INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

A Component-based Approach to Provide IN Type of Services on IP Network

submitted by

Jing Qu (B. Engineering)

**A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of**

Master of Applied Science,

**Ottawa-Carleton Institute of Electric and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6, Canada**

February 12, 2002

© 2002, Jing Qu

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**385 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**385, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72069-1

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

A Component-based Approach to Provide IN Type of Service on IP Network

submitted by

Jing Qu,

In partial fulfillment of the requirements
for the degree of Master of Applied Science

Dr. Bernard Pagurek, Thesis Supervisor

Chairman, Department of Systems and Computer Engineering

Carleton University

February 12, 2002

ABSTRACT

The emergence of new telecommunication services especially the surge of Internet challenges the existing Intelligent Network-based service deployment architecture.

This thesis demonstrated the design and implementation process for an Open Service Architecture (OSA) which supports third party service providers using the component-oriented and distributed computing technologies. The OSA deployed a messaging framework – event-oriented messaging. An event-driven messaging mechanism is independent of the underlying signalling system, so that it is easy for a service provider to integrate components from different sources. Besides the open service provisioning architecture, this study also considers code reuse to be an important issue. Unlike the Service Independent Building Block (SIB) approach in traditional IN Concept Module, an active development idea is shown using Beans. The research on service build-up process focused on service logics instead of service entities as most other projects did. An automatic extendable framework was proposed, based on Java Reflection and Serialization APIs. With such framework, a service provider can easily integrate services from multiple vendors and develop new services using other service providers' existing services. Therefore the research achieved two levels of reuse. One is to integrate the existing feature to build up a new service (SIB-oriented approach) which takes place in the deployment phase. Another is to create a new service with existing services, which is achieved at runtime.

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my supervisor, Professor Bernard Pagurek, for his always guidance and encouragement in this research. I also want to thank my wife as well as my family for their love and support to help me finish my study.

My colleagues at the Network Management Lab have been very helpful and supportive during my study. At the same time, I would also like to express my gratitude to all those who have helped me in this research.

Last not the least, I would like to thank Communication Information Technology Ontario (CITO) for providing grants to help me finish this research.

TABLE OF CONTENTS

ABSTRACT	3
ACKNOWLEDGEMENTS	4
LIST OF FIGURES	7
LIST OF TABLES	8
ABBREVIATION	9
1. INTRODUCTION.....	10
1.1 OBJECTIVE OF STUDY	11
1.2 CONTRIBUTION OF STUDY	12
2. PROBLEMS WITH TODAY'S SERVICE ARCHITECTURE.....	17
2.1 INTELLIGENT NETWORK CONCEPT	17
2.2 CONSTRAINT OF IN FOR OPEN SERVICE DEPLOYMENT	19
2.3. OPEN SERVICE FRAMEWORK	22
2.3.1 PINT	23
2.3.2 ETSI TIPPHON.....	25
3.2.1 Terminal:.....	26
3.2.2 GateKeeper:	27
3.2.3 Gateway:	27
2.3.3 JAIN	29
2.3.3.1 JAIN Architecture.....	30
2.3.3.2 JAIN Service Logic Execution Environment.....	34
2.3.4 LIMITATION OF EXISTING OPEN SERVICE ARCHITECTURE	35
3. DESIGN OF OPEN SERVICE ARCHITECTURE.....	37
3.1 COMPONENTS DEFINITION	39
3.2 SYSTEM ARCHITECTURE	41
3.3 COMPONENTS OF THE OPEN SERVICE ARCHITECTURE.....	44
3.3.1 End User Layer.....	44
3.3.2 Network layer.....	44
3.3.3 Adaptive Layer.....	46
3.3.4 Service Layer	47
4. IMPLEMENTATION TECHNOLOGY	52
4.1 EUT IMPLEMENTATION TECHNOLOGY.....	52
4.1.1 Microsoft's Telephony API	53
4.1.1.1 TAPI and its Call Control Architectures	54
4.1.2 Sun's Java Telephony API (JTAPI)	56
4.1.2.1 JTAPI Classes.....	56
4.1.2.2 The JTAPI Call Model	57
4.1.3 Signal Computing Signal Architecture(SCSA).....	59
4.1.3.1 SCSA Software Standards or TAO	60
4.1.3.2 SCSA Hardware Model.....	61
4.1.4 Choose the right telephony APIs for my OSA.....	61
4.2 ADAPTIVE LAYER AND SERVICE LAYER.....	62
4.2.1 Sun's Enterprise JavaBean (EJB)	63

4.2.1.1 Java Beans.....	64
4.2.1.2 Enterprise Java Beans (EJB)	65
4.2.1.3 EJB Client.....	66
4.2.1.4 EJB Container.....	67
4.2.1.4.1 EJB Remote Interface EJB Object.....	67
4.2.1.4.2 EJB Home Interface and Home Object	67
4.2.1.5 EJB Server.....	68
4.2.1.6 EJB Design Patterns	68
4.2.1.6.1 Session Beans.....	68
4.2.1.6.2 Entity Beans	68
4.2.3 <i>Microsoft Transaction Server (MTS)</i>	70
4.2.3.1 Component Object Model	71
4.2.3.2 MTS Architecture.....	72
4.2.4 <i>Choose the right framework for my OSA</i>	73
5. IMPLEMENTATION OF SERVICE ARCHITECTURE	78
5.1 DESIGN OF UNIVERSAL CALL FORWARD.....	78
5.2 SIMPLE SERVICE SCENARIO	80
5.3 END USER LAYER IMPLEMENTATION	82
5.4 NETWORK LAYER IMPLEMENTATION	87
5.4.1 <i>Framework Discussion</i>	91
5.5 ADAPTIVE LAYER IMPLEMENTATION	92
5.6 SERVICE LAYER IMPLEMENTATION	96
5.6.1 <i>Intelligent Service Proxy</i>	96
5.6.2 <i>Abstract-factory based service composition</i>	101
5.6.3 <i>Component-based Service Independent Building Block</i>	104
5.7 ARCHITECTURE TEST	112
6. SUMMARY	124
6.1 CONCLUSION	124
6.2 FUTURE WORK.....	126
REFERENCE:	128

LIST OF FIGURES

<i>Number</i>	<i>Page</i>
Figure 1 Intelligent Network Structure.....	18
Figure 2 PINT Architecture	23
Figure 3 TIPHON Basic Call Reference Model.....	26
Figure 4 Architecture of TIPHON services.....	28
Figure 5 JAIN architecture.....	30
Figure 6 JAIN Abstraction.....	32
Figure 7 Open Service Architecture	43
Figure 8 Service Repository Components.....	48
Figure 9 TAPI Architecture	53
Figure 10 Third Party Call Control In TAPI	55
Figure 11 JTAPI packages	57
Figure 12 JTAPI Call Model	58
Figure 13 SCSA Component Architecture.....	60
Figure 14 EJB architecture.....	66
Figure 15 Microsoft Transaction Server Architecture	71
Figure 16 Universal Call Forwarding Scenario.....	80
Figure 17 Terminal Feature-Subscription Procedure.....	85
Figure 18 Client GUI Snapshot	86
Figure 19 Class diagram of Plug-in Framework	87
Figure 20 Distributed InfoBus	94
Figure 21 HalfObject Class Diagram	99
Figure 22 Call Transfer based on AbstractFactory Pattern.....	103
Figure 23 Screen Bean Class Diagram	107
Figure 24 Service Logic of Universal Call Forward	113
Figure 25 Called Party and Transfer Party Configuration	114
Figure 26 Calling Party Issues the Call	115
Figure 27 Network Layer Processing	116
Figure 28 Service Layer Processing	117
Figure 29 Called Party End-User Terminal.....	117
Figure 30 Call Diversion Processing on Network Layer	118
Figure 31 Transfer Party receives call	119
Figure 32 Called/Transferred Party EUT Log File	120
Figure 33 Calling Party EUT Log File	121
Figure 34 Compare bean Integration within BDk1.1	122

LIST OF TABLES

<i>Number</i>	<i>Page</i>
Table 1 Platform Comparison.....	76
Table 2 Operation, POI and PORs of SIBs	112

ABBREVIATION

BCP	Basic Call Processing
CCM	CORBA Component Model
CCS7/SS7	Common Channel Signaling No.7
CID	Call Instance Data
CORBA	Common Object Broker Architecture
CPE	Customer Premise Equipment
CS-1,2,3,4	Capability Set-1,2,3,4
DCOM/COM	Component Object Model
EJB	Enterprises Javabeans
EUT	End User Terminal
HOPP	Half Object Plus Protocol
IN	Intelligent Network
INAP	Intelligent Network Application Protocol
INCM	Intelligent Network Concept Model
JAIN	Java Advance Intelligent Network
JTAPI	Java Telephony API
JVM	Java Virtual Machine
MA	Mobile Agent
MTS	Microsoft Transaction Server
OSA	Open Service Architecture
POTS	Plain Old Telephone Service
SCE	Service Creation Environment
SCP	Service Control Point
SDP	Service Data Point
SIB	Service Independent Building Block
SIP	Session Initiated Protocol
SLEE	Service Logic Execution Environment
SMS	Service Management System
SSD	Service Specific Data

Chapter 1

INTRODUCTION

It's an information era. The emergence of new telecommunication services has already and will continue to challenge the existing service deployment architecture. The surge of Web-oriented technology is generating a need for rapid, secure and flexible service provisioning framework.

The Intelligent Network (IN) has played an important role in the evolution of the telecommunication business since the end of the 80s. It has enabled the introduction of new telephony services without applying all local switches for every new service. By outsourcing the service logic to the Service Control Point (SCP) and other components, it has promoted the opening up of the telecommunication infrastructure, creating new services and new markets. IN has set up a base for faster and easier service provisioning in telecommunication market.

While IN has experienced fast growth, the telecommunication industry has witnessed a boom in the Internet. The Web offers a wide range of primary data applications. Compared with IN, services on the Internet can be developed and deployed in a shorter time-to-market frame. Because of the open environment provided by the Internet and the fascinating growth of E-Commerce, many carriers are investigating the development and deployment of telecommunication services over the Internet or even in an integrated IN-Internet environment.

The business environment for telecommunication services is undergoing rapid change. There is an increasing user demand for “simple-to-use” and customizable services which integrate multi-media applications. The ability to merge the ubiquitous telephony services while achieving user friendliness is recognized as the Internet’s major strength in promoting new services. The explosive growth of Internet and POTS (Plain Old Telephone Service)-related services, as well as the demand for integrated management of IN services, require a generic architecture that can bridge both domains, IN and the Internet. [1]

1.1 Objective of Study

The integration and evolution of the IN world with Information Technology and Internet-based solutions has led to a new view of intelligence in the network. Network Intelligence is no longer limited to a technology-specific solution from one aspect of telecommunication, but can encompass many different technologies, mechanisms and methodologies for the rollout of new and advanced services.

This study attempts to provide answers for the issue of where/how Network Intelligence can/must be utilized or augmented to provide or to facilitate greater flexibility in service provisioning and customization. In this thesis, an open service architecture is deployed, relying on the introduction of advanced Information Technology and strategies to enhance flexibility and openness for traditional IN systems. Through the design and implementation of an IN-Internet integrated service, I have presented answers to the following issues, which should be the main concerns for a service provisioning framework:

- How to integrate, organize and distribute services from different vendors so that a service provider can reuse the available features to construct a new service.
- How to help customers dynamically choose service providers.

- How to provide a generic service platform that incorporates flexibility for future extension.

1.2 Contribution of study

The provisioning of a generic architecture for the development and the deployment of services is of utmost importance to operators. In my thesis, I consider this challenge in more detail. Since services are ultimately realized through the signalling system and integration of a new service logic that guides the behaviour of each of the system elements, my design and implementation focus on those two aspects.

In my thesis, each service is deployed using component-oriented technology (Javabeen) which defines the standard interface to be manipulated by other applications. Compared with building blocks reuse theory in traditional IN, my bean-oriented approach not only provides a unified functional abstraction but also defines the underlying implementation technology. So these components tend to be more easily reused across application and developers. I also defined a distributed event service as the information exchanging mechanism between software components. The distributed event service extends the existing InfoBus APIs with distributed computing technology (RMI/CORBA). Traditionally, the InfoBus API only supports data exchange within one Java Virtual Machine (JVM). With my extension, two JavaBean components can communicate across a network with a standard Java event service. Based on my service architecture, a service can be deployed by seamlessly integrating different vendors' available functionalities (components). The components are not only service independent but also vendor neutral. Components from different vendors can exchange information smoothly without setting up a specific interface for new components, or reinventing the interfaces for existing components.

While my bean-oriented component approach provides more flexibility in service composition than traditional IN (i.e. application developers can integrate building blocks from different vendors based on a unified information exchange mechanism), there is no major difference in the reuse theory. Both solutions achieve the reuse in development time instead of runtime. Such a static approach cannot fit the requirements for an open service architecture, especially in a third-party service provisioning environment. For example, a service provider wants to develop an Internet Call Forwarding feature based on an existing Call Waiting and Internet Call feature. With the building block solution, the developer needs to import the building blocks from both features to generate the new service. This means that if we try to integrate a new building block, we need to recompile the whole system. In the service provisioning framework, how a new service is integrated into an existing application will directly influence the system's effectiveness and adaptability. In my thesis, I extend the reuse theory from static to dynamic by defining an auto-pluggable service repository based on Java Reflection technology. The service repository can be extended at runtime without influencing the existing feature and, more importantly, it supports third party service provisioning. By integrating different vendors' services into its service repository, a third party service provider can offer new services to end users. The end user deals with the third party service providers in the same way as he/she deals with "normal" service providers. The only difference is at runtime – when the end user utilises the service, the third party service provider dynamically links them to a normal service provider and invokes the underlying service. For example, by integrating a different vendor's Call Forwarding service, a third party service provider can offer a new service called Preference Call Forwarding (PCF). In PCF, an end user can choose to use the service

based on pre-defined strategy. For example, the end user can choose the call forwarding service which is cheapest among those available from several service providers. Such a solution offers more services and choices for the consumer, because operators are able to offer a wide spectrum of services, and service providers can roll out their services cheaper and faster. My design provides the means for rapid conception, development, and deployment of a wide range of new services, and it is easy for third parties to create new services based on existing features. The environment provides flexibility for end users: they can purchase different services from different service providers (i.e. call forward from Bell Canada and call waiting from Sprint).

Since the mid-90s, new telecommunication and multimedia services have been available through voice-over IP telephony. Different architectures have been proposed, such as the H.323 [2] defined by ITU, or the SIP [3] defined by IETF. Current Internet telephony standards only consider basic call services and supplementary services supported by network elements (e.g. a proxy server in SIP and Gatekeepers in H.323). However, in order to introduce value-added services promptly and to give operators the ability to personalize their service offerings, a value-added service function should be incorporated with the Internet Telephony framework. In this thesis, I have defined a distributed messaging system that supports information exchange at the application level. Since the signalling control function is realized on top of the network/IP layer, such a messaging system can integrate major signalling stacks (i.e. H.323, SIP and SS7) from both Internet and Telecommunication domains. The signalling information wrapped up in distributed event service is in a pure text format. By separating the context and the underlying dispatching technology, it makes the whole signalling system easy to extend.

Some research projects had been carried out to define new types of service architecture based on IP technology. GbaGuidi has presented a service architecture which offers service across PSTN and the Internet [4]. His architecture focused on providing Gateways which connect two domains together. In the Artificial Intelligence and Network Management Lab of Carleton University, a service framework has been deployed based on H.323 using Mobile Agent technology [5]. The major contribution of this architecture is to realize an IP-based IN service with an H.323 signalling system using Mobile Agent technology. Service providers publish their service and end users can browse and choose features through a network using a JINI type of lookup service. Unlike the traditional IN approach, service providers can deploy new services faster and more economically using these frameworks. However, some issues are not covered in these proposals. My research attempts to consider the service provisioning issue from a different point of view. Suppose the service-enabling technologies mentioned above are ready and the new services are available, what kind of mechanism should be applied for distributing the services to customers? If the end user is interested in the features offered by different service providers, who should manage and organize these features for the end user and how can they be integrated? This thesis focused on providing solutions to these questions. The detailed design and implementation are presented in the following chapters.

In Chapter 2, I analyze the problems of traditional IN architecture in the IP world and the latest attempts that standard bodies have made to extend IN to fit IN/IP integration. In my analysis, I compare their pros, cons and limitations.

I describe my Open Service Architecture design in Chapter 3. My Open Service Architecture provides a unified framework to support IN-type service provisioning on an IP network.

In Chapter 4, I focus on the evaluation of available implementation technologies for each layer of the Open Service Architecture.

In Chapter 5, I present the implementation details for each layer of the Open Service Architecture. A new service called “unified call forwarding” is deployed as a pilot service to demonstrate the flexibility and extendibility of the Open Service Architecture.

Chapter 6 includes the summary and the future work of this study.

Chapter 2

PROBLEMS WITH TODAY'S SERVICE

ARCHITECTURE

2.1 Intelligent Network Concept

Today, the Intelligent Network (IN) is recognized as the standard architecture by telecom service providers to offer advanced services to customers. The intention of the intelligent network is to provide a means to meet the challenge of time-critical and complexity in new service development. The characteristic of the IN concept is that the logic for controlling calls and call-related services is separated from physical switching and call set-ups. [6] The IN is based on a simple principle: the service-specific software is separated from the basic call processing and is run on a specialized node which is called the Service Control Point (SCP), while the switches are called Service Switching Points (SSP). The Service Management System (SMS) contains functions to enable the management of the IN system. The Service Data Point (SDP) is a database that holds information related to the service (e.g. customer profile). The IN is provided with a Service Creation Environment (SCE) that contains service development tools (Figure 1). Communication among those elements is achieved through the Common Channel Signalling No.7 (CCS7). The pilot IN application is the freephone service. When the SSP detects the "1-800" pattern, it temporarily suspends the basic call procedure and requests the collaboration of the SCP. The SCP realizes that the communication fees must be charged to the called party by

analyzing the service stored in its database. Usually it asks the SDP for the physical address of this party.

