 6.4 Performance Conclusions

Sum up the test results, the latency of H.323/H.245 signaling is determined by a number of factors, such as ORB binding and message marshaling/demarshaling costs. The ORB binding time is an important, but proprietary factor, which might be changed for other vendor products. The sizes or the types of the complex messages affect the marshaling/demarshaling costs, but they are not in the range that may cause network or CPU resource problems. Reducing the number of remote calls with composed messages to carry on enhanced functionality may give a better system performance. The first time bootstrapping has a longer overhead than the rest of the times for invocations. The performance is also influenced by the implementation language, target machine (underlying instruction set architecture) and optimization of approaches.



## Chapter 7

### Conclusions and Future Work

---

#### 7.1 Conclusions

We viewed this project as an exploring and learning process. This CORBA-based interface-centric signaling approach was positioned to provide a vision for future telephony services. We proved that CORBA clients with limited resources can provide all the H.323-like end system capabilities. We found this approach:

- ?? Made it easier for program developers to express the service capability on a distributed computing environment as it defines the message and operation in common accessible IDL interfaces. It had the advantage not only of all the inherent distribution transparencies, but also the simplicity that came from using a single messaging protocol, IIOP.
- ?? Using CORBA as a network wrapper eliminated a lot of low-level (and traditionally error-prone) coding tasks such as parsing typed data and performing byte-order conversions, and facilitated interoperability between different platforms and programming languages. CORBA also made it easy to move the locations of application components around within the network, without altering any code.
- ?? Using various CORBA services helped programmers to achieve advanced functionality in the implementation.

?? The H.245 message with complex data structure in CORBA requests did have an impact on call set-up delays, although the amount was less important compared to costs in ORB binding, and end processing. For distributed applications where bandwidth and latency requirements were well below what was available at the transport layer, CORBA was seen as an obvious choice as the development architecture.

## **7.2 Future Work**

The work of this thesis offered a number of research opportunities:

?? While comprehensive in scope, H.323 has been faulted for being too bulky, both in terms of documentation as well as in the complex interplay of several protocols. This could lead to more than 15 signaling messages for single point-to-point call set-up. Its competitor, SIP, while equally comprehensive, offered a simpler alternative by being “Internet ready” when introducing telephony value-added services. Future research may be directed to implementing our CORBA-based interface-centric approach in SIP.

?? CORBA provided a set of high-level, reusable services, which potentially save a great deal of development time. Besides the ones mentioned in our approach, other services such as notification service, persistency object service, object transaction service, concurrency services are worth exploring in the telecommunication environment.

?? As with IP telephony, wireless Internet is another fast growing industrial sector with similar concerns and challenges. Further research could test our interface-centric approach on hand-held devices through a set of Wireless Application Protocols (WAP), which uses ASN.1-based binary XML (eXtensible Markup Language) for narrow-band communications [WAPForum2000].

?? One of the advantages offered by CORBA was the application's scalability, which was recognized as a primary factor to be considered in the design of distributed system. Further research on CORBA's usage in IP telephony services could be conducted on factors affecting scalability, such as multithreading offered by ORB core, demultiplexing offered by object adapter and implementation repository.

## REFERENCES

1. [Ahmad1999] Ahmad, I, Majumdar, S., “Achieving High Performance in CORBA-based Systems with Limited Heterogeneity”, under review, 1999
2. [ASNHome1997] ASN.1 Home, <http://www-sop.inria.fr/rodeo/personnel/hoschka/asn1.html>
3. [ASNResource2000] ASN.1 Resource Links, <http://asn1.elibel.tm.fr/en/links/index.htm#tools>
4. [Balabanian1996] Balabanian, V., et al., “An Introduction to DSM-CC (Digital Storage Media – Command and Control)”, IEEE Communications Magazine, pp122-127, November 1996
5. [Beddus2000] Beddus, S., et al., “Opening Up Networks with JAIN Parlay”, IEEE Communications Magazine, 136-143, April 2000
6. [BellLabs1998] Bell Labs Translation Tools, <http://nsm.research.bell-labs.com/~mazum/CorbaSnmp/register.html>
7. [Berg1998] Berg, H. A., Brennan, S., “CORBA and Intelligent Network (IN) Internetworking”, IS&N’98, pp463-475, 1998
8. [Biswas1999] Biswas, J., et al., “White Paper on Application Programming Interfaces for Networks”, IEEE P1520, January 1999
9. [DAVIC1998] DAVIC 1.4 Specification, 1998
10. [DSRG2000] Distributed Systems Research Group, Charles University, “CORBA Comparison Project”, <http://nenya.ms.mff.cuni.cz/thegroup/COMP/>, 2000
11. [Dubuisson2000] Dubuisson, O., “ASN.1 – Communication between Heterogeneous Systems”, Academic Press, 2000
12. [Fischbeck1999] Fischbeck, N., Kath, Kath, O., “CORBA Internetworking over SS.7”, IS&N’99, pp101-113, 1999
13. [Gokhale1998] Gokhale, A. S., Schmidt, D.C., “Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks”, IEEE Transaction on Computers, pp391-413, April 1998
14. [Hamdi1999] Hamdi, M., et al., “Voice Service Internetworking for PSTN and IP Networks”, IEEE Communications Magazine, pp104-111, May 1999
15. [Henning1998] Henning, M., Vinoski, S., “Advanced CORBA Programming with C++”, AWL, 1998
16. [Inprise1999] Inprise, “Visibroker for Java 3.3/4.0 Programmer’s Guide”, 1998/1999
17. [InprisePress2000] Inprise, <http://www.inprise.nl/about/press/2000/ericsson.html>, 2000
18. [ITU1995] ITU RM-ODP, “Reference Model of Open Distributed Processing”, 1995-2000, <http://www.dstc.edu.au/Research/Projects/ODP/standards.html>

19. [ITU1996a] ITU-T Rec. H.323, “Visual Telephone Systems and Terminal Equipment for Local Area Networks which Provide a Non-Guaranteed Quality of Service”, May 1996
20. [ITU1996b] ITU-T Rec. H.225, “Line Transmission of Non-Telephone Signals”, May 1996
21. [ITU1996c] ITU-T Rec. H.245, “Control Protocol for Multimedia Communication – Line Transmission of Non-Telephone Signals”, June 1996
22. [JacORB2000] JacORB, <http://jacorb.inf.fu-berlin.de/>, 2000
23. [Jepsen2000] Jepsen, T., “Java and Telecommunications in the New Millennium”, IEEE Communications Magazine, January 2000
24. [JIDM1997] Joint Inter-Domain Management (JIDM): Specification Translation, <http://www.opengroup.org/onlinepubs/8349099/toc.htm>, 1997
25. [Kaliski1993] Kaliski, B. S., “A Layman’s Guide to a Subset of ASN.1, BER, and DER”, <ftp://ftp.rsa.com/pub/pkcs/ascii/layman.asc>, An RSA Laboratories Technical Note, November 1, 1993
26. [Lu1999] Lu, T., Pagurek, B., “Report on the Comparison of Multimedia Control Specification”, Project Report, October 1999
27. [Lu2000] Lu, T., Pagurek, B., “H.323 Signaling Using CORBA – Implementing H.245 Control Protocol for Multimedia Communication”, Project Report, May 2000
28. [Mico1999] Kay Roemer, <http://www.cs.uni-magdeburg.de/~aschultz/mico/mico-announce/msg00000.html>, 1999
29. [Mitra1999a] Mitra, N., “A Long-Term Approach to Signaling for IP-based Services”, Ericsson Internal, April 1999
30. [Mitra1999b] Mitra, N., Glitho, R., “Research Proposal for: H323 Signaling Using CORBA”, Ericsson Internal, April 1999
31. [Mitra1999c] Mitra, N., Brennan, R., “Design of the CORBA/TC Inter-working Gateway”, IS&N’99, pp84-100, 1999
32. [Munjee1999] Munjee, S., Surendran, N., Schmidt, D. C., “The Design and Performance of a CORBA Audio/Video Streaming Service”, HICSS-32 Int’l Conference on System Sciences, 14pp, January 1999
33. [Nexus1999] Nexus Translation Tools, <http://ain.kyungpook.ac.kr/nexus/download.html>
34. [Nokalva2000] Nokalva, OSS ASN.1 Tools for Java (PER), <http://www.oss.com/products/asn1java/javatools.html>, 2000
35. [OMEX2000] OMEX benchmark tests for performance, scalability, interoperability and conformance, for Orbix2.3, Orbix 3, Visibroker, M3, OmniORB, Mico, <http://www.omex.ch/CorbaTB/corbatb.htm>
36. [OMG1998] OMG Specification, “CORBA Services: Common Object Services Specification”, 1998, <http://www.omg.org/library/csindx.html>

