Services for Internet Telephony

Jonathan Michael Lennox

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2004

ABSTRACT

Services for Internet Telephony

Jonathan Michael Lennox

Internet telephony — voice transmission and call signalling over IP networks — can provide services far beyond those of the circuit-switched telephone network. This thesis discusses Internet telephony services in four broad areas: user-location services; multi-party conferencing; the interworking of Internet telephony and mobile telephony; and Internet telephony feature interaction.

User-location services are services which modify how a telephony server locates a user. Service authors need a way to control this process; this thesis presents two of them. The SIP Common Gateway Interface (SIP CGI) is a low-level server interface which allows fine-grained control of message processing in Session Initiation Protocol (SIP) servers. The Call Processing Language (CPL) is a protocol-independent, inherently safe high-level language for describing services in a way that is easily created and edited. The thesis also describes a general service framework providing a straightforward and powerful API atop which these and other service execution environments can be implemented, and an event thread architecture that makes implementation of transaction-based protocols such as SIP efficient and scalable.

Multi-party conferencing involves calls in which three or more people communicate simultaneously. This thesis presents a new approach to conferencing in which a fully-distributed, decentralized protocol establishes a fully connected mesh of signalling and media connections between conference participants.

Internet telephony needs to be able to connect to circuit-switched mobile telephony networks. The thesis presents a family of system architectures which allow SIP and UMTS networks to be connected directly, allowing traffic to flow directly to the mobile switching center handling a user's mobile terminal. These architectures eliminate triangular routing, transcoding, and other

inefficiencies of indirect connections.

Finally, whenever services are defined, the issue arises of feature interaction, in which several features or services interact in unexpected and potentially undesirable ways. This thesis explains how Internet telephony alters the feature interaction problem, discusses the applicability of existing resolution techniques, and presents some new approaches for resolving interactions in the Internet environment.

Contents

List of Figures viii

List of Tables xi

Acknowledgments xii

Chapter 1 Introduction 1

1.1 Problem Statements and Original Contribution 2

1.1.1 User-Location Services 2

1.1.2 Multi-Party Conferencing 4

1.1.3 Internet Telephony Interaction with Circuit-Switched Mobile Networks . 5

1.1.4 Feature Interaction 5

1.2 Overview of Thesis 6

Chapter 2 Internet Telephony and User Location Services: Background 7

2.1 Overview of Telephony Services 7

2.1.1 Definition of Telephony Services 8

2.2 Internet Telephony Architecture 9

2.2.1 Model components 9

2.2.2 Interactions of Model Components 11

2.3 User Location Services 12

2.3.1 Example User-Location Services 13

2.3.2 Non-User-Location Services 14

2.4 Service Creation and Execution 15
2.4.1 What a Service Execution Environment Does 16
2.4.2 Which Service is Executed 16
2.4.3 Where a Service Executes 17
2.5 Creation and Transport of a Service Description 18
2.6 Properties of a Service Creation Mechanism 19
2.6.1 Program Invocation Times 20
2.6.2 Resource Restrictions 21
2.6.3 Interface 22
2.7 The Session Initiation Protocol 22
2.8 Conclusion 26

Chapter 3 User-Location Services: Related Work 28
3.1 Intelligent Network Services 28
3.2 Internet Services 32
3.2.1 Dynamic Web Content 32
3.2.2 E-Mail Services 33
3.2.3 Active Networks 36
3.3 Telephony Application Programming Interfaces 36
3.4 Telephony Scripting Languages 37

Chapter 4 The Common Gateway Interface for SIP 42
4.1 Introduction 42
4.2 Motivations 43
4.3 Comparison between HTTP CGI and SIP CGI 44
4.3.1 Basic Model 45
4.3.2 Persistence Model 46
4.3.3 SIP CGI Triggers 48
4.3.4 Naming 48
4.3.5 Environment Variables 48

4.3.6 Timers . . . 49
4.4 Services Enabled . . . 49
4.5 Example CGI Operation . . . 50
4.6 Overview of SIP CGI . . . 52
4.7 SIP CGI Specification Details . . . 54
4.7.1 Invoking the Script . . . 54
4.7.2 Data Input to the SIP CGI Script . . . 55
4.7.3 Data Output from the SIP CGI Script . . . 55
4.7.4 Local Expiration Handling and Locally-Generated Responses . . . 58
4.7.5 SIP CGI and REGISTER . . . 58
4.8 SIP CGI Implementation Experience . . . 59
4.8.1 Projects using SIP CGI . . . 59
4.9 Strengths and Weakness of SIP CGI . . . 60
4.10 Conclusion . . . 61

Chapter 5 The Call Processing Language 62

5.1 Introduction . . . 62
5.1.1 Rejected Solutions . . . 63
5.2 Inspirations for This Work . . . 64
5.3 The Call Processing Language: Overview . . . 64
5.4 Structure of CPL Scripts . . . 65
5.4.1 Abstract structure . . . 65
5.4.2 XML Structure . . . 66
5.4.3 High-level Structure . . . 67
5.4.4 Location Model . . . 68
5.5 Switches . . . 68
5.6 Location Modifiers . . . 70
5.7 Signalling Operations . . . 71
5.8 Non-signalling Operations . . . 73
5.9 Subactions . . . 73

5.10 Default Behavior . . . 74
5.11 CPL Extensions . . . 75
5.12 Examples . . . 76
5.12.1 Example: Call Redirect Unconditional . . . 76
5.12.2 Example: Call Forward Busy/No Answer . . . 76
5.12.3 Example: Call Forward: Redirect and Default . . . 77
5.12.4 Example: Call Screening . . . 77
5.12.5 Example: Priority and Language Routing . . . 78
5.12.6 Example: Outgoing Call Screening . . . 79
5.12.7 Example: Time-of-day Routing . . . 80
5.12.8 Example: Location Filtering . . . 80
5.12.9 Example: Non-signalling Operations . . . 81
5.12.10 Example: Hypothetical Extensions . . . 82
5.12.11 Example: A Complex Example . . . 83
5.13 Design Evaluation . . . 84
5.14 Implementation Experience . . . 86
5.15 Conclusion . . . 87

Chapter 6 Design and Implementation of the CINEMA Policy Framework 88

6.1 Introduction . . . 88
6.1.1 The Policy Framework . . . 89
6.2 Other Systems' Approaches to Policy Definitions . . . 90
6.2.1 Apache . . . 90
6.2.2 Intelligent Networks . . . 92
6.2.3 Firewalls and Application-Layer Gateways . . . 93
6.3 CINEMA Policy API Architecture . . . 94
6.3.1 Policy Methods . . . 94
6.3.2 How Policy Methods Perform Actions . . . 96
6.3.3 Utility Functions for Policies . . . 96
6.3.4 How the policy core is started and completes . . . 97

6.4 Example Policy Flows . . . 98
6.4.1 A Simple Policy Flow . . . 98
6.4.2 A User Policy Flow . . . 98
6.5 Implementation Notes on Specific Policies . . . 101
6.5.1 SIP CGI . . . 101
6.5.2 Java Servlets . . . 101
6.5.3 CPL . . . 102
6.5.4 Stepped Proxy . . . 102
6.6 Further Development . . . 103
6.6.1 Multiple Policies for One Transaction . . . 103
6.7 Conclusion . . . 104

Chapter 7 Design and Implementation of the CINEMA Reactive System Model 105
7.1 Introduction . . . 105
7.2 The Initial CINEMA Architecture . . . 106
7.2.1 Bottlenecks in this Architecture . . . 107
7.3 Related Work . . . 108
7.4 The Reactive System Model . . . 109
7.4.1 Details of the Reactive System Approach . . . 110
7.4.2 Use of Reactive Systems in the New CINEMA Architecture . . . 112
7.5 Performance Analysis . . . 113
7.5.1 Performance Results . . . 114
7.6 Future CINEMA Improvements . . . 116
7.7 Conclusion . . . 117

Chapter 8 A Protocol for Reliable Decentralized Conferencing 118
8.1 Introduction . . . 118
8.2 Related Work . . . 119
8.3 Existing Conferencing Models . . . 120
8.3.1 Existing Conferencing Architecture: Multicast . . . 120

8.3.2 Existing Conferencing Architecture: Mixing . . . 123
8.4 Full Mesh Conferencing . . . 123
8.4.1 Example . . . 125
8.4.2 Protocol Messages . . . 128
8.4.3 Membership Maintenance and State Communication . . . 129
8.4.4 The Double-Dialog Glare Problem . . . 130
8.4.5 Immediate Departure and Reconnection . . . 131
8.5 Security and Authentication . . . 132
8.6 Verification of Full Mesh Protocol . . . 134
8.6.1 The Verification Framework . . . 134
8.6.2 Test Runs Performed . . . 135
8.7 Analysis and Rationale . . . 137
8.7.1 Protocol Correctness . . . 137
8.7.2 Rationale for Three-Phase Session Establishment . . . 138
8.8 Realization of the Full Mesh Protocol in SIP . . . 138
8.9 Future Work . . . 140
8.10 Conclusion . . . 140

Chapter 9 Interworking Internet Telephony and Wireless Telecommunications Networks 142

9.1 Introduction . . . 142
9.2 Background . . . 146
9.2.1 UMTS Mobility and Call Delivery . . . 146
9.2.2 SIP Mobility and Call Delivery . . . 148
9.3 Architecture . . . 149
9.3.1 SIP/UMTS Interworking: Calls from UMTS to SIP . . . 150
9.3.2 SIP/UMTS Interworking: Mobile-Terminated Calls . . . 151
9.4 Analysis . . . 156
9.5 Compatibility With Non-IP-Enabled Visited Networks . . . 162
9.5.1 Interoperation Approaches for the Three Proposals . . . 163

9.5.2 Analysis of Non-IP-enabled Scenarios . . . 168
9.6 Discussion . . . 171
9.6.1 Further Work . . . 172
9.7 Conclusion . . . 173

Chapter 10 Feature Interaction in Internet Telephony **174**
10.1 Introduction . . . 175
10.2 Internet Telephony Architectural Model . . . 176
10.3 Differences From the PSTN . . . 176
10.3.1 Advantages of Internet Telephony for Handling Feature Interaction . . . 177
10.3.2 New Complications . . . 183
10.4 Applicability of Existing Feature Interaction Work . . . 185
10.5 Examples of New Interactions in Internet Telephony . . . 186
10.5.1 Cooperative Interactions . . . 187
10.5.2 Adversarial Interactions . . . 188
10.6 New Approaches for Managing Internet Interactions . . . 190
10.6.1 Explicitness . . . 190
10.6.2 Universal Authentication . . . 191
10.6.3 Network-level Administrative Restriction . . . 191
10.6.4 Verification Testing . . . 191
10.7 Conclusion . . . 192

Chapter 11 Conclusion **193**
11.1 Internet Services: Distributed Intelligence . . . 193
11.2 Extensions, In and Beyond Telephony . . . 195
11.3 Conclusion . . . 197

List of Figures

2.1 Possible Paths of Call Setup Messages 11
2.2 Example SIP Message with SDP 25
2.3 SIP Operation 26

3.1 IN Conceptual Model, from Q.1201 29
3.2 Call Flow for a Call to an 800 Number 30
3.3 A Sample Sieve Script 35
3.4 A Sample VoiceXML Script 37
3.5 A Sample CCXML Script 39
3.6 A Sample SCML Script 40
3.7 A Sample LESS Script 41

4.1 HTTP CGI Model 45
4.2 SIP CGI Model 45
4.3 A Sample SIP CGI Script, in Perl 52

5.1 Sample CPL Script: Graphical Version 66
5.2 Sample CPL Script: XML Version 67
5.3 Rejected Alternate Approach to CPL Extensions 76
5.4 Example Script: Call Redirect Unconditional 76
5.5 Example Script: Call Forward Busy/No Answer 77
5.6 Example Script: Call Forward: Redirect and Default 78
5.7 Example Script: Call Screening 78

5.8 Example Script: Priority and Language Routing 79
5.9 Example Script: Outgoing Call Screening 79
5.10 Example Script: Time-of-day Routing 80
5.11 Example Script: Location Filtering 81
5.12 Example Script: Non-signalling Operations 81
5.13 Example Script: Hypothetical Distinctive-Ringing Extension 82
5.14 Example Script: Hypothetical Regular-Expression Extension 82
5.15 Example Script: A Complex Example 83

6.1 SIPD policy flow for 480 Temporarily Unavailable 99
6.2 SIPD policy flow for a CGI invocation 100

7.1 Memory Usage and Performance of CINEMA SIPD 1.21 115
7.2 Memory Usage and Performance of CINEMA SIPD 1.23 115

8.1 Conferencing: End System Mixing 122
8.2 Conferencing: Conference Server Mixing 122
8.3 Conferencing: Full Mesh 124
8.4 Conferencing: Combination of Conference Servers (S) and Full Mesh 125
8.5 Example Full Mesh Message Flow: A New Member Is Invited 126
8.6 Example Full Mesh Message Flow: Two New Members Are Invited Simultaneously 127

9.1 Illustration of Triangular Routing in Mobile Networks 144
9.2 UMTS Call Setup Procedure 147
9.3 Registration Procedure for Proposal 1 153
9.4 Call Setup Procedure for Proposal 2 154
9.5 Registration Procedure for Proposal 3 155
9.6 Call Setup Procedure for Proposal 3 156
9.7 Weighted Signalling Load of the Three Proposals: Call Rate and Call / Mobility Ratio Both Vary 159
9.8 Weighted Signalling Load of the Three Proposals: Call / Mobility Ratio Varies . 159

9.9 Line of Intersection: Modified Call Setup = Modified Registration (w_{map} varying) 161
9.10 Line of Intersection: Modified Call Setup = Modified Registration (w_{dns} varying) 161
9.11 Total Weight of Modified Registration 162
9.12 Call Setup Procedure for Proposal 1 — Non-IP-enabled Visited Network 164
9.13 Call Setup Procedure for Proposal 2 — Non-IP-enabled Visited Network 165
9.14 Registration Procedure for Proposal 3 — Non-IP-enabled Visited Network 167
9.15 Call Setup Procedure for Proposal 3 — Non-IP-enabled Visited Network 167
9.16 Weighted Signalling Load of the Three Proposals: Non-IP-enabled Visited Network: Call Rate and Call / Mobility Ratio Both Vary 169
9.17 Weighted Signalling Load of the Three Proposals: Non-IP-enabled Visited Network: Call / Mobility Ratio Varies 169
9.18 Comparison of Modified HLR Signalling Load With and Without IP-enabled Visited Network: Call Rate and Call / Mobility Ratio Both Vary 170
9.19 Comparison of Modified HLR Signalling Load With and Without IP-enabled Visited Network: Call / Mobility Ratio Varies 171

List of Tables

4.1 Standard SIP CGI Metavariables 56

7.1 SIPstone Scores for Servers: Summary 116

8.1 Full Mesh Conference Scenarios Explored with Verifier 136

9.1 Analogous Entities in SIP and UMTS 148
9.2 Message Weights 157
9.3 Mobility Parameters 158
9.4 Protocol Parameters 158
9.5 Weighted Packet Counts for Each Proposal 158
9.6 Weighted Packet Counts for Each Proposal: Non-IP-enabled Visited Network . . 168

10.1 Comparable Components of Internet Telephony and the PSTN 176
10.2 Comparable Addressing Concepts in Internet Telephony and the PSTN 179

Acknowledgments

First and foremost, I acknowledge with great pleasure the contributions of Professor Henning Schulzrinne, without whom this work would not have been possible. Professor Schulzrinne has provided guidance, knowledge, advice, and direction whenever it was needed. He provided opportunities for travel and introductions to researchers in the field, which have proved invaluable.

The work presented in this thesis was supported financially by Lucent Technologies and by SIPQuest, both of which have also provided direction and scope for the work. Lucent Technologies' Bell Laboratories also provided a fertile environment for the development of my work, and I would like to acknowledge the many contributions of its research staff, particularly Thomas F. La Porta, Kazutaka Murakami, and Mehmet Karaul, who contributed to the work interworking Internet Telephony and wireless telephony, and Yow-Jian Lin, who provided helpful discussions and comments on feature interaction.

I have enjoyed the privilege of working with my fellow graduate students, particularly the other members of the Internet Real-Time research group. They have provided spirited discussions, comments, and feedback, both in research group meetings and in one-on-one communications. Members of the group who deserve special credit include Kundan Singh, a co-developer of the CINEMA project; Xiaotao Wu, whose work extending the Call Processing Language helped establish the base language more firmly; and Weibin Zhao, whose work inspired the structure of full mesh conferencing.

In addition, the above lists of research collaborators and fellow graduate students are incomplete without the inclusion of Jonathan Rosenberg, who was both a fellow graduate student and a researcher, first at Lucent Technologies and then at dynamicsoft. His discussions, contributions, and suggestions significantly developed many of the sections of this thesis's work.

Jonathan Rosenberg is also a notable contributor and leader of the Internet Engineering Task Force, in which capacity he made many of his contributions. Many members of IETF's SIP and IPTel working groups were helpful to the work described in the SIP Common Gateway Interface and (especially) the Call Processing Language chapters. Besides Jonathan, members who deserve particular acknowledgment for the latter are Richard Gumpertz, Kenny Hom, and Paul E. Jones.

The LaTeX style file used to format this thesis was written by by Dinesh Das, and modified for Columbia University by Erez Zadok and Shu-Wie F. Chen.

Last but far from least, two people have provided essential encouragement and understanding throughout the progress of my thesis. Gabe Wiener initially encouraged me to go to graduate school, and helped me greatly with my choice of fields and research areas. His untimely death in 1997 was a great loss to all who knew him. More recently, Merav Hoffman has provided encouragement and understanding to help me complete the work of this thesis, and I am profoundly grateful to her.

To Gabe, Merav, and my parents

Chapter 1

Introduction

Internet telephony — voice transmission and call signalling over IP networks — has in recent years become an area of intense development. As IP networks have become ubiquitous, it has become increasingly clear that having separate networks for voice and data communications is unnecessary and redundant. Combining the two systems into one can result in significant simplifications of data networks, from the physical level (only one set of wires needed, rather than two), through having fewer and simpler switches and routers, up to simplifications of software and administrative management.

Circuit-switched telephone networks, however, are complicated systems. As Internet telephony systems have been developed and deployed, experience has revealed many features of circuit-switched networks which the initial versions of IP telephony systems did not completely replicate. Thus, much of the recent development work on these systems, following the initial development of the basic protocols, has been to create systems which allow IP telephony systems to replicate features, architectures, or attributes of circuit-switched networks. For example, recent work has enabled IP telephony systems to offer call transfer services; to allow end systems to communicate with automated systems using Touch-Tone^{TM} (DTMF) tones; and to ensure quality of service and reliability of voice transport over potentially unreliable data networks.

This is all interesting and important work, and essential to ensure that Internet telephony can provide a satisfactory replacement for circuit-switched telephone networks. However, Internet telephony can provide services far beyond those of the circuit-switched network, exactly

because it is on the Internet. The Internet environment is different from telephone networks in two fundamental ways. First of all, it is not limited to a single form of communication. Internet end systems can simultaneously be reading e-mail, browsing web pages, and sending instant messages, as well as communicating by voice. Secondly, the Internet is decentralized. Every end system can (in principle) communicate with every other one; intermediate devices are only necessary if they provide some service to the end systems.

As a consequence, Internet telephony can provide fundamentally new services, over and beyond those which were available in circuit-switched networks. In this thesis, I offer some examples of this — new ways of creating services, and improved architectures for the implementation of existing services, which become possible in the Internet environment.

1.1 Problem Statements and Original Contribution

1.1.1 User-Location Services

User location, or, viewed from another point of view, *user mobility*, is one of the fundamental aspects of Internet telephony. Users can use any end system anywhere in the Internet. In order to receive calls, they *register* the terminal, associating it with their identity, with a server in a well-known location. When the server receives a call setup request, it forwards it to the user's current registered terminals.

Many services can be implemented by modifying the server's user-location process. Many traditional telephony services, such as call blocking, call forwarding, or time-of-day routing, can be generalized as programmatic modifications of the user-location process.

Service authors, therefore, need a way to control this process, in order to tell servers what actions to take. In this thesis, I present two such methods. The first, the *SIP Common Gateway Interface* (SIP CGI), is a low-level server interface which allows fine-grained control of message processing in Session Initiation Protocol (SIP) [1] proxy servers.[1] Based on the well-known HTTP CGI interface, this interface is a straightforward mechanism by which SIP

[1] SIP CGI has been published by the Internet Engineering Task Force as an Informational RFC, RFC 3050 [2], on which I am the primary author, with Jonathan Rosenberg and Henning Schulzrinne. It is also discussed, along with CPL, in a paper in IEEE Network magazine [3], written by Jonathan Rosenberg, myself, and Henning Schulzrinne.

requests and responses can be communicated to external programs, which can then respond with the appropriate actions to take. This interface is designed for use by server administrators and sophisticated, trusted users, as SIP CGI services are implemented as programs that are executed on the proxy server.

"Ordinary" users also would like to be able to control the process by which they are located, and by which their calls are set up. A system which allows services to be created by ordinary users, or third parties on behalf of the users, has some significant design constraints. It must be creatable and modifiable by automated tools, so that ordinary users don't have to engage in complex coding. It must be easily verifiable for correctness, so that users can be assured that they have not, for instance, accidentally made themselves unreachable. Finally, it must be executable in a safe manner — service providers cannot trust users not to write scripts which, accidentally or maliciously, attempt to use excessive resources, attack the servers or other users, or generally attempt to violate the security of the network.

My second method of controlling the user-location process, the *Call Processing Language*, addresses these issues.[2] It is powerful enough to describe a large number of services and features, but it is limited in power so that it can run safely in Internet telephony servers. The has been designed to make it impossible for users to do anything more complex (and dangerous) than describing Internet telephony services. The language is not Turing-complete, and provides no way to write loops or recursion. The CPL is also designed to be easily created and edited by graphical tools. It is based on XML, so parsing it is easy based upon the many XML parsers publicly available. The structure of the language maps closely to its behavior, so an editor can understand any valid script, even ones written by hand. The language is also designed so that a server can easily confirm scripts' validity at the time they are delivered to it, rather that discovering them while a call is being processed.

[2]CPL has been approved as a Proposed Standard RFC by the Internet Engineering Task Force. It has not yet been published, as it is currently blocked on a normative dependence on another specification which has not been finalized, but it has passed the Internet Engineering Standards Group Last Call procedure, the final step in the IETF publication process. In the mean time, it is available as an Internet-Draft [4], of which I am the primary author, with Henning Schulzrinne. A description of the Framework and Requirements for the CPL has been published as an Informational RFC as RFC 2824 [5], also by myself and Henning Schulzrinne. CPL is also discussed in the IEEE Network magazine paper [3], mentioned above. In addition, a number of organizations and companies have implemented the CPL, and several derivative standards are being created based on it.

I have implemented both SIP CGI and the CPL as service environments in our SIP proxy server, which is part of the Columbia InterNet Extensible Multimedia Architecture (CINEMA). In the process, I designed and implemented a general service framework which defines a straightforward and powerful API to allows services to control the proxy server's user-location and call setup procedures, and I developed an event thread architecture that makes implementation of transaction-based protocols efficient and scalable.

1.1.2 Multi-Party Conferencing

Another feature of existing telephone networks is multi-party calls, in which three or more end systems can communicate with each other simultaneously. The basic SIP protocol, however, is only engineered for point-to-point communications, and does not, inherently, provide any support for communications among more than two parties, other than loosely-controlled multicast conferences in which the users' media is sent to a multicast group. More tightly-controlled conferencing is useful and necessary in a number of circumstances — from simple three-way calling, in which two people on an ordinary call decide to add a third, to large-scale conference calls.

There are a number of ways to provide conferencing with existing SIP mechanisms. It is possible, for instance, to set up a centralized conferencing server which mixes communications from many endpoints; it is also possible for end systems to act as bridges themselves. However, all these mechanisms have shortcomings. I have proposed a new approach, giving a fully-distributed, decentralized protocol for conferencing which establishes a fully connected mesh of signalling and media connections between the conference participants.[3]

This approach is not intended to replace the other solutions, but rather to complement them. The existing solutions are designed for certain problem domains, and are useful in those domains; however, they are over-engineered or architecturally inappropriate in some common scenarios. The new proposal addresses these scenarios.

This conferencing approach is also applicable to additional environments. Numerous scenarios require multiple networked devices to be able to communicate with each other without a single point of failure, and the topology of a full mesh is often very useful for robustness. Such

[3] A paper on this work, which I wrote with Henning Schulzrinne [6], was presented at the 2003 ACM conference on Network and Operating System Support for Digital Audio and Video.

topologies often need to be dynamically assembled, with end systems entering or leaving the group as they become available and unavailable. Thus, the mechanism described in this paper is also useful for such environments as text messaging, highly-reliable alerting or event systems, establishing router peering relationships, distributed simulation, or clusters of network servers which need to share state information.

1.1.3 Internet Telephony Interaction with Circuit-Switched Mobile Networks

User location and user mobility are not a new invention of Internet telephony. Some circuit switched networks use them — most notably, circuit-switched wireless ("cellular") networks.

Internet telephony needs to be able to connect to standard mobile telephony networks. As both mobile and Internet telephony are already designed to interconnect with the Public Switched Telephone Network (PSTN), the easiest way to interconnect them would be simply to use the PSTN as an intermediate link. This is, however, inefficient and suboptimal, as compared to connecting the networks by interworking the protocols directly.

I have created a family of system architectures which allow SIP and UMTS networks to be connected directly, allowing traffic to flow directly to the mobile switching center handling a user's mobile terminal.[4] These architectures allow voice traffic to flow efficiently, eliminating triangular routing, transcoding, and other inefficiencies of the direct connections. After designing the three approaches, I have also analyzed them to consider their relative merits in terms of signalling load and delay.

1.1.4 Feature Interaction

Finally, whenever services are defined, the issue arises of *feature interaction*. A well-known problem from traditional telephone networks, feature interaction occurs when several features or services interact in unexpected and potentially undesirable ways. It can arise whenever two features or services are able to affect the same call or call environment. While many basic feature interaction problems from circuit switched networks remain, Internet telephony adds additional

[4] This work was published in Computer Communications Review [7], in a paper by myself, Kazutaka Murakami, Mehmet Karaul, and Thomas F. La Porta.

complications. These complications arise since functionality tends to be more distributed, users can program the behavior of end systems and signalling systems, the distinction between end systems and network equipment largely vanishes and the trust model implicit in the PSTN architecture no longer holds. On the other hand, Internet telephony makes end point addresses plentiful and its signalling makes it easy to specify in detail the desired network behavior.

Accordingly, I have analyzed the nature of feature interactions in Internet telephony and the ways in which they differ from those of traditional telephony.[5] I explain how the differences between traditional telephone networks and Internet telephony alter the feature interaction problem, discuss the applicability of existing resolution techniques, and present some new approaches for resolving interactions in the Internet environment.

1.2 Overview of Thesis

This thesis is organized as follows. After this introduction, it begins with background information in Chapter 2 and an overview of related work in Chapter 3. Following this, I present descriptions of the various aspects of user-location services. SIP CGI is discussed in Chapter 4, and Chapter 5 discusses the Call Processing Language. Chapters 6 and 7 describe the implementation of both of these service environments in the CINEMA SIP server: Chapter 6 describes the policy framework, atop which user-location services are implemented, and Chapter 7 describes improvements to the server's event and thread model.

The protocol for decentralized multi-party conferencing is explained in Chapter 8. Following this, I present the techniques for interconnecting SIP with UMTS in Chapter 9. Feature interactions are considered in Chapter 10.

I finish with some general conclusions and observations in Chapter 11.

[5] A paper on this work, which I wrote with Henning Schulzrinne [8], was presented at the Sixth International Workshop on Feature Interaction.

Chapter 2

Internet Telephony and User Location Services: Background

This chapter presents some background on telephony services, describes the architecture of Internet telephony, and gives some examples of advanced services that can be enabled using it. It then specifically describes user location services, the type of services which are discussed in this part of the thesis, and discusses some architectural considerations which arise when creating these services, and some requirements for an feature environment aimed at the end-user. It also gives some background to the Session Initiation Protocol (SIP), atop which the service creation environments described in this part of the thesis are implemented.

2.1 Overview of Telephony Services

Internet telephony enables a wealth of new service possibilities [9, 10, 11]. Traditional telephony services, such as call forwarding, call transfer, and 800-number services, can be enhanced by adding integration with e-mail, web, presence, instant messaging and directory services. Users can have calls redirected to web pages, use streaming media tools to record voicemail, or use instant messages in place of call waiting notifications or call completion. IP telephony can offer improved speech quality through advanced speech and audio codecs. Communications can encompass not just voice, but video, application sharing, and even virtual reality. Much more

powerful user interfaces can allow these services to be made easily accessible. Gateways to the Public Switched Telephone Network (PSTN) and to other telecommunications networks can allow these services to extend to traditional landline phones, cellular phones, and pagers.

With such a wide range of services possible, it becomes critical to provide means by which these services can be rapidly conceived, developed, and deployed. It should not be necessary to add new network elements for each new service, nor should it be necessary to reinvent the interfaces to existing elements for each new service. In addition, it should be possible for third parties to create new services easily. By third parties, we mean individuals or organizations besides the ones that own the routers, hubs, switches, and servers which actually implement the service. This separation allows end users the ultimate in flexibility. They can purchase certain services from one provider, and other services from other providers. It also opens new markets for service providers.

This notion of service programmability has been a goal for the telephone network for some time [12], but it has never fully materialized. The Internet provides a new opportunity to realize this goal. Its end-to-end architecture, which allows any host to send any data to any other host, is an ideal platform on which to achieve this service model. Servers which provide the program code for the service can be located in completely separate domains from the servers which implement it. This means they can be owned and run by different providers.

2.1.1 Definition of Telephony Services

All telephony — traditional and Internet — assumes at its root a *basic call model*. In this model, calls are placed from a single, fixed source to a single, fixed destination. The destination is entirely determined by the address identifier specified by the caller. Once the call is placed, the destination is alerted, and the destination can choose either to answer the call, or not. If the call is answered by the destination, the call continues until one side or the other terminates it; if the call is not answered, the caller receives an indication of this and aborts the call attempt.

Telephony *services* are those aspects of a telephone system which go beyond this basic call model. The model can be extended in any number of ways. A call may, for example, involve multiple participants. A user can choose to have calls automatically directed to them regardless

of their network or physical location, or choose who is allowed to reach them, and when. A caller may choose to have special, custom codes ("speed-dial") for commonly-dialed destinations. Busy or unanswered calls can be directed to alternate destinations or voicemail systems. An ongoing call can be transfered to another destination, and an alert signal indicating a new call can interrupt the media of an existing one. Any number of other services have been invented in existing and proposed telephony networks [13].

There are two goals for services for Internet telephony. First of all, as Internet telephony is intended to be a replacement for existing telephone networks, it must be possible to create comparable services to those of these traditional networks. Furthermore, we want to take advantage of the possibilities the Internet introduces to allow additional services. Because of the richness and flexibility of the Internet, we can introduce new types of services, and bring sophisticated service programming to the end-user. These new services are the topic of this thesis.

2.2 Internet Telephony Architecture

The discussion of these new Internet telephony services and service creation methods must begin with an overview of how Internet telephony works. To this end, this section presents a generalized model of an Internet telephony network. While the details of various protocols differ, on an abstract level all major Internet telephony architectures are sufficiently similar that their major features can be described uniformly.

2.2.1 Model components

An Internet telephony network contains two types of components: *end systems* and *signalling servers*. Roughly speaking, these are analogous to end hosts and routers, respectively, at layer 3 (the network layer) of an IP network.

End systems

End systems are devices which originate and/or receive signalling information and media. These include simple and complex telephone devices, conferencing servers, PC telephony clients, and

automated voice response systems. An end system can originate a call; and it can accept, reject, or forward incoming calls. The details of this process (ringing, multi-line telephones, and so forth) are not important for this model of telephony.

For the purposes of this model, gateways — for example, a device which connects calls between an IP telephony network and the PSTN — are also considered to be end systems. Other devices, such as media translators or firewalls, are not directly dealt with by our service creation methods, though they can be (and in some cases have been) adapted to use them.

Signalling servers

Signalling servers are devices which relay or control signalling information. In SIP [1], they are proxy servers, redirect servers, or registrars; in H.323 [14], they are gatekeepers.

The most significant type of signalling processing performed by signalling servers is the handling of call setup requests. Signalling servers can perform three types of actions on these requests. They can

proxy them: forward them on to one or more end systems or other signalling servers, returning one of the responses received;

redirect them: return a response informing the sending system of a different address to which it should send the request; or

reject them: inform the sending system that the setup request could not be completed.

End systems can also reject and redirect call setup requests. Some illustrations of proxying and redirection by signalling servers (in the specific case of SIP) are given in Section 2.7.

Signalling servers also normally maintain information about user location. Whether by means of registrations (SIP REGISTER or H.323 RAS messages), static configuration, or dynamic searches, signalling servers must have some means by which they can determine what destinations are currently associated with a user, in order to make intelligent choices about their proxying or redirection behavior.

Signalling servers are usually general-purpose, Internet-located computers. As such, telephony services can take advantage of all the functionality that such devices are capable of, for

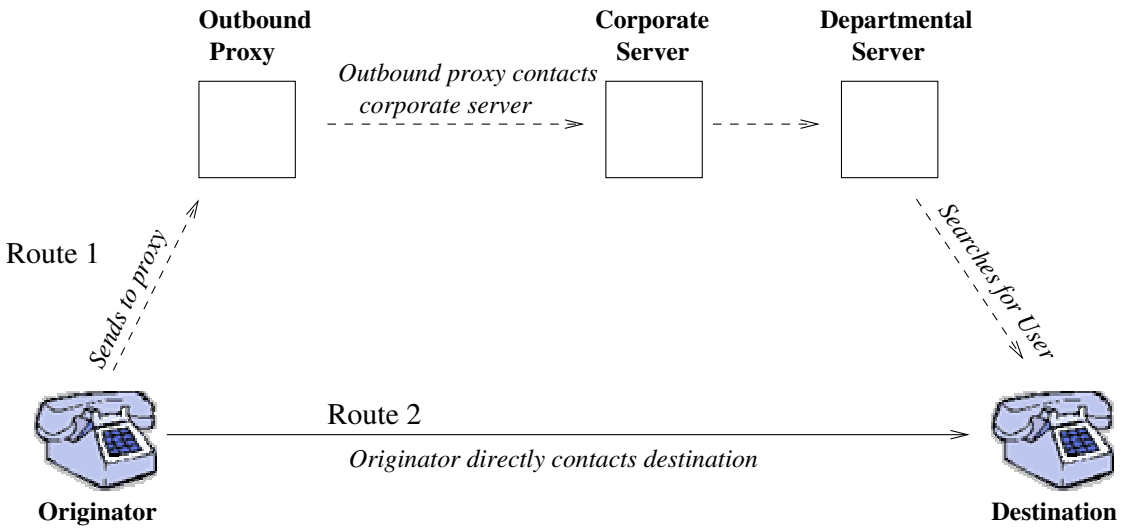

Figure 2.1: Possible Paths of Call Setup Messages

example to store information for later retrieval, or to initiate or participate in other Internet-based activities.

2.2.2 Interactions of Model Components

A call is established when a user uses an end system to contact another user in the network. When this end system places a call, the call establishment request can proceed by a variety of routes through components of the network. To begin with, the originating end system must decide where to send its requests. There are two possibilities here: the originator may be configured so that all its requests go to a single local server; or it may resolve the destination address to locate a remote signalling server or end system to which it can send the request directly.

Once the request arrives at a signalling server, that server uses its user location database, its local policy, DNS resolution, or other methods, to determine the next signalling server or end system to which the request should be sent. A request may pass through any number of signalling servers: from zero (in the case when end systems communicate directly) to, in principle, every server on the network. What's more, any end system or signalling server can (in principle) receive requests from or send them to any other.

Figure 2.1 illustrates this. In the figure, there are two paths the call setup request may

take. For Route 1, the end system initiating the call (the **Originator**) knows only a location-independent address for the user it is trying to contact, and it is configured to send outgoing calls through a local proxy (the **Outbound Proxy**). Therefore, it forwards the call setup request to this proxy server, which finds the server of record for that address, and forwards it on to that server.

In this case, the organization the destination user belongs to uses a multi-stage setup to find users. The **Corporate Server** identifies which department a user is part of, then forwards the request to the appropriate **Departmental Server**, which actually locates the user. (This is similar to the way e-mail forwarding is often configured.) The response to the request will travel back along the same path.

By contrast, Route 2 illustrates the case in which the originator knows a specific device address to contact, and it is not configured to use a local outbound proxy. In this case, the originator can directly contact the destination without having to communicate with any network servers at all.

As this shows, in Internet telephony signalling servers cannot in general know the state of end systems they "control," since signalling information may have bypassed them. This architectural limitation implies a number of restrictions on how telephony services can be implemented through the user location process. For instance, a signalling server cannot reliably know if an end system is currently busy or not; a call may have been placed to the end system without traversing that signalling server. Thus, signalling messages must explicitly travel to end systems to find out their state; in the example, the end system must explicitly return a "busy" indication.

2.3 User Location Services

The first part of this thesis specifically addresses *user-location services*. These services are those which can be implemented by controlling the process by which the destination of a call — the user — is located. As described in Section 2.2.1 above, signalling servers route call setup requests across the network. Many services can be implemented by modifying this process.

2.3.1 Example User-Location Services

This section gives some specific examples of telephony services which can be implemented in a signalling server through modifications of the user location process. Note that some of these examples are deliberately somewhat complicated, so as to demonstrate the level of decision logic that should be possible.

Call forward on busy/no answer: When a new call comes in, the call should ring at the user's desk telephone. If it is busy, the call should always be redirected to the user's voicemail box. If, instead, there's no answer after four rings, it should also be redirected to his or her voicemail, unless it's from a supervisor, in which case it should be proxied to the user's cell phone if it is currently registered.

Information address: A company advertises a general "information" address for prospective customers. When a call comes in to this address, if it's currently working hours, the caller should be given a list of the people currently willing to accept general information calls. If it's outside of working hours, the caller should get a webpage indicating what times they can call.

Intelligent user location: When a call comes in, the list of locations where the user has registered should be consulted. Depending on the type of call (work, personal, etc.), the call should ring at an appropriate subset of the registered locations, depending on information in the registrations. If the user picks up from more than one station, the pick-ups should be reported back separately to the calling party.

Intelligent user location with media knowledge: When a call comes in, the call should be proxied to the station the user has registered from whose media capabilities best match those specified in the call request. If the user does not pick up from that station within four rings, the call should be proxied to the other stations from which he or she has registered, sequentially, in order of decreasing closeness of match.