Figure 1 Intelligent Network Structure

The response from the SDP is forwarded to the SSP which can then request a connection to be set up between the two parties. The development of the IN has been organized in several phases, each of them being characterized by a specific capability set. The earliest one is the Capability Set 1 (CS-1), which covers single-ended services. In CS-1, services are applied to one and only one party in a call and are independent of any other parties that may be participating in the call [7]. Examples are number translation services (e.g. call forward, or universal personal telecommunication), alternate billing services (e.g. credit card calling) and screening services (i.e. calling/called party screening, or security screening). The CS-1 is followed by two other phases (CS-2 and CS-3) which introduce more sophisticated capabilities. Since most of the capabilities defined in CS-2 and CS-3 are intended to support IN feature for mobile users, my research just focused on CS-1.

2.2 Constraint of IN for Open Service Deployment

Nowadays, IN is widely used in the telephony network by services like freephone, Global Virtual Network, and so on. In general, some of IN's strengths include:

- Legacy gained from the deployment of a large number of services.
- Service creation using a simple and small set of reusable Service Independent Building blocks (SIB).
- Independence of the service (and the control) from the network infrastructure.
- A thorough conceptual model for service creation which shows a gradual "shrinking" of a service's complexity across the model's plane.

The architectural and conceptual advantage of the IN solution is based on the following assumptions:

- The circuit-switching system is the backbone for the services.
- Voice is the major type of media source.
- The service only involves a centralized data source.

In the circuit switching system, the telephone number is the only identity attribute in a service profile. It is used not only to identify the type of service but also to indicate the origin of billing information (except for a credit card call service). The great advantage with the number-oriented approach is that it is easy to implement, but it also severely undermines the system's capability to handle complex services (i.e. multimedia and real time control). With the growth of the Internet, more and more people believe that packet-switching technology is going to replace circuit-switching technology in the next generation's transmission backbone. It is also estimated that in the future, voice will not be the only media type for telephony services. The situation with respect to services in the IP world is quite different to that on legacy circuit switches. The openness of service authoring, the ubiquity of access and devolvment, and scalable architecture have led to

unprecedented growth and creativity in the creation of third party services. It means we need a more powerful service-provisioning mechanism other than IN to define, locate and implement the services [8].

Another limitation of IN is that the IN architecture has poor capability to support the service with distributed data sources. IN's service distribution is achieved by a routing service to a centralized database system such as SCP or SDP. In the legacy switching system, the user data cannot be shared among switches. Let's assume a customer has subscribed to distinguished ringing and call forwarding features. When he/she gets a phone call transferred to a number located in another switch, the user cannot get a distinguished ringing tone no matter whether the local switch supports such services or not. The reason is that the user's service profile (feature list) is stored in the local switch and cannot be shared with other systems. This is easy to understand in today's business model, because the service provider is also the network operator in the real world. This means that the user service profile (user data, feature attribute and billing information) is hard-coded within its local switch. There is no standard interface defined between the service provider and the network operator to publicly access the service profile. This constraint has to be resolved if we want to integrate telephony services with those belonging to other domains, such as the Internet. In the web application, however, the data is stored in a distributed way and can be shared freely among applications. So the new service architecture should have unified the interfaces between service provider and network operator.

System openness and service integration are annoying issues for current service providers as well. Traditionally, IN-Platforms are closed systems. There never have been and still are no other standards for interconnecting IN-Platforms with other systems than

through SS7/INAP. Likewise, protocols for Management and Service Creation remain proprietary to individual vendors and platforms. However, telecommunications are now moving towards open systems, leading to a distributed/component-based service development based on open IT standards. Due to needs in rapid services development and service interaction, Telecom operators and manufacturers are investigating object and distributed technologies in the domains of activities such as network management, intelligent networks, services creation environments and information systems.

Implementation technology is also a major weakness in the traditional IN system. In the IN Concept Model (INCM), the service is composed of SIBs. SIBs are a group of software components which are neutral to specific features and can be wired together to provide specific services. Although SIBs are independent from the service, they are tightly coupled with underlying switching systems. The reuse theory of SIBs is not object-oriented but based on a kind of algorithmic (functional) approach. In INCM, SIBs behave like standard software modules dealing with Basic Call Processing (BCP) which provides primary switching functionality. As each vendor's implementation of BCP is quite different, there is little commonality available to be shared among SIBs. In the ITU-T CS-1 recommendation, thirteen SIBs are defined to support only twelve IN services and most of these are default features offered by the switch. Telecom systems must evolve to support software technologies that favour service assembly, such as object technologies (CORBA, DCOM/COM+, and EJB). IN service creation environments, which are currently quite closed, need to open further in order to support the integration of Internet applications [9]. The software architecture needs to enable a new service/application design focused on application functionality without being concerned with the complexities that arise from the

distribution of application components (i.e. details of message construction, recovery mechanisms, and so on). The application should be deployable using components provided by different suppliers (application interoperability) and the same service should be deployable for different switching and transport technologies (technology independence). The architecture should support the use of standard programming languages like C, C++, Java etc. for service creation and execution. The location of the distributed services must be transparent to both the user and the service creator.

Today, the usage of IN services is relatively rare compared to the total number of calls. In this respect the current IN architecture with centralized SCP/SMS (Service Management System) is appropriate for supporting the traffic. However, in the future a strong increase in the demand for IN-intensive services is foreseen, especially for IP-based services. Hence it would be necessary to change the classic centralized IN architecture in order to introduce distributed processing mechanisms on a geographical scale. The telecom systems represent a huge investment and must imperatively stay in operation, together with the equipment of the new generation within the new service framework.

2.3. Open Service Framework

In order to improve the adaptability of the traditional IN and to accommodate the Internet into future service architectures, several service architectures have been deployed by industry organizations for public review. The most influential of these include:

1. PINT (PSTN/Internet Interface), defined by IETF;
2. TIPHON, defined by ETSI;
3. JAIN, defined by Sun Microsystems.

2.3.1 PINT

Created in 1997, the PINT working group is part of the IETF Transport Area. It addresses connection arrangements, through which Internet applications can request and enrich PSTN services. It proposed a framework in the form of an IETF specification [10]. The PINT protocol is an enhancement of SIP and SDP protocols; it systematically uses IN as a means to unite the PSTN and the Internet.

Figure 2 PINT Architecture

PINT defines an IP-based infrastructure on top of the traditional POTS (Plain Old Telecommunication Service) network to enable Web-based access and service activation. Sets of interfaces have been defined among the components of the PINT reference model. Through such interfaces, the PINT service is able to interconnect with applications under other standard bodies such as IETF and ITU-T. The PSTN users depicted connect to both

the central office via a wireline and the mobile switching centre via wireless communications.

Figure 2 shows the major components in the PINT architecture. A PINT system is composed of three basic elements: PINT Client (SIP user agent), PINT Gateway (SIP user agent server) and an executive system. According to the PINT architecture, “PINT clients and servers are SIP clients and servers. SIP is used to carry the request over the IP network to the correct PINT server in a secure and reliable manner”. [11] The PINT server has two interfaces to underlying resources: an interface to the SN and an interface to the SMS. The first interface is based on TCP/IP, and the latter one is based on Simple Network Management Protocol (SNMP).

The PINT protocol enables the invocation of telephony services from terminals in an IP-based network environment. More specifically, a host in the IP network forwards the service request to a PINT server, which relays the request to the relevant PSTN network resource, such as an SCP node.

The SCP then executes the requested service, possibly reporting the service session status back to the originating IP terminal. The information exchange is achieved based on the IETF standard Session Initiation Protocol (SIP) and the associated standard Session Description Protocol (SDP). While the overall aim of the PINT initiative is to enable Internet resources and telephony services in broad terms, it will effectively standardize the access from the Internet to the IN domain and enable development of novel services that are executed across the traditional telephony domain and the Internet. Another consequence is that due to the reuse of standard IETF protocols and methodologies, the solution provided will be very lightweight in nature. Importantly, the PINT protocol fits into the

existing SIP architecture for Internet-based media session control, which will be significant in the future if SIP forms the basis for IP telephony. In addition, PINT is being actively considered by ITU-T SG11 for inclusion in the IN CS-4 functional architecture.

2.3.2 ETSI TIPHON

The provision of voiceband services in technologies other than circuit switching has been growing over recent years, with more efforts on providing such services in packet switching infrastructures using the Internet Protocol. Today, the main IP telephony architecture is based on standards developed by the ITU-T Study Group 16 (SG16). This architecture is described in the H.323 Recommendation, which defines a multimedia or video-conferencing terminal over IP style networks. In parallel to ITU-T, several other organizations, such as ETSI (Europe Telecommunication Standard Institute) and IMTC (International Multimedia Teleconferencing Consortium), have proposed service architectures based on H.323. The IMTC VoIP (Voice Over IP Forum) produced an Implementation Agreements 1.0 or profile for H.323. The ETSI TIPHON project focuses on gateways between H.323 and SCNs (Switched Circuit Networks, such as the PSTN, ISDN and GSM) [12]. Up to now, TIPHON has covered the widest scope of issues for IP Telephony among these frameworks. The objective of the ETSI Project, "Telecommunications and Internet Protocol Harmonisation Over Networks" (TIPHON), is to support the market for real-time telecommunication services, including voice and related voiceband communication over multiple network technologies. TIPHON addresses the service-level inter-working between traditional Switched Circuit Networks (SCNs) and the emerging Next Generation Networks (NGN) based on VoIP. TIPHON enables users connected to IP-based networks to communicate between themselves and also with users in

SCNs, especially those served by PSTN, ISDN or GSM networks. In this regard TIPHON addresses the difficult, yet extremely important, area of multi-network inter-working across multiple administrative and technology domains. In doing so, it is considering a broad and diverse set of requirements including security, quality of service, numbering and the very approach to communications standardization itself.

The TIPHON architecture consists of three main functional blocks: terminals, gateways, and gatekeepers (See Figure 3).

Figure 3 TIPHON Basic Call Reference Model

3.2.1 Terminal:

A terminal is a leaf node in the TIPHON reference configuration. It is connected to an IP network via some form of IP access. A terminal enables a user to make telephone calls to a user in the SCN. Such calls should be under the supervision of the Gatekeeper with which the user or terminal is registered.

3.2.2 GateKeeper:

The gatekeeper is the element in the network that is responsible for the Registration, Admission and Status (RAS) of terminals and gateways. The gatekeeper participates in zone management, call processing and call signalling.

3.2.3 Gateway:

A gateway is physically connected to one or more IP networks and to one or more SCN networks. A gateway is composed of:

- **Signalling Gateway;**
- **Media Gateway;**
- **Media Gateway Controller.**

One or more of these different functions can be co-located and may also be combined with gatekeepers or with other gateways.

The Signalling Gateway provides a signalling mediation function between the IP and the SCN world. It supports functional or signalling mediation between the IP domain (e.g. H.323) and call signalling in the SCN domain (e.g. SS7).

The Media Gateway provides the media mapping and/or transcoding functions. It maps or transcodes the media in the IP domain (e.g. media transported over UDP/TCP) and media in the SCN domain (e.g. PCM encoded voice etc.). The Media Gateway Controller sits between the Media Gateway, the Signalling Gateway and the Gatekeeper. It provides the call processing (call handling) function for the Gateways. It controls the Media Gateways: it receives SCN signalling information from the Signalling Gateway and IP signalling from the Gatekeeper.

Back-end services may be used by gateways and gatekeepers to provide functionalities (e.g. authentication, billing and rating/tariffing, or address resolution etc.).

These back-end services may be provided by equipment within the SCN, within the IP Network, or elsewhere.

Figure 4 Architecture of TIPHON services

To many observers TIPHON is viewed as a Voice over Internet Protocol (VoIP) project. In the development of VoIP, two bodies have been developing protocols:

- IETF, developing SIP, SIGTRAN, MPLS, IPTel, etc.
- ITU, developing H.323

Neither of these bodies has yet taken a significant lead in the market. However, recent developments in 3GPP suggest that SIP will be the protocol of choice for UMTS (Universal Mobile Telecommunications System) with an IPv6 backbone. For TIPHON the goal is rather broader: it has to support both H.323 and SIP, it has to support ISDN and PSTN in all its variants, and it has to support any radio access technique including DECT, [13] GSM, [14] 3GPP [15] and TETRA [16].

The aspect of TIPHON that makes it independent of the specific capabilities of networks is the development of service capability definitions which lead to the

specifications of service capabilities. The capabilities are defined as visible at particular points of reference.

The picture presented in Figure 4 offers a simple view in which a terminal accesses the TIPHON service through an access network, and the TIPHON service is completed by combining the capabilities of a number of access and transit networks through a network service layer.

TIPHON has addressed the challenge of providing public communication services in a heterogeneous environment by defining a generic means of creating services that is independent of any specific underlying network technology - irrespective of whether it is switched circuit- or packet-based. To achieve this objective, TIPHON has identified the need for an overarching technology and domain-independent protocol framework, known as the TIPHON meta-protocol. This is used to generate profiles for the protocols associated with any given communications network technology (e.g. including H.323, SIP, H.248 etc.). By mapping this 'meta-protocol' to individual network technologies, TIPHON ensures a higher degree of end-to-end capability.

2.3.3 JAIN

Java Advanced Intelligent Network (JAIN) is a service provider's extension to the Java platform. It addresses the needs of next-generation telecom networks by developing a set of industry-defined APIs for Integrated Networks [17]. Members of the JAIN community have joined forces to define open APIs based on Sun's Java platform, thus allowing service providers to rapidly create and deploy new flexible, revenue-generating services. By providing a new level of abstraction and associated Java interfaces for service creation across heterogeneous network, JAIN technology enables the integration of Internet

(IP) and Intelligent Network (IN) protocols. This shifts the telecommunications market from many proprietary closed networks towards a unified network architecture. This allows services to be created and deployed rapidly. Since JAIN allows Java applications to have secure access to resources inside the network, it is possible to deliver thousands of services rather than the dozens currently available.

2.3.3.1 JAIN Architecture

JAIN provides a new breed of technologies to solve the fundamental differences between telecommunication and the Internet world. JAIN defines a unified framework to integrate services from different domains (i.e. wireline, wireless, Internet etc.). The JAIN initiative consists of two API Specification areas of development, which include Protocol Layer and Application Layer (See Figure 5). The Protocol Layer proposes a generic framework that new types of protocol can be plugged into seamlessly. It defines a series of Protocol Adapters that encapsulate the underlying resources into Java methods and interfaces.

Figure 5 JAIN architecture

The Protocol Adapters support common information exchange with other protocol stacks. For example, through an H.323 protocol adapter, information from other protocols (i.e. SS7 or SIP) can be translated into the H.323 format. Other interfaces defined in the JAIN Protocol Layer include:

- **JAIN Session Initiation Protocol (SIP) API:** addresses the call setup and teardown mechanisms for IP-based networks.
- **JAIN Media Gateway Controller Protocol (MGCP) API:** controls voice and media over packet gateways.
- **JAIN Mobile Application Protocol (MAP) API:** manages the Pan European and North American standards for cellular processing.
- **JAIN Intelligent Network Application Protocol (INAP):** defines the Java technology-based standard API extension for Intelligent Network applications, based on the ANSI/Telcordia AIN and ITU-T CS-1 INAP specifications.
- **JAIN ISDN User Part (ISUP):** provides an API for the signalling functions needed to support switched voice and data applications

The initiative of JAIN is to bring service portability, convergence, and secure network access to telephony and data networks, thus simplifying network service delivery for service providers [18]. The application layer abstracts the protocols covered by protocol layer into a single call control, coordination, and transaction model which supports JAIN compliant services. The fundamental idea is to provide a unified call-processing interface, helping software developers to write a single service application which involves multiparty, multimedia and multi-protocol sessions. The application layer also provides a secure access mechanism to the network capabilities through JAIN Secure Network Access APIs.

Figure 6 demonstrates the three basic abstractions supported by JAIN, which include protocols on the protocol layer, trusted services and untrusted services on the application layer.

In protocol layer, JAIN adapters are deployed to fill the gap between the applications and the supporting protocols. The adapters encapsulate the underlying resource through Java computing technologies such as class methods, callbacks, events or Java interfaces. The resources are conformant on API level rather than implementation languages. Through JAIN API, an application can run on top of protocol products from different vendors.

On the application layer, JAIN defines the application interfaces available to the service creator. Since the interfaces do not describe how the network provides these capabilities, they are architecture neutral.

Figure 6 JAIN Abstraction

The application APIs provide service developers – whether independent software vendors, service providers or network equipment providers – with a common model and a consistent run-time environment for developing and deploying network services in Java. The JAIN Call Control (JCC) API and the JAIN Coordination and Transactions (JCAT) within the JAIN community are defining such a call control/session interface. The objective of the JCC and JCAT APIs are to provide applications with a consistent mechanism for call control and call processing transactions. JCC is a unified call model that incorporates the basic features of the Java Telephony API (JTAPI) as well as Parlay's Call Control Service, and provides an extensible Java framework for supporting complex call processing applications. JCAT extends the JCC call control model with terminal capabilities and it enriches JCC's state diagrams so that an even more diverse range of applications can be supported, such as the AIN/IN class of applications. Services which conform to JCC/JCAT are called trusted services. The vendor's proprietary interfaces have been masked through protocol adapters.

JAIN provides a third level of abstraction through the JAIN Parlay interface. Parlay is a technology-independent API for telecom services which allows applications to access the functions of the network and generic support functions, in a secure way [19]. Because Parlay API enables applications residing outside the network to access and direct network resources and devices to carry out specific actions or functions, a new and open environment is created for developers and users. Faster time-to-market and a less complex development cycle are some of the key benefits of the Parlay APIs.

In Figure 6, the small disks inside the untrusted service plane present JAIN-Parlay-based services. JAIN Parlay acts as a firewall to integrate trust and untrusted services in a

secure way. [20]. It benefits network operators by creating additional services, while still maintaining the security and integrity of the network. It benefits service providers since new applications and services can be distributed to end users much more easily through the Internet. It enables application developers to create applications that are portable between networks. It also benefits end users by offering a much wider set of network applications.

