

Mobile Agent Based Advanced Service Architecture for H.323 Internet Protocol Telephony

by
Jingrong Tang, B.Eng.

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

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada K1S 5B6

July 7, 2000

© Copyright 2000, Jingrong Tang

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

“Mobile Agent Based Advanced Service Architecture for H.323
Internet Protocol Telephony”

submitted by Jingrong Tang, B.Eng.

in partial fulfillment of the requirements for the degree of
Master of Electrical Engineering

Department Chair: Rafik Goubran

Thesis Supervisor: Bernard Pagurek

Thesis Supervisor: Tony White

Carleton University

July 7, 2000

Abstract

Internet Protocol (IP) based communication is fast becoming a viable alternative for voice communications. While the Intelligent Network (IN) represents the world wide accepted basis for the uniform provision of advanced telecom services, mobile agents offer unique opportunities for structuring and implementing open distributed service architectures, facilitated by the dynamic downloading and movement of service code to specific network nodes. In this thesis, a new service architecture for ITU-T standard H.323 based IP Telephony is proposed. The new service architecture uses Mobile Agents and Jini as enabling technologies. It also makes use of the existing architectural concepts of IN. This IP service architecture enables telecom services being deployed through mobile service agents on a per user basis, which results in several advantages when compared to centralized service architectures. The thesis demonstrates that the proposed service architecture not only can accommodate existing services but is flexible and extensible enough to accommodate a wide variety of future services. In addition, it is shown that the new architecture addresses the full management life-cycle of advanced services, from open third party creation to subscription and utilization, and ultimately to maintenance and withdrawal.

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisors, Prof. Bernard Pagurek and Dr. Tony White, for their invaluable guidance, inspiration, encouragement and patience throughout the research work and the preparation of this thesis. Their constant striving for creative and rigorous research influenced many aspects of my life. I benefitted greatly from their profound knowledge and experience in the field.

I am grateful to all my colleagues who are in the Network Management lab, for their help. I would also like to thank Mr. Roch Glitho and Ericsson Research Canada for providing the thesis project and their financial support.

Finally, I would like to thank my family, without their immeasurable love, support and encouragement, I would not have achieved what I did.

List of Figures

Figure 2.1	H.323 Scope	7
Figure 2.2	Single Gatekeeper Routed Call Signaling.....	10
Figure 3.1	IN Approach.....	20
Figure 4.1	Jini Architecture.....	27
Figure 5.1	MA Based Advanced Service Architecture for IP Telephony	39
Figure 5.2	SCC Uploads VPN Service Proxy onto the Lookup Service.....	41
Figure 5.3	ESC Downloads VPN Service Proxy Object.....	42
Figure 5.4	ESC Uploads an Enterprise Service Proxy onto an Enterprise Lookup Service.....	42
Figure 5.5	User Service Agent	44
Figure 5.6	Services Subscription and Activation	46
Figure 5.7	VPN Service Subscription	50
Figure 5.8	CFU (End-to-end call Signaling) Invocation	52
Figure 5.9	CFU (Gatekeeper Routed Call Signaling) Invocation	54
Figure 5.10	CT (Gatekeeper Routed Call Signaling) Invocation	56
Figure 5.11	VPN Service Invocation - OGA/OGR (Gatekeeper Routed Call Signaling)	58
Figure 6.1	Hierarchical Component Structure of Grasshopper.....	60
Figure 6.2	Multi-Protocol Support	65
Figure 6.3	Location Transparent Communication of Grasshopper	65
Figure 6.4	Access of an Agent's Methods.....	68

Figure 6.5	Basic Call Model.....	72
Figure 6.6	Class Diagram.....	75
Figure 6.7	Component Service Creator Proxy Implementation.....	78
Figure 6.8	Enterprise Service Creator Proxy Implementation.....	78
Figure 6.9	Service Component Creator.....	78
Figure 6.10	Enterprise Service Creator Implementation.....	79
Figure 6.11	SMU Implementation.....	79
Figure 6.12	User Service Agent Creation Code Example.....	80
Figure 6.13	The UserServiceAgent ID is Stored in the SMU.....	80
Figure 6.14	A Sample Implementation of Method live() of USA.....	81
Figure 6.15	Sample Implementation of action() Method.....	82
Figure 6.16	Advanced Service Subscription GUI.....	83
Figure 6.17	Service Customization Form for CFU.....	83
Figure 6.18	TalkServer Message Log.....	84
Figure 6.19	Caller Messaging Information Log.....	84
Figure 6.20	Message log of Forwarded-to Terminal.....	85
Figure 6.21	Message log of the forwarded-to Terminal.....	85
Figure 6.22	USA Construction.....	86
Figure 6.23	USA Moving to the End User's Agency.....	86
Figure 6.24	Instantiation of a CA.....	87

Table of Contents

Abstract.....	I
Acknowledgement.....	II
Table of Contents	III
List of Figures.....	VI
Chapter 1 Introduction.....	1
1.1 Motivation and Objective.....	1
1.2 Thesis Contribution.....	2
1.3 Thesis Organization	3
Chapter 2 IP Telephony Related Standards	4
2.1 ITU-T Recommendation H.323.....	4
2.1.1 H.323	4
2.1.2 H.225.0/RAS/Q.931/H.245.....	7
2.2 A Close Look at H.323 Service Architecture	11
2.2.1 Service Life Cycle and Service Architecture Requirements.....	11
2.2.2 Limitations of H.323 Supported Service Architecture	12
Chapter 3 Existing Service Architectures for Traditional Telephony.....	15
3.1 Telecommunications Information Networking Architecture	15
3.2 Telecommunications Management Network	17
3.3 Intelligent Networks.....	18
3.3.1 Intelligent Networks and Service Independent Building Blocks.....	19
Chapter 4 Enabling Technologies for A New Advanced Service Architecture ...	23
4.1 Mobile Agent Technology	24

4.2 Jini	27
4.2.1 The Discovery Process.....	29
4.2.2 The Join Process	30
4.2.3 The Lookup Process.....	31
4.2.4 Client / Server Interaction.....	31
4.2.5 The Jini Technology Programming Model.....	33
4.3 JavaBeans.....	35
Chapter 5 Mobile Agent Based Advanced Service Architecture.....	37
5.1 A New Advanced Service Architecture Based on Mobile Agent.....	38
5.2 Service Subscription and Utilization Using Mobile Agents.....	41
5.3 Usage Scenarios for the New Service Architecture.....	49
5.3.1 VPN Service Subscription	50
5.3.2 Call Forwarding Unconditional Invocation	51
5.3.2.1 CFU Invocation Using End-to-end Call Signaling	51
5.3.2.2 CFU Invocation Using Gatekeeper Routed Call Signaling	53
5.3.3 Call Transfer Invocation	54
5.3.4 Outgoing Call Allowance/Outgoing Call Restriction Invocation.....	56
Chapter 6 Implementation of the MA Based Advanced Service Architecture	
Using Grasshopper	59
6.1 Grasshopper - Mobile Agent Programming Environment.....	59
6.1.1 Grasshopper Platform	59
6.1.1.1 Distributed Agent Environment.....	60
6.1.1.2 Agencies	61
6.1.1.2.1 Core Agency	61
6.1.1.2.2 Places	63
6.1.1.3 Regions	63
6.1.1.4 Communication Concepts.....	64
6.1.1.4.1 Multi-protocol Support	64

6.1.1.4.2 Location Transparency.....	65
6.1.1.5 Agent Development	66
6.1.1.5.1 Accessing the Grasshopper Functionality.....	66
6.1.1.5.2 Agent Class Structure	67
6.1.1.6 Remotely Accessible Functionality	69
6.1.1.6.1 The AgentSystem Interface.....	69
6.1.1.6.2 The AgentSystemListener Interface	70
6.1.1.6.3 The RegionRegistration Interface.....	70
6.2 Grasshopper’s Limitations	70
6.3 Implementation of the New Service Architecture.....	71
6.3.1 Design Issues	71
6.3.1.1 Platforms and Enabling Technologies	71
6.3.1.2 Implementation Basis.....	72
6.3.2 Class Diagram.....	74
6.3.3 Service Subscription and Realization Using Jini	77
6.3.4 Service Utilization Realization Using Grasshopper	80
6.3.5 Simulation Results	82
6.3.6 Limitation of the Simulation.....	87
Chapter 7 Conclusions And Future Work.....	89
7.1 Conclusions.....	89
7.2 Future Work	90
Appendix A Acronyms	92
Appendix B References	94

Chapter 1 Introduction

1.1 Motivation and Objective

Internet Protocol telephony or IP telephony [1] refers to communication services - voice, facsimile, and/or voice-messaging applications - that are transported via the Internet, rather than the Public Switched Telephone Network (PSTN). In February 1995, when Vocaltec, Inc. introduced its Internet Phone software [2], the possibility of voice communications traveling over the Internet became a reality for the first time. Because of the low price and efficient use of bandwidth, it has progressed rapidly in a relatively short period of time.

As the Internet is an open, distributed and evolving entity, it is expected that there will be many extensions to IP telephony. Now it is still a challenging task to provide high quality and reliable voice calls over internet, but it is evident that providing advanced voice services will make IP telephony more competitive in the telephony market and more appealing to consumers. Because the existing IP telephony advanced service architecture is not very suitable for the provisioning of advanced IP telephony services, sound service architectures are needed for the control and management of services. Designing a new service architecture which is more open and more flexible becomes the main motivation of this thesis.

The objective of this thesis is to investigate the current IP telephony protocols and analyze their pros and cons for IP telephony service provisioning, and to propose a new service architecture for advanced IP based telephony services.

1.2 Thesis Contribution

In this thesis, H.323 [3] - an IP telephony protocol which was under review when this project started is analyzed. The emphasis of the analysis is placed on its suitability for supplementary services such as Call Redirection, Call Transfer, etc. which are offered by today's traditional telephony carriers. Other H.323 related protocols such as H.225 [4], H.245 [5], H.450 [6, 7, 8], etc. are also reviewed to give a whole picture of the H.323 protocol. Through the analysis, it is found that H.323 based service architecture will be quite complicated. Under H.323, each individual service needs a separate specification. Further more, service creation, deployment and withdrawal are not clearly addressed. A new service architecture based on Mobile Agent (MA) technology and Intelligent Networks (IN) is proposed. The proposed service architecture will be more open and flexible for provisioning advanced services of IP telephony.

Existing service architectures are also studied. These architectures include IN which is commonly used in PSTN for advanced telephony services, Telecommunication Information Network Architecture (TINA) which is a service architecture to provide means to build and support telephony services, and Telecommunication Management Network (TMN) which aims at the operation, administration, main-

tenance and provisioning of telecommunications networks and services in today's open, multivendor environment.

For the implementation of the new service architecture, Jini and JavaBeans, two complementary technologies developed by Sun Microsystems, were investigated and Jini was used for service subscription and enterprise service customization.

1.3 Thesis Organization

The rest of the thesis is organized as following. In chapter 2, H.323 standard and other IP telephony related protocols are analyzed and compared. This is followed by an analysis of the limitations of the existing service structures defined in these standards. The basic concept of IN is explained and some existing service architectures are briefly described in chapter 3. In chapter 4, the enabling technologies - Mobile Agents (MAs), Jini and JavaBeans are discussed, of which MAs and Jini are used in the newly proposed IP telephony service architecture. JavaBeans might be one of the possible technologies to implement the advanced service classes. Then the MA based new IP telephony service architecture is introduced in chapter 5, this is followed by some scenarios of using this MA based service architecture for service subscription and service invocation. Chapter 6 gives an overview of Grasshopper, the platform used for developing MA based applications, in this case, the MA based service architecture. The design and implementation of the MA based service architecture is then detailed. And finally, chapter 7 summarizes the thesis with conclusions. It also presents directions for related future works. References and acronyms are listed at the end of this thesis.

Chapter 2 IP Telephony Related Standards

Currently, there are two major protocols that address IP telephony issues. One is H.323 from International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) and the other is Session Initiation Protocol (SIP)[9] from Internet Engineering Task Force (IETF). The first version of H.323 specification was approved in 1996 by the ITU-T's study group 16. Version 2 was approved in January 1998. The standard is broad in scope and includes both stand-alone and embedded personal computer technologies as well as point-to-point and multi-point conferences. Compared to H.323, SIP is rather lightweight, it reuses many of the header files, encoding rules, error codes, and authentication mechanisms of HTTP. The following sections will discuss H.323 related protocols and its service architecture.

2.1 ITU-T Recommendation H.323

2.1.1 H.323

The ITU-T H.323 [10, 11] specification, namely, “Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service” (this title was changed to “Packet-based multimedia communications systems” in 1998), specifies the components, protocols, and procedures to provide

multimedia communications over packet-based networks. One of the many media that H.323 can be applied to is voice over Internet Protocol, i.e., IP Telephony.

Following are some important definitions of H.323:

- **Call:** Point-to-point multimedia communication between two H.323 endpoints. A call begins with the call set-up procedure and ends with the call termination procedure. The call consists of the collection of reliable and unreliable channels between the endpoints.
- **Call signaling channel:** Reliable channel used to convey the call set-up and teardown messages (following recommendation H.225.0) between two H.323 entities.
- **Endpoint:** An H.323 terminal, gateway, or Multiple Control Unit (MCU). An endpoint can call and be called. It generates and/or terminates information streams.
- **H.323 Entity:** Any H.323 component, including terminals, gateways, gatekeepers, Multipoint Controllers (MCs), Multipoint Processors (MPs), and MCUs.
- **Local Area Network (LAN):** A shared or switched medium, peer-to-peer communications network that broadcasts information for all stations to receive within a moderate-sized geographic area, such as a single office building or a campus.
- **Zone:** A zone is the collection of all terminals, gateways, and MCUs managed by a single gatekeeper.

The H.323 components defined by the standard are:

- **Terminals** which provide real time bi-directional multimedia communication with another H.323 terminal, gateway, or MCU. They must support voice communications.
- **Gatekeepers (GKs)** which provide address translation and control access to the local area network (LAN) for H.323 terminals, gateways, and MCUs. gatekeepers can intercept all the call signaling between endpoints and use it to provide “signaling-based” advanced services. They can provide those services that cannot be decentralized and implemented by endpoints. Gatekeepers in H.323 are optional. But if they are present in the network, endpoints must use their services.
- **Gateway** that provides real-time, two-way communications between H.323 terminals in the LAN, or to another H.323 Gateway or non-H.323 terminals. More explicitly, it provides interpretability between H.323 terminals and ITU-T terminals on the circuit switched networks.
- **Multipoint Control Unit (MCU)** that provides the capability for three or more terminals and gateways to participate in a multipoint conference. It may also connect two terminals in a point-to-point conference which may later develop into a multipoint conference.

Figure 2.1 shows the scope of H.323. As shown in the figure, all H.323 entities are connected to the enterprise LAN. They join the gatekeeper’s zone through gatekeeper discovery and endpoint registration process.

The H.323 standard series include H.245 for control, H.225 for connection establishment, H.332 for large conferences, H.450.x for supplementary services, H.235 [12] for security and H.246 [13] for interoperability with circuit-switched services. The encoding mechanisms, protocol, and basic operations are somewhat simplified versions of the Q.931 [14] ISDN signaling protocol.

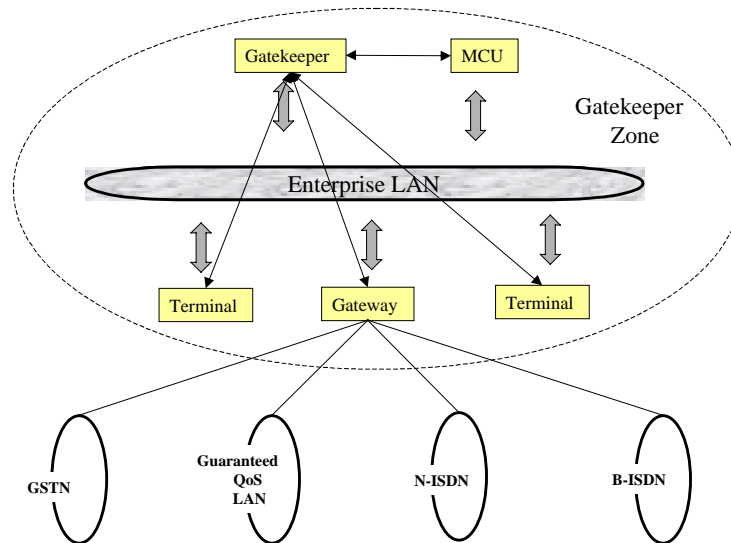


Figure 2.1 H.323 Scope

2.1.2 H.225.0/RAS/Q.931/H.245

ITU-T recommendation H.225.0, named “Line Transmission of Non-Telephone Signals”, describes the means by which audio, video, data, and control are associated, coded, and packetized for transport between H.323 equipment on a packet based network. This includes the use of an H.323 gateway, which in turn may be connected to H.320 [15], H.324 [16], or H.310/H.321 [17] terminals on N-ISDN, GSTN, or B-ISDN respectively. The equipment and procedures are described in H.323 while H.225.0 covers protocols and message formats. Communication via an H.323 gateway to an H.322 [19] gateway and thus to H.322 endpoints for guar-

anteed quality of service (QoS) LANs is also possible. The scope of H.225.0 communication is between H.323 entities on the same packet based network, using the same transport protocol.

When an endpoint is switched on, it performs a multicast discovery for a GK and registers with it. Therefore, the GK knows how many users are connected and where they are located. After GK has been discovered, the endpoint sends a Registration Administration (RAS) Registration Request (RRQ) to the GK. The RAS protocol is the key GK protocol. RAS messages are carried in User Datagram Protocol (UDP) packets exchanged between an endpoint and its GK. The RAS message contains information such as the terminal transport address, user alias, and E.164 telephone number. If the GK accepts the registration, it sends a Registration Confirm message (RCF), otherwise, it sends a Registration Reject (RRJ) message. The GK and its registered endpoints are called a *Zone*.

The H.323 call setup life cycle can be split into three phases:

1. RAS: Anytime an H.323 endpoint wants to make a call, it asks for permission from the gatekeeper by sending a RAS Admission Request (ARQ) message. This message contains the destination alias, the name or phone number of the user the calling party wants to contact and other parameters. The GK may grant the permission for the call by sending back an Admission Confirm (ACF) message containing the actual address associated with called party alias sent along with the ARQ. The GK may also reject the request with an Admission Reject (ARJ) message giving a variety of reasons, such as “not enough

bandwidth” or “security violated”. Therefore, during this phase, the GK accomplishes three different functions: address translation, call authorization and bandwidth management.