Client billing allocation — lawyer's office: When a call comes in, the calling address is correlated with the corresponding client, and client's name, address, and the time of the call is

logged. If no corresponding client is found, the call is forwarded to the lawyer's secretary.

2.3.2 Non-User-Location Services

For contrast, this section gives some examples of common telephony services which are *not* user location services, and are not addressed by the first part of this thesis.

Call waiting is a service in which a user who is engaged in a call can be alerted — usually by means of an audible tone — when another call request arrives. This is not a user-location service for several reasons. First of all, user notification is involved; user-location services do not have any means of creating a tone or other user-interface indication, either by inserting media into the existing call's media stream or by directly controlling the end system's user interface. Second of all, user-location services cannot tell, on their own, when an end system is busy: information about end system states is not kept in user-location servers, and calls might arrive at end systems through any of a number of routes through the network.

Caller ID notifies a user of the identity of a caller before the user decides whether to accept the call. Whether a telephony protocol carries caller information is part of the protocol; a user-location service cannot add it if it is not already present. (Caller ID, as a service, usually means encoding caller information so that it can be carried over an analog POTS line. This is not a level of control that user-location services can effect.)

Call return allows a user who has missed a call attempt to automatically call back the number of the most recent caller. This requires two attributes: caller ID information, as described in the previous paragraph, and some way of triggering the returned call. Since network servers can't, in general, intercept all such calls in Internet telephony, this must be a user-interface feature at the end system.

Customized ringing allows distinctive user notification (specifically, alert sounds) based on the identity of the caller or other attributes of the call. This is also an end-system user-interface feature.

Music-on-hold causes music to be played when the other party in a call is placed on hold. This is generated entirely by end systems.

Three way calling (also known as **conference calling**) allows calls between more than two people. User-location services cannot accomplish this; it is a significant departure from the standard call model, of which at least one end system in the call must be aware. Multi-party calls are discussed in detail in Chapter 8.

Multimedia calls allow media beyond the standard voice-grade audio. Any end system which supports this must, of course, be able to send and/or receive these media. Most telephony signalling systems which support multimedia separate (at least on a logical level) the description of supported media, and of course the actual transport of these media, from the signalling involved in user location. Whether a caller, or a callee's end systems, *support* multimedia calls can, of course, be an input into the user-location process, however.

Charging and billing describes a number of methods by which it can be determined who pays for a call. Payment is independent of user location, as defined by this chapter; thus, user-location servers cannot affect it.

User prompting allows users to be interactively queried as to the handling of their calls. This must be done by end systems, as user-location servers have no way of sending the media to query the users.

2.4 Service Creation and Execution

The first part of this thesis specifically addresses *programmable* user-location services. These are services which are not intrinsically part of the basic behavior of a signalling server. Rather, these services can be created separately from the signalling server, and then the signalling server can be configured to use the service. This section describes this process in more detail.

These separately-created services are described by *service descriptions* or, equivalently, *service logic*: detailed programmatic descriptions of the service. These service descriptions are created by the user or programmer using a *service creation environment*, and are executed within

a *service execution environment*. The service execution environment defines how services are executed, and establishes how service descriptions are interpreted. A signalling server which allows services to be executed is known as a *service executor*.

2.4.1 What a Service Execution Environment Does

The service execution environments described in the first part of this thesis run in a signalling server, and control that system's proxy, redirect, or rejection actions for the set-up of a particular call. They do not attempt to co-ordinate the behavior of multiple signalling servers, or to describe features on a "Global Functional Plane" as in the Intelligent Network architecture [13].

More specifically, they replace the user location functionality of a signalling server. As described in section 2.2.1, a signalling server typically maintains a database of locations where a user can be reached; it makes its proxy, redirect, and rejection decisions based on the contents of that database. The execution environments replace this basic database lookup functionality; they take the registration information, the specifics of a call request, and other external information they want to reference, and choose the signalling actions to perform.

2.4.2 Which Service is Executed

Services are usually associated with a particular Internet telephony address. When a call establishment request arrives at a signalling server which is a service executor, that server associates the source and destination addresses specified in the request with its database of service descriptions; if one matches, the corresponding service is executed.

Once the service has executed, if it has chosen to perform a proxy action, one or more new Internet telephony addresses will result as the destination of that proxying. Once this has occurred, the server again checks its database of services to see if any of them are associated with any of the new addresses; if one is, that service as well is executed, recursively (assuming that a service has not attempted to proxy to an address which the server has already tried). If there are services associated with both the originating and destination addresses for a call, the server handles the former first, and then the latter as appropriate.

In general, in an Internet telephony network, most addresses will denote one of two

things: either a user or a device. A *user address* refers to a particular individual, for example `sip:joe@example.com`, regardless of where that user actually is or what kind of device he or she is using. A *device address*, by contrast, refers to a particular physical device, such as `sip:x26063@phones.example.com` or `sip:128.59.19.39`. The type of address used affects how calls are routed to it, as described in Section 2.2.2. Other sorts of addresses are also possible. They may refer to a service or a role, for example `sip:sales@example.com`. Or they can refer to more than one device but not a location-independent user: an address could, for instance, refer to "my mobile devices, wherever they currently happen to be registered." These have some use, but we expect them to be less common; this sort of functionality is better described by callee capability descriptions, such as those provided by [15]. The services described in this thesis are agnostic to the type of address they are associated with; while services associated with user addresses are probably the most useful, there is no reason that one could not be associated with any other type of address as well. The recursion process described above allows services to be associated with several of a user's addresses; thus, a user service could specify an action "try me at my cell phone," whereas a device service could say "I don't want to accept cell phone calls while I'm out of my home area."

It is also possible for a service to be associated not with one specific Internet telephony address, but rather with all addresses handled by a signalling server, or a large set of them. For instance, an administrator might configure a system to prevent calls from or to a list of banned incoming or outgoing addresses; these should presumably be configured for everyone, but users should still to be able to have their own custom scripts as well. Exactly when such scripts should be executed in the recursion process depends on the precise nature of the administrative script. This is discussed further in Section 2.6.

2.4.3 Where a Service Executes

Users can have custom services on any network server which their call establishment requests pass through and with which they have a trust relationship. For instance, in the example in Fig. 2.1, the originating user could have custom services on the outbound proxy, and the destination user could have them on both the corporate server and the departmental server. These

services would typically perform different functions, related to the role of the server on which they reside; a service on the corporate-wide server could be used to customize which department the user wishes to be found at, for instance, whereas a service at the departmental server could be used for more fine-grained location customization. Some services, such as filtering out unwanted calls, could be located at either server, assuming that these calls have no way of bypassing either server.

2.5 Creation and Transport of a Service Description

Users create service descriptions, typically on end devices, and transmit them through the network to signalling servers. Scripts persist in signalling servers until changed or deleted, unless they are specifically given an expiration time; a network system which supports user-based scripting will need stable storage. (Soft-state refresh of services has been considered, but is generally not useful; many services, such as call-forward-not-available, need to be executed even when the service owner is unavailable.)

The end device on which the user creates the service description need not bear any relationship to the end devices to which calls are actually placed. For example, a service might be created on a PC, whereas calls might be intended to be received on a simple audio-only telephone. Indeed, the device on which the service description is created may not be an "end device" in the sense described in Section 2.2.1 at all; for instance, a user could create and upload it from a non-multimedia-capable web terminal.

The service description also might not necessarily be created on a device near either the end device or the signalling server in network terms. For example, a user might decide to forward his or her calls to a remote location only after arriving at that location.

If a user has trust relationships with multiple signalling servers (as discussed in Section 2.4.3), the user may choose to upload services to any or all of those servers. These services can be entirely independent.

2.6 Properties of a Service Creation Mechanism

The key to programming Internet telephony services is to add logic that guides behavior at each of the elements in the system. For user location services, this logic dictates where the requests are proxied to, how protocol packets are formatted, and how the results are processed. For example, a simple service, such as call forwarding based on time of day, would require logic in the signalling server to obtain the time when a call setup arrives, and based on it, proxy the request to the appropriate destination. In general, the logic can direct the server's actions based on any sort of input — for instance, the time of day, the caller, call subject, session type, media composition, data obtained from a web page, or data obtained from an external directory. The logic may also instruct the server to generate new requests or responses.

Logic can also be added to user agents. However, since user agents are usually owned by end users, not network service providers, providing logic for them is a different problem [16]. The breadth of platforms used, the security implications, and the trust models, are substantially different. For this reason, the first part of this thesis considers only services implemented in network servers.

The basic model for providing logic for user location services is that a signalling server is augmented with *service logic*, which is a program that is responsible for creating the services. An interface exists between the two. When requests and responses arrive, the server passes information "up" to the service logic. The service logic makes some decisions based on this information, and other information it gathers from different resources, and passes instructions back "down" to the server. The server then executes these instructions.

In order to define the details of this model, a number of issues must be resolved. These are:

- Where does the logic reside?
- When does the logic execute?
- What are the restrictions on the resources available to the program?
- What information about the protocol messages are provided to the program?

- What level of control does the program have over the server's execution?

There is no one solution for each of these issues. In particular, the solution for the last three issues depends very much on the level of trust between the server and the program. If the level of trust is low, very specific, structured information should be passed from the server to the program, and a very narrowly defined set of controls should be exposed to the program from the server. This restricts the set of services that can be defined, but provides a greater level of security. The server can be sure that the program cannot perform malicious operations, or cause the server itself to crash. On the other hand, if the level of trust is high, a large amount of information should be given, and the maximum amount of controls exposed. This broadens the set of services than can be developed.

The amount of trust depends on who is creating the service logic. This can be the owner of the server, some third party service provider, or even an end user. In the former two cases, the service logic is created by a network administrator, who has access to testing facilities, and can verify the service logic before it is placed in the network. In the latter case, the logic is created by a regular end user. There may be thousands or millions of end users. A network administrator cannot possibly test each piece of logic out ahead of time, under all conditions; nor can the network administrator trust that end users will provide bug free, non-malicious code. These differing trust models demand different solutions.

2.6.1 Program Invocation Times

Not all services require the service logic to be consulted for every event or message that is received. A large class of services require the logic to be executed only when the initial call setup message is received. Subsequent message processing rules can follow standard procedures defined by the protocol itself. Furthermore, some calls won't require any services at all. The signalling server should behave as it normally would, and not consult the service logic at any point. It is therefore necessary to have some means to specify at what point service logic is executed. The execution points can be defined by some administratively set policy, or they can be controlled dynamically by the service logic itself.

A related issue is whether the service logic is persistent or not. If the service logic runs

as a separate process, it can remain active for the duration of the call (and beyond), and therefore be persistent. This would mandate an asynchronous interface between the logic and the server. It also introduces cleanup issues. Protocol or server errors may cause the service logic process for a particular call to remain active long after the call is over. Some means for cleanup is then needed to destroy these old processes. The advantage is that the service logic can pass control instructions back to the server at any time, rather than depending on the server to execute the service logic only on specific events. This enables numerous services (such as the click-to-dial service defined in [17]) which would otherwise be difficult to support.

As an alternative, the service logic can be executed synchronously. When the server receives a message, it begins execution of the service logic. The logic passes the control information back to the server, and ceases execution. This is the model most commonly used for web services and other programmable web pages, as discussed in Chapter 3. This is most easily accomplished by having the service logic executed as a function call from the server. However, the service logic can also be executed as a separate process, but terminate once the control information is passed back to the server.

2.6.2 Resource Restrictions

The service logic can have access to a large number of resources. On the Internet, this includes name services (i.e., the Domain Name Service (DNS)), web pages, directories, mail servers, media servers, QoS controls, policy repositories, presence systems, and instant messaging services, to name a few. The logic can also have access to resources on other networks, such as the telephone network. The ability to query 800 number databases, for example, would allow migration of freephone services to the Internet.

With a breadth of resources comes a wide range of failure modes. The likelihood of bugs increases. The ability of a program to perform malicious actions increases. The possibilities for unusual cases or untested results increases. The right operating point, as we have indicated above, depends on the level of trust between the server and the logic. For end user defined services, access to resources should be restricted. For administrator-defined services, they should be more flexible.

2.6.3 Interface

The servers need to pass information about the call transaction, including message information and call states, to the service logic. This information can range from very abbreviated to very verbose. In the abbreviated case, only the message types (for instance, for SIP, whether it's a INVITE, ACK or BYE message for requests, or the response code for responses – see Section 2.7) and current state might be passed. In the verbose case, the entire protocol message might be passed.

The right operating point, once again, depends on the level of trust and desired amount of flexibility. Verbosity aids flexibility, but increases complexity and the margin for error.

Information must also be passed from the service logic back to the server. This information is control data, instructing the server what to do next. This can also range from simple (a list of URIs to proxy to), to complex (an entire message to be sent).

2.7 The Session Initiation Protocol

The Internet telephony services described in this thesis have been implemented atop the Session Initiation Protocol, SIP [1], as a platform for programming telephony. SIP is suitable for service development for a number of reasons. Its clean request-response model is amenable to simple programming. Its textual formatting and simple header structure makes it easy to use text processing languages, such as Perl, and textual interfaces, such as CGI, for developing services. The control it provides in the amount of state maintained at a server is very useful for service programming. Finally, its ability to work in a fully distributed fashion, avoiding routing loops and maintaining consistent behavior across servers, helps in avoiding feature interactions when programming services. A more detailed description of SIP can be found in [18]. We briefly overview it here, touching on the features relevant for our discussions.

SIP is a client-server protocol, similar in both syntax and semantics to the HyperText Transfer Protocol (HTTP) [19]. However, it defines its own methods and headers for providing the functions required in IP telephony signaling. Requests are generated by one entity (the client), and sent to a receiving entity (the server) that processes them, and then sends responses. A

request and the responses which follow it are called a *transaction*. As a call participant may either generate or receive requests, SIP-enabled end systems include a protocol client and server (a *user agent client* and *user agent server* respectively). The user agent server responds to the requests based on human interaction or some other kind of input. Furthermore, SIP requests can traverse many *proxy servers*, each of which receives a request and forwards it towards a next hop server, which may be another proxy server or the final user agent server. As a result, proxy servers are primarily responsible for *call routing*. A server can make use of any means at its disposal to determine where to route a call, including database queries or local program execution.

A server may also act as a *redirect server*, informing the client of the address of the next hop server, so that the client can contact it directly. There is no protocol distinction between a proxy server, redirect server, and user agent server; a client or proxy server has no way of knowing which it is communicating with. The distinction lies only in function. A proxy or redirect server cannot accept or reject a request, whereas a user agent server can. This is similar to the HTTP model of clients, origin and proxy servers. A single host may well act as client and server for the same request.

As in HTTP, the client requests invoke *methods* on the server. Requests and responses are textual, and contain header fields which convey call properties and service information. SIP reuses many header fields used in HTTP, such as the entity headers (e.g., Content-Type) and authentication headers.

SIP defines several methods. INVITE invites a user to a call. BYE terminates a connection between two users in a call. OPTIONS solicits information about capabilities, but does not set up a connection. ACK is used for reliable message exchanges for invitations. CANCEL terminates a search for a user. REGISTER conveys information about a user's location to a SIP server. Many SIP extensions have been written; some of these define other methods as well.

A call is set up by issuing an INVITE request. This request contains header fields used to convey information about the call. Examples of some of the header fields are To, which lists the callee, From which lists the caller, Subject, which identifies the subject of the call, Call-ID which contains a unique call identifier, Contact which lists addresses where a user can be contacted, and Require, which allows for feature negotiation.

At any time after the call is set up, either party may send a new INVITE request to the other to change parameters of the call. This includes changing media codecs, adding parties to the call, or changing addresses (for mobility support). Once the call is over, either side may hang up by sending a BYE request to the other.

Like HTTP, SIP requests and responses carry bodies, which can be any defined MIME [20] type. The body conveys information about the session between the parties. Normally, the Session Description Protocol (SDP) [21] is used. SDP describes multimedia sessions, including codec types, port numbers, addresses, and session start and stop times.

An example SIP message with an SDP payload is shown in Figure 2.7. The SIP message is an INVITE request, from Alice to Bob. The message contains a call identifier in the Call-ID field, and a sequence number in the CSeq field. The body is of type `application/sdp`. It describes a multicast session on "Mbone Engineering Issues", on multicast group 224.2.0.1. The `m=audio 3456 RTP/AVP 0` line indicates the session is audio only, using codec 0 (which is PCM μ-law) on UDP port 3456.

SIP runs on top of UDP, TCP or SCTP, providing its own reliability mechanisms when used with UDP. For addressing, SIP makes use of uniform resource identifiers (URIs) [22], which are generalizations of Uniform Resource Locators (URLs), in common usage in the web. SIP defines its own URI, but its header fields can carry other URIs, such as http, mailto [23], or tel [24] URLs.

Like HTTP, a proxy or redirect server can destroy all transaction state after the transaction is complete. SIP transactions can complete even if a server crashes and reboots in the middle, losing all transaction state. SIP messages contain sufficient information to allow a rebooted server to treat the message correctly. This also means a server can safely clean up old state which has collected to due unusual failures or cases where the caller lets the phone ring for a long time. In addition, SIP allows subsequent transactions (such as a call termination, or feature invocation) to occur directly between the caller and callee without traversing through the same proxy or redirect servers. However, a proxy can insist on being in the signaling path for subsequent transactions for the same call through means of the Route and Record-Route header fields.

A typical SIP transaction is depicted in Figure 2.3. The caller creates an INVITE re-

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKnashds8
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/sdp
Content-Length: 187

v=0
o=user1 53655765 2353687637 IN IP4 128.3.4.5
s=Mbone Audio
i=Discussion of Mbone Engineering Issues
e=mbone@somewhere.com
c=IN IP4 224.2.0.1/127
t=0 0
m=audio 3456 RTP/AVP 0
```

Figure 2.2: Example SIP Message with SDP

quest for some user, `sip:joe@company.com`. This request is forwarded to a local proxy (1). This proxy looks up `company.com` in DNS, and obtains the IP address of a server handling SIP requests for this domain. It then proxies the request to this server (2). The server for `company.com` knows about the user `joe`, but this user is currently logged in as `j.user@university.edu`. The server at `example.com` can know this through a static configuration, database entry, or a dynamic binding set up by the user using a SIP REGISTER message. So, the server redirects the proxy (3) to try this address. The local proxy looks up `university.edu` in DNS, and obtains the IP address of its SIP server. The request is then proxied there (4). The university server consults a local database (5), which indicates (6) that `j.user@university.edu` is known locally as `j.smith@cs.university.edu`. So, the main university server proxies the request to the computer science server (7). This server knows the IP address where the user is currently logged in, so it proxies the request there (8). The user accepts the call, and the response is returned through the proxy chain (9),(10),(11),(12)

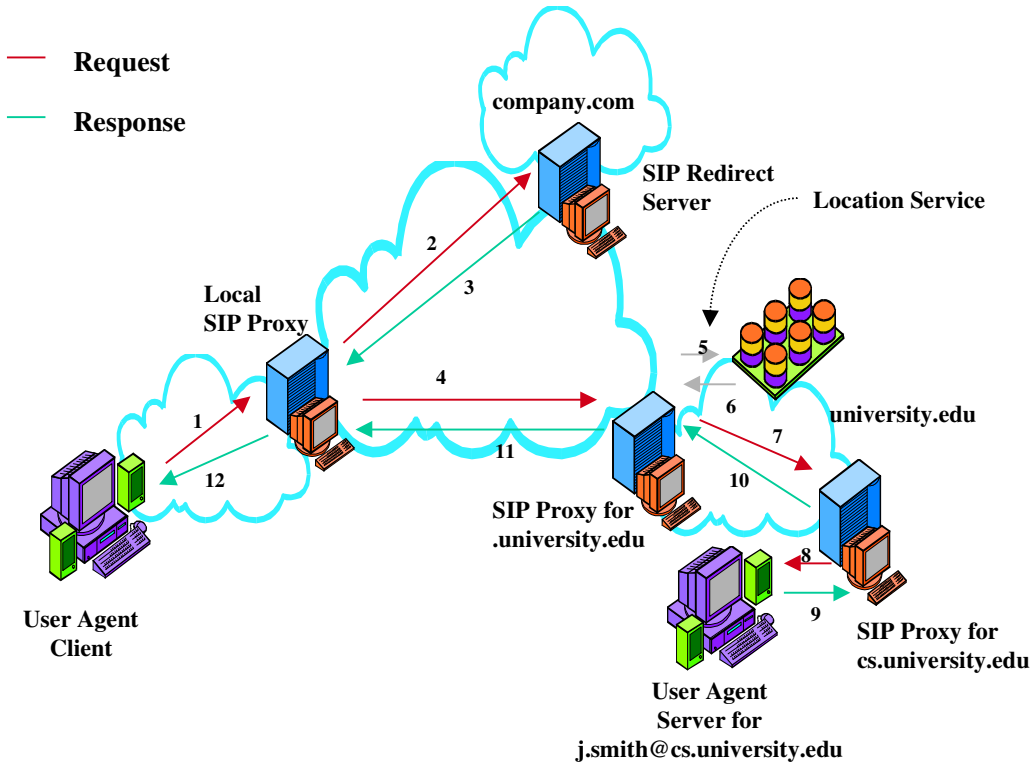

Figure 2.3: SIP Operation

to the caller. This process is similar to, and inspired by, the way that uniform resource locators (URLs) are resolved in the web, and the way that Internet e-mail is routed.

2.8 Conclusion

As mentioned in Section 2.6, there are a number of ways in which different types of service execution environments have different requirements. Following a discussion of related work in Chapter 3, the next two chapters address this by presenting two quite different such environments.

Chapter 4 presents the *SIP Common Gateway Interface*, a low-level service programming interface for administrators and trusted service creators. SIP CGI, based upon the popular Common Gateway Interface used to program services in the World Wide Web, is tightly coupled to

SIP, much as the original CGI is tightly coupled to HTTP. SIP-CGI is independent of any programming language, and offers service creators full access to the capabilities of the server on which it is run.

Chapter 5, in contrast, presents a high-level, restricted language intended for users: the *Call Processing Language*. It is simple, extensible, easily edited by graphical clients, and largely independent of underlying signalling protocols. It is suitable for running on a server where users may not be allowed to execute arbitrary programs, as it has no variables, loops, or ability to execute external programs.

Chapter 3

User-Location Services: Related Work

User-location services were described in Chapter 2. There has been a good deal of work on creating these services in a variety of environments. There has been much work in the PSTN, and in non-telephony-related Internet services. There are also several techniques that provide Application Programming Interfaces to control both traditional and Internet telephony services. This chapter reviews a number of mechanisms for service creation: in the traditional telephone network, the world wide web, e-mail, and Internet telephony. This review prepares for the mechanisms which will be described in Chapters 4 and 5.

3.1 Intelligent Network Services

Traditional telephone networks have been developing advanced services since the development of sophisticated digital telephone switches in the 1970s. Much of the early work was both proprietary to specific vendors, and tightly integrated with those vendors' software. In the 1980s, PSTN telephony services were standardized with the development of the *Intelligent Network* [25, 26].

Today, a number of PSTN telephony services are commonly offered. The Custom Local Area Signalling Services (CLASS) provide a set of services to consumers: call forwarding, three-way calling, call waiting, speed dial, automatic callback, call return, distinctive ringing, various simple user-location services, and voicemail. Similar sets of services are provided to institutional customers through their Private Branch Exchanges (PBXes).

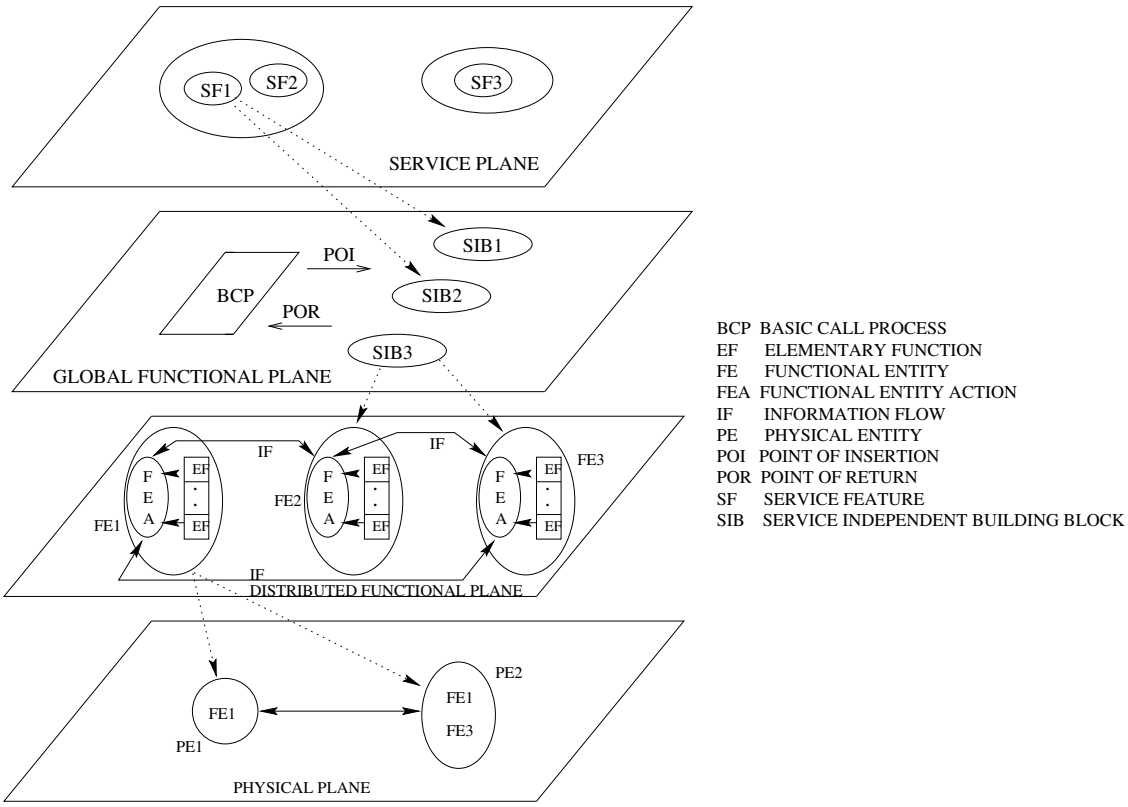

Figure 3.1: IN Conceptual Model, from Q.1201

The Intelligent Network is a collection of standards codified by the International Telecommunications Union, Telecommunications Sector (ITU-T). It is first introduced in Recommendation Q.1201 [12], which provides the model for IN, duplicated in Figure 3.1.

The figure shows that the IN can be modeled as a number of distinct planes. The first is the Service Plane, which contains services (such as call-forward and freephone). The Global Functional Plane (GFP) defines these services in terms of atomic operations, called Service-Independent Building Blocks (SIBs). A SIB is a specific, small unit of functionality, with simple inputs and outputs, and can be thought of as a "network level" machine instruction inside a computer. It defines a basic function which analyzes information about the call, or manipulates the call's state. Some of the SIBs are *Compare* for comparing two values, *Queue* for queuing a call, and *Screen* for rejecting calls. Services are assembled out of SIBs by combining them in a directed acyclic graph.

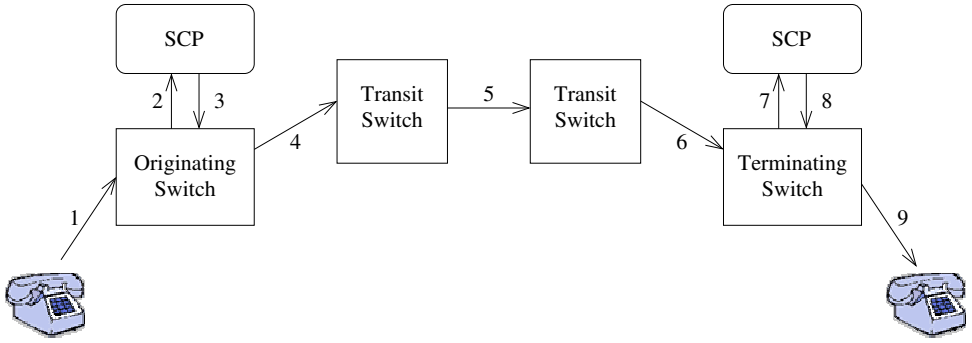

Figure 3.2: Call Flow for a Call to an 800 Number

The GFP also defines the interaction between the telephone switch and the service logic on the general purpose computer, by means of a state machine on the switch (the Basic Call Process, or BCP), and remote procedure calls to the computer. The telephone switch has certain points in the progress of a call where remote procedure calls can be made (Points of Initiation (POI)), and points in the call where the service logic can instruct the switch to return to (Points of Return (POR)). Examples of POIs are *Address Collected* and *Busy*, which occur in the BCP when the digits for the call are collected, and when the remote party signals busy, respectively. Examples of PORs are *Initiate Call* and *Clear Call*.

The next plane is the Distributed Functional Plane (DFP), which maps the abstract GFP into functional blocks, and then defines the required flow of information between them. Finally, the DFP is realized in the Physical Plane, which maps the functional entities into real devices. The IN defines a number of devices, including a Service Switching Point (SSP), which is a telephone switch, a Service Control Point (SCP), which is a general purpose computer that can execute the remote procedure calls from the switch, and a Service Node (SN), which executes service logic, but also has a switch fabric so that it can also generate tones, play announcements, and provide additional services. The physical plane also defines the protocols among these entities. Most important among these is the IN Application Protocol, or INAP [27], which carries the information for the information flows defined in the DFP. INAP can be thought of as a special-purpose remote procedure call protocol. This separation between SCPs and SSPs allows service execution environments to be provided, maintained, and upgraded separately from telephony providers' core circuit switches.

An simplified example of a call flow for a phone call that makes use of IN services is shown in Figure 3.2. In this figure, a call originates at the calling phone, which dials an 800 number. The call signaling arrives at the originating switch (1). Since the switch does not know how to handle the 800 number directly, it asks the SCP for further instructions (2), and the response (3) specifies the further actions the switch should take, including a routing number where the call should be connected to (4). The call passes through another transit switch (5) and eventually arrives at the terminating switch. This switch also consults an SCP for instructions (6), and the response (7) tells it to complete the call to a subscriber line (8).

The IN has been standardized incrementally, starting with a baseline set of services and associated call models and protocols (called IN capability set 1, or CS1). Increasingly more complex services, models, and protocols have been developed as part of capability set 2, and more recently capability set 3. The services enabled with CS1 include freephone, televoting, follow-me, call screening, and call forwarding, among others. IN Capability Sets 2 and 3 (CS2 and CS3) support more powerful features, such as call transfer, call waiting, message store and forward, conference calling, and call completion to busy subscriber.

The Intelligent Network architecture does not, however, allow ordinary end-users — as opposed, say, to sophisticated corporate users deploying custom-programmed switches — to provision customized advanced telephony services. There are two major reasons for this. First of all, service description is not, itself, standardized. Though Service Creation Environments (SCEs) often use SIBs to describe services that can be executed on Service Control Points, these environments are custom-designed, and are wedded to their SCPs. Willner and Lee [28] have a more in-depth discussion of how the notion of SIBs has varied in the actual practice of IN deployment.

Furthermore, and more fundamentally, SCPs and SSPs are integrally part of the "network" side of traditional telephone networks, and are not exposed to ordinary users. Traditional telephony models do not have a notion of per-user services, other than services that can be individually enabled or configured. Direct exposure of such SCPs and SSPs would be a significant security issue in the network model of the PSTN, as the communications between network elements is assumed to be physically secure and thus not in need of the "defensive programming" required of Internet services. Moreover, before the Internet, telephone networks had no "rich"

communications channel which allowed users to communicate custom services with network devices; user-based control of telephony services is generally assumed to be limited to that which can be done with a twelve-digit telephone.

3.2 Internet Services

3.2.1 Dynamic Web Content

There are a number of mechanisms which provide "services" — dynamic content — in the WWW. These can be broken into two types: server side and client side.[1] Server side mechanisms allow the content of the web response to be generated by either a separate process (in the case of the Common Gateway Interface (CGI) [30]), or by the web server (in the case of server side Javascript [31], Java servlets [32], PHP [33], or Active Server Pages (ASP) [34]. In the case of client side services, the web page returned by the server can contain programs which execute to either interact with the user, or provide dynamic content. Java and Javascript are examples of this.

In the case of CGI, the generation of the content for the response is performed by a separate process. When the web server receives a request, it spawns a separate process (called a co-process) to execute the script. The standard output of the script process is connected to a handle on the server, as is the standard input.

Before spawning the process, the server also sets a number of environment variables. These environment variables are data that are accessible to the script process. The server uses the environment variables to pass information to the script. The information that is passed includes:

Request Headers: The values of various headers in the HTTP request are passed to the script. This includes information on who is making the request, what browser they are using, etc.

Server Information: Information about the server is passed to the script. This includes the type of server and the version of CGI.

[1] These types of services for the web should not be confused with the similar term "Web Services." This term refers to the use of web protocols as a medium for remote procedure calls, usually via the Simple Object Access Protocol (SOAP) [29].

Requested Data: The Uniform Resource Identifier (URI) that has been requested in the request is passed to the script process.

Body Information: Information on the length and type of the body in the request are passed to the script process. The body of the request often contains form data.

User Information: Information on the IP address and identity of the requesting user are passed to the script.

In addition to setting these environment variables, the server passes the body of the request (often web form data) to the script process through the standard input of the script process. The script then processes the data, and returns a result.

The result is in the form of a HTTP response to be sent to the client, usually a success message with a web page of some sort in the body. This response is passed through the standard output of the script process, and is read by the server. The server fills in any necessary message headers, and sends to the response to the client. The script process then terminates.

In-process web services (server-side Javascript, Java Servlets, PHP, ASP, etc) follow a similar model: data about a client's request is passed to an execution environment, which then generates an HTTP response. However, rather than passing the request data to a co-process, these services instead pass it internally to a module of the web server. This can result in significant efficiency gains — process initiation can be expensive — at the cost of reduced flexibility. Services must be described in the specified language, whereas with CGI they can use any execution environment available on the web server's host. (There are some service creation mechanisms which attempt to provide the best of both of these mechanisms, most notably FastCGI [35], but these have not achieved wide deployment.) The mechanism used by the popular Apache web server to implement these services is described in Section 6.2.1.

3.2.2 E-Mail Services

E-mail processing is another important area of services in the Internet environment. As the volume of users' e-mail gets increasingly large, they often want to enable automated processing of mail messages. Common actions are to sort mail into separate mail folders, modify it, discard it,

or selectively forward it, based on a number of criteria, such as the sender, the mailing list that forwarded it, certain text in its headers or body, or based on an automated classification as Spam (unsolicited commercial e-mail) or a virus.

Initial work on e-mail processing was done in Mail User Agents — users' e-mail clients — or Mail Delivery Agents such as `procmail` [36], programs which transfer received mail into a user's mail spool. These programs can generally accomplish all the tasks detailed above; however, they have the disadvantage that they operate on the client side. In the common case, users' mail is kept on a mail server, and read using an access protocol such as POP [37] or IMAP [38]. Unless system administrators allow users access to the mail server, messages must be downloaded to the client to be processed. This can preclude the usefulness of e-mail processing, if, for example, the goal is to discard or defer the handling of certain messages, to avoid having to retrieve them over low-bandwidth access links.

The Sieve language [39] solves this problem by defining a standardized language to describe e-mail processing in mail servers. An example of a Sieve script, taken from the specification, is shown in Figure 3.3. The language defines a set of operations and patterns which users can then upload to their mail servers, to have their mail processed in a custom manner. Sieve is also significant in its lack of expressive completeness; it does not allow loops or recursion. This inherently prevents certain harmful constructs, and imposes resource restrictions (of the form described in Section 2.6.2) on how much "harm" to the system or network a user's script could do, through malice or incompetence.

Sieve's lack of expressive completeness was a major influence on the design of the Call Processing Language, described in Chapter 5. While the CPL does not use Sieve's syntax, it similarly does not allow loops, variables, or recursion. However, Internet telephony service creation differs from e-mail processing in several significant ways: there is no meaningful analogy of a "busy" or "no answer" response with e-mail, and indeed very few situations where one can say immediately that message delivery failed. Sieve scripts tend to be a single list of rules and processing instructions, whereas the CPL forms a directed acyclic graph in which an instruction can be performed in the case where an earlier one failed.

```
# Example Sieve Filter
# Declare any optional features or extension used by the script
#
require ["fileinto", "reject"];

#
# Reject any large messages (note that the four leading dots get
# "stuffed" to three)
#
if size :over 1M
        {
        reject text:
Please do not send me large attachments.
Put your file on a server and send me the URL.
Thank you.
.... Fred
.
;
        stop;
        }
#
# Handle messages from known mailing lists
# Move messages from IETF filter discussion list to filter folder
#
if header :is "Sender" "owner-ietf-mta-filters@imc.org"
        {
        fileinto "filter";  # move to "filter" folder
        }
#
# Keep all messages to or from people in my company
#
elsif address :domain :is ["From", "To"] "example.com"
        {
        keep;               # keep in "In" folder
        }

#
# Try and catch unsolicited email.  If a message is not to me,
# or it contains a subject known to be spam, file it away.
#
elsif anyof (not address :all :contains
               ["To", "Cc", "Bcc"] "me@example.com",
             header :matches "subject"
               ["*make*money*fast*", "*university*dipl*mas*"])
        {
        # If message header does not contain my address,
        # it's from a list.
        fileinto "spam";   # move to "spam" folder
        }
else
        {
        # Move all other (non-company) mail to "personal"
        # folder.
        fileinto "personal";
        }
```

Figure 3.3: A Sample Sieve Script

3.2.3 Active Networks

There are also service creation mechanisms which apply at lower levels of the Internet protocol stack. Active Networks [40, 41] are a mechanism by which routers in the network receive instructions on how individual packets should be handled, either by having the packet encapsulate its own instructions, or by having scripting instructions resident on to the routers on the path the packets will take. The service creation mechanisms described in this thesis are different in that they implement control at the application layer. These mechanisms do not attempt to control the overwhelming majority of packets of an Internet telephone call, the media packets, and indeed the majority of the *signalling* packets of a call do not need special handling by a service creation mechanism. By working at this higher level, we can be independent of the underlying signalling mechanisms; and since these service environments need only to be deployed in specific signalling servers, it is possible to deploy them on Internet hosts as they exist today.

3.3 Telephony Application Programming Interfaces

There have been several efforts to create standardized APIs for telephony services. The Java Advanced Intelligent Networks (JAIN) APIs [42] are a suite of interfaces for programs written in Java to control telephony services. In addition to a number of fairly low-level APIs for the direct control of protocol stacks, its core specification is Java Call Control, Coordination, and Transactions [43], which provides means of controlling both intelligent network and Internet telephony services. This specification is further split into an API for controlling intelligent peripherals (end systems), and an API for controlling network switches. The call model of the JAIN Call Control API follows fairly closely to the Intelligent Network model described in Section 3.1: the model of a reasonably traditional telephony switch. Services are triggered call events, and calls are possibly transitioned into alternate call states. The protocol to control network switches assumes the capacities of a fairly traditional switch environment, with control of media flows, mid-call signalling, and the like.

The Parlay APIs [44] are similar to JAIN Coordination and Transactions, but define a distributed object model, rather than a Java API. In other respects the two approaches are similar,

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
  <form>
  <field name="drink">
     <prompt>Would you like coffee, tea, milk, or nothing?</prompt>
     <grammar src="drink.grxml" type="application/srgs+xml"/>
  </field>
  <block>
     <submit next="http://www.drink.example.com/drink2.asp"/>
  </block>
 </form>
</vxml>
```

Figure 3.4: A Sample VoiceXML Script

and mappings between the two have been defined.