2.3.3.2 JAIN Service Logic Execution Environment

JAIN defines a JavaBeans-based component factory that brings convergence, public access, and service portability into the telephony marketplace. JAIN architecture provides a carrier-grade Service Logic Execution Environment (SLEE) for trusted next-generation network services. It is expected that many services and implementations of the JAIN SLEE will be implemented with Enterprise JavaBeans. Untrusted services, like trusted services, might have similar requirements for the SLEE, depending on the scope.

JAIN's SLEE is based on the Enterprise JavaBean (EJB) component model, providing services for transactions, persistence, load balancing, security, object and connection instance pooling, and so on. The JAIN architecture leverages the platform independence of Java technology. Based on the EJB component model, JAIN can be integrated with most of the distributed computing middleware infrastructure (e.g., Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), and Distributed Common Object Model (DCOM)). The JAIN architecture can also be extended by integrating many existing and on-going development efforts in the Java programming library space. Some of the most important related initiatives are developing (or have developed) a Java technology-based encapsulation to database access (JDBC), access to

naming and directory services (JNDI) and a Java Transaction Service (JTS) based on CORBA's Object Transaction Service.

2.3.4 Limitation of existing Open Service Architecture

The architectures presented above try, from different perspectives, to bring together the advantages of the Internet and the telecommunication world. Much research has been carried out and some service architectures have been derived from the above three major frameworks. The PINT approach only devolves part of the request mechanisms to be run on the Internet while still using the PSTN as a transport infrastructure. The new services proposed are traditional PSTN services integrated with the Internet (such as Web-based voicemail, etc.). From a more holistic viewpoint, it can be argued that the PINT is quite limited in its scope – it prevents IP hosts from participating directly in call control. In the meantime, clear interface specifications between framework elements are still ambiguous so that it undermines large-scale implementation in the real world. A potentially significant drawback of the PINT work is that no standard API will be defined as part of the IETF process. This will either lead to an emergence of a *de facto* standard API, or multitude vendor-specific implementations with resultant porting difficulty for code to run with different products. This may hinder the development of PINT as a basic service architecture for IP telephony.

The TIPHON solution alleviates the PSTN as the principal infrastructure for the transportation of user data, and promotes the Internet as a serious alternative for conveying this data. TIPHON is best described as a capable network environment. It offers capabilities that serve to support the capabilities of terminals while at the same time ensuring that business needs are met. The latter really means the ability to sell and resell

network side capabilities, to sell and re-sell terminals, and to ultimately generate revenues. It is not true to say that TIPHON is a network, nor is it true to say that TIPHON is a system. It is an environment. This environment goes beyond IP but is in many ways a candidate for a new generation of interworking networks that uses the best of IP and the best of OSI. It does that by simplifying the knowledge required of the lower layers (extending IP), and by making the upper layers less obscure (simplification of OSI). [21]

Although TIPHON takes advantage of the flexibility and scalability of the Internet, it is successful only on the transport layer. The service deployment process is still the same as the traditional IN solution. There is no clear definition of how to provide third-party services and how to distribute control intelligence through the network.

The JAIN architecture provides more opportunities to solve problems and meet challenges in the rapidly-changing telecommunications environment than the other two solutions. JAIN not only defines the service composition process but also provides standard switching capability (BCP in the IN model). Furthermore, the unified implementation technology with JAVA also improves the application's inter-operability and reduces feature inter-dependency. Whilst being a sound architecture, JAIN only defines a mechanism to integrate different vendors' capability components (signal stack and basic call processing). There is no clear description of how to integrate the service logics of various vendors. In JAIN, communication between different components is realized through the JAVA event service. Event service is only applicable among applications on the same Java virtual machine (i.e. JTAPI event). How to provide standard communication mechanism across JVMs is still an open issue for JAIN architecture.

Chapter 3

DESIGN OF OPEN SERVICE ARCHITECTURE

In order to take advantage of the Internet's capability to distribute service intelligence, a flexible framework should be developed to facilitate and manage the use of intelligence on the IP-based transmission backbone. As mentioned in the previous chapter, the service provisioning frameworks defined by industry groups do not have enough capability to support fast/open service deployment. A number of research projects have been proposed to try and deploy the more powerful service provisioning frameworks. Unfortunately, their efforts have not gone very far compared to the definitions provided by industry organizations.

In Teltec's research project [22], a framework which extends the PINT architecture was defined. The solution is to add a CORBA-based middleware layer in order to replace the PINT interfaces with SN and SCP. This approach provided more adaptive capability for bridging the IN and Internet domains, but it still makes use of the Internet just for service access.

Lucent's INSeCT (IN Services for Converged Internet Telephony) project [23] defines an IP-based soft SSP which possesses interfaces conforming to the current circuit switching system. Since there is no new requirement for SCP, it can transparently integrate a legacy IN system into the packet switching system. Although INSeCT supports multiple signalling stacks (SIP and H.323), it does not define how to support multiple service providers because there is no clear explanation of the inter-connection between the soft SSPs of different vendors.

There are other service provisioning projects based on JAIN architecture, such as ChaiTime [24]. ChaiTime has defined an open service framework based on JTAPI. It supports multi-service providers and the new service can be dynamically integrated with existing features. Although ChaiTime defines the process of integrating new services, it does not specify how to organize and manage services from different vendors. Since the end-user gets the service directly from the service provider, it is difficult to manage the same kind of service from various providers. Most of the processes introduced in ChaiTime for new service integration are to add new processing capability to the client application, in order to support new types of media.

Based on my introduction in Chapter 2, the primary challenge for multi-vendor service provisioning is how to manage services for end-users and how to allocate the service capability through the network (i.e. which part of the service processing capability should reside with the end-user and which part should be kept by service provider, and how these different parts should be coordinated).

In this chapter, I focus on presenting my design for an open service architecture that provides value-added service features. The work focuses on two aspects:

- **A flexible framework which supports multi-party and open access for service provisioning on an IP network (service provisioning and service management) and data sharing among service providers.**
- **Using advanced software engineering technology to compose the software components and integrate them together with distributed computing technology.**

In the following sections, a detailed introduction will be presented, focusing on the structure of a service framework and the essential issues which need to be solved.

3.1 Components Definition

Before introducing the system architecture, it is necessary to provide some definitions about the players in Open System Architecture and their role in the service provisioning process.

- **Service**

A service is an application that is developed by service providers and provides specific functionality for customers to use (e.g. Call Forwarding).

- **Feature**

A feature is the granular entity of a service. Features are deployed using component technology and do not contain service-specific implementation, so they could be reused to compose different services. For example, Call Waiting is a service, the process of which can be divided into the following steps:

- **Inquire into the state of the called party.**
- **Screen if called party has call-waiting service.**
- **Hold the incoming call.**
- **Notify called party that there is a new incoming call (alerting).**
- **Drop one call based on called party's choice (take the new call or keep the old call).**

In this example, "inquire", "screen", "hold", "notify" and "drop" are generic actions occurring in every call, and they are called features.

- **Service Provider**

A Service Provider is the vendor/developer who deploys a specific functionality (i.e. call forwarding) and offers it to end users. The service provider not only owns the knowledge property (i.e. application software) but is also responsible for setting up the underlying resource required by the feature (i.e. call processing server and data storage system).

- **Third-Party Service Provider**

The Third-party Service Provider (TSP) develops the service by integrating services from one or several service providers, or making value-added improvements over and above existing functionality. A TSP does not own the features integrated in its service and does not need to set up the underlying resources required by those features.

- **Service Carrier**
A Service Carrier offers capability to access the network to end users. The Service Carrier is responsible for setting up the network infrastructure and connecting a Service Provider to an end user. The Service Carrier holds specific services chosen from different Service Providers. These services are stored in the form of runtime executables in the Service Carrier's Service Node. The Service Carrier categorizes services into different subjects, helping End Users to apply these services more easily.
- **Service Logic**
A Service Logic refers to the control entities in a service. It manages the state machine of the service (i.e. responding to an end user's state change – if an end user is off-hook, for example) and instructs the capability components to take action for the request (i.e. tells the ringing resource to send a busy tone when a called party is busy).
- **Service-related Data**
Service-related data refer to the information related to a specific service. This type of data is usually dynamic and includes the state of each party in every stage of the service period, underlying resource readiness as well as the billing information, and so on.
- **User profile**
The User profile contains information for a specific user. The user profile is usually static data, including a list of the services to which a user has subscribed, identity information, and financial data.
- **Feature repository**
A feature repository is a software component pool in which each component executes a specific functionality. In a Feature Repository, software components may come from different service providers or independent software developers (i.e. Oracle may not provide any service but it develops an application that can be used to inquire on data in an Oracle database). A feature repository provides deployment phase reuse by integrating feature components from sources.
- **Service Repository**

A Service Repository is owned by the Service Provider and serves as an application pool to carry all the services developed by the Service Provider. The Service Repository provides an interface for the Service Provider to publish its services and help other Service Providers and Service Carriers to access these services.

- **Service Manager**

The Service Manager is the management entity of the Service Repository. It coordinates the service in multiple ServiceProxy Repositories and synchronizes the code with the service in the Service Repository.

- **Service Node**

A Service Node is the processing entity in the Service Carrier, providing the resource capabilities required by the end users to set up the service. As the Service Carrier may contain heterogeneous service provisioning infrastructure (e.g. traditional IN, IP Telephony based on SIP or H.323), different types of the Service Node are required to support the various service scenarios. It could be the SCP in the Intelligent Network, the PINT server in the PINT infrastructure or the GateKeeper in TIPPHON. The Service Node also maintains a ServiceProxy Repository which provides a runtime level of reuse to integrate services from different service providers.

- **ServiceProxy Repository**

A ServiceProxy Repository contains the runtime executables supported by the Service Carrier. The executables are the runtime format of the service and are downloaded from the Service Repository when the Service Carrier chooses to support the service. The repository is implemented based on an auto-pluggable framework which could be dynamically extended to add new services. Runtime executables are categorized in the repository and can be activated dynamically, using Java Reflection APIs.

3.2 System Architecture

In my service model, a “user” is no longer an entity which is local to one specific machine. Each user is a machine-independent entity that exists in the network and can transparently carry its profile (user information and feature lists) across hardware and

software platforms. The network-centric, reflective service management system manages and allocates the resources to support the user on the selected platforms. Software components are distributed over the network to where they are required. Although there are some mobile code-centric service provisioning systems deployed in practice [25], most are simple client-server applications, such as ChaiTime, which was mentioned in the previous section. End users download and configure the client code and connect to the server to get service. Under such a mechanism, distribution is only achieved for the user interaction part while the service logic control and the data management are still centralized. All user profile and control processes are located on the server side and it is difficult to share them with other servers. In fact, such an approach is no different to what a traditional circuit switching system does – it only treats the Internet as a different type of transmission media.

In my solution architecture, the service is deployed through a four-layered framework (Figure 7) which includes:

- Service Layer
- Adaptive Layer
- Network Layer
- End-User Layer

Figure 7 Open Service Architecture

Service capabilities (i.e. call setup and routing) are realized through service nodes in the network layer. The service-related data (i.e. data from other service providers and billing information) are stored in the service layer, while end-user related data (user profile and state) are kept in the service node. In the service layer, each Service Provider manages its services through a Service Repository. In Figure 7, the small disks in each service represent Feature Components and the different colors are used to distinguish the ownership. Feature Components are reusable building blocks which are deployed in the form of software components and can be assembled to offer a specific service. The Service Logic (the black lines connecting feature components together in Figure 7) in the service

layer controls the behaviour of the service nodes in the network layer through distributed event services provided by the adaptive layer. Since the service provider publishes its services in multiple network nodes, a configurable, policy-based service management mechanism is deployed to manage the service in each network node (update application and optimize resource allocation).

3.3 Components of the Open Service Architecture

3.3.1 End User Layer

The End User Layer is where the End User Terminal (EUT) resides. The EUT integrates traditional CPE (Customer Premise Equipment, such as the telephone set) and an Intelligent Terminal such as a computer or a PDA (Personal Digital Assist). This layer provides an access interface for the end user. Compared with traditional end user equipment, the EUT may have its own processor which brings more intelligence to the device. In the world of traditional telecommunications, each type of device is hard coded with a specific protocol/solution. In order to introduce a new protocol, new hardware has to be developed and installed. This is quite different to the rule in the computer world. The EUT deals with the new service in a “computer way”, which could greatly simplify the introduction of a new type of service/protocol. When a new feature is introduced, the EUT just downloads the code from the service provider and reconfigures (adds new features to) the local application software.

3.3.2 Network layer

The Network layer contains the network (public and private) and special resources managed by the control components in the service layer. It includes the service nodes in LANs within enterprises, Intranets at ISPs and legacy IN from telcos. The service node

provides resource capabilities in the network layer. Examples of network resources (in the private or public environment) are:

- **SSPs that implement a BCSM (CS-1 or CS-2) and related signalling system (INAP-interface etc.)**
- **SCPs where IN-services are executed. An SCP could act as a gateway between the SSP and the service platform (in such a case the service platform should not be aware of signalling- (ISUP) or INAP-specific information).**
- **Flexible switching systems (e.g. PBX, programmable switches, service nodes, multi-conference units) that provide some advanced features to control switched communications (e.g. third party call/connection control, audio/video bridges);**
- **Network Access Servers (NAS) which provide authentication to the end user;**
- **Signal Servers (SIP or H.323) to send invitations to IP terminals (SIP client);**

The Network Layer provides service capability required by the end users, such as switching capability for telephony service and code delivery from service provider to end user. It supports multiple types of signalling standards which are either the legacy (SS7) or the niche for the future (H.323, SIP). The Network Layer is both multi-service and service-independent if we look at it from different points of view. From an end user's point of view, the Network Layer carries services deployed by different service providers, so it is a multi-service platform. From a service provider's point of view, applications on the service layer talk to those on the network layer through a unified communication mechanism (distributed event service) and any service in the service repository could be integrated into the ServiceProxy Repository, so that the network layer is service independent. The network layer provides the packet-level transportation, from end-to-end, for all services that are deployed through the higher layer protocols and software. It is an open service runtime environment in which multiple vendors' features can be integrated. Although the network layer is the place where the service is actually running, the owner of this layer does not

really implement the services, as the current Telcos do. It just provides access and transmission capability for the end user as well as a pluggable framework – a ServiceProxy Repository to hold the features of multiple service providers. The role of the network layer is something similar to that of a television cable company, like Rogers. Programming is produced by media giants such as CNN and MGM, which are called content providers. Rogers then sets up a distribution channel to help customers access the content. The opening-up of the local carrier to service providers will completely change the rule of the current telecommunications market. End users should be able to choose services from several (IN) service providers so that they are no longer limited by what the local carrier offers. This broadens the service spectrum as several different services or equivalent services with varying quality and price are offered. In the meantime, since service providers do not need to build up the access or transmission system to reach end users, it can greatly shorten the time frame for introducing new services into the market.

3.3.3 Adaptive Layer

The adaptive layer is a middleware layer. It provides communication capability for service providers/components to talk to their peers on the service layer (i.e. query the end user's state and exchange billing information) or to the underlying service nodes on the network layer (update and backup user profile). The information is sent/received in the form of an event which provides a unified interface in a multi-vendor environment. Resource adapters, or wrappers, are used to abstract the upper layer from the details of specific protocols (e.g. INAP, H.323).

Therefore, the adaptive layer allows in principle the independence of the service platform from the underlying layers. It ensures the seamless integration of different

signalling domains into the network layer. The adaptive layer provides the entity in the service layer with a unified communication interface to manage the underlying service nodes in different domains. It helps end users locate desired features from the service providers. The communication mechanism in the adaptive layer supports both point-to-point and push/pull types of message transmission. The adaptation can be implemented by the same resource (wrappers), or by external servers (external adapters). In the former case, the resource is offered directly to the service platform (call control or message control) through an open API. In the latter, an adaptation unit is required to map the standard/proprietary protocol onto an open API (i.e. IDL/RMI interface) accessible from the service platform.

3.3.4 Service Layer

The service layer contains a collection of functionalities that provide runtime management and control for the services running in the network layer. The composition and interaction of feature components allow the designer to build value-added telecommunication services, while being abstracted away from the underlying network and special resources. In addition, it provides “service features” to services and applications in the external domain.

Services are deployed by various service providers in the service layer. They could be traditional Telcos, ISP or ISVs (Independent Software Vendors). Services can be offered/sold to either end users or to third party service providers. The components in the service layer include the service repository, feature repository, service logic control point and message control point (Figure 8). The service repository contains a set of services which are deployed by the service provider. The services developed by each service

provider are put into its own service repository. A feature repository is a feature bank managed by a service provider. It offers a common call control interface to all services and provides mechanisms for controlling calls across heterogeneous networks (i.e. call forwarding and call waiting). The features in the feature repository are granular software components.

Figure 8 Service Repository Components

They only implement the primary activities in a service – for example fulfillment, assurance and billing – and all of the myriads of small details of which they are comprised are implemented in the network layer. Since no service- and platform-dependent information is defined in the features, they could be reused to deploy different services. As mentioned above, the adaptive layer offers a unified communication mechanism between feature components, so that the service can be developed by integrating features coming from different service providers. A Service Logic Control Point (SLCP) is the intelligent part of the service layer. It defines how a service works and how it reacts to the internal and external stimuli, such as an internal timer expiring or a new TCP connection being established. In the classic view, the service logic is a state machine reacting to the actions,

events and transitions that guide the execution of the service. The service providers can composite new functionality by altering the service logic in the SLCP.

Message Control is an interface in the adaptive layer and provides an abstract interface for SLCP to control the operation of the service dealing with the underlying layers. As the messages are defined in the interface level instead of the content level, it allows third parties (e.g. service providers, various types of consumers) to have access and use the functionality (Service Repository) provided by the service platform and the underlying transport network. It provides services with a generic messaging API to control/access the underlying resources. Adaptation is also needed here to provide both synchronous and asynchronous communication mechanisms.