2. Q.931: This phase is derived from ISDN end-to-end call setup signaling (SETUP, PROCEEDING, ALERTING, CONNECT) and provides the logical connection between the two endpoints, the calling party and the called party. In H.323, Q.931 is implemented on top of TCP.
3. H.245: As soon as the Q.931 phase is finished, the two end-points exchange their capabilities. During this stage they agree on the nature of the information that will be exchanged through the media channel (audio, video, or data) and its format (compression, encryption, etc.). H.245 is implemented on top of TCP.

After these three phases, the Real Time Protocol (RTP)/Real Time Control Protocol (RTCP) media channels between the two endpoints are opened according to the capabilities exchanged, and actual media communication starts. Data communications are based on the T.120 [20] specification. During the call, dual-tone multifrequency (DTMF) touch tones are transmitted over the LAN through the H.245 User Input Indication message.

Call signaling can be routed through the GK or routed directly between the endpoints. RAS is, by nature, routed by GK. A GK can decide to route Q.931 and H.245 through itself, so that it can act as a proxy between endpoints. If the GK intercepts the call signaling, it can perform call management. The GK maintains a

list of ongoing H.323 calls in order to keep endpoints' state or to provide information to the bandwidth management function. GK is just a signaling entity and cannot be called.

Before any call is made, an endpoint may discover/register with a GK. If this is the case, it is necessary for the endpoint to know the information of the GK it is registering with. It is also desirable for the GK to know the details of the endpoints that are registering with it. For these reasons, both the discovery and registration sequences may contain optional non-standard message parts to allow endpoints to establish non-standard relationships. At the end of this sequence, both GKs and endpoints are aware of the version numbers and the non-standard status of each other.

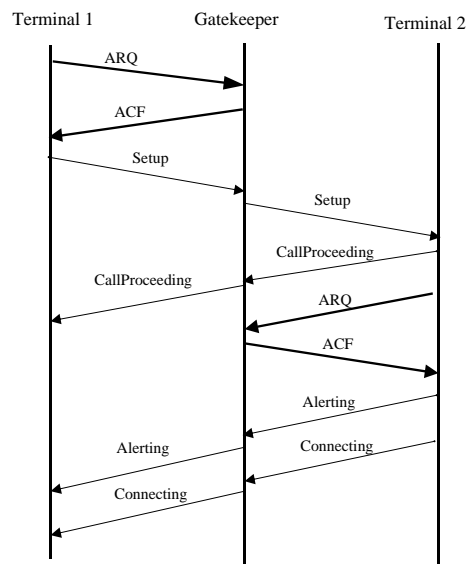


Figure 2.2 Single Gatekeeper Routed Call Signaling

Figure 2.2 shows a signaling sequence using GK routed call signaling. The unreliable channel for registration, admissions and status messaging is called the RAS channel. The general approach to start a call is to send a mandatory admission

request on the RAS channel if the endpoint has registered with its GK, followed by an initial *setup* message on a reliable channel transport address. As a result of this initial message, a call setup sequence commences based on Q.931 operations with enhancements described below. The sequence is complete when the terminal receives in the *Connect* message a reliable transport address on which to send H.245 control message.

Terminals may support optional Q.931 and H.450 messages. These messages shall contain all of the mandatory information elements and may contain any of the optional information elements as defined in Q.931. The H.225.0 endpoint may ignore all optional messages it does not support. Each H.225.0 endpoint shall be able to receive and identify an incoming Q.931 or H.450 message as such. It shall be capable of processing the mandated Q.931 messages, it may also be capable of processing unknown messages without disturbing operation. Each H.225.0 endpoint shall be able to interpret and generate the information elements mandated for the respective Q.931 and H.450 messages.

2.2 A Close Look at H.323 Service Architecture

In this section, a brief analysis of H.323 service architecture is provided.

2.2.1 Service Life Cycle and Service Architecture Requirements

While service architecture requirements for IP Telephony have yet to reach maturity, a set of requirements for Telecommunications Information Networking Architecture Consortium (TINA-C) service architecture described in [21] are gradually being adopted. In this discussion, these requirements are used to evaluate the ser-

vice architecture defined in H.323 and the new service architecture which will be presented later. The service life cycle described by TINA-C service architecture consists of: service construction, service deployment, service utilization and service withdrawal. Each stage of the service life cycle is explained in [21].

The requirements listed below are used to evaluate IP telephony advanced service architectures [21]:

- Support of all life cycle phases
- Support of a wide range of services
- Support of multi-player environments
- Rapid service creation and deployment
- Tailored services
- Independent evolution of services and network infrastructures
- Universal access
- Interworking with other advanced service architectures

2.2.2 Limitations of H.323 Supported Service Architecture

Overall, H.323 is a very complicated protocol. H.323 supplementary services are defined in H.450.x series specifications. For example, Call Transfer is defined in H.450.2 and Call Forwarding is defined in H.450.3. Until February 2000, 8 supplementary services are defined. Although H.323 specifies mechanisms for activating and deactivating services, it does not address service creation, deployment and withdrawal.

With the broad usage and wide adoption of IP telephony, customers will expect more advanced services in addition to the ones defined in H.450.x. With each supplementary service defined in a separate specification, H.323 does not offer a generic specification, adding more advanced services means adding more specifications to the already complicated protocol. It does not support the wide range of services needed to compete in the market place of today, let alone tomorrow. For example, an IP telephony Virtual Private Network (VPN) service could not be implemented using the services defined in the H.450.x standards thus far.

Service creation is not addressed in the H.323 protocol. It is also not clear how a service may be broken down into reusable building blocks, which means that rapid service creation and deployment are likely to become a problem. Additionally, there is little room left for third party service providers, which may lead to unfair competition in the market.

In H.323, service activation and execution are not clearly defined. This leaves room for designers to reuse IN principles or MA technologies for service utilization.

The service architecture supported by H.323 relies on the underlying network infrastructure. In this service architecture, the gatekeeper manages the supplementary services, so it can not be ported to another network infrastructure without building a new service architecture.

Service customization is another very important issue, it provides customers with the ability to tailor the services to satisfy their special requirements. However, cur-

rently, service customization is not addressed in H.323.

According to the H.323 protocol, users can not access services from elsewhere other than their terminals, and interworking with other advanced service architectures is not addressed.

Despite these limitations, there are many products using H.323 as the standard in today's markets. For example, HP OpenCall, Ericsson's H.323 gatekeeper, Gatekeeper Platform and Gateway Platform, H.323-compliant video conferencing endpoints by Intel, RADVision's H.320/H.323 Gateway by Lucent Technologies, Microsoft's NetMeeting 2.0 supporting the H.323 standard for audio and video conferencing, Elemedia's H.323 Protocol Stack, etc., all these products compete for attention in the IP telephony marketplace.

Chapter 3 Existing Service Architectures for Traditional Telephony

In the classical telephony world which is based on circuit switched networks, a number of service architectures have been developed in the last decade, for example, the Intelligent Networks (IN) framework, Telecommunications Management Network (TMN), Telecommunications Information Networking Architecture (TINA), etc. The purpose of these service architectures is to increase the quality and range of services offered in communication networks. In order to compete with classical telephony in today's market, one of the challenges that IP telephony faces is to offer not only the same high quality voice calls, but also a set of call features (i.e., advanced services) which classical telephony offers today. While the high quality of voice calls has not yet been achieved in the IP telephony world, sound architectures are needed for the control and management of advanced services.

In the following sections, the existing service architectures offered in the traditional telephony world will be discussed briefly with the emphasis on IN, which is widely used.

3.1 Telecommunications Information Networking

Architecture

The Telecommunications Information Networking Architecture Consortium (TINA-C) [22] is a multinational worldwide consortium, which aims at defining and validating an “open” architecture for telecommunications services in the emerging broadband, multimedia and “information super-highway” areas. The architecture is based on distributed computing, object orientation, and other concepts and standards from the telecommunications as well as the computer industry, e.g. IN, TMN, and Common Object Request Broker Architecture (CORBA). It is applicable to various networks, broadband (e.g. Asynchronous Transfer Mode (ATM)) or narrowband.

The basic computing architecture is expanded with concepts and principles gathered in what TINA-C called a Service Architecture. This Service Architecture provides means to build services and a service support environment. It can be applied to a wide range of service types including management services, information services, transport services and access services.

The service architecture contains a definition of the stakeholders that need to be considered when defining a TINA-C service and the roles they play. The main roles that are deemed important are those of users, subscribers, network providers, service providers, service/network designers, and service/network managers. A service life-cycle model is also elaborated, in which the need of the service, its construction, deployment, operation, and withdrawal are addressed.

3.2 Telecommunications Management Network

The Telecommunications Management Network (TMN) [23] standard was defined by the ITU-T, to specify management of telecommunications networks. Its principles aim at the operation, administration, maintenance and provisioning (OAM&P) of telecommunications networks and services in today's open, multivendor environment. The TMN principles recommend the use of management networks for the management of telecommunications networks and services. Elements in the telecommunications networks (managed networks) communicate with managing systems (in the managing networks) via well-defined and standardized interfaces. It should be emphasized that these interfaces are more than protocols because they include information models.

The advantage of TMN is that it enables telecommunications companies to integrate legacy systems and newer equipment from different vendors into the same network management structure. TMN functions will be used as building blocks to implement service management, network restoration, customer control/reconfiguration and bandwidth management.

In today's dynamic telecommunication environment, changes are occurring on many fronts. Services and network technologies are advancing rapidly. Competition among service providers is intensifying. Customer's demand for network access and customized services is increasing.

TMN allows telecommunications companies to cut time to market on new services, by separating the management model from the physical details of the net-

work devices. This means that new services can be released without affecting the switching network. It also allows telecommunications companies to save their existing investments, by integrating legacy systems and new equipment into the same TMN network management structure. TMN makes the network more robust, by allowing network management to be distributed and decreasing management traffic over the network. It also makes telecommunications companies more competitive, through the integration of service, management and accounting systems. This streamlines business processes and improves quality of service.

TMN is more focused on the management of the telecommunications networks and services, the proposed service architecture only needs the latter part.

3.3 Intelligent Networks

A primary objective for Intelligent Network (IN) [24, 25, 26, 27] is the control of the definition and deployment of new services to satisfy customer needs. With the existing network infrastructure, switch vendors define the implementation and operation of new services and offer them via traditional software-generic release cycles. These services emphasize availability and operation, resulting in additional training, documentation, trouble identification, trouble reporting and repair costs.

Achieving service ability across the regional network requires a service platform that is switch-type independent, widely deployed, and with uniformed execution environment. To this end, the telecommunications industry is enhancing switch call models to conform to the Bellcore (now Telecordia) Advanced Intelligent Network (AIN) call model. Rapid service deployment will require advances in service

logic creation technology and modification of the existing operations infrastructure to accept programmable operations software packages. Ultimately, OS software programmability will be essential to meet the rapid deployment objectives. Distributed call control and call-processing services enable the optimum placement of service functions within the network. Modularity will allow new services to be quickly and cost-effectively constructed using existing feature components. Cost reduction of network operations represents perhaps the greatest challenges and opportunity for the IN program. The present network operations infrastructure was developed to efficiently deliver mass-market services. Achieving such high efficiencies has taken a long time, as networks have evolved into their current forms. This operation infrastructure, however, is not agile, and it interferes with the objective of achieving rapid service delivery. Operations planners believe that flexibility and high service-deployment efficiency are simultaneously achievable goals: one need not be sacrificed to obtain the other.

3.3.1 Intelligent Networks and Service Independent Building Blocks

A key objective of the IN is to provide service-independent functions that can be used as building blocks to construct a variety of services. This allows easy specification and design of new services.

A second key objective is network implementation independent provision of services. This objective aims to isolate the services from the way the service-independent functions are actually implemented in various physical networks, thus providing services that are independent of underlying physical network infrastructure. Figure 3.1 shows the IN approach.

IN services are based on additional service logic and data on top of different switched telecommunication networks. Centralized service nodes known as Service Control Points (SCPs) control the telecommunications network via a dedicated out of band signaling network, i.e. the international Signaling System No. 7 (SS7) network. The bearer switching nodes, known as Service Switching Points (SSPs), provide only the basic call processing capabilities. IN service deployment and management is realized through a Service Management System (SMS), which interacts with IN elements via a data communication network. Since the SSPs and the SCP have to interact for each IN service call (usually multiple times), the signaling network and the central SCP may become serious bottlenecks. Furthermore, SCP failures would result in global service unavailability.

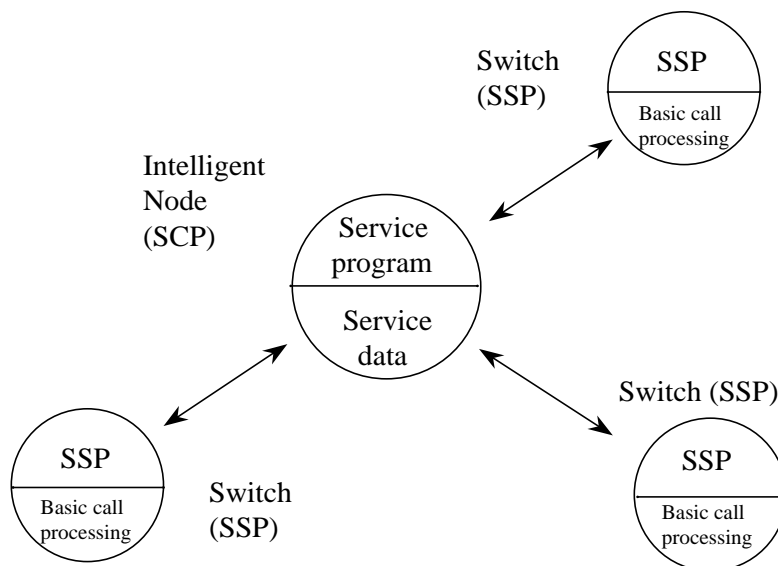


Figure 3.1 IN Approach

IN Conceptual Model (INCM) is a general framework for developing international standards for IN. It is structured into four planes - Service Plane (SP), Global

Functional Plane (GFP), Distributed Functional Plane (DFP) and Physical Plane (PP). The first two planes focus on service creation and implementation, whereas the last two planes addressing the physical IN architecture. As the IN standards evolving, each phase of development intended to define a particular set of IN capabilities, known as a Capability Set (CS). CS-1 represents the first actual standardized stage of the IN. It supports the first set of IN services.

Defined by CS-1 is a high-level logical programming interface. This programming interface consists of a set of Service Independent Building blocks (SIBs) in the GFP. This is used by the service designer for the definition of service logic programs (software programs that contain the service logic that runs in an SCP). Hence, service features in the service plane are defined by one or more SIBs in the GFP. Some of the SIBs defined by the CS-1 standards are:

- Algorithm: applies a mathematical algorithm to data in order to produce a result.
- Verify: provides confirmation that the information received is syntactically consistent with the expected form.
- Basic Call Process: a dedicated SIB responsible for providing basic call connectivity between parties in the network.

In order to build intelligent network service logic, the SIBs must be chained together. IN service logic built using a SIB chain, is referred to as global service logic. At specific points in the call processing, the SIB chain must interact with the initial basic call in order to correctly handle service requests.

The IN platform provides greater flexibility for service creation in general and also for tailoring services to suit the exact requirements of a particular customer. IN-based services rely on SIBs which are the smallest units in service creation. SIBs are standard reusable units and can be chained together in various combinations to realize services. They are defined to be independent of the specific service and technology for which or on which they will be realized.

Chapter 4 Enabling Technologies for A New Advanced Service Architecture

In the following sections, some of the enabling technologies such as Java, Mobile Agent (MA) and Jini that will be used to build the proposed advanced service architecture are briefly discussed. The Java language, developed by Sun Microsystems, has a number of advantages that make it particularly appropriate for MA technology. Its major appeals for agents are its portability, the use of bytecodes and its interpreted execution environment. This means that any system with sufficient resources can host Java programs.

The Java Virtual Machine and Java's class loading model, coupled with several of Java features – most importantly serialization, remote method invocation, multithreading and reflection – have made building first-class mobile agent systems a fairly simple task [28].

MA as one of the enabling technologies is introduced in section 4.1. Jini and JavaBeans (both from Sun Microsystems) are two good complementary candidate technologies for the implementation of an IP Telephony service architecture, they are described in section 4.2 and 4.3.

4.1 Mobile Agent Technology

Software agents originated from Distributed Artificial Intelligence (DAI) [29] research. The concept of an agent can be traced back to the early days of 1970s. There is a broad range of companies and universities that are actively pursuing agent technology and the number is growing.

An agent can be described as a software component that performs a specific task autonomously on behalf of a person or an organization [30]. It contains some level of intelligence, ranging from predefined rules to self-learning Artificial Intelligence (AI) mechanisms. Thus agents operate rather asynchronously to the user and need to communicate with the user, system resources and other agents as required to perform their tasks. They are often event - or time – driven.

A mobile agent is one of the seven agent types identified in [29]. An agent is an object with its private thread of execution, also known as an “active object” [31]. A MA is the kind of agent that is not bound to the host where it begins execution. It has the unique ability to transport itself from one host in a network to another. As it travels, it performs work on behalf of a network user. Agent mobility is probably the most challenging property, which provides an intelligent agent with the potential to influence the traditional way of communications and service realization. Customizability is the result of the diffusion of network services and applications. It allows users to tailor services according to their specific needs and preferences. Flexibility and extensibility are due to the dynamic nature of the underlying network infrastructure and service demand [32].

In the past, the main motivations for the application of mobile agents [33] were the lack of capacity to execute programs locally, and the desire to share resources and improve load balancing in a distributed system. In contrast to these concepts designed for rather specific or closed environments, new agent concepts aim for open environments (e.g. the Internet). Today, flexibility is a key design issue for emerging network service architectures to adapt quickly to the changing customer service demands. The following are some of the reasons for using MA technologies for the new service architecture:

- MA-based approach may reduce the network load when compared to an RPC (Remote Procedure Call) – based approach.
- Asynchronous and autonomous execution provides the possibility for realization of advanced services by means of using mobile agents.
- Being independent of the underlying network infrastructure makes the service architecture extendable.
- MAs allow new services to be provided dynamically by either customization or (re) configuration of existing services.
- MAs provide an effective way for deployment and utilization of advanced services within a distributed environment.

The mobile agent paradigm and emerging agent technologies are considered key for implementing open, flexible and scalable services. There are many commercial and nearly commercial agent platforms, such as Grasshopper (IKV++), Aglets Workshop (IBM), Voyager (ObjectSpace), Concordia (Mitsubishi) and Odyssey

(General Magic). With so many different platforms, interoperability is becoming an important issue. Sound standards are needed. The Mobile Agent System Interoperability Facility (MASIF) specification by OMG represents the first effort for standardizing agent platforms. It is a specification of an agent framework to support agent mobility based upon the use of the Common Object Request Broker Architecture (CORBA). Although the MASIF standard stresses language independence, it is interesting to note that most notable mobile agent frameworks are implemented in Java.