37. [OMG2000] OMG Specification, “CORBA: Common Object Request Broker Architecture and Specification”, v 2.4, OMG, October 2000, <http://www.omg.org/library/c2indx.html>
38. [OMGBench1998] OMG PSIG, “ORBOS Platform Task Force Benchmark RFI”, May 1998
39. [OMGTC1998a] OMG TC Document, “CORBA Messaging”, May 1998
40. [OMGTC1998b] OMG Telecom Specification, “Notification Service”, June 1998
41. [OMGTelecom1998a] OMG Telecom Specification, “Control and Management of Audio/Video Streams”, June 1998
42. [OMGTelecom1998b] OMG Telecom Specification, “Interworking between CORBA and TC Systems”, October 1998
43. [OMGTelecom1998c] OMG Telecom RFP, “JIDM Interaction Translation – Final Submission to OMG’s CORBA/TMN Interworking RFP”, October 1998
44. [OpenH3232000] The Open H323, [www.openh323.org](http://www.openh323.org), 2000
45. [Orbycom2000] Orbycom Translation Tools, <http://www.orbycom.fr/translators.html>
46. [Parlay2000] The Parlay Forum, [www.parlay.org](http://www.parlay.org), 2000
47. [Rasmussen1998] Rasmussen, S., “A CORBA to CMIP Gateway: A Marriage of Management Technologies”, IS&N, pp477-492, 1998
48. [Sample1993] Sample, M., “Implementing Efficient Encoders and Decoders for Network Data Representations”, IEEE INFOCOM, pp1144-1153, 1993
49. [Schmidt2000] Schmidt, D.C., <http://www.cs.wustl.edu/~schmidt/corba-overview.html>
50. [Schulzrinne1997] Schulzrinne, H., “A Comprehensive Multimedia Control Architecture for the Internet”, Network and Operating System Support for Digital Audio and Video, Proceedings of the IEEE 7<sup>th</sup> International Workshop on, 1997
51. [Schulzrinne1998] Schulzrinne, H., Rosenberg, J., “Signaling for Internet Telephony”, IEEE Proceedings, 6<sup>th</sup> Int’l Conference on Network Protocols, pp298-307, 1998
52. [Siegel1999] Siegel, J., “A Preview of CORBA 3”, IEEE Computer, pp114-116, May 1999
53. [SunJAIN2000] “The JAIN APIs”, Sun Microsystems, <http://www.java.sun.com/products/jain/>, 2000
54. [SunJMF1999] “Java Media Framework API Guide”, JMF 2.0 FCS, Sun Microsystems, November, 1999
55. [WAPForum2000] [www.wapforum.org](http://www.wapforum.org), 2000

## **Appendix A: Conversion of H.245 Message Syntax (ASN.1 to IDL)**

This document attached with h245.idl file for a better demonstration of the conversion.

### 1. Partial hierarchy illustration for Capability Exchange Module in h245.idl:

Level 1: h245CLIENT.idl

Level 2: tLM.idl

Level 3: cE.idl

Level 4: cETLC.idl, cEMC.idl

Level 5: cEVC.idl, cEAC.idl, cEDC.idl, cEC.idl

Level 6: nSM.idl, sNM.idl

Level 7: ASN1Types.idl

### 2. Full Module definitions:

module tLM: Top level message

module sNM: Sequence number message definition

module nSM: Non standard message definition

module mSD: Master-slave determination definition

module cE: Capability exchange definition

module cETLC: Capability exchange definitions : top level capability description

module cEMC: Capability exchange definition: Multiplex capabilities

module cEVC: capability exchange definition: Video capabilities

module cEAC: Capability exchange definition: Audio capabilities

module cEDC: Capability exchange definitions: Data capabilities

module cEC: Capability exchange definition: Conference

module ICS: Logical channel signaling definitions

module h223MT: H.223 multiplex table definitions

module rM: Request mode definitions

module rMM: Request mode definitions: Mode description

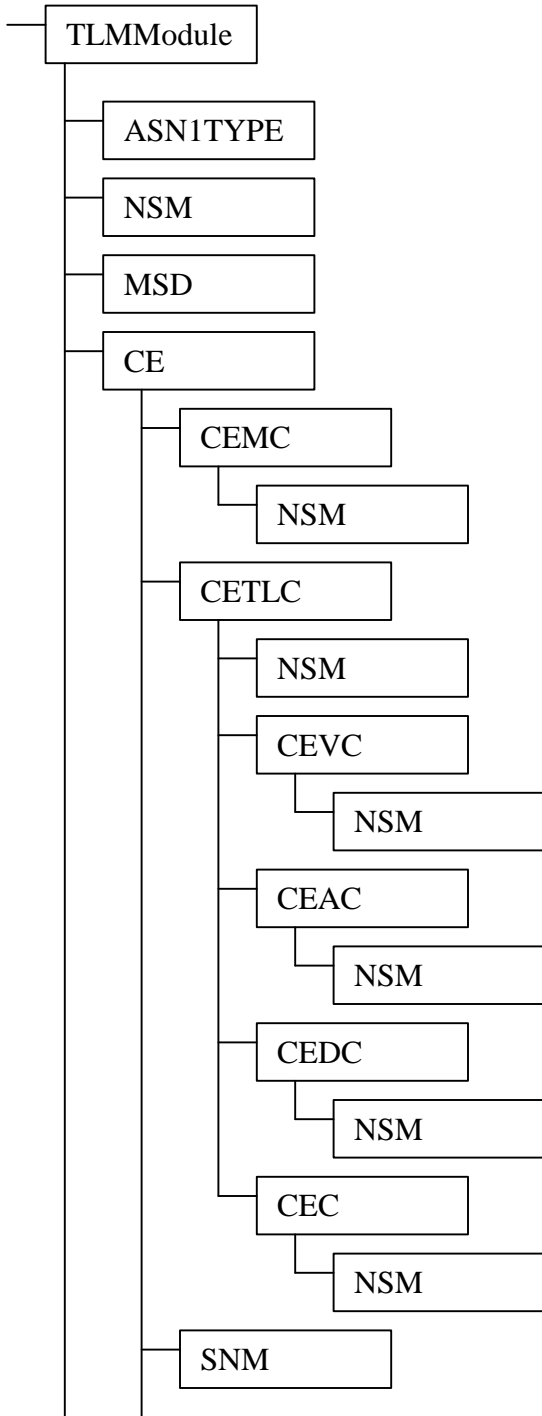
module rMV: Request mode definitions: Video modes  
module rMA: Request mode definitions: Audio modes  
module rMD: Request mode definitions: Data modes  
module rME: Request mode definitions: Encryption modes  
module rTD: Round trip delay definitions  
module mL: maintenance loop definitions  
module cCOMM: Communication mode definitions  
module cREQ: Conference request definitions  
module cRSP: Conference response definitions  
module h223ARR: H223AnnexA reconfiguration request definitions  
module h223AARR: H223 Annex A reconfiguration response definitions  
module cMSTCS: Command message: Send terminal capability set  
module cME: Command message: Encryption  
module cMFC: Command message: Flow control  
module cMCES: Command message: Change or end session  
module cMCC: Command message: Conference commands  
module cMMH230C: Command message: Miscellaneous H.230-like commands  
module iM: Indication message definitions  
module iMFNU: Indication message module: Function not understood  
module iMFNS: Indication message: Function not supported  
module iMC: Indication message: Conference  
module iMMH230I: Indication message: Miscellaneous H.230-like indication  
module iMJI: Indication message: Jitter indication  
module iMH223LCS: Indication message: H.223 logical channel skew  
module iMH225MLCS: Indication message: H.225.0 maximum logical channel skew  
module iMMCLI: Indication message: MC location indication  
module iMVI: Indication message: Vendor identification  
module iMNATMVCI: Indication message: New ATM virtual channel indication  
module iMUI: Indication message: User input