The Open Service Architecture achieves two levels of software reuse capability. One is in the deployment phase, while the other is in the runtime phase. The feature repository provides deployment phase reuse through a component-based service composition with feature components. Such an approach benefits service providers by simplifying the service provisioning process and saving on development time. Once the service is developed, all the feature components are wrapped up by the service and cannot be seen by other service providers/end users. The developed services are managed by each service provider in its service repository and can be reused by other service providers, including the third party service provider. For example, using the Call Waiting and Internet Phone services, we can develop a new service called Internet Call Waiting. Unlike the feature components reuse, building a new service with off-the-shelf functions in the service repository is deployed at runtime. It is achieved through an auto-pluggable framework in the network layer.

Once the service provider has deployed a new service, it needs to distribute the service to end users. Much research has been carried out in an attempt to solve this problem. One of the more efficient solutions is through the JINI type of lookup service which is defined in [26]. My research work does not focus on how to distribute the new service, so this issue is not covered in this thesis. More specifically, my research focuses on how to control/manage services from different vendors (i.e. where to put the executable and how to coordinate service activation at runtime). Regarding the service location and discovery process, I have assumed that those distribution mechanisms are fully developed and ready-to-use. The Service Provider publishes the service in its service repository and any interested service carrier (telecos or ISP) in the network layer can request the service (again, I have assumed that they can use either the JINI lookup service or other available technologies). When a service carrier requests a service, it downloads the service from a specific service repository as a runtime application and puts the application into its local ServiceProxy repository. Meanwhile, the service carrier updates its local service list to include the new service. During the service execution, there are two scenarios which will cause the ServiceProxy repository to load the executables from the service provider. One is the service carrier subscribes to the service, as discussed above. Another scenario occurs when a service involves a function that is not supported by the current service carrier. For example, a service provider deploys an Internet Call Waiting service by integrating the Internet Call and Call Waiting services. When an end user requires this service, the service node finds that the Internet Call is now not supported on its own. In this situation, it asks the service carrier to download the service into its ServiceProxy repository in order to continue the service process.

As the service provider distributes its service to multiple service nodes and each copy of the application has its own state- and service-related data, the service repository needs to provide the management with the functionality to coordinate interaction between the service repository and the ServiceProxy repository, and the behaviours within the Service Repository as well. In the service repository, a service manager is deployed to deal with the management issues for each service, such as the code update distribution and synchronization, etc. Overseen by the service manager, all service processing features (call setup, resource allocation etc.) are covered by the service node (because this can reduce the network traffic), while the service management features (e.g. billing and code update) are handled in the Service Layer (otherwise, it causes synchronization problems).

While a similar service provisioning mechanism has been defined in [26], my design work focuses on the different aspects of the service provisioning process. In the previous project, the design focused on service discovery and distribution (i.e. the JINI type of lookup service and code delivery with a mobile agent), but this work is focussed on deployment procedure. In my thesis, I have developed bean-oriented service components and the composition mechanism with a distributed event service. I also defined an automatic service activation framework through Java reflection APIs.

Chapter 4

IMPLEMENTATION TECHNOLOGY

This chapter focuses on the evaluation of the candidate implementation technologies for each layer of my Open Service Architecture. It covers issues such as functionality, flexibility and the succession for future deployment.

4.1 EUT implementation technology

The End User Terminal (EUT) is the direct interface for end-users. It provides the standard (unified) interface to the upper layer and various capabilities to different end-user access scenario. More importantly, it should define the standard interconnection mechanism between legacy telephony applications and new computer applications. Telephony APIs are the niche applications deployed by industry organizations as the solution for breaking the barrier between telephone companies and telephone hardware vendors [27]. It provides a framework for application developers to quickly integrate common telephone services such as call forwarding, voicemail and so on into their other software applications. The major roles in this field include:

- **Telephony Services application programming interface (TAPI).**
This call-control environment was introduced by Microsoft and Intel. It gives WinTel-based applications the ability to integrate with telephony sub-systems.
- **Java Telephony API (JTAPI)**
JTAPI is an API layer interface which Sun Microsystems provided for their Java application environment. It gives Java applets and applications the ability to access and control World Wide Web-based telephony resources.
- **Signal Computing Signal Architecture (SCSA)**

SCSA uses a bus standard to interconnect multiple telephony resources within a PC platform. It is a hardware/software standard specifying how different vendors' telephony resources can be interconnected via a pulse code modulated time division multiplexed bus.

In the following paragraph, I would like to computer these computer telephony standards which are currently regarded as the best and most momentous. My evaluation focuses on their technical approaches as well as the pros and cons with integrating them into the service architecture proposed in this thesis.

4.1.1 Microsoft's Telephony API

The Windows Telephony API is a part of the Windows Open Services Architecture (WOSA), which provides a single set of open-ended interfaces to enterprise computing services. TAPI consists of two interfaces. One is the Application Programming Interface, which is used by developers to deploy call processing capabilities. The other is the service provider interface (SPI), which is used to establish the connection to the specific telephony sub-system [28]. Figure 9 shows the architecture of TAPI.

Figure 9 TAPI Architecture

The call control functionalities in the TAPI architecture include origination, disconnection and answering of telephone calls, and so on. Applications can combine the

TAPI with other Windows capabilities (i.e. Media Player) to provide a combination of services.

4.1.1.1 TAPI and its Call Control Architectures

TAPI has introduced two types of call control mode in its two major releases respectively (TAPI1.0 and TAPI2.0). The TAPI 1.0 call control architecture is based on the first party call control. In the first party call control, the telephone set is controlled through a direct link to the client PC platform with which it is associated. This is considered to be a PC-centric architecture.

The alternative to this approach is known as the third party call control, which is deployed in TAPI 2.0. In this approach, there is a logical association between the client PC and the associated telephone set. In a third party call control environment a telephony server plays the role of a gateway interconnecting the PC application and the telephony sub-system. The telephony server accepts client PC applications requests like originating, disconnecting or answering calls and passes them to the telephony sub-system. Responses and events from the telephony sub-system sent to the telephony server are similarly passed to the respective client PC applications. Figure 10 shows the standard configuration of the third party call control environment.

Figure 10 Third Party Call Control In TAPI

This approach has a considerable advantage as the client PC platform does not need any special hardware upgrades to run computer telephony applications. Typically the only upgrade to the client PC is client-side system software which allows communications with the telephony server. An important concept relating to third party call control is that there is a logical association between the client PC and the associated telephone set.

The latest version of TAPI is TAPI 3.0, which is one of the key packages within Windows 2000. TAPI 3.0 integrates multimedia stream control with legacy telephony. Additionally, it is an evolution of the TAPI 2.x API for the COM model, allowing TAPI applications to be written in any language, such as C/C++ or Microsoft Visual Basic.

Besides supporting classic telephony providers, TAPI 3.0 supports Internet Telephony standards such as H.323 conferencing and IP multicast conferencing. TAPI 3.0 uses the Windows 2000 Active Directory service to simplify deployment within an

organization, and it supports quality-of-service (QoS) features to improve conference quality and network manageability.

4.1.2 Sun's Java Telephony API (JTAPI)

The Java Telephony API (JTAPI) is a portable, object-oriented application-programming interface for Java-based computer-telephony applications. JTAPI serves a wide audience, from call centre application developers to web page designers. JTAPI supports both first- and third-party telephony application domains. The API is designed to make programming simple and easy, while providing those features necessary for advanced telephony applications [29]. Sun has some simple goals for JTAPI, which are listed below.

- Supports both the third party and the first party call control scenarios;
- Brings simplicity to the most basic telephony applications;
- Interfaces applications directly with service providers or acts as a Java interface to existing telephony APIs, such as TSAPI, and TAPI;
- Is based on a simple core augmented with standard extension packages;
- Runs on a wide range of hardware configurations, wherever Java run-time can be used;
- Compatibility with the Java Media framework.

In the JTAPI model developers will implement their own JTAPI classes by mapping telephony functionality to TAPI, TSAPI or any other call control standard or proprietary call control mechanism. The aim of JTAPI is to allow telephones, computers and other devices to be linked through the Internet.

4.1.2.1 JTAPI Classes

The Java Telephony API is, in fact, a set of APIs. The "core" API provides the basic call model and rudimentary telephony features, such as placing and answering telephone calls. The core API is surrounded by standard extension APIs providing

functionality for specific telephony domains, such as call centres and media stream access. The following diagram (Figure 11) shows all the packages currently defined in the JTAPI architecture.

Figure 11 JTAPI packages

More sophisticated advanced telephony functionality can be found in the “Call Control” package. Examples of this type of advanced telephony functionality are call transferring and multiple caller conferencing. This type of telephony functionality is found in most PBX switch environments. Application domains have been localized into packages, allowing developers to implement only the packages oriented to their particular application.

4.1.2.2 The JTAPI Call Model

The JTAPI call model provides object level abstraction corresponding to the physical entities in the real telephony world. Examples of such objects are the provider, call, connection, terminal connection, terminal and address (Figure 12). The provider object represents the telephony sub-system. This could be a fax/voice card in a desktop computer

which implements a desktop model configuration, or, alternatively a PBX switch which implements a network model configuration. A call object is associated with a provider object. The call object represents a telephone call. The address object represents a telephone number. An address object is considered to be a logical endpoint of a call. A connection object represents the link between a call object and an address object. When two parties are talking to each other, the parties are connected to the call by one call object, two address objects and two connection objects. The terminal connection object is a representation of the phone jack. This object is responsible for notifying the terminal object of incoming calls. It also handles call progress during outgoing calls. A terminal object represents a physical telephone set and all of its features. One important part of a terminal object is its address object(s). A terminal object can have one or more address objects associated with it. Each terminal object has one terminal connection object associated with it. The figure below outlines the relationship between these six basic telephony objects found in the “Core” package.

Figure 12 JTAPI Call Model

In the computer telephony world there are many scenarios where the telephony sub-system delivers asynchronous events to an application. These events are related to the changes occurring in the telephony sub-system. To handle asynchronous telephony events JTAPI uses the observer/observable model to notify Java applications of these events.

The main goal of JTAPI is to create an API that allows applications to run on a variety of operating systems and hardware platforms, against a variety of telephony networks. JTAPI is intended to be a simple API as well. JTAPI still requires application developers to be knowledgeable about telephony, but reduces the amount of implementation-specific knowledge required to develop applications. Another positive aspect of JTAPI-enabled applications is the ability to control a new generation of Java compatible devices, like the smart telephones designed by both Lucent and Nortel [30]. With the momentum that Sun has generated with Java there is a good chance that JTAPI will be embraced by the industry.

4.1.3 Signal Computing Signal Architecture(SCSA)

SCSA is a comprehensive reference architecture for building computer telephony systems that incorporate applications and technologies from different vendors. SCSA provides a common set of hardware and software specifications that manufacturers can use to build computer telephony. SCSA specifications extend from low-level bus and hardware interfaces to high-level applications programming interfaces. Figure 13 highlights the major divisions and key elements within this architecture.

SCSA has two separate components. They are SCSA Telephony Application Objects (TAO) framework and the SCSA Hardware model [31]. These two models define the interfaces, APIs and protocols in the computer telephony implementation layer.

4.1.3.1 SCSA Software Standards or TAO

SCSA TAO is the media-processing framework for SCSA. It defines the first open, hardware-independent architecture for deploying multi-vendor, client-server computer telephony applications. SCSA TAO allows multiple vendors' applications to share a common server and its resources, making it easier to add or expand existing services.

Figure 13 SCSA Component Architecture

On the bottom level, a standard message interface (such as ECTF S.300 for media-oriented hardware and CSTA for switch connectivity) ensures flexibility and vendor independence at the hardware level. The standard interface makes it possible for the hardware resources (such as voice and fax boards) to either reside on the same physical server or be distributed among several networked systems.

4.1.3.2 SCSA Hardware Model

The key element in the SCSA Hardware Model is a high-bandwidth, time division multiplexed (TDM) telephony bus that provides distributed, symmetrical switching. The SCSA bus is a stand-alone component of SCSA that makes it easier to build scalable systems using telephony network and media resources from multiple vendors.

The switching fabric within the SCSA bus is completely distributed and treats all devices on the bus as peers. The switching fabric is non-blocking, guaranteeing that any device can talk to another at any time. Any two devices can communicate in full-duplex mode regardless of their geographic location on the bus. The SCSA bus is a real time bus designed for computer telephony applications.

4.1.4 Choose the right telephony APIs for my OSA

While the above telephony APIs enable voice, data and video collaboration over existing IP-based LANs, WANs and the Internet, they provide different foci and capabilities. Unlike TAPI, SCSA and JTAPI's software models allow applications to control resources through a high-level interface, rather than force developers to use a low-level interface that directly manipulates the physical resources. This design makes it possible for applications to connect to a variety of underlying hardware implementations which are both open and proprietary. More importantly, JTAPI and SCSA have little dependence on the underlying network topology and runtime Operating System. It makes JTAPI and SCSA more desirable than TAPI considering the deployment flexibility. Another major issue is the adaptability to new technology. As mentioned in the previous chapter, Internet Telephony growing fast at present and is forecast to be the major source of telephony traffic in near future. TAPI has provided support for Internet Telephony in the

TAPI 3.0 architecture. In the meantime, JTAPI can support the Internet Telephony architecture with integration with JAIN models. However, there is no clear definition about the role of Internet Telephony in SCSA yet. SCSA also has a capacity problem since its bus architecture can only support up to 8MHz of bandwidth. It is not user friendly with regard to building multimedia applications. Based on above analysis, JTAPI is proposed as the ideal implementation technology for the end-user layer.

4.2 Adaptive Layer and Service Layer

The adaptive layer provides a service-independent platform which integrates features deployed in the service layer. Features and communication protocols are encapsulated as software components and can be shared amongst network domains through code delivery. Middleware components run within a controlled runtime environment provided by the server vendor, which manages their creation, management, and destruction. With the middleware component, the developer no longer needs to write code that handles transactional behaviour, security, database connection pooling or threading, because the architecture delegates this task to the server vendor.

The benefit in adopting a component-based framework is that developers can focus more on business problems. Programmers do not have to worry about low-level programming details, such as database access and security. A component framework is a set of interfaces and rules of interaction that govern how components 'plugged into' a framework may interact [32]. Today there are three major forces in the component framework arena:

- Sun's Enterprise Java Beans (EJBs)

EJB provides a fully scalable, distributed and cross-platform architecture that makes the most of business resources. Not only can these components run on

any platform, but they are also completely portable across any vendor's EJB component execution system. The EJB environment automatically maps the component to the underlying vendor-specific execution services.

- **The OMG's CORBA Components (CCM)**

CCM is a specification that focuses on the strength of using CORBA as a server-side object model. It concentrates on issues that must be addressed to provide a complete server-side middleware component model. It can be described as a cross-platform, cross-language superset of EJB. CCM gives developers the ability to quickly build web-enabled enterprise-scale e-commerce applications while leveraging the industrial strength of CORBA. Tight integration with EJB leverages CORBA's cross-platform and multiple-language capabilities.

- **Microsoft's Transaction Server (MTS).**

MTS is built on DCOM and brings in mainframe-like transactional reliability to the PC world following a "write once, run many" strategy. Developers use MTS to deploy scalable server applications built from COM components, focusing on solving business problems instead of on the programming application infrastructure. MTS delivers the "plumbing" — including transactions, scalability services, database connection management, and point-and-click administration — providing developers with the easiest way to build and deploy scalable server applications for business and the Internet.

For effective component assembly, components must be able to inter-operate. Obviously without interoperability the component market will be fragmented and less successful. Hence component standards such as EJB and CORBA Components Model have put interoperability as the top issue. In the following section, these three component frameworks will be discussed.

4.2.1 Sun's Enterprise JavaBean (EJB)

Enterprise JavaBean technology is a specification framework that supports the development and deployment of distributed object-oriented business applications with

component technology in Java. EJB could be treated as server-side JavaBean component, so before EJB is discussed, it is worthwhile mentioning the component standard that preceded it, namely JavaBeans.

4.2.1.1 Java Beans

JavaBeans is a portable, platform-independent component model written in the Java programming language [33]. Developers take advantage of the platform independence of Java and write reusable components once and run them anywhere.

Beans have been defined as objects or components created with a set of characteristics to do their own specific job [34]. They have the ability to take on other characteristics from the container in the server in which they currently reside. This enables a bean to behave differently, depending on the specific job and environment in which it is placed.

A JavaBean is a component that has interfaces or properties associated with it. It can be interrogated by and integrated with other beans that were developed by different parties at different times. A JavaBean is different from other objects in that it has a properties interface. This interface can be read by certain tools as it describes what the component does. With this information the JavaBean can be hooked up with other beans and plugged into other environments.

JavaBeans, in contrast to Enterprise Java Beans, are generally intended to be local to a single process. An individual bean will behave differently, but typical unifying features that distinguish a bean are:

- **Introspection** - this enables a builder tool to analyze how a Bean works
- **Customization** enables a developer, using an application builder tool to customize the appearance and behaviour of a Bean

- Events enable Beans to communicate and connect together
- Properties enable developers to customize Beans and program with them
- Persistence enables developers to customize Beans in an application builder, and then retrieve those Beans with customized features intact for future use.

4.2.1.2 Enterprise Java Beans (EJB)

Enterprise JavaBeans take JavaBeans to the next level, that being to server-based components or “distributed JavaBeans”. The Enterprise Java Bean API allows developers to build large-scale business applications as reusable server components which can interact with each other through the network [35].

Previously, developing multi-tier servers was a complex task, where developers had to deal with the issues of concurrency, transactions, security and scalability, and had to manage threads, memory and network connections. EJB aims to make the development of server side components easier, by decoupling application semantics from infrastructure issues. However EJB is simply a model and vendors provide EJB solutions.

Enterprise JavaBeans, unlike JavaBeans, are generally non-visual and designed to run on a server and to be invoked by clients. An EJB could be built out of non-visual JavaBeans.

Figure 14 EJB architecture

Figure 14 illustrates the architecture of EJB which contains:

- EJB server
- EJB containers that run within the server
- Home objects, Remote EJBObjects and Enterprise Beans that run within containers
- EJB clients

The following paragraph introduces each component of the EJB architecture.