In April 1997, CLIMATE – The Cluster for Intelligent Mobile Agents for Telecommunication Environments, a pool of projects within the European Union collaborative research and development program on Advanced Communications Technologies and Services (ACTS), was launched to explore the usage of agent technologies. Most of these projects are located within Service Engineering, Security and the Communications Management domains. CLIMATE is taking an active part in contributing to relevant agent standards (e.g., OMG, FIPA) and telecommunication standards (e.g., IN, TMN, UMTS standardization). The Grasshopper MA framework has been developed under the CLIMATE umbrella. With more efforts put on to standardize agent platforms, agent platforms are maturing gradually. So is Java as an enabling tool for implementing MA. MAs have brought tremendous opportunities for development of MA-based service architecture for IP telephony.

4.2 Jini

Jini technology [34] takes advantage of the Java language. It brings network based services, seamless expansion, reliable smart devices and easy administration to the network facilities for distributed computing. Jini provides lookup services and a network bulletin board (or blackboard) for all services on the network. It allows the search for services connected by the communication infrastructure and stores not only pointers to the service on the network, but also the code and /or code pointers for these services.

Figure 4.1 shows the Jini architecture. The components of the Jini system can be segmented into three categories: infrastructure, programming model and services. The infrastructure is a set of components that enables building a federated Jini system, while the services are the entities within the federation. The programming model comprises interfaces that enable the construction of reliable services.

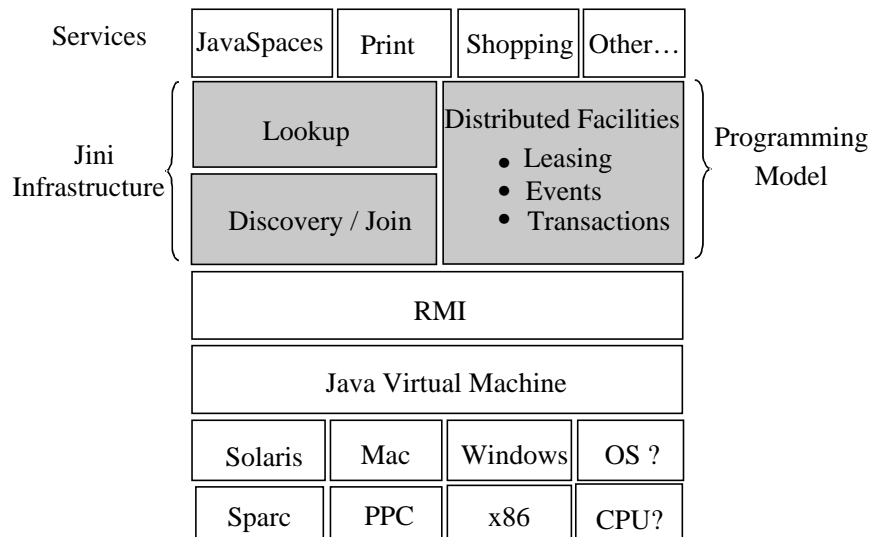


Figure 4.1 Jini Architecture

The runtime infrastructure of Jini technology resides in two places: Lookup services that sit on the network, and the Jini software-enabled devices themselves. Lookup services are the central organizing mechanisms for Jini technology-based systems. When devices are plugged into the network, they register themselves with a lookup service and become part of the federation. When clients wish to locate a service to assist with some task, they consult the lookup service.

Lookup services organize the services they contain into groups. A group is simply a set of registered services identified by a string. For example, the “Advanced IP Telephony Services” group could be populated by the IP telephony services offered by all enterprise service providers on the local network. The “East Telecom” group could be populated by the services offered by all the devices in East Telecom (including, potentially, one or more members of the “Advanced IP Telephony Services” group). As shown by this example, in which IP telephony services could belong to both “Advanced IP telephony Services” and “East Telecom” groups, a service can be a member of multiple groups. Moreover, multiple lookup services can maintain the same group. They can store the group name and its services. This kind of redundancy can help make the Jini technology based system more fault tolerant. For example, if the “Advanced IP telephony Services” group is maintained by multiple lookup services, and one of those lookup services goes off the network, clients will still be able to locate the “Advanced IP telephony Services” group via the remaining lookup services.

The runtime infrastructure enables services to register with lookup services through a process called discovery and join. Discovery is the process by which a

Jini technology-enabled device locates lookup services on the network and obtains references to them. Join is the process by which a device registers the services it offers with lookup services.

4.2.1 The Discovery Process

The discovery process works like this: Assume you have a Jini technology-enabled device capable of offering the service of “persistent storage” to a Jini federation. As soon as you connect the device to the network, it broadcasts a “presence announcement” by dropping a multicast packet onto a well-known port. Embedded in the presence announcement are two important pieces of information: the IP address and port number where this device can be contacted by a lookup service, and a list of names of groups the device is interested in joining. Assume, for example, that the drive you just plugged into the network declares in its presence announcement packet that it wants to join the “IP Telephony Services” group.

Lookup services monitor the well-known port for presence announcement packets. When a lookup service receives a presence announcement, it inspects the list of group names contained in the packet. If the lookup service maintains any of those groups, it contacts the sender of the packet directly (using IP address and port number from the packet) and sends it an RMI stub that will allow it to interact with the lookup service. Thus, in the previous example, assume a lookup service that maintains the group named “IP Telephony Services” received the device’s announcement packet. Because the announcement packet mentions this device’s interest to become a part of the “IP Telephony Services” group, the lookup service will contact the originator of the announcement packet directly at the specified IP

address and port number. The lookup service will send to this device an object that implements an interface through which the disk drive can register itself, via the join process, as a member of the “IP Telephony Services” group.

4.2.2 The Join Process

Once a device has discovered a lookup service, it can register its own services on that lookup service via the join process. The join process begins when a service connects to a lookup service via the object it received from that lookup service during the discovery process. Through the stub, the service sends information about itself to the lookup service. The lookup service stores the information uploaded from the device and associates that service with the requested group. At this point, the service has joined the group on that lookup service.

The information sent from the lookup service includes an instance of a class that implements a “service interface“. It can also include other attributes, including applets that provide graphical user interfaces through which users can directly interact with the service.

The service is identified by the type of the “service interface” uploaded to the lookup service via the join process. Each kind of service is associated with one such Java technology-based interface. The lookup service stores and locates a service based on the type of that interface. Clients interact with the service by invoking methods on an object that implements that interface. Thus, for a telephony service, for example, its service interface would be uploaded to a lookup service

during the join process and clients will interact with the IP telephony service through this interface later during the Lookup process.

4.2.3 The Lookup Process

Once a service has joined at least one group in a particular lookup service, that service is available for use by clients who query that lookup service. To build a federation of services that will work together to perform some tasks, a client must locate and enlist the help of individual services. To find a service, clients interact with lookup servers via a process called lookup.

The lookup process begins when a client contacts a lookup service and requests services of a particular type. The type specified in this request is a Java technology-based interface that defines the way in which client interact with the service being requested. This is the “service interface” that uploaded from the service to the lookup service during the join process.

The lookup service returns to the client zero to many objects that match the type (that implement the service interface) specified in the client’s request. Once a client has an object, it can interact with the service represented by that object. A client interacts with a service by invoking methods on the downloaded object that implements the service interface.

4.2.4 Client / Server Interaction

The client can interact with a service by invoking methods declared in the service interface on the object that represents the service. In addition, a client can use reflection to look for other interesting methods declared by that object. If the client

finds methods that it understands how to use, it can interact with the service by invoking those methods as well, even through those methods aren't part of the service interface.

The object that represents the service can grant the client access to the service in several ways. For example, the object may actually represent the complete service, which can be downloaded to the client during lookup process and then executed locally. Alternatively, the object can merely serve as a proxy to a remote service. When the client invokes methods on the proxy object, the proxy sends requests across the network to the service, which does the real work. An in-between approach is also possible. In this case, the local object and a remote service each do part of the work. Proxies that fully or partially implement the service themselves are called smart proxies.

Note that the protocol used to communicate between a proxy object and the remote service does not need to be understood by the client. This service protocol is a private matter decided upon by the service itself. The client can communicate with the service via this private protocol because the service has in effect injected some of its own code (the object that represents the service) into the client's address space. The injected object could be an RMI stub that enables the client to invoke remote methods on an object that exists in the address space of the remote service. Or the injected object could communicate with the service via CORBA, DCOM, or some proprietary protocol.

Different implementations of the same service interface can use completely different approaches and completely different protocols. A service may use specialized hardware to fulfill client requests, or it may do all its work in software. To the client, a service just looks like a service, regardless of how it is implemented.

4.2.5 The Jini Technology Programming Model

The Jini technology programming model offers a small set of APIs that can help users create reliable distributed systems. Most of the interaction between clients and services during the processes of discover, join, and lookup is built around these APIs, so clients and services will use the Jini technology programming model during those processes. Clients and services can also make use of the programming model to do the work for which the federation was assembled in the first place.

The Jini technology programming model consists of three parts: leasing, transactions, and distributed events. Leasing provides a way to manage the lifetimes of distributed objects that can not be managed by the usual rules of garbage collection. In a single address space, the garbage collector can grab an object when there are no references to it. But a garbage collector does not know if there are any remote references to an object. A lease is a grant of guaranteed access to a remote resource, such as an object, for a specified period of time. It is a guarantee that during the period of the lease, the resource won't be garbage collected away.

For example, if a client wishes to make use of an object of a particular service, the client can make a lease request to the service that includes a desired lease period. The service can, at its discretion, award the lease to the client. The service has to

decide the duration of the lease, presumably taking the requested time period into account, and communicate that duration back to the client. If the client does not renew the lease before the time period decided upon by the service elapses, the service can assume the object is no longer needed by the client and can discard the object. But as long as the client keeps renewing the lease before it expires (and the service continues to allow the renewal), the service will not garbage collect the object and the object will remain available to the client.

Another aspect of the Jini technology programming model that can help users build reliable distributed systems is transaction. The API that supports transactions enables operations that involve multiple clients and services to either succeed or fail as a unit. If some aspect of the operation managed by a transaction fails, for example, one of the involved services disappears from the network, the participating parties can be instructed to “roll back” to a known good state.

The third aspect of the programming model that facilitates the building of reliable distributed systems is the distributed event model. This model extends the 1.1 JavaBeans/AWT/Swing event model, which works in a single address space, to distributed systems. Using the Jini technology event model, an object can register itself as a listener interested in events generated by a remote source. When the remote source fires an event, the event will travel across the network to the registered listeners.

4.3 JavaBeans

The JavaBeans component architecture is the platform-neutral architecture for the Java application environment. It's the ideal network-aware solution for heterogeneous hardware and operating system environments -- within the enterprise or across the Internet.

The JavaBeans component architecture extends the "Write Once, Run Anywhere" capability to reusable component development. It enables developers to create reusable software components that can then be assembled together using visual application builder tools. Ideally, any Java component conforming to the JavaBeans component model can be reused in any other JavaBeans compliant application. Every Bean not only complies with the JavaBeans model, it also carries with it all its properties and methods, which can be easily garnered through introspection – a JavaBeans property whereby any visual builder tool can analyze and report on how a Bean operates.

The JavaBeans API makes it possible to write component software using the Java programming language. Components are self-contained, reusable software units that can be visually composed. Considering using JavaBeans to implement the supplementary services defined in H.323/H.450.x, these services can be created by assembling JavaBeans components. When a new service is requested, it will be very easy to use existing components and create new service components as less as possible. By saving service creation time, service creators/providers will be more competitive in the service market. These are possible advantages by using Java-

Beans, but due to the time limitation, JavaBeans will not be used to implement the proposed advanced service architecture.

Chapter 5 Mobile Agent Based Advanced Service Architecture

The traditional telephone system has very primitive end-terminals (telephones) and considerable intelligence inside the network [35]. Advanced service architectures separate call setup and call processing functions. On the other hand, in general, the Internet represents a different balance, with intelligent end-terminals (computers) and a simple set of functions inside the switches of the network. Switches are composed of software and general-purpose hardware. It is reasonable to foresee that long-term evolution of IP Telephony will have much more intelligence implemented in the end terminals rather than inside the network. Advanced services such as call diversion and call transfer, which are implemented inside the telephone network today, can be implemented in user's computer.

In order to realize this view of IP telephony, appropriate protocols and technologies are needed. As mentioned previously, there are two protocols previously mentioned that address this issue, one is H.323 and the other is SIP.

Supplementary services supported by H.323 are specified in H.450.x. Each of the defined supplementary services has its own specification. As pointed out in [36], "How they may be broken down into reusable building blocks is not clear and this will lead to specification and implementation inefficiency". The H.323 specifica-

tion does not address service control and management. In addition, there are no third party defined services. But obviously there is room left for developers to design more flexible service architectures using enabling technologies, e.g., MA.

IP telephony is real-time data communications over IP transport network. As the Internet is an open, distributed and evolving entity, flexibility, scalability and robustness are very important issues to be considered when designing new service architectures. MA technology has the ability to provide solutions addressing all these issues. As pointed out in [37], the key advantage of MA is its flexibility. It can enhance service architectures, provide easy service customization and instant service provisioning.

5.1 A New Advanced Service Architecture Based on Mobile Agent

Because of the limitations of the existing service architecture defined by H.323, we propose a MA based advanced service architecture for implementing H.323 supplementary services using the widely accepted service provisioning basis (IN), enabling technologies (MA, Jini/JavaBeans) and the requirements described in [21].

The major contributions of this architecture are unified provision of H.323 supplementary services and support of dynamic deployment of services. For these purposes, MA platforms are introduced into the devices that are connected to the enterprise LAN. H.323 supplementary services are realized by means of mobile service agents. The key of this approach is to deploy service agents to the service

users, i.e., the call parties. This makes the new service architecture open, distributed and flexible.

The proposed service architecture allows open service creation. Supplementary services can be created by a different Service Component Creator (SCC) using Service Components from a Service Component Repository (SCR). Service utilization is realized by activating caller's User Service Agent (USA) and ultimately activating callee's USA, Call Agent (CA) will be instantiated as the result of USA's creating new CA. The new service architecture supports universal access to a service through the Jini Lookup process. Service customization is also supported by this service architecture. Each user connected to the network can define his own service data. Service logic is not embedded in the network nodes but ultimately in the end user's terminal. This makes the new service architecture highly distributed. Because of the USA, messages and procedures for call setup are independent of the underlying network architecture.

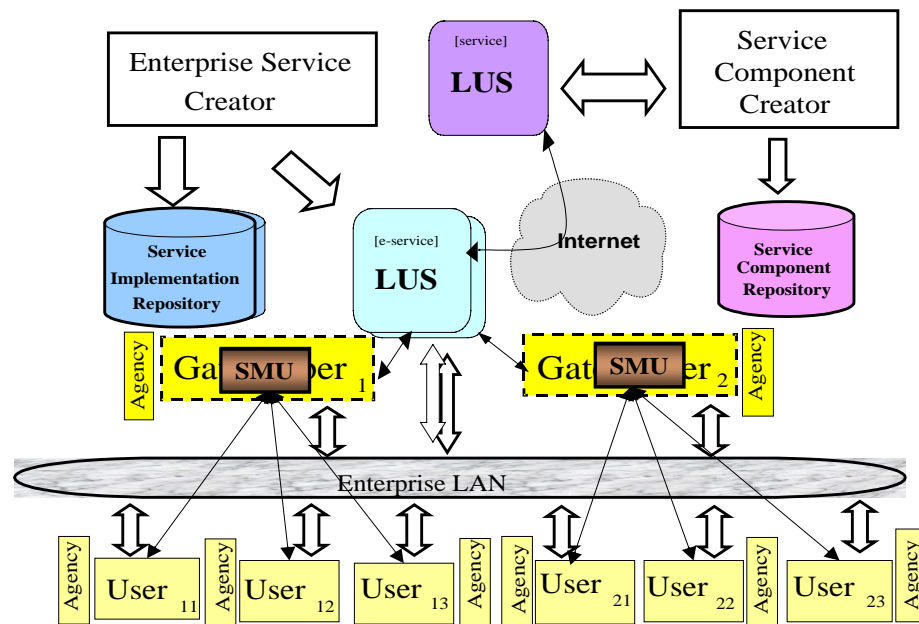


Figure 5.1 MA Based Advanced Service Architecture for IP Telephony

As illustrated in Figure 5.1, H.323 gatekeepers and H.323 terminals (Users) are connected to the Enterprise LAN. An Agency that provides an agent execution environment is attached to each gatekeeper/terminal. User terminals join one gatekeeper's zone through the gatekeeper discovery and endpoint registration process. Lookup Services (LUSs) are more like a blackboard where all the available services' proxy code (interface of a service) is placed. A Service Component Creator (SCC) is responsible for creating components which are made available to Enterprise Service Creator (ESC) and advertising its services on a LUS. All the service components are stored in the Service Component Repository (SCR). These service components can be assembled into new services by SCC. End users can not subscribe services from SCC directly. The SCC and SCR bring opportunities for third party service creators and providers, enable them to compete in the service market.

LUSs can be local or remote, they are linked by the Internet. Service Management Unit (SMU) is a device that can be placed in the gatekeeper, or completely separated from the gatekeeper. It manages service subscription using protocols not defined by H.323, e.g., HTTP. If a service were to be dynamically upgraded, the SMU would be involved. Additionally, the SMU could be involved in ongoing network management of the services. An SMU can discover an enterprise LUS using a multicast protocol. A unicast protocol is used to discover remote LUSs that are outside of the enterprise LAN. In the latter case, the SMU has to know where the LUS is before it sends out a request to the remote LUSs. An Enterprise Service Creator is responsible for customizing and assembling the service components into

services that are provided to the end user, using the available code from Service Implementation Repository (SIR).

In the following discussions, we will present one scenario for service subscription in which only one gatekeeper is involved and one scenario for service utilization where one gatekeeper is involved. At the same time, all the related components' functionality will be explained.

5.2 Service Subscription and Utilization Using Mobile Agents

As mentioned before, there are four phases in the service life cycle, they are service creation, service subscription, service utilization and service withdrawal. Service subscription and utilization of the proposed MA based advanced service architecture are described in this section. The example of VPN service will be used to illustrate the key points.