Java SIP servlets [45] — now also part of JAIN, but originally developed separately — modify the web's Java servlet model (discussed in Section 3.2.1) to make it applicable to SIP in much the same way that SIP CGI, the approach described in Chapter 4, extends HTTP CGI to SIP. Java SIP Servlets address many of the same problems as SIP CGI, and use many similar interfaces; SIP CGI was, indeed, an inspiration for Java SIP Servlets. The trade-offs between Java SIP Servlets and SIP CGI are much the same as the trade-offs for their web counterparts: SIP Servlets are language-specific, but avoid the overhead of process invocation, whereas SIP CGI is language-independent but must execute in a separate process space every time a service is invoked.

3.4 Telephony Scripting Languages

There are several instances of scripting languages which use the common syntax of the Extensible Markup Language (XML) to describe telephony services. They share this characteristic with the Call Processing Language, described in Chapter 5, but have different goals or application domains.

VoiceXML [46] is a language to describe Interactive Voice Response services. Figure 3.4 shows an example VoiceXML script, which asks the user for a choice of drink and then submits it to a server script. (This example script comes from the VoiceXML specification; it is not clear how the example intends for the drink to be delivered to the user.) VoiceXML itself statically describes interactions between users and voice systems. Like HTML, the actual server logic is left to back-end scripts, which dynamically create the output presented to the user. This is designed to use the same mechanisms as the "ordinary" (browser-based) web services described in Section 3.2.1; based on the URL naming, the example in Figure 3.4 appears to be using Active Server Pages (ASP).

The Call Control Extensible Markup Language (CCXML) [47] is a work-in-progress language, related to (but not dependent on) VoiceXML, which is a similar model allowing scripts to control call events. A sample CCXML script, handling events related to call transfer, is shown in Figure 3.5. As can be seen in the sample script, CCXML uses variables and loops extensively; as it also has a `<goto>` operator, it is Turing-complete.

The Service Control Markup Language (SCML) [48] is similar, in some respects, to the Call Processing Language described in Chapter 5, and adapts a number of its protocol-independent parts, but is designed to more closely map to the trigger APIs defined by Intelligent Network models. An example SCML script for the Voicemail on Busy service is shown in Figure 3.6. Compare it with a similar CPL script in Figure 5.5. SCML primitives are closely mapped to the IN call model; SCML scripts can perform in-call actions, initiate calls, and connect to end systems.

Finally, the Language for End System Services (LESS) [16] is a language based on the Call Processing Language (CPL) described in Chapter 5. Figure 3.7 shows a sample LESS script. When `sip:xyz@foo.com` is online, the script automatically places an outgoing call to `sip:xyz@foo.com` and alerts the user. LESS generalizes the CPL, extending it to a number of additional problem domains.

```
<ccxml version="1.0">

  <var name="in_callid" />
  <var name="out_callid" />
  <var name="dialogid" />

  <eventhandler statevariable="mystate">
    <transition event="dialog.transfer" name="in_event">
      <assign name="mystate" expr="'calling_out'" />
      <assign name="in_callid" expr="in_event.callid" />
      <assign name="dialogid" expr="in_event.dialogid" />

      <createcall dest="in_event.URI" name="out_callid" />
    </transition>

    <transition state="calling_out"
     event="connection.CONNECTED" name="out_event">
      <assign name="mystate" expr="'outgoing_call_active'" />
      <join sessionid1="in_callid"
       sessionid2="out_event.callid " duplex="full" />
      <if cond="in_event.values.bridge == 'false'">
        <send event="connection.disconnect.transfer"
         dest="dialogid"/>
      </if>
    </transition>

    <transition state="outgoing_call_active"
     event="connection.DISCONNECTED"
     name="out_event">
      <assign name="mystate" expr="'outgoing_call_finished'" />
      <if cond="in_event.bridge == 'true'">
        <if cond="in_event.callid == out_callid">
          <assign name="results" expr="far_end_disconnect" />
          <send event="dialog.vxml.transfer.complete"
           dest="in_event.dialogid" namelist="results" />
        <else/>
          <send event="connection.disconnect.hangup"
           dest="dialogid" />
        </if>
      </if>
    </transition>
  </eventhandler>
</ccxml>
```

Figure 3.5: A Sample CCXML Script

```
<scml>
 <terminating>
  <address-switch field="terminating">
   <address is="sip:smith@phone.example.com">
    <disconnected causeCode="CAUSE_BUSY">
     <routeCall connectionPtr="conC">
      <arguments>
       <targetAddress>sip:smith@voicemail.
          example.com</targetAddress>
      </arguments>
     </routeCall>
    </disconnected>
   </address>
  </address-switch>
 </terminating>
</scml>
```

Figure 3.6: A Sample SCML Script

```
<LESS:LESS
 xmlns:LESS="urn:ietf:params:xml:ns:LESS"
 xmlns:Generic="...:ns:LESS:generic"
 xmlns:Presence="...:ns:LESS:presence"
 xmlns:UI="...:ns:LESS:ui"
 xmlns:xsi="http://.../XMLSchema-instance"
 xsi:schemaLocation="......"
 name="xyzOnlineCall" priority="0.8">
 <subaction name="generalCall">
  <Generic:call/>
  <UI:alert message="Calling %address%"/>
 </subaction>
 <Presence:notification
   direction="incoming" package="presence">
  <Presence:presence-switch>
   <event package="presence" is="OPEN">
    <LESS:address-switch field="origin">
     <address is="sip:xyz@foo.com">
      <sub ref="generalCall"/>
     </address>
     <otherwise>
      <UI:alert message="%address% online"/>
     </otherwise>
    </LESS:address-switch>
   </event>
  </Presence:presence-switch>
 </Presence:notification>
</LESS:LESS>
```

Figure 3.7: A Sample LESS Script

Chapter 4

The Common Gateway Interface for SIP

One of the two environments needed for the creation of Internet telephony services is a low-level service programming interface for administrators and trusted service creators. In the World Wide Web, the Common Gateway Interface (CGI) has served as a popular means towards programming web services. Due to the similarities between the Session Initiation Protocol (SIP) and the Hyper Text Transfer Protocol (HTTP), CGI is a good candidate for service creation in a SIP environment. This chapter describes the SIP CGI interface for providing SIP services on a SIP server.

4.1 Introduction

We concluded in Chapter 2 that two types of mechanisms are needed for a complete service programming solution — a flexible, general purpose one primarily targeted at administrators, and a simpler, more restricted one for end users.

The WWW has evolved with its own set of tools for service creation. Originally, web servers simply translated URLs into filenames stored on a local system, and returned the file content. Over time, servers evolved to provide dynamic content, and forms provided a means for soliciting user input. In essence, what evolved was a means for service creation in a web environment. There are now many means for creation of dynamic web content, including server

side JavaScript, PHP, servlets, and the common gateway interface (CGI) [30].

Multimedia communications, including Internet telephony, also require a mechanism for creating services. This mechanism is strongly tied to the features provided by the signaling protocols. The Session Initiation Protocol (SIP) [1] has been developed for initiation and termination of multimedia sessions. SIP borrows heavily from HTTP, inheriting its client-server interaction and much of its syntax and semantics. For this reason, the web service creation environments, and CGI in particular, is attractive as a starting point for developing a SIP based service creation environment.

4.2 Motivations

CGI has a number of strengths which make it attractive as an environment for creating SIP services:

Language independence: CGI works with Perl, Python, C, Visual Basic, Tcl, and many other languages, as long as they support access to environment variables.

Exposes all headers: CGI exposes the content of all the headers in an HTTP request to the CGI application. An application can make use of these as it sees fit, and ignore those it doesn't care about. This allows all aspects of an HTTP request to be considered for creation of content. In a SIP environment, headers have an even greater importance than in HTTP. They carry critical information about the transaction, including caller, callee, subject, contact addresses, organizations, extension names, registration parameters and expirations, call status, and call routes, to name but a few. It is therefore critical for SIP services to have as much access to these headers as possible. For this reason, CGI is very attractive.

Creation of responses: CGI is advantageous in that it can create all parts of a response, including headers, status codes and reason phrases, in addition to message bodies. This is not the case for other mechanisms, such as PHP, which are focused primarily on the body. In a SIP environment, it is critical to be able to generate and modify all aspects of a response (and all aspects of a proxied request), since the body is usually not of central importance in SIP service creation.

Access to any resources: Since the CGI script is a general purpose program, it can use existing APIs to access any desired network service.

Because of these properties, CGI makes an ideal starting point for service creation in an IP telephony context. The interface between the server and the service logic is very flexible (enabling entire packets to be sent back and forth), and the set of services accessible by the service logic is unlimited.

CGI has some disadvantages; notably its separate-process execution model makes service execution somewhat slower than other models. This is discussed further in Section 4.9.

The similarity of HTTP and SIP makes its application to Internet telephony straightforward. Usage of CGI for administrator defined services has a number of other advantages:

Component reuse: Many of the HTTP CGI utilities allow for easy reading of environment variables, parsing of form data, and often parsing and generation of header fields. Since SIP reuses the basic RFC 822 [49] syntax of HTTP, many of these tools are applicable to SIP CGI.

Familiar environment: Many web programmers have familiarity with CGI. Reusing it for SIP means one less environment programmers must be familiar with.

Ease of extensibility: Since CGI is an interface and not a language, it becomes easy to extend and reapply to other protocols, such as SIP.

4.3 Comparison between HTTP CGI and SIP CGI

While SIP and HTTP share a basic syntax and a request-response model, there are important differences. Proxies play a critical role in services for SIP, while they are less important for HTTP services. SIP servers can fork requests (proxying multiple requests when a single request is received), an important capability absent from HTTP. SIP supports additional features, such as registrations, which have no analog in HTTP. The differences between SIP CGI and HTTP CGI primarily reflect these differences in their underlying protocols. SIP CGI runs primarily on proxy, redirect, and registrar servers, rather than user agent servers (which are the equivalent of

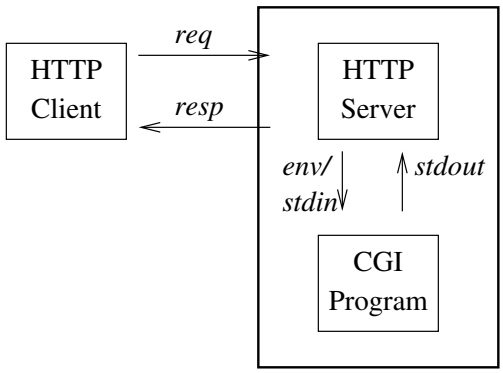

Figure 4.1: HTTP CGI Model

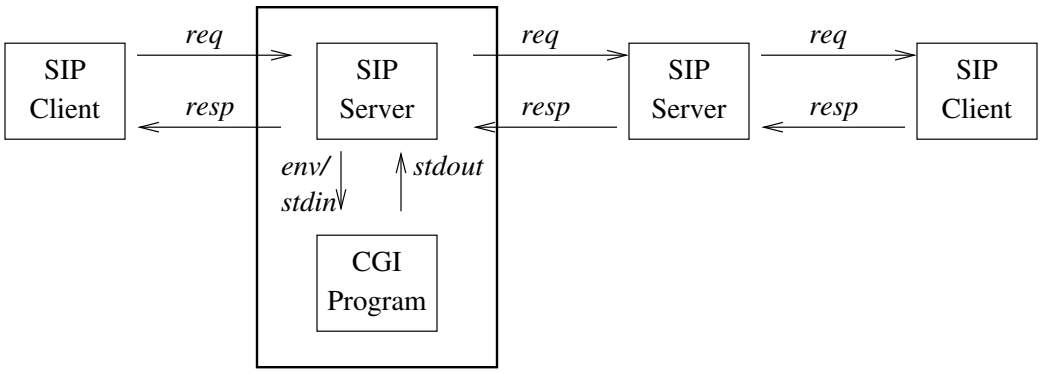

Figure 4.2: SIP CGI Model

origin servers in HTTP). SIP CGI allows the script to perform specific messaging functions not supported in HTTP CGI (such as proxying requests), and SIP CGI introduces a persistence model that allows a script to maintain control through multiple message exchanges. HTTP CGI has no inherent support for persistence for scripts.

4.3.1 Basic Model

The basic model for HTTP CGI is depicted in Figure 4.1. A client issues an HTTP request, which is passed either directly to the origin server (as shown), or is forwarded through a proxy server. The origin server executes a CGI script, and the CGI script returns a response, which is passed back to the client. The main job of the script is to generate the body for the response. Only origin servers execute CGI scripts, not proxy servers.

For SIP CGI, the model is different, and is depicted in Figure 4.2. The client generates a request, which is forwarded to a server. The server may generate a response (such as an error or redirect response). Or, if the server is a proxy server, the request can be proxied to another server, and eventually to a user agent, and the response is passed back upstream, through the server, and back towards the client. A SIP proxy server may additionally fork requests, generating multiple requests in response to a received request. Generally, a proxy server will not generate the content in responses. These contain session descriptions created by user agents. Services, such as call forward and mobility services, are based on the decisions the server makes about (1) when, to where, and how many requests to proxy downstream, and (2) when to send a response back upstream.

An HTTP server is mainly concerned about the generation of responses. A SIP server is generally concerned about performing three basic operations:

Proxying requests: Receiving a request, adding or modifying any of the headers, deciding on a set of servers to forward the request to, and forwarding it to them.

Returning responses: Receiving a response, adding or modifying any of the headers, and passing the response towards the client.

Generating responses: Receiving a request, generating a response to it, and sending it back to the client.

When a request is received, one or more of the above operations may occur at once. For example, a SIP server may generate a provisional response and proxy the original request to two servers. This implies that SIP CGI must encompass a greater set of functions than HTTP CGI. These functions are a super-set of the simple end-server request/response model.

4.3.2 Persistence Model

In HTTP CGI, a script is executed once for each request. It generates the response, and then terminates. There is, as a general rule, no state maintained across requests from the same user (although this can be done — and is — for more complex services such as database accesses,

which encapsulate state through mechanisms such as client-side cookies [50] or dynamically-generated URLs). A transaction is just a single request and a response.

In SIP CGI, a request can generate many further proxied requests, which themselves will generate responses. A service will often require these responses to be processed, and additional requests or responses to be generated. As a result, whereas an HTTP CGI script executes only once per request, a SIP CGI script must maintain control over an entire transaction, encompassing numerous events.

There are two ways in which this persistence could be accomplished:

- Allow the script to be executed once, and then continue running for the duration of the transaction. This would require new IPC mechanisms, since the use of environment variables for server to script CGI wouldn't support asynchronous notification of packet events.
- Stick with the current HTTP CGI model, so that the script is called, generates some output, and then terminates. Persistence can be achieved by re-executing the script on the next event with a state token.

While the first option has lower process overhead, it departs significantly from the clean model presented by HTTP CGI. Therefore, SIP CGI chooses the second approach. Scripts are executed for every incoming request and response. A state token (called a *script cookie*) is passed from the script to the server through a SIP CGI meta-header. When the script is re-executed for the same transaction at some later point, the server passes the cookie back to it through environment variables. This token is opaque to the server, and can contain anything of use to the script. In essence, the execution of the script is a remote procedure call, with the particular procedure dependent on the semantic of the cookie.

For example, consider a request which arrives at a SIP server. The server calls a CGI script, which generates a provisional response and a proxied request. It also returns a token to the server, and then terminates. The response is returned upstream towards the client, and the request is proxied. When the response to the proxied request arrives, the script is executed again. The environment variables are set based on the content of the new response. The script is also passed back the token. Using the token as its state, the script decides to proxy the request to a

different location. It therefore returns a proxied request, and another token. The server forwards this new request, and when the response comes, calls the CGI script once more, and passes back the token. This time, the script generates a final response, and passes this back to the server. The server sends the response to the client, destroys the token, and the transaction is complete.

4.3.3 SIP CGI Triggers

In many cases, calling a CGI script on the reception of every message is inefficient. CGI scripts come at the cost of significant overhead since they generally require creation of a new process. Therefore, it is important in SIP CGI for a script to indicate, after the first time it is called, under what conditions it needs to be called for the remainder of the transaction. If the script is not called, the server takes a "default" action. This allows an application designer to trade off flexibility for computational resources. Making an analogy to the Intelligent Network (IN), a script is able to define the triggers for its future execution.

In summary, whereas an HTTP CGI script executes once during a transaction, a single SIP CGI script may execute many times during a transaction, and may specify at which points it would like to have control for the remainder of the transaction.

4.3.4 Naming

In HTTP CGI, the CGI script itself is generally the resource named in the request URI of the HTTP request. This is not so in SIP. In general, the request URI names a user to be called. The mapping to a script to be executed may depend on other SIP headers, including To and From fields, the SIP method, status codes, and reason phrases. As such, the mapping of a message to a CGI script is purely a matter of local policy administration at a server. A server may have a single script which always executes, or it may have multiple scripts, and the target is selected by some parts of the header.

4.3.5 Environment Variables

In HTTP CGI, environment variables are set with the values of the paths and other aspects of the request. As there is no notion of a path in SIP, some of these environment variables do not make

sense. However, many environment variables do continue to make sense; these are re-used.

4.3.6 Timers

In SIP, certain services require that the script gets called not only when a message arrives, but when some timer expires. The classic example of this is "call forward no answer." To implement this service with SIP CGI, the first time a script is executed, it generates a proxied request, and also indicates a time at which to be called again if no response comes. This kind of feature is not present in HTTP CGI, and some support for it is needed in SIP CGI.

4.4 Services Enabled

SIP CGI provides a powerful tool for enabling new Internet telephony services to be developed rapidly. Many of the traditional telephone services, such as automatic call distribution (where the calls are forwarded to operators based on some specified logic), call forward busy, call forward unconditional, and follow-me, are easily supported with small Perl scripts.

However, these are just the tip of the iceberg. Since the CGI script is a normal process, it can have access to network and system resources, such as e-mail, web, database, and file store. Consider some of the possibilities:

Redirection to web: When a call arrives at a server, the CGI script can lookup the called user in a corporate database, and using the information found there (such as picture, phone number, e-mail, etc) generate a web page dynamically containing reach information for that user. This web page can be returned to the caller in a SIP response.

Instant message when busy: When a call arrives at a server, the CGI script accesses an IN database to find a phone number for the user. The script finds a telephony gateway to complete the call over the phone network. If the user's phone is busy, the script assumes this is because they are logged in, and sends an instant message to them over the Internet, indicating who called. The script then generates a busy message for the caller.

Forward to e-mail: When a call arrives at a server, the CGI script looks up the callee in a

database and obtains their e-mail address. The script then sends e-mail to the user, containing the caller, time of call, and call subject. The caller is redirected to a voicemail server to record a message.

Access to e-mail, web, and instant messaging services are already available through APIs in existing languages. These can be used directly to provide these services for IP telephony.

4.5 Example CGI Operation

This section illustrates a simple execution of a SIP CGI script. Assume the following request was received by a SIP proxy server, triggering the execution of a CGI script:

```
INVITE sip:astromomer@lab2.university.edu SIP/2.0
Via: SIP/2.0/UDP ganymede.university.edu
Subject: Io's orbit
From: sip:physicist@university.edu
To: sip:astronomer@university.edu
Call-ID: 089y30n0983h2f0@112.34.55.2
CSeq: 1 INVITE
Contact: sip:j.smith@ganymede.university.edu
Call-Info: <http://www.university.edu/j.smith/photo.jpg>;
    purpose=icon
```

The script outputs the following:

```
CGI-PROXY-REQUEST sip:b.jacobs@lab2.university.edu SIP/2.0
CGI-Remove: Call-Info
Subject: Earth's rotation

SIP/2.0 180 Ringing

CGI-SCRIPT-COOKIE asd-9unas SIP/2.0
```

The script output contains three messages, which are separated by blank lines. The first instructs the server to proxy the received request to `b.jacobs@lab2.university.edu`. The CGI-Remove header instructs the server to remove the Call-Info header from the proxied request. The Subject header instructs the server to replace the Subject header in the proxied request with the one specified. The server will perform these operations and then generate the following proxied request:

```
INVITE sip:b.jacobs@lab2.university.edu SIP/2.0
Via: SIP/2.0/UDP ganymede.university.edu
Subject: Earth's rotation
From: sip:physicist@university.edu
To: sip:astronomer@university.edu
Call-ID: 089y30n0983h2f0@112.34.55.2
CSeq: 1 INVITE
Contact: sip:j.smith@ganymede.university.edu
```

The second message in the script output instructs the server to generate a ringing response towards the caller. The server will fill in all of the required header fields and send the following response:

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP ganymede.university.edu
From: sip:physicist@university.edu
To: sip:astronomer@university.edu
Call-ID: 089y30n0983h2f0@112.34.55.2
CSeq: 1 INVITE
```

The last message in the script output instructs the server to set the script cookie to `asd-9unas`, a string with meaning only for the script. The next time the script is invoked, when a response to the proxied request arrives, this cookie is passed back to the script in an environment variable.

The header processing rules provide a script flexibility in choosing its level of control. A simple script can let the server handle all header processing. A more complex script can completely manage the server processing by generating all of the headers.

```
#! /usr/bin/perl -w

# Reject messages whose 'From:' matches 'sip:.*@example.com'
# by responding with 600 Busy.

if (defined $ENV{SIP_FROM} &&
    $ENV{SIP_FROM} =~ "sip:.*@example.com") {
    print "SIP/2.0 600 I can't talk right now\n\n";
}
```

Figure 4.3: A Sample SIP CGI Script, in Perl

An example of a simple SIP CGI script, written in Perl [51], is shown in Figure 4.3. The script blocks all SIP requests whose From header matches the pattern `sip:.*@example.com`.

4.6 Overview of SIP CGI

When a request arrives at a SIP server, initiating a new transaction, the server will set a number of environment variables, and call a CGI script. The script is passed the body of the request through its standard input stream.

The script returns, on its standard output, a set of SIP action lines, each of which may be modified by CGI and/or SIP headers. This set is delimited through the use of two carriage returns. The action lines allow the script to specify requests and responses to send, and modifications to the script execution state, in addition to the default operation. Generating a response is done by copying the the status line of the response into an action line of the CGI output. For example, the following will create a 200 OK to the original request:

```
SIP/2.0 200 OK
```

The operation of proxying a request is supported by the CGI-PROXY-REQUEST CGI action, which takes the URL to proxy to as an argument. For example, to proxy a request to `dante@inferno.com`:

```
CGI-PROXY-REQUEST sip:dante@inferno.com SIP/2.0
Priority: urgent
```

In this example, the server will take the original request, and modify any header fields normally changed during the proxy operation (such as decrementing Max-Forwards and adding a Via field). This message is then "merged" with the output of the CGI script — SIP headers specified below the action line in the CGI output will be added to the outbound request. In the above example, the Priority header field will be added, replacing any Priority headers in the original message. Note that the action line looks like the request line of a SIP request message. This is done in order to simplify parsing.

To delete headers from the outgoing request, the merge process also supports the CGI header field CGI-Remove. Like SIP headers, CGI header fields are written underneath the action line. They are extracted by the SIP server, and used to provide the server with additional guidance. CGI header fields always begin with CGI- to differentiate them from SIP headers. In this case, the supported values for the CGI-Remove header are the names of headers in the original message.

Returning of responses is more complex. A server may receive multiple responses as the result of forking a request. The script should be able to ask the server to return any of the responses it had received previously. To support this, the server will pass an opaque token to the script through an environment variable, unique for each response received. To return a response, a CGI script needs to indicate which response is to be returned. For example, to return a response named with the token abcdefghij, the following output is generated:

```
CGI-FORWARD-RESPONSE abcdefghij SIP/2.0
```

Finally, if the script does not output any of the above actions, the server does what it would normally do upon receiving the message that triggered the script.

A SIP CGI script is normally only executed when the original request arrives. If the script also wants to be called for subsequent messages in a transaction — due to responses to proxied requests, or (in certain circumstances) ACK and CANCEL requests, it can perform the CGI-AGAIN action:

```
CGI-AGAIN yes SIP/2.0
```

This action applies only to the next invocation of the script; it means to invoke the script one more time. Outputting "no" is identical to outputting "yes" on this invocation of the script

and outputting nothing the next time the script is called.

When the script is re-executed, it may need access to some state in order to continue processing. A script can generate one piece of state, called a cookie, for any new request or proxied request. It is passed to the server through the CGI-SET-COOKIE action. The action contains a token, which is the cookie itself. The server does not examine or parse the cookie. It is simply stored. When the script is re-executed, the cookie is passed back to the script through an environment variable.

```
CGI-SET-COOKIE khsihppii8asdl SIP/2.0
```

Finally, when the script causes the server to proxy a request, responses to these requests will arrive. To ease matching of responses to requests, the script can place a request token in the CGI CGI-Request-Token header. This header is removed by the server when the request is proxied. Any responses received to this request will have the token passed in an environment variable.

4.7 SIP CGI Specification Details

The full SIP CGI specification is not given here; the full description is available in the specification, RFC 3050 [2]. This section gives some of the significant details of the protocol, expanding on the previous section.

4.7.1 Invoking the Script

A SIP CGI script is invoked as an executable program. Only one execution of a CGI script at a time may be outstanding for a given SIP transaction. If subsequently arriving responses would cause a CGI script to be invoked, handling of them is deferred, except for protocol-level ACK responses suppressing retransmission, until CGI scripts for previous messages in the transaction terminate. Messages are processed in the order they are received.

4.7.2 Data Input to the SIP CGI Script

Information is passed to a SIP CGI script through two different sources: a set of named parameters called *metavariables*, and the script's standard input. Metavariables, passed to the script through environment variables, encode information about the message's header fields and other information about the message and the execution environment. On the script's standard input, the server sends the script the request or response's message body.

Metavariable names are conventionally written in all upper-case, following the conventional representation of environment variables on Unix systems. There are two sorts of metavariables: *protocol-specific metavariables*, and the standard metavariables defined by SIP CGI. Protocol-specific metavariables encode the header fields that were included in the original message. Each of these protocol-specific metavariables' names begins with `SIP_`, followed by the header field from the SIP message, converted to upper case, with - changed to _. Their values are taken from the values of the header fields in the message.

Metavariables whose names do not begin with `SIP_` encode other information about the request or the operating environment. The metavariables defined by SIP CGI are listed in Table 4.1; implementations are also free to define additional ones. These metavariables encode information about the server, the message, and the user. Not all metavariables are always provided. If the information they encode is not available or is not relevant to a particular request, they are omitted.

As mentioned, the server sends the message's body on the script's 'standard input' file descriptor. The `CONTENT_LENGTH` metavariable tells the script how large a body is available to be read.

4.7.3 Data Output from the SIP CGI Script

Once the script reads and processes the information about a message, it responds to the server via its 'standard output' file descriptor.

A SIP CGI script's output consists of any number of messages, each corresponding to actions which the script is requesting that the server perform. Messages consist of an action line, whose syntax is specific to the type of action, optionally followed by CGI header fields

Metavariable name	Semantics
AUTH_TYPE	The authentication scheme, if any, used in the message.
CONTENT_LENGTH	The length of the message's message-body entity.
CONTENT_TYPE	The MIME type [52] of the message body.
GATEWAY_INTERFACE	The dialect of SIP CGI being used by the server.
REGISTRATIONS	The current list of contact addresses the user has registered.
REMOTE_ADDR	The IP address that sent the message to the server.
REMOTE_HOST	The fully-qualified domain name corresponding to REMOTE_ADDR.
REMOTE_IDENT	Information about an identity supported by an RFC 1413 [53] request, if available.
REMOTE_USER	The user-ID supplied for validated authentication information.
REQUEST_METHOD	The request method, for a triggering request.
REQUEST_TOKEN	For a response to a proxied request, a token provided to the script in the CGI-PROXY-REQUEST method.
REQUEST_URI	The Request-URI specified for a request.
RESPONSE_STATUS	The status code specified for a response.
RESPONSE_REASON	The reason phrase specified for a response.
RESPONSE_TOKEN	For a response, a token which the script can use in a future CGI-FORWARD-RESPONSE method.
SCRIPT_COOKIE	The value an earlier invocation of the script set with a CGI-SET-COOKIE request.
SERVER_NAME	The hostname of the server.
SERVER_PORT	The port on which the message was received.
SERVER_PROTOCOL	The name and revision of the protocol ("SIP/2.0") on which the message arrived.
SERVER_SOFTWARE	The name and version of the server software.

Table 4.1: Standard SIP CGI Metavariables

and SIP header fields, optionally followed by a SIP body. Action lines determine the nature of the action performed; CGI header fields pass additional instructions or information to the server. Action lines and header fields are followed by a blank line; if the header fields included a Content-Length field, this is followed by a body of the specified length, otherwise another message follows.

CGI Action Lines

Each action line indicates an action the script is requesting of the server. An action line is syntactically either a SIP response, or a SIP request with a method name beginning with CGI-.

An action line which is syntactically a SIP response is known as a "status" action. This causes the server to generate a SIP response and relay it upstream towards the client, copying any SIP headers supplied in the output message, and generating the rest of the headers from the original request. If the output message includes a body, that body is included in the response.

An action line containing a CGI-PROXY-REQUEST method causes the transaction's original request to be forwarded to a specified URI. If the output message includes any SIP headers, these override any headers in the original message. The request's original body is preserved unless the output message includes a new body or the header Content-Length: 0.

Similarly, a CGI-FORWARD-RESPONSE method relays a previously-received response back to the transaction initiator. The same rules apply for accompanying SIP headers and message bodies as for CGI-PROXY-REQUEST. The Request-URI slot of the method indicates what response is to be forwarded. It contains either a response token which the server previously submitted in a `RESPONSE_TOKEN` metavariable, or the string this, indicating that the triggering response is to be sent.

Two action lines control the interaction of the server and the script, rather than causing SIP messages to be sent. The CGI-SET-COOKIE method causes the server to store a script cookie; subsequent script invocations for messages within the same transaction carry this cookie in the `SCRIPT_COOKIE` metavariable. The CGI-AGAIN method controls whether the script will be invoked again for this transaction. If this method is not given, scripts will not be invoked again for the transaction.

If none of the action lines CGI-PROXY-REQUEST, CGI-FORWARD-RESPONSE, or a status response are output — that is to say, the script outputs only CGI-AGAIN, CGI-SET-COOKIE, or nothing — the server performs the default action. The default action is the action that the server would perform had no script been invoked. The default action is also invoked for all script actions after a message for which CGI-AGAIN was not output.

CGI Header Fields

The SIP CGI specification also defines several CGI header fields. These header fields syntactically resemble SIP header fields, but their names all begin with the string CGI-. The SIP server strips out all CGI header fields from any message before sending it.

To assist in matching responses to proxied requests, the script can place a CGI-Request-Token CGI header field in a CGI-PROXY-REQUEST output message. This header contains a token, opaque to the server. When a response to this request arrives, the token is passed back to the script as the `REQUEST_TOKEN` metavariable. This allows scripts to fork a proxy request, and correlate which response corresponds to which branch of the request.

The CGI-Remove header field allows the script to remove SIP headers from the outgoing request or response. The value of this header is a comma-separated list of SIP header fields which should be removed before sending out the message. This complements the ability to add or alter SIP header fields in a request by including them in the output message.

4.7.4 Local Expiration Handling and Locally-Generated Responses

If a CGI script specifies an Expires header field along with CGI-PROXY-REQUEST, the SIP server tracks the expiration timeout locally as well as sending the message to the remote server. When the timeout expires, the server locally generates a 408 Request Timeout response. Locally-generated responses such as this are sent to the CGI script in the same manner as received messages are. However, messages which merely report a problem with a message, such as 400 Bad Request, are not.

This allows a SIP CGI script to implement services like "Call Forward No Answer" to trigger after a user-determined time, even if the remote user agent server is not responding or does not properly handle the Expires header field.

4.7.5 SIP CGI and REGISTER

The specific semantics of a SIP CGI script which is triggered by a REGISTER request are somewhat different than that of those triggered by call-related requests; however, allowing user control of registration may in some cases be useful. The two specific actions for REGISTER that

need to be discussed are the response 200 and the default action. In the former case, the server assumes that the CGI script is handling the registration internally, and does not add the registration to its internal registration database; in the latter case, the server does add the registration to its own database. The server also does not add the registration if a 3xx, 4xx, 5xx, or 6xx status was returned, or if the registration request was proxied to another location.

4.8 SIP CGI Implementation Experience

SIP CGI has been implemented in the Columbia CINEMA server; this implementation is described in more detail in Chapter 6. As a result, users of CINEMA have gained experience with using the SIP CGI interface.

The CINEMA SIP CGI implementation associates scripts with specific users' incoming calls, and also makes possible a "default" script for users without scripts. This association is limited in that it is not currently possible to associate scripts with users' outgoing calls, and it is not currently possible to execute both a default script and a user-specific script. Nonetheless, SIP CGI has been found useful to implement services.

4.8.1 Projects using SIP CGI

As described in Section 4.4, SIP CGI enables many different types of services. Several CINEMA-related projects have used SIP CGI to implement sophisticated services.

One project [54] uses SIP CGI to implement a SIP application-level gateway. SIP INVITE messages (and their responses) describe the RTP media streams they wish to set up by providing SDP session descriptions in their payloads. These session descriptions include IP addresses and ports at which the end system wishes to receive media. If a SIP message traverses a firewall or a network address translator (NAT), an application-level gateway must provide for the RTP packets to flow in both directions. In the case of NAT, the session descriptions must also be modified to describe the translated addresses.

The project uses SIP CGI to implement this application-level gateway. A global SIP CGI script was defined which reads and modifies SDP payloads in INVITE requests and their

responses. The script then communicates with the network address translator to establish a relationship between addresses on either side of the NAT, and modifies the requests' SDP payloads to include the appropriate IP addresses.

Another project [55] used SIP CGI as a platform for implementing E911-style services for SIP users. The goal of the project was to provide emergency service centers with the physical location of a user contacting emergency service. The project achieved this by querying router SNMP tables to determine the location of the physical network port corresponding to the user's source IP address. Because the lookup process can be fairly slow, users' locations are computed when they REGISTER with the SIP server; if a user dials a special emergency number, the list of computed locations is consulted and the user's physical location is sent to the emergency service center.

4.9 Strengths and Weakness of SIP CGI

The projects mentioned in Section 4.8.1 illustrate several of the strengths of SIP CGI. Scripts can modify all aspects of SIP messages, including their bodies, on a very fine-grained level. They can access other network resources, unrelated to the SIP proxy server. And script implementors can implement sophisticated services without needing to alter, or even understand, the proxy server itself.

There are two primary weaknesses of SIP CGI. First of all, its low-level nature tends to force a script author to duplicate algorithms which are already implemented in the server. For example, a call-forward-no-answer script needs to track the number of registration branches started, so that it knows when all branches have returned unsuccessful responses. This could likely be ameliorated with the development of a SIP CGI support library to handle many of these common tasks.

Secondly, and more fundamentally, SIP CGI's process-execution model imposes a fairly substantial overhead on SIP request processing. While measurements of this are not yet available, the overhead of executing a process and waiting for its output limits the number of requests per second that a SIP server can process.

Finally, as per its design, SIP CGI allows script authors full access to the server on which

the script is being executed. This is both a strength and a weakness. It allows the implementation of sophisticated services like those described in Section 4.8.1, but service providers would be naturally reluctant to allow unaudited scripts written by untrusted users.

4.10 Conclusion

SIP CGI is a good solution for its purpose: the creation of sophisticated call handling services by a service provider or trusted third party. Its limitations, however, are such that it should not be a system's only method for service creation. The next chapter addresses an alternate approach: the *call processing language*.

Chapter 5

The Call Processing Language

In addition to the SIP Common Gateway Interface presented in Chapter 4, the discussion in Chapter 2 concluded that there also needs to be a mechanism to describe and control Internet telephony user location services at a higher level, one aimed at users, not administrators. We therefore developed the Call Processing Language (CPL), presented in this chapter.

5.1 Introduction

The SIP Common Gateway Interface, described in Chapter 4, is a low-level administrative interface to control user location services. As such, it solves only part of the problem of the creation of user-location services. To complement it, a high-level service creation mechanism is needed: one aimed at comparatively inexperienced users, who may not be fully trusted by service providers.

There are a number of requirements for such a service creation mechanism. First of all, it must be *safe*. It must be designed so that service providers can feel comfortable making it available to semi-trusted users: users who, through malice or incompetence, might attempt to create invalid or ill-conceived service descriptions. Such service descriptions might, if not prevented, reveal security-sensitive information, cause denial of service, or otherwise interfere with or damage the correct operation of the provider's system. In addition, it should make it difficult for users to accidentally interrupt their phone service; and it should be possible to automatically check a service for correctness.

Second of all, the service creation mechanism must be *usable*. Many members of this service's target audience — ordinary users — cannot be expected to be experienced programmers. They must be able to create service descriptions through the use of graphical tools, and, moreover, must be able to edit pre-existing service descriptions. However, this must not be the entire mechanism; the service descriptions should still be comprehensible and editable, in raw form, by more experienced users and programmers. These two cases need to be fully interchangeable: automated script editors need to be able to read and modify human-generated scripts, and vice-versa.

5.1.1 Rejected Solutions

One possible approach to creating a safe service description language is to adopt an existing programming language for this purpose. The Java Sandbox model [56], for example, has many safety properties: sandboxed Java bytecode executables can, for example, be restricted in what types of network operations they may perform, and what system resources they can access. The Safe Tcl [57] mode of the Tcl [58] language similarly allows restrictions on what is available. Service creation environments of this sort, notably Java SIP Servlets [45], have been proposed for Internet telephony service creation.

This approach was considered and rejected for this type of service description. While such service creation environments are useful, they do not fully satisfy the requirements described above. First of all, restricted execution environments are rarely so restricted as to reliably prevent a runaway process from claiming all available processor time. While it is certainly possible to monitor processes and kill them if they cross some threshold of processor use, this threshold is difficult to standardize and is dependent on the server's execution environment. Moreover, execution environments of this sort are, despite being limited, nonetheless Turing-complete, so it is impossible to automatically prove their behavior correct in the general case.

More significantly, however, program execution environments of this sort fail the other requirement. While automatic graphical program-generators exist, program-editors generally do not; a graphical interface for an arbitrary program written by a human is extremely unlikely to be able to do anything more than represent the language constructs in the most basic manner.

5.2 Inspirations for This Work

Chapter 3 discusses a number of models for the creation of services in various environments. Two in particular are inspirations for the work described in this chapter.

The IN conceptual model [12], described in Section 3.1, has a number of properties that make it a good starting point for end-user defined services. IN defines a very restricted interface between the switch and service logic; services are constructed by the interconnection of a fixed set of building blocks. This creates a natural interface for service creators creating services graphically. It also limits the set of resources available to the program and the ways in which programs can be constructed. Since services are restricted to interconnection of Service-Independent Building Blocks (SIBs) in a directed acyclic graph (DAG), automatic tools can be applied to bound running times. This property is extremely attractive for end user service creation.

The Sieve mail filtering language [39], which is described in Section 3.2.2, also provides a model for this chapter's work. The Sieve language is deliberately restricted, and in particular is not Turing-complete. It provides no way to write a loop or recursion, and variables are not provided. This guarantees that a Sieve script will use only a bounded amount of CPU, and makes programmatic analysis of script behavior possible.

5.3 The Call Processing Language: Overview

The Call Processing Language (CPL) is a language that can be used to describe and control Internet telephony services. It is not tied to any particular signalling architecture or protocol; it can be used, in particular, with both SIP [1] and H.323 [14].

The CPL is powerful enough to describe a large number of services and features, but it is limited in power so that it can run safely in Internet telephony servers. The intention is to make it impossible for users to do anything more complex (and dangerous) than describing Internet telephony services. The language is not Turing-complete, and provides no way to write loops or recursion.

CPL service descriptions are known as CPL *scripts*. These descriptions are designed to be easily created and edited by graphical tools. They are written in XML [59], so parsing them

is easy and many parsers are publicly available. The structure of the language maps closely to its behavior, so an editor can understand any valid script, even ones written by hand. The language is also designed so that a server can easily, at the time that scripts are delivered to it, confirm that scripts are syntactically valid and do not violate any of the server's policy restrictions, rather that discovering this while a call is being processed.

5.4 Structure of CPL Scripts

5.4.1 Abstract structure

Abstractly, a CPL script consists of a collection of nodes, each of which describes an action that can be performed or a choice that can be made. A node may have several parameters, which specify the precise behavior of the node; they usually also have outputs, which depend on the result of the condition or action.

Nodes are arranged in a directed acyclic graph, starting at a single root node; outputs of nodes are connected to additional nodes. When a CPL script is run, the action or condition described by the root node is performed; based on the result of that node, the server follows one of the node's outputs, and that action or condition is performed; this process continues until a node with no specified outputs is reached. Because the graph is acyclic, this will occur after a bounded and predictable number of nodes are visited.

Figure 5.1 shows a flowchart-like graphical representation of a sample CPL script. Nodes and outputs can be thought of informally as boxes and arrows; the CPL is designed so that it can be conveniently edited graphically using this representation. In the example script, an incoming call is checked for its sender's Internet domain. If it is from the domain `example.com`, the call is proxied to the URL `sip:jones@example.com`. There is a ten second timeout on answering the call. If the call was not answered after this time, the call is forwarded to a voicemail server at the URL `sip:jones@voicemail.example.com`. Calls not from the `example.com` domain are redirected directly to the voicemail server (using, e.g., a SIP 302 Moved Temporarily response) without first being proxied.

If an output to a node is not specified, it indicates that the CPL server should perform a

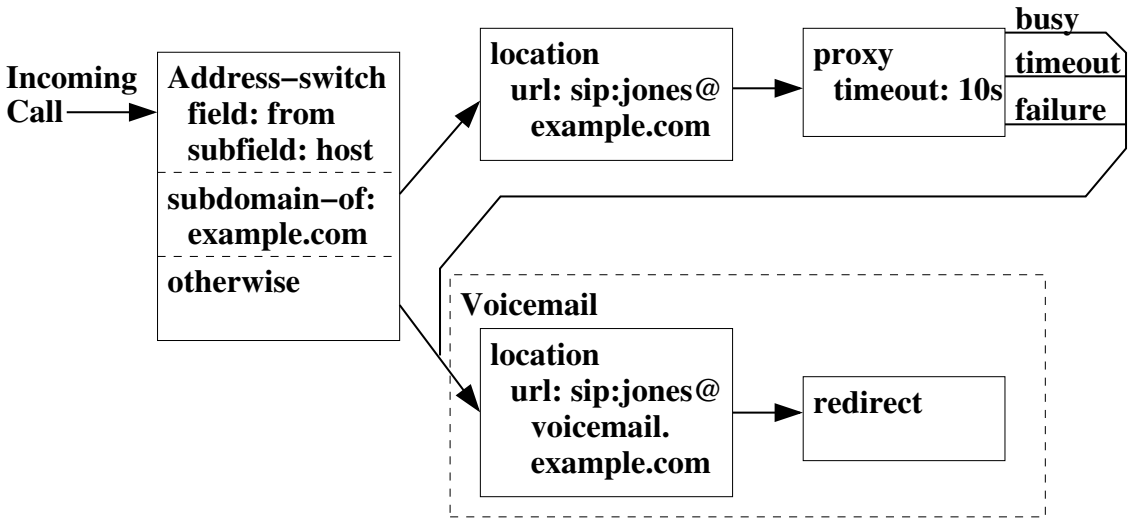

Figure 5.1: Sample CPL Script: Graphical Version

node- or protocol-specific action. Some nodes have specific default actions associated with them; for others, the default action is implicit in the underlying signalling protocol.

5.4.2 XML Structure

Syntactically, CPL scripts are written in XML [59], the extensible markup language. XML consists of a hierarchical structure of tags; each tag can have a number of attributes. It is visually and structurally very similar to HTML [60], as both languages are simplifications of the earlier and larger standard SGML [61]. An XML Schema [62] for CPL defines the structure of a CPL script.

Figure 5.2 shows an XML document corresponding to the CPL script represented graphically in Figure 5.1. Both nodes and outputs in the CPL are represented by XML tags; parameters are represented by XML tag attributes. Typically, node tags contain output tags, and vice-versa.

The connection between the output of a node and another node is represented by enclosing the tag representing the pointed-to node inside the tag for the outer node's output. Convergence (several outputs pointing to a single node) is represented by subactions, discussed further in Section 5.9.