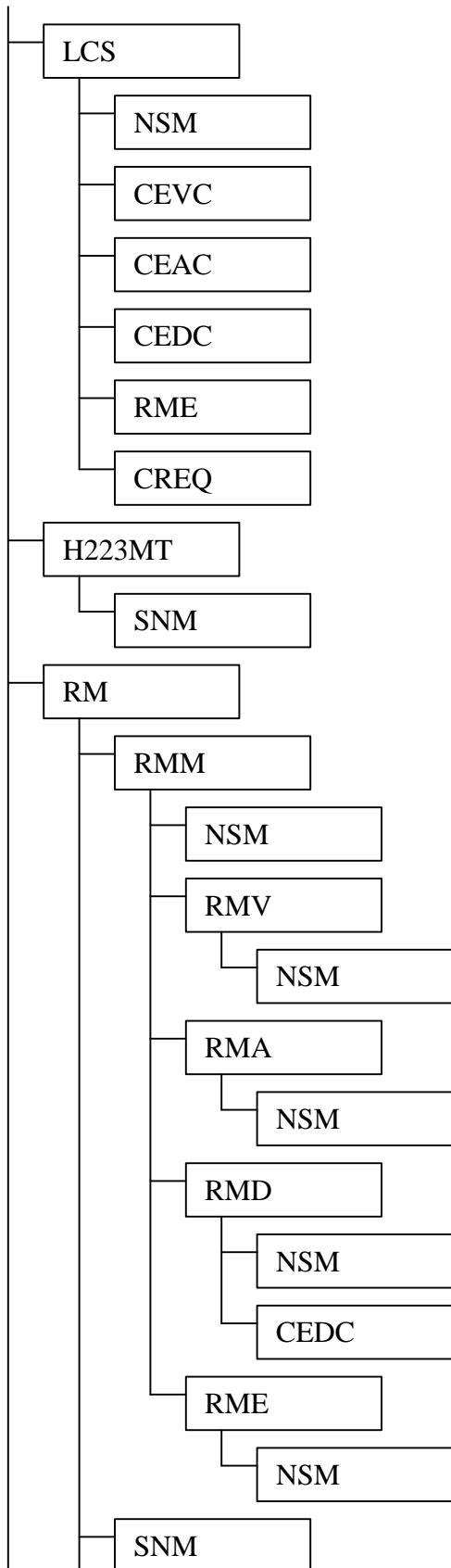


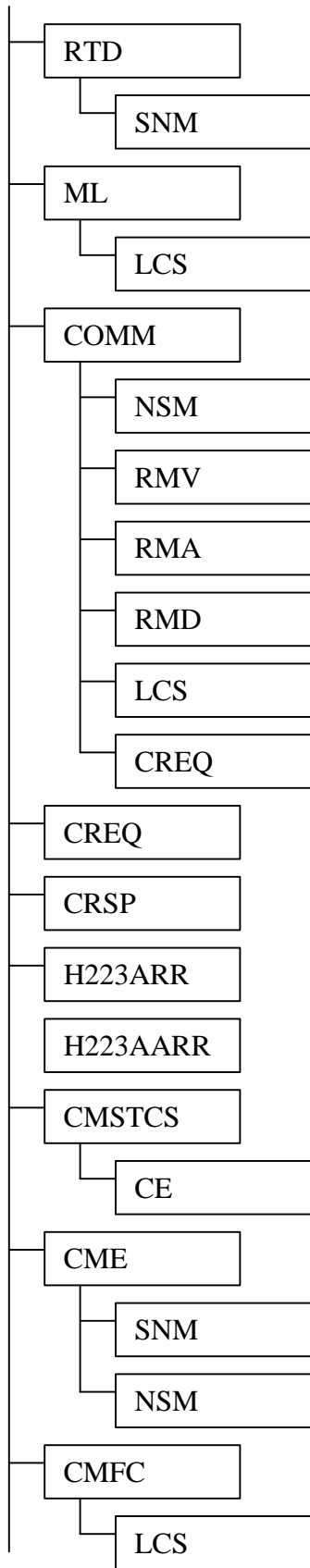
### 3. Diagram for H245 Message Modules

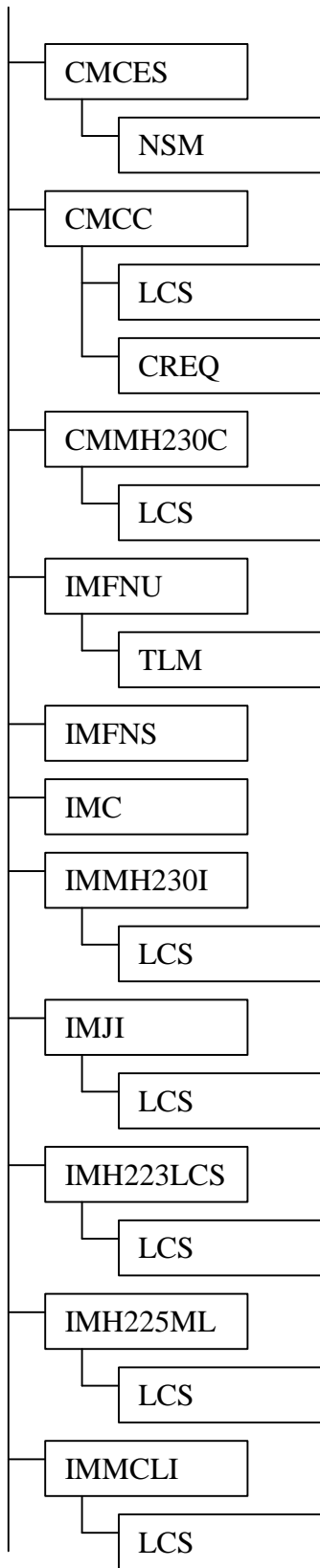
Notes:

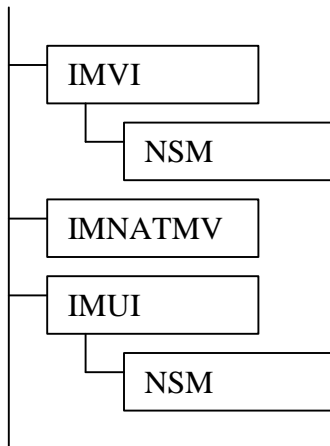
1. “—— ”: include relationship;
2. All modules include ASN1Types Module;
3. H245SignalingEntity Module working on top of TLModule with user specific interfaces based on procedure definitions











#### 4. Content of ASN1Types.idl

```

// ASN1Types.idl
#ifndef _ASN1TYPES_IDL_
#define _ASN1TYPES_IDL_
#pragma javaPackage GlobalASN1Types
#pragma prefix "GlobalASN1Types"
// ASN.1 base types
typedef octet      ASN1_Null;
typedef boolean   ASN1_Boolean;
typedef short     ASN1_Integer16;
typedef unsigned long ASN1_Integer;
typedef long      ASN1_Integer64[2];
// unsigned integers
typedef unsigned short ASN1_Unsigned16;
typedef unsigned long  ASN1_Unsigned;
typedef unsigned long  ASN1_Unsigned64[2];

typedef double      ASN1_Real;
typedef sequence<octet> ASN1_BitString; // PIDL defined
typedef sequence<octet> ASN1_OctetString;
typedef string      ASN1_ObjectIdentifier;
typedef any         ASN1_Any;
typedef any         ASN1_DefinedAny;
struct             ASN1_External {
    ASN1_ObjectIdentifier syntax;
    ASN1_DefinedAny      data_value; // by syntax

```

```
};  
// ASN.1 strings which may not contain binary zeros  
typedef string      ASN1_IA5String;  
typedef string      ASN1_NumericString;  
typedef string      ASN1_PrintableString;  
typedef string      ASN1_TeletexString;  
typedef string      ASN1_T61String;  
typedef string      ASN1_VideotexString;  
typedef string      ASN1_VisibleString;  
typedef ASN1_VisibleString ASN1_GeneralizedTime; // PIDL defined  
typedef ASN1_VisibleString ASN1_UTCTime;  
  
// ASN.1 strings which may contain binary zeros  
typedef sequence<octet> ASN1_BMPString;  
typedef sequence<octet> ASN1_GeneralString;  
typedef sequence<octet> ASN1_GraphicString;  
typedef sequence<octet> ASN1_ISO646String;  
typedef sequence<octet> ASN1_UniversalString;  
  
typedef ASN1_GraphicString ASN1_ObjectDescriptor;  
#pragma javaPackage  
#endif  
/* _ASN1TYPES_IDL_ */
```

## Appendix B: CORBA-based Interface-centric Approach Implementation for H.323 Signaling (Screen Shot)

### H.323 Gatekeeper:

Step 1. Start Visibroker Smart Agent (osagent):



When programmers develop CORBA application, all attributes and operations of the remote implementation are defined in CORBA Interface Definition Language (IDL). However, IDL does not cover any concept about where the remote object is located or how to connect to it. The CORBA specification defines how an object implementation makes itself available to start receiving invocations and how it creates a unique reference for itself, the Interoperable Object Reference (IOR). Visibroker smart agent (osagent) is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. Object implementations register their objects with the osagent so that client applications can locate and use those objects. When an object or implementation is destroyed, the osagent removes them from the list of available objects.