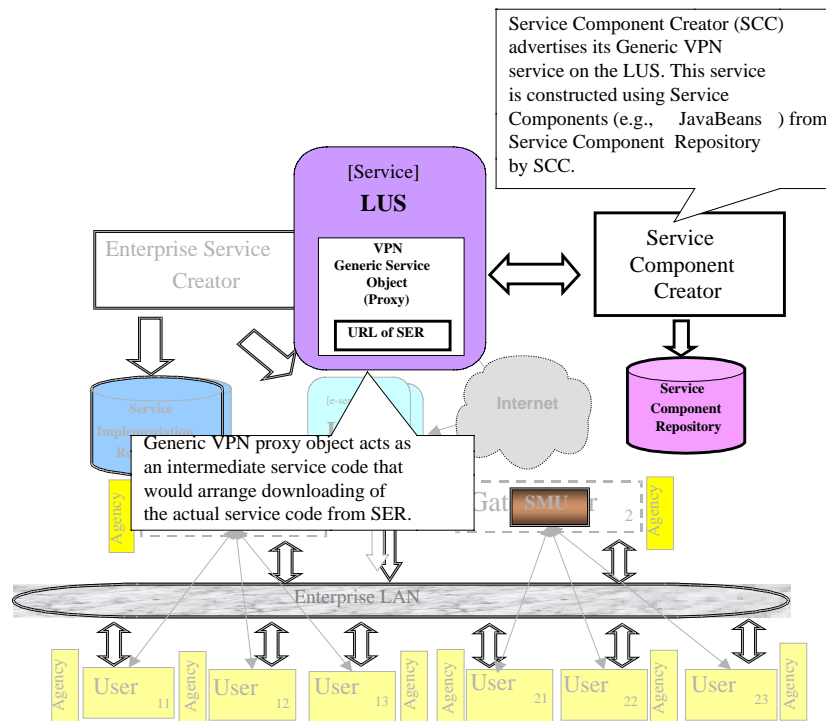


Figure 5.2 SCC Uploads VPN Service Proxy onto the Lookup Service

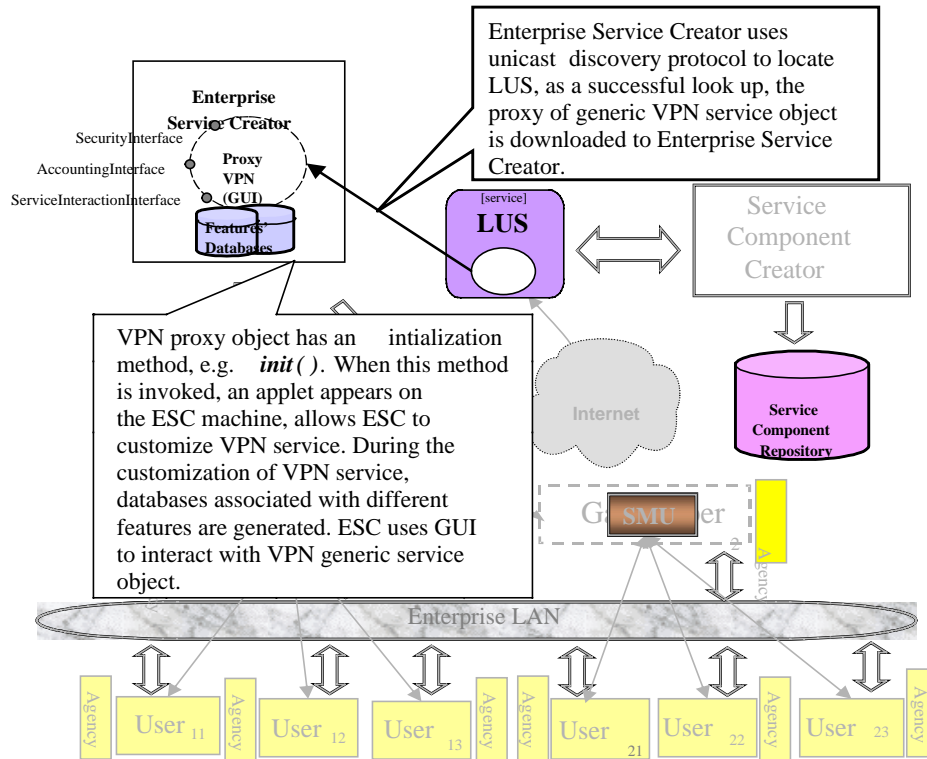


Figure 5.3 ESC Downloads VPN Service Proxy Object

The subscription of VPN service in the new service architecture consists of three steps, they are explained in Figure 5.2, Figure 5.3 and Figure 5.4 respectively.

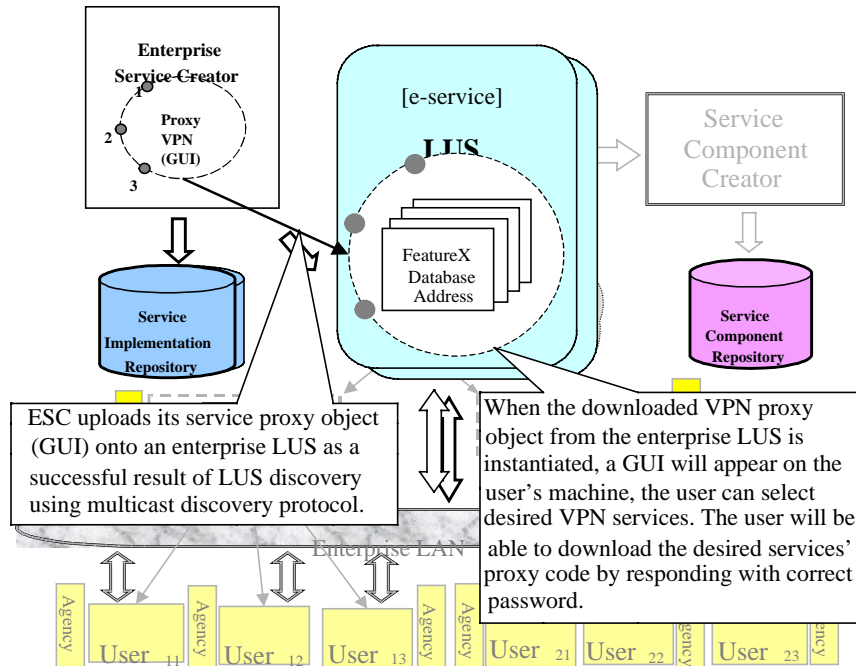


Figure 5.4 ESC Uploads an Enterprise Service Proxy onto an Enterprise Lookup Service

In the first step, as shown in Figure 5.2, the Service Component Creator advertises a generic VPN service proxy object on the LUS. The service is composed of service components which could be a JavaBean or combined JavaBeans. These Beans can be customized later to meet any specific needs.

The second step is that the Enterprise Service Creator discovers the LUS. Using the unicast discovery protocol, the ESC downloads the VPN service proxy object to its machine, and then it uses the Graphical User Interfaces to customize the VPN service for its enterprise users. The details of this step are outlined in Figure 5.3.

Figure 5.4 shows the third step, in which the Enterprise Service Creator uploads its customized service proxy object to the enterprise LUS, and then this service proxy will be downloaded to the Service Management Unit. There is one thing need to be clarified, that is, there could be many such kind LUSs, and the Enterprise Service Creator should find (discover) one lookup service that matches it's searching criteria.

After these three steps, the user has finished the subscription of a customized VPN service. In the following discussion, how the service is invoked and utilized will be elaborated.

As mentioned before, service utilization is realized by activating caller's and callee's User Service Agent (USA). As illustrated in Figure 5.5, a USA consists of a ServiceClass, one or more Code Repository URLs and Service Logic and Data. It defines how a call will be processed, for example, the management of feature ordering. A USA can also handle service management.

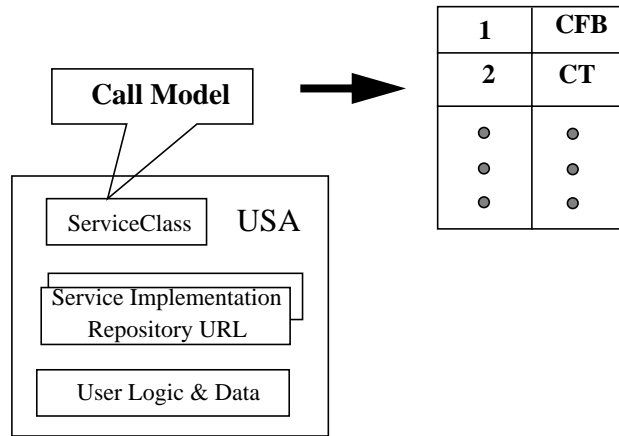


Figure 5.5 User Service Agent

In Figure 5.5, the ServiceClass (call Model) is specific to an end user. The Call Model component of the USA is an IN call model consisting of two separate sets of call processing logic: originating and terminating call processing logic. The originating call processing logic provides support to the Calling Party, and is modeled by the IN Originating Basic Call Model (O-BCSM). The Terminating call processing logic provides support to the Called Party, and is modeled by the IN Terminating Basic Call Model (T-SCSM).

The overall call model provides support for a finite state machine with points of interaction with advanced services. In the traditional IN view of advanced services, these points of interaction would be implemented as trigger points. In the service architecture proposed here, using component-based technology, Java Beans could be used with well-known interfaces and the interaction mode would be via method calls.

The User Logic in Figure 5.5 represents processing that is required for user subscribed services. For a specific service, in one call processing state, the user logic

specifies how to deal with this service and what the next step is in the call processing. For sophisticated services the call model may even make it possible for the user to write scripts (rules, perhaps) that add a degree of intelligence to the service. For instance, when a user wants to subscribe the VPN service described before, he might want to write scripts that filter out particular callers, or callees from a specific set of network addresses.

The service implementation repository URL gives the reference where the service code can be found. The User Data in the previous figure is the service-related data. For example, after a user chooses the Call Forwarding Unconditional (CFU) service, he will also be asked to provide the phone numbers to which he would like the calls to be forwarded. To reiterate, the User Service Agent consists of a Call Model, one or more Service Implementation Repository URLs and Service Logic and Service Data.

A USA is constructed when an end user sends a request for service subscription. The call model will be unique to the user according to the subscribed services at subscription time. For example, user B may subscribe to Call Forwarding Unconditional (CFU) and Call Transfer (CT). The ServiceClass component of USA for user B will be a call model that has different trigger points during a call. For CT, the service trigger point would be in the originating call model. For CFU, the service trigger point would be in the terminating call model. Once constructed, the USA moves to the user local agency from the gatekeeper. When the user has subscribed several services, the construction of a USA could be a tricky endeavor due to potential interactions between the services.

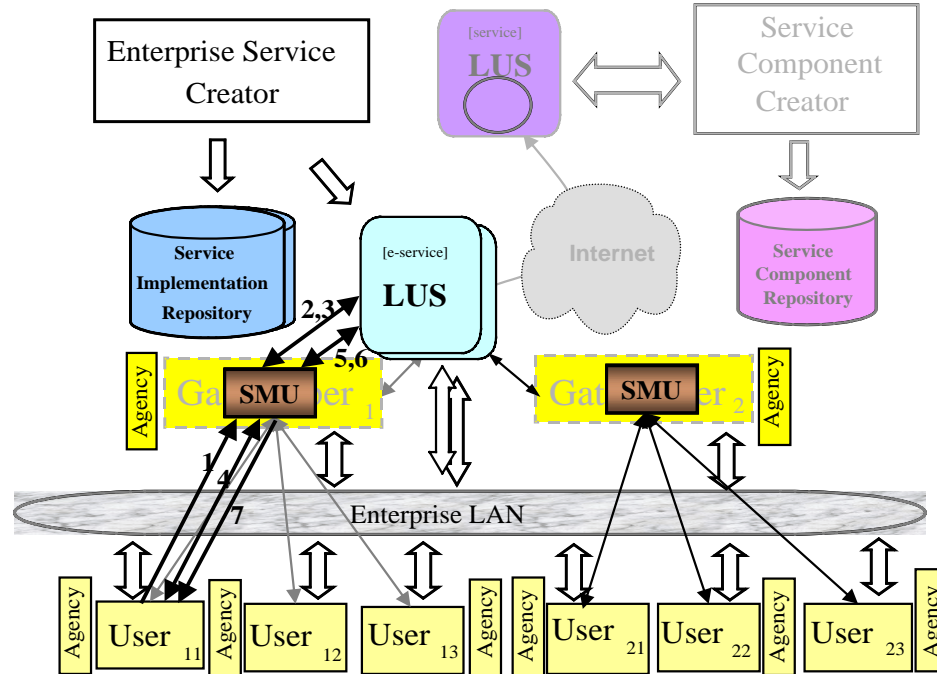


Figure 5.6 Services Subscription and Activation

The steps for service subscription and activation are summarized in Figure 5.6.

Following are the explanations of each step:

1. Any User can send request to its SMU to subscribe to advanced services.
2. After the SMU receives a service subscription request from the end user, it multicasts the requests in the network to discover LUSs which have supplementary services.
3. Following the discovery, the SMU gets a response from the LUS listing all the supplementary services it has on the network and the addresses/URLs of other LUSs outside of the enterprise network which have the same kind of services available. Using this list, the SMU constructs a service subscription graphical user interface and sends it to the end user stating that these are the supplementary services available.

4. The end user selects the needed services that he wants, fills in the form and sends it back to the SMU. The form includes a facility for the user to specify service related data.
5. The user may not find all the services he wants. In this case, the user should send a message to the SMU indicating that he wants particular services that are not in the form, SMU then query other available LUSs external to the enterprise LAN using the addresses/URLs from the last query.
6. After the SMU receives the completed form from the user, it checks its User Profile which contains all the services that are already in use by each user. It then sends a request to the lookup service to download the proxy code (interfaces) of the services that this user has subscribed to.
7. When the SMU is checking the user profile, the following things may occur:
 - The services that the user wants to subscribe are already there, then the SMU sends a notification to the user indicating that the requested services are already provisioned.
 - Some of the services the user has requested are already available to the user, then the SMU sends a notification to the user, and sends the proper request to the LUS.
 - If all the services are new to the user, the SMU sends a request to the lookup service to download the proxy code of the services that the user has requested as described in step 6.

The service proxy code will be downloaded to the SMU if the service has not been

downloaded already. The proxy code contains the interfaces that a gatekeeper needs to construct a USA, and the location of a SIR where to find the actual service code (step 6 in Figure 5.6).

There may be many URLs for the addresses of multiple SIRs. A customized USA will be sent to the user agency, and stays in the user's terminal. Once the user receives the USA, it acknowledges the SMU.

After the successful subscription of a service, the service can be invoked from the call model. The sequence of how the call is set up using USA and CA is as following:

- If the gatekeeper routed call signaling mode is used, the USA moves to the gatekeeper agency after construction.
- When user A who has not subscribed to any advanced services starts a new call, the Basic Call Processing (BCP) functions in the user terminal will be invoked. If A has subscribed to any advanced services, a USA associated with A has been stored in the gatekeeper/SMU. Before A makes a call, it has to send a RAS message to the gatekeeper, and the gatekeeper will activate A's USA. A call agent will be instantiated for A, which will perform call's originating part for A. As a result of the function call, a setup message is sent to the called party (user B) via the gatekeeper.
- Call signaling message callProceeding is sent back.
- SMU checks its user profile, user B has subscribed to CFU and CT, th-

en there must be a USA for user B. So user B's USA is activated and a Call Agent (CA) which implements B's call model is instantiated. For instance, the USA can instantiate a CA by doing *new CallAgent ()*. At this time, the USA and the CA reside in the same agency. The CA gets a handle of the USA so that these two can communicate to each other by method calls. The CA also makes use of the BCP functions available in the gatekeeper and can obtain the code it needs (i.e. code related to advanced services) through a URL provided by the USA. Then the CA will take over the call processing, and acts on behalf of user B who has subscribed the services.

- The call control messages are sent between user A and the CA on behalf of user B. So the CA will know whenever it gets an incoming call, it will then forward the call to the phone number (IP Address) that B has specified in the service subscription of CFU.

5.3 Usage Scenarios for the New Service Architecture

A MA based advanced service architecture for H.323 IP telephony has been presented and its usage has been discussed in the previous two sections. In this section, four usage scenarios of the service architecture will be presented. One scenario is the VPN service subscription which has been discussed briefly before. The other three examples are used to illustrate USA and CA's movement to deploy supplementary services using H.323 messaging (with one gatekeeper involved and gatekeeper routed call signaling). They are Call Forwarding in section 5.3.1, Call

Transfer in section 5.3.2, and VPN in section 5.3.3. CF and CT are defined by ITU-T standard H.450.x. VPN is not defined in the standards.

5.3.1 VPN Service Subscription

The subscription sequence of VPN service in the new advanced service architecture is outlined in Figure 5.7.

ASSUMPTION: The messages used in Figure 5.7 are not defined by H.323, they need to be defined in the future.

DESCRIPTION:

1. The Service Creator Component (SCC) discovers the LUS and advertises the generic VPN service proxy object.

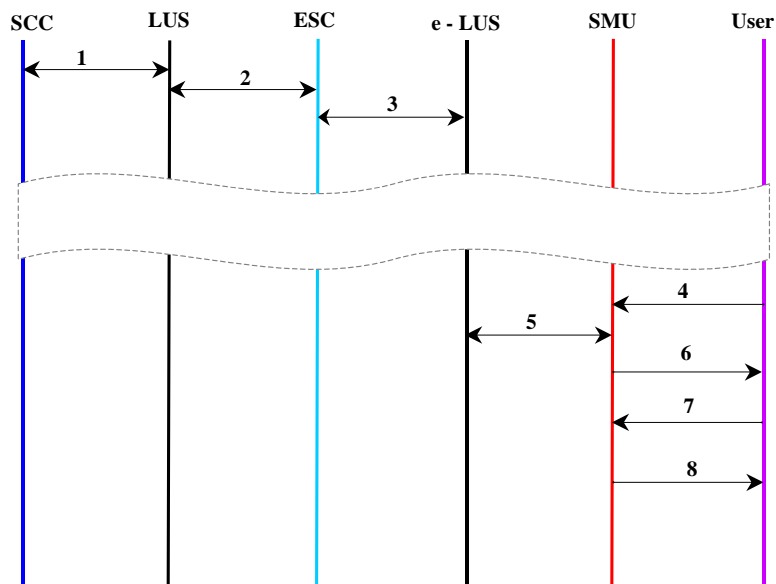


Figure 5.7 VPN Service Subscription

2. The Enterprise Service Creator (ESC) discovers the generic VPN and downloads the VPN service proxy object.

3. After the ESC customizes the generic VPN service for its enterprise users, it discovers and uploads the customized VPN service to the enterprise LUS (e-LUS).
4. The end user makes a request for VPN service.
5. The SMU discovers the e-LUS that has VPN service and downloads the service proxy object.
6. The SMU uses the downloaded proxy object to construct a USA, and sends it to the user agency.
7. When the user receives the USA, it sends an acknowledgment to the SMU indicating that the USA is received.
8. The SMU sends a message to the end user indicating that the service subscription process is completed.