```
<?xml version="1.0" encoding="UTF-8"?>
<cpl xmlns="urn:ietf:params:xml:ns:cpl"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd ">
  <subaction id="voicemail">
    <location url="sip:jones@voicemail.example.com">
      <redirect />
    </location>
  </subaction>
  <incoming>
    <address-switch field="origin" subfield="host">
      <address subdomain-of="example.com">
        <location url="sip:jones@example.com">
          <proxy timeout="10">
            <busy> <sub ref="voicemail" /> </busy>
            <noanswer> <sub ref="voicemail" /> </noanswer>
            <failure> <sub ref="voicemail" /> </failure>
          </proxy>
        </location>
      </address>
      <otherwise>
        <sub ref="voicemail" />
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

Figure 5.2: Sample CPL Script: XML Version

5.4.3 High-level Structure

At a high level, a CPL script contains of a set of *call processing actions*. A call processing action is a structured tree of nodes and outputs, starting with a node, that describes the operations and decisions a telephony signalling server performs on a call set-up event. There are two types of call processing actions: *top-level actions* and *subactions*. Top-level actions are actions that are triggered by signalling events that arrive at the server. Two top-level action types are defined: incoming, the action performed when a call arrives whose destination is the owner of the script; and outgoing, the action performed when a call arrives whose originator is the owner of the script. Subactions are actions which can be called indirectly by other actions. The CPL forbids subactions from being called recursively: see Section 5.9.

The XML structure of a CPL script consists of tags corresponding to each subaction and

top-level action. This higher-level information is all enclosed in a special tag cpl, the outermost tag of the XML document.

Nodes and outputs are both described by XML tags. There are four categories of CPL nodes: *switches*, which represent choices a CPL script can make; *location modifiers*, which add or remove locations from the location set (see Section 5.4.4); *signalling operations*, which cause signalling events in the underlying protocol; and *non-signalling operations*, which trigger behavior which does not effect the underlying telephony protocol.

5.4.4 Location Model

For flexibility, one piece of information necessary for the function of a CPL is not given directly as node parameters: the set of locations to which a call is to be directed. (There will often be more than one such location; call forking, sending a request to more than one destination, is common in SIP.) Instead, this set of locations is stored as an implicit global variable throughout the execution of a processing action and its subactions. This allows locations to be retrieved from external sources or filtered without requiring general language support for such operations (which could harm the simplicity and tractability of understanding the language). The specific operations which add, retrieve, or filter location sets are given in Section 5.6.

For the incoming top-level call processing action, the location set is initialized to the empty set. For the outgoing action, it is initialized to the destination address of the call.

5.5 Switches

Switches represent choices a CPL script can make, based on either attributes of the original call request or items independent of the call.

All switches are arranged as a list of conditions that can match a parameter from the call request or the environment. Each condition indicates an output of the switch node; the output points to the next node to execute if the condition holds. These conditions are checked in the order they are listed in the script; the output corresponding to the first matching node is taken. Additionally, switches also have two optional outputs. The output not-present matches if the

variable the switch was to match was not present in the original call setup request. (This is sometimes described by saying that the information is "absent.") The output otherwise follows all other outputs and matches if no other condition matched.

If no condition matches and no otherwise output was present in the script, the default script behavior is taken. See Section 5.10 for more information on this.

The following switch types are defined.

Address Switch: An *address switch*, indicated by the tag address-switch, allows a CPL script to make decisions based on one of the addresses present in the original call request. Three address field types are defined: origin, destination, and original-destination, respectively corresponding to the sender of the request, the current destination of the request, and the original destination of the request before any forwarding was performed.

Address switches can either match on an entire address, or one of a number of subfields of it: address-type, for determining the type of the address (tel, sip, or h323); tel, for the telephone-number part of the address; user, for a username part; host, for a hostname part; port for an IP port number; and display for the display name associated with the address.

Each output, indicated by an address tag, defines a pattern to compare to the matched address or address subfield. Three pattern types are defined: is, indicating an exact match with the address or subfield; contains, indicating a substring match; and subdomain-of, indicating that the pattern should be subdomain of the specified field. The subdomain-of match type is defined only for the host and tel address fields. For telephone numbers, a "subdomain" indicates a prefix: for example, the telephone number +1 212 939 7018 matches `subdomain-of="1212939"`.

String Switch: A *string switch*, indicated by the string-switch tag, allows string matching on free-form strings included in the call setup request. The defined string fields are subject, organization, and user-agent, mapped naturally from headers in the request. As with address matching, string outputs (indicated by string) can perform either exact is matching or substring contains matching.

Language Switch: A *language switch*, with tag language-switch allows a CPL script to make

decisions based on the languages in which the originator of the call indicated that they wished to communicate. Each language output gives an RFC 3066 [63] language tag representing a particular language.

Priority Switch: A *priority switch* (with tag priority-switch and output tag priority) allows a CPL script to make decisions based on the priority level (emergency, high, normal, or low) the caller specified for the call.

Time Switch: Finally, a *time switch* (tag time-switch allows call handling to be based on the time and/or date that the call arrives at the service execution environment. Each output condition in the time output tags specifies a recurring time interval, and the server determines whether the time of the call falls within that interval, in the time zone specified by the switch. The syntax of the recurring time intervals is based on the the specification of recurring intervals of time in the Internet Calendaring and Scheduling Core Object Specification, RFC 2445 [64]. This allows CPL scripts to be generated automatically from calendar books; it also allows the CPL to re-use the extensive existing work specifying time intervals. The full details of the recurring time intervals are quite complex and are not given here; however, see the example in Section 5.12.

5.6 Location Modifiers

The abstract location model of the CPL is described in Section 5.4.4. To summarize, a number of destination locations are stored by the execution environments, and the behavior of several of the signalling operations (defined in Section 5.7) is dependent on the current location set specified. CPLs use *location modifier* nodes to add or remove locations from the set.

Three types of location modifier nodes are defined.

Explicit Location: Explicit location nodes (which have the tag name location) specify a location literally, as a URL. They take one mandatory argument, the desired location represented as a URL, and optionally a priority value. Only one location may be specified per location node; multiple locations may be specified by cascading these nodes.

Location Lookup: Locations can also be looked up through external means, through the use of location lookups. The node, described by the tag lookup, specifies a source for the list of locations. Two types of sources are defined. The registration source indicates that the destination's list of registered locations should be added to the location set. Alternately, a list of locations can be specified by a URL, indicating an external source to query for locations. This node also has an optional attribute, timeout, which specifies the time the script is willing to wait for the lookup to be performed. Lookup has three outputs: success, notfound, and failure. Notfound is taken if the lookup process succeeded but did not find any locations; failure is taken if the lookup failed for some reason, including that specified timeout was exceeded.

Location Removal: A CPL script can also explicitly remove locations from the location set. The remove-location tag performs this action, and takes one tag arguments, location, specifying the location to remove.

5.7 Signalling Operations

Signalling operation nodes cause signalling events in the underlying signalling protocol. Three signalling operations are defined: "proxy," "redirect," and "reject."

Proxy: The proxy node is perhaps the most important node of the CPL. It causes the triggering call to be forwarded to the current location set, and allows the script to react based on the returned response. After a proxy operation has completed, the CPL server chooses the "best" response to the call attempt, as defined by the signalling protocol or the server's administrative configuration rules. If the call attempt was successful, CPL execution terminates and the server proceeds to its default behavior (normally, to allow the call to be set up). Otherwise, the next node corresponding to one of the proxy node's outputs is taken. The busy output is followed if the call was busy; noanswer is followed if the call was not answered before the timeout parameter expired; redirection is followed if the call was redirected; and failure is followed if the call setup failed for any other reason. If one of these conditions holds, but the corresponding output was not specified, the default output

of the proxy node is followed instead. If there is also no default node specified, CPL execution terminates and the server returns to its default behavior (normally, to forward the best response upstream to the originator). (CPL extensions to allow in-call or end-of-call operations — see Section 5.11 — would require an additional output, such as success, to be defined.)

Proxy has three optional parameters. The timeout parameter specifies the amount of time to wait for the call to be completed or rejected. The recurse parameter specifies a boolean value (yes or no) indicating whether the server should automatically attempt to place further call attempts to telephony addresses in redirection responses that were returned from the initial server. Finally, the ordering parameter specifies how the various locations in the current set should be tried: parallel indicates that they should all be tried simultaneously; sequential, that they should be tried one at a time, in priority order, until one succeeds; and first-only that the server should try only the highest-priority address in the set, and then follow one of the outputs. For sequential and first-only the priorities are either specified explicitly by the location node, or supplied by the external source of the locations provided by the lookup node.

Once a proxy operation completes, if control is passed on to other nodes, all locations which have been used are cleared from the location set. In the case of a redirection output, the new addresses to which the call was redirected are then added to the location set.

Redirect: The redirect node causes the server to direct the calling party to attempt to place its call to the currently specified set of locations. Redirect immediately terminates execution of the CPL script, so this node has no outputs and no next node. It has one parameter, permanent, which specifies whether the result returned should indicate that this is a permanent redirection.

Reject: The reject node causes the server to reject the call attempt. This immediately terminates execution of the CPL script, so this node has no outputs and no next node. This node has two arguments: status and reason. The status argument is required, and can take one of the values busy, notfound, reject, and error, or a signalling-protocol-defined status. The

reason argument optionally allows the script to specify a reason for the rejection.

5.8 Non-signalling Operations

In addition to the signalling operations, the CPL defines several operations which do not affect and are not dependent on the telephony signalling protocol.

Mail: The mail node causes the server to notify a user of the status of the CPL script through electronic mail. It takes one argument: a `mailto` URL giving the address, and any additional desired parameters, of the mail to be sent. The server sends the message containing the content to the given URL, and also includes other status information about the original call request and the CPL script at the time of the notification.

Using a full `mailto` URL rather than just an e-mail address allows additional e-mail headers to be specified, such as
`<mail url="mailto:jones@example.com?subject=lookup%20failed" />`.
Other URL schemes are not supported.

Log: The Log node causes the server to log information about the call to non-volatile storage. It takes two arguments, both optional: name, which specifies the name of the log, and comment, which gives a comment about the information being logged. Servers should also include other information in the log, such as the time of the logged event or information that triggered the call to be logged.

5.9 Subactions

XML syntax defines a tree of tags. To allow more general call flow diagrams, and to allow script re-use and modularity, the CPL defines *subactions*.

Two tags are defined for subactions: subaction *definitions* and subaction *references*. Subactions are defined by subaction tags. These tags are placed in the CPL before any top-level tags. They take one argument: id, a token indicating a script-chosen name for the subaction.

Subactions are called from sub tags. The sub tag is a "pseudo-node": it can be used anyplace in a CPL action that a true node could be used. It takes one parameter, ref, the name of the subaction to be called. The sub tag contains no outputs of its own; control instead passes to the subaction.

References to subactions must refer to subactions defined before the current action. A sub tag must not refer to the action which it appears in, or to any action defined later in the CPL script. Top-level actions cannot be called from sub tags, or through any other means. Allowing only back-references of subactions forbids any sort of recursion. Recursion would introduce the possibility of non-terminating or non-decidable CPL scripts, a possibility that was specifically excluded above by the CPL's design requirements. Every sub tag refers to a subaction ID defined within the same CPL script; no external links are permitted.

5.10 Default Behavior

When a CPL node reaches an unspecified output, either because the output tag is not present, or because the tag is present but does not contain a node, the CPL server's behavior is dependent on the current state of script execution. This section gives the operations that should be taken in each case.

No location modifications or signalling operations performed, location set empty: Look up the user's location through whatever mechanism the server would use if no CPL script were in effect. Proxy, redirect, or send a rejection message, using whatever policy the server would use in the absence of a CPL script.

No location modifications or signalling operations performed, location set non-empty: (This can only happen for outgoing calls.) Proxy the call to the addresses in the location set.

Location modifications performed, no signalling operations: Proxy or redirect the call, whichever is the server's standard policy, to the addresses in the current location set. If the location set is empty, return notfound rejection.

Proxy noanswer output, no timeout given: (This is a special case.) If the noanswer output of a proxy node is unspecified, and no timeout parameter was given to the proxy node, the call should be allowed to ring for the maximum length of time allowed by the server (or the request, if the request specified a timeout).

Proxy operation previously taken: Return whatever the "best" response is of all accumulated responses to the call to this point, according to the rules of the underlying signalling protocol.

5.11 CPL Extensions

Servers may support additional CPL features beyond those defined by the basic specification. A number of extensions have been informally suggested, including a means of querying caller authentication information provided by a call request; richer control over H.323 addressing; administrator-specific features; regular-expression matching for strings and addresses; control of presence and instant messaging events; use and manipulation of SIP caller preferences [65] information; and mid-call or end-of-call controls. In addition, the Language for End System Services (LESS) [16] is a package of CPL extensions for controlling Internet telephony end systems.

CPL extensions are indicated by XML namespaces [66]. Every extension has an appropriate XML namespace assigned to it, and XML tags and attributes that are part of the extension are be appropriately qualified so as to place them within that namespace.

A CPL server rejects any script which contains a reference to a namespace which it does not understand, and also rejects any script which contains an extension tag or attribute which is not qualified to be in an appropriate namespace.

As an aside, a syntax such as the one illustrated in Figure 5.3 has been suggested as an alternate way of handling extensions. In such a syntax, a script could be written so as to allow alternative behaviors depending on whether or not a given extension is supported. This would allow scripts to be uploaded to a server without requiring a script author to somehow determine which extensions a server supports. However, experience developing other languages, notably Sieve [39], was that this added excessive complexity to languages; in particular, the complexity

```
<extension-switch>
  <extension has="http://www.example.com/foo">
     [extended things]
  </extension>
  <otherwise>
     [non-extended things]
  </otherwise>
</extension-switch>
```

Figure 5.3: Rejected Alternate Approach to CPL Extensions

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <location url="sip:smith@phone.example.com">
     <redirect />
    </location>
  </incoming>
</cpl>
```

Figure 5.4: Example Script: Call Redirect Unconditional

of determining script correctness is exponential in the number of different extensions requested. The example's extension-switch tag could, of course, itself be defined as a CPL extension.

5.12 Examples

This section illustrates CPL-based services implemented by simple scripts.

5.12.1 Example: Call Redirect Unconditional

The script in Figure 5.4 unconditionally redirects all incoming calls to a single fixed location.

5.12.2 Example: Call Forward Busy/No Answer

The script in Figure 5.5 illustrates some more complex behavior. Incoming calls are initially proxied to one address; if that fails, an alternate destination is tried. This script shows how several outputs take the same action subtree, through the use of subactions.

```
<?xml version="1.0" ?>

<cpl>
  <subaction id="voicemail">
    <location url="sip:jones@voicemail.example.com">
      <proxy />
    </location>
  </subaction>

  <incoming>
    <location url="sip:jones@jonespc.example.com">
      <proxy timeout="8">
        <busy>
          <sub ref="voicemail" />
        </busy>
        <noanswer>
          <sub ref="voicemail" />
        </noanswer>
      </proxy>
    </location>
  </incoming>
</cpl>
```

Figure 5.5: Example Script: Call Forward Busy/No Answer

5.12.3 Example: Call Forward: Redirect and Default

The script in Figure 5.6 illustrates further proxy behavior. The server initially tries to proxy to a single address. If this attempt is redirected, a new redirection is generated using the locations returned. In all other failure cases for the proxy node, a default operation — forwarding to voicemail — is performed.

5.12.4 Example: Call Screening

The script in Figure 5.7 illustrates address switches and call rejection, in the form of a call screening script. Note also that because the address-switch lacks an otherwise clause, if the initial pattern did not match, the script does not define any operations. The server therefore proceeds with its default behavior, which would presumably be to contact the user.

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <location url="sip:jones@jonespc.example.com">
      <proxy>
        <redirection>
          <redirect />
        </redirection>
        <default>
          <location url="sip:jones@voicemail.example.com">
            <proxy />
          </location>
        </default>
      </proxy>
    </location>
  </incoming>
</cpl>
```

Figure 5.6: Example Script: Call Forward: Redirect and Default

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <address-switch field="origin" subfield="user">
      <address is="anonymous">
        <reject status="reject"
                reason="I don't accept anonymous calls" />
      </address>
    </address-switch>
  </incoming>
</cpl>
```

Figure 5.7: Example Script: Call Screening

5.12.5 Example: Priority and Language Routing

The script in Figure 5.8 illustrates service selection based on a call's priority value and language settings. If the call request had a priority of "urgent" or higher, the default script behavior is performed. Otherwise, the language field is checked for the language "es" (Spanish). If it is present, the call is proxied to a Spanish-speaking operator; other calls are proxied to an English-speaking operator.

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <priority-switch>
      <priority greater="urgent" />
      <otherwise>
        <language-switch>
          <language matches="es">
            <location url="sip:spanish@operator.example.com">
              <proxy />
            </location>
          </language>
          <otherwise>
            <location url="sip:english@operator.example.com">
              <proxy />
            </location>
          </otherwise>
        </language-switch>
      </otherwise>
    </priority-switch>
  </incoming>
</cpl>
```

Figure 5.8: Example Script: Priority and Language Routing

```
<?xml version="1.0" ?>

<cpl>
  <outgoing>
    <address-switch field="original-destination" subfield="tel">
      <address subdomain-of="1900">
        <reject status="reject"
                reason="Not allowed to make 1-900 calls." />
      </address>
    </address-switch>
  </outgoing>
</cpl>
```

Figure 5.9: Example Script: Outgoing Call Screening

5.12.6 Example: Outgoing Call Screening

The script in Figure 5.9 illustrates a script filtering outgoing calls, in the form of a script which prevent 1-900 (premium) calls from being placed. This script also illustrates subdomain match-

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <time-switch tzid="America/New_York"
        tzurl="http://zones.example.com/tz/America/New_York">
      <time dtstart="20000703T090000" duration="PT8H"
            freq="weekly" byday="MO,TU,WE,TH,FR">
        <lookup source="registration">
          <success>
            <proxy />
          </success>
        </lookup>
      </time>
      <otherwise>
        <location url="sip:jones@voicemail.example.com">
          <proxy />
        </location>
      </otherwise>
    </time-switch>
  </incoming>
</cpl>
```

Figure 5.10: Example Script: Time-of-day Routing

ing.

5.12.7 Example: Time-of-day Routing

Figure 5.10 illustrates time-based conditions and time zones. Calls placed on weekdays, between 9 AM and 5 PM in New York's time zone, are forwarded to the user's registered locations; calls at other times are directed to voicemail.

5.12.8 Example: Location Filtering

Figure 5.11 illustrates filtering operations on the location set. In this example, we assume that version 0.9beta2 of the "Inadequate Software SIP User Agent" is broken — in particular, it cannot talk successfully to one particular mobile device we may have registered under the URL "sip:me@mobile.provider.net". If the call is coming from the broken user agent, we remove the mobile device's location from the location set, and then allow call setup to proceed normally.

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <string-switch field="user-agent">
      <string is="Inadequate Software SIP User Agent/0.9beta2">
        <lookup source="registration">
          <success>
            <remove-location location="sip:me@mobile.provider.net">
              <proxy />
            </remove-location>
          </success>
        </lookup>
      </string>
    </string-switch>
  </incoming>
</cpl>
```

Figure 5.11: Example Script: Location Filtering

```
<?xml version="1.0" ?>

<cpl>
  <incoming>
    <lookup
       source="http://www.example.com/cgi-bin/locate.cgi?user=jones"
       timeout="8">
      <success>
        <proxy />
      </success>
      <failure>
        <mail
          url="mailto:jones@example.com?subject=lookup%20failed" />
      </failure>
    </lookup>
  </incoming>
</cpl>
```

Figure 5.12: Example Script: Non-signalling Operations

5.12.9 Example: Non-signalling Operations

Figure 5.12 illustrates non-signalling operations; in particular, alerting a user by electronic mail if the lookup server failed. The primary motivation for having the mail node is to allow this sort of out-of-band notification of error conditions, as the user might otherwise be unaware of any

```
<?xml version="1.0" ?>
<cpl xmlns="urn:ietf:params:xml:ns:cpl"
     xmlns:dr="http://www.example.com/distinctive-ring"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd
        http://www.example.com/distinctive-ring distinctive-ring.xsd">
  <incoming>
    <address-switch field="origin">
      <address is="sip:boss@example.com">
         <dr:ring ringstyle="warble" />
      </address>
    </address-switch>
  </incoming>
</cpl>
```

Figure 5.13: Example Script: Hypothetical Distinctive-Ringing Extension

```
<?xml version="1.0" ?>

<cpl xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd ">
  <incoming>
    <address-switch field="origin" subfield="user"
        xmlns:re="http://www.example.com/regex">
      <address re:regex="(.*\.smith|.*\.jones)">
         <reject status="reject"
            reason="I don't want to talk to Smiths or Joneses" />
      </address>
    </address-switch>
  </incoming>
</cpl>
```

Figure 5.14: Example Script: Hypothetical Regular-Expression Extension

problem.

5.12.10 Example: Hypothetical Extensions

The example in Figure 5.13 shows a hypothetical extension which implements distinctive ringing. The XML namespace "http://www.example.com/distinctive-ring" specifies a new node named ring.

The example in Figure 5.14 implements a hypothetical new attribute for address switches, to allow regular-expression matches. It defines a new attribute regex for the standard address

```
<?xml version="1.0" ?>

<cpl>
  <subaction id="voicemail">
    <location url="sip:jones@voicemail.example.com">
      <redirect />
    </location>
  </subaction>

  <incoming>
    <location url="sip:jones@phone.example.com">
      <proxy timeout="8">
        <busy>
          <sub ref="voicemail" />
        </busy>
        <noanswer>
          <address-switch field="origin">
            <address is="sip:boss@example.com">
              <location url="tel:+19175551212">
                <proxy />
              </location>
            </address>
            <otherwise>
              <sub ref="voicemail" />
            </otherwise>
          </address-switch>
        </noanswer>
      </proxy>
    </location>
  </incoming>
</cpl>
```

Figure 5.15: Example Script: A Complex Example

node. In this example, the global namespace is not specified.

5.12.11 Example: A Complex Example

Finally, Figure 5.15 is a complex example which shows the sort of sophisticated behavior which can be achieved by combining CPL nodes. In this case, the user attempts to have his calls reach his desk; if he does not answer within a small amount of time, calls from his boss are forwarded to his mobile phone, and all other calls are directed to voicemail. If the call setup failed, no operation is specified, so the server's default behavior is performed.

5.13 Design Evaluation

As discussion in Section 5.1, there were a number of design goals for the Call Processing Language. This section describes these goals more formally, and explains how the language presented in this chapter addresses them, where other design choices — including the other existing systems described in Section 3.4, and the rejected CPL design possibilities described in Section 5.1.1 — would not.

First of all, the CPL is designed to **control services in signalling servers**. As discussed in Section 2.2.1, the operations that a signalling server can perform are to proxy a call, redirect it, or reject it. This immediately motivates CPL's three signalling operations. Furthermore, it rules out a number of languages described in Section 3.4, which are designed for a telephony switch or end system, and as such can perform actions such as call initiation, bridging, or announcements, beyond those which are available to a signalling server. (CPL can be extended to perform such actions, as is done by LESS [16], but they should not be part of the core language.)

Secondly, the CPL must be **parsable and editable by humans and machines, by ordinary users and experienced programmers**. There are a number of consequences of this. It strongly recommends against the use of a general-purpose programming language. An arbitrary program written in a general-purpose programming language cannot be presented comprehensibly to a user who is not familiar with programming. While it is certainly possible for automated tools to produce and edit scripts in a subset of a language, programmers cannot be expected to write scripts which conform to this subset. Perl, Java, or Tcl code can be, essentially, arbitrarily expressive, and as such cannot be translated into a simple graphical representation of the behavior of a service.

Instead, this requirement indicates that the language should have a regular syntax, representing a form that is quite close to the higher-level semantics of a service. XML is not, to be sure, the only possible choice for this purpose; any reasonably regular syntax capable of representing trees or directed acyclic graphs could be used. However, the broad availability of well-tested XML parsers and of tools for manipulation of XML, and the wide prevalence of experience with the language, made it a better choice than using some other existing syntax or inventing one from scratch.

The choice of XML, which describes trees rather than directed acyclic graphs, led to the addition of subactions to allow tree branches to "merge." Alternately, a "go to" model could be used, in which node outputs can point to other nodes in the tree. This is topologically equivalent; assuming a node does not point to one of its parents, both models describe a directed acyclic graph. The subaction model was thought to be preferable; it makes it somewhat easier to ensure acyclicity, and is somewhat easier to represent. The general disfavor of "go to" in computer science [67] also played a role.

This requirement of **accessibility** also induces a number of CPL's other design choices. Services are described as a tree of operations, starting from the initial processing of a call, and ending with either its successful completion or its abandonment. As an alternative, a model was considered where events are triggered by call events — initial receipt of a call, receipt of a successful or unsuccessful response, etc. — in much the same way as SIP CGI (described in Chapter 4). However, initial attempts at service design quickly revealed this to be too complex for the ordinary user, who has difficulty keeping track of the service's state; it is much better for such state to be described automatically and directly by the service description.

Additionally, the CPL must be **safe for service providers**. This means that a broken or malicious CPL script cannot harm a provider's system. This further rules out general-purpose programming languages, in a number of respects. Obviously, users cannot be allowed to execute arbitrary programs on the CPL server. It is not enough, however, to run a service in a restricted environment such as a Java sandbox, since this does not, in itself, prevent a service from looping arbitrarily and thus consuming excessive amounts of CPU time. In order to improve predictability, as well as assisting with accessibility to ordinary users, it was thought to be better to design the language such that such loops are inherently impossible, rather than having the execution environment terminate long-running services.

Another requirement was that services described by CPL scripts be **protocol-independent**. Thus, the operations which can be performed by CPL scripts are those which are common to all signalling servers. Very few specific details of signalling protocols appear in the language, and those which do are not essential to the operation of the language. In addition to improving service portability, this also enhances accessibility, as users do not need to know

the details of network signalling protocols. For this reason as well, the CPL does not use specific details of, for example, the Intelligent Network model, as CCXML and SCML (described in Section 3.4) do.

CPL also must **interoperate with existing location mechanisms**. This is the primary motivation for the location set model. Early on, an approach was considered in which locations were explicit arguments to the proxy and redirect nodes. However, it was not at all clear how to associate this model with SIP's REGISTER method, through which a user's end points announce their locations to a signalling server. Thus, instead, the implicit location set was created, which allows REGISTERed locations to be added using the lookup node.

Finally, CPL needs to **interwork cleanly with other network protocols**. This led to the design of the time-switch node, which is based directly on the method of describing recurring intervals defined in RFC 2445 [64]. Since it is entirely likely that users will want to base their telephony service routing on their calendars, it was decided that the two languages should share a syntax. Unfortunately, this decision conflicts somewhat with some of CPL's other design goals: this syntax is complex and can be hard to understand, and in some pathological cases it can take quite a long time to determine whether an instant of time falls within a recurrence. However, compatibility was thought to be a more important issue, especially as the representation of recurring intervals is inherently complex, due to the nature of calendars and human behavior.

5.14 Implementation Experience

Numerous Internet telephony software vendors have implemented the CPL. The Columbia CINEMA server, of course, has an implementation; this implementation is described in some detail in Chapter 6. In addition, a large number of other vendors of Internet telephony servers have implemented the language. Notable products include dynamicsoft's AppEngine; Hughes Software Systems's SIP Server Framework; Indigo Software's SIP Server & SDK; IPTel.org's open-source SIP Express Router; and Ubiquity Software's SIP Application Server.

CPL is gradually increasing its presence in its intended niche of easy-to-use user-created services; it is perhaps somewhat held back by the fact that it has been unable until quite recently to advance to Proposed Standard status in the IETF because of a now-removed normative depen-

dence on SIP Caller Preferences and Callee Capabilities [65]. Also, unlike the uses of SIP CGI services described in Section 4.8.1, CPL's limited environment means that it is less able to be a platform for unusual new services beyond its original conception.

5.15 Conclusion

The Call Processing Language makes a good complement to the SIP Common Gateway Interface, providing a mechanism for users to create custom services. Its record of implementation indicates that service providers agree that it can provide an environment which they can reliably make available to their users. As such, it fills a very useful role in the taxonomy of Internet service creation mechanisms.

Chapter 6

Design and Implementation of the CINEMA Policy Framework

In order to implement user-location services in a SIP proxy server, a framework is needed on which to implement the services in a consistent manner. This chapter reviews a number of existing frameworks, and considers their strengths and weaknesses. It then presents a new one, designed for supporting numerous models of service creation in the specific context of the CINEMA SIP proxy server.

6.1 Introduction

The Columbia InterNet Enhanced Multimedia Architecture, CINEMA, is a project of the Columbia University Department of Computer Science which includes a number of SIP and RTSP [68] projects, including an RTSP server, an RTSP client library, a SIP unified messaging server, conferencing server, user agent, H.323 [14] gateway, and, most significantly for the purposes of this discussion, a SIP proxy server, SIPD.

SIPD is the CINEMA module which performs user location, and thus it is in SIPD that user-location services are implemented. The SIP Common Gateway Interface, described in Chapter 4, and the Call Processing Language, described in Chapter 5, have both been implemented in SIPD by the author. In addition, Java SIP Servlets [32] have also been implemented by Sangho Shin,

another member of the IRT research group. These environments differ in their features, capabilities, and execution models.

6.1.1 The Policy Framework

The policy framework is the mechanism by which SIPD determines how to handle a SIP request.

There are two sorts of criteria which must be taken into account when deciding how to process a SIP request. First, there is the type of the request. SIPD is able to fill a number of roles in a SIP network: it can be an inbound or outbound proxy server, a registrar, a redirection server, a subscription agent, or a user-agent server for failure conditions. Additionally, it can handle proxied requests statefully or statelessly, and it can handle routed requests. Each of these roles requires different processing of the message.

Secondly, some classes of requests can be handled by user-specific policies. SIPD supports the SIP Common Gateway Interface of Chapter 4, the Call Processing Language of Chapter 5, and Java SIP Servlets [32]. After determining the user whose policy applies to a request, SIPD consults its internal databases to determine what policy types the user has selected. Different users can select different policy types; at the same time the server can be executing a CPL policy for one user and a SIP CGI policy for another. (At present, SIPD does not support multiple policies being invoked for a single transaction; see Section 6.6.)

The design of a policy framework is complicated by the fact that the various user-specific policy languages have very different process models. SIP CGI works by executing a new subprocess every time a SIP message is received. Handling of other messages in the transaction is suspended until the subprocess exits and returns control to the proxy server. Java servlets, by contrast, instantiate a servlet object in its own Java thread; this object persists for the duration of the transaction, and periodically has its methods invoked when messages arrive. Finally, CPL is an abstract representation of the flow of control for a transaction. Some CPL nodes abstractly represent the process of forwarding requests and receiving their responses, but this complexity is hidden from the CPL programmer; the CPL implementation must keep its own state internally. Because of this complexity, a policy framework must be flexible enough to accommodate all these possible process models, and, for future extensibility, others.

6.2 Other Systems' Approaches to Policy Definitions

The problem of policy extensibility arises in many programming domains. For example, HTTP servers, telephone networks, and firewalls all must be extensible in order to allow new services and new service description types to be added to existing code.

In this section we review several such systems' approaches to policy definitions.

6.2.1 Apache

The Apache HTTP server [69] is one of the most commonly-used web servers on the Internet. A huge number of frameworks and programming environments exist to create web services — the dynamic handling of, and response to, HTTP requests — and Apache provides an extensibility architecture which allows such services to be added dynamically to the server. (The description here comes from the documentation of Apache 1.3; the documentation of version 2.0 has not yet been finished.)

The Apache API [70] allows extensibility by breaking request handling into a series of steps. These steps are:

- URI to filename translation;
- authentication ID checking (is the user who they say they are?);
- authorization access checking (is the user authorized *here*?);
- access checking other than authorization;
- determination of the MIME type, or internal pseudo-MIME type, of the returned object;
- "fixups" — a hook for possible future extension that don't fit well elsewhere;
- response transmission;
- request logging.