Step 2. Start CORBA Naming Service (H.225):

```

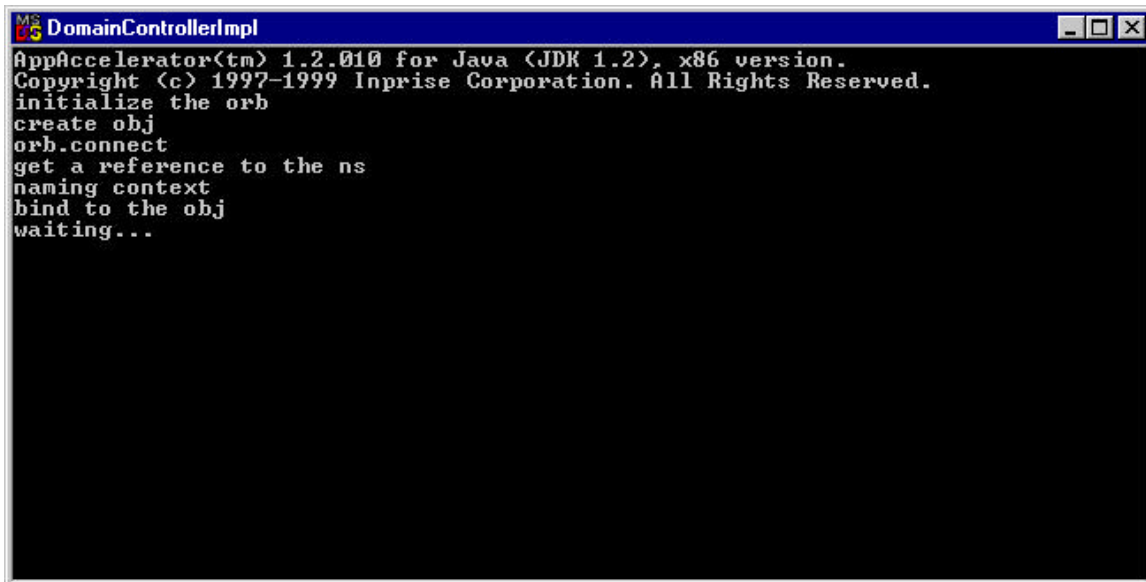
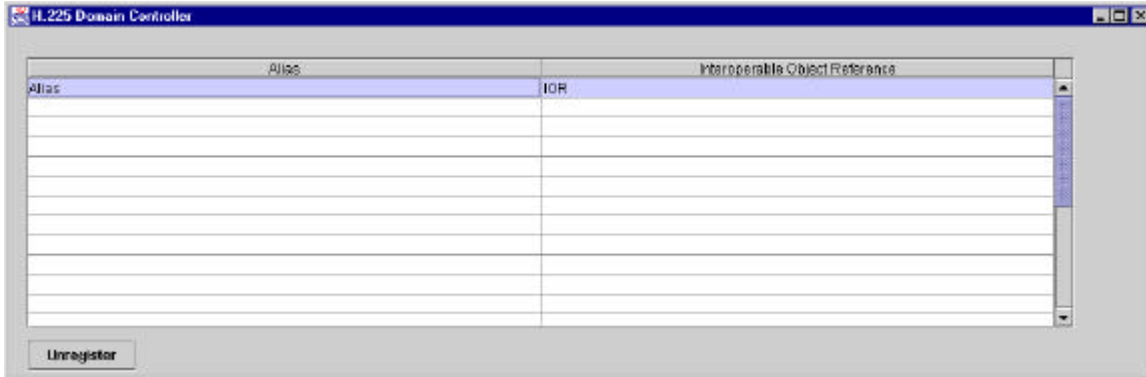
C:\WINNT\System32\CMD.exe
C:\JBuilder3\bin>vbj -DDEBUG -DORBservices=CosNaming -DSUCnameroot=NamingService
-DJDKrenameBug com.visigenic.vbroker.services.CosNaming.ExtFactory NamingService
e namingServicesLog
Extended Naming Factory
  name: NamingService
  log file: namingServicesLog
  debug: true
byte order is false
file is singly typed
typecode is LogRecord
LogRecord typecode is LogRecord
Log::replay: LogRecordOperation=new_context,context=<fact>/1]
Log::replay: LogRecordOperation=bind_context,<fact>/1,name=LName[<id=ObjectLife
timeManager,kind=>],obj=[UnboundStubDelegate,ior=struct IOR<string type_id="IDL:
se/ericsson/lmc/uu/corba/objectManager/ObjectLifetimeManager:1.0";sequence<Tagge
dProfile> profiles=<struct TaggedProfile<unsigned long tag=0;sequence<octet> pro
file_data=<124 bytes: (0)(1)(0)(0)(0)(0)(0)(15)[1][3][4][1].][1][1][7][1].][15][7][1].
[1][8][1][0)(0)(12)(156)(0)(0)(0)(0)(0)(0)(0)(1)(0)(0)[P][M][C]1(0)(0)(0)(1)(0)(0)(0)
[EB][I][D][L]:][s][e][l][e][r][i][l][c][s][s][o][n][l][m][f][c][l][u][l][f][c][o][l][r][b][
a][l][l][o][b][l][j][e][l][c][t][M][a][l][n][a][g][e][l][r][l][f][0][b][j][e][l][c][t][L][i][f][
e][l][l][m]....>];>];>]]
rebind(name=LName[<id=ObjectLifetimeManager,kind=>], type=nobject, object=[Unboun
dStubDelegate,ior=struct IOR<string type_id="IDL:se/ericsson/lmc/uu/corba/objec
tManager/ObjectLifetimeManager:1.0";sequence<TaggedProfile> profiles=<struct Tagg
edProfile<unsigned long tag=0;sequence<octet> profile_data=<124 bytes: (0)(1)(0)
(0)(0)(0)(15)[1][3][4][1].][1][1][7][1].][15][7][1].][1][8][1][0)(0)(12)(156)(0)(0)
(0)(0)(0)[\](0)[P][M][C]1(0)(0)(0)(1)(0)(0)(0)(0)[E][I][D][L]:][s][e][l][e][r][i][l][c]
[s][s][o][n][l][m][f][c][l][u][l][f][c][o][l][r][b][a][l][l][o][b][l][j][e][l][c][t][M][a][l]
[n][a][l][g][e][l][r][l][f][0][b][j][e][l][c][t][L][i][f][e][l][l][m]....>];>];>]]
Log::replay: LogRecordOperation=bind_context,context=<fact>/1,name=LName[<id=Na
mingService,kind=>],ctx=<fact>/1]
rebind(name=LName[<id=NamingService,kind=>], type=ncontext, context=com.visigeni
c.vbroker.services.CosNaming.NamingContextImpl[Server,oid=PersistentId[repId=IDL:
omg.org/CosNaming/NamingContext:1.0,objectName=NamingService/1]]
Log::replay: LogRecordOperation=bind_context,<fact>/1,name=LName[<id=DomainCont
roller,kind=>],obj=[UnboundStubDelegat,ior=struct IOR<string type_id="IDL:se/er
icsson/lmc/uu/corba/h225/gatekeeper/DomainController:1.0";sequence<TaggedProfile
> profiles=<struct TaggedProfile<unsigned long tag=0;sequence<octet> profile_dat
a=<133 bytes: (0)(1)(0)(0)(0)(0)(15)[1][3][4][1].][1][1][7][1].][15][7][1].][1][7][1][8]
(0)(0)(19)[=]1(0)(0)(0)(0)(0)[e]1(0)[P][M][C]1(0)(0)(0)(0)(0)(0)(0)[B][I][D][L]:][s][e][l][e][r][i][l][c]
[s][s][o][n][l][m][f][c][l][u][l][f][c][o][l][r][b][a][l][l][o][b][l][j][e][l][c][t][M][a][l]
[n][a][l][g][e][l][r][l][f][0][b][j][e][l][c][t][L][i][f][e][l][l][m]....>];>];>]]
rebind(name=LName[<id=DomainController,kind=>], type=nobject, object=[UnboundStu
bDelegate,ior=struct IOR<string type_id="IDL:se/ericsson/lmc/uu/corba/h225/gatek
eeper/DomainController:1.0";sequence<TaggedProfile> profiles=<struct TaggedProfi
le<unsigned long tag=0;sequence<octet> profile_data=<133 bytes: (0)(1)(0)(0)(0)
(0)(15)[1][3][4][1].][1][1][7][1].][15][7][1].][1][7][1][8](0)(0)(19)[=]1(0)(0)(0)(0)(0)
[e]1(0)[P][M][C]1(0)(0)(0)(0)(0)[B][I][D][L]:][s][e][l][e][r][i][l][c][s][s][o][n][l][m][f][c]
[l][u][l][f][c][o][l][r][b][a][l][l][o][b][l][j][e][l][c][t][M][a][l][n][a][l][g][e][l][r][l][f][0]
[l][o][l][m][a][l][l][i][n][c][l][o][l][n][t][r]....>];>];>]]
Log::writeBackup: Writing 664 bytes
Extended Naming Factory: root = com.visigenic.vbroker.services.CosNaming.NamingC
ontextImpl[Server,oid=PersistentId[repId=IDL:omg.org/CosNaming/NamingContext:1.0
,objectName=NamingService/1]]
rebind(name=LName[<id=NamingService,kind=>], type=ncontext, context=com.visigeni
c.vbroker.services.CosNaming.NamingContextImpl[Server,oid=PersistentId[repId=IDL:
omg.org/CosNaming/NamingContext:1.0,objectName=NamingService/1]])

```

The naming service allows us to associate one or more logical names with an object implementation and stores those names in a namespace. There are important differences between the Visibroker naming service and osagent. The naming service allows object implementation to bind logical names to its object at runtime. The Visibroker 3.3 naming service must be start with vbj, which is the JDK provide by Inprise, since some irregularities have been noticed with JDK1.2.2 from SUN.

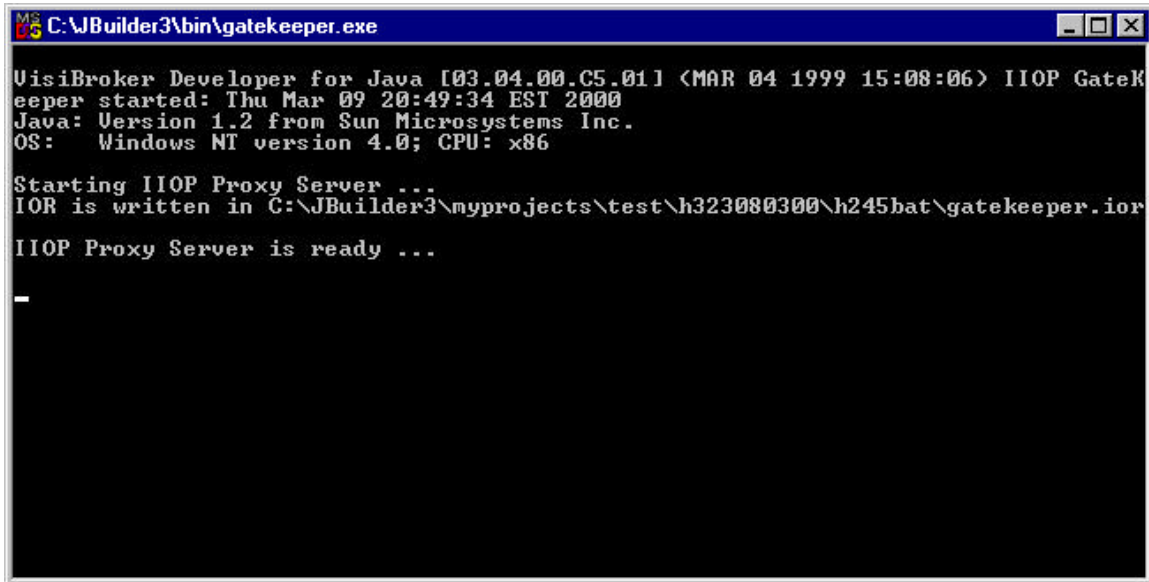


Step 4. Start Gatekeeper (Domain Controller) (H.225):



### H323 Caller Side

Step 1. Start URL Naming Service (H.245 optional):



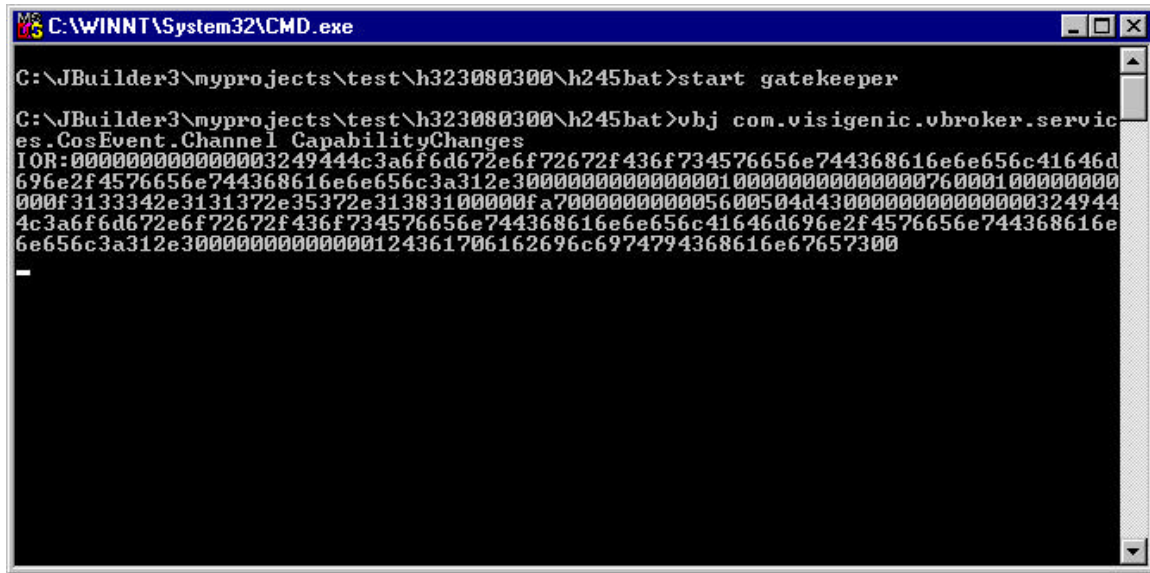
```
C:\JBuilder3\bin\gatekeeper.exe
UisiBroker Developer for Java [03.04.00.C5.01] <MAR 04 1999 15:08:06> IIOP GateKeeper started: Thu Mar 09 20:49:34 EST 2000
Java: Version 1.2 from Sun Microsystems Inc.
OS: Windows NT version 4.0; CPU: x86

Starting IIOP Proxy Server ...
IOR is written in C:\JBuilder3\myprojects\test\h323080300\h245bat\gatekeeper.ior
IIOP Proxy Server is ready ...