5.3.2 Call Forwarding Unconditional Invocation

5.3.2.1 CFU Invocation Using End-to-end Call Signaling

Figure 5.8 shows the invocation process of CFU service using the new advanced service architecture.

ASSUMPTION: The client application will be responsible for sending messages specified by the Call Agent to other applications that are dealing with the call setup process. The CA can accomplish this by calling the client application's interfaces.

DESCRIPTION:

1. The originator (Caller) application sends a SETUP message to the called party

Originator CallAgent Called Diverted-to

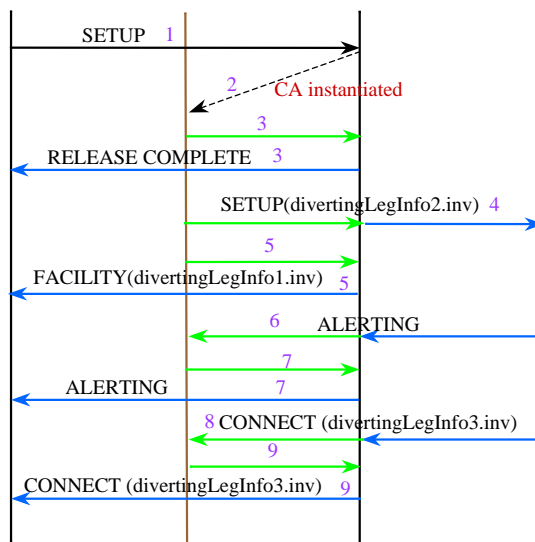


Figure 5.8 CFU (End-to-end call Signaling) Invocation

(Callee). A flag will be set in the SETUP message’s NonStandardControl field to activate the USA, so that a Call Agent is instantiated.

2. After the call agent which resides in the callee’s user agency is instantiated, it takes over the call processing from the client application and sends out all the call setup related call signaling messages.
3. The CA sends a RELEASE COMPLETE message to the caller.
4. The CA sends a SETUP message with divertingLegInfo2.inv to the diverted-to party.
5. The CA sends a FACILITY message with divertingLegInfo1.inv to the calling party.
6. The diverted-to party sends an ALERTING message to the Call Agent.
7. The CA sends an ALERTING message to the calling party.

8. The diverted-to party sends a CONNECT message to the Call Agent.
9. CA sends a CONNECT message to the calling party.

5.3.2.2 CFU Invocation Using Gatekeeper Routed Call Signaling

Figure 5.9 shows the steps of invoking CFU using gatekeeper routed call signaling within the new advanced service architecture.

ASSUMPTION: Same as CFU using end-to-end call signaling.

DESCRIPTION:

Following are the detailed description of call set up steps as shown in Figure 5.9:

1. The originator sends a SETUP message to its gatekeeper.
2. The gatekeeper responds with a CALL PROCEEDING message.
3. The gatekeeper sends a SETUP message with divertingLegInfo4.inv to the called party.
4. The calling party sends a RELEASE COMPLETE message with USA in the NonStandardData field.
5. Once the gatekeeper receives the USA, a Call Agent is instantiated.
6. The CA sends a SETUP message with divertingLegInfo2.inv to the diverted-to party.
7. The CA sends a FACILITY message with divertingLegInfo1.inv to the calling party.
8. The diverted-to party sends an ARQ to the gatekeeper indicating that it will accept the call.

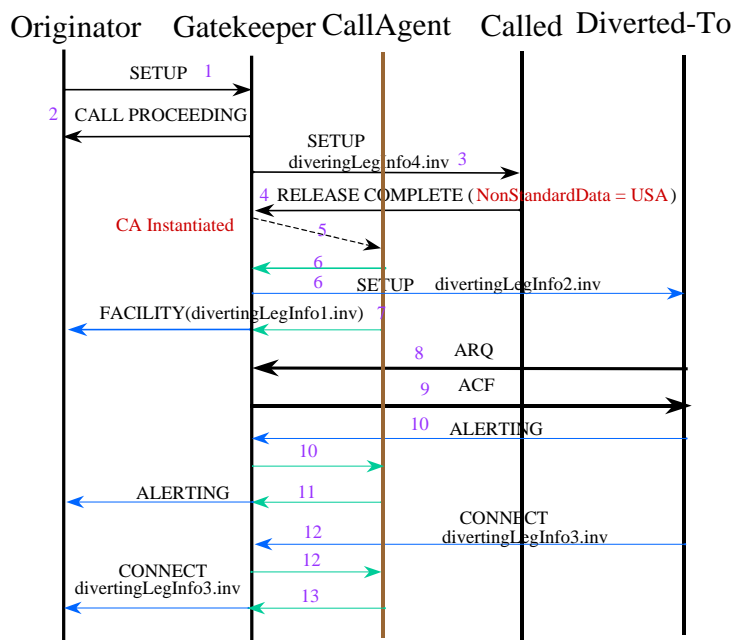


Figure 5.9 CFU (Gatekeeper Routed Call Signaling) Invocation

9. The gatekeeper sends an ACF message to the diverted-to party with the gatekeeper’s call signaling transport address.
10. The diverted-to party sends an ALERTING message to the call agent via the gatekeeper.
11. The CA sends an ALERTING message to the originator (calling party).
12. The diverted-to party sends a CONNECT with divertingLegInfo3.inv to the CA via the gatekeeper.
13. The CA sends a CONNECT message with divertingLegInfo3.inv to the originator (calling party).

5.3.3 Call Transfer Invocation

Figure 5.10 shows the call set up steps for invoking Call Transfer using the new service architecture.

ASSUMPTION: A CA has been instantiated in the caller agency when the caller started making call.

Following are the detailed description of the call set up steps as shown in Figure 5.10:

DESCRIPTION:

1. The transferring endpoint (A) sends a FACILITY message with CTInitiate.inv to the CA.
2. The CA sends a SETUP message with CTSetup.inv, opt.CTUpdate.inv to the transferred-to endpoint (C).
3. The CA sends a CONNECT message with CTSetup.rr, opt.CTUpdate.inv to the transferred-to endpoint (C).
4. The CA sends a FACILITY message with CTComplete.inv to the transferring endpoint (A).
5. The CA sends the *Terminal Capability Set* to the transferred-to endpoint.
6. The transferred-to endpoint sends TCS = 0 to the CA.
7. The CA sends TCS = 0 to the transferring endpoint.
8. The CA sends TCS = 0 to the transferred endpoint.
9. Close Channels between the transferring point and the transferred endpoint.
10. 10'. The CA sends a *Terminal Capability Set* (H.245 message) to the transferred and the transferred-to endpoints.

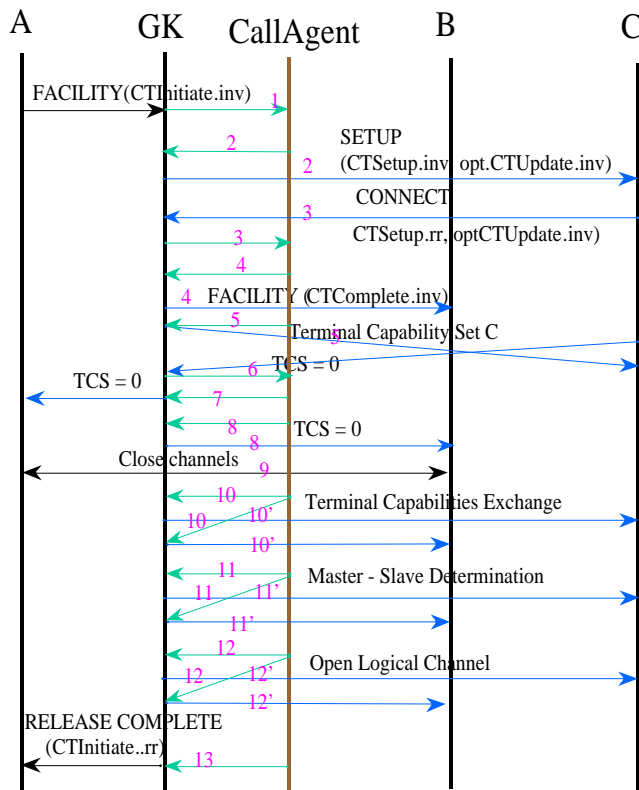


Figure 5.10 CT (Gatekeeper Routed Call Signaling) Invocation

11. 11'. The CA sends a *Master/Slave Determination* (H.245 message) to the transferred and the transferred-to endpoints.

12. 12'. The CA sends an *Open Logic Channel* (H.245 message) to the transferred and the transferred-to endpoints.

13. The CA sends a *RELEASE COMPLETE* message to the transferring endpoint.

5.3.4 Outgoing Call Allowance/Outgoing Call Restriction Invocation

Figure 5.11 shows the invocation of Outgoing Call Allowance (OCA)/Outgoing Call Restriction (CGR) of VPN service.

ASSUMPTION: Since VPN service is not defined by H.323 when this service architecture was designed, the H.225.0 call signaling will be used in order to illus-

trate the call management sequence with a call agent. This will also make it easier to understand by using the same style of call sequence diagram. The messages used here need to be identified in the future.

Following is detailed description of steps in Figure 5.11.

DESCRIPTION:

1. The originator and its gatekeeper starts to exchange the admission messages, the originator sends an Admission Request (ARQ) to the gatekeeper.
2. An Admission Confirmation (ACF) is sent back to the caller (originator).
3. The caller (originator) application sends a SETUP message to the called party, a flag will be set in the SETUP message's NonStandardControl field to activate the USA that is residing in the gatekeeper.
4. A CA is instantiated.
5. 5'. The CA sends a SETUP message to the called party via the gatekeeper.
6. 6'. The called party sends a CALL PROCEEDING message to the CA via the gatekeeper.
7. 7'. The CA sends a CALL PROCEEDING message to the originator via the gatekeeper.
8. The called party sends an ARQ to the gatekeeper.
9. The gatekeeper sends an ACF to the called party.

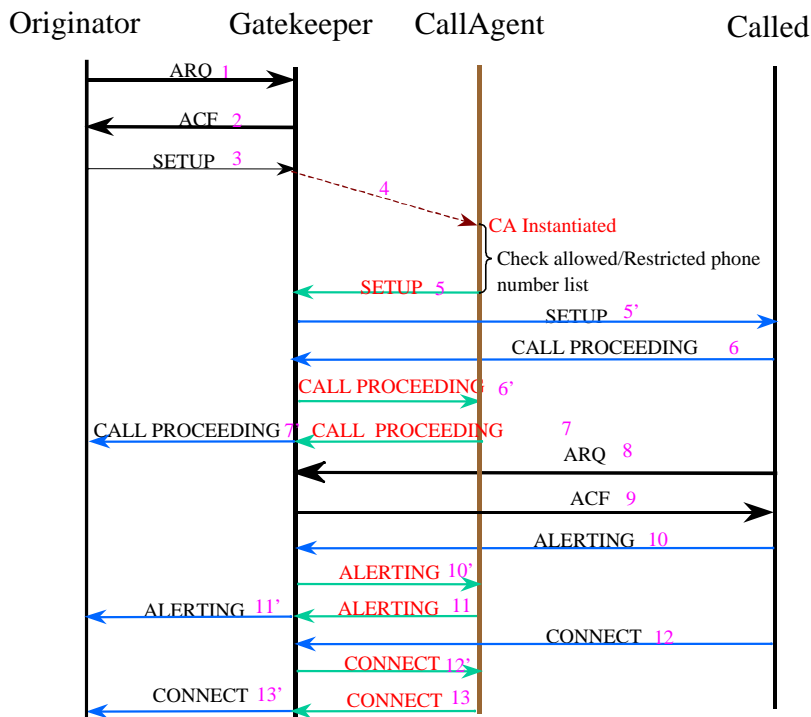


Figure 5.11 VPN Service Invocation - OGA/OGR (Gatekeeper Routed Call Signaling)

10. 10'. The called party sends an ALERTING message to the CA via the gatekeeper.

11. 11'. The CA sends an ALERTING message to the originator via the gatekeeper.

12. 12'. 13. 13'. The called party sends a CONNECT message to the CA via the gatekeeper, and the CA sends a CONNECT message to the originator via the gatekeeper.

Chapter 6 Implementation of the MA Based Advanced Service Architecture Using Grasshopper

In this chapter, implementation details of the new advanced service architecture will be presented. Grasshopper, the platform on which the implementation of MA is based, will be introduced first. Then the followed sections will discuss the mechanisms of using the Grasshopper platform to realize SMU functionalities. Then an example implementation of the H.323 supplementary services - Call Forwarding Unconditional will be described within the Grasshopper platform.

6.1 Grasshopper - Mobile Agent Programming Environment

This section introduces the mobile agent platform - Grasshopper [38]. It is used for implementing mobile agents (UserServiceAgent and CallAgent).

6.1.1 Grasshopper Platform

Grasshopper is a mobile agent platform that is built on top of a distributed processing environment. By using Grasshopper, the integration of traditional client/server paradigm and mobile agent technology can be achieved.

6.1.1.1 Distributed Agent Environment

This section describes the structure of the Grasshopper Distributed Agent Environment (DAE). The DAE is composed of regions, places, agencies and different types of agents. Figure 6.1 shows an abstract view of these entities.

Up to now, there is no standardized definition of a (software) agent. However, agents can be characterized by a set of attributes. The only attribute that is commonly accepted is autonomy. Taking this into account, an agent is a computer program that acts autonomously on behalf of a person or organization.

Figure 6.1 illustrates the hierarchical component structure of grasshopper platform. A region consists of a region registry and agencies.

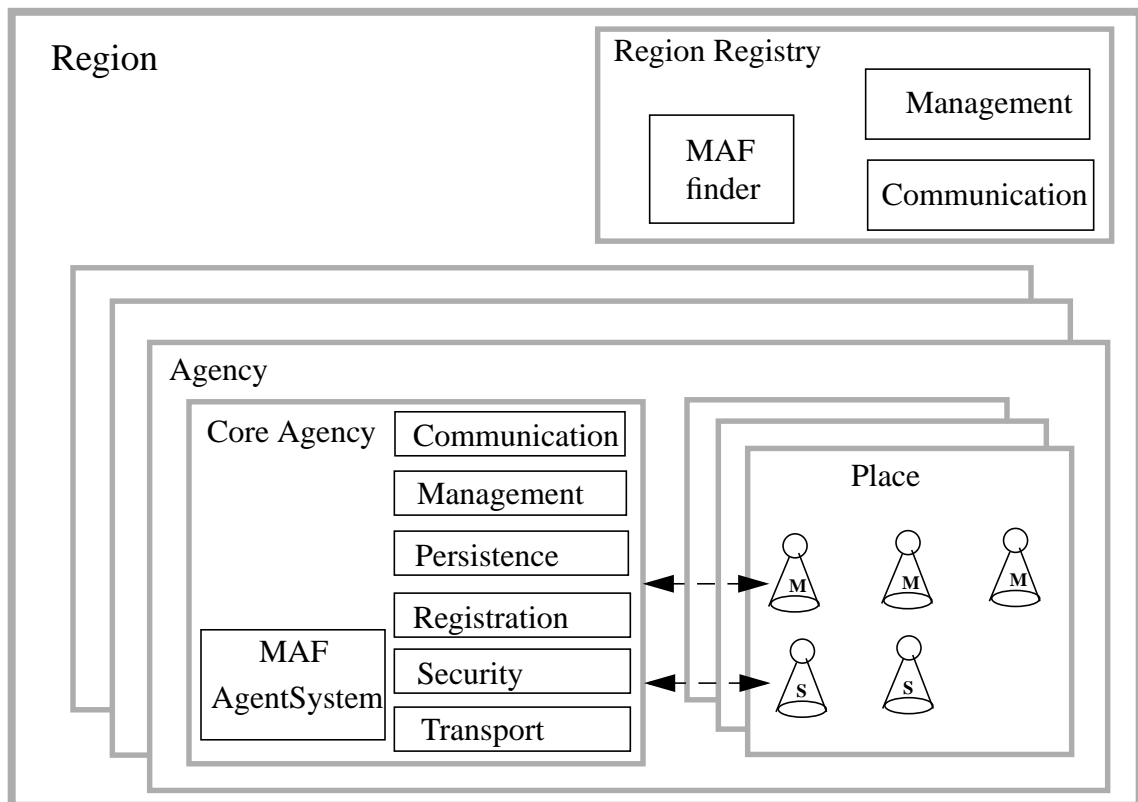


Figure 6.1 Hierarchical Component Structure of Grasshopper

Two types of agents are known in the Grasshopper context: mobile agents and stationary agents.

- **Mobile Agents:** Mobile agents are able to move from one physical network location to another. In this way, they can be regarded as an alternative or enhancement of the traditional client/server paradigm. While client/server technology relies on remote procedure calls across a network, mobile agents can migrate to the desired communication peer and take advantages of local interactions. By doing this, several goals can be achieved, such as reduction of network traffic and reduction of the dependency of network availability.
- **Stationary Agents:** In contrast to mobile agents, stationary agents do not have the ability to migrate actively between different network locations. Instead, they are associated with one specific location.

6.1.1.2 Agencies

An agency is the actual runtime environment for mobile and stationary agents. At least one agency must run on each host that shall be able to support the execution of agents. A Grasshopper agency consists of two parts: the core agency and one or more places.

6.1.1.2.1 Core Agency

Core agencies represent the minimal functionality required by an agency in order to support the execution of agents. The following services are provided by a Grasshopper core agency:

- **Communication Service:** This service is responsible for all remote interactions that take place between the distributed components of Grasshopper, such as location-transparent inter-agent communication, agent transport, and the localization of agents by means of the region registry.
- **Registration Service:** Each agency must be able to know about all currently hosted agents and places, on one hand for external management purposes and on the other hand in order to deliver information about registered entities to hosted agents.
- **Management Service:** It allows the monitoring and control of agents and places of an agency by (human) users. It is possible, among others, to create, remove, suspend and resume agents, services and places, to get information about specific agents and services, to list all agents residing in a specific place, and to list all places of an agency.
- **Security Service:** Grasshopper supports two security mechanisms, external and internal security. External security protects remote interactions between the distributed Grasshopper components, i.e., between agencies and region registries. On the other hand, internal security protects interactions between local Grasshopper components.
- **Persistence Service:** The Grasshopper persistence service enables the storage of agents and places (the internal information maintained inside these components) on a persistent medium. This way, it is possible to recover agents or places when needed, e.g., when an agency is restarted after a system crash.