The Apache code consists of a number of modules, any of which may either be compiled into the server or dynamically loaded. Each of these modules contains a number of handlers

(pointers to functions), which correspond to the steps of request handling. Modules also contain additional information, such as configuration options specific to the module.

For every phase, the handlers for that phase for all applicable modules are invoked. A handler performs actions, and then returns one of three possible values: OK, indicating that the module handled the request, usually terminating the phase; DECLINED, indicating that the module declines to handle the phase, so that the another module may attempt it; or an HTTP error response, which aborts handling of the request.

Most handlers perform their actions simply by returning the appropriate status code and by modifying the internal data structure which describes the request. Response handlers are the exception: they must perform the necessary actions to write the response. They do this with relatively low-level code which writes HTTP headers and textual output to the response socket, forwards the contents of a file descriptor to a socket, or performs internal redirection, re-starting the process with another internal URI.

The means by which the server determines which modules' handlers to invoke depends on the phase. The earlier phases are configured on a server-wide or per-directory basis. Once the MIME type has been determined, however, the response transmission handler is chosen based on the determined MIME type. Special pseudo-MIME type handlers are defined for special services; for example, HTTP CGI handling is described by the MIME type "application/x-httpd-cgi".

As of version 1.3 of Apache, all requests are handled synchronously – request handling blocks waiting for handlers' responses. Multi-processing of requests is handled by having several Apache processes running simultaneously. (Threading has been added in Apache 2.0.)

The Apache Model and SIP

The Apache model, while quite flexible for its domain of applicability, is in several ways not directly applicable to the problem of SIP user location services.

First of all, SIP user location policies must describe the behavior of an entire SIP transaction, not simply for a request and response as in an HTTP server. A SIP transaction handler must be able to handle specially the initial request of a transaction, any subsequent requests of the transaction, the various responses to the transaction, and any timeouts. The Apache model is

not designed for this mechanism.

Secondly, policies for SIP servers are primarily indexed by users. A policy for a SIP server would not be determined based on the MIME type of the request or response — bodies of SIP requests, and their MIME types, are supposed to be ignored by SIP proxy servers, and response bodies are not even created by them — but rather would be determined based on the user addressed. Due to the structure of HTTP, the Apache model separates URI resolution from determination of the requesting user, but for SIP these two parts are integrated.

Finally, for proxies, unlike for servers, request-response transactions must await the arrival of responses from downstream servers. The Apache synchronous model would be quite inefficient for SIP proxy servers.

6.2.2 Intelligent Networks

The Intelligent Network model (IN) [13] is a model of the Public Switched Telephone Network which allows services to be controlled by Service Control Point (SCP), which can be remote from the Service Switching Point (SSP) which performs the actual call control.

The Intelligent Network model describes calls using a call model, defined as a finite state machine. The states in the finite state machine are referred to as Points in Call, or PICs. The transitions between these states are Trigger Detection Points, or TDPs.

A set of TDPs can be provisioned for a specific call. When a provisioned trigger point is hit, the SSP queries the SCP for instructions on what actions to perform next. The SCP responds with the next PIC to be entered, along with any associated parameters that are required, such as a destination number. The SSP then carries on processing the call starting from the new PIC state.

Intelligent Networks and SIP

While work has been done to integrate existing Intelligent Network models with SIP [71], the IN model is not generally sufficient to provide all the SIP service policy environments that we wish to support in a general SIP proxy.

First of all, SIP does not have a standardized fixed call model. The various service execution environments to be supported all define, implicitly or explicitly, their own models of how

SIP call processing works. Thus, creating a call model which unites all the processing models would be both difficult and lacking in extensibility.

Furthermore, SIP transactions cannot easily be defined by only a single state machine. Because SIP supports forking, a state machine for SIP would have to describe many different possible orders in which the responses for the various branches could be received; this is both trouble-prone and unnecessary, as policies can easily keep their own state in more sophisticated and appropriate ways.

The specific call models used by IN also tend to be inappropriate for a SIP proxy server. Some of the states they describe, e.g‘‘ringing,” are not states that are often relevant for a SIP proxy server. Worse, these call models have action states such as “collect digits,” which cannot be performed by a SIP proxy server at all; a SIP proxy server receives the call signalling only, whereas in SIP dialed digits travel with the media, using RFC 2833 [72]. While alternate models which avoid these issues could conceivably be designed, and so describe SIP proxy features more appropriately, this is largely unfeasable for the reasons described above.

6.2.3 Firewalls and Application-Layer Gateways

Firewalls are another instance of services which often need application-specific extensibility. While many network transaction services can be supported solely through transport-level decoding of network flows — for example, policy rules that allow outgoing TCP connections and UDP request/response pairs — many others cannot, notably SIP itself if standard RTP is used for its media. Typically, this is resolved with a packet routing model. Packets destined for these special services are routed specially, being directed to applications running on the firewall host rather than being forwarded normally or dropped. These firewall applications then can re-write and forward traffic as necessary. For example, a SIP application can receive SIP packets, search them for SDP bodies, and based on the session description open ports to allow incoming RTP packets. This model cleanly separates the low-level packet forwarding services from the application services, but it does not easily generalize to services operating at other layers in a network stack.

6.3 CINEMA Policy API Architecture

The CINEMA project implements user location services by using a *policy API*. The CINEMA policy API provides the interface between LIBSIP, the library which handles and processes SIP transactions, and the higher-level code using LIBSIP, such as SIPD. These actions are performed by means of collections of functions known as *policies*.

Every SIP transaction has two policies associated with it. One, the *transaction policy*, controls the standard handling of the transaction, given the type and context of the request. Some example of transaction policies are outbound proxy, inbound proxy, register, and subscribe. The various transaction policies are described in detail below.

The second, higher-level type of policy is the *user policy*, which implements specific user features. Not every transaction has a user policy; if there is no user policy, the transaction policy is simply executed directly. Typically, a user policy implements a feature-implementation language, such as SIP CGI, CPL, or SIP Servlets. These user policies do not typically make transaction handling decisions on their own, but simply translate and relay the state of a transaction to the user execution layer. However, one special user policy, the "stepped proxy" policy, implements its own decision algorithm directly; see Section 6.5.4 for more details on this user policy.

The portion of LIBSIP which provides the policy API is known as the *policy core*. The policy core invokes policy functions and provides the basic state machinery of policy invocation. The flow of control passes back and forth between LIBSIP and the policy code frequently; the policy core calls the specific policy's individual functions, and those functions in turn call functions in LIBSIP to perform the actual protocol actions to be performed. (The method by which the policy core itself is executed is described in Chapter 7.)

6.3.1 Policy Methods

Policies have six methods, called at various times during the course of a SIP transaction. Every policy method is represented as a pointer to a function.

A SIP transaction, for these purposes, extends from the arrival of an initial SIP request, through all handling and proxying done for it, and ends with the transmission of its final response, ACK, and any response retransmissions.

The two simplest methods are `init`, invoked when a transaction begins, and `cleanup`, invoked when it is terminated. These functions are responsible for allocating and de-allocating, respectively, the policy's transaction-specific data and resources.

Three methods are defined for SIP requests and responses. First, `handle_initial_request` is invoked when the request starts. It is responsible for performing the initial actions required by the policy, such as forwarding the request or sending a response. Secondly, `handle_subsequent_request` is invoked when a subsequent request for a transaction, such as CANCEL or ACK, is received. It is responsible for handling these requests as appropriate — canceling outstanding proxied requests, or forwarding ACK messages. Finally, the `handle_response` method is invoked when a response to a request that the policy forwarded is received. The policy may choose to forward the response back to the initial requester, to save it for future consideration, or to invoke additional methods.

The sixth method is `timeout_expired`. Policies may set timeout values, if they wish to be invoked in the future regardless of what SIP messages are received.

The rules for when policy methods are invoked are different for the different classes of policy methods. The `init` and `cleanup` methods are always invoked for both types of policies for every transaction. The `timeout_expired` method is invoked only for the policy which set the appropriate timeout. The interesting case is for the three message-handling methods. In each of these cases, the user policy's method is invoked first. The user policy can do one of three things: it can indicate that it has handled the request; it can indicate an internal error; or it can *defer* to the transaction policy. The transaction policy's method is only invoked in the case when the user policy has deferred, or when a transaction has no user policy at all. Transaction policy methods, by contrast, may not defer; they may only return success or failure.

Because a user policy may defer in some situations but not others, transaction policies must be written carefully. They base their behavior on the current state of the transaction, not on their own previous actions; in fact, they must be able to handle correctly transaction states which they could not themselves have entered.

6.3.2 How Policy Methods Perform Actions

Policy methods perform actions by invoking LIBSIP functions. There are multiple actions possible that a policy may invoke. It may:

- Cause the request to be proxied to a policy-chosen destination, starting a new SIP branch. This causes LIBSIP to initialize all the standard SIP client machinery which is used to send the request to the new location.
- Cause locally-created responses, such as redirections or failure statuses, to be sent back toward the request originator. The policy may, optionally, add additional custom header fields to the response, or remove some of the standard header fields from it.
- Cause received responses to be sent back toward the request originator. The policy may also modify or edit these responses.
- Cause subsequent requests such as ACK to be forwarded towards existing SIP branches.
- Cancel existing SIP branches.
- Set or cancel a timeout for itself. This will cause (or prevent) the policy's `timeout_expired` method to be invoked at some point in the future.
- Indicate that it is done, and does not wish to be invoked again for this transaction, other than for `cleanup`. This implicitly causes all further methods for this policy for this transaction to be implicitly deferred or handled as no-ops.

6.3.3 Utility Functions for Policies

The actions described in Section 6.3.2 allow quite low-level control over the course of a SIP transaction. In some cases this is necessary. However, in the process of implementation, we discovered that many policies, and particularly many transaction policies, share many aspects of their behavior, and thus ought to share code. As such, we have defined a number of utility functions for these common cases.

The function `ProxyDispatchResponse` handles the receipt of a response by a transaction policy, based on the type of the response. Informational (1xx) responses are sent back to the request originator, if the transaction has not yet completed. Success (2xx) responses are always sent back to the sender. Redirection (3xx) responses are, depending on server configuration, used to invoke new server branches to the destinations specified in the response. All other responses are placed into a *response pool*, where they can be stored until all the transaction's branches have completed. The specific redirection behavior can also be directly invoked, regardless of the server's configuration, by the function `Do3xxRecursion`.

The function `PickBestResponse` picks the "best" response from this response pool, using the choice algorithm described by the SIP specification, and returns it to the policy. The function `SendBestResponse`, higher-level than this, causes this best response to be sent back to the client.

Subsequent requests — CANCEL and ACK — can be handled by the function `StandardSubsequentRequestHandling`. This function cancels any outstanding proxied requests when CANCEL is received, and forwards ACK messages to the appropriate branches when ACK is received. For policies which need special handling for these subsequent requests, the standard behavior for each of these can be invoked independently by the functions `CancelUnfinishedBranches` and `ForwardAckToBranches`, respectively.

6.3.4 How the policy core is started and completes

Once LIBSIP has parsed a request and determined that it belongs to a new transaction, it invokes the calling environment by executing the function `SIP_OnIncomingRequest`. It is then this function's responsibility to choose the policies for this transaction, and start the policy core. It does this by invoking the function `execute_policy`.

The transaction ends some amount of time (usually 30 seconds, but this is configurable) after one of the policies causes a final response to be sent. At this time both policies' `cleanup` methods are invoked.

In the current architecture, the function `execute_policy` does not return until the transaction has completed. At this time, `SIP_OnIncomingRequest` returns as well.

6.4 Example Policy Flows

In this section we will illustrate the flow of control in SIPD in several simple scenarios.

6.4.1 A Simple Policy Flow

Figure 6.1 shows a simple example. In this example, a request arrives at SIPD for a user who is not currently registered, and who has no user-specific policies installed. SIPD processes the request, and then responds with the status code 480 Temporarily Unavailable.

Specifically, after the request has been parsed and validated, LIBSIP invokes SIPD's `SIP_OnIncomingRequest` function. This function checks the type of the request, and looks up the user's policy preferences. In this case, this is an incoming request, and the user has no policy preferences specified. The transaction therefore gets the "incoming" transaction policy and no user policy.[1] The policy state machine is started with the function `execute_policy`.

Since there is no user policy, the policy state machine executes methods only for the transaction policy. First, the transaction policy's `init` method is invoked, which allocates policy-specific data. Then, the policy's `handle_initial_request` method is invoked.

Since the user has no contact information registered, the policy determines that 480 Temporarily Unavailable is the correct SIP response code. It invokes the LIBSIP function `policy_send_new_response` to send this response.

Since a final response has now been sent for the transaction, the policy core starts a final timeout timer. Once the timer has expired, the policy core invokes the transaction policy's `cleanup` function, and then frees all transaction-related state.

6.4.2 A User Policy Flow

Figure 6.2 is a more complex example, in which a CGI script is invoked. The CGI script proxies the request to a single location.

The initial stages of the flow are the same as that of the previous flow up through the invocation of `SIP_OnIncomingRequest`. At this point, this flow determines that the CGI

[1] Actually, the user would get the "stepped proxy" policy in this case; this example is simplified for expository purposes. See Section 6.5.4.

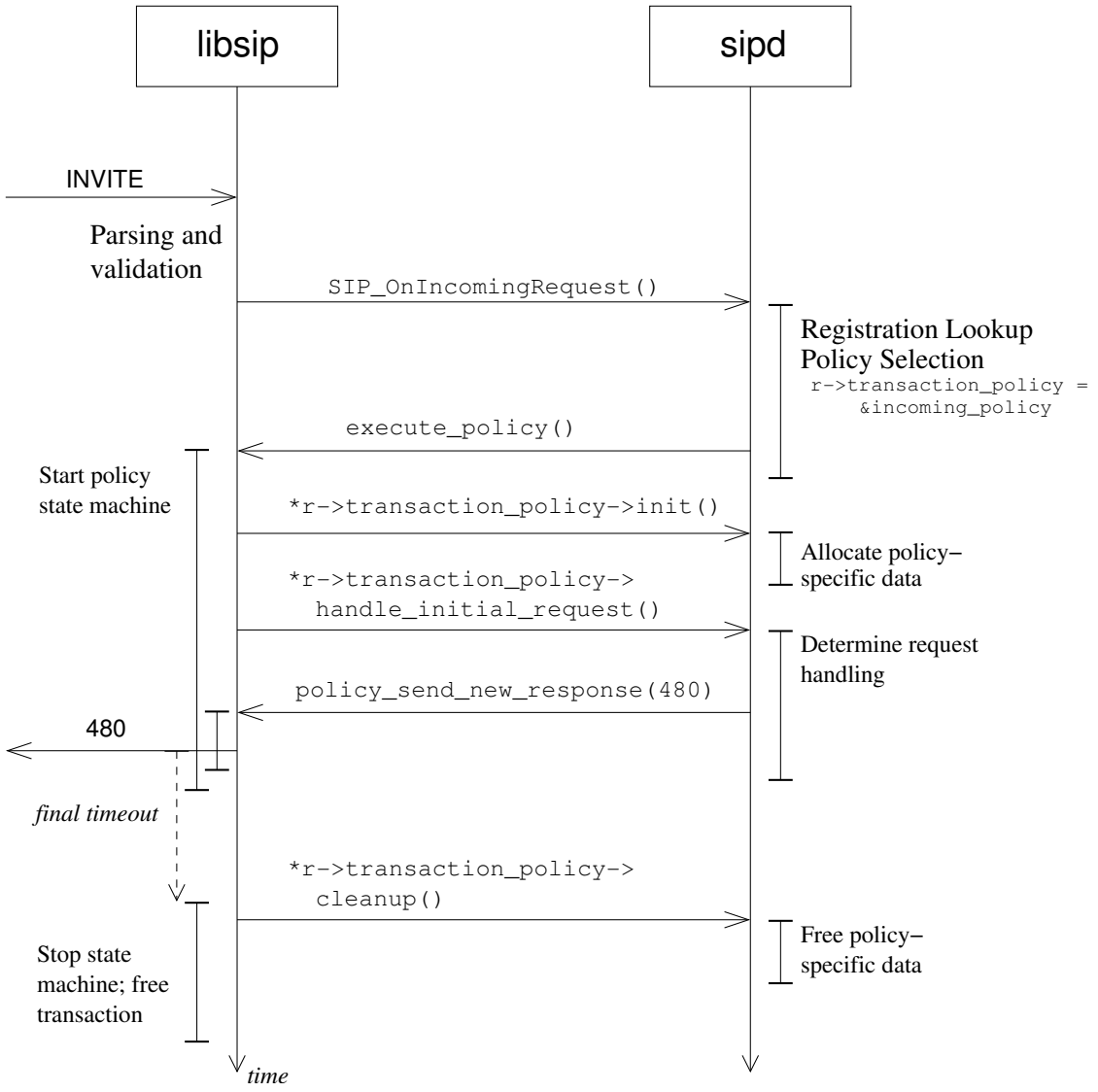

Figure 6.1: SIPD policy flow for 480 Temporarily Unavailable

policy should be used as a user policy, and this argument is then passed to `execute_policy`.

When the policy is executed, both policies' `init` methods are invoked. The user policy's `handle_initial_request` method is then invoked. Since this is the SIP CGI policy, it invokes the CGI script, and performs the appropriate action. In the example, the script returns CGI-PROXY-REQUEST, but does not return CGI-AGAIN.

By the definitions of SIP CGI request handling, this indicates several things. The initial

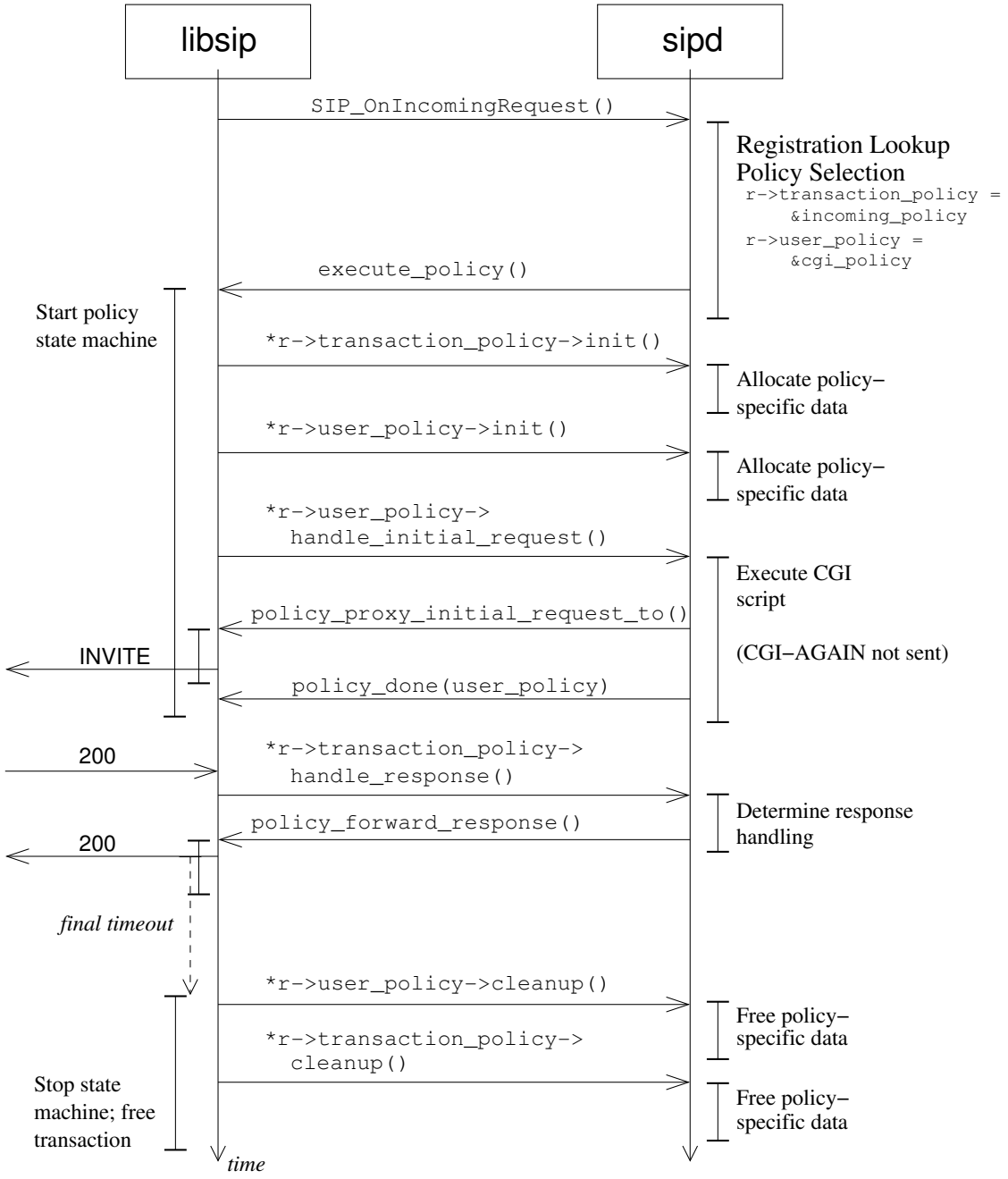

Figure 6.2: SIPD policy flow for a CGI invocation

request should be proxied to another, given, location; no “default action” should be taken for the handling of the initial request; and the SIP CGI should not be reinvoked for this transaction. These behaviors map closely onto policy actions. The CGI policy, first of all, invokes the function

`policy_proxy_initial_request_to` to forward the request to another location. It also invokes `policy_done` to indicate that it does not need to be reinvoked, and it returns *ok*, rather than *defer*, to indicate that it has completed the handling of the initial request. Because it returns *ok*, the transaction policy's `policy_proxy_initial_request_to` method is not invoked.

Eventually, a 200 OK response arrives for the request. Because the user policy has invoked `policy_done`, it is not reinvoked in this case. Instead, only the transaction policy's `handle_response` method is invoked. Because this is a 200 response, the response is immediately forwarded with `policy_forward_response`. This starts the final timeout. When the final timeout expires, both policies' `cleanup` methods are invoked before the transaction is freed.

6.5 Implementation Notes on Specific Policies

6.5.1 SIP CGI

SIP CGI is the simplest of the three implemented policies to implement from an architectural standpoint, as it is the lowest-level of the three. The model of the policy methods was designed with SIP CGI in mind, as this policy requires quite low-level control over the transaction state. In general, the three message-based policy methods directly invoke a CGI executable, and translate the response back into the appropriate API calls.

The SIP CGI policy's `init` method ensures that the policy script is marked as an executable file on the disk. Other than that, the `init` and `cleanup` methods only create and clean up internal state, such as the table mapping response tokens to responses.

6.5.2 Java Servlets

Java SIP Servlets are fairly similar in terms of message invocation to SIP CGI. Rather than invoking a script, however, a Java object is instantiated and its methods are invoked. The Java object is created by `init`, and destructed by `cleanup`.

One complication in the handling of Java servlets is that Java code can start additional threads. This is the standard way that servlet code handles "no-answer"-style timeouts. The

CINEMA code is designed to be multi-threaded, and thus this does not present a problem; threads outside the context of the policy core can safely invoke operations on the transaction.

6.5.3 CPL

CPL, by contrast, is a much higher-level representation of the state of a call. Effectively, all SIP messaging is encapsulated within the CPL `<proxy>` node, which abstractly represents the act of sending messages to remote locations, receiving all the responses, and determining which of the responses was the best; the next CPL action is taken based on the class of the best response. (The CPL nodes `<redirect>` and `<reject>` also initiate SIP messaging, but these both end the transaction, and so are not nearly as complex from an implementation standpoint.) CPL is implemented as a state engine which walks the CPL tree. When a `<proxy>` node is encountered, the state engine temporarily returns, with a value which indicates the locations to which the CPL wishes to proxy the request. Once the proxied locations have been contacted and the best response determined, the CPL engine is re-invoked, with this status code of the best response.

6.5.4 Stepped Proxy

For incoming calls, when a user has no specific user policy selected, a special policy known as the "stepped proxy" is used instead. This is designed to provide a simple form of call-forward-no-answer service, based on the preference values ("q" parameters) for the user's registered locations, which range from 0.0 to 1.0. The stepped proxy simultaneously proxies the request to all the contact addresses with q values within a certain range (by default, 0.05), and then waits 30 seconds (using the policy's timeout action). If no response has arrived within this time, a group of locations with lower header values are tried, until all the locations have been consumed.

This is implemented as a user policy because its behavior would tend to be incorrect when it interacted with policies such as CPL or CGI. These policies are not expecting the proxy action to some registered locations to be deferred after the initial request handling. Thus, this policy is used as an alternative to user-specific policies.

6.6 Further Development

The architecture described in this chapter is sufficient for what it does, but it has a few limitations which could be addressed in further development.

6.6.1 Multiple Policies for One Transaction

The current policy architecture supports only two policies for a transactions — the user policy and the transaction policies. There are a number of scenarios in which this is insufficient.

First, it is possible that a user could wish to have services defined in multiple policy execution languages; for instance, a user could have both a SIP CGI script and a CPL script defined for the same calls. Second, an administrator might wish to use one of these policy environments to impose site policy on all calls traversing the server. Third, and most complicated, policies could be invoked for multiple users. A user can proxy a request to another user served by the same server; or one user could have an outgoing policy, and another user an incoming policy, when both users are handled at this server.

This raises several issues. For policies of a single user, or of a user and an administrator, ordering issues arise. There must be a way for the user to indicate that certain policies should be invoked before or after other ones. This is not as much a concern for multi-user scenarios, as these have a natural ordering, but in this scenario a more complex issue arises: it is not always possible to know ahead of time whose policies are going to be invoked. Since one user's policy may change the destination of the request, the other policies must be determined dynamically; and since one policy may proxy to multiple other users, a whole tree of policies may result. The current architecture is not well-equiped to handle this complexity — functions which send requests and responses would have to be adapted to forward them to other policy handlers instead.

Fixing these issues should not require any changes to the policy API itself; the API architecture does not assume excessively much about what is causing the policy to be invoked. However, these fixes *will* require significant changes to the policy core; for example, the meaning of `policy_proxy_original_request_to` can now mean "start a new policy handler," and there will need to be a data structure to arrange policies in a tree.

6.7 Conclusion

Extensible service architectures require a generic API which allow a variety of services to be plugged in. A number of such architectures have been defined in the past, but none of the ones we have investigated are precisely applicable to the needs of user location services in a SIP proxy server.

The CINEMA policy architecture, by contrast, is well-suited to the needs of services in a SIP proxy server. We have implemented three major service creation platforms on it: SIP CGI, CPL, and Java SIP Servlets. Some further development will be needed to handle some of more complex cases, and some minor changes will significantly boost resource utilization efficiency. However, the current architecture solves the basic problem areas well.

Chapter 7

Design and Implementation of the CINEMA Reactive System Model

For network services, a naïve thread implementation can be a substantial impediment to server performance. Previous versions of the Columbia CINEMA SIP proxy server had their throughput limited by the architectural decision to use a single thread per server transaction. This chapter discusses the architectural drawbacks of this, and explains a generally-applicable new architecture, called *reactive systems*, which significantly improves throughput.

7.1 Introduction

Because SIP messages can be transported over unreliable protocols such as UDP, and because the decision to answer a call is made by humans, who may take many seconds or minutes, the SIP protocol has a number of fairly complex retransmission and timeout rules for its requests and responses, to determine when a message might need to be retransmitted or when a request can be abandoned. Proxy servers, as they both receive and send messages, must handle both ends of these transactions simultaneously, for a single transaction.

Proxy servers can operate in two modes — "stateful," in which they remember information about a transaction until it has received a final response, and "stateless," in which messages are forgotten after they are forwarded. Stateless mode induces much less of a load on the server —

in this mode, for example, the retransmission rules can be disregarded, as either end point is responsible for retransmission — but it has many fewer capabilities. For example, if a proxy server wants to "fork" a request, that is to send a single request to multiple downstream destinations, it must be stateful; similarly, a proxy server which performs any nondeterministic routing, such as a randomized call distribution center, cannot be stateless.

Because of the nature of proxying, most of the time a stateful SIP proxy server spends on a transaction will be taken up with waiting — waiting for a response to a request, waiting for a retransmission timer, or waiting for a final timer which indicates that other entities retransmissions have ceased, and it is safe to delete the transaction state.

The Columbia InterNet Enhanced Multimedia Architecture, or CINEMA, is a project of the Columbia University Department of Computer Science which includes a number of SIP and RTSP [68] projects, including an RTSP server, an RTSP client library, a SIP unified messaging server, conferencing server, user agent, H.323 [14] gateway, and, most relevant for the purposes of this discussion, a SIP proxy server, SIPD. This paper discusses the architecture by which stateful proxying is implemented in SIPD.

7.2 The Initial CINEMA Architecture

A proxy server needs to be able to process many requests and responses simultaneously. It also needs to be able to handle retransmission of every request it is sending further downstream. Up through version 1.21, the CINEMA SIPD server accomplished this in a straightforward manner: each message, and each client transaction, was assigned its own thread. (The server used POSIX threads [73] to implement our thread calls, with a thin wrapper for Windows systems.)

More specifically, the SIPD server had threads listening for received TCP and UDP messages. When a complete message was received over either of these protocols, a new thread was allocated for it. This thread parsed it, and then determined if the message was a part of an existing transaction.

If the message was not part of an existing transaction, the thread became the *transaction thread* for that transaction. This thread was responsible for determining how the request would be handled. It would determine the user's identity and registered locations and invoke any policy

handlers (described in Chapter 6) the user had requested for user-specific features such as SIP CGI (Chapter 4) or CPL (Chapter 5), so as to determine the downstream locations to which the request should be forwarded. It would then initiate a handler function for each of these downstream locations. After doing this, the transaction thread would enter a timed wait state, waiting for some other thread to notify it of an incoming message, or for its own expiration timer to expire. Each of the downstream location handlers was itself was assigned a thread, known as a *client thread*. This thread was responsible for resolving the destination location's URI into a network address, building the forwarded request, and transmitting it over the network. Like the transaction thread, after sending its message the client thread would enter a timed wait state, waiting for retransmission timers or further messages.

If a received message, instead, was determined to be part of an existing transaction — if it was a received response, a retransmitted request, or an acknowledgment message — the parser thread would instead notify the client thread (for responses) or transaction thread (for retransmissions and acknowledgments) about the message. These threads would then update their states and timers, and if necessary take further actions. Client threads forwarded significant responses to the transaction thread, and the transaction thread invoked further policy handlers, started additional client threads, or forwarded responses back to the sender. Once the transaction was complete, the transaction thread would wait for all the client threads to complete, and then free the transaction data and terminate.

7.2.1 Bottlenecks in this Architecture

This architecture, though straightforward, proved to be inefficient. The primary reason for this as that almost all the threads described above spent most of their time waiting. Client threads waited for responses; transaction threads waited for interesting messages from client threads.

Initially, a waiting thread might not seem like a burden. It does not, after all, occupy any CPU time; any decent thread scheduler should be able to figure out that a thread waiting for a timer or a condition variable does not need to be scheduled. However, this view overlooks the actual implementation details of threading systems.

Many thread implementations have implementation limitations on the number of threads

they can support simultaneously (for example, Solaris 8 is limited to 3,398 simultaneous threads [74]). More significantly, however, every thread must, inherently, have space allocated for its stacks. Stack requirements can be large — a typical implementation allocates a megabyte of virtual memory per stack — and in common usage, if numerous objects are allocated on-stack, or if recursion is deep, a fair amount of this memory is actually accessed frequently. The amount of locality of reference of constant thread-swapping is also poor, resulting in bad cache and virtual memory performance. System profiling confirmed that CINEMA's thread model — which allocated at least two threads for every outstanding transaction — was a major bottleneck in the processing of SIP transactions.

7.3 Related Work

Matt Welsh *et al* [75] have studied a number of approaches to implementing Internet services. Their paper discusses a number of models. One, which they call "thread-based concurrency," is essentially the old CINEMA model: each request gets a thread to process it. In a variant of this, "bounded thread pools," the number of simultaneously-processed requests is bounded. (In the old CINEMA model, some, but not all, processing threads were allocated out of a thread pool of bounded size; this allocation was fairly ad-hoc and inconsistent.)

Another model described by Welsh *et al*, "event-driven concurrency," eschews the use of threads entirely, and uses an event loop that continuously processes events. Event processing is dispatched to components which handle them asynchronously. The primary limitation of this model is that event processing operations must not block. The paper largely discusses implementation of web servers, and the non-blocking requirement is particularly difficult, on most operating systems, for disk I/O.

Welsh's paper instead proposes a new model, their "Staged Event-Driven Architecture" (SEDA). In the SEDA model, events are organized into *stages*. Each stage has its own thread pool, and events are passed between stages through event queues. This allows a limited amount of blocking; a number of threads process events for a single stage, so blocking operations can be handled.

The SEDA model is not directly applicable to SIP, since SIP has transactions which must

process multiple messages, with long waiting times in between. Additionally, the need to evolve the existing CINEMA code base limited somewhat the full scope of the possible changes that could be made to its architecture. Nonetheless, the new CINEMA architecture draws upon SEDA's models in a number of ways, as explained below.

7.4 The Reactive System Model

In order to remedy the inefficiency of the previous CINEMA thread model, it was necessary to redesign the model in which CINEMA executes events. There were a number of design criteria which needed to be met in this redesign. The basic requirement was simple: as much as possible, a thread shouldn't be occupied waiting for an external event. Instead, the necessary state should be stored in some local data, and the thread should switch to processing some other transaction. However, we still want to be able to take advantage of multiprocessing on multi-CPU systems. Additionally, due to limitations of system APIs, it is not always possible to write code to completely avoid waiting: for example, the common POSIX API for DNS host name resolution (`gethostbyname_r`) is blocking. Therefore, the system should be designed so that multiple threads can still execute at once, but threads can be re-used for multiple transactions.

Events executed by the server are generally triggered by two types of occurrences: *messages* are received, and *timers* expire. The new event model had to be structured to support both of these. As a final requirement, we wanted to minimize the necessary changes to existing code, as much as possible; while, clearly, code written as a loop around the POSIX thread call `pthread_cond_timedwait` would have to be restructured, we wanted the changes to be relatively minimal outside of this.

Thus, beginning with CINEMA version 1.22, the event model was altered. We called the new event model a *reactive system*. A reactive system is a self-contained object which receives messages, sets timers, and can be shut down on demand. Common code handles the delivery of messages and timers, and schedules the execution of events; each reactive system, meanwhile, has its own code to handle the execution of events. (The term "reactive system" is from computer science theory, indicating a logic system for which the inputs are not all available in advance, but instead arrive in endless and perhaps unexpected sequences. This is as opposed to a "trans-

formational system," which has all its inputs ready when it begins processing [76]. We used this terminology, rather than the perhaps more expected "state machine," for two reasons. First of all, the common code does not provide any sort of state-transition framework; it simply handles the delivery of messages and timers. Secondly, reactive systems are not limited, computationally, to the capabilities of a finite state machine; they are implemented with general source code. For both these reasons, we felt that the term "state machine" would be misleading, so we avoided it.)

7.4.1 Details of the Reactive System Approach

A reactive system is realized as an object[1] which implements implements five methods: `start`, called after the system is created; `message`, which is called on the receipt of a message; `timeout`, which is called when a timeout has expired; `system_deleted`, which is called when some other system is deleted, and that deletion indicated that this system wanted to know when that happened; and `destruct`, which frees the system's internal data. Each of these methods can perform any action; while, for design reasons, these actions should be non-blocking, they can, if necessary, perform operations that block, such as DNS lookups or the acquisition of mutexes.

Reactive system objects are entirely serialized. While a method is executing, other messages, timeouts, and actions are queued and will not execute until the previous method has finished. Messages and timeouts are handled in a strictly first-in-first-out order, except that timeout events may be dropped if they are canceled.

Several operations may be performed on a reactive system. No operation causes any actions to be taken synchronously; this would violate the serialization requirement. Instead, each operation schedules an event, which is separately scheduled for execution.

The first type of operation is that a reactive system can be created. Every newly-created reactive system is of a specific type, depending on the methods it will execute; there is no sense in which a "generic" reactive system can be created. When a system is created, an event executing

[1]The code implementing reactive systems is written in C. However, for simplicity of presentation, this paper uses C++ / Java terminology: reactive systems are "objects," which implement "methods," etc. The actual code uses the C equivalents of object-orientation: reactive systems have structures of pointers to functions as a "method table" and opaque `void*` pointers to their internal data.

its `start` method is scheduled.

Secondly, a message can be sent to the reactive system. "Messages" are opaque from the point of view of the reactive system code proper; their content is established by the user of the reactive system code. Messages include external network messages, but are not limited to them; they can also indicate internal status indicators or control commands being passed to or between reactive systems. This operation enqueues an event to execute the system's `message` method.

Thirdly, a timer can be set, cleared, or rescheduled. A timer, upon expiration, will cause the reactive system's `timeout` method to be scheduled. A reactive system may have multiple pending timers simultaneously, each identified with a unique timer ID. If a timer is cleared or rescheduled, pending executions of `timeout` methods are purged from the list of pending events.

Finally, a reactive system can be deleted. A reactive system is not deleted immediately. Instead, a deletion event, which will execute the `destruct` method, is placed on the reactive system's event queue, just as with every other type of event. Previously pending events are still executed, though a flag is set to prevent further events from being scheduled for the reactive system. Furthermore, in some cases, one reactive system may need to know when another one has been deleted. For instance, if two reactive systems share some data, and one of them is responsible for freeing the data on cleanup, it must know that the other system has been successfully deleted before it can safely free the data. For this reason, there is an optional second argument to reactive system deletion: another system to be notified. If this option is provided, then once the `destruct` method of the reactive system being deleted has completed, a `system_deleted` event is enqueued for the other reactive system.

Operations that execute on a reactive system are executed out of a *thread pool*. The system maintains a common pool of threads (by default, 128 of them, though the value is configurable) which execute reactive system events. Unlike the SEDA architecture [75], a single pool of threads is maintained for the entire system; thus, processing is automatically allocated to whatever stage of the system currently most needs it, without explicit event re-scheduling needed. (If the queue of events waiting for the thread pool reaches an administrator-specified threshold, new requests can be rejected by the system.)

In addition to the reactive systems themselves, there are also low-level interfaces available

for event scheduling and timers. This allows the code to allocate an event to a thread in the thread pool, or to schedule a timer, outside the context of a reactive system. The number of reactive systems that can be running simultaneously in a server is limited only by available memory.

The reactive system architecture is quite generally applicable; it should be useful for any network server that has non-trivial, multi-stage processing. It could be applied, for example, to web servers or proxies, or mail servers.