-
```

This is an alternative to the CORBA Naming Service and osagent to locate objects as used in H.225 part. The URL Naming Service uses any commercial Web Server as a Directory Service for retrieving stringified object IORs. The only requirement is that the Web Server/firewall enables HTTP PUT commands. Here, we use Visibroker Gatekeeper (IIOP Proxy Server) to simulate the function of Web Server.

Step 2. Start Event Service for Capability Changes (H.245):



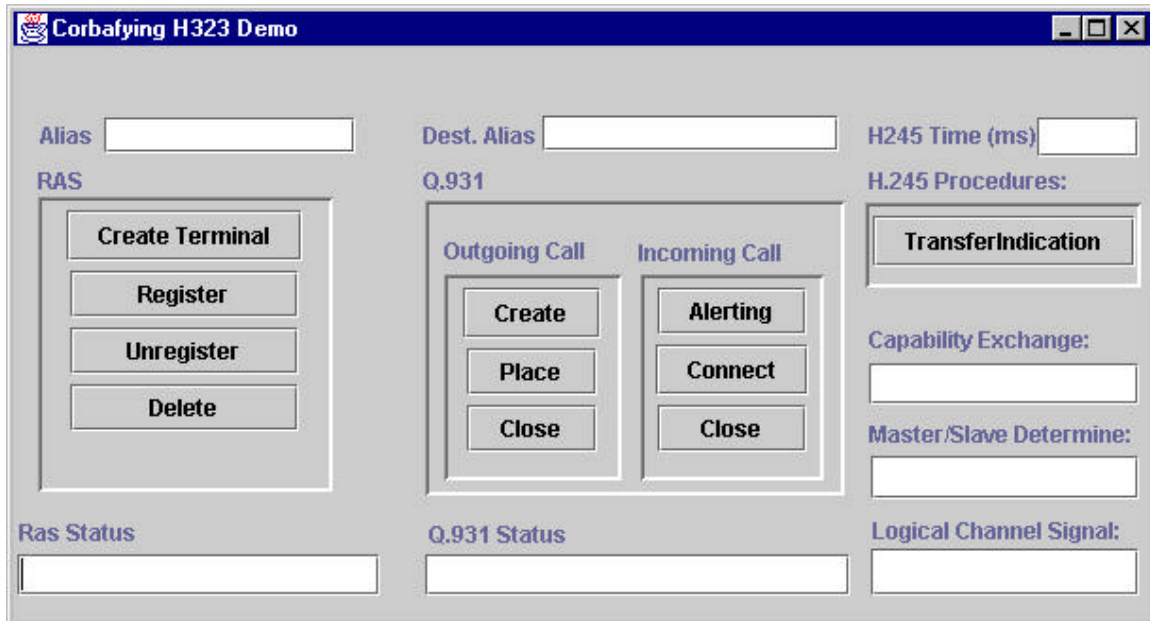
```

C:\WINNT\System32\CMD.exe
C:\JBuilder3\myprojects\test\h323080300\h245bat>start gatekeeper
C:\JBuilder3\myprojects\test\h323080300\h245bat>obj com.visigenic.vbroker.servic
es.CosEvent.Channel.CapabilityChanges
IOR:00000000000000003249444c3a6f6d672e6f72672f436f734576656e744368616e6e656c41646d
696e2f4576656e744368616e6e656c3a312e30000000000000010000000000000076000100000000
000f3133342e3131372e35372e31383100000fa700000000005600504d43000000000000324944
4c3a6f6d672e6f72672f436f734576656e744368616e6e656c41646d696e2f4576656e744368616e
6e656c3a312e30000000000000124361706162696c6974794368616e67657300
-

```

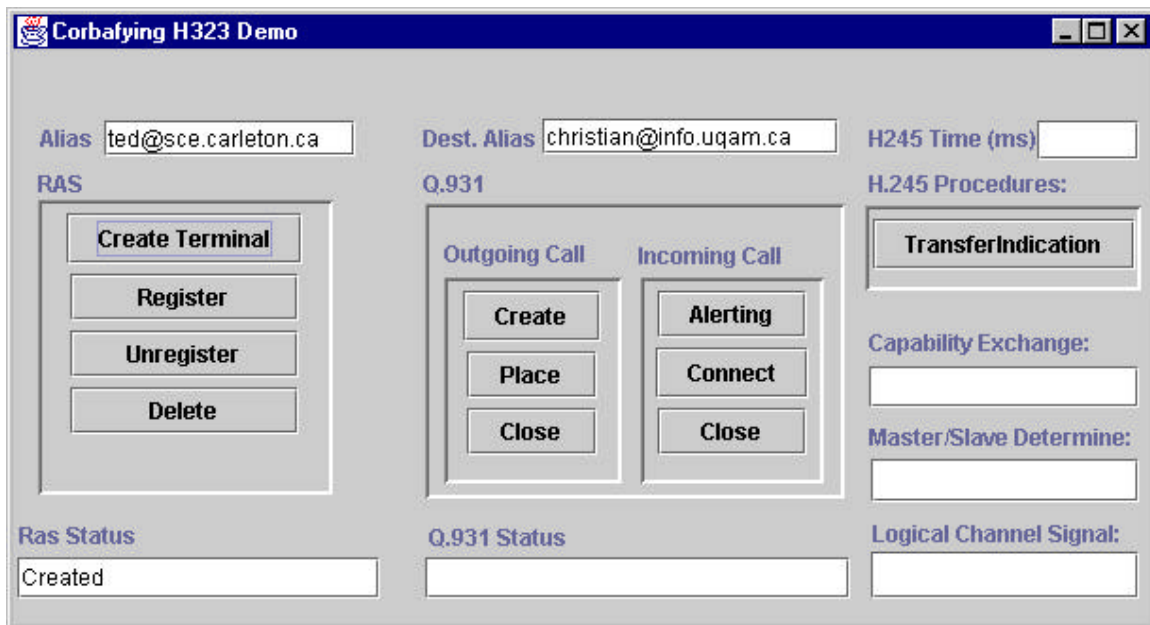
CORBA Event Service supports asynchronous, disconnected communications between CORBA clients. There are three primary participants in the Event Service: Consumer, Supplier, and Channel. There are two general approaches for initiating event communication between suppliers and consumers: The Push Model and the Pull Model. The contents of events are of type Any, which provides a loosely typed interface between consumers and suppliers. Here, we implemented the Push Model to allow H.245 Capability Exchange Signaling Entity (CESE) to notify its capability changes to the other end. This is done after regular H.245 signaling.

Step 3. Start H323 Terminal:



This is the GUI of H.323 Caller for demonstrating H.225/H.245 signaling procedures. This part starts after Step. 2 in the Callee side.

Step 3.1



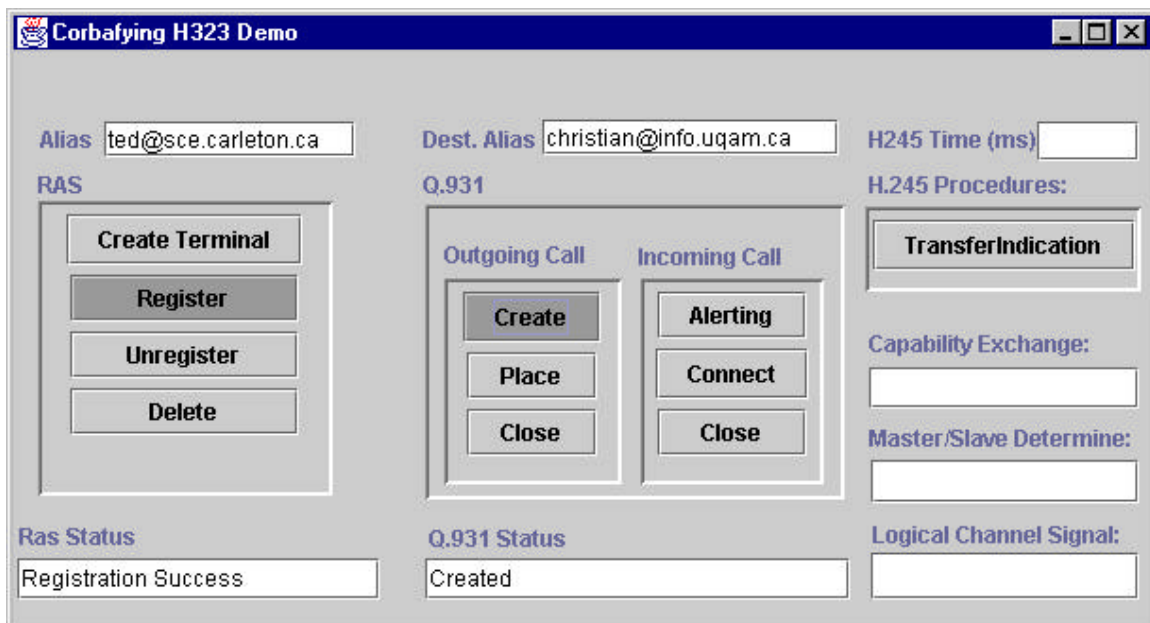
An H.323 Caller enters the alias of its own address, as [ted@sce.carleton.ca](mailto:ted@sce.carleton.ca), followed with the destination address, as [christian@info.uqam.ca](mailto:christian@info.uqam.ca). The user clicks the “Create Terminal” button to create the H.225 RAS terminal.