6.1.1.2.2 Places

A place provides a logical grouping of functionality inside an agency. For example, there may be a communication place offering complex communication features, or there may be a trading place where agents offer or buy information or service access. The name of the place should reflect its purpose.

6.1.1.3 Regions

The region concept facilitates the management of the distributed components (agencies, places and agents) in the Grasshopper environment. Agencies as well as their places can be associated with a specific region by registering themselves within the accompanying region registry. All agents that are currently hosted by those agencies will also be automatically registered by the region registry. If an agent moves to another location, the corresponding registry information is automatically updated. A region may comprise of all agencies belonging to a specific company or organization.

The region registry maintains information about all components that are associated with a specific region. When a new component (i.e. agency, place, or agent) is created, it is automatically registered with the corresponding region registry. While agencies and their places are associated with a single region for their entire life time, mobile agents are able to move between the agencies of different regions. The current location of mobile agents is updated in the corresponding region registry after each migration. By contacting the region registry, other entities (e.g. agents or human users) are able to locate agents, places, and agencies residing in a

region. Besides, a region registry facilitates the connection establishment between agencies or agents.

The Grasshopper environment can be established even without a region registry. However, in this case agents and agencies must know all information that is required for remote interactions, such as host name, port numbers, communication protocols, etc.

6.1.1.4 Communication Concepts

The communication facilities of Grasshopper are realized by the *Communication Service (CS)* which is an essential part of each core agency. This communication service allows location-transparent interaction between agents, agencies and non-agent-based entities.

6.1.1.4.1 Multi-protocol Support

Remote interactions are generally achieved by means of a specific protocol. The CS supports communication via the *Internet Inter-ORB Protocol (IIOP)*, Java's *Remote Method Invocation (RMI)* and *plain socket connections*. To achieve a secure communication, RMI and the plain socket connection can optionally be protected using the *Secure Socket Layer (SSL)*.

Within a region, Grasshopper is able to determine dynamically the protocols supported by a desired communication peer and select the most suitable protocol for the remote interaction. Since the supported communication protocols are realized via a plug-in interface, developers can easily integrate new communication protocols by writing their own protocol plug-ins. This way Grasshopper is open for

future requirements that may come up in the changing communication world. The multi-protocol support of Grasshopper is shown in Figure 6.2.

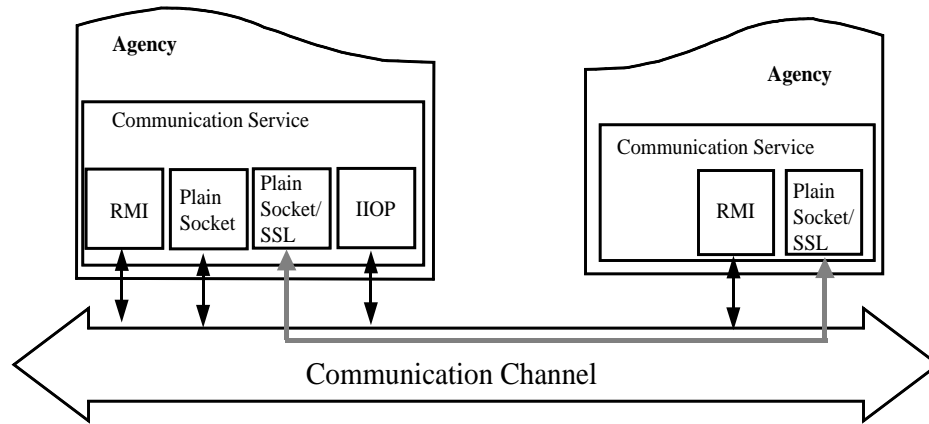


Figure 6.2 Multi-Protocol Support

6.1.1.4.2 Location Transparency

On one hand the communication service is used internally by the Grasshopper system for the agent transport, for locating entities within the DAE, etc. On the other

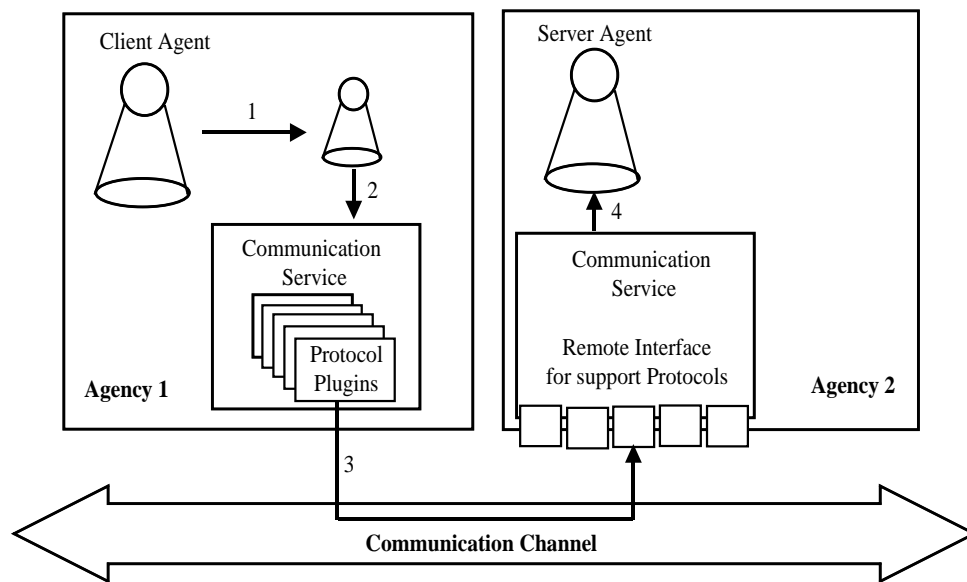


Figure 6.3 Location Transparent Communication of Grasshopper

hand, agents can use the communication service to invoke methods on other agents. Since an agent does not care about the location of the desired communica-

tion peer, the communication is totally location-transparent. Within the agent code, there is no difference between remote method invocations and local method invocations.

This is achieved through the so-called *proxy objects* that are directly accessed by a client. The proxy object forwards the call via the communication channel to the remote target object. By doing this, these proxy objects are equivalent to the client stubs used by CORBA implementation. Figure 6.3 shows this concept on an abstract level.

6.1.1.5 Agent Development

6.1.1.5.1 Accessing the Grasshopper Functionality

The functionality of Grasshopper is provided by the platform itself, i.e., by *core agencies* and *region registries*, and as well as by *agents* that are running within the agencies. By doing this, the platform's functionality is enhanced. The following possibilities regarding the access to the Grasshopper functionality must be distinguished:

- Agents can access the functionality of the local agency, i.e. the agency in which they are currently running, by invoking the methods of their super classes *Service*, *StationaryAgent* and *MobileAgent*, respectively. These super classes are provided by the platform in order to build the bridge between individual agents and agencies. Each agent has to be derived from one of the classes, *StationAgent* or *MobileAgent*.

- Agents as well as other DAE or non-DAE components, such as user applications, are able to access the functionality of *remote agencies* and *region registries*. For this purpose, each agency and region registry offers an external interface which can be accessed via the Grasshopper communication service.
- Agencies and region registries may optionally be accessed through the MASIF-compliant interfaces *MAFAgentSystem* and *MAFFinder*.

6.1.1.5.2 Agent Class Structure

In the context of Grasshopper, each agent is regarded as a service, i.e. as a software component that offers functionality to other entities within the DAE. Each agent/service can be subdivided into a common and an individual part. The common (or core) part is represented by classes that are part of the Grasshopper platform, namely classes *Service*, *MobileAgent* and *StationaryAgent*, whereas the individual part has to be implemented by the agent programmer.

A Grasshopper agent consists of one or more Java classes. One of these classes builds the actual core of the agent and is referred to as *agent class*. Among others, this class has to implement the method *live* which specifies the actual task of the agent. The agent class must be derived either from class *StationaryAgent* or class *MobileAgent* which in turn inherit from the common super class *Service*. The methods of these classes represent the essential interfaces between agents and their environment. The following two ways of method usage have to be distinguished:

- Some of the super class methods of an agent enable the access to the local core agency. For example, an agent may invoke the method *listMobileAgents()*,

which it has inherited from its super class *Service*, in order to retrieve a list of all other agents that are currently in the same agency.

- The remaining super class methods are defined to access individual agents. These methods are usually invoked by other agents or agencies via the *communication service* of Grasshopper. For instance, any agent may call the method *getState()* of another agent in order to retrieve information about that agent's actual state. Note that this way of access is not performed directly on an agent instance, but instead on an agent's *proxy* object.

Figure 6.4 shows how to access these two different kinds of agent methods.

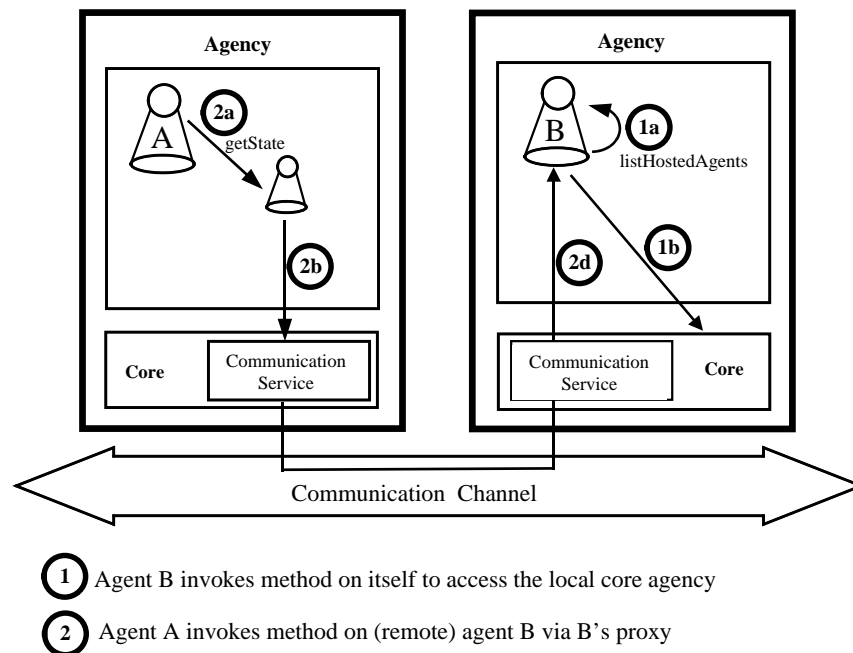


Figure 6.4 Access of an Agent's Methods

Two other ways of platform access are available for Grasshopper agents:

- The class *RegionRegistrationP* enables an agent to access the region registry in order to retrieve information about registered components, i.e. agencies,

places, services and agents. The region registry is accessed through a corresponding proxy object via the communication service.

- Apart from Grasshopper-specific platform access, the MASIF-compliant interfaces may be used, i.e. MAFAgentSystem for agencies and MAFFinder for the region registry.

6.1.1.6 Remotely Accessible Functionality

Classes *Service*, *StationaryAgent* and *MobileAgent* comprise methods that provide access to the local agency, i.e., the agency in which an agent is currently running. However, for this purpose, they provide external interfaces that can be accessed via the Grasshopper communication service.

6.1.1.6.1 The AgentSystem Interface

In order to contact a remote agency, the client (e.g. an agent, agency, or user application) must have access to an agency proxy object. The remotely accessible functionality of each Grasshopper agency can be separated into the following categories:

- Registration functionality offers detailed information about all places as well as agents/services that are currently hosted by a remote agency.
- Service control functionality enables the remote control of places and agents/services within an agency, such as agent creation, suspension, resumption, transportation and termination.
- Persistence functionality supports the persistent storage of agents/services and places within a remote agency.

- Listener functionality enables the registration and de-registration of AgentSystemListeners for remote agencies.

6.1.1.6.2 The AgentSystemListener Interface

This interface can be used to monitor the events occurring in an agency, and to present them to a user. The listener is notified about any changes associated with agents/services and places, and it retrieves any output from the agency. Each listener is identified by a unique identifier. By default, the Grasshopper platform provides one implementation of the AgentSystemListener interface. However, new implementations may be realized by platform users in order to e.g., create individual graphical user interfaces.

6.1.1.6.3 The RegionRegistration Interface

In order to contact a remote region registry, the client must have access to a registry proxy. The functionality of a Grasshopper region registry comprises the registration and de-registration of agents/services, places, and agencies. In addition, lookup methods enable the retrieval of specific information about the registered components.

6.2 Grasshopper's Limitations

Grasshopper is a lightweight tool to implement mobile agent systems. It relies heavily on the underlying system resources. Each mobile agent is one thread of control, if many mobile agents are running at the same time, the performance of this architecture will be reduced dramatically. The Grasshopper GUI is user friendly but consumes a lot of system resources.

6.3 Implementation of the New Service Architecture

Based on the advanced service architecture introduced in chapter 5 and the MA implementation platform introduced in section 6.1, this section presents the results of a simple simulation of the proposed advanced service architecture. The discussion will also show how the service subscription and actual call setup are implemented.

6.3.1 Design Issues

6.3.1.1 Platforms and Enabling Technologies

The implementation of the MA based advanced service architecture consists of two parts. First part is service subscription using Jini, and the second part is service utilization using Grasshopper. Java is chosen to be the implementation language because of its special characteristics that are suitable for implementing MAs. Java's portability is the main appeal for agents. The use of bytecodes and its interpreted execution environment will make the application run in any system with sufficient resources.

Jini is one of the available distributed technologies. Taking advantage of the Java programming language, it allows the search of services connected by the communication infrastructure and stores not only pointers to the service on the network, but also the code and/or code pointers of these services.

Grasshopper, as described in section 6.1, is a lightweight agent programming environment with a friendly graphical user interface. The best thing about Grasshopper is that, for a user who is new to agent programming, it provides a complete set of

documentation, from technical overview to user’s guide and programmer’s guide. It is also easy to install. More conveniently, there are a variety of agent examples implemented on Grasshopper which illustrate how the agent platform works and explain how most of the important methods function.

6.3.1.2 Implementation Basis

In the following discussions, some basic concepts used for implementing the advanced service architecture are introduced.

1. Basic Call Model

The Basic Call Model describes the state machine of establishing a two-party call. It consists of originating and terminating half-call models. The switching system is

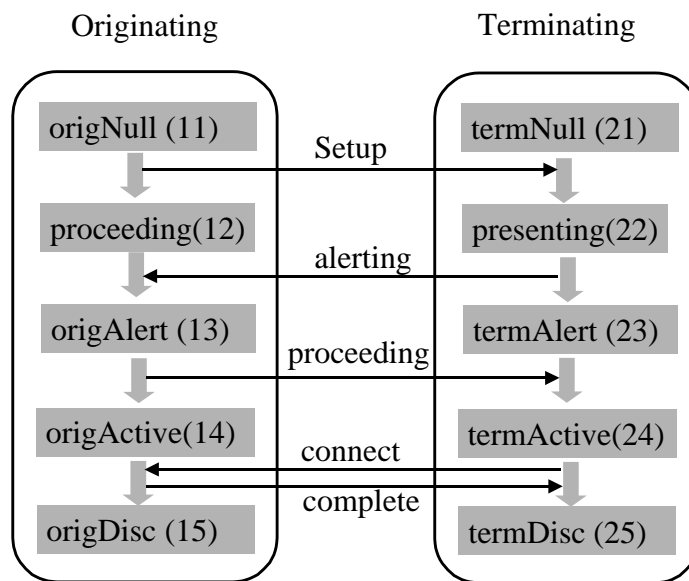


Figure 6.5 Basic Call Model

viewed as two functionally separate sets of call processing logic that create and maintain a basic two-party call. The call origination logic is modeled by the Originating Basic Call Model and the call termination logic is modeled by the Terminat-

ing Basic Call Model. Figure 6.5 shows a simplified version of the IN basic call model.

2. Threads

Threads enhance performance and functionality of various programming languages, including Java, by allowing a program to efficiently perform multiple tasks as if simultaneously. It is a very important tool in implementing the new service architecture.

Simply, a thread is a program's path of execution. The implementation of the new service architecture has to compromise the situation that multiple events occur at the same time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows this to be done. For example, a socket connection between two users needs a thread to keep the connection open. A user service agent and a call agent are different threads of control.

3. Socket

Transporting packets over internet, sources and destinations are specified as socket addresses. Each socket address is a communication identifier that consists of an IP address and a port number. When messages are sent, the messages are queued at the sending socket until the underlying network protocol has transmitted them. When they arrive, the messages are queued at the receiving socket until the receiving process makes the necessary calls to process them.

There are two communication protocols that one can use for socket programming: datagram communication (e.g. IP) and connection-oriented stream communication (e.g. TCP). In our case, call processing signaling messages need to be received in the order they were sent, and all available signaling messages are processed immediately in the same order they were received. TCP guarantees that the packets sent will be received in the order in which they were sent, so TCP is chosen to be the transport protocol.

6.3.2 Class Diagram

Figure 6.6 shows the class diagram of the implementation of the proposed MA based service architecture. SMU is the center of the whole structure. It downloads an enterprise service proxy object in response to a user's service subscription request. When a user makes a call, the SMU creates a *TalkServer* to manage the call set up procedure. A *CallServerThread* is generated by *TalkServer*, and it manages the call signaling processes. In the case of a call agent being activated, it will transfer signaling messages to the call agent. Following are descriptions of the main interfaces and classes.

Interfaces:

- *CreatorInterface*: provides SMU with accessing interfaces to the *ServiceComponentCreator*.
- *EnterpriseInterface*: provides access to the *EnterpriseServiceCreator*.
- *RemoteCreatorInterface*: provides remote access to the *ServiceComponentCreator*.

- *RemoteEnterpriseInterface*: provides remote access to the *EnterpriseServiceCreator*.
- *SubscribeInterface*: provides subscription interface for user.

Classes

- *EntityCreator*: creates service components that are stored in the *ServiceComponentRepository*, advertises its proxy object onto the Jini Lookup service, has a web server running for *CallAgent* to download service classes.
- *EnterpriseCreator*: customizes the services downloaded from the lookup service, advertises the tailored services proxy code onto Jini lookup service within the enterprise, provides the service subscription applets.
- *EntityCreatorProxy*: contains the proxy for the service object.
- *EnterpriseProxy*: contains the proxy for the service object.
- *DiscoverMgr(Uni)*: unicast discover manager, looks up services from the enterprise lookup service.
- *DiscoverMgr(Multi)*: multicast discover manager, looks up services outside the enterprise lookup service.
- *SMU*: service management unit, manages the service subscription and service utilization.
- *UserServiceAgent*: carries the names of the service classes which will be used by *callAgent*, and also carries service data, it moves between the gatekeeper agency and the end user agency.