7.4.2 Use of Reactive Systems in the New CINEMA Architecture

In versions 1.22 and later, SIPD uses reactive systems to implement SIP transactions. In order to avoid excessive modifications to the code, the model is conceptually somewhat similar to the older structure, described in Section 7.2; however, the new architecture uses reactive systems where the old one uses threads. There are two broad classes of reactive systems in CINEMA: *transaction* systems and *client* systems, which fill the roles of the old transaction and client threads, respectively. Every (non-retransmitted) SIP request causes one transaction system to be created; if the request is proxied, client systems are created for each proxied destination.

The new architecture, like the old one, has threads listening for received SIP messages over TCP and UDP. When a complete message is received, an event is scheduled for the message using the low-level event API. This event then parses the message, and determines if it is part of an existing transaction. (This does not occur within the context of a reactive system.)

If the message is not part of an existing transaction, a new transaction reactive system is created. The transaction reactive system's `start` method then performs the initial transaction processing actions, such as determining the user's identity and registered locations and invoking policy handlers. Once the downstream locations are determined, client reactive systems are created. The client reactive systems' `start` methods then each resolve the destination location's URI into network addresses, build the forwarded request, and transmit it over the network.

If, on the other hand, the initial parsing thread determines that the received message is part of an existing transaction, it sends a message to the appropriate reactive system — to a transaction system for retransmissions and acknowledgments, and to a client system for responses. These reactive systems then update their states and timers, and if necessary take further actions. This

behavior is the same as that of the old threads, except that we also took the opportunity of this rewrite to update the clients' SIP logic from the old RFC 2543 [77] specification to the new RFC 3261 [1].

Once the transaction has completed, the transaction system invokes destructors for the client systems, notifying itself. Once all the client systems have been shut down, the transaction system can delete itself. The transaction system's `destruct` method frees the transaction data.

7.5 Performance Analysis

In order to evaluate the benefits of the new approach, we used the SIPstone test suite [78] to test the performance of both the new and old versions of SIPD. SIPstone tests the number of transactions per second a SIP proxy server can support in various configurations. In order to compare our server against another implementation, we also ran the same tests on IPTel.org's freely available SIP Express Router (SER).

Because the implementation of the SIPstone test code is not complete, we did not perform the full SIPstone-A test. Instead, we performed two representative tests: the **Registration** and the **Proxy 200** tests, over UDP. (These two tests are the highest-weighted components of the weighted average comprising the total SIPstone-A score.) In each case, the server was run on a dedicated Sun Netra X1, a 500 MHz UltraSPARC IIe with 128 MB of memory, running Solaris 2.9. The standard Solaris pthreads library was used, which supports "M:N" threading mapping user-level threads to a pool of kernel light-weight processes. (Each user-level thread must still have its own run-time stack.) The SIP servers referenced a MySQL database, running version 3.23.52 of the MySQL server on the same Netra. Communication was over a single 100base-T Ethernet connection. (This connection was not, as required by the SIPstone test, a dedicated network; it was instead our internal production lab network. This network was lightly loaded, however, so the results should be comparable.) The servers were configured to use the database in read-only mode — user authentication information was read from the database, but the servers were configured not to write it back.

The SIPstone tests work by having a number of load generators (in our tests, four) send requests to the server at a fixed rate, and measuring the transaction failure probability — the

percentage of requests that do not complete successfully within a given time limit. The server is then restarted, and the number of requests is increased. This process is repeated until the transaction failure probability exceeds 5%.

Specifically, for the **Registration** test, the clients send the server an unauthenticated SIP REGISTER request, to which the server responds with a 401 Unauthorized response containing an authentication challenge. The client then sends an authenticated REGISTER request answering the challenge, to which the server sends a 200 OK. If any other behavior occurs, or if the final 200 OK response has not arrived within 500 milliseconds of the initial request, the test is considered a failure.

For the **Proxy 200** test, a number of call handlers (also four for our tests) are used as well as load generators. Initially, the call handlers register a large number of destination locations with the proxy server. The load generators then send INVITE requests at a fixed rate to the proxy server, randomly selecting from among the registered addresses. The proxy server forwards each request to the appropriate call handler, which immediately responds with 180 Ringing and 200 Ok messages. These are forwarded back to the load generator. Upon receiving the 200 Ok message, the load generator sends an ACK message for the initial transaction and a BYE request for a new transaction. The BYE is similarly forwarded to the call handler, which again responds with 200 Ok. A time limit of two seconds is imposed between the initial INVITE request and the receipt of the 200 Ok response to the INVITE; if it is not received within that time, or if any other behavior occurs, the test is considered a failure.

7.5.1 Performance Results

The results of the tests are summarized in Table 7.1. The versions of SIPD chosen were version 1.21 — the final version to use the old thread architecture — and 1.23, the latest version. (Version 1.22, the first version to use reactive systems, was found to be too unstable under load for the tests to complete.) For comparison purposes, we also used version 0.8.10 of IPTel.org's SIP Express Router, at the time the most recent stable release.

Between versions 1.21 and 1.23, CINEMA's performance increased substantially. **Registration** performance increased by 76%, and **Proxy 200** performance increased by more than four

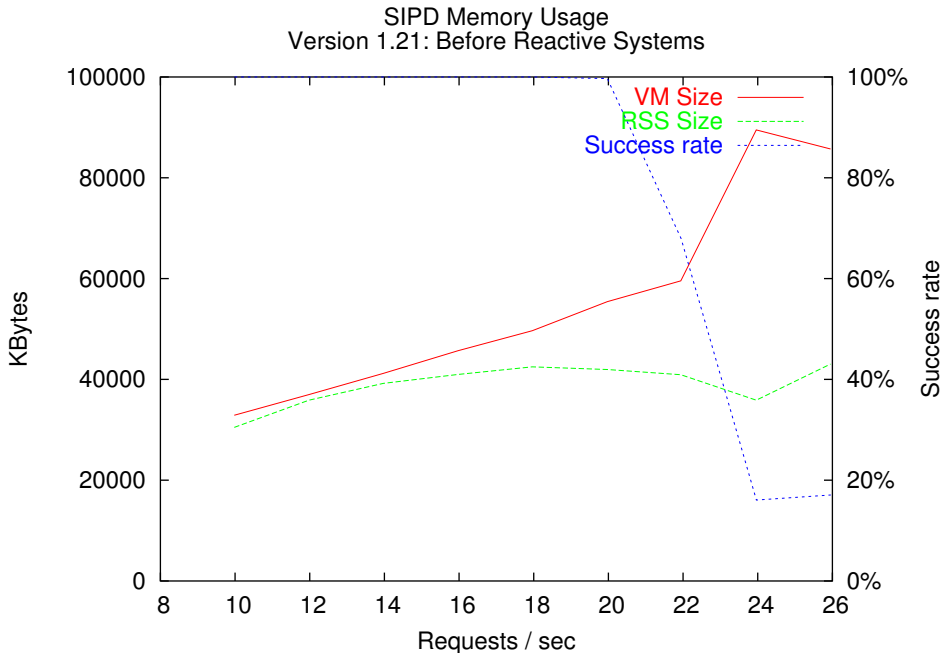

Figure 7.1: Memory Usage and Performance of CINEMA SIPD 1.21

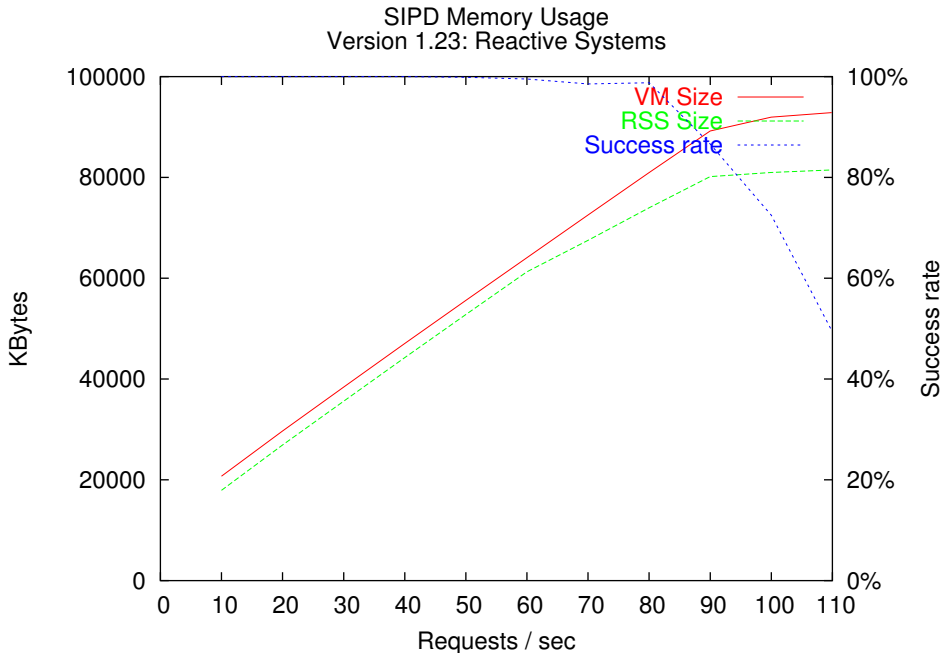

Figure 7.2: Memory Usage and Performance of CINEMA SIPD 1.23

Server	**Registration** (Req / sec)	**Proxy 200** (Req / sec)
CINEMA SIPD 1.21	168	21
CINEMA SIPD 1.23	296	86
SIP Express Router 0.8.10	400	539

Table 7.1: SIPstone Scores for Servers: Summary

times. Since the primary differences between version 1.21 and 1.23 lie in the change to reactive systems, we can conclude that they did cause a substantial performance improvement, especially for INVITE requests.

Performance analysis shows that the primary bottleneck is memory usage. Figures 7.1 and 7.2 show the memory usage of the CINEMA SIPD server under various loads of the **Proxy 200** test, and also the percentage of requests that complete successfully within two seconds. (A server's SIPStone score is the highest load at which 95% of requests complete successfully in this time.) As can be seen, the newer version of SIPD, which uses reactive systems, is both much more efficient in the amount of memory it uses per request, and also takes much better advantage of the memory it uses.

Table 7.1 shows, however, that the performance of the SIP Express Router is considerably better still: 35% better for **Registration**, and over seven times better for **Proxy 200**, than CINEMA 1.23. This indicates that there is still substantial room for performance improvements in CINEMA, especially in its handling of proxied requests.

7.6 Future CINEMA Improvements

Figure 7.2 shows that the primary limitation of CINEMA 1.23 is its memory usage. The SIPStone tests were run on a server with 128 MB of memory. The performance of the server drops off very abruptly when the resident set size of the server gets much larger than the system's available non-kernel memory. This observation was reinforced by observating that code changes to reduce processing did not affect the code's performace, whereas some initial changes to reduce memory usage (by, *e.g.*, removing unnecessary fields in data structures) did.

For historical reasons, CINEMA uses different data structures to represent messages it

receives from the network, and messages it sends to the network. (In its initial design, SIPD was only a redirect server and registrar; all received messages were requests, and all sent messages were responses.) As a result, proxying a request requires forwarded requests and responses to be copied from the received-message to the sent-message data structure. This effectively doubles the amount of data that must be stored per message, as well as requiring unnecessary processing to perform the copying.

Furthermore, the message passing architecture — now used to pass parsed messages between reactive systems, but originally used to pass them among threads — is fairly poorly designed in terms of memory ownership. Each message recipient assumes that it owns the parsed message it receives, and thus that it can modify it as needed, and that it is responsible for freeing it. Thus, each time a message is passed between reactive systems, it must be copied, again imposing a significant penalty both in memory use and processing. It would likely be significantly more efficient to allocate parsed headers on a per-transaction basis; however, this would require auditing the code to ensure that nothing assumes that it can safely modify the parsed headers.

CINEMA also uses separate threads to listen to UDP and TCP sockets; in particular, a separate thread listens to each open TCP connection. As with the old processing threads, this requires a full thread stack for each TCP socket, which spends most of its time waiting for TCP data. This could be replaced by a system which uses `select()` or `poll()`.

Finally, message parsing seems to be somewhat inefficient. This has not yet been thoroughly investigated.

7.7 Conclusion

A naïve approach to thread handling can cause network servers to be inefficient. We explained how the the CINEMA SIP proxy server was converted from a thread-per-transaction model to a model based on *reactive systems*, in which transactions are assigned to threads on an as-needed basis. This increased the performance of the server substantially. However, comparison with another similar server shows that there is still substantial room for further performance improvement.

Chapter 8

A Protocol for Reliable Decentralized Conferencing

Many approaches and topologies — including multicast and media mixing — have been proposed for distributed Internet conferencing. While existing solutions can work well for large or pre-arranged conferences, they can be less appropriate for smaller, impromptu ones. This chapter presents an alternative, *full mesh conferencing*, which allows any number of parties to communicate in a conference without a central point of control. The protocol allows parties to join and leave the conference at any time, and ensures that all members of the conference are always informed of new members. The chapter gives an overview of the protocol, analyzes it, describes a simulation environment for it, and discusses its applicability to the Session Initiation Protocol (SIP) and to other forms of decentralized communication.

8.1 Introduction

The Session Initiation Protocol [1], SIP, is described in Chapter 2. SIP is the Internet Engineering Task Force's standard for setting up multimedia sessions; it provides a means by which users can establish, maintain, and terminate calls between them. To aid this, it provides sophisticated user location and media description facilities. It provides facilities to set up diverse types of media, including instant messaging, distributed notification, and presence, as well as traditional audio

and video.

The basic SIP protocol is only engineered for point-to-point communications, and does not, inherently, provide any support for communications among more than two parties, other than loosely-controlled multicast conferences in which the users' media is sent to a multicast group. More tightly-controlled conferencing is useful and necessary in a number of circumstances — from simple three-way calling, in which two people on an ordinary call decide to add a third party, to large-scale conference calls.

There are a number of ways to provide conferencing with existing SIP mechanisms. However, all these mechanisms have some shortcomings, as described in Section 8.3. They are heavy-weight or architecturally inappropriate for certain types of conferences. This chapter proposes a new approach, describing a fully-distributed, decentralized protocol for conferencing which establishes a fully-connected mesh of signalling and media connections between the conference participants. We call this approach *full-mesh conferencing*.

This approach is not intended to replace the other solutions, but rather to complement them. The existing solutions are designed for certain problem domains, and are useful in those domains; however, they are over-engineered or architecturally inappropriate in some common scenarios. The new proposal addresses these scenarios.

This conferencing approach is also applicable to additional environments. Numerous scenarios require multiple networked devices to be able to communicate with each other without a single point of failure, and the topology of a full mesh is often very useful for robustness. Such topologies often need to be dynamically assembled, with end systems entering or leaving the group. Thus, the mechanism described in this chapter is also useful for such environments as group text messaging, highly-reliable alerting or event systems, establishing router peering relationships, distributed simulation, distributed databases, or clusters of network servers which need to share state information.

8.2 Related Work

The 'Sticky' Conference Control Protocol [79] is an early example of decentralized conferencing. It establishes an arbitrary topology, so that not all users can necessarily hear all the others. The

Mesh-enhanced Service Location Protocol [80] offers another example of a service in which fully connected meshes of devices need to be maintained as systems arrive and leave. This work establishes a protocol which lets Service Location Protocol Directory Agents exchange service registration information, so they can maintain consistent data for shared scopes. Unlike the work presented in this chapter, however, this protocol has no real notion of peer discovery or invitation, except via the Service Location Protocol's normal multicast advertisement. It deals only with state synchronization.

Explicit Multicast, or Xcast [81] offers a networking technology that can be complementary to full mesh conferencing, in networks which support it. With this technique, an IP device can explicitly specify a list of destinations in the IP header for a single packet; replication then occurs in the network. In a fully meshed conference, therefore, this could allow a conference member to save significantly on its bandwidth use. The full mesh protocol could complement this technique by providing a mechanism for conference members to know the addresses of the other participants in the conference.

The full mesh conferencing model has been proposed before in the evolution of the SIP protocol [82]. The work at the time foundered on the difficulty of ensuring that all users maintained full knowledge of the other members of the conference in complex scenarios. This chapter revives and completes this work.

8.3 Existing Conferencing Models

There are several ways to support multi-party conferencing in basic SIP. Rosenberg and Schulzrinne discuss this in an Internet-Draft [83]. To simplify somewhat, there are two primary ways to support conferencing with basic SIP: *multicast* and *mixing*.

8.3.1 Existing Conferencing Architecture: Multicast

Large-scale multicast conferences were the original motivation for the development of SIP. In a large-scale multicast conference, one or more multicast addresses are allocated to the conference. Each participant joins the multicast groups, and sends their media to the groups. Signalling is not

sent to the multicast groups. The sole purpose of the signalling messages is to inform participants of which multicast groups to join.

Multicast conferences can work reasonably well in networks that support them. They have the advantage that they do not require tight coordination between end systems; conference members can join and leave the conference independently, and conferences can survive network trouble and reconnect themselves seamlessly. The primary disadvantage of multicast conferences, however, is that multicast can be burdensome for networks and routers. Multicast (PIM-DM, PIM-SM) requires that each multicast router at least stores the group identity. In some cases, state is actually (S, G), i.e., you need to store sender state as well. With lots of very small groups where everyone sends, i.e., the typical 3-party phone calls, routers effectively store session state. Source-specific multicast (SSM) [84] makes the load on routers lighter, but does not eliminate this problem, and separate SSM group is required for each sender. Also, since subscription to multicast groups is usually not authenticated (since routers would need to keep the keys for users), anybody can subscribe to any group, thus directing traffic to random destinations. A single misconfigured or compromised system could fairly easily subscribe to all IPv4 dynamically-allocated multicast addresses and thus flood the network. As a result, very few Internet backbones support multicast. While multicast conferences can be useful in LANs, enterprise environments, or Internet 2, in the current commercial Internet they are largely impractical.

Multicast conferences are inherently loosely-coupled, and so they are not a good choice when tighter control of conference membership is desired. Communication of conference membership is carried out only using RTCP, so speakers may be unaware of who is currently able to hear them. They have no restriction, other than encryption, on users joining a conference, and key distribution and management can be cumbersome. Additionally, transition from a two-party to a multiparty session is awkward. Thus, while multicast can be useful for "webcasts," in networks which support it, it tends to be architecturally less applicable to the "conference call" model of group communications.

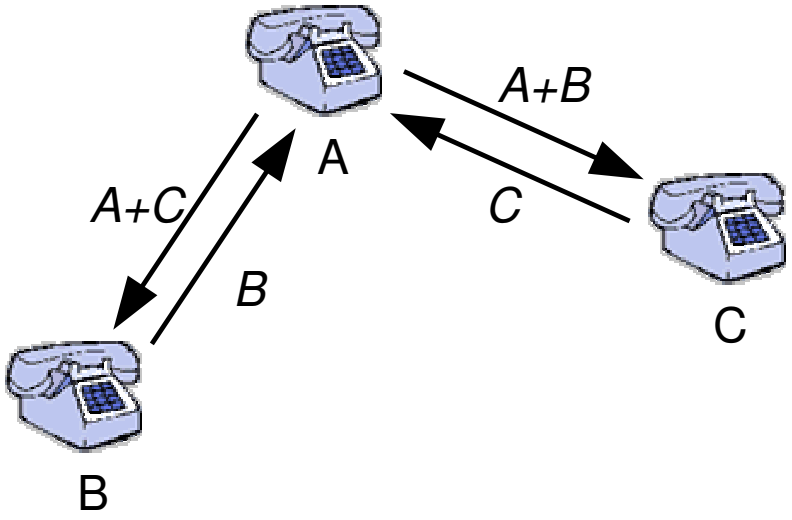

Figure 8.1: Conferencing: End System Mixing

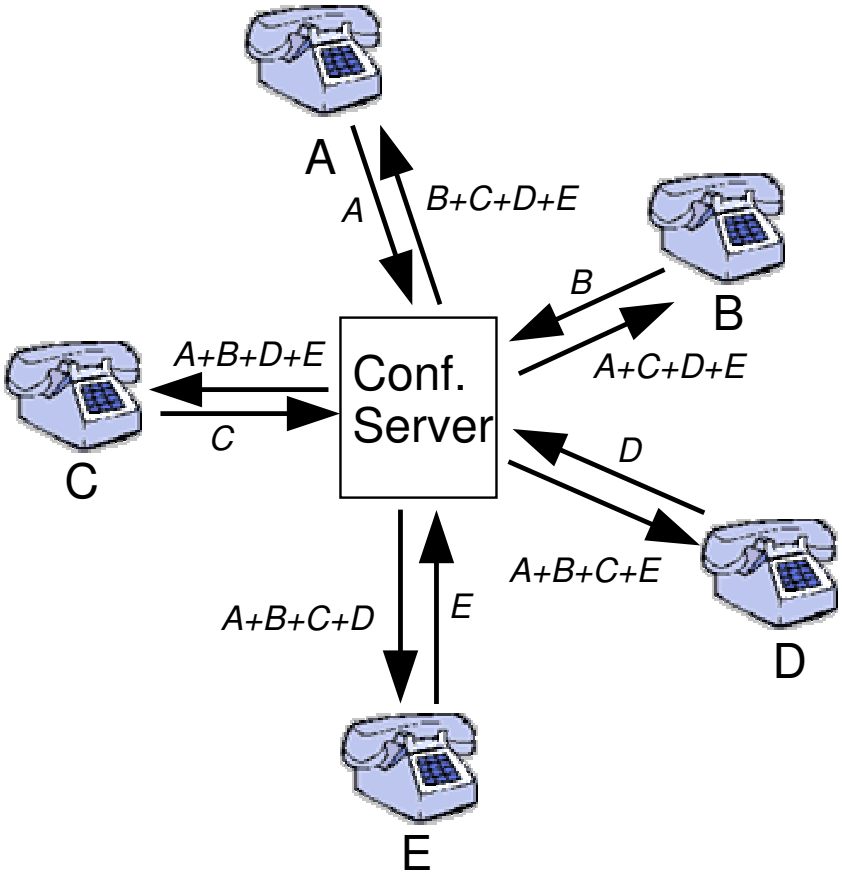

Figure 8.2: Conferencing: Conference Server Mixing

8.3.2 Existing Conferencing Architecture: Mixing

The other existing approach to conferencing is to have a SIP endpoint which connects the members of a conference, which mixes and forwards their media streams. There are two possible variants on this model: in *end system mixing*, shown in Figure 8.1, one member of the conference takes responsibility for mixing audio traffic; in *server-based mixing*, shown in Figure 8.2, an independent network entity performs it.

This model is probably the most common way of doing SIP conferencing. From the point of view of those end systems which are not performing mixing functions, the call can be treated as a standard SIP call. However, the model has several disadvantages. First of all, the existence of the conference is dependent on the mixer; if the mixer goes away, the call immediately ends. (This is more of a concern for end system mixing than for server-based mixing.) Secondly, the computational load on the mixer can be high; it may need to encode up to $N-1$ audio streams for an N-party conference. (Hierarchical mixing can lessen this computational load, while making mixer setup correspondingly more complex.) Finally, transitioning from a simple two-party call to a conference can be complex, particularly in the server-based mixing case, as the parties must locate a server, and then transition the existing call to the control of the server. Overall, of the two, server-based mixing is more reliable, and it works well for moderately large or pre-arranged conferences. However, it can be unwieldy for smaller conferences.

8.4 Full Mesh Conferencing

This chapter presents a new approach to conferencing, *full mesh conferencing*. It is intended for tightly-coupled, impromptu, small-to-medium size conferences (with up to, perhaps, 10 members) — that is to say, "conference calls," not the larger "presentation" sessions for which the dedicated resources of a conference server or the loose coupling of a multicast conference are likely more appropriate.

Figure 8.3 illustrates this model. In the full mesh model, every endpoint directly communicates with every other one. All the parties in the conference are "equal" — no user is topologically special, or has any additional rights or abilities beyond those of the others. Any

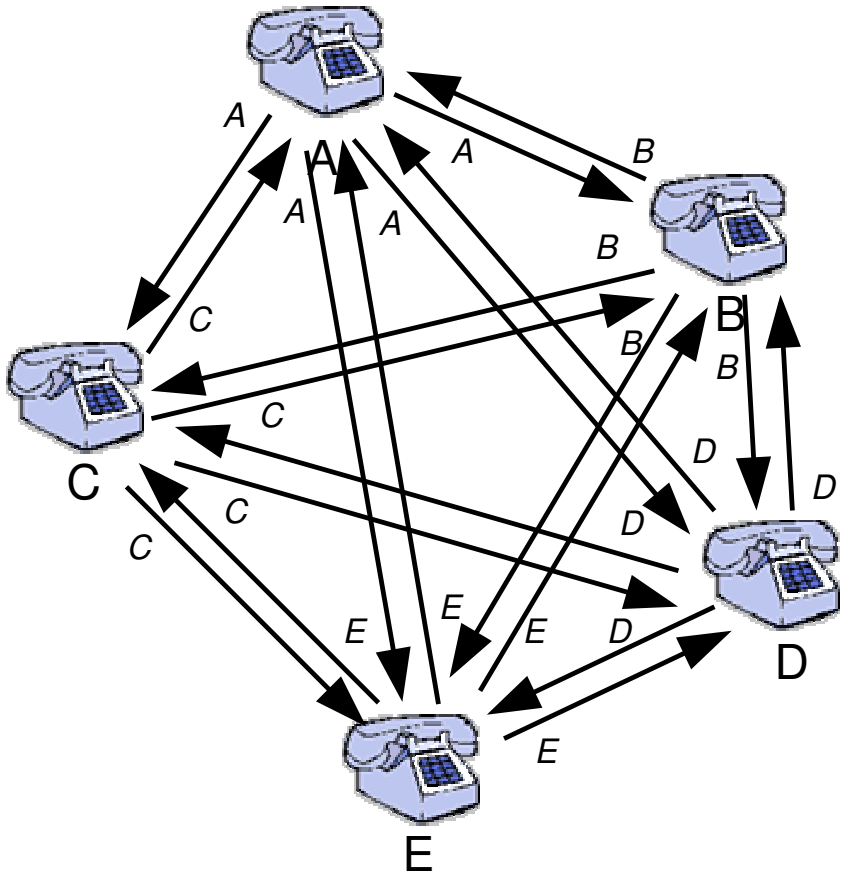

Figure 8.3: Conferencing: Full Mesh

member of the conference can at any time invite a new user to the conference. If the new member accepts, it establishes connections to the other parties in the conference. Similarly, any member of the conference can drop out of it at any time, without affecting the remaining conference participants.

Audio is mixed only for playout at end points; mixed audio is never sent over the network. This has advantages and disadvantages. The primary advantage is that no end system needs to encode more than one media stream, per outgoing codec. For most voice codecs, encoding tends to be much more computationally complex than decoding. Each user will be decoding up to $N - 1$ media streams in an N-member conference, but needs to encode only one. However, in an N-member conference, each user must have the bandwidth available to be able to send $N - 1$ simultaneous streams. (For audio conferences, users will normally only need to be able

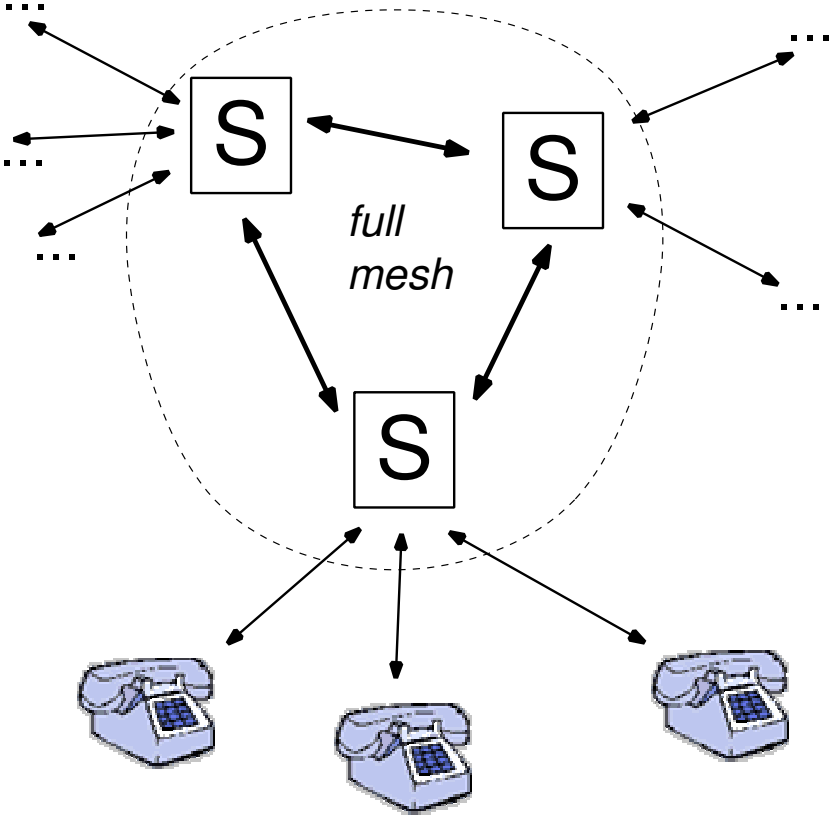

Figure 8.4: Conferencing: Combination of Conference Servers (S) and Full Mesh

to receive and decode one or two simultaneous streams. However, for video conferences, in most circumstances all the conference members will be sending at all times; while the user agent may, for instance, only choose to decode and show the video from active speakers, other video streams will still use up bandwidth.) Thus, this mechanism is less practical for bandwidth-limited end systems such as wireless devices, users with 56 kb/s modems, or users with asymmetric DSL connections with low upstream bandwidth, and it does not scale well to large conferences. A hybrid model, illustrated in Figure 8.4, can ameliorate this issue; this is a matter for future research.

8.4.1 Example

The presentation of the protocol will begin with some examples. This presentation of the protocol uses an abstract representation of point-to-point communication between peers. These messages

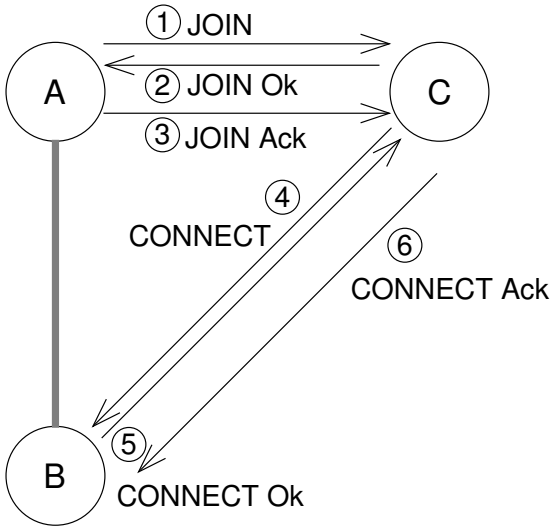

Figure 8.5: Example Full Mesh Message Flow: A New Member Is Invited

are inspired by SIP, but should not be interpreted as being actual SIP messages. A proposed mapping of these messages to SIP is presented later in Section 8.8. For the purposes of the examples, it is sufficient to understand that the abstract message **JOIN** invites a new member to join a conference; the message **CONNECT** establishes communication between two endpoints which are already members of the conference. Each of these messages can be answered with an **Ok** response, indicating that the request was accepted, or a **Reject** response, indicating that it was refused; the **Ok** responses are in turn acknowledged with **Ack** messages. This three-phase call setup procedure is needed to ensure conference security, for reasons explained in Section 8.7.2. These abstract messages are described in more detail in Section 8.4.2 below. Every end system maintains a list of the other end systems in the conference. Whenever a new member is invited, the inviter passes it a list of all the other members of the group. The new member then establishes communication with all the listed members.

Figure 8.5 shows the simplest case: a third endpoint being invited to a join two-party call. Initially, A and B are in a call. A then decides to ask C to join the call. To do this, A sends a **JOIN** message to C. C responds with a **JOIN Ok** message, indicating that it wishes to join the group. A then sends C a **JOIN Ack** message. This message lists A's view of the current membership of the group: A, B, and C. Upon receiving this message, C determines that it does

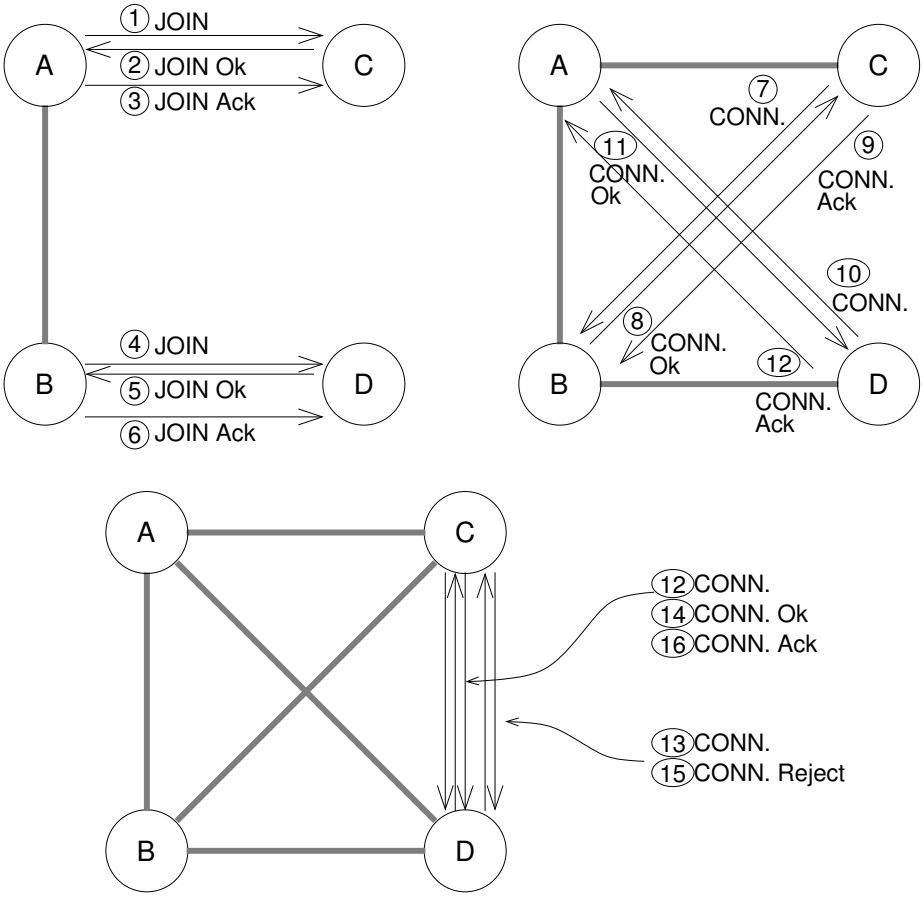

Figure 8.6: Example Full Mesh Message Flow: Two New Members Are Invited Simultaneously

not have a connection to B, and thus sends B a **CONNECT** message. When B receives the **CONNECT** message, it replies with **CONNECT Ok**; C responds to this with **CONNECT Ack**. At this point, every member has a communications channel established with every other.

Figure 8.6 illustrates what happens when both A and B invite new members, C and D, simultaneously. This illustrates that the protocol's message flow can quickly become quite complex. In the example, first A invites C, and B invites D, using the same procedure as for the simple message flow above. In the **CONNECT Ok** responses in messages 6 and 8, C and D are informed of each other. Since neither has yet established communications with the other, they both send each other **CONNECT** messages. In the specific instance illustrated in the figure, these **CONNECT** messages pass each other; at this point, one of the two must arbitrarily be chosen,

and the other rejected, so that the only one communications dialog is set up. The mechanism of this is explored in more depth in Section 8.4.4.

Note that the order of these messages is not fixed. The transaction between A and C (messages 1, 2, and 3) triggers the transaction between C and B (messages 7, 8, and 9); but these are entirely independent of the transaction between B and D (messages 4, 5, and 6). The resulting behavior of each of the end points, and which systems contact which other ones, depends on the exact order in which messages are sent and received. The full mesh protocol is designed to work correctly in all these cases, and the group will always converge to a proper full mesh.

8.4.2 Protocol Messages

The protocol uses ten abstract messages: four initial messages, **JOIN**, **CONNECT**, **LEAVE**, and **UPDATE**, and the responses **JOIN Ok**, **JOIN Ack**, **JOIN Reject**, **CONNECT Ok**, **CONNECT Ack**, and **CONNECT Reject**. Messages are sent within the context of, and control, *dialogs*. A dialog is a communications session between two end systems. It corresponds to the existence of bidirectional media exchange between the end systems. Every dialog is identified by a globally unique *dialog identifier*. Additionally, every conference has a globally unique *conference identifier*. A conference dialog falls within exactly one conference.

The **JOIN** and **CONNECT** messages are largely similar, as are their responses. Each message requests the initiation of a dialog between two end systems. The **Ok** responses accept the dialog initiation, whereas the **Reject** responses refuse it. The **Ack** messages confirm the receipt of the **Ok** messages; these are necessary for reasons explained in Section 8.7.2. The difference between **JOIN** and **CONNECT** is in their semantics: the **JOIN** message is sent to a user not in a group, to ask it to join the group; its handling typically requires a user decision, to accept or reject the request. The **CONNECT** message, by contrast, is sent from an end system that has joined the group to the other pre-existing group members, to establish point-to-point dialogs. In this case, the message (if validated) will not normally require human interaction. The distinction between **JOIN** and **CONNECT** is necessary; without it, it would not be possible to distinguish the cases of an end system being re-invited to a conference, after having left it, from that of a newly-arrived end system attempting to connect with a recently-departed one, not having

yet been informed of the latter system's departure.

The **LEAVE** message terminates a dialog, regardless of how the dialog was established. The **UPDATE** message does not affect the state of the dialog. It informs a party of new information about the conference membership list. This is discussed further in Section 8.4.3. All messages are assumed to be transmitted reliably. (The **Ack** messages are not for reliability, but rather carry the third message of the three-phase session establishment.)

From each end system's point of view, a dialog can be in two possible states: *pending* or *established*. For the party that initiates the dialog, the dialog is pending until it receives the **Ok** message; for the party that answers it, it is pending until it receives the **Ack** message.

8.4.3 Membership Maintenance and State Communication

Several messages of the full mesh protocol — **JOIN Ok**, **CONNECT Ok**, **JOIN Ack**, **CONNECT Ack**, and **UPDATE** — carry information about the sender's current view of the conference membership. In these messages, the sender lists all the conference members with which it has a dialog and whose conference tags it knows. (Conference tags are described in Section 8.4.5. A sender will always know the tags of members with which it has an established dialog, but may not yet know them for pending dialogs; pending dialogs for which tags are not known are not listed.) Each member in the list is marked with the state, pending or established, of the sender's dialog with that member. Additionally, the **JOIN** message may carry an advisory list of conference members, so the recipient knows who is in the conference, and can use this information to decide whether to join it. However, in this case the list does not result in any protocol actions.