4.2.1.3 EJB Client

EJB clients make use of EJB Beans for their operations. They find the EJB container holding the bean through the Java Naming and Directory (JNDI) interface. They then make use of the EJB Container to invoke EJB Bean methods. The client supports multiple programming languages. If the client is written in Java then the client will communicate with the EJB by using the Remote Method Invocation (RMI) API. If the client is a non-Java client then it will communicate with the EJB using CORBA communication protocols, such as IIOP.

4.2.1.4 EJB Container

The EJB component model provides an environment in which server-side components can be deployed. This environment (the container) provides transactional capabilities, security and management to the component. Developers create components that inherit the enterprise attributes from the container. The component interacts with the container and takes advantage of its transactional or security features without being aware of how they are implemented by the container.

There are two types of containers which are session containers that may contain transient, non-persistent EJBs whose states are not saved at all, and entity containers that contain persistent EJBs whose states are saved between invocations.

4.2.1.4.1 EJB Remote Interface EJB Object

The remote interface lists the business methods available for the Enterprise Bean. The EJBObject is the client's view of the Enterprise Bean and implements the remote interface. The EJB object works with the EJB container to apply transactions, security and other operations to the bean at runtime. When a client calls a method on an EJB object, the EJB object communicates with the EJB container and requests that the same method be called with the same parameters on the EJB component.

4.2.1.4.2 EJB Home Interface and Home Object

The EJB Home Interface defines factory methods for locating, creating and removing instances of EJB classes. The Home object is the implementation of the Home interface. The EJB home object is generated by the EJB vendor when an EJB is installed in a container based on its Home Interface. This object is involved with the bean's lifecycle, and is responsible for the location, creation and removal of the Enterprise JavaBean.

4.2.1.5 EJB Server

The EJB server provides an organized framework or execution environment for EJB containers to run in. It makes available system services for multiprocessing, load-balancing, and device access for EJB containers. The EJB server also makes EJB containers running within them visible to the outside world.

4.2.1.6 EJB Design Patterns

There are two types of Enterprise JavaBeans that can be created: the Session Bean and the Entity Bean.

4.2.1.6.1 Session Beans

Session Beans are non-persistent objects that implement some business logic [36]. A Session Bean is usually associated with a single client and cannot survive server crashes. There are two types of Session Beans - Stateful Beans and Stateless Beans. The Stateless Beans do not store any information relating to the client between calls while Stateful Beans store information between calls from a client. Values and states are retained from previous communications.

4.2.1.6.2 Entity Beans

An entity bean maps a Java class to a data source. This source could be a row in a database or an entire table [37]. Each entity bean has a primary key associated with it that identifies the data within. In general, Entity Beans always have states. Each Entity Bean may, however, be shared by multiple EJB Clients. Their states can persist and be stored across multiple invocations. Hence they can survive system shutdowns.

4.2.2 CORBA Component Model (CCM)

The CORBA Component model is part of the CORBA 3.0 specification [38]. This component model specifies a framework for the creation of plug-and-play CORBA objects. The integration of Enterprise JavaBeans is also an integral part of the component model, and the model will be helpful in the further integration of other object-based technologies. The CORBA Component Model (CCM) is a specification for creating server-side scalable, language-neutral, transactional, multi-user and secure enterprise-level applications. It provides a consistent component architecture framework for creating distributed n-tier middleware. It would be appropriate to call a component written in CCM specification a CORBA component. In brief, CCM is essentially a language-independent extension of EJB. The problem that many organizations have with EJB is that EJB only supports Java. Many organizations have software written in other languages and so EJB does not fit their environment. CCM supports EJB components as well as components developed in other languages such as C++, Smalltalk, COM/DCOM and COBOL.

The CORBA Components model, like EJB, is related to server-side CORBA Components. CORBA Components extend the core CORBA object model, and provide a higher level of abstraction, thus greatly simplifying CORBA programming. CCM Components use services such as transactions, security, events and persistence.

A Component is a new meta-type in CORBA. It is an extension of the Object meta-type. A Component can be specified in IDL and stored in the Interface Repository. A component maintains its own reference, which is analogous to an object reference. Clients interact with components through surface features called ports. There are five types of ports:

- Facets - distinct named interfaces provided by the component for client interaction
- Receptacles - named connection points
- Event sources - named connection points that emit events

- **Event sinks** - named connection points into which events may be pushed
- **Attributes** - named values exposed through accessor or mutator methods

The CORBA Component model provides extensions to IDL that will support components and the relations between them. CCM defines a new version of IDL – CIDL, which is a mechanism for automatic code generation for defining servant implementations. CCM provides a means to support navigation across multiple interfaces of a CORBA Component and it includes a deployment model using XML, to describe the runtime properties of a component. It defines a container model and introduces the container programming model to offer server side system services such as servant lifetime, security and persistence. CCM also provides inter-operability with EJB components by defining a mapping to Enterprise JavaBeans so that EJB can be supported as a CORBA Component.

4.2.3 Microsoft Transaction Server (MTS)

Microsoft Transaction Server (MTS) targets the creation of component-based, transaction-oriented applications in Windows NT environment. MTS is based on the Component Object Model (COM) [39], and an MTS application is implemented as one or more components. A basic MTS architecture is shown in Figure 15 below and is made up of:

- The MTS Executive (mtxex.dll)
- The Factory Wrappers and Context Wrappers for each component
- The MTS Server Component
- MTS clients
- Auxiliary systems like COM runtime services, the Service Control Manager (SCM), the Microsoft Distributed Transaction Coordinator (MS-DTC), the Microsoft Message Queue (MSMQ), the COM-Transaction Integrator (COM-TI), etc.

Figure 15 Microsoft Transaction Server Architecture

4.2.3.1 Component Object Model

The Component Object Model (COM) is a way for software components to communicate with each other. It is a binary and network standard that allows any two components to communicate regardless of what machines they are running on (as long as the machines are connected), what operating systems the machines are running (as long as it supports COM) and what language the components are written in. COM also provides location transparency, in that it does not matter if components are in-process DLLs, local EXEs or if they are located on another machine. Interfaces are used to communicate with a COM object. An interface is a contract between the component and its clients, which defines what functions are available and what the object does when the functions are called.

COM does not support implementation inheritance type of reuse. To reuse them, containment and aggregation are used. Containment means the object required is instantiated, and requests are passed to it. If deep containment hierarchies occur, they can

become a performance overhead, so COM defines aggregation. Aggregation means that instead of forwarding requests to an inner object, the inner object's interface reference should be given to the client. In practice containment is used more often than aggregation. Aggregation is generally used where there are deeply nested constructions. Aggregation also brings an extra level of complexity.

COM supports inter-process communication, but not communication across machines, so Microsoft introduced DCOM technology to support distributed computing capability [40]. DCOM builds on COM to provide communication across process and machine boundaries. DCOM creates proxy objects on the client side and stub objects on the server side to deal with differences in data representations across machines. DCOM marshals data into network data representation (NDR), which is a platform-independent format.

4.2.3.2 MTS Architecture

MTS provides a surrogate for in-process server-side components and it can also be defined as a component-based programming model [41]. Just as a COM component can be modeled on the basis of interfaces and their implementation, MTS enforces modeling based on the component's state and behaviour. MTS, through the COM infrastructure, handles communication between components using DCOM. This allows MTS to expose its components to Windows applications from anywhere on the network. As long as only Windows NT is running, MTS components can be deployed on top of existing transaction processing systems, including traditional transaction processing monitors, web servers, database servers, application servers, and so on. Legacy system integration with non-Windows systems and MTS is achieved using the COM Transaction Integrator (COM-TI)

technology. Also, it is important to realize that MTS is a stateless component model, whose components are always packaged as an in-process DLL. Since they are composed of COM objects, MTS components can be implemented in a variety of different languages including C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL!

4.2.4 Choose the right framework for my OSA

All the technologies introduced in the above sections target the creation of component-based, transaction-oriented applications. These technologies will now be compared. More specifically, the evaluation will focus on analyzing the similarities and differences between the ways that the component models handle objects, transaction support, controlling complexity, portability, interoperability and language choice. The requirements of my OSA will also be discussed.

In standard evolution, the CORBA Component Model and the Enterprise JavaBean Model are actively evolving in parallel. Each new release of the specifications re-tests the leader/follower relationship. Unlike EJB and CCM, MTS is an actual implementation rather than just a specification.

With respect to openness, as CCM is owned by a multi-vendor group (OMG), it receives more industry support. EJB and MTS are the property of a single company, so they cannot achieve the same level of vendor-neutral capability that CCM does. Although CCM has a huge member support, it has the lowest implementation base as it uses the latest technology and the development is quite complex. Of the three, MTS possesses the highest implementation base due to the popularity of the Windows NT system.

From the point of view of communication protocol, CCM specifies IIOP as its unified wiring protocol. EJB is less specific: some EJB-based Java application servers

require RMI, and others support IIOP, or both. This lack of specification with EJB leads to design complexity. MTS supports both IIOP and DCOM as the distributed computing technologies.

With regard to multiple state support, EJB has two component abstractions: Session and Entity Beans. CCM adds a process component abstraction, similar to a Stateful Session Bean in EJB. MTS is a stateless component which exists in the form of a Dynamic Link Library (DLL) or .EXE file.

On the topic of available services, CCM uses the CORBA Services (COSS) for events, naming, life cycle, persistence, security and transactions. EJB has adapted subsets of COSS features in transactions and naming and redesigned the other COSS themes, which means they cannot be seen as COSS-compatible subsets. MTS realizes such server-side services under its proprietary implementation so that it cannot manage components implemented in other technologies.

Regarding interoperational support, CCM defines a bi-directional mapping to EJB. EJB specifies a CORBA mapping for client communication and inter-container interoperability. EJB could not address component model compatibility because CCM did not exist at that time. CCM is also working on providing capabilities to access COM components in MTS.

From the view of portability, there are two types of portability: platform portability, and portability between server containers. EJB is the best on the platform level because JVM supports a multiple OS environment. At the server container level, since the EJB specification is ambiguously defined, an EJB component cannot run in another vendor's container. With the CCM specification, it tries to solve the problems faced by EJB. It

defines a contract which supports interoperability between different vendors' ORB. With regard to MTS, since Windows NT is the only runtime environment and Microsoft is the only platform vendor, there is no portability issue.

MTS appeared on the market first (in 1996) and has more stability than the other two. EJB still undergoes major changes with each release and there is no implementation available for CCM as yet.

The following table provides a summary of the evaluation of the above three standards. From a technical aspect, I propose CCM as my preferred component-framework because adaptability is the major concern of my design. More importantly, in the telecommunication industry, there are few customer bases using the NT platform, especially on the server side.

Platform Issues	EJB	CCM	MTS	Preference
Owner	Sun	OMG	Microsoft	CCM
OS support	Many	Many	Windows NT/2000	CCM
Development Language	Java(Free)	Many(Most Not Free)	Many(Not Free)	EJB
Integration	CCM	EJB	None	CCM/EJB
Message Passing	RMI/IIOP SOAP	RMI/IIOP SOAP CCM	IIOP SOAP	CCM
Event Service	N/A	Notification Service	CCM Events	CCM
Transaction Support	Java Transaction Service (JTS)	CORBA Object Transaction Service(OTS)/JTS	MTS	CCM
Persistence	Java Serialization	Persistence Storage Service (PSS)	Database	CCM
Asynchronous Messaging Support	Java Messaging Service (JMS)	Time-independent Invocation(TII) in CORBA Mess. Spec	Microsoft Message Queuing Service (MSMQ)	EJB
Naming Service	Java Name & Directory Interface (JNDI)	CORBA Naming Service	Active Directory Service	EJB
Available Implementation	Few	Some	Many	MTS
Maturity	Latest to market still in expansion	Need expand to fit new features in CCM	Mature	MTS
Portability	JVM Level portability	Proposed to support Inter-ORB portability	No	EJB
Dev. Tool	Many	N/A	Many	MTS
Runtime Env.	Many(No Free)	N/A	Integrated in Windows NT/2000	EJB

Table 1 Platform Comparison

Unfortunately, there is no industrial CCM platform available, so my implementation only used CCM as a reference model. The component technology I chose was JavaBean. Compared to JavaBean, EJB is closer to CCM, but its advantage is based on providing server-side management capabilities (e.g. transaction and directory services) and it requires an application server runtime environment (i.e. IBM WebSphere) to realize these functionalities. More importantly, while EJB offers many more distributed computing

functions than JavaBean does, it cannot support the Distributed Event Service which is the key messaging mechanism used in my OSA. For my implementation, the two major features I picked up from the Java Component Model were event service and Java Reflection APIs. Both JavaBean and EJB share the same functionality with regard to these two capabilities. Compared with the EJB-based application, implementation using JavaBean had the same level of clarity to demonstrate the design but involved much simpler coding work.

CHAPTER 5

IMPLEMENTATION OF SERVICE ARCHITECTURE

This chapter will focus on introducing the implementation details of my Open Service Architecture (OSA). Based on the OSA framework a Universal Call Forwarding (UCF) feature is deployed and implemented. The first two sections illustrate the business logic and service scenario of the UCF service. Section 3 to Section 5 will demonstrate the implementation of the functional components for each layer.

5.1 Design of Universal Call Forward

Universal Call Forward is a value-added service designed to integrate traditional IN service with Internet technology. If the called party is unavailable when there is an incoming call, instead of sending back a busy signal, the call processing application will hold the call and query the callee's connection mode. If the user is on the phone, the call will be transferred to another number (as in a traditional call forward service). To make the feature more flexible, the user can specify the numbers to be transferred based on either the time schedule or the callee's preference. If it is an Internet connection, a message will flash on the user's screen to notify him/her that there is an incoming call. The called party can choose to either take the call or discard it. If he/she decides to answer the call, the Internet connection will be dropped; otherwise, the call will be routed to a mailbox. Other similar services are available, such as Internet Call Waiting, offered by Bell Canada, and Internet Mailbox from the web-based third-party service providers. The Internet Call Waiting

solution is to install client software on the user's machine. Whenever the user connects to the Internet, the client software will be activated and register the user's status in the carrier's database. Whenever an incoming call gets a busy feedback, the call-processing program will hold the call and query the database for the user's connection status. If the user is in on-line mode, a message will be sent to the user's client software and will bring up a notification window. The whole process is handled by one service provider using one central database. The Internet mailbox requires the user to subscribe to a "call forward when busy" feature on his/her local carrier first. If the incoming call gets a busy response, the call will be forwarded to the company's processing server. The server will check the user's connection mode. If the user is surfing the Internet, the call will be transferred to the mailbox server and the caller will be prompted to leave a message. While these two approaches seem to provide the same functionality as my UCF feature, they have drawbacks due to the limitation of the traditional IN architecture. The telcos' solution cannot provide both the Internet Call Waiting and Mailbox service at the same time, since their call control entities belong to different domains which are private and cannot interoperate. The existing Internet Call Waiting cannot provide any call control functionality for the incoming call (i.e. call forwarding). More importantly, the end user has to subscribe to the service from the service provider directly, and there are few choices available (i.e. for the Internet Call Waiting Service, you can only subscribe from your ISP).

My service framework provides a great deal of flexibility for end users to choose services. Based on the OSA described in previous chapters, services are deployed by service providers in the service layer and they publish their services in the service nodes (local carrier) in the network layer. Whenever the end-user wants to register a service, he

can choose the same feature from multiple service providers. The customer can either register with a specific service provider or just define a rule (i.e. choose the service which is the cheapest). The service node will automatically assign a service provider based on the policy defined.

5.2 Simple Service Scenario

Figure 16 below shows how a sample UCF service scenario could fit into the Open Service Architecture as defined in the previous chapters.

Figure 16 Universal Call Forwarding Scenario

When the calling party initiates a call (1) and gets a busy signal from the called party (2), the end-user terminal will hold the call and send the called party a busy message,

which contains the called party's identification, to the service layer through the adaptive layer (3,4,5). In the meantime, it sends a `Service_Processing` signal to keep the caller connected. The message control point in the corresponding service provider receives the message and dispatches it to the service control point. The service control point sends a "called-party-busy" message to the service logic control point asking for action (6). The service logic control point returns the instruction to "check-connection-state" (7). The Service Control Point sends out this request through the message control point (8). The adaptive layer routes the message to the proper service provider (in the picture, it is service provider2) (9). The service provider2 receives the request and asks the specific network node to send back the user's connectivity state (10,11,12,13,14) and returns it to the requested service provider(in the picture, it is service provider1) (15,16,17). When the service provider1 receives the user's connectivity state, it sends that information to the service logic control point. If the user is on the Internet, then the service logic control point sends a "`Busy_Transfer_On_Internet`" message back to the service node (18,19,20). The service node receives the message and checks the user's profile to find out the right Internet Call Transfer to activate (21). Assuming the user has a request for the cheapest service, the network node will go through the call transfer repository, finds out the cheapest service and activate it. The service in the service repository is deployed with bean-oriented technology. While the Communication Bean sends the notification to called party through the Internet, the Service Management Bean transfers the billing information and related service data to its home service provider (22). If the called party chooses to stay on the Internet (23), the transfer service sends a "`User_Keep_On_Line`" message back to its service provider (24,25). The service logic control in the service provider2 checks the state and sends back a

“Transfer_To_Mailbox” message to its underlying network node (26,27,28). The network node gets the message and goes through its Mailbox service repository, finds the right one based on the user profile and activates it (29,30). The calling part will get a voicemail prompting him to leave a message (31). With the separation of service providers and service implementation, end users get more choices for the new service compared with the traditional IN framework.

In the following sections, implementation details for each layer will be described.

5.3 End User Layer Implementation

As discussed in the previous chapter, the end user layer is implemented with JTAPI technology. JTAPI provides applications with a standard call model for maintaining the call state and hides the hardware API or other APIs (i.e. TAPI and TSAPI etc.) from the application.