- *CallAgent*: contains call model which is used for the call processing, moves between the gatekeeper agency and the end user agency.
- *UserServiceAgentP*: the communication proxy for *UserServiceAgent*.
- *CallAgentP*: the communication proxy for *CallAgent*.
- *OrigClient*: the end user who makes the call.
- *TermClient*: the end user who receives the call.
- *CallServerThread*: manages the call processing, including transferring call signaling messages and maintaining the sockets between the caller and the callee.
- *TalkServer*: generates *CallServerThread*.

6.3.3 Service Subscription and Realization Using Jini

In the following discussion, details of implementing service subscription and realization for the new service architecture using Jini will be presented.

1. *EntityCreatorProxy*, as shown in Figure 6.7, is the object that *ServiceComponentCreator* uploads onto the Lookup service which resides outside an enterprise. It contains all the interfaces for accessing the *EntityCreator*, such as the API to get the code repository URL and the API to get the service subscription Graphical User Interface for subscribing generic services. Following are the two major APIs:

- *getCodeURL()* gets the component repository URL, so that a SMU can download required service component from it.

- *getGUI()* gets generic VPN service subscription GUI, this GUI is used to customize the generic service for the end user by filling out enterprise specific data.

```
public class EntiCreatorProxy implements CreatorInterface{
    .....
    public String getCodeURL() throws Exception {.....}
    public Applet getGUI() {.....}
    .....
}
```

Figure 6.7 Component Service Creator Proxy Implementation

2. *EnterpriseProxy*, as shown in Figure 6.8, is the object EnterpriseServiceCreator uploads onto the Lookup service. SMU invokes *getGUI()* to get a service subscription form from the EnterpriseServiceCreator in response to a user's service subscription request.

```
public class EnterpriseProxy implements EnterpriseInterface{
    .....
    public Applet getGUI() {.....}
    .....
}
```

Figure 6.8 Enterprise Service Creator Proxy Implementation

```
public class EntityCreator extends UnicastRemoteObject
    Implements RemoteCreatorInterface,
    ServiceIDListener{
    .....
    public static void main(String [] args) throws RemoteException{
        EntityCreator entityCreator = new EntityCreator(
            http://remoteHost:8080);
        entityCreator.discoverAndJoin();
    }
    .....
}
```

Figure 6.9 Service Component Creator

3. *EntityCreator* (Service Component Creator), as shown in Figure 6.9, discovers the available lookup service throughout Internet using multicast method, and uploads its own service onto the found lookup service.
4. *EnterpriseCreator*, shown in Figure 6.10, is responsible for discovering an available lookup service which has the same type of service and uploads its service proxy object onto it. It is also responsible for discovering lookup services outside the enterprise and looking up for SMU specified service.

```

public class EnterpriseCreator implements RemoteEnterpriseInterface{
    .....
    public static void main(String [] args) throws RemoteException{
        EnterpriseCreator eCreator = new EnterpriseCreator();
        eCreator.discoverAndJoin();
        eCreator.discover();
        eCreator.lookup();
        .....
    }
    .....
}

```

Figure 6.10 Enterprise Service Creator Implementation

5. SMU, shown in Figure 6.11, responds to user's service subscription request, discovers the lookup service and looks up the type of services user requested. It also downloads the service proxy object from this lookup service.

```

public class SMU {
    .....
    public static void main(String [] args) {
        SMU managementUnit = new SMU();
        managementUnit.discover();
        managementUnit.lookup();
        .....
    }
    .....
}

```

Figure 6.11 SMU Implementation

6.3.4 Service Utilization Realization Using Grasshopper

In this section, a few code examples are presented to show how mobile agents, user service agent and call agent are implemented using Grasshopper for service utilization in the new service architecture.

Service utilization is performed by the SMU, which can reside in the gatekeeper or can be a stand-alone device. After a successful service subscription, when the user clicks the “finish” button on the subscription applet, the SMU creates a new `UserServiceAgent`. The sample code for this process is shown in Figure 6.12.

```

.....
AgentSystem agentSystemP = new AgentSystem(
                                "de.ikv.grasshopper.agency.AgentSystem",
                                gkAddress);
try{
    agentInfo = agentSystemP.createService("UserServiceAgent",
                                           codeSource,
                                           "ServicePlace",
                                           null);
    .....
}

```

Figure 6.12 User Service Agent Creation Code Example

Once the `UserServiceAgent` is created successfully, its globally unique identifier along with the end user’s IP address and name will be stored in the SMU’s *userProfile*, so that when an end user makes or receives a call, the SMU will know with which user this USA is associated. Figure 6.13 shows the code that implements this.

```

.....
usald = agentInfo.getIdentifier();
userProfile.put(userName, IPAddress, usald);
.....
}

```

Figure 6.13 The UserServiceAgent ID is Stored in the SMU

The `UserServiceAgent` is a mobile agent, it moves to the end user agency after it is created. When a user makes or receives a call, if there is an USA for the user, the USA will be activated. If the gatekeeper routed call signaling mode is used, the USA will move to the gatekeeper agency, otherwise it will stay in the end user agency.

The class `de.ikv.grasshopper.agency.MobileAgent` has to be inherited by each Grasshopper mobile agent. Method `live()` is the core of each agent. The method must be overridden, since it specifies the designated task. Method `action()` can be used by agent programmers to implement any agent behavior that a user will be able to invoke. Method `move()` moves the agent to another location. Optionally, the name of a method can be specified as an input parameter. In this case, the specified

```

public void live(){
    if (state == 0){
        try{
            String remoteAddress = new String(userAddress);
            try{
                remote = new Location(remoteAddress);
                String [] homeAddress = Configurator.getConfigurator().
                    getCommunicationServer().
                    getReceiverAddressAsString ();
                home = new Location(homeAddress);
                state++;
                move(remote);
            } catch (Exception e){.....}
        } catch (Exception e){.....}
    } else if (state == 1){
        try{
            state++;
            move(home);
        } catch (Exception e){.....}
    }
    else if (state == 2){
        state = 0;
        this.setState(State.SUSPENDED);
    }
}

```

Figure 6.14 A Sample Implementation of Method `live()` of USA

method will be executed directly after the migration. If no method name is given, the *live()* method will be executed by default.

Figure 6.14 and Figure 6.15 show two pieces of code that are sample implementation of the *live()* method and *action()* method from *UserServiceAgent*.

- A sample implementation of method *live()* is shown in Figure 6.14, in which the USA is created in the gatekeeper with *homeAddress*. The USA will move to a remote user agency whose address is *remoteAddress*. It changes its state as it moves to another agency, “*state ==2*” means that the USA has moved back to the gatekeeper. Because there is no clear instruction for the USA as what to do next, it is suspended by using *state.setState(State.SUSPENDED)*.
- A sample implementation of method *action()* is shown in Figure 6.15. When a user makes a call, if there is a *userId* stored in the SMU’s *userProfile* of either the calling party or the called party, a CA will be instantiated from the USA and this CA’s code is downloaded from *codeSource*.

```

public void action(){
    this.setState (State.ACTIVE);
    ServiceInfo agentInfo = createService ("CallAgent",
                                           codeSource,
                                           "ServicePlace",
                                           new Object [] {getIdentifier().toString(),
                                                           gkAddress});
}

```

Figure 6.15 Sample Implementation of action() Method

6.3.5 Simulation Results

In this section, some of the simulation results are presented to show how the new service architecture is used for service subscription and service invocation. The

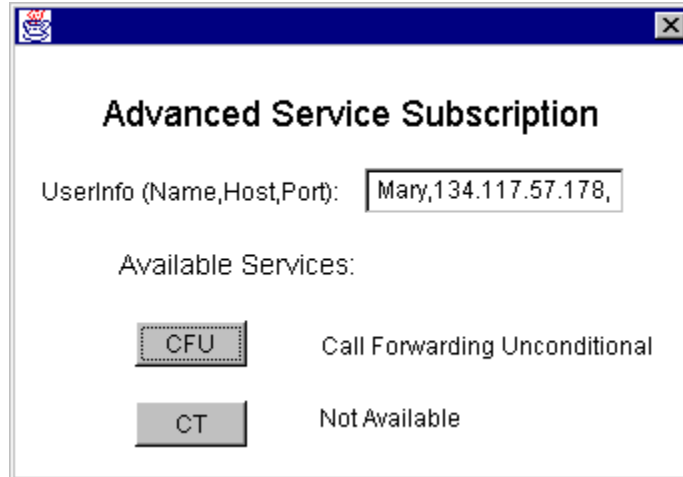


Figure 6.16 Advanced Service Subscription GUI

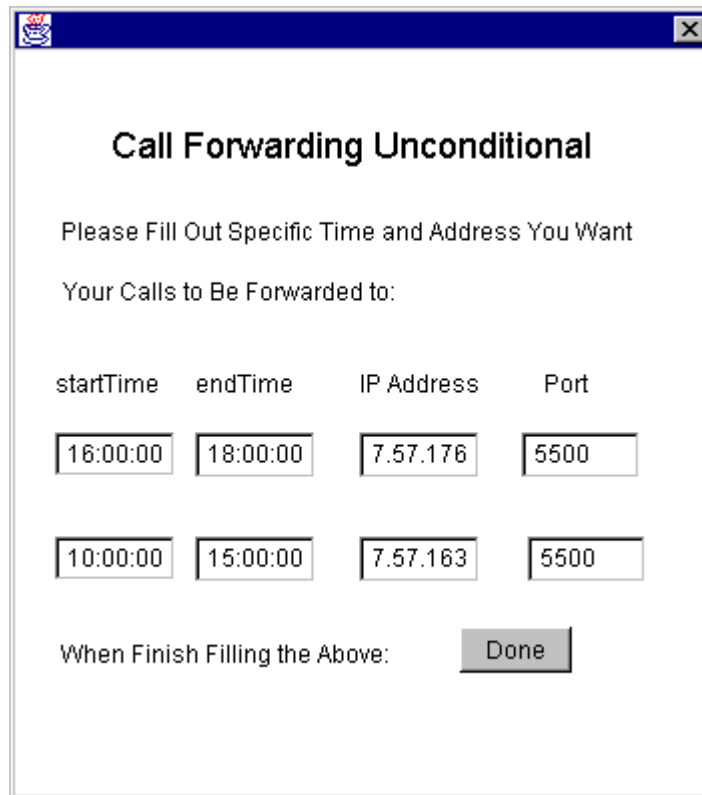
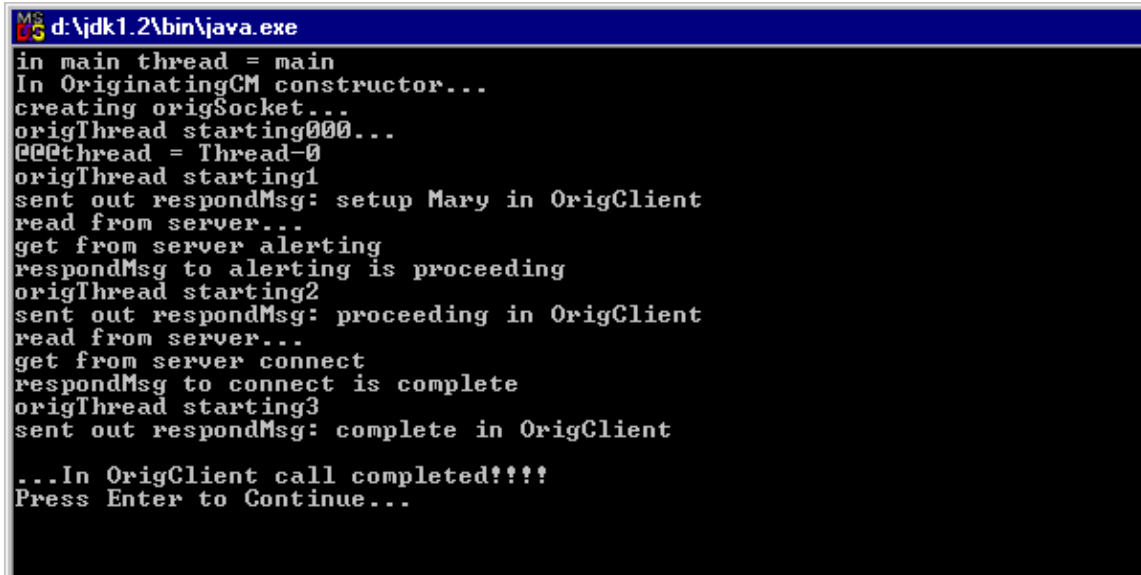


Figure 6.17 Service Customization Form for CFU

following figure shows the service subscription form a user gets. When the user clicks “CFU” on Figure 6.16, a form for customization of CFU service will pop up as shown in Figure 6.17. The user can specify which period of time his phone call

should be forwarded. Once the customization is finished, the user clicks the “Done” button.

The following window as shown in Figure 6.18 illustrates the log of messages passed by the *TalkServer* between parties.



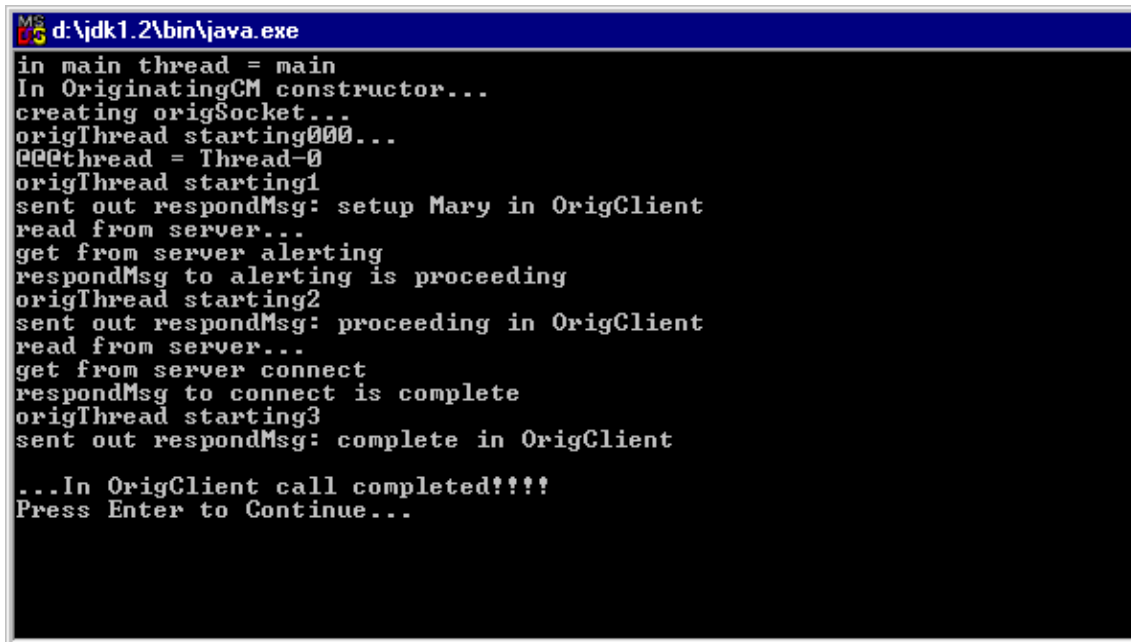
```

MS-DOS d:\jdk1.2\bin\java.exe
in main thread = main
In OriginatingCM constructor...
creating origSocket...
origThread starting000...
@@@thread = Thread-0
origThread starting1
sent out respondMsg: setup Mary in OrigClient
read from server...
get from server alerting
respondMsg to alerting is proceeding
origThread starting2
sent out respondMsg: proceeding in OrigClient
read from server...
get from server connect
respondMsg to connect is complete
origThread starting3
sent out respondMsg: complete in OrigClient

...In OrigClient call completed!!!!
Press Enter to Continue...

```

Figure 6.18 TalkServer Message Log



```

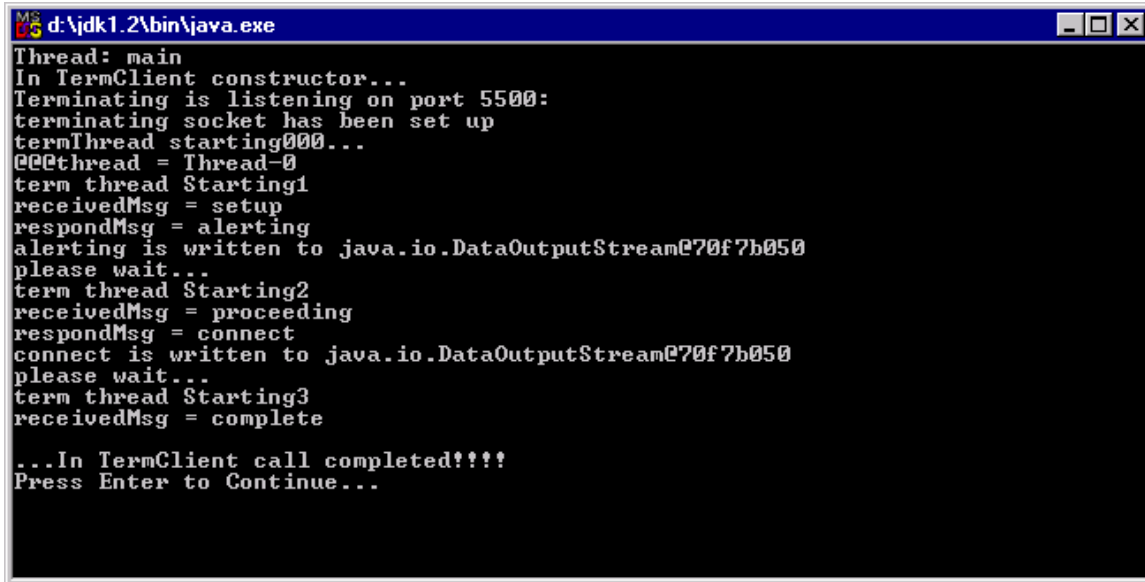
MS-DOS d:\jdk1.2\bin\java.exe
in main thread = main
In OriginatingCM constructor...
creating origSocket...
origThread starting000...
@@@thread = Thread-0
origThread starting1
sent out respondMsg: setup Mary in OrigClient
read from server...
get from server alerting
respondMsg to alerting is proceeding
origThread starting2
sent out respondMsg: proceeding in OrigClient
read from server...
get from server connect
respondMsg to connect is complete
origThread starting3
sent out respondMsg: complete in OrigClient

...In OrigClient call completed!!!!
Press Enter to Continue...