## Step 3.2



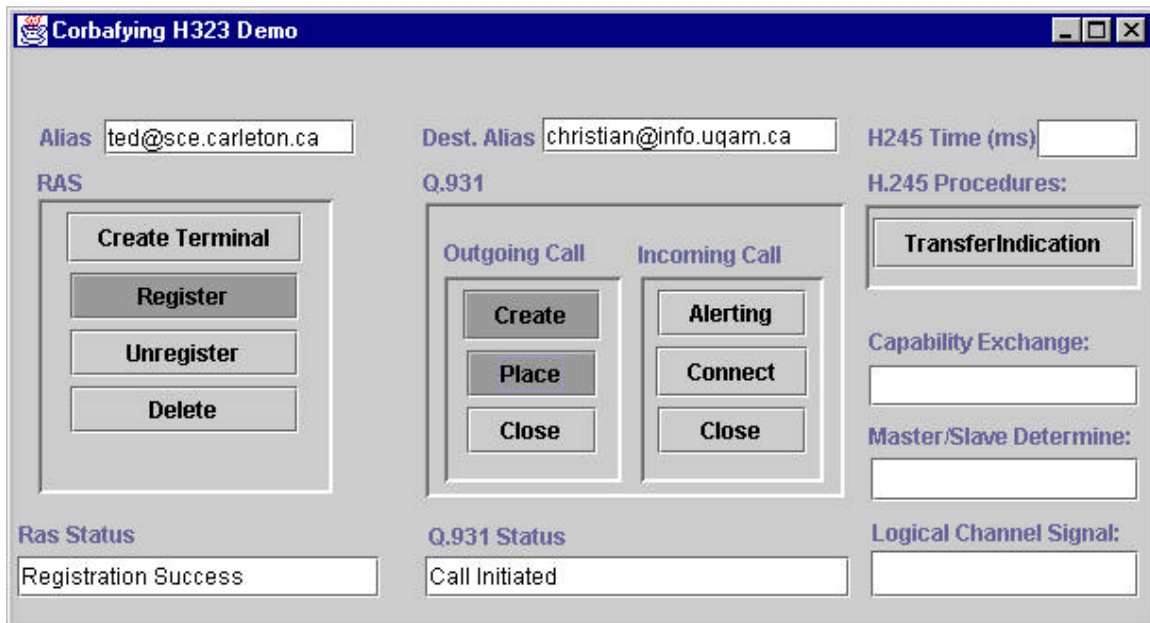
An H.323 Caller registers in Domain Controller. Please check Step 3.3 on the Callee side for Domain Controller's reaction.

## Step 3.3



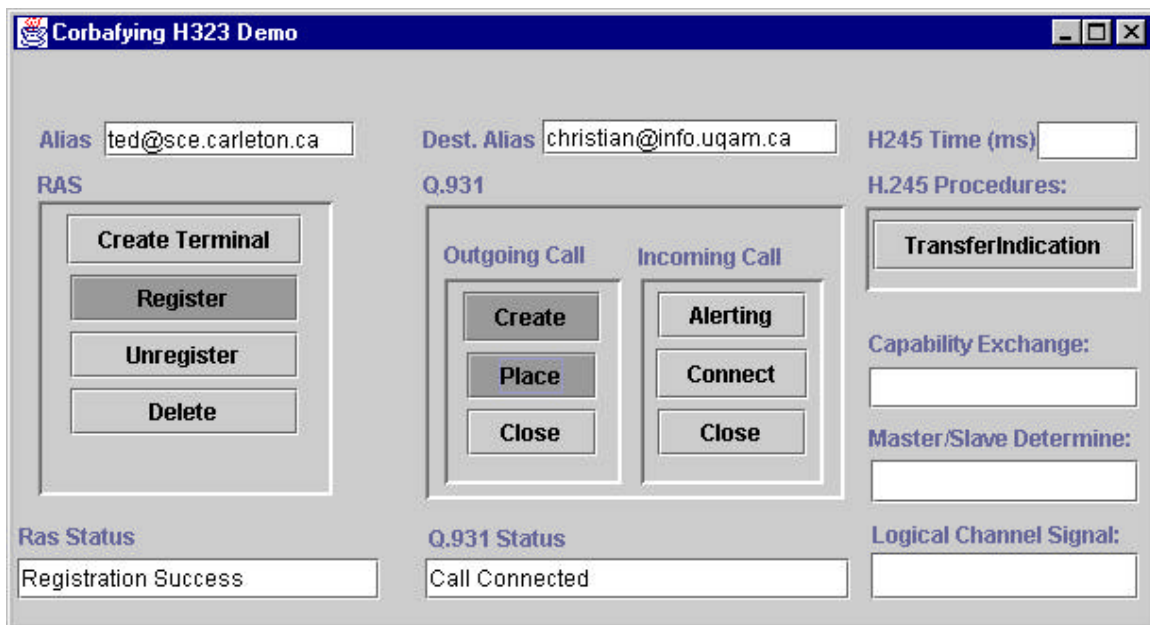
An H.323 Caller creates an outgoing call with Q.931 messages.

## Step 3.4



An H.323 Caller places a direct call to the Callee. Please check Step 3.4 on the Callee side for the Callee's reaction.

## Step 3.5



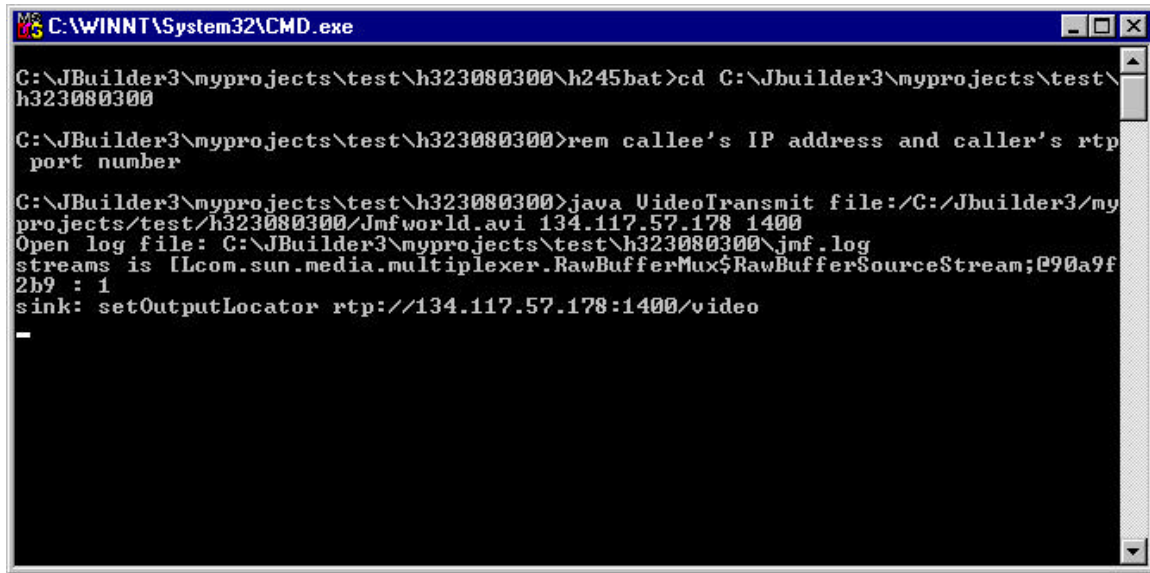
An H.323 Caller gets the reaction from the Callee.

## Step 3.6



An H.323 Caller starts H.245 signaling. There are three procedures altogether, i.e. Capability Exchange, Master/Slave Determination and Logical Channel Signaling. Messages are translated from their original definition in ASN.1 to CORBA IDL user defined data types. The time box indicates the time for all three procedures. We have done some analysis and made improvements on the performance by using Visibroker message interceptor.

Step 4. Start RTP video transmission:



```

C:\WINNT\System32\CMD.exe
C:\JBuilder3\myprojects\test\h323080300\h245bat>cd C:\Jbuilder3\myprojects\test\
h323080300
C:\JBuilder3\myprojects\test\h323080300>rem callee's IP address and caller's rtp
port number
C:\JBuilder3\myprojects\test\h323080300>java VideoTransmit file:/C:/Jbuilder3/my
projects/test/h323080300/Jmfworld.avi 134.117.57.178 1400
Open log file: C:\JBuilder3\myprojects\test\h323080300\jmf.log
streams is [Lcom.sun.media.multiplexer.RawBufferMux$RawBufferSourceStream;090a9f
2b9 : 1
sink: setOutputLocator rtp://134.117.57.178:1400/video
-

```

This is implemented with Java Media Framework (JMF) 2.0. Using the RTP port number agreed to in previous H.245 Logical Channel Signaling procedures, a caller starts to transfer the video across the wire. Instead of transmitting the local file as “file:/C:/Jbuilder/myprojects/test/h323080300/Jmfworld.avi”, the program also can capture datasource as “vfw://0” on Windows or “sunvideo://0/1/JPEG” on Solaris through a video camera and corresponding video board. Some formats of video may not be transferred because of the limitation in support for codec.