JTAPI itself, while providing those features necessary for advanced telephony applications, is somewhat limited for the requirements of the OSA described in this thesis. In particular, JTAPI seems to be deployed to support two types of scenarios:

- Call set up occurring in a close domain (as PBX).
- The JTAPI call control signal is event-based and all events pass within the same Java Virtual Machine.

As JTAPI assumes that all applications are running on a single platform, JTAPI Provider controls all the legs of a call. While this assumption may bring the convenience of managing a centralized call centre, it is not realistic for multi-layer call control of Internet Telephony [42].

In my design, the client works in a distributed way, so my implementation extends the standard JTAPI to provide distributed call control capability. The extension provides

communication support for multiple providers to co-ordinate and interact within one call lifetime. The JTAPI model assumes that the provider object is sustained throughout the lifetime of a call. However, it also defines the behaviour when the user wants to have different providers for different portions of the network traversed by a single call (i.e. call set-up through service node and service providers) or different legs of a multi-party call (i.e. with the Universal Call Forward feature, call transfer and mail box belong to different service providers). In the Open Service Architecture service scenario, it is very common to change the provider during the process of a call. As mentioned in Chapter 3, voice may not be the only media type in OSA. If we try to involve different media capabilities, we cannot guarantee that they are all offered by the same service provider. In general, the main goals of the extension are to:

- Provide greater flexibility in service creation, e.g. provide distributed event service which hides the implementation detail of the signalling system.
- Facilitate greater scope of service by providing an interface for interaction with higher-level call control functions (i.e. service logic in service providers).
- Offer adaptability to multiple communication protocols, such as SIP and H.323, and across platforms using Java.

In the client's implementation, new types of call set-up states have been introduced to the JTAPI call model in order to support the open service architecture. The new states introduced are `Service_Request`, `Service_Negotiate`, `Service_Subscribe` and `Service_Process`. All state transitions occur in response to events that are generated by the parties involved. Each state instance represents a specific leg of the call connection. The sender and receiver states are used by the caller and the receiver's connection. In the following paragraph, the extended Call model will be described in detail, together with the dynamic service subscription example.

Suppose the caller initiates a call while the called party is on the Internet. Instead of sending back a busy signal, there is a new service called Internet phone available whereby the caller can reach the callee via the Internet. So a Switch_Service event is sent back to caller. If the caller has such a switch capability, it sends back a Receipt_Switch confirmation. The callee returns the Switch_Type indicating the service is Internet Call. The caller checks its terminal capability and if it supports such feature, it activates the function to make the Internet Call. If it cannot initiate such a feature at its local terminal, it sends a Get_Service to its service node and moves its connection state to Service_Request. In this state, the caller just suspends the call set-up process with the callee and interacts with its service node to get the requested service. If the caller's service node does support the Internet Call feature, it sends a Service_Ready to the caller and moves the caller's state to Call_Process. In the Call_Process state, the caller updates its local terminal with an application downloaded from the service node. If the caller's local service node does not support the service, it consults the callee's service node to locate the feature. The callee's service node then returns the service's location. Let's assume the service is not free and it requires the user's subscription. The service node notifies the caller through a New_Service_Available event and sends a suggestion to the caller, indicating that the service provider is available through URL or Plug-in Software (i.e. Java Plug-in for Netscape). If the caller accepts the suggestion, it initiates a negotiation with that service provider. Since this operation takes place during call set-up but is actually conducted with a party not involved in the call, the connection to the caller moves to the Service_Negotiate state. In this state the user may be consulted to provide payment information to receive the service. Once the subscription process finishes, the caller's service node downloads the

application from the proper service provider. It transfers the caller's state to Service_Process with a Service_Available event and updates the local terminal's capability. When the update is finished, the caller resends the Initial_Call event to the service node to resume call processing for the new Internet Call. Figure 17 shows this procedure. Timers are defined at each stage to provide an upper bound on the negotiation time. Based on this extended call model, the JTAPI can support different caller/callee during the call set-up phases.

Figure 17 Terminal Feature-Subscription Procedure

For the sake of simplifying the GUI development, my client implementation extends the JTAPIChat [43]. JTAPIChat uses text-based media to simulate the real call set-up process. Users are identified by their IP address during the communication. All the control and signal behaviour of a voice call can be fully simulated in this environment. The only difference is the media. Since JTAPI deals solely with the call control process instead of media handling, such an implementation is sufficient to represent the real scenario. In my implementation, the end user can also choose to generate Internet Traffic or General Telephony Traffic through EUI. Figure 18 is a snapshot of the client GUI.

Figure 18 Client GUI Snapshot

5.4 Network Layer Implementation

The network layer not only provides access service for end users, but also serves as a repository to hold applications from different service providers. The primary challenge in the network layer implementation is how to provide sufficient adaptability to handle the frequent update of the service repository without interfering with existing services. My network layer development is focused on providing the answer to this question. In the implementation of my network layer, I define a dynamic extensible framework which is data-type independent. The resulting framework enables both objects and classes to be integrated dynamically without touching the framework source code [44]. In the following section, I will illustrate the framework's benefits and drawbacks in detail by constructing a service repository for the call-forward feature.

Figure 19 Class diagram of Plug-in Framework

Figure 19 is the class diagram of the framework in this layer. The framework has two interfaces, Transferable and TransferFactory, and one coordinating class, Transfer. The Transferable interface wraps up the call transfer action. The following code presents the definition of a Busy_Online_Transfer Service.

```
public class Busy_Online_Transfer implements transferable, Serializable, ui.Sock.SockListener
```

```
/*  
  ui.Sock.SockListener is the interface in User Interaction Bean whose implementation detail will  
  be introduced in section 5.6.3.  
  */  
/
```

The Transfer object is responsible for coordinating the transfer service and deciding whether call transfer requests can be met. The TransferRepository implements a TransferFactory interface and plays the role of a service repository which hosts different vendors' implementation of Call Transfers in the form of a Transferable instance. The TransferRepository also serves to categorize the different types of call transfer service (e.g. Busy_Transfer and NoAnswer_Transfer, etc.). In the Transfer class, once the service logic control has determined which type of call transfer to invoke, it finds the specific TransferFactory instance through the information in the Properties object. The Transfer object is responsible for executing call transfer requests for different types of Transferable objects. In order to accomplish this task, the TransferFactory implementation has to support the given type and dynamically load it, based on the policy associated with the user profile (i.e. the cheapest call forward) in that storage. There is a factory method in the transfer object which loads the class using the static forName() method in Class and then create a new instance of the loaded class. In my implementation, the service name is categorized in the form of a trigger event plus the operation. For example, the Call Transfer is named as Busy_Transfer when the called party is busy while it is named as NoAnswer_Transfer

when there is no answer recognized from the called party. This naming convention can simplify the process in service logic control because it saves the implementation to map the call state into related service. In the real world, if we want to have a complicated service processing feature, we can use XML to define the service logic control policy and use a parser to parse the policy database to locate the service name based on the call state information. The utilization of the framework is based on an assumption that the loaded class implements the TransferRepository and has a constructor with one Properties argument. If not, the constructor of the loaded class throws an exception and it is wrapped by a newInstance in an InvocationTargetException. Following is the sample code for the GetTrasferFactory method in the Transfer class and it loads the specific TransferFactory instance based on transfer service type.

```

try
{
    Class TransferFactory = Class.forName( className ); //The className represents a specific
    Constructor FactoryConstructor // service (i.e. Busy_Online_Transfer)
    = TransferFactory.getConstructor( new Class[]{ Properties.class } );
    Object aTransferFactory
    = FactoryConstructor.newInstance( new Object[]{ myProperties } );
    System.out.println("Return class is "+className);
    return (TransferFactory)aTransferFactory;
}
catch( ClassNotFoundException e )
{
    System.out.println( "It seems that " + type + " isn't supported." );
    System.out.println( "Couldn't find the class " + className );
}
catch( NoSuchMethodException e )
{
    System.out.println( "It seems that " + type + " isn't supported." );
    System.out.println( "Couldn't find an appropriate constructor in "
        + className );
}
catch( InvocationTargetException e )
{
    System.out.println( "It seems that " + type + " isn't supported." );
    System.out.println( "The constructor of " + className +
        " did throw an exception." );
}

```

```

    }
    catch( IllegalAccessException e )
    {
        System.out.println( "It seems that " + type + " isn't supported." );
        System.out.println( "Couldn't access " + className );
    }
    catch( InstantiationException e )
    {
        System.out.println( "It seems that " + type + " isn't supported." );
        System.out.println( className +
            " is an interface or an abstract class." );
    }
    return null;
}

```

java.util.Properties class is used to help categorize the different types of service in the ServiceRepository. Each service type has a property file called dynamic.<type>.properties. The naming standard for the class name is “dynamic.<type>.storage” where “type” is a name describing a category of service. The init properties may have any name except names starting with “dynamic.” and ending with “.storage”. It is advisable that a naming standard is adopted for the init properties as well (my format is trigger event plus operation). This file is read by the framework in order to find classes implementing the TransferRepository. The following fragment is an example of a call transfer service property file.

```

# dynamic.transfer.properties
# The class used to store a Transfer instance
dynamic.B_O_Transfer.storage=Busy_Online_TransferFactory
# The storage directory for a transfer repository.
dynamic.transfer.storagedir=c:\\service\\transfer

```

To integrate different vendors’ service into the service repository, the application needs to implement the framework’s interfaces. The class which implements the Transferable interface reflects the customized behaviours of a feature (i.e. performance and

service rate). The TransferRepository implementation is responsible for managing the CallTransfer object's persistence. My implementation uses serialization to achieve the persistence. Following is the code of load() method in the Busy_Online_TransferFactory class which implements the TransferFactory interface. It loads a service repository instance for a Busy_Online_Transfer service.

```

try
{
    storagefile = new File( storagedir, name + ".ser" );
    fis = new FileInputStream( storagefile );
    ois = new ObjectInputStream( fis );
    Object storedObject = ois.readObject();
    return (Busy_Online_Transfer)storedObject;
}
catch( ClassNotFoundException e )
{
    throw new IOException( "The file " + storagefile + " doesn't have a " +
                           "correctly serialized object of type: " +
                           Busy_Online_Transfer.class );
}

finally
{
    if( ois != null ){ ois.close();}
    if( fis != null ) fis.close();
}

```

When a service provider develops a new service and wants to publish it on the service node, it should do the following:

- Add compiled run-time files (usually .class files) that implement the Transferable and TransferRepository in the service repository (defined in dynamic.type.properties file)
- Edit a Properties file if it is a new service category.

5.4.1 Framework Discussion

The framework has the following benefits:

- Since each vendor's implementation is wrapped up by the ServiceRepository and the ServiceRepository could be invoked by Service Logic without knowing its

existence in advance, a new type of service and new implementation in the framework is independent of its extensions. This independence is especially useful when extensions are made by the third parties or if the framework needs to be updated frequently.

- User extensions need no modifications to the framework. In fact, users don't even need to access the source code of framework. The user just saves the implementation classes under the service directory, edits the properties file, and it is ready to use.

The framework also results in some limitations in quality, including:

- The framework provider has no influence over the quality of the extensions.
- Since the framework uses Java Reflection API, the performance will decrease sharply with the growth of ServiceRepository numbers in the framework.
- Some programming errors are not detected at compile-time, so it is difficult to construct the service management application to handle such runtime problems.
- It is not possible to update an extension to the framework while running the program.

5.5 Adaptive Layer Implementation

The adaptive layer plays the same role as the signalling system does in the traditional telecommunication service architecture. It provides a unified data sharing mechanism for applications running in the access layer to communicate with their peer as well as their primitive in the service layer. Since there data exchange is more frequent in the Open Service Architecture (i.e. the user profile roams among different service providers and service information, such as billing data, is transferred from the access layer to the service layer), the solution should be flexible enough to fit different vendors' requirements (i.e. providing abundant data types). For the purposes of easy integration, the implementation technology should be in accordance with that of the other layers in a JavaBean-oriented approach. My implementation is based on an InfoBus which is deployed

by IBM [45]. In general, InfoBus is a data broker system. It enables the dynamic exchange of data between JavaBeans' component architecture by defining a small number of interfaces between co-operating Beans and specifying the protocol for the use of those interfaces. The protocols are based on the notion of an information bus. The data to be communicated has to be converted into a data item form and placed in the information bus. Any Bean implementing the InfoBus interface can register itself on the communication channel.

Although the current InfoBus specification includes robust functionality for encapsulating data bits and sharing them among many Javabeans within both applications and applets, its running scope is within the same JVM (Java Virtual Machine) instance [46]. This cannot meet the OSA's service distribution requirement since data sources should be "network enabled" in my OSA architecture.

Therefore I extended the InforBus specification to support a distributed data exchange which allows us to pass InfoBus information between JVMs, whether located in the same machine or on the other side of the world.

Suppose App1 and App2 represent two service providers in Figure 20. An exchange of information (i.e. user profile) is required. If App1 wants to send information to App2 through the Distributed InfoBus, App1 should implement the InfoBusDataProducer interface while App2 implements the InfoBusDataConsumer interface. A data producer is an InfoBus participant that implements the InfoBusDataProducer to listen for requests and announces data availability or revocation by firing events on the bus. A data consumer is an InfoBus participant that implements an InfoBusDataConsumer to listen for availability and revocation events, and requests data items by firing events on the bus. Similar to the

producer mechanism, it controls events by calling `addDataConsumer()` and `removeDataConsumer()`. Before a consumer and a producer start the data exchange, they both have to register with their local `InfoBus`. When the data is ready on App1, it creates a `DataItem` Object which wraps up the data with the `InfoBus` format. Then App1 generates an `InfoBusItemAvailable` event through calling methods `fireItemAvailable()` on the `InfoBus` class, which sends the event to the registered consumers (1,2). The `DataConsumerProxy` object behaves like a `DataConsumer` on the local `InfoBus` of App1. Since the consumer and producer are located on two different JVMs, I introduced an `InfoBridge` object to carry information across the JVMs. `InfoBridge` is a factory object and it creates a `Producer` object (`ProducerProxy`) that clones the `DataItem` object on JVM where App2 resides (3,4). The following code is the definition of an `InfoBridge` interface.

```
public interface InfoBridge extends Remote
{
    public ProducerProxy retrieveProducerProxy() throws RemoteException;
    public void removeProducerProxy(ProducerProxy pp) throws RemoteException;
}
```

Figure 20 Distributed InfoBus

The InfoBridge uses distributed technology to create a producer on the remote side. Any distributed computing technology is applicable as the implementation technology for the InfoBridge object (my implementation made use of Java RMI since all applications are Java-based). The producer registers with its local InfoBus and fires the InfoBusItemAvailable event (5,6). App2 receives the notification and checks the data by calling getDataItemName() method. If the name is what it is looking for it will get the data, otherwise it will discard it (7). In above example, the DataConsumer works in a passive way. It can also work in an active way to request the data from the producer by generating the InfoBusItemRequestEvent through calling the DataItemRequested() method (8,9,10,11). The best method for sending the reply back will now be discussed. The most straightforward solution is to replicate the above process in reverse. This approach requires one more set of ConsumerProxy, ProducerProxy and InfoBridge instance and related temporary objects. Once the ConsumerProxy sets up a connection with the ProducerProxy, they know each other so there is no need to repeat the process to build up the channel in reverse. To make the application more efficient, I chose to use the Callback mechanism to pass the result data. In the Callback mechanism, both ConsumerProxy and ProducerProxy implement the Remote Interface. There is a register() method on the ProducerProxy side while a notify() method is within the ConsumerProxy. When the ConsumerProxy locates the ProducerProxy, it calls the register() method to pass its reference to the ProducerProxy. When the result is ready on the ProducerProxy side, it calls the notify method to send back the information. Below are the ProducerProxy and ConsumerProxy remote interfaces.

```
public interface ProducerProxy extends java.rmi.Remote
{
    public void sendData(Object presentationObject,String dataName) throws RemoteException;
    public void stop() throws RemoteException;
    public void register(ConsumerProxy cp) throws RemoteException;
}
```

```

    }

    public interface ConsumerProxy extends java.rmi.Remote
    {
        /* Callback method */
        public void notify(Object presentationObject, String dataName) throws RemoteException;
    }

```

My implementation of the distributed InfoBus successfully extends the event service from the local JVM to multiple JVMs. This distributed event service solution provides a more flexible information exchange mechanism for the OSA than a traditional signalling-based approach. At present, the implementation of the signalling system is OS-dependent while the event service is OS-independent. This greatly simplifies the requirements for information-sharing amongst service providers, which is considered to be the greatest challenge to supporting an open service concept. Compared with other distributed event service proposals such as JINI Distributed Event Service, my design is much more flexible in that it is neutral to the underlying distribution technologies [47].

5.6 Service Layer Implementation

5.6.1 Intelligent Service Proxy

The Service Layer implementation recommended in this thesis tries to reconcile two areas of conflict – flexibility and manageability. In the Open Service Architecture, a service's end users do not talk to the service provider directly, as is the case at present. Instead, all services are realized through service proxies (service node) on the network layer. This approach brings flexibility and scalability to the system, but it also causes tricky management problems [48]. Unlike a simple client code, system control functionality is user-dependent, which means that the service logic and user data are subject to change based on the situation of the user state and service process. If the application is distributed

to multiple service nodes, each copy of the code is varied in state and changes dynamically. The service provider should keep track of each copy of the code. This requirement is tough to achieve in a traditional centralized service management solution because the system will lose track of the code state sooner or later. In the meantime, once the state and the user/service data have been changed on the service node, the code should be auto-configurable without any recompiling by the service provider. Otherwise, the reliability and service quality of the system will be undermined, since recompiling may cause the service node to restart frequently. In my OSA implementation, I have deployed a simple but reliable mechanism which can synchronize and automatically configure distributed code stubs. Each copy of the distributed code serves as the image object in a proxy pattern and can synchronize automatically with a predefined protocol. The approach is a Half Object Plus Protocol (HOPP) pattern [49]. HOPP allows distributed objects to be accessed in real time, whilst saving the cost of messaging to a remote system. This is achieved by creating a copy of the object, known as a half-object or replicate, in each processing context that requires real-time access to the object. The half-objects are kept consistent with each other by using asynchronous change notifications (events or messages). This synchronization protocol is completely independent of the public interface of the object. HOPP hides the processor boundary from the other objects in the system so that it allows the processor boundary to pass through objects rather than between them. Compared with other distributed patterns such as RPC and remote proxy, HOPP reduces message passing and is more reliable. In case the remote object is temporarily out of service, the half object can still function, thus improving system reliability. In the meantime, the user can define the synchronization protocol for different scenarios among the half objects. This solution

successfully distributes the service management functions onto the network nodes and simplifies the management work in the service layer. For the sake of simplicity and flexibility, my solution just defines a time signature-based synchronization protocol to coordinate the change among half objects. Whenever a new image object is created, a time signature on each of the variables that may be changed is also created. Once these variables have been retrieved, the program will determine whether or not to check the persistence by evaluating the algorithm predefined in the synchronization protocol. If the system determines that the data is trustable, it uses the local copy of data; otherwise, it will call the remote interface to get the data. In this way, the image copy of the code could automatically re-configure its state without recompiling on the server side. Furthermore, the user can choose to pick up the data locally or remotely by changing the persistent protocol.