```

Figure 6.19 Caller Messaging Information Log

Figure 6.19 shows the log of messaging information of the user who makes the phone call. The originating messages are sent to TalkServer and the terminating messages from called party are received, as recorded in the figure.



```

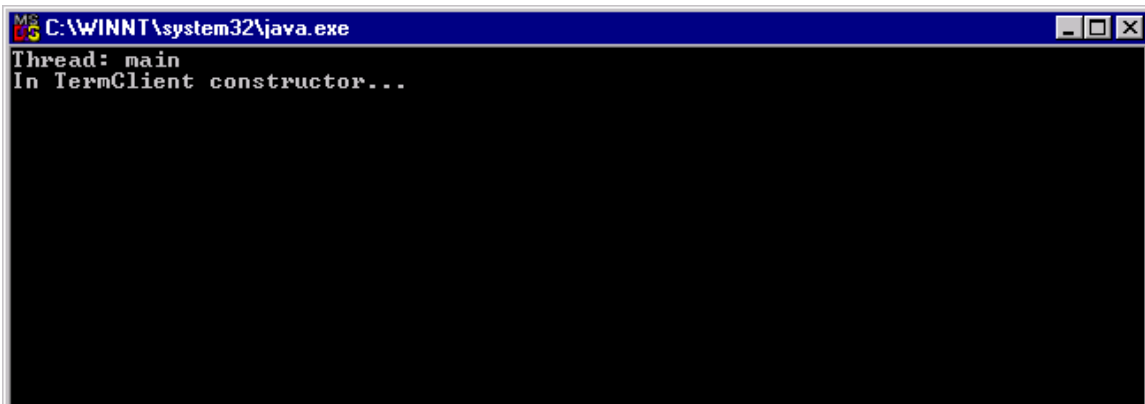
d:\jdk1.2\bin\java.exe
Thread: main
In TermClient constructor...
Terminating is listening on port 5500:
terminating socket has been set up
termThread starting000...
@@@thread = Thread-0
term thread Starting1
receivedMsg = setup
respondMsg = alerting
alerting is written to java.io.DataOutputStream@70f7b050
please wait...
term thread Starting2
receivedMsg = proceeding
respondMsg = connect
connect is written to java.io.DataOutputStream@70f7b050
please wait...
term thread Starting3
receivedMsg = complete

...In TermClient call completed!!!!
Press Enter to Continue...

```

Figure 6.20 Message log of Forwarded-to Terminal

Figure 6.20 is a snapshot of the log for the user terminal with IP address 134.117.57.178. Figure 6.21 shows a snapshot of the log for the user terminal with IP address 134.117.57.163. The phone call is redirected to the IP Address 134.117.57.178 specified by the user since the user has subscribed to the CFU service.



```

C:\WINNT\system32\java.exe
Thread: main
In TermClient constructor...

```

Figure 6.21 Message log of the forwarded-to Terminal

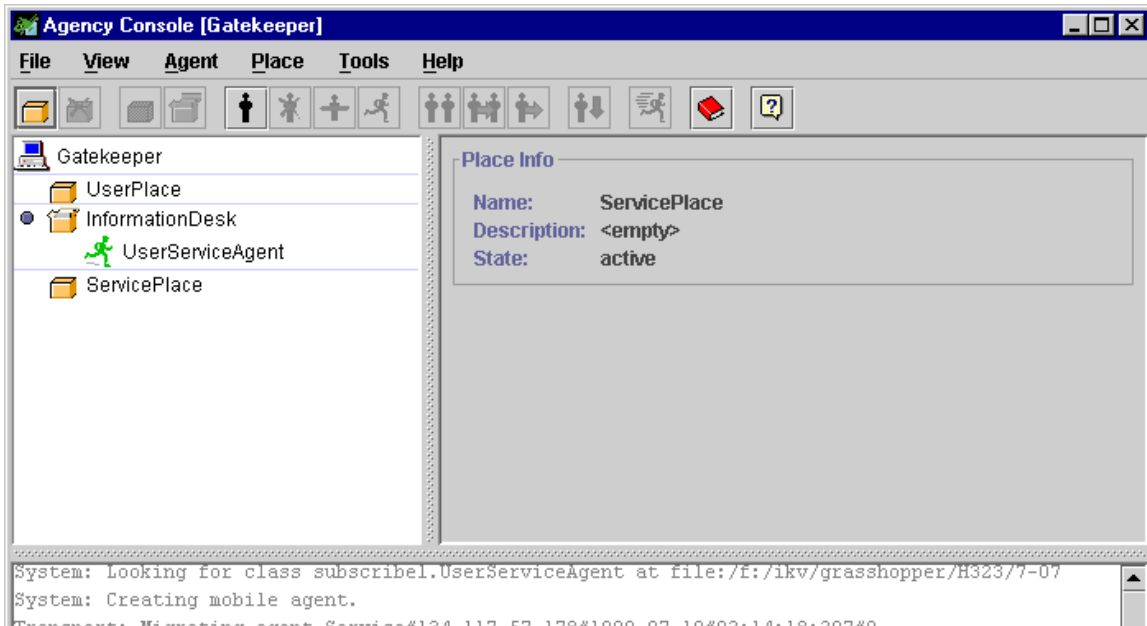


Figure 6.22 USA Construction

Figure 6.22 illustrates that an USA has been constructed in the gatekeeper’s agency in the ServicePlace according the information received from the end user during the service subscription.

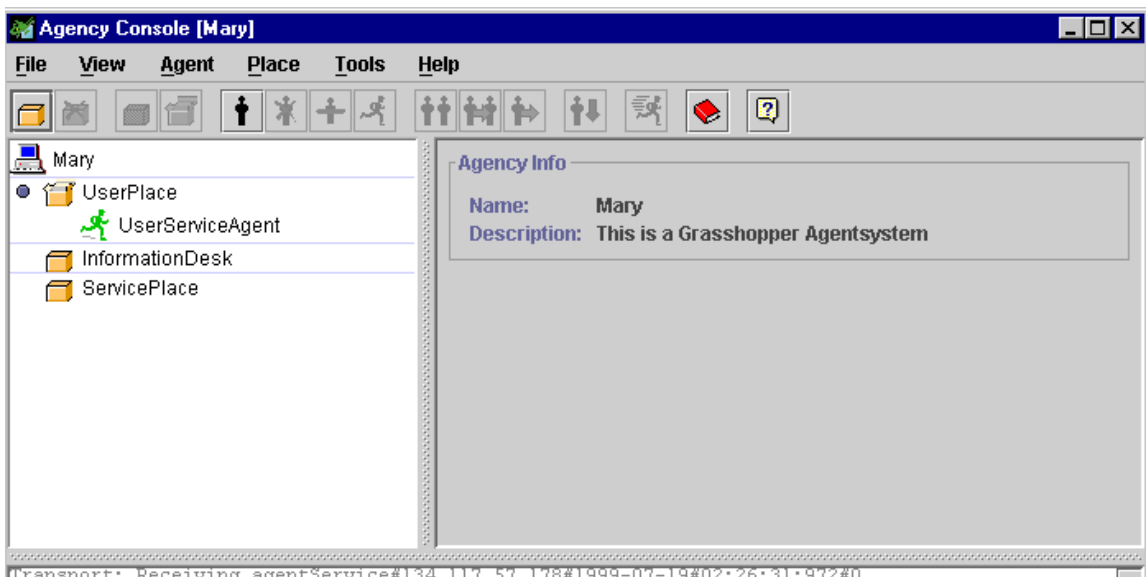


Figure 6.23 USA Moving to the End User’s Agency

Figure 6.23 shows that after the construction of the USA in the gatekeeper's agency, the usa moved to the end user's agency in the servicePlace.

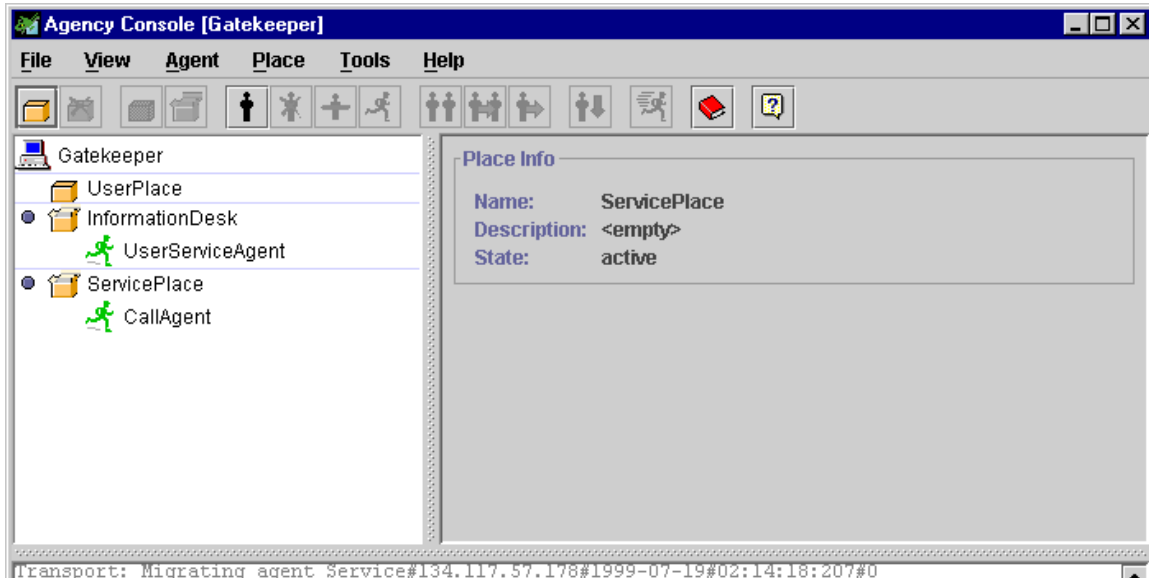


Figure 6.24 Instantiation of a CA

Figure 6.24 shows the MAs in the gatekeeper's agency. An USA moved back to the gatekeeper's agency from the end user's agency and when a user associated with this USA makes or receives a phone call, a CA is instantiated in the servicePlace.

The above figures have shown a set of test results which provide visual information indicating how the new service architecture can be implemented and utilized. There could be other scenarios that come out of this implementation depending on the IP addresses specified by the end user.

6.3.6 Limitation of the Simulation

This simulation is presented to show an example as how the new advanced service architecture can be implemented. It is by no means a complete or the only implementation. There are some limitations of this simulation:

- Due to the limitation of Grasshopper's demonstration version, only 3 mobile agents can be created at maximum, more advanced services will be hard to simulate using the current Grasshopper infrastructure.
- The performance of the simulation is hard to measure because of the low speed of the machine used in the experiment.

Chapter 7 Conclusions And Future Work

In this thesis, a new Mobile Agent based advanced service architecture for IP telephony is presented. A sample implementation of the service architecture using Grasshopper and Jini is outlined to show how the service architecture supports flexible service subscription and service invocation. From the above discussions, we come to some conclusions and some working items for future research.

7.1 Conclusions

From the previous presentation and the testing results, we can see that the proposed mobile agent based advanced service architecture solution is flexible, distributed and open. Comparing to the traditional service architectures, it can:

- support universal access to services through the Jini Lookup process.
- enable the provision of flexible software solutions, where H.323 supplementary services software is partitioned into mobile service agents realizing dedicated functionalities (e.g., IN service features).
- enable on demand provision of customized supplementary services by dynamically constructing user service agents using service code downloaded from the SCC or the ESC to the gatekeeper.

- allow a decentralized realization of supplementary services, by means of bringing the user service agent directly onto the user terminals.

On the other hand, there are some issues of this mobile agent service architecture that need to be noted or addressed in future study:

- It consumes a lot of resources due to the fact that each user who has subscribed the advanced services needs a USA and a CA. This may not be an issue as computer speed and memory increases dramatically from year to year.
- Call set up procedure relies on the end users' computer resources. If the processing capability is limited, this process would take longer time. But same as the previous point, this may not be a problem at all as computer resources become cheaper and cheaper.
- Using this service architecture, end users should have the ability to tailor their own services which means that they have to be clear what kind of services they want. This issue can be solved by providing a default service setting for the customer.
- The presented simulation of the new service architecture is based on the IP address of the end user's computer, this leads to the possibility that people other than the owner can use it if they know the correct login name and password. The security issue is for sure a future work direction.

7.2 Future Work

During the course of this work, several directions have been identified for future research and development.

Security of mobile agents is a very important issue. What are the security mechanisms that can be used to prevent the system from malicious agents' attack? More research is needed to develop reliable security for mobile agents.

Another direction for future research is to investigate the performance of the newly proposed service architecture, and understand how the system works while handling a high volume of calls simultaneously.

More supplementary services such as call transfer can be implemented, using the proposed service architecture. Additional services will likely to improve and enhance the service architecture as well.

Appendix A Acronyms

AIN	Advanced Intelligent Network
CA	Call Agent
CFU	Call Forwarding Unconditional
CORBA	Common Object Request Broker Architecture
CT	Call Transfer
DAI	Distributed Artificial Intelligence
DFP	Distributed Functional Plane
ESC	Enterprise Service Creator
GFP	Global Functional Plane
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IN	Intelligent Networks
INCM	Intelligent Networks Conceptual Model
IP	Internet Protocol
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LAN	Local Area Network
LUS	Lookup Service
MA	Mobile Agent
OCA	Outgoing Call Allowance
OGR	Outgoing Call Restriction

PP	Physical Plane
PSTN	Public Switched Telephone Network
RPC	Remote Procedure Call
SCC	Service Component Creator
SCP	Service Control Point
SCR	Service Component Repository
SIB	Service Independent Building Block
SIP	Session Initiation Protocol
SIR	Service Implementation Repository
SMS	Service Management System
SMU	Service Management Unit
SP	Service Plane
SS7	International Signaling System No. 7
SSP	Service Switching Point
TINA	Telecommunications Information Networking Architecture
TINA-C	Telecommunications Information Networking Architecture Consortium
TMN	Telecommunications Management Network
USA	User Service Agent
VPN	Virtual Private Network

Appendix B References

1. “Internet Telephony”, <http://www.webproforum.com/siemens2>.
2. D. B. Lange, “Present and Future Trends of Mobile Agent Technology”, Second International Workshop on Mobile Agents’98 (MA’98) Stuttgart, Germany, September 1998.
3. ITU-T Rec. H.323, “Visual Telephone Systems and Terminal Equipment for Local Area Networks which Provide a Non-Guaranteed Quality of Service”, 1996.
4. ITU-T Rec. H.225.0, “Media Stream Packetization and Synchronization for Visual Telephone Systems on Non-Guaranteed Quality of Service LANs”, 1997.
5. ITU-T Rec. H.245, “Control Protocol for Multimedia Communication”, 1998.
6. ITU-T Rec. H.450.1, “Generic Functional Protocol for the Support of Supplementary Services in H.323”, 1998.
7. ITU-T Rec. H.450.2, “Call Transfer Supplementary Service for H.323”, 1998.
8. ITU-T Rec. H.450.3, “Call Diversion Supplementary Service for H.323”, 1998.
9. H. Schularinne, J. Rosenbery, “comparison of H.323 and SIP”, <http://www.cs.columbia.edu/~hgs/sip/h323.html>.

10. "H.323 Tutorial", <http://www.webproforum.com/trillium/index.html>.
11. A. Gary, "H.323: The Multimedia communications Standard for Local Area Networks", IEEE Communications Magazine, December 1996.
12. ITU-T Rec. H.235, "Security and Encryption for H-series (H.323 and H.245-based) Multimedia", 1998.
13. ITU-T Rec. H.246, "Internetworking of H-series Multimedia Terminals with H-series Multimedia Terminals and Voice/voiceband Terminals on GSTN and ISDN", 1998.
14. ITU-T Rec. Q.931, "ISDN User-network Interface Layer 3 Specification for Basic Call Control", 1998.
15. ITU-T Rec. H.320, "Narrow-band Visual Telephone Systems and terminal Equipment", 1999.
16. ITU-T Rec. H.324, "Terminal for Low Bit-rate Multimedia Communication", 1998.
17. ITU-T Rec. H.310, "Broadband Audio Visual Communication Systems and terminals", 1998.
18. ITU-T Rec. H.321, "Adaption of H.320 Visual Telephone Terminals to B-ISDN Environments", 1998.
19. ITU-T Rec. H.322, "Visual Telephone Systems and Terminal Equipment for Local Area Networks which Provide a Guaranteed Quality of Service", 1996.
20. ITU-T Rec. T.120, "Data Protocols for Multimedia Conferencing", 1996.

21. R. H. Glitho, "Advanced Services Architectures for Internet Telephony: State of the Art and Prospects".
22. R. Minetti, E. Utsunomiya, "The Service Architecture", <http://www.tinac.com/specifications/abstract.htm>.
23. TMN, <http://www.uhc.dk/tmn.html>.
24. T. Magedanz, "Intelligent Networks", International Thomas Computer Press, 1996.
25. T. Jan, "Intelligent Networks", Mass., Artech House, 1994.
26. V. Avery and J. Matta, "Intelligent Networks: A Concept for the 21st Century", <http://www.dse.doc.ic.ac.uk>.
27. IN, <http://www.webproforum.com/microlegend/index.html>.
28. J. Kiniry, D. Aimmerman, "A Hands - On Look At Java Mobile Agents", <http://computer.org/internet/ic1997/w402labs.htm>.
29. H. S. Nwana, "Software Agents: An Overview", Knowledge Engineering Review, Vol. 11, No.3, pp.1-40, september, 1996.
30. M. Breugst and T. Magedanz, "Mobile Agent - Enabling Technology for Active Intelligent Network Implementation", IEEE Network, May/June 1998.
31. D. B. Lange, "Present and Future Trends of Mobile Agent Technology", Second International Workshop on Mobile Agents'98 (MA'98) Stuttgart, Germany, September 1998.
32. A. Fuggetta, G. P. Picco, G. Vigna, "Understanding Code Mobility", IEEE Transactions on Software Engineering, vol. 24, 1998.

33. D. Chess, C. Harrison, A. Kershenbaum, "Mobile agents: Are they a good idea? - update", Mobile Object Systems: Towards the Programmable Internet, pp.46-48/ Springer-Verlag, April 1997. Lecture Notes in computer Science No.1222.
34. Jini specifications, <http://www.sun.com/jini/specs>.
35. D. Clark, "A Taxonomy of Internet Telephony applications", <http://itel.mit.edu/itel/publications.html>.
36. B. Pagurek, T. White, "A Quick Evaluation of H.323/H.450", Technical Report SEC-99-02, Systems and Computer Engineering, Carleton University, April 1999.
37. T. Magendanz, K. Rothermel, S. Krause, "Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?", INFOCOM 96, San Francisco, USA.
38. Grasshopper, <http://www.ikv.de/products/grasshopper/grasshopper.html>.