## H323Callee:

Step 1. Start URL Naming Service (H.245 optional):

```

E:\JBUILDER3\bin\gatekeeper.exe

UisiBroker Developer for Java [03.04.00.C5.01] <MAR 04 1999 15:08:06> IIOP GateK
eeper started: Thu Mar 09 21:03:38 EST 2000
Java: Version 1.2 from Sun Microsystems Inc.
OS:   Windows NT version 4.0; CPU: x86

Starting IIOP Proxy Server ...
IOR is written in C:\users\gatekeeper.iior
IIOP Proxy Server is ready ...

-

```

```

E:\JBUILDER3\bin\gatekeeper.exe

UisiBroker Developer for Java [03.04.00.C5.01] <MAR 04 1999 15:08:06> IIOP GateK
eeper started: Thu Mar 09 21:03:38 EST 2000
Java: Version 1.2 from Sun Microsystems Inc.
OS:   Windows NT version 4.0; CPU: x86

Starting IIOP Proxy Server ...
IOR is written in C:\users\gatekeeper.iior
IIOP Proxy Server is ready ...

134.117.57.178:15000    134.117.57.178:1165    PUT /cESEclient.iior HTTP/1.0
134.117.57.178:15000    134.117.57.178:1165    null
134.117.57.178:15000    134.117.57.178:1166    PUT /mSDSEclient.iior HTTP/1.0
134.117.57.178:15000    134.117.57.178:1166    null
134.117.57.178:15000    134.117.57.178:1167    PUT /lCSEclient.iior HTTP/1.0
134.117.57.178:15000    134.117.57.178:1167    null

```

Step 2. Start Object Activation Daemon (H.245 optional)

```

C:\WINNT\System32\CMD.exe
E:\JBuilder3\myprojects\Test\h323080300\h245bat>Rem "cd" to current directory fo
r project files
E:\JBuilder3\myprojects\Test\h323080300\h245bat>cd E:\Jbuilder3\myprojects\test\
h323080300
E:\JBuilder3\myprojects\TEST\h323080300>oadj
IOR is written in E:\JBuilder3\myprojects\TEST\h323080300\oadj.ior
oad now running: IOR:0000000000000002549444c3a7669736967656e69632e636f6d2f416374
69766174696f6e2f4f41443a312e30000000000000010000000000000006700010000000000f31
33342e3131372e35372e3137380000047b00000000004700504d4300000000000002549444c3a76
69736967656e69632e636f6d2f41637469766174696f6e2f4f41443a312e300000000000000f31
33342e3131372e35372e31373800

```

```

C:\WINNT\System32\CMD.exe
E:\JBuilder3\myprojects\Test\h323080300\h245bat>cd E:\Jbuilder3\myprojects\test\
h323080300
E:\JBuilder3\myprojects\TEST\h323080300>oadj
IOR is written in E:\JBuilder3\myprojects\TEST\h323080300\oadj.ior
oad now running: IOR:0000000000000002549444c3a7669736967656e69632e636f6d2f416374
69766174696f6e2f4f41443a312e30000000000000010000000000000006700010000000000f31
33342e3131372e35372e3137380000047b00000000004700504d4300000000000002549444c3a76
69736967656e69632e636f6d2f41637469766174696f6e2f4f41443a312e300000000000000f31
33342e3131372e35372e31373800
#1=vhj -UBJprop Ooad_uid=1 -UBJprop OaactivateIOR=IOR:0000000000000002549444c3a7
669736967656e69632e636f6d2f41637469766174696f6e2f4f41443a312e30000000000000010
0000000000006700010000000000f3133342e3131372e35372e3137380000047b0000000000470
0504d430000000000002549444c3a7669736967656e69632e636f6d2f41637469766174696f6e2
f4f41443a312e3000000000000000f3133342e3131372e35372e31373800 -UBJprop ORBagentP
ort=14000 se.ericsson.lmc.uu.corba.h245.H245Callee

```

The Object Activation Daemon (OAD) handles large systems of object implementation. OAD works in conjunction with the CORBA Implementation Repository database to start up object implementation on demand. It uses state objects to indicate whether the Server Object is active, inactive or waiting for activation as for H.245 signaling entities.

## Step 3. Start H323 Terminal

The screenshot shows the 'Corbafying H323 Demo' application window. It features several input fields and button groups:

- Alias:** An empty text input field.
- Dest. Alias:** An empty text input field.
- H245 Time (ms):** An empty text input field.
- RAS:** A group of four buttons: 'Create Terminal', 'Register', 'Unregister', and 'Delete'.
- Q.931:** Two columns of buttons. The 'Outgoing Call' column contains 'Create', 'Place', and 'Close'. The 'Incoming Call' column contains 'Alerting', 'Connect', and 'Close'.
- H.245 Procedures:** A 'TransferIndication' button.
- Capability Exchange:** An empty text input field.
- Master/Slave Determine:** An empty text input field.
- Ras Status:** An empty text input field.
- Q.931 Status:** An empty text input field.
- Logical Channel Signal:** An empty text input field.

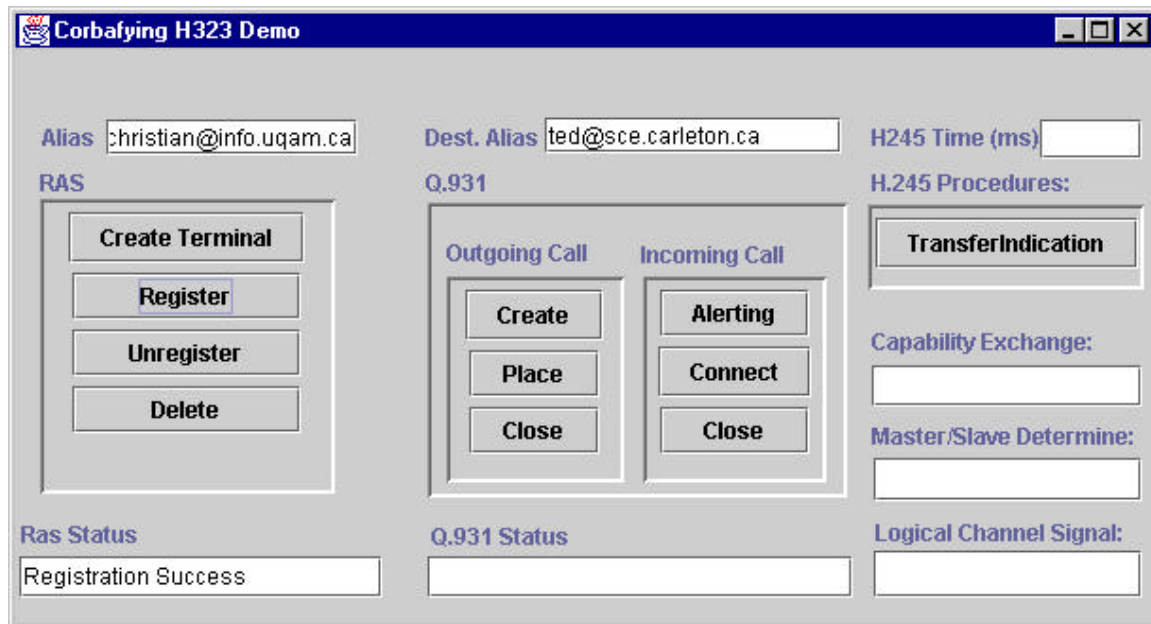
## Step 3.1

The screenshot shows the 'Corbafying H323 Demo' application window after the 'Create Terminal' button has been clicked. The 'Ras Status' field now displays the text 'Created'. The other fields and buttons remain the same as in the previous screenshot:

- Alias:** christian@info.uqam.ca
- Dest. Alias:** ted@sce.carleton.ca
- H245 Time (ms):** (empty)
- RAS:** Buttons for 'Create Terminal', 'Register', 'Unregister', and 'Delete'.
- Q.931:** Buttons for 'Create', 'Place', 'Close' (Outgoing) and 'Alerting', 'Connect', 'Close' (Incoming).
- H.245 Procedures:** 'TransferIndication' button.
- Capability Exchange:** (empty)
- Master/Slave Determine:** (empty)
- Ras Status:** Created
- Q.931 Status:** (empty)
- Logical Channel Signal:** (empty)

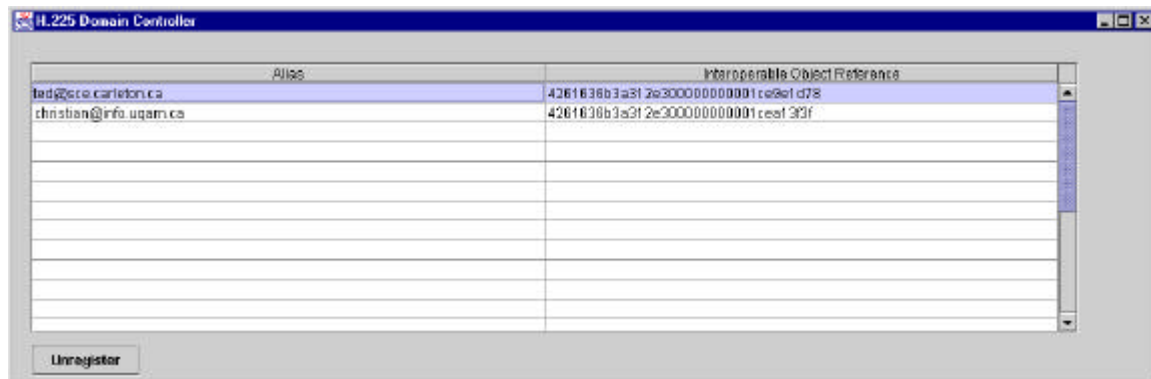
An H.323 Callee enters the alias of its own address, as [christian@info.uqam.ca](mailto:christian@info.uqam.ca), followed with the destination address, as [ted@sce.carleton.ca](mailto:ted@sce.carleton.ca). Click the “Create Terminal” button to create the H.225 RAS terminal.