In this thesis, I have used RMI as the communication technology between the half object and its primitive. RMI enables developers to either get a remote reference to a distributed object, in which all method calls are processed on the server object, or to get a copy of the remote object and invoke it locally. RMI gives clients access to objects in the server virtual machine (VM) in one of two ways: by reference or by value. To access a remote object by reference, the object must be an instance of a class that:

- implements an interface that extends `java.rmi.Remote`
- has a properly generated RMI stub class that implements the same interface
- is properly exported to allow incoming RMI calls.

With a pass-by-reference, all methods on the object are remote calls. That lets the remote object have access to the server's resources and services. Since multiple clients deal with a server reference, there is only one copy of the server object. Any changes made to that

object's state by one client are visible to all other clients. As the call is actually realized remotely, all the problems inherent in network communications – latency, disconnects, and time-outs, for example – still apply.

Clients access an object by value when a remote call results in a return value of an object that implements the `java.io.Serializable` interface. When a pass-by-value is chosen, a copy of the data is passed so that the target method uses a copy, rather than the original data. Any changes to the argument are reflected only in the copy, not the original. Methods invoked on this copy are just like any other Java method – they execute locally, without communicating with the server. With a pass-by-value, none of the network communication problems occur; however, the copy does not have easy access to server-side resources and services. Furthermore, each time a client makes a remote method call to get the object, a new copy is created in the client VM. Since each copy maintains its own state, the changes made in one object cannot be seen by any other object.

The HOPP case requires a blend of those two strategies: some methods execute locally without network latencies when the application is “trustable”, while others invoke methods that execute on the server to realize synchronization.

Figure 21 HalfObject Class Diagram

Figure 21 is the class diagram of my HOPP implementation. The Server interface defines all the methods that supported by a server object. The ServerRemote Interface extends the Server Interface and java.rmi.Remote. A ServerRemote interface is created to separate the server object's interface from its remoteness. The solution is based on the fact that Java does not support multiple inheritances, while allowing an interface to have more than one parent. Now the Server interface is not remote. The remoteness is added in the ServerRemote interface, since it extends both the Server and the java.rmi.Remote. The semantics and behaviour of the ServerImpl have not changed from a hierarchical inheritance (similar to implementing a Server interface which extends the Remote). The server class ServerImpl implements the ServerRemote Interface, while it also delegates to a ServerProxy object which implements the Server interface and java.io.Serializable interface. Since the ServerProxy implements java.io.Serializable, a remote call can return a copy of it. The ServerProxy also implements a Server interface, so it appears the same to a client as a remote reference to a ServerImpl object.

A ServerProxy needs to store a reference to ServerImpl and forward method calls to the reference. In order to achieve this, we create a delegation in the ServerProxy constructor to ensure that every instance created has a reference to a ServerRemote object. That reference will really be an instance of ServerImpl but, since the client doesn't need to know about anything more than the remote interface (ServerRemote), the details of the ServerImpl are hidden from it.

If the client makes a remote method call that returns a ServerProxy, the ServerProxy's data is serialized and a copy of the object is instantiated in the client VM. As RMI serializes the ServerProxy, it notices that the ServerImpl field is a reference to an

object that should be replaced by its RMI stub, so it replaces it. The result is that the ServerProxy instantiated on the client side contains a remote reference to a ServerImpl object and can delegate its calls via RMI to that server object.

Now the half object is ready and it is time to define the protocol. The easiest way to accomplish that is to change the RemoteServerImpl.getServer(String) to return a properly constructed ServerProxy instead of a RemoteServer. The policy can be defined to control the factory method to return a copy or a reference based on the state of the data. The client code doesn't change: it still requests and gets a reference to an object that implements the Server. If returned by reference, it gets the RMI stub to a ServerImpl object; if by value it gets a ServerProxy object. As the client only deals with the Server interface, the substitution is invisible to it.

The benefit of the HOPP pattern is that it provides a reliable synchronization mechanism between the service layer and the network, while retaining flexibility of load balancing through the network. The synchronization protocol is easy to change without influencing the application programs. For example, we can separate the protocol contract from an application by putting the re-sync definition into a configuration file. The application reads the configuration file to determine whether the local copy of data is trustable or not. So according to the traffic we can put different configuration files onto the network. If the workload is heavy on the network, we lengthen the re-sync period to help reduce network traffic generated by remote call.

5.6.2 Abstract-factory based service composition

In the network layer, I have defined an auto-pluggable framework which supports the easy integration of services from different vendors into the Network Node. Based on

this architecture, service providers enter the market much faster than the traditional way since they do not need to set up the required communication infrastructure. What about the service itself? If the new features influence the existing ones, fast service deployment becomes useless. Therefore the service organization issue must be investigated further.

Usually, an abstract collaboration defined in a framework is easily customized by an application through static sub-classing. However, this implies a non-independent development of the application and framework models, and excludes the possibility of dynamically deploying the framework.

The challenge in extending the system comes from the change of service logic when a new type of service is introduced. In most service-provision frameworks, people just use script language to wire service components together. This static approach has the several drawbacks, which are delineated below.

Service configuration decisions must be made too early in the development cycle. This is undesirable as developers may not know *a priori* the best way to co-locate or distribute service components. Forcing developers to commit prematurely to a particular service configuration impedes flexibility and can reduce performance and functionality.

Modifying a service may adversely affect other services: The implementation of each service component is tightly coupled with its initial configuration. This makes it hard to modify one service without affecting other services. It is especially noticeable when software components come from different service providers.

The service logic should be open for extensions and closed to modifications in that feature semantics in an open service scenario are supposed to be changed more frequently than in the traditional IN architecture. It should be possible to add new services without

making any changes to the rest of the applications. I propose an Abstract Factory pattern to model the service deployment hierarchy [50]. In the Abstract Factory pattern, a single class defines an interface for creating families of related objects, without specifying their concrete types. The subclasses of an abstract factory are responsible for creating concrete classes that collaborate among themselves. An abstract product separates the client from the real product (service) and thereby eases the adaptation, configuration and evolution of class hierarchies, frameworks and applications. The service logic is defined in a structural way and any modification of the service will be restricted from propagating to other services.

In my implementation, I deployed my call forwarding class hierarchy with an abstract factory pattern. For the call forwarding service, there are multiple “forwarding conditions” (i.e. call forward when no answer and call forward when busy, etc.). If all types of call forwarding are placed into one cumbersome block, whenever a new forwarding scenario is introduced, the modification will influence all other call forward services.

Figure 22 Call Transfer based on AbstractFactory Pattern

When the service is modelled with an abstract factory pattern, the forwarding is defined as an abstract product. Each forwarding scenario will be a concrete product which inherits the abstract product and creates the real call forwarding service. If a new transfer service is added, we only need to deploy a concrete class and there is no need for existing call forwarding services.

5.6.3 Component-based Service Independent Building Block

As mentioned in the previous chapters, the service is realized through a component-based approach with JavaBean as the implementation technology. In the traditional IN concept, the feature components are called Service Independent Building Blocks (SIBs). My implementation could be treated as a Bean-oriented SIB implementation. One of the key design challenges for SIBs is how to make a flexible component that preserves the adaptability to support different operation scenarios (i.e. in the Compare operation, it may handle two telephone numbers or two IP address). From a design point of view, each operation scenario could be viewed as a procedure dealing with a specific type of data source. The generality could be achieved by decoupling an operation's implementation details from the required underlying data. In the IN Concept Module (INCM), there are two kinds of data involved in an SIB's operation. One is dynamic, such as Call Instance Data (CID) (Calling/Called numbers); the other is static such as Service Specific Data (SSD) (User Profile). As the natures of these two types of data differ, we should use different strategies to handle them. Although JavaBeans provide some of the standard API for data communication between components, such as the Object Serialization mechanism and Event Notification, these technologies are not suitable for CID data passing. Since CID information is transient and totally application-dependent, its value varies not only with

state but also with type. We cannot simply implement every possibility in our data-parsing program because this solution may generate too much complexity and require that the SIB knows the type and format of the data in advance. This will greatly undermine the generality of components (SIBs) in service deployment. In my design the data sharing is more challenging than in the traditional approach. In the open service architecture, the CID may come from different service providers, so it is even more difficult to predefine the data processing procedure. Since I use InfoBus APIs as the data transmission mechanism between service beans, the data could be passed without caring about the information details of the type and format. In the InfoBus specification, when a data item is sent, the data object needs to implement an ImmediateAccess interface which has a getValueAsObject() method to help process the information details in the data object. When the other side receives the data, the information is parsed out by the data object itself. Thus we can define a generic data-parsing object which hides the details in the CID information, and which calls the data retrieving method in every CID data object to handle the difference in state and type. For SSD, whenever there is a new service provider (new SSD source) or a new service (new type of operation) introduced, we need to change the application that deals with SSD. My solution for this problem is to allow the bean to carry out such an update at runtime, without influencing the existing features. Except for the data transparency, the operation also requires the distributed computing technology to provide data access among service components (location transparency). One of the major intentions of this thesis was to define the Adaptive Layer in the service semantics. Six SIBs were implemented in order to realize the universal call forwarding service and each of them was in the form of a JavaBean Component. Four of the SIBs are defined in the ITU-T

Capability Set 1 (CS-1) service including Compare, Screen, Translate and DataManagement. Another SIB – the Data Access SIB – is my specific design to fit the Open Service Architecture. It performs the role of the adaptive layer in the OSA architecture. A Sock Bean was also defined, which plays the role of the User Interaction Bean to coordinate interaction with end users. The reason it is not called a User Interaction Bean is because some other applications such as EUT also execute some User Interaction functionalities. There is no Basic Call Processing (BCP) SIB implementation and in theory the Internet plays the role of BCP. Figure 23 shows the class diagram of a Screen Bean component.

Figure 23 Screen Bean Class Diagram

Each bean contains three major processing components: EventHandler, OperationController and Operation. The EntryEvent is generated by other beans and is the entry point of each operation. The event contains a DataItem object which wraps up the Call Instance Data (CID) information. As mentioned above, the CID data is read out by the CID data object itself. Each CID information contains four fields: CIDName, CIDType, CIDValue and OperationType (e.g. indicates screening of the call number in a Screen Bean). Since the CID includes describing information for the data it carries (CIDName,

CIDType), the EventHandler does not need to be bean-specific and the data is handled transparently. For example, in a Screen Bean, the screen operation for telephone number and IP address is supported. The number can be represented as an integer whilst the IP address is in String format. Their CID DataItems should be:

CIDName:telephone_number CIDType:int CIDValue:1234567 OperationType:Outgoing_Traffic

CIDName:IP_Address CIDType:String CIDValue:1.2.3.4 OperationType:Incoming_Traffic

Through such a mechanism, each bean can support heterogeneous lists in arguments and various data types in the CID. Once all the necessary information has been parsed out by a DataParser within the EventHandler, the EventHandler requests the SessionManager to create a session for the operation. Each session contains an OperationController and an Operation object. The OperationController is bean-specific since it manages a database (in my implementation a vector was used for simplicity) which helps to locate the SSD source through CID and OperationType. In the real world, a difficult situation arises which requires the periodic updating of the OperationController to support a new OperationType and SSD source. In this circumstance, it cannot simply be assumed that the OperationController is generic enough to include all possible OperationType and SSD sources. To support situations like this, my implementation allows the OperationController to add new OperationTypes and SSD sources at runtime. The OperationController has a vector of SSDMapper objects which act as a prototype of the SSD information. When an operation request arrives, the OperationController calls its LocateSSD() method to process the CID data. The LocateSSD() method consequentially calls HandleCID() methods in each SSDMapper object to see whether the OperationType is recognized or not. The following is a snapshot of the OperationController code.

public class OperationController implements Runnable

```

{
    ...
    static int session_id=0; //ensure each session has a unique session id.
    Vector SSDPrototypes; //keep all the SSDMapper prototypes.
    ...
    public void addSSDType(SSDMapper prototype) //expand Prototype List to include
    {
        //new SSDMapper Prototype.
        SSDPrototypes.addElement(prototype);
    }
    protected SSDInfo locateSSD(CIDData cid) //Check if we recognize requested SSD
    {
        //information
        SSDInfo ssd=null;
        for(int i=0; i<SSDPrototypes.size(); i++)
        {
            SSDMapper map=(SSDMapper)SSDPrototypes.elementAt(i);
            if(map.handleCID(cid)) //For the sake of simplicity, SSDMapper just
            {
                // checks the operation type item to determine
                ssd=map.getSSDInfo(); // the retrievability of SSD information.
                break;
            }
        }
    }

    public void run()
    {
        while ( true)
        {
            CIDDataItem cid=getCID();
            if( (SSDInfo s_info= locateSSD(cid)) != null) //If we can get SSD info
                createSession(cid,s_info,session_id); //we create the session
            ...
            //otherwise execute lookup
        }
        //procedure
    }
}

```


If the `HandleCID()` returns true, the `OperationController` creates a `SSDProxy` which represents the SSD information on the local site. In the meantime, the `OperationController` creates an `Operation` object which executes the operation. Once the SSD and CID information is ready, the `Operation` object issues the operation and generates the result in the form of an event notification. If the `HandleCID()` returns false (meaning that the specific operation is not supported by the current bean), the `OperationController` asks the `DataManagement Bean` to help find the SSD source. The `DataManagement Bean` keeps a list of SSD data sources and queries each of them as to whether it contains the SSD information requested. If it does, it returns a lookup success event and notifies the bean to add the new `OperationType` by calling the `addOperationType()` method in the `OperationController`. Then the `OperationController` reissues the `LocateSSD()` method to repeat the above process to retrieve the SSD information. The `OperationController` also defines the policy to which the operation should apply. The policy information is contained in an `OperationLogicControl` class. The `Operation` class contains the details for each bean's operation (e.g. screen a data item in a list in a `Screen Bean`, or compare two data items in a `Compare Bean`). The result is generated in the form of an event notification by the `EventDispatcher` class, based on the policy defined in `OperationLogicControl`.

In this thesis, three operations defined in ITU-T Capability Set 1 (CS1) were implemented. These were `Compare`, `Screen` and `Translate`, together with the structure of the bean, making them easy to expand. For example, to define a bean to support a new type of operation, we just need to extend the `OperationController` and `OperationLogicControl` class and put the operation-specific information in the subclasses. Aside from operation beans I also defined a `Socket Bean`, which plays the role of User-Interaction functions. In my

implementation, the interaction with end-users is achieved through a socket. In order to fit JavaBean's event-oriented communication, I use a Socket bean to map the socket operation into a Java event. The code below shows how to generate a DataReceiving event when sending information through a socket.

```

while(connected == true) {
    try { //Assume we just send Text-based information
        String data = stin.readLine();
        if(data != null)
        {
            notifyRcv(data);
        }
        else
        {
            System.out.println("client is close");
            close();
        }
    }
    catch (IOException ioe) {
        System.out.println("I/O Error");
        close();
    }
}
... ..
private void notifyRcv(String data)
{
    Vector l;
    SockRcvEvent sre = new SockRcvEvent(this, data);
    synchronized(this) {
        l = (Vector)SockServerListeners.clone();
    }

    for(int i = 0; i < l.size(); i++) {
        SockServerListener sl = (SockServerListener)l.elementAt(i);
        sl.rcv(sre);
    }
}

```

To build up a specific service with SIBs, we must create an SIB chain according to the service requirement (in my case, it is service logic). An SIB chain starts at POI (Point of Initiation) and ends at POR (Point of Return). In the traditional IN definition, an SIB chain is launched by the BCP SIB and terminated at the corresponding POR in BCP. In my

case, there is no specific BCP implementation because the call processing is realized through a multi-layer structural method. For the sake of convenience to describe deployment procedure, the EUT is treated as a BCP SIB. In theory, my JTAPChat implementation includes some of the BCP function (call set-up) as well as the User Interaction capabilities. Table 2 is a summary of SIBs as well as the POI and POR currently defined in my implementation.

	SIB	Functionality	POI	POR
Operation	Compare	Determin if Two values are equal	Call Originated,	Continue with Existing Data
	Screen	Check if the data is the member of List	Address Presented	
	Translate	Change state info into config info	Address Analyzed	Proceed with new Data,
Management	Data Management	Data persistence	Issue Call Setup	Handle Transit
			Busy Online,	Enable call party handling
	Communication	Provide unified data exchange	On the Web	
Utility	User Interaction	Response User Action	No Answer	
BCP	EUI	Call Control	Accept Call	Initiate Call
			Active State	
			End of Call	

Table 2 Operation, POI and PORs of SIBs

5.7 Architecture Test

In this thesis, a Universal Call Forwarding has been implemented as the pilot service to demonstrate an Open Service Architecture design. The detailed, logical process was introduced at the beginning of this chapter. The section below will focus on how different components are interconnected to realize service functionality. The services defined include Internet Call Waiting (ICW) and Call Forwarding on specific number

(based on time). Figure 24 defines the service logic of universal call forwarding. To realize the service, three different IP addresses are required to deploy the EUTs. One IP address serves as Calling party; the other two play the roles of called party and transferred party. The service was set up on two networked computers. As one of the computers contained two network interfaces, one EUT was used to simulate both the Called Party and the Transferred Party.