When the recipient B receives a message (other than **JOIN**) carrying a membership list from a sender A, and chooses to act on it (i.e., it does not respond to it negatively), it does two things. First of all, B consults its own membership list, and checks to see if any of the members with established dialogs listed are members it was not previously aware of. If there are, it sends new **CONNECT** messages to all these members.

Secondly, B prepares its own list of members in response. If the A's message was a **JOIN Ok** or **CONNECT Ok**, B always includes the list of members in its **Ack** response. If, however, A's message was an **Ack** or an **UPDATE**, normally B would have no response to send.

However, if B has *established* connections to conference members which A did not know of (either as established or pending), B initiates a new **UPDATE** message within the same dialog, to inform A of all the members it knows of. Note that this response includes all established members A mentioned in its initial message, since B will now be setting up dialogs with all these members and will list them as pending. This ensures that this message will not itself trigger another **UPDATE** message unless A learns of further additional conference members.

The separation between established and pending members in the membership list ensures that every member's first introduction to a conference is the initial **JOIN** message it receives. If B were to send **CONNECT** messages to A's pending members, it is conceivable that the **CONNECT** message from B to A's pending member C could outrace a **JOIN** message from A to C, if, for example, A invited B and C simultaneously. This violates the definitions of **JOIN** and **CONNECT**.

8.4.4 The Double-Dialog Glare Problem

Because of the way the full mesh protocol floods membership information, it quite often happens that two end systems may attempt to establish dialogs with each other simultaneously for the same conference. In these cases, it is necessary to ensure that only one dialog is actually established, and the other is rejected. This is analogous to the problem of "glare" in the PSTN, in which two telephone switches simultaneously attempt to seize the same voice circuit. If both dialogs were to be set up, there would be two simultaneous connections between the end points; this is undesirable, as it is wasteful of bandwidth and causes unnecessary state complexity.

There are two possible scenarios for this. The simpler case is when a dialog establishment request (**JOIN** or **CONNECT**) arrives from A to B, when B already has an established dialog with A. In this case, B can simply always send A a **Reject** response to this request. The more complex case is when a dialog establishment request from A to B arrives when B has a *pending* dialog with A. The new dialog establishment request indicates that A also has a pending dialog with B; the situation is thus symmetric.

To solve this, a symmetry-breaking mechanism must be defined, so that both end systems can agree as to which dialog will be established, and which will be rejected. The simplest solution

is to establish some global ordering for end systems, such that the connection from the "earlier" system to the "later" one is chosen. The exact nature and mechanism for this ordering is arbitrary, as long as it is deterministic and universally agreed-to. (A lexicographic ordering of end systems' globally-unique identifiers is one possibility, so long as these identifiers are communicated in **JOIN Ok** and **CONNECT Ok** messages.) Which dialog "wins" in this situation has little import in practice, as the direction in which a dialog was established does not matter for future communications. (The direction may matter for minor low-level details of the communications protocol, but these do not affect session and media semantics.)

8.4.5 Immediate Departure and Reconnection

It is possible for a user to be re-invited to a conference while in the process of leaving it. For example, a user hangs up accidentally, and is immediately invited back by the other conference members. In this situation, the system's **CONNECT** message for its new dialog could out-race the **LEAVE** message terminating the old dialog. Absent any mechanism to prevent this, the destination system for these two messages would perceive the **CONNECT** as setting up a double dialog, as described in Section 8.4.4, and would reject it.

Therefore, the protocol introduces *conference tags*. Whenever an end system joins a conference, it generates a unique identifier which will serve to identify this "instance" of its conference membership. It communicates this identifier in every message it sends. Additionally, end systems include conference tags for each member in the membership lists they send in **Ok** and **Ack** messages. Finally, whenever an end system knows the conference tag of the party to which it is sending a message, it includes the remote party's tag in the message. The only messages for which end systems do not know their counterparties' tags are the initial **JOIN** message, and a **LEAVE** message which was sent immediately following a **JOIN**, before **JOIN Ok** has arrived. (This can occur if an end system invites another party to the conference, and then immediately leaves.)

These conference tags are used in two ways to eliminate the problem of departure and reconnection. First of all, if an end system B receives a **CONNECT** message from another system A with which it already has a connection, but A's conference tag is different, it knows

that this is not a double dialog. For example, consider the new **CONNECT** message to be from $A2$, and the old one to be from $A1$. In this case, B establishes a connection with $A2$, and can conclude that a **LEAVE** message from $A1$ will be forthcoming shortly.

Additionally, if an end system receives a message addressed to it with an unknown or outdated conference tag, it rejects the message, just as it would if it received such a message for an unknown conference. In the example, if A receives a **CONNECT** message from C addressed to $A1$, it knows that C has outdated information about A's state, and rejects it; it can conclude that once updated information has propagated to C, C will send a correct **CONNECT** message to $A2$.

8.5 Security and Authentication

Security is a significant consideration for full mesh conferences. In addition to all the security requirements of point-to-point calls — authentication of the identities of callers and called parties, privacy and authentication of media traffic, and privacy of callers' and called parties' identities from third parties, for example — conferences have the additional requirement that only end systems authorized by an existing conference member are allowed to join the conference.

Under the model described in Section 8.4, an end system which receives a **CONNECT** message for an existing conference will automatically establish a dialog with the sender of the message, and then send the sender a media stream containing all the media generated by that end system. Clearly, it is very important that the end system have some way to know that this **CONNECT** message is coming from a legitimate conference member, i.e., one who has been invited to the conference by an actual user. Otherwise, if an adversary were able to observe or guess conference IDs, he or she would be able to barge into a conference without the consent of its members.

To resolve this, there must be a way to verify that the **CONNECT** message was indeed triggered by a legitimate **JOIN** from a legitimate user. To accomplish this, the protocol uses a cryptographic public key solution. Whenever an end system joins a conference, it generates a purpose-built [85] public key which it will use for the duration of the conference.[1] All **JOIN**,

[1] In practice, as key generation is an expensive operation, end systems will probably use longer-lived public keys,

CONNECT, **JOIN Ok**, and **CONNECT Ok** messages communicate the sender's public key to the other members of the conference. Then, whenever an end system A sends a membership list (in **Ok** or **Ack** messages) to another end system C, it includes in this invitation a "letter of invitation", signed with its private key, indicating that A has invited C to be a member of the conference. If this message's membership list informed C of the existence of B, C, as described above, sends a **CONNECT** message to B. In this message, it includes a copy of the letter of introduction, signed by A. B has already received a copy of A's public key, so it can verify the signature on the introduction, and so know that C is legitimately allowed to join the conference.

Because of race conditions between a session member sending a **JOIN** message, and its own departure from the conference — if, for instance, A invites C to a conference, and then immediately leaves, before C has contacted B — it is necessary for end systems to remember the public keys of members who have departed the conference. The length of time for which keys need to be remembered depends on the maximum length of time that letters of invitation could persist in ongoing transactions between conference members; roughly speaking, this will be twice the maximum duration of a transaction.

This security mechanism can be attacked by an attacker who can intercept and modify messages, by altering the public keys that members advertise to each other. Such an attacker could create its own letters of introduction at will. However, this same vulnerability exists for point-to-point communications. Communications systems need to be able to secure their point-to-point communications in order to provide user security. SIP, for example, uses mechanisms such as TLS and S/MIME for this purpose. These security mechanisms, combined with the approach described above, should be sufficient to protect against conference barge-in. This solution also does not prevent a member who has left the group from re-entering it, by replaying an existing certificate. It is possible that certificates could be set up to have limited lifetimes. This works if clocks are synchronized, but will require further investigation if there is no globally synchronized clock shared among all the conference participants.

---

and the signing mechanism will include a means to bind signed messages to conference IDs and tags.

8.6 Verification of Full Mesh Protocol

As mentioned earlier, there have been previous attempts [82] to describe full mesh conferencing for SIP. (The author was a participant in these previous attempts.) These attempts established the basic concept of the full mesh conference, but foundered on the difficulty of verifying manually that the protocol always converged.

The primary difficulty in verifying of the full mesh protocol is that its behavior depends strongly on the order in which events occur. For example, consider the example from Figure 8.6 in Section 8.4.1. In the example, messages 2 and 5 — the "horizontal" **JOIN Ok** messages from C to A and from D to B — are received before the "diagonal" **CONNECT** messages 7 and 10 from C to B and from D to A. Thus, at the time of the processing of the **JOIN Ok** messages, A and B are unaware of the existence of D and C respectively.

If, instead, for example, the **CONNECT** messages were to outrace the **JOIN Ok** messages, A and B would know already know about a new fourth member of the group when they received the **JOIN Ok** message. Thus, by the procedure of Section 8.4.3, A and B would include D and C in their **JOIN Ack** messages to C and D, respectively, informing them of the new member.

The number of possible orders in which events can occur is, in fact, exponential in the number of members in the group and the number of actions (**JOIN** messages and group departures) which will occur. Thus, two facts quickly became clear: protocol verification would need to be automated, and this automation would have to be heavily optimized in order to keep verification tractable on standard hardware.

8.6.1 The Verification Framework

A custom program was written in C++ to verify the protocol. The simulator maintains two items: firstly, the *state* of the system, describing which end points are in the system and each end point's knowledge of its dialogs; and secondly, a list of pending *events* which are to be executed, consisting of event actions (members inviting other end systems to the group, and members leaving the group) and sent messages. To simulate a particular scenario, the state is set up with some number of users in a fully-connected conference, and the event list contains the actions to be taken.

Recursively, the verifier picks an events from the pending event list, and applies the actions it specifies to the state. These actions may involve the addition of additional events to the event list, as when an event causes messages to be sent. The verifier is then executed on the new state and event list. Once the sub-list has completed, the verifier chooses the next event from the list, and continues until all events have been exhausted. In this way, the verifier exhaustively searches every possible event ordering.

When the verifier is executed with no pending events, it instead *validates* the resulting final state. A final state is valid if every user which believes itself to be a member of the group has exactly one, alive, dialog with every other such user. If the group is disconnected, not fully linked, or any dialogs are in the wrong state or doubled, the verifier prints an error message and the exact sequence of events and states that led to this outcome.

Because many events execute independently of one another, this simulation environment can end up repeating many scenarios. For example, in the message flow of Figure 8.6, if the first two events executed are the transmission of message 1 and the transmission of message 4, it is irrelevant which of these two occurs first; the state, and pending events, afterwards will be the same. Thus, the verifier maintains a *state cache*. After state has finished being executed, it is recorded in the state cache. If a future execution of the verifier for this verification run results in the same state and list events being visited, the execution is pruned as redundant. This greatly reduces the number of test cases explored by the validator, but it means that simulations can fail if the cache fills all available memory.

8.6.2 Test Runs Performed

Table 8.1 lists all the simulated mesh actions executed by the verifier. Conference members are named A, B, C, . . ., in order.

In almost all cases, the verifier confirmed that every possible ordering of the events and messages of the full mesh protocol resulted in a fully-connected and self-consistent conference. There are, however, two exceptions to this. First of all, in two cases (marked with a †) the state cache grew so large as to exhaust all RAM and swap on the computer on which the simulation was executing. This exhaustion occurred after several tens of millions of orderings had been

Run	Initial	Actions	Run	Initial	Actions	Run	Initial	Actions
1	A	$-A$	20	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $A{\rightarrow}C$, $-A$	39	A, B	$A{\rightarrow}C$, $B{\rightarrow}D$, $-C$
2	A, B	$-B$	21	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $A{\rightarrow}C$, $-B$	$40*\dagger$	A, B	$A{\rightarrow}C$, $B{\rightarrow}D$, $-A$, $-B$
3	A, B, C	$-C$	22	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $A{\rightarrow}C$, $-C$	41	A, B	$A{\rightarrow}C$, $B{\rightarrow}D$, $-A$, $-C$
4	A	$A{\rightarrow}B$	23	A	$A{\rightarrow}B$, $-B$, $A{\rightarrow}B$	42	A, B	$A{\rightarrow}C$, $C{\rightarrow}D$
5	A	$A{\rightarrow}B$, $A{\rightarrow}C$	24	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $-B$, $A{\rightarrow}B$	43	A, B	$A{\rightarrow}C$, $C{\rightarrow}D$, $-A$
6	A	$A{\rightarrow}B$, $-B$	25	A	$A{\rightarrow}B$, $-A$, $B{\rightarrow}A$	44	A, B	$A{\rightarrow}C$, $C{\rightarrow}D$, $-B$
7	A	$A{\rightarrow}B$, $-A$	26	A, B	$A{\rightarrow}C$	45	A, B	$A{\rightarrow}C$, $C{\rightarrow}D$, $-C$
8	A	$A{\rightarrow}B$, $-B$, $A{\rightarrow}B$	27	A, B	$A{\rightarrow}C$, $A{\rightarrow}D$	46	A, B	$A{\rightarrow}C$, $C{\rightarrow}D$, $-D$
9	A	$A{\rightarrow}B$, $-A$, $B{\rightarrow}A$	28	A, B	$A{\rightarrow}C$, $B{\rightarrow}C$	47	A, B	$A{\rightarrow}C$, $B{\rightarrow}C$
10	A	$A{\rightarrow}B$, $A{\rightarrow}C$, $-A$	29	A, B	$A{\rightarrow}C$, $-A$	48	A, B	$A{\rightarrow}C$, $B{\rightarrow}C$, $-A$
11	A	$A{\rightarrow}B$, $A{\rightarrow}C$, $-B$	30	A, B	$A{\rightarrow}C$, $-B$	49	A, B	$A{\rightarrow}C$, $B{\rightarrow}C$, $-C$
12	A	$A{\rightarrow}B$, $A{\rightarrow}C$, $B{\rightarrow}C$	31	A, B	$A{\rightarrow}C$, $-C$	$50\dagger$	A, B	$A{\rightarrow}C$, $B{\rightarrow}C$, $C{\rightarrow}D$, $-C$
13	A	$A{\rightarrow}B$, $A{\rightarrow}C$, $B{\rightarrow}C$, $C{\rightarrow}B$	32	A, B	$A{\rightarrow}C$, $A{\rightarrow}D$, $-A$	51	A, B	$A{\rightarrow}C$, $-B$, $A{\rightarrow}B$
14	A	$A{\rightarrow}B$, $A{\rightarrow}C$, $-A$, $B{\rightarrow}C$	33	A, B	$A{\rightarrow}C$, $A{\rightarrow}D$, $-B$	52	A, B	$A{\rightarrow}C$, $-B$, $C{\rightarrow}B$
15	A	$A{\rightarrow}B$, $A{\rightarrow}C$, $-A$, $B{\rightarrow}C$, $C{\rightarrow}B$	34	A, B	$A{\rightarrow}C$, $A{\rightarrow}D$, $-C$	53	A, B	$B{\rightarrow}C$, $-B$, $A{\rightarrow}B$
16	A	$A{\rightarrow}B$, $B{\rightarrow}C$	35	A, B	$A{\rightarrow}C$, $-C$, $A{\rightarrow}C$	54	A, B	$B{\rightarrow}C$, $-B$, $C{\rightarrow}B$
17	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $-A$	36	A, B	$A{\rightarrow}C$, $-C$, $C{\rightarrow}A$	55	A, B	$-A$, $-B$
18	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $-B$	37	A, B	$A{\rightarrow}C$, $B{\rightarrow}D$	56	A, B, C	$-B$, $-C$
19	A	$A{\rightarrow}B$, $B{\rightarrow}C$, $-C$	38	A, B	$A{\rightarrow}C$, $B{\rightarrow}D$, $-A$	57	A, B, C	$-A$, $-B$, $-C$

Key:

Initial The initial conference members, before any actions are executed.

Actions The actions to be executed, in some arbitrary order, during the scenario.

$X{\rightarrow}Y$ X will attempt to invite Y to the conference, if X is currently a member and does not know about Y.

$-X$ X will leave the conference, if it is currently a member.

$*$ Some event orderings can lead to disconnected conferences. See the text for an explanation.

† The simulation's state cache exhausted all available memory before completing, on a Sun Fire 280R with 2.0 GB of RAM and 5.0 GB of swap, running Solaris 8.

Table 8.1: Full Mesh Conference Scenarios Explored with Verifier

considered and pruned. As mentioned above, the number of event orderings is exponential in the size of the groups and the number of initial actions. This is why none of the simulated groups involve more than four members.

In one case (marked with a $*$ in the table), the simulation did not result in a single fully-connected conference, but instead resulted in several smaller disconnected ones, as the "bridging" members of a conference left the conference before the new members found out about each other. Specifically, in simulation 40, B and C both join what they view as three-party conferences, and then have both their peers leave. Because A and B are gone, C and D never discover each other. This is not a 'bug' in the protocol; in the absence of a central repository of information about conferences, there is no way in this scenario that information about C could reach D, or vice-versa.

8.7 Analysis and Rationale

The examples of Table 8.1 cover a large number of the possible scenarios of full mesh operations, and each one exhaustively searches the possibilities for a particular set of operations. These simulations do not, however, fully explore the possibilities of the full mesh signalling, as the potential size of conferences is, of course, unlimited. This section will attempt to justify the belief that the cases considered adequately cover all the possible ways that a conference membership changes can interact. We will also give rationales for some of the more unusual features of the protocol, to illustrate how a more naïve protocol can fail.

8.7.1 Protocol Correctness

The first point to consider is to ensure that knowledge of a member joining the conference is always flooded to all other members of the conference. In the absence of simultaneous conference departures, this is clear. Once A invites B to the conference, B will send **CONNECT** messages to every member C, D, etc., that A knows about. In the responses to these **CONNECT** messages, B will transitively be informed of every member these members know about, and so, recursively, will eventually learn of, and be connected to, every member of the conference.

Similarly, departure from the conference is straightforward. **LEAVE** messages are sent to every member of the conference; even if other members of the conference subsequently attempt to connect to the new member, due to out-of-date lists of conference membership, the departing member will reject these connection attempts. Because conference tags distinguish instances of an end system's conference memberships, there is no ambiguity between near-simultaneous departures and re-connections, and so the analyses of these two scenarios can be considered independently.

The remaining case, therefore, is to consider simultaneous connections to, and departures from, the conference. It is possible in this instance for the conference to degenerate into several sub-conferences. This can happen if a "bridging" member of the conference — a conference member which alone knows about both portions of the conference — departs from the conference before propagating information between the two sides. In this case, however, both sides will still become fully-connected within themselves, by the argument above.

8.7.2 Rationale for Three-Phase Session Establishment

The most unusual feature of the protocol as it is described in Section 8.4 is likely the three-phase nature of the **JOIN** and **CONNECT** messages. This feature is necessary in order to ensure the correct behavior of the security mechanism described in Section 8.5. As described in that section, if B sends a membership list to C, it includes a signed "letter of introduction" certifying that C is allowed to connect to other members of the conference, say A. C can then include this in a **CONNECT** message to A, so A knows that C has been authorized. In order for this to work, however, A must already have received B's public key, so it can validate the signature on the letter of introduction.

In a two-phase connection model, there are some scenarios in which this would not be true. Consider, for example, a two-phase connection model in which A invites B to the conference, and then B immediately invites C. In a two-phase model, B could consider itself fully connected to A once it had sent a **JOIN Ok** response to a **JOIN** message from A. It could therefore immediately send a **JOIN** message to C, advertising A in its membership list, which could trigger a **CONNECT** message to A. However, in this scenario, the **CONNECT** message from C to A, with a letter of introduction signed by B, could out-race the **JOIN Ok** message from B to A, including B's public key. A would therefore not be able to verify the validity of the letter of introduction, and would reject the **CONNECT** message; a full mesh would therefore not be established.

In the actual three-phase model used by the protocol, however, B may not advertise A until B has an established connection to A. As described in Section 8.4.2, B does not have an established connection to A until it has received a **JOIN Ack** message from A. At this point, B knows that A has received a copy of its public key, so it can safely advertise A in future conference membership lists.

8.8 Realization of the Full Mesh Protocol in SIP

The abstract protocol described in Section 8.4 was designed to be a simplified representation of SIP messaging. It was designed to be expressive enough to capture all the behavior

necessary to represent point-to-point communications, yet simple enough to make the implementation and complexity of the automatic verifier, described in Section 8.6, tractable. This section describes how this abstract protocol can be expressed in actual SIP messages.

SIP communication sessions are organized into dialogs. When SIP is being used to control multimedia communications, dialogs are initiated with the INVITE method. If the other side agrees to initiate the dialog, it responds with a 200 OK response, which is acknowledged with an ACK request; otherwise, it sends one of a large number of potential failure responses. Once established, dialogs, and their associated multimedia communication, continue until they are terminated by either party, using the BYE method and its 200 OK response. If the session initiator wants to terminate the dialog before it has received a final response to its initial INVITE, it can do so by sending the CANCEL request.

To implement the full mesh protocol in SIP, this section provides a possible mapping of the full mesh protocol's abstract methods to concrete SIP methods. As both **JOIN** and **CONNECT** establish dialogs in the abstract protocol, they are both mapped to the SIP INVITE method. For similar reasons, **LEAVE** is mapped to either BYE or CANCEL, depending on the state of the dialog when it is invoked, and the **UPDATE** method can be mapped either to a re-INVITE or to a newly-defined SIP method (potentially, indeed, UPDATE [86]). The two subsequent phases of the connection process maps naturally: **Ok** becomes a 200-class success response, **Reject** becomes a 400-, 500-, or 600-class failure response, and **Ack** is ACK.

All of these requests must include several additional header fields, beyond those defined for a standard point-to-point call, in order to support full mesh conferences. First of all, to identify conferences, we define a header field Conference-ID, which uniquely identifies a full mesh conference. The value of this header field is established by the end system which initially creates the conference, and is globally unique. It is created using the same procedure as that used to create globally unique values for the existing required header field Call-ID. Secondly, to distinguish between **JOIN** and **CONNECT** messages, we define another new header field, Invited-By, which is included only for **CONNECT** messages. This header field carries the identity of the system which initially invited the sender of the message to the conference. This field can also have some cryptographic authentication; this was discussed further in Section 8.5. Finally, to provide the

list of conference members, another header field, Conference-Member, is provided for those messages which include the member list. This lists the Contact addresses of all the conference members the request's sender knows about. A header field parameter, status, indicates whether the members are established or pending; another one, tag, gives the member's conference tag. Cryptographic formats for the public keys and letters of introduction remain to be determined; work for this should be developed based on ongoing work with purpose built keys [85]. Since these keys can be several kilobytes, they will most likely be carried as multi-part session bodies.

8.9 Future Work

As designed, the full mesh conference protocol is perfectly decentralized — no member of a conference has special privileges. While this is the proper model for many conferences, it is not universally applicable. Development is ongoing [87] on how to offer sophisticated admission and floor control for conferences. This development generally assumes centralized conferencing models, usually those involving a conference server. It would be useful to be able to use these capabilities in the decentralized environment, but this will require significant further investigation to see how well the assumptions of the centralized model can carry over to the decentralized case. In particular, there would need to be some mechanism by which the members of the mesh conference can agree about who has control over the conference and is authorized to make these decisions.

As discussed in Section 8.4, there are also circumstances in which combination topologies, falling somewhere between the full mesh and the centralized server, are useful. This is difficult to arrange in a decentralized manner, without prior configuration, as these topologies do not have the inherent symmetries of a star or a mesh. However, there are such environments: for example, as illustrated in Figure 8.4, a set of conference servers could form a mesh topology among them, and then provide a star topology to clients.

8.10 Conclusion

After reviewing a number of solutions for Internet conferencing, it is clear that they all have some limitations. This chapter therefore presents an additional mechanism which complements

these approaches, allowing conferences to be established in a reliable, decentralized manner. This mechanism is also applicable to other environments which require the decentralized establishment of a full mesh communications topology. The protocol's correctness was verified for for a large number of scenarios; its correctness was analyzed, and a mapping to SIP was established. Some potential security issues with the protocol were discussed, and solutions for them were provided.

Chapter 9

Interworking Internet Telephony and Wireless Telecommunications Networks

Internet telephony and mobile telephony are both growing very rapidly. Directly interworking the two presents significant advantages over connecting them through an intermediate PSTN link. This chapter proposes three novel schemes for the most complex aspect of the interworking: call delivery from an Internet telephony (SIP) terminal to a mobile telephony (UMTS) terminal. It then evaluates the proposals both qualitatively and quantitatively. However, existing equipment may not support packet interfaces needed for such interworking. Therefore, the chapter also considers techniques for backward compatibility, and analyzes their performance as well.

9.1 Introduction

Two of the fastest growing areas of telecommunications are wireless mobile telephony and Internet telephony. Second and third-generation digital systems such as the Global System for Mobile communications (GSM) [88], the Universal Mobile Telecommunications System (UMTS) [89], and wideband CDMA [90] are bringing new levels of performance and capabilities to mobile communications. Meanwhile, both the Internet Engineering Task Force's Session Initiation Protocol (SIP) [1] and the International Telecommunications Union's H.323 [14] enable voice and multimedia telephone calls to be transported over an Internet Protocol (IP) network. Subscribers

to each of these networks need to be able to contact subscribers on the other. There is, therefore, a need to interconnect the two networks, allowing calls to be placed between them.

Some research has been performed investigating various aspects of interworking mobile communication systems with IP-based systems. The iGSM system [91] allows an H.323 terminal to appear to the GSM network as a standard GSM terminal, so that a GSM subscriber can have his or her calls temporarily delivered to an H.323 terminal rather than a mobile device. Several papers [92, 93, 94] describe a system for interworking GSM's in-call handover procedures with H.323. The Unified Mobility Manager (UMM), as described by Haase, Murakami, and La Porta [95], is an implementation of (among other things) a modified HLR, as described by Section 9.3.2 in this chapter. (The work described in this chapter was performed with this group at Lucent Technologies' Bell Laboratories, and originally described in a paper [7] co-authored with Kazutaka Murakami, Thomas F. La Porta, and Mehmet Karaul. This chapter is an adaptation of that paper.)

Other than the UMM, however, none of these approaches solves the general interworking question: what is the best way for calls to be delivered and routed between the two networks?

As both mobile and Internet telephony are already designed to interconnect with the Public Switched Telephone Network (PSTN), the easiest way to interconnect them would be simply to use the PSTN as an intermediate link. This is, however, inefficient and suboptimal, as compared to connecting the networks by interworking the protocols directly, for a number of reasons.

First of all, routing calls via the PSTN can result in inefficient establishment of voice circuits. This is a common problem in circuit-switched wireless systems called "triangular routing," as illustrated in Figure 9.1. Because a caller's local switch does not have sufficient information to determine a mobile's correct current location, the signalling must travel to an intermediate switch which can locate the subscriber correctly.[1] This intermediate switch can be far away from the

[1]There is an architectural difference here between the American mobile system based on ANSI 41 [96] and the European systems based on GSM/UMTS MAP. In the American system, calls are always routed through a home mobile switching center, which is in a fixed location for each subscriber, so the voice traffic for all of the subscriber's calls travels through that switch. By contrast, GSM improves on this routing by sending calls through a gateway mobile switching center, which can be located close to the originating caller. However, as discussed in [97], there are some cases, such as international calls, where an originating PSTN switch does not have enough information to conclude that a call is destined for the GSM/UMTS network, and thus routes it to the subscriber's home country. Because there is no way for circuit paths to be changed once they have been established, the call's voice traffic travels first to the user's home country and only then to his or her current location.

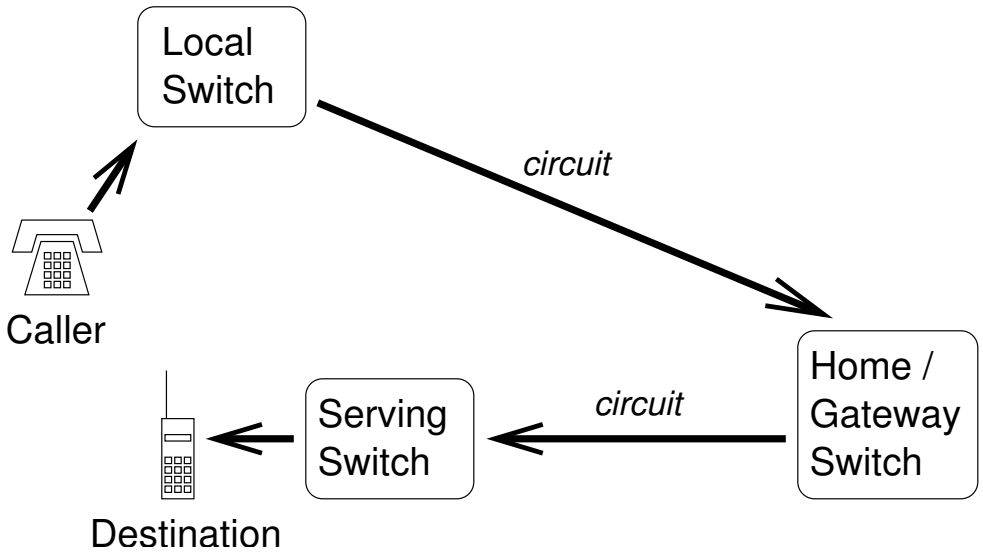

Figure 9.1: Illustration of Triangular Routing in Mobile Networks

caller and the destination even if these two endpoints of the call are located in a geographically close area. Since voice circuits are established at the same time as the call signalling message is routed, the voice traffic could be transported over a long, inefficient route.

In Internet telephony, by contrast, the path of a call's media (its voice traffic, or other multimedia formats) is independent of the signalling path (assuming Mobile IP is not being used, as described in Section 9.2.2.) Therefore, even if signalling takes a triangular route, the media travels directly between the devices which send and receive it. Since each device knows the other's Internet address, the packets making up this media stream are sent by the most efficient routes that the Internet routing protocols determine.

As we interwork Internet telephony with mobile telephony, we would like to maintain this advantage. We can accomplish this by supporting a direct IP connection between mobile base stations and IP terminals. With PSTN signalling, this is not possible, so IP telephony signalling must be used to establish this connection.

Another motivation for direct connection between mobile and Internet telephony is to eliminate unnecessary media transcoding. The Real-Time Transport Protocol (RTP) [98], the media transport protocol common to both H.323 and SIP, can transport almost any publicly-defined media encoding [99]. Most notably, the GSM 06.10 encoding [100] is implemented by many

clients. If a GSM mobile device talks to an RTP-capable Internet telephone with an intermediate PSTN leg, the media channel would have to be converted from GSM 06.10 over the air, to uncompressed (μ-law or A-law) audio over a PSTN trunk, and then again (likely) to some compressed format over the RTP media channel. The degradation of sound quality from multiple codecs in tandem is well known, and multiple conversions induce unnecessary computation. A direct media channel between a base station and an IP endpoint allows, by contrast, communication directly using the GSM 06.10 encoding without any intermediate transcodings.

Finally, on a broader scale, an integrated architecture supporting Internet and mobile telephony will evolve naturally with the expected telecommunications architectures of the future. Third-generation wireless protocols will support wireless Internet access from mobile devices. New architectures such as RIMA [101] for Mobile Switching Centers (MSCs) are using IP-based networks for communications between MSCs and base stations. In the fixed network, meanwhile, IP telephony is increasingly becoming the long-haul transport of choice even for calls that originate in the PSTN. The direct connection between Internet telephony and mobile networks takes advantage of all these changes in architecture and allows us to build on them for the future.

This chapter will consider the issue of how to interwork Internet telephony and mobile telecommunications, such that all the issues discussed above are resolved. For concreteness, the architecture will be illustrated using SIP [1] for Internet telephony and UMTS [89] Release 1999 for mobile telephony. UMTS Release 1999 is an evolution of the older GSM [88] system, and as such is the most recent version of this widely deployed infrastructure. Newer UMTS releases will be directly IP-based, but systems based on GSM will likely persist for some time.

The rest of the chapter is structured as follows. Section 9.2 gives an architectural background on the mobility and call delivery mechanisms of UMTS and SIP, to provide a basis for the following discussions. Section 9.3 proposes three different approaches to interworking UMTS and SIP, under the assumption that UMTS visited networks are IP-enabled. Section 9.4 provides mathematical and numerical analyses of the three proposals. Section 9.5, describes and analyzes how efficiently the three proposals can interwork with existing non-IP-enabled infrastructure. A higher-level discussion of the proposals' relative merits is offered in Section 9.6, and the chapter finishes with some conclusions in Section 9.7.

9.2 Background

This section reviews the mobility and call delivery mechanisms of UMTS and of SIP.

9.2.1 UMTS Mobility and Call Delivery

The key elements of a UMTS Release 1999 network are as follows. The MSC is a switching and control system in a wireless network. The MSC controlling the service area where a mobile is currently located is called its serving MSC. It routes calls to and from all the mobile devices within a certain serving area, and maintains call state for them. Associated with the serving MSC is a Visitor Location Register (VLR), a database which stores information about mobile devices in its serving area. (For the purposes of this chapter we assume the predominant configuration in which the serving MSC and VLR are co-located.) Elsewhere in the fixed network there are two other classes of entities. A Home Location Register (HLR) maintains profile information about a subscriber and keeps track of his or her current location. A gateway MSC directs calls from the PSTN into the mobile access network.

When a UMTS mobile device first powers up or enters the serving area of a new serving MSC, it transmits a unique identification code, its International Mobile Subscriber Identity (IMSI) to the MSC. From the IMSI, the serving MSC determines the mobile's HLR and informs this HLR of the mobile's current location using the UMTS Mobile Application Part (UMTS MAP) protocol. The HLR stores this information and responds with profile data for the subscriber.

The procedure by which a call is sent to a mobile subscriber is illustrated in Figure 9.2. When a call is placed to a mobile subscriber, the public telephone network determines from the telephone number called (the Mobile Station ISDN number, or MSISDN) that the call is destined for a mobile telephone. The call is then directed to an appropriate gateway MSC. (In many countries, certain area codes are reserved for mobile numbers, so the PSTN can easily do this for domestic calls. For international calls, or for calls in countries such as the United States without such reserved area codes, mobile calls must instead be directed over the PSTN to a home MSC.) Call delivery from the gateway MSC is performed in two phases. In the first phase, the gateway MSC obtains a temporary routing number called a Mobile Station Routing Number (MSRN) in order to route the call to the serving MSC. For this purpose, the gateway MSC first locates the

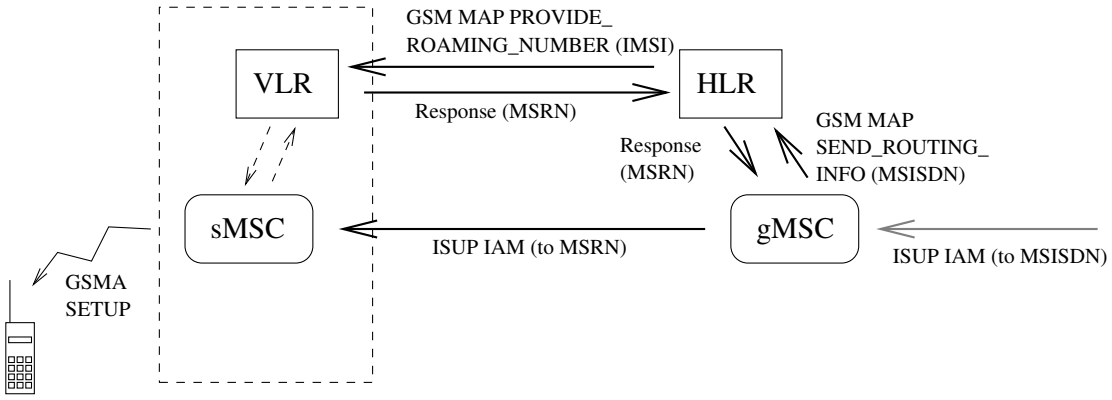

Figure 9.2: UMTS Call Setup Procedure

subscriber's HLR based on the MSISDN and requests routing information from it using UMTS MAP. The HLR then contacts the VLR at the serving MSC. The VLR returns an MSRN that the HLR forwards to the gateway MSC. In the second phase, the gateway MSC routes the call to the serving MSC using the standard ISDN User Part (ISUP) protocol of the PSTN.

The MSRN is a temporarily assigned number which is allocated at the time the HLR contacts the VLR; it is valid only until the associated call is set up, and it is then recycled. This dynamic allocation of an MSRN is required because ISUP messages can only be directed to standard telephone numbers, and the quantity of these that can be allocated to a given serving MSC is limited. This has some costs, however, in the time needed to set up a call, as the serving MSC must be contacted twice during call setup.

When a subscriber moves from one location to another while a call is in progress, two possible scenarios result: intra-MSC or inter-MSC handovers. An intra-MSC handover occurs when a subscriber moves between the serving areas of two base stations controlled by the same serving MSC. In this case, the serving MSC simply redirects the destination of the media traffic. No signalling is necessary over the PSTN or UMTS MAP. An inter-MSC handover, on the other hand, occurs when the subscriber moves from one serving MSC's area to another. The old serving MSC contacts the new one in order to extend the call's media circuit over the PSTN. The old serving MSC then acts as an "anchor" for both signalling and voice traffic for the duration of the call.

All of the globally-significant numbers used by the UMTS system — in particular, for the

UMTS	SIP
HLR	Registrar
Gateway MSC	Home proxy server
Serving MSC	End system (for REGISTER)
MSISDN	User address (in INVITE)
IMSI	User address (in REGISTER)
MSRN	Device (Contact) address

Table 9.1: Analogous Entities in SIP and UMTS

purposes of this chapter, the MSRN, and the identifying number of the MSCs, in addition to the MSISDN — have the form of standard E.164 [102] international telephone numbers. Therefore they can be used to route requests in Signalling System no. 7 (SS7), the telephone system's signalling transport network.

9.2.2 SIP Mobility and Call Delivery

Architecturally, a pure SIP network is rather simpler than a UMTS network, as it is significantly more homogeneous and much of the work takes place at the network layer, not the application layer. All devices communicate using IP, and all signalling occurs with SIP. (Much of the information in this section is also discussed in Chapter 2; however, the discussion of SIP mobility in this section serves as a parallel and counterpoint to the previous discussion of UMTS mobility.) Table 9.1 lists some analogous entities in UMTS and SIP networks.

When a SIP subscriber becomes reachable at a new network address (either because she is using a new network device or because her device has obtained a new IP address through a mobility mechanism), the SIP device sends a SIP REGISTER to the user's registrar to inform it of the new contact location. This registration is then valid for only a limited period of time. Because end systems are assumed not to be totally reliable, registration information must be refreshed periodically (typically, once per hour) to ensure that a device has not disappeared before it could successfully de-register itself.

Unlike systems that use traditional telephone-network numbering plans, addresses in SIP are based on a "user@domain" format, similar to that of e-mail addresses. Any domain can,

therefore, freely create an essentially unlimited number of addresses for itself. For the purposes of this discussion, it is useful to consider two types of addresses — "user addresses," analogous to an MSISDN number, to which external calls are placed, and "device addresses," roughly comparable to a non-transient MSRN. A device can create a temporary address for itself and have it persist for any period it wishes.