Step 3.2



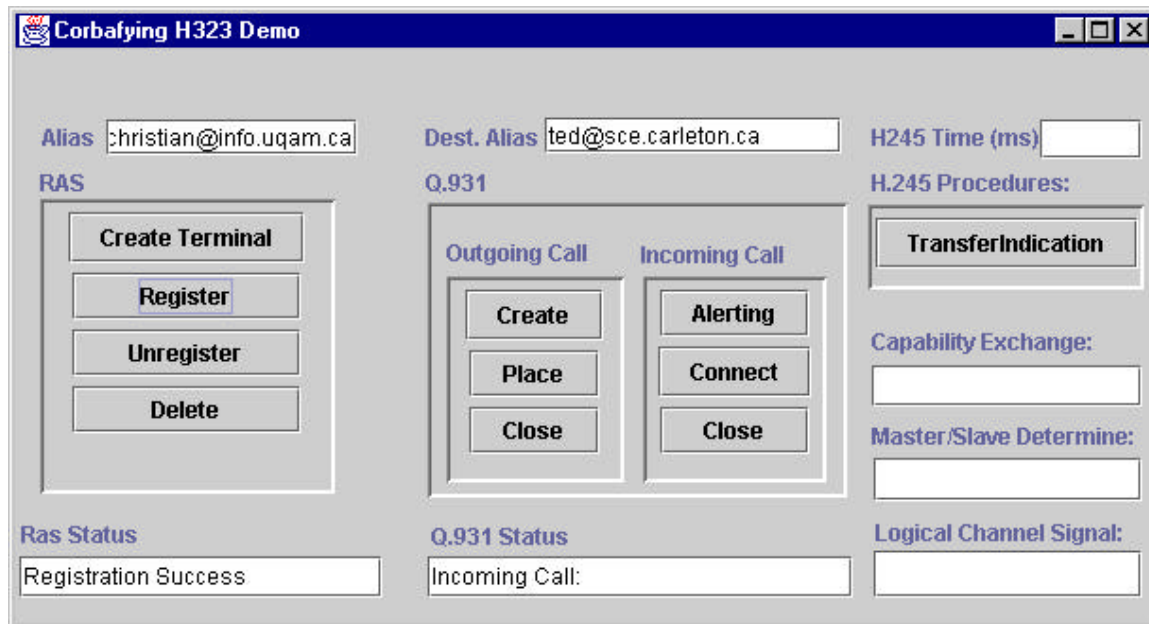
Terminal created and registered at Domain Controller.

Step 3.3



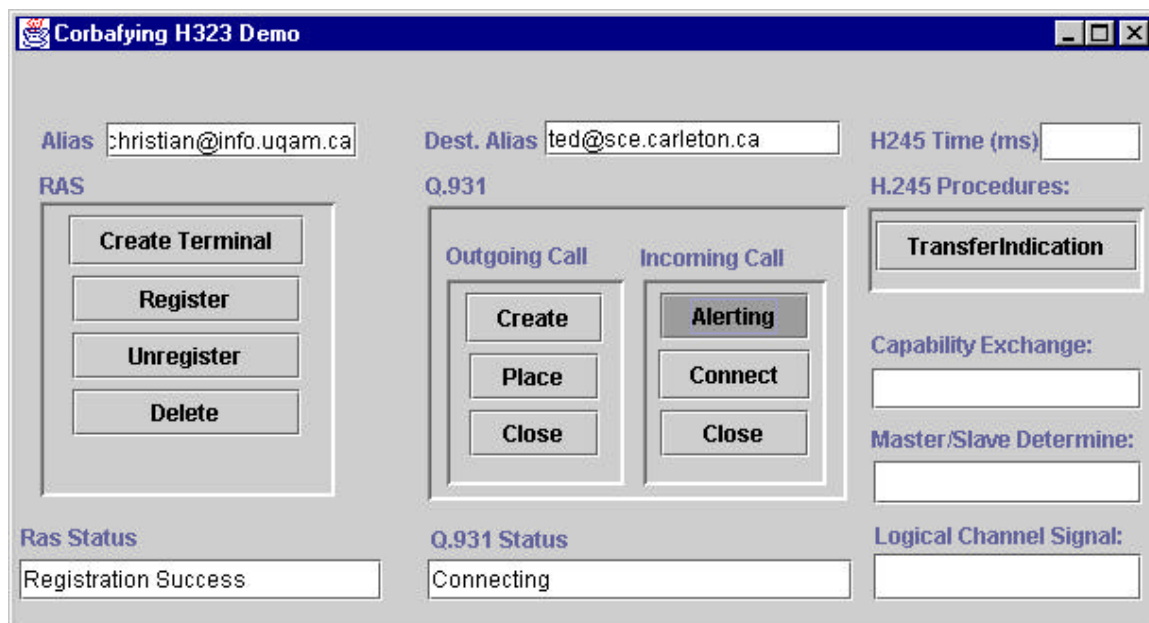
Domain Controller matches alias to IOR.

## Step 3.4



Receive Incoming call.

## Step 3.5



Send back "Alert".

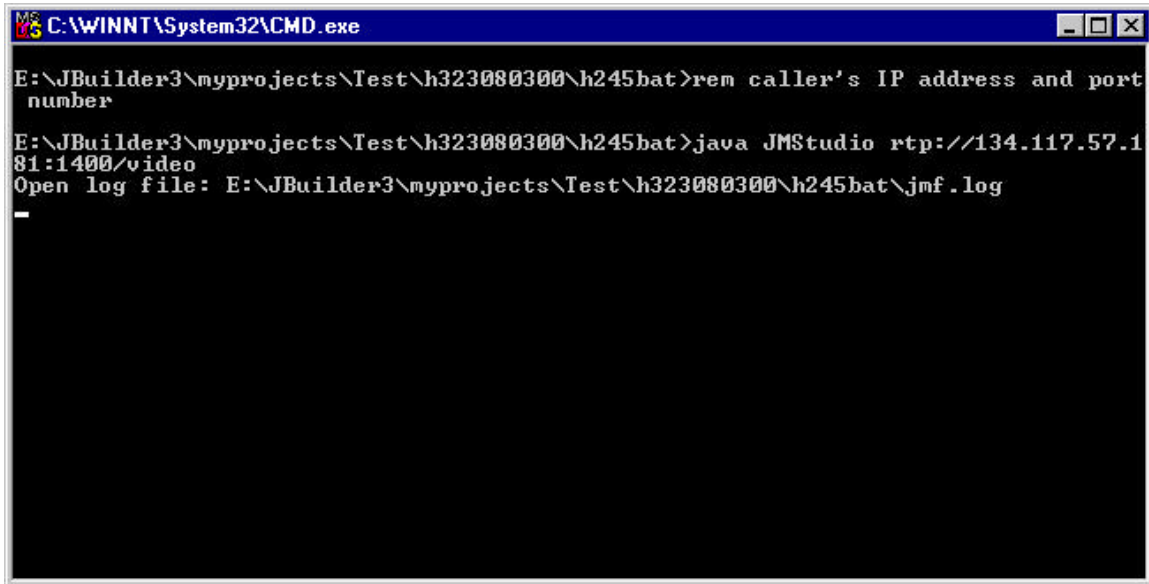
## Step 3.6

The screenshot shows a window titled "Corbafying H323 Demo" with a blue title bar. The interface is divided into several sections:

- Alias:** A text field containing "christian@info.uqam.ca".
- Dest. Alias:** A text field containing "ted@sce.carleton.ca".
- H245 Time (ms):** An empty text field.
- RAS:** A section containing four buttons: "Create Terminal", "Register", "Unregister", and "Delete".
- Q.931:** A section divided into two columns: "Outgoing Call" and "Incoming Call".
  - Outgoing Call:** Contains three buttons: "Create", "Place", and "Close".
  - Incoming Call:** Contains three buttons: "Alerting", "Connect", and "Close".
- H.245 Procedures:** A section containing a button labeled "TransferIndication".
- Capability Exchange:** A section with an empty text field.
- Master/Slave Determine:** A section with an empty text field.
- Ras Status:** A text field containing "Registration Success".
- Q.931 Status:** A text field containing "Connected".
- Logical Channel Signal:** An empty text field.

Send back "Connect".

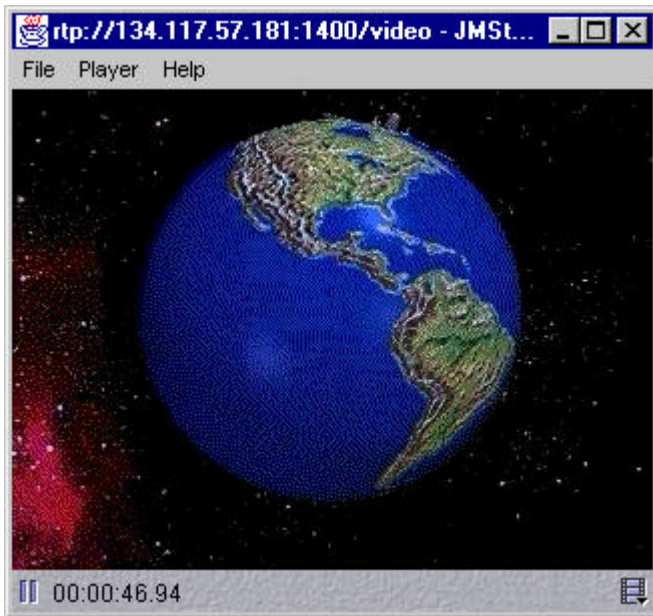
Step 4. Be ready to receive RTP Video:



```
C:\WINNT\System32\CMD.exe
E:\JBuilder3\myprojects\Test\h323080300\h245bat>rem caller's IP address and port
number
E:\JBuilder3\myprojects\Test\h323080300\h245bat>java JMStudio rtp://134.117.57.1
81:1400/video
Open log file: E:\JBuilder3\myprojects\Test\h323080300\h245bat\jmf.log
-
```



Step 4.1



Receiving video transmitted from Caller. See, the globe is rolling. ↗