Figure 24 Service Logic of Universal Call Forward

Figure 25 shows the detailed configuration of the platform. The computer has two IP addresses –192.168.0.1 and 24.156.231.80 respectively. The first address serves as the

Called Party number while the latter is the Transferred number. The process to be started includes:

- **rmiregistry** – rmi service which supports distributed event service
- **StartSPDB** – Service Provider DataBase where the Service Layer intelligence resides
- **CallControlServer** – the processing/control functionality in the Network Layer
- **ChatDemo** – EUT (BCP) functionality in the End-User Layer.

The adaptive layer does not serve as an independent process. Its functionality will be revealed when the data communication occurs between the Service and Network layers.

Figure 25 Called Party and Transfer Party Configuration

For the demonstration, the called party first had to be set to an On-The-Web state. This was achieved by creating an IP call which represents the Internet connection on the called party's EUT. Now when the calling party issued a call, it got the called-party-busy state (Figure 26) .

Figure 26 Calling Party Issues the Call

The Busy state triggered the Universal Call Forwarding service logic. The service logic deployed the Screen SIB to detect whether the called party was on the Internet. As the end user state information was stored on the Service Provider, a Communication SIB was issued to set up the information exchange channel between the service layer and the network layer. If the called party were on the Web, the service logic would ask the service node on the network layer to activate the Internet Call Waiting process. In the network layer, different vendors' Internet Call Waiting implementations are stored in the

ServiceProxy repository. The network layer provides a value-added Internet Call Waiting feature for end users by helping them to choose the cheapest ICW service. The service node (CallControlServer) located the service in its ServiceProxy Repository and activated it. An Incoming Call notification was displayed on the called party terminal. Figure 27, 28 and 29 display the action paths taking place in the service layer, network layer and end user layer.

Figure 27 Network Layer Processing

Figure 28 Service Layer Processing

Figure 29 Called Party End-User Terminal

If the called party chooses to stay on the Web, the service logic moves to the Call Transfer process. The call transfer feature can forward the call to a specific number based on the time the service is activated. The transfer schedule is defined in the form of a configuration file (shown in Figure 25) and can be changed without changing any call transfer source

code. As in the ICW service, Transfer services are also stored in the ServiceProxy Repository and dynamically invoked by the service node. Figure 30 shows the Transfer action process in the network layer. Each Transfer action involves a Compare and Translate operation respectively. As I mentioned before, they have the same operation structure as Screen bean does, so I omit the trace information for these beans for the sake of simplicity.

Figure 30 Call Diversion Processing on Network Layer

If the transfer action is successful, the transferred party's EUT shows alerting information. Figure 31 shows the Information on Called/Transfer Party's EUI.

Figure 31 Transfer Party receives call

Since the EUT is a GUI-based application, I provided a log window to indicate the call setup/disconnect process. Figures 32 and 33 are EUT log windows for the calling and the called/transferred party respectively.

```

$ GenJTAPI: using default hardcoded configuration
$ JtapiObjectNotifier: enqueue events =>
  >> GenProvinServiceEv
$ RmiChatServerObject:connect() invoked from host: 192.168.0.2
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallActiveEv
  >> GenConnCreatedEv | Addr: 192.168.0.2 (remote)
  >> GenConnConnectedEv | Addr: 192.168.0.2 (remote)
  >> GenTermConnCreatedEv | Term: A (remote)
  >> GenTermConnActiveEv | Term: A (remote)
  >> GenConnCreatedEv | Addr: 127.0.0.1 (local)
  >> GenConnInProgressEv | Addr: 127.0.0.1 (local)
  >> GenTermConnCreatedEv | Term: A (local)
  >> GenTermConnUnknownEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtConnEstablishedEv | Addr: 192.168.0.2 (remote)
  >> GenCallCtTermConnTalkingEv | Term: A (remote)
  >> GenCallCtConnUnknownEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenTermConnCreatedEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtConnAlertingEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenConnAlertingEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtTermConnRingingEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenTermConnRingingEv | Term: A (local)

```

Figure 32 Called/Transferred Party EUT Log File

```

$ GenJTAPI: using default hardcoded configuration
$ JtapiObjectNotifier: enqueue events =>
  >> GenProvinServiceEv
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallActiveEv
$ JtapiObjectNotifier: enqueue events =>
  >> GenConnCreatedEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtConnUnknownEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenConnInProgressEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallActiveEv
  >> GenConnCreatedEv | Addr: 127.0.0.1 (local)
  >> GenConnInProgressEv | Addr: 127.0.0.1 (local)
  >> GenTermConnCreatedEv | Term: A (local)
  >> GenTermConnUnknownEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtConnUnknownEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenTermConnCreatedEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenConnCreatedEv | Addr: 24.156.231.80 (remote)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtConnEstablishedEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenConnConnectedEv | Addr: 127.0.0.1 (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtTermConnTalkingEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenTermConnActiveEv | Term: A (local)
$ JtapiObjectNotifier: enqueue events =>
  >> GenCallCtConnUnknownEv | Addr: 24.156.231.80 (remote)
$ JtapiObjectNotifier: enqueue events =>

```

Figure 33 Calling Party EUT Log File

While the Bean-oriented approach to service deployment provides great flexibility for service providers, it also raises the issue of how to seamlessly integrate components from different vendors. It must be ensured that the information moving among beans is standard. Since the beans defined in this study are non-visual, it is necessary to find a reliable way to verify the beans' inter-operability. For this reason the Bean Development Kit (BDK)1.1[51] from Sun was used as the platform to test the interconnections with beans from other vendors. Figure 34 demonstrates the integration of the Compare bean with other beans.

Figure 34 Compare bean Integration within BDK1.1

Figure 34 above shows the Compare bean loaded into the platform and identified by the tool (list on the bottom of left menu). In the test scenario, the Compare bean interoperates with three other beans – one jogger bean and two OurButton beans. The Compare bean can generate an event to trigger the method in the jogger bean (i.e. start() and stop()) by starting its method (e.g. isEqual()) which is triggered by an event notification (i.e. the MousePressed Event from the OurButtonBean) from other beans. This test scenario demonstrates that the beans deployed in this study can interoperate with Beans from other vendors.

The implementation of the Universal Call Forwarding service shows that my OSA is able to offer a third party service provisioning capability. It also demonstrates that the

implementation technologies chosen for each layer can co-operate smoothly. Thus my OSA design is feasible and practical.

Chapter 6

SUMMARY

6.1 Conclusion

The chapters above demonstrated the design and implementation process for an Open Service Architecture which supports third party service providers using component-oriented and distributed computing technologies. While some open standard service frameworks such as Java Advanced Intelligent Network (JAIN) have defined a similar functionality, they mainly focus on information exchange by defining open interfaces with legacy systems such as OAM and TCAP APIs. There is no clear description of the new service deployment process and service management issues. One great advantage of the Open Service Architecture described in this thesis is that the control capability of the system is balanced and distributed over a number of service nodes across the network instead of in a centralized system. Each node can easily be put into service or change its service functionality by downloading and integrating the specific code from service providers.

This thesis not only defined the unified service components management mechanism but also deployed a messaging framework – event-oriented messaging. An event-driven messaging mechanism is independent of the underlying signalling system, so that it is easy for a service provider to integrate components from different sources. Unlike the Javabeans' Event Service, the event mechanism described in this study can be extended across multiple Java Virtual Machines (JVM). Furthermore, my event framework is independent from the underlying distribution technology. The framework provides an open

interface for users to choose their preferred technology (i.e. CORBA or RMI) and plug into the framework as a standard component. Since such event-based messaging system share the same type of information exchange mechanism with underlying Call Control applications (JTAPI), end user action (event) can be directly transmitted to upper layer without any translation. It makes the whole system more efficient and adaptive.

The service framework defines the Service Provider as one part of the model, thus providing answers to the current deregulation and opening trend. It also provides standard interfaces that allow federation and internetworking between heterogeneous and multi-vendor systems. With the separation of the service provider and the service carrier, service components from different vendors can be wired together through standard event-driven messaging in the service node. In the traditional IN approach, a service can only be deployed using components from the same vendor. Now the user can subscribe to desired services from different service providers without changing the service carrier.

Besides the open service provisioning architecture, this study also considers code reuse to be an important issue. In the IN concept model, reuse is approached by the Service Independent Build Blocks (SIBs). An SIB-oriented solution realizes this idea in an algorithmic way. In my implementation work, I propose an Object-Oriented solution based on JavaBean technology. Unlike the general approach that causes the derived or inherited classes to keep building on top of the generic class, another scheme was used on this study. This does not mean that OO technology is disposed of completely, but rather that the executable entity is reused. An active development idea is shown using Beans. The Beans are always kept active when they are used or reused without any recompiling. Bean objects are deployable when they are alive, and relationships are built upon the current active states.

Last but not least, the research on service build-up process focused on service logics instead of service entities as other projects did. An automatic extendable framework was proposed, based on Java Reflection and Serialization APIs. With my framework, a service provider can easily integrate services from multiple vendors and develop new services using other service providers' existing services. Therefore this thesis achieved two levels of reuse. One is to integrate the existing feature to build up a new service (SIB-oriented approach) which takes place in the deployment phase. Another is to create a new service with existing services, which is achieved at runtime.

In conclusion, it is imperative to mention that service provisioning in the IP network is still under active study and there are other projects in our group which provide for similar functionality. My work presented in this paper uses, discusses and attempts to find a working solution based on these concepts and ideas.

6.2 Future work

Some topics were beyond the scope of this research study but are worth further exploring, such as improving the performance of event notification mechanism for InfoBus and how to integrate the Software Components implemented other than by using JavaBean technology. There are some performance issues with my distributed InfoBus that could be optimized. One of the major constraints is the event dispatching mechanism. Chained event sources or a large number of clients may cause the generation of a large number of temporary objects during the process of sending out the event notifications (the most common synchronization technology is to clone the event listener list which is usually implemented in the vector). It is possible to make notification more efficient by reducing the number of temporary objects. Just before this thesis was completed, SUN announced

the latest version of JDK-JDK1.3. One of the most promising extensions is the introduction of Dynamic Proxy API [52]. Dynamic proxy classes allow applications to create new classes at runtime which implement a set of interfaces. Calls to interface methods are redirected to the instance of the dynamic proxy class, which controls the behaviour of method invocations. In other words, dynamic proxy classes permit programmers to implement a type-safe, reflection-based dispatch of method invocation on objects implementing one or more interfaces at runtime. This is an interesting feature for hot plugging of components based on different implementations, such as dynamic switchable protocol stacks in the OSA.

REFERENCE:

1. Mark Finkelstein, Jim Garahan, "*The future of the Intelligent Network*", IEEE Communications Magazine, June, 2000.
2. ITU-T Recommendation H.323 "*Packet Based Multi-media Communication System*", Geneva Switzerland, Feb., 1998.
3. Handley, M., H. Schulzrinne and E. Schooler, "*SIP: Session Initiation Protocol*", IETF Draft, draft-ietf-mmusic-sip-04.txt, Nov. 11, 1997.
4. Constant Gbaguidi, Jean-Pierre Hubaux, Giovanni Pacifici and Asser N. Tantawi "*An Architecture for the Integration of Internet and Telecommunication Services*" OpenArch 1999.
5. Tang, J., White, T., Pagurek, B., Glitho, R. "*Advanced Service Architecture for H.323 Internet Protocol Telephony*". Computer Communications, special issue on Mobile Software Agents for Telecommunications Applications, Vol.23 No.8, April 2000, pp 740-753.
6. ITU-T Q.1200 Recommendation Q.1200 "*General series Intelligent Network Recommendation structure*", July, 1997.
7. ITU-T Recommendation Q.1211 "*Introduction to intelligent network capability set 1*", March, 1993
8. ITU-T Supplement 27 to Series Q Recommendations - Technical Report "*Overview of signaling and protocol framework for an evolving environment*", Dec. 1999.
9. ITU-T Recommendation Q.1237 – "*Extensions to intelligent network capability set 3 in support of B-ISDN*", March, 2000.
10. S. Patrack and L. Conroy, "*The PINT Service Protocol: Extensions to SIP and SDP for IP Access to Telephone Call Services*" Internet Draft, IETF, Feb. 2000.
11. Jonathan Lennox, "*Implementing Intelligent Network Services with the Session Initiation Protocol*". Columbia University Technical Report CU-CS-002-99, New York, NY, January 1999.

12. ETSI TR 101 300, "*Telecommunication and Internet Protocol Harmonization Over Network(TIPHON): Description of Technical Issues*" Oct., 1999
13. ETSI ETS 300 175-5 ed.3, "*Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 5: Network (NWK) layer*", 1997.
14. "*MOBILE COMMUNICATIONS - STANDARDS - GSM based UMTS a leading Member of the IMT2000 family*", ITU-T document, Feb.,2000.
15. ETSI TS 122 101 V3.11.0, "*Universal Mobile Telecommunications System (UMTS); Service aspects; Service principles*", Oct., 2000.
16. ETSI ETS 300 392-1 ed.1, "*Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D);Part 1: General network design*", Feb., 1996.
17. D. Tait, J. de Keijzer, and R. Goodman, "*JAIN: A New Approach to Services in Communication Networks*", IEEE Communications Magazine, January 2000
18. Ravi Raj Bhat and Rajeev Gupta, "*JAIN Protocol APIs*", IEEE Communication Magazine, January 2000.
19. Sun Microsystems, "*JSR #000024 JAINTM Service Provider API*",
http://java.sun.com/aboutJava/communityprocess/jsr/jsr_024_sna.html
20. S. Beddus, G. Bruce, and S.Davis, "*Opening Up Networks with JAIN Parlay*", IEEE Communication Magazine, January 2000.
21. Mahbub Hassan, Alfandika Nayandoro and Mohammed Atiquzzaman, "*Internet Telephony: Services, Technical Chanllenges, and Products*", IEEE Communication Magazine, April 2000.
22. Rob Brennan, Brendan Jennings, Conor Mcardle, and Thomas Curran, "*Evolutionary Trends in Intelligent Network*", IEEE Communication Magazine, June 2000.
23. Tsun-Chieh Chiang, Janet Douglas, and Kumar Venuri, "*IN Service for Converged(Internet) Telephony*", IEEE Communication Magazine, June 2000.
24. F. Anjum, et al., "*ChaiTime: A system for rapid creation of next generation telephony services using third-party software components*", Proc. IEEE OpenArch, March 1999
25. Roch H. Glitho, "*Advanced Service Architectures for Internet Telephony: A Critical Overview*", IEEE Network, August 2000.

26. Jingrong Tang, "*Mobile Agent Based Advanced Service Architecture for H.323 Internet Protocol Telephony*", Master Thesis, System and Computer Engineering Department, Carleton University, 2000.
27. R. Jain, F. Anjum, P. Misser, and S. Shastry, "*Java Call Control, Coordination, and Transactions*", IEEE Communication Magazine, January 2000.
28. Microsoft White paper, "*IP Telephony with TAPI 3.0*", April, 1999.
29. Sun Micro Systems "Java Telephony API v.1.3 Specification",
<http://java.sun.com/products/japi>
30. Bilhanan Silverajan, Jarmo Harju "*An Experimental Environment for Distributed Intelligent Network Services*", IN'2000, February 2000.
31. Signal Computing System Architecture
<http://www.scsa.org/>
32. Pamela Zave, Michael Jackson, "*A Component-Based Approach to Telecommunication Software*", IEEE Transaction on Software Engineering (TSE), 1998.
33. Sun Microsystems. Javabeans API specification, version 1.01, July 1997.
<http://java.sun.com/beans/docs/beans.101.pdf>
34. Tom Valesky, "*Enterprise JavaBeans—Developing Component-Based Distributed Applications*", Addison-Wesley, 1999.
35. A. Thomas, "*Enterprise JavaBeans: Server Component model for Java*", Patricia Seybold Group White Paper, Dec. 1997.
36. Richard Monson-Haefel, "*Enterprise JavaBeans*", O'Reilly, 1999.
37. Mike Day. (1998). "*Contrasting JavaBeans and Enterprise JavaBeans, A Developer's Round Table Discussion*". IBM Online Library.
38. OMG. The CORBA Component Model, Final submission
<http://www.omg.org/>
39. Donald Box, "*Essential COM*", Addison Wesley, 1998.
40. JP Morgenthal. (July 1999). "*Microsoft COM+ will Challenge Application Server Market*". Microsoft COM+ Whitepaper, July 1999.
41. Chappel, "*Understanding Windows 2000 Distributed Services*", Microsoft Press, 2000.

42. Donna Bergmark and S.Keshav, "*Building Blocks for IP Telephony*", IEEE Communication Magazine, April 2000.
43. IBM JTAPIChat homepage
<http://alphaworks.ibm.com/tech/jtapichat>
44. Bill Venners, "*Design with Dynamic Extensions*", Javaworld Design Techniques, January 1999.
<http://www.javaworld.com/javaworld/jw-01-1999/jw-01-techniques.html>
45. InfoBus1.2 specification,
<http://java.sun.com/products/javabeans/infobus/spec12/IB12Spec.htm>
46. Reaz Hoque, "*Connecting JavaBeans with InfoBus*", Wiley Computer Publishing, 1999.
47. Sun Microsystems. Jini™ Distributed Event Specification, version 1.0, January 1999,
<http://www.sun.com/jini/specs/index.html>
48. Linda Rising, "*Design Patterns in Telecommunication Software*", pp-131, Cambridge University Press, 2000
49. U. Black *The Intelligent Network*, Prentice Hall, 1998.
50. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.
51. Sun Microsystem, BDK home page.
<http://java.sun.com/products/beans/>
52. Sun Microsystem, Dynamic Proxy Classes
<http://java.sun.com/products/jdk1.3/docs/guide/reflection/proxy.html>