When a SIP call is placed to a subscriber's user address, a SIP INVITE message is directed to a proxy server in the domain serving this address. The proxy server consults the recipient's registrar and obtains his or her current device address. The proxy server then forwards the INVITE message directly to the device. Because the device address is not transient, the two-stage process used by UMTS is not necessary. Once the call is established, media flows directly between the endpoints of the call, independently of the path the signalling has taken.

Though not explicitly defined as part of the basic SIP specification, in-call handover mobility is also possible within SIP. A mechanism for an environment based entirely on SIP, with mobile devices which have an Internet presence, is described in [103]. This mechanism does not use Mobile IP [104], as Mobile IP suffers from a similar triangular routing issue as does circuit switching, and its handovers can be slow. Instead, it exploits SIP's in-call media renegotiation capabilities to alter the Internet address to which media is sent, once a device obtains a new visiting address through the standard mobile IP means. Therefore, Internet telephony calls can send their media streams to mobile devices' visiting addresses directly, rather than forcing them to be sent to the home addresses and then relayed by a home agent as in mobile IP.

There are two significant architectural differences between mobility in SIP and UMTS. First of all, a SIP network does not have an intermediate device analogous to the serving MSC. Instead, end systems contact their registrars directly, and proxy servers directly contact end systems. Second, in SIP a two-phase process is not needed to contact the device during call establishment.

9.3 Architecture

This section describes the proposals for interworking SIP and UMTS networks. In the design, UMTS mobile devices and their air interfaces and protocols are assumed to be unmodified. They use standard UMTS access signalling protocols and media encodings atop the standard underlying

framing and radio protocols. Some UMTS entities within the fixed part of the network, however, are upgraded to have Internet presences in addition to their standard UMTS MAP and ISUP interfaces. Serving MSCs send and receive RTP packets and SIP signalling. In some of the proposals other UMTS fixed entities, such as HLRs, have Internet presences as well. These entities still communicate with each other using UMTS MAP and other SS7 signalling protocols, however.[2]

Section 9.5 will discuss compatibility with existing infrastructure, in the case where serving MSCs are not IP-enabled.

There are two primary issues to consider when addressing this interworking: how calls may be placed from SIP to UMTS, and how they may be placed from UMTS to SIP. The latter point is relatively straightforward, and will be addressed first. The former is more challenging and represents the main focus of this chapter.

9.3.1 SIP/UMTS Interworking: Calls from UMTS to SIP

Calls originating from a UMTS device and directed at a SIP subscriber are not, in principle, different from calls from the PSTN to a SIP subscriber. The primary issue when placing calls from a traditional telephone network to SIP is that traditional telephones can typically only dial telephone numbers, whereas SIP addresses are of a more general form, based roughly on e-mail addresses, which cannot be dialed on a keypad. Work is ongoing to resolve this problem, but one currently envisioned solution is to use a distributed database based atop the domain name system, known as "ENUM," [106] which can take an E.164 international telephone address and return a SIP universal resource locator. For example, the E.164 number +1 732 332 6063 could be resolved to the SIP URI 'sip:lennox@bell-labs.com'. A SIP subscriber wishing to be reachable from the PSTN would obtain a telephone number in a special telephone exchange controlled by a switch which understands SIP. This switch would perform this ENUM lookup to obtain a SIP address, and then place the call over SIP.

Since globally significant UMTS numbers take the form of E.164 numbers, several of the proposals below use ENUM-style globally distributed databases in order to locate Internet servers

[2] It is possible that this SS7 signalling itself takes place over an IP network, using mechanisms such as the Stream Control Transmission Protocol (SCTP) [105].

corresponding to these addresses. However, for such databases it would not be desirable to use the actual global ENUM DNS domain, as the semantics of the URIs returned is different: they do not indicate generally-reachable end systems, but rather intermediate network nodes.

9.3.2 SIP/UMTS Interworking: Mobile-Terminated Calls

The most complex aspect of SIP/UMTS interworking is the means by which a SIP call is placed to a UMTS device. As discussed in the introduction, it is desirable to set up media streams directly between the calling party and the serving MSC. In order to accomplish this, SIP signalling must travel all the way to the serving MSC, as only the serving MSC will know the necessary IP address, port assignment conventions, and media characteristics.

In our model, the signalling between the serving MSC and the mobile device is unchanged from standard UMTS. This is actually a rather complicated procedure, involving communication between the serving MSC, base-station controllers, and base-stations. Devices may be in standby mode, requiring initiation of paging to locate them, or they may be turned off or in a region where no service is available, causing them to be unreachable. All these points, however, are elided in our descriptions of our architecture, as this complexity does not affect the nature of our arguments. Thus, for the purposes of discussion, this communication can be simplified into a simple pair of alert-answer messages between the serving MSC and the mobile device.

We propose three methods as to how SIP devices can determine the current MSC at which a UMTS device is registered. These have various trade-offs in terms of complexity, amount of signalling traffic, and call setup delay.

Proposal 1: Modified Registration

The first proposal is to enhance a serving MSC's registration behavior. The basic idea is that a serving MSC registers the mobile device not only with the subscriber's HLR, but also with a "home SIP registrar." This registrar maintains mobile location information for SIP calls.

The principal complexity with this technique lies in how the serving MSC locates the SIP registrar. The proposal, illustrated in Figure 9.3, is to use a variant of the ENUM database described above. Once the serving MSC has performed a UMTS registration for a mobile device,

it knows the mobile's MSISDN number. From this information, an ENUM database is consulted to determine the address of the device's home SIP registrar, and the serving MSC performs a standard SIP registration on behalf of the device.[3] A SIP call placed to the device then uses standard SIP procedures.

Because of authentication needs, this proposal uses either eight or ten UMTS MAP messages (depending on whether authentication keys are still valid at the VLR) and six DNS messages[4] per initial registration, and four SIP messages per initial or refreshed registration. Call setup requires a single SIP message and four DNS messages, though some DNS queries may be cached.

Compared to the other proposals, this proposal has two primary advantages. First, the only changes to the existing infrastructure are the modifications in the serving MSC and the addition of a variant ENUM database which locates registrars rather than end systems. Neither the SIP registrar and proxy server, nor the UMTS HLR and gateway MSC, need to be altered. Second, because the complexity of the proposal occurs only in registration, call setup shares the single-lookup efficiency of SIP and is therefore relatively fast.

The disadvantages of this proposal, however, also arise due to the separation of the two registration databases. First, once a system requires the maintenance of two separate databases with rather incomparable data, the possibility arises that the information in the databases becomes inconsistent due to errors or partial system failure. This is especially true because of the differing semantics of SIP and UMTS registrations — UMTS registrations persist until explicitly removed,

[3]Because they travel over the public internet, SIP registrations must be authenticated. In this model, the serving MSC and the SIP proxy must have some sort of pre-existing trust relationship established. The exact mechanism for this is for future study; however, most likely some sort of public key system, with a root certificate authenticating that a MSC is a legitimate UMTS provider, would be the best approach.

[4]Only two of these six DNS messages are shown in Figure 9.3. In addition, four DNS messages (two request/response pairs) are necessary to resolve the destination of a SIP request. The originator of the request must first perform an SRV query on the destination, which will return an A record giving an actual hostname. The returned hostname, or the original name if no SRV record was present for the host, must then be resolved with another query, to return the actual IP network address. (Some DNS servers may optimize these queries so that a response to an SRV query also contains response information to the corresponding A query, making it unnecessary, but this is not always possible, if for example the canonical server of the A record is different from that of the SRV record.) Thus, all the message counts in this section, and in Section 9.5, include four DNS messages for every SIP request sent, in addition to any DNS messages used for ENUM queries.

However, these DNS queries can often be cached, so the computations of signalling load in Sections 9.4 and 9.5 adjust the weight due to DNS queries by a probabilistic factor of how likely it is that the query was cached. In cases where one can be *certain* the query will be cached — as for refreshed registrations — no DNS queries are listed, or included in the computations.

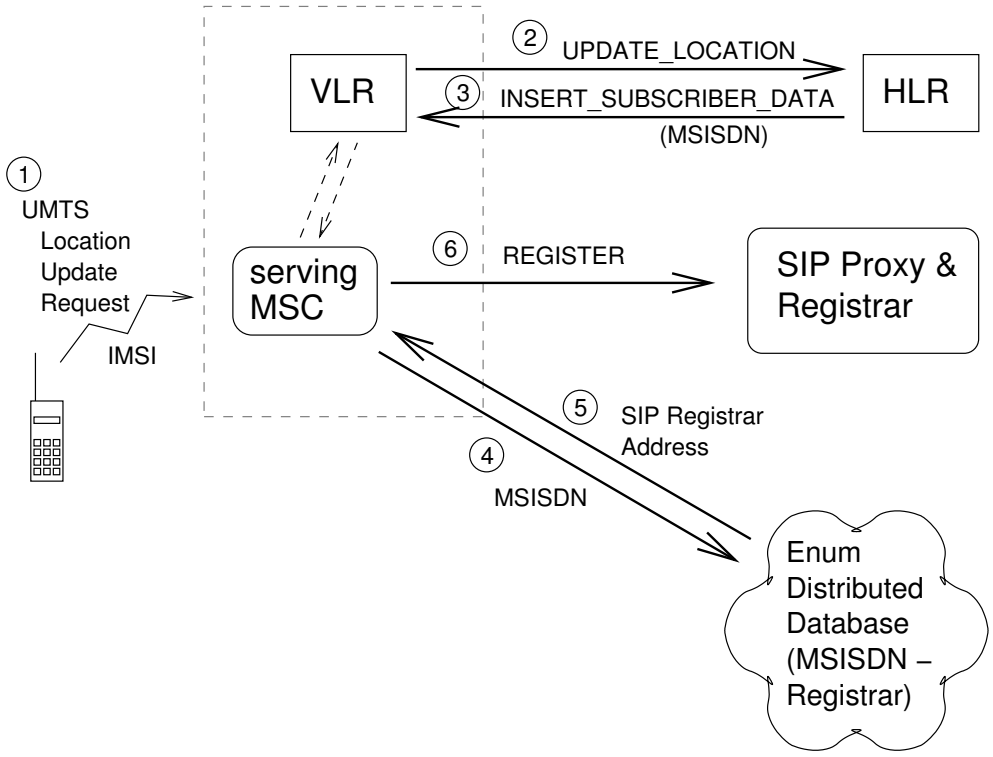

Figure 9.3: Registration Procedure for Proposal 1

whereas SIP registrations have a timeout period and must be refreshed by the registering entity. Furthermore, when mobility rates are low, the dual registration procedure imposes significantly more signalling overhead than UMTS registration alone, since SIP registrations must be refreshed frequently.

Proposal 2: Modified Call Setup

By contrast, the second proposal does not modify the UMTS registration procedure. Instead, it adds complexity to the call setup procedure. It essentially adapts the UMTS call setup to SIP. This is illustrated in Figure 9.4. When a SIP call is placed to a UMTS user, the user's home SIP proxy server determines the MSISDN corresponding to the SIP user address, and queries the UMTS HLR for an MSRN. The HLR obtains this through the normal UMTS procedure of requesting it from the serving MSC's VLR. The SIP proxy server then performs an ENUM lookup on this MSRN, and obtains a SIP address at the serving MSC to which the SIP INVITE

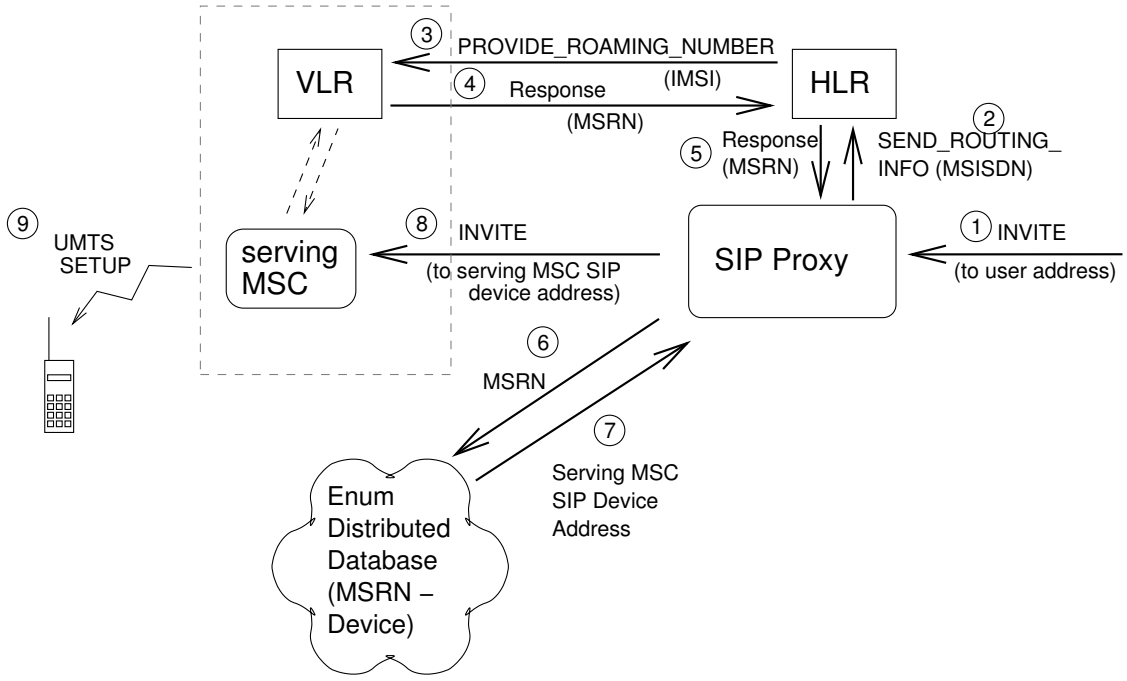

Figure 9.4: Call Setup Procedure for Proposal 2

message is then sent.

This approach uses either eight or ten MAP messages, as with standard UMTS, for registration, and four MAP messages, six DNS messages, and one SIP message for a call setup.

Because this proposal does not modify the UMTS registration database, it has several advantages over the previous proposal. Specifically, there is no possibility for data to become inconsistent, and the overhead of registration is as low as it is for standard UMTS. However, both the signalling load and the call setup delay are high, as call setup now involves a *triple*-phase query: a UMTS MAP query for the MSRN, an ENUM lookup for the SIP device address, and finally the actual call initiation. Additionally, there is a new requirement that the SIP proxy server and the HLR need to be able to communicate with each other. This imposes additional complexity in both these devices, as it requires new protocols or interfaces.

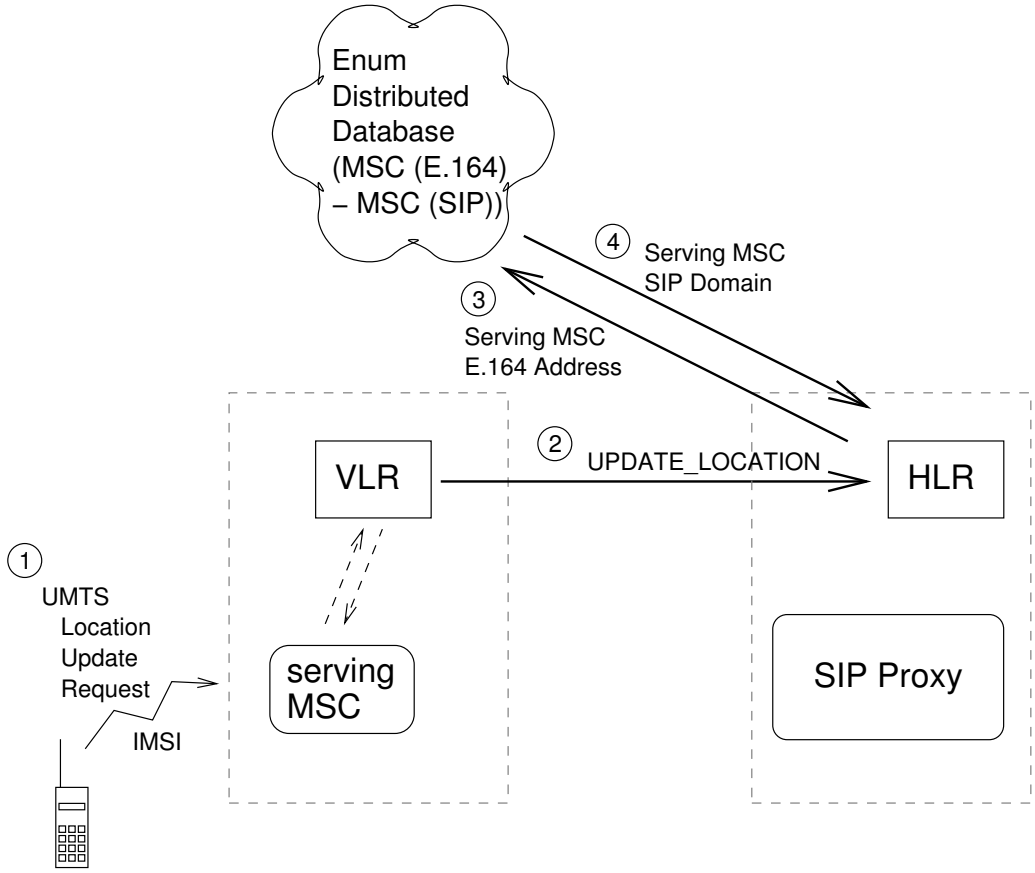

Figure 9.5: Registration Procedure for Proposal 3

Proposal 3: Modified HLR

The final proposal is to modify the UMTS HLR. In this proposal, the serving MSC registers the mobile at the HLR through standard UMTS means. The HLR then has the responsibility to determine the mobile's SIP device address at the serving MSC.

The overall registration procedure for this proposal is illustrated in Figure 9.5. When a serving MSC communicates with an HLR, the HLR is informed of the serving MSC's address, which, as mentioned earlier, is an E.164 number. The HLR performs a query to a specialized ENUM database to obtain the name of the serving MSC's SIP domain, based on the serving MSC's address. While the previous two proposals treat the SIP device address as an opaque unit of information whose structure is known only to the serving MSC, this proposal takes advantage of its structure.

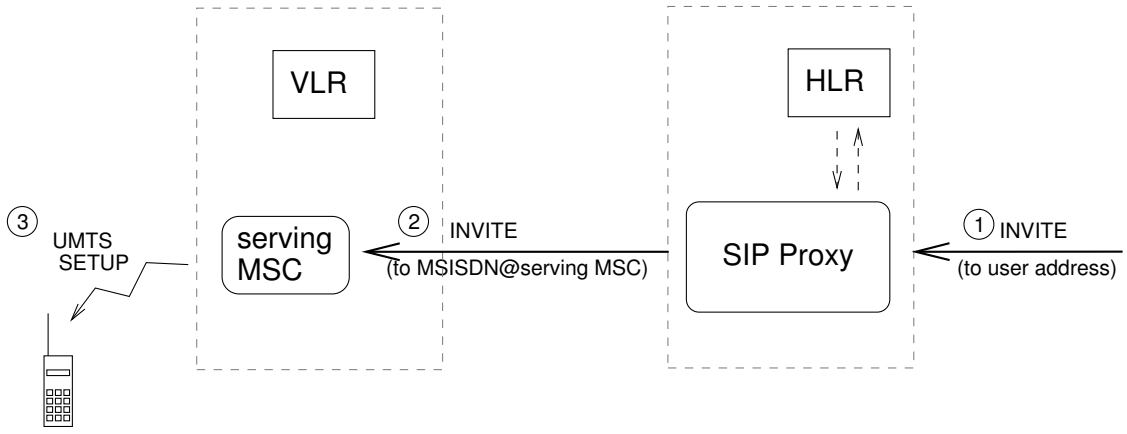

Figure 9.6: Call Setup Procedure for Proposal 3

Figure 9.6 shows how a SIP call is placed. The SIP proxy server queries the HLR for a SIP address and the HLR returns an address of the form "MSISDN@hostname.of.serving.MSC" to which the SIP proxy then sends the call. This proposal uses either eight or ten MAP messages, and two DNS messages, for registration, and four DNS messages and one SIP message for call setup. Because in this proposal the HLR and the SIP proxy are assumed to be co-located, the communication between them is local and therefore can be considered as "free."

This approach has the advantage that its overhead is relatively low for registration and quite low for call setup. The time requirements for call setup are similarly low. It does, however, require invasive modifications of HLRs. Additionally, the SIP proxy server and the HLR must be co-located, or else they must also have a protocol defined to interface them. (This is the work described by Haase *et al* in [95].)

9.4 Analysis

Two important criteria for evaluating the signalling performance of these three proposals for interworking SIP and UMTS are signalling load and call setup delay. A detailed study of call setup delay remains for future investigation. This chapter focuses on performance in terms of signalling load.

Each of the proposals involves the use of several different protocols, in varying ratios. In

Symbol	Parameter	Value
w_{sip}	Weight of a SIP message	1.0
w_{isup}	Weight of an ISUP message	1.0
w_{dns}	Weight of a DNS message	0.5
w_{map}	Weight of a MAP message	1.5

Table 9.2: Message Weights

order to compare total signalling load imposed by each protocol, we assigned signalling messages of each protocol a weight. The default values of these weights are listed in Table 9.2. The weights represent the impact each protocol has on the total signalling load of the system. The weights were chosen to reflect the complexity of each protocol, as well as the number of nodes and geographical distance each message must cross. The effect of these weights on the total signalling load are discussed in the sensitivity analysis later in this section.

Tables 9.3 and 9.4 list the parameters for the model. It assumes equal rates of originating and terminating call delivery r_{in} and r_{out}, as is commonly observed in European settings. We assign an exponential distribution to the probability $P_t(t)$ that a mobile remains in a particular MSC's serving area for longer than time t. DNS caching was accounted for by assigning the probabilities P_{nr}, P_{ur}, and P_{ns} to the likelihood that particular DNS queries have been performed recently, within the DNS time-to-live period.

Table 9.5 shows the equations for the weighted signalling loads for registration and call establishment in each proposal. These equations are based on the packet counts for each proposal in Section 9.3.

Figure 9.7 graphs the total weighted signalling load (registration plus call setup costs) for each of the three proposals, as both the incoming call rate and the call / mobility ratio vary. The intersection line at which modified registration and modified call setup are equal is shown in bold. Figure 9.8 shows the weighted signalling load as the call / mobility ratio varies but the incoming call rate is kept constant at 8.0 calls per hour. (It represents a two-dimensional cross section of the graph in Figure 9.7.)

From these graphs, one can observe some general characteristics of the proposals' signalling load. First, the modified HLR proposal consistently has the lowest signalling load of the

Symbol	Parameter	Value
$r_{\text{in}}, r_{\text{out}}$	Rate of call delivery / origination	variable
r_{bc}	Average boundary crossing rate	variable
$P_t(t)$	Boundary crossing rate prob. distribution ($P(t_0 \geq t)$)	$e^{-r_{\text{bc}}t}$
s	Call / mobility ratio	$\frac{r_{\text{out}}+r_{\text{in}}}{r_{\text{bc}}}$
P_{nr}	Prob. that a device is new to a serving MSC	50%
P_{ur}	Prob. that a device has a unique registrar at its serving MSC	20%
P_{us}	Prob. that a device has a unique serving MSC at its HLR/registrar	20%

Table 9.3: Mobility Parameters

Symbol	Parameter	Value
t_{sip}	SIP registration refresh interval	3 hr
t_{dns}	DNS cache time-to-live	24 hr
c_{auth}	Number of pieces of authentication data cached at VLR	5

Table 9.4: Protocol Parameters

Case	Formula
Modified Registration	
Registration	$r_{\text{bc}}((8 + 2/c_{\text{auth}})\, w_{\text{map}} + (2P_{\text{nr}} + 4P_{\text{ur}})\, w_{\text{dns}} + 4\,(1 + \sum_{i=1}^{\infty} P_t(it_{\text{sip}}))\, w_{\text{sip}})$
Call setup	$r_{\text{in}}\,(4P_{\text{us}}w_{\text{dns}} + 1w_{\text{sip}})$
Modified Call Setup	
Registration	$r_{\text{bc}}\,(8 + 2/c_{\text{auth}})\, w_{\text{map}}$
Call setup	$r_{\text{in}}\,(4w_{\text{map}} + 6P_{\text{us}}w_{\text{dns}} + 1w_{\text{sip}})$
Modified HLR	
Registration	$r_{\text{bc}}((8 + 2/c_{\text{auth}})\, w_{\text{map}} + 2P_{\text{us}}w_{\text{dns}})$
Call setup	$r_{\text{in}}\,(4P_{\text{us}}w_{\text{dns}} + 1w_{\text{sip}})$

Table 9.5: Weighted Packet Counts for Each Proposal

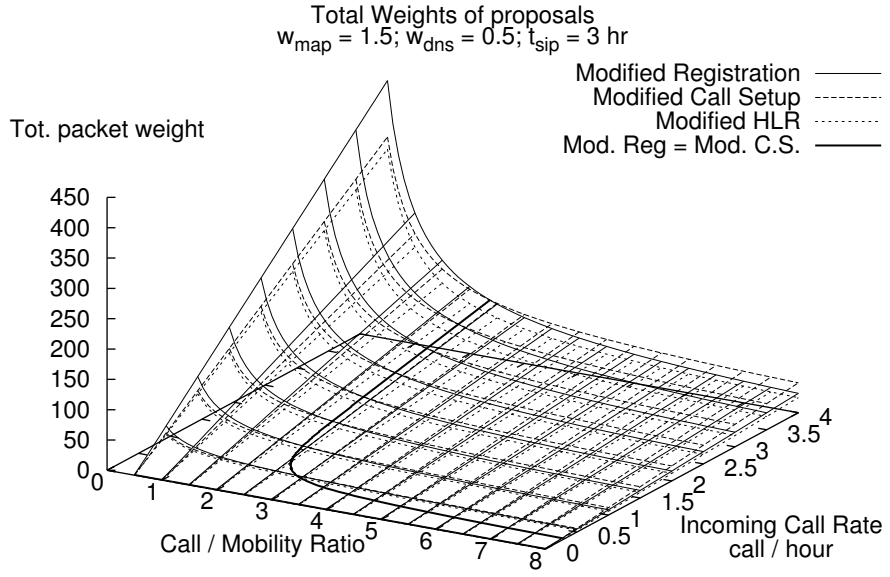

Figure 9.7: Weighted Signalling Load of the Three Proposals: Call Rate and Call / Mobility Ratio Both Vary

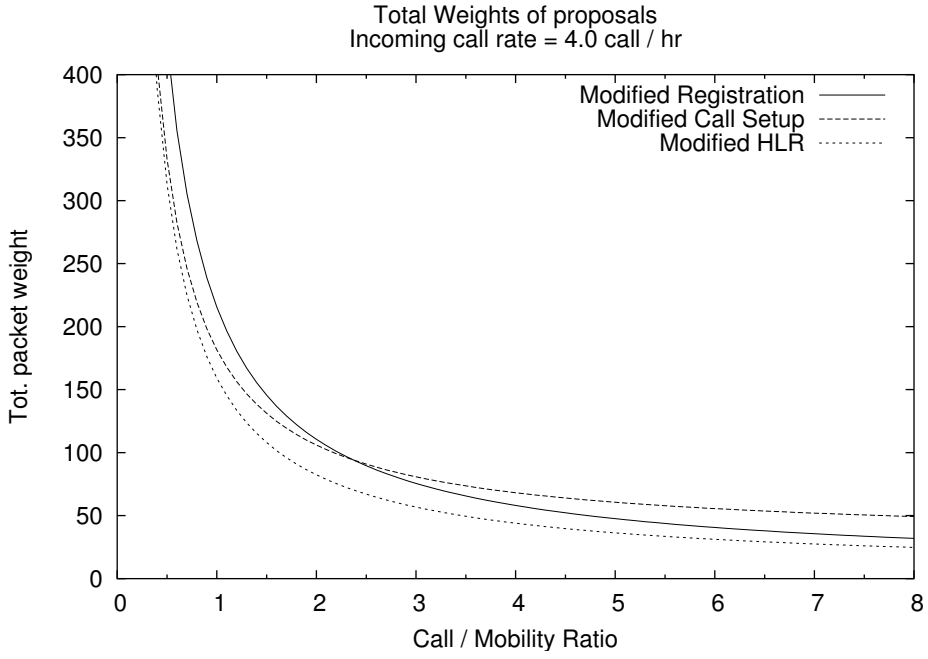

Figure 9.8: Weighted Signalling Load of the Three Proposals: Call / Mobility Ratio Varies

three, typically 20 – 30% less than the others. This corresponds to intuition, as it combines the "best" aspects of each of the other two proposals, unifying both an efficient registration and an efficient call setup procedure.

Second, the relative signalling loads for the other two proposals depend on the values of the traffic parameters. Modified call setup is more efficient for a low incoming call rate or a low call / mobility ratio (i.e., fast mobility), while modified registration is more efficient when both parameters are high. A closer look at the equations in Table 9.5 reveals the reasons. Consider the relative efficiency of the two approaches for varying incoming call rates: modified call setup performs less well for high incoming call rates because its call setup procedure requires four additional UMTS MAP messages and possibly two additional DNS messages compared to that of modified registration. Similarly, modified call setup outperforms modified registration for low call / mobility ratios because the latter has higher registration message overhead due to dual registration and SIP registration soft-state.

In order to increase the confidence in the above results, we performed sensitivity analyses to validate our choice of various parameters.

Sensitivity analyses for the weights assigned to MAP and DNS messages are shown in Figures 9.9 and 9.10, respectively. These graphs illustrate how, as the protocol weighting changes, the position of the intersection line in Figure 9.7 changes.

Figure 9.9 shows that as the weight assigned to the MAP protocol increases, the area in which modified registration is more efficient — the right-hand side of the graph, where call rate and call/mobility ratio are both high — increases as well. This fits with the intuitive understanding of the approaches, as modified registration uses fewer MAP messages than modified call setup. Similarly, Figure 9.10 shows that as the weight assigned to the DNS protocol increases, the area in which modified registration is more efficient shrinks slightly. This also fits with intuition, as modified registration uses more DNS packets. However, the total packet load is generally less sensitive to the weight assigned to DNS messages, which explains why the lines in Figure 9.10 are relatively close to each other.

The signalling load of the modified HLR proposal is always less than the other two. Thus, it is not shown in the sensitivity graphs. In regards to the other two protocols, though the

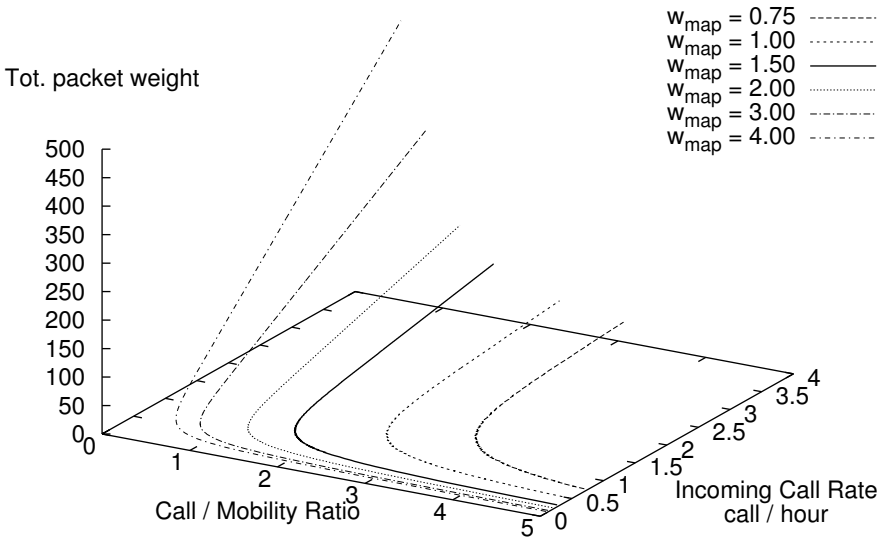

Figure 9.9: Line of Intersection: Modified Call Setup = Modified Registration (w_{map} varying)

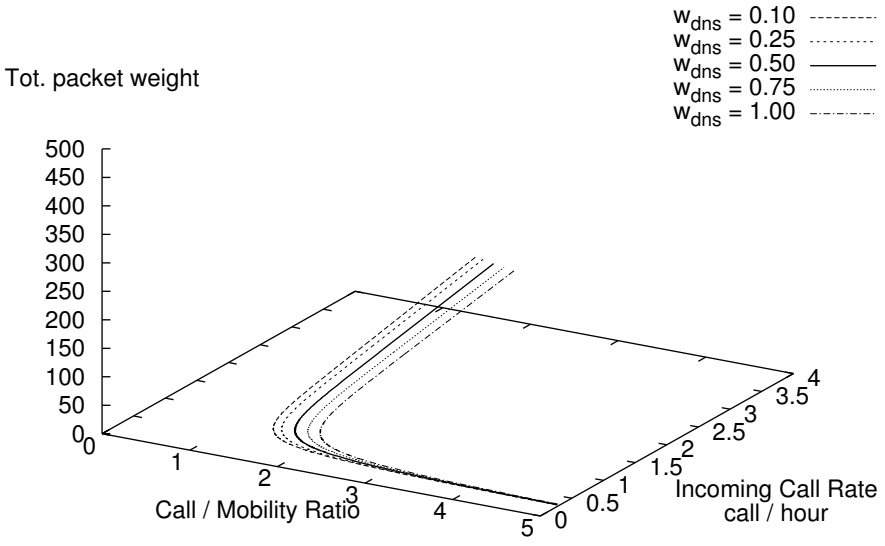

Figure 9.10: Line of Intersection: Modified Call Setup = Modified Registration (w_{dns} varying)

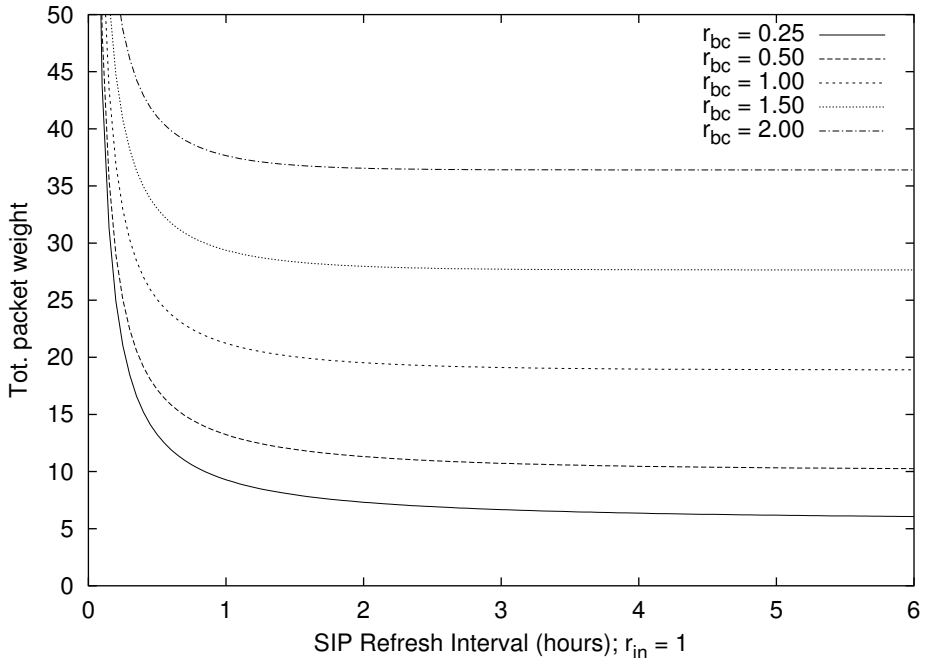

Figure 9.11: Total Weight of Modified Registration

crossover point moves as the weights assigned to the protocols vary, these sensitivity analyses show that the general shape of the graph, and therefore the conclusions that can be drawn from it, do not change.

Figure 9.11 shows the effect of various choices of values for the SIP registration timeout period. (This value only affects the modified registration proposal, as the other proposals do not use SIP registration.) The value for this parameter should be chosen so that the additional cost of SIP registration is relatively minor, that is, so that the graph has roughly flattened out. This optimal value therefore depends on the boundary crossing rate, but generally, a timeout of three hours is a good choice for most reasonable boundary crossing rates. This value can be larger than the standard value of one hour used by SIP, as serving MSCs can be assumed to be more reliable and available than regular SIP end systems.

9.5 Compatibility With Non-IP-Enabled Visited Networks

As has been demonstrated, using IP for wide-area communication to a serving MSC can

be much more efficient than using the circuit-switched network. However, the existing deployed circuit-switched networks cannot be ignored, and any system for connecting voice over IP networks to mobile telephony networks will have to be able to connect to networks which have not been upgraded to the new protocols.

As discussed in Section 9.1, both SIP and UMTS are designed to be able to interwork with the public switched telephone network. The entity which connects SIP to a circuit-switched network is called a *SIP gateway*. This gateway can terminates SIP and RTP connections from IP, and translate them into equivalent ISUP and circuit trunks on its circuit-switched side.

This same device can be used to interwork SIP and UMTS networks.[5] Conceptually, this can be viewed as decomposing the SIP-enabled serving MSC into two devices: a traditional circuit-switched serving MSC, and a SIP-enabled gateway that communicates with it. Indeed, each of the schemes described above could be implemented in this manner. However, in the general case, it must be assumed that the user's visited network has no support for voice over IP networks at all. In this case, the SIP system does not have the cooperation of the VLR and Serving MSC for registration, and no ENUM database has records for the serving network's E.164 number space.

The Telephony Routing for IP (TRIP) protocol [107, 108] is used to locate an appropriate gateway from SIP to the PSTN, based on a telephone number and on a provider's routing policy. Gateways can advertise routes to telephone numbers, with parameters indicating the "quality" of the route based on various criteria such as cost or geographic proximity. For SIP to UMTS routing, this means that a gateway can be located close to a telephone number, minimizing the amount of triangular routing needed to reach that number. This route advertisement takes place off-line — the advertised data is stored in a local database in or near a device which needs to consume the data, and therefore these lookups are "free" in terms of the call setup message flows.

9.5.1 Interoperation Approaches for the Three Proposals

Each of the three proposals for SIP-to-UMTS calls in Section 9.3 can support interoperation with non-IP-enabled systems in a different way. This section reviews techniques for interoperation for

[5] In standard UMTS, a pure SIP/RTP—ISUP/Circuit gateway can be used. If UMTS with Route Optimization, or ANSI 41, is used instead, the gateway will also need to be able to understand some UMTS MAP or ANSI MAP messages for some supplementary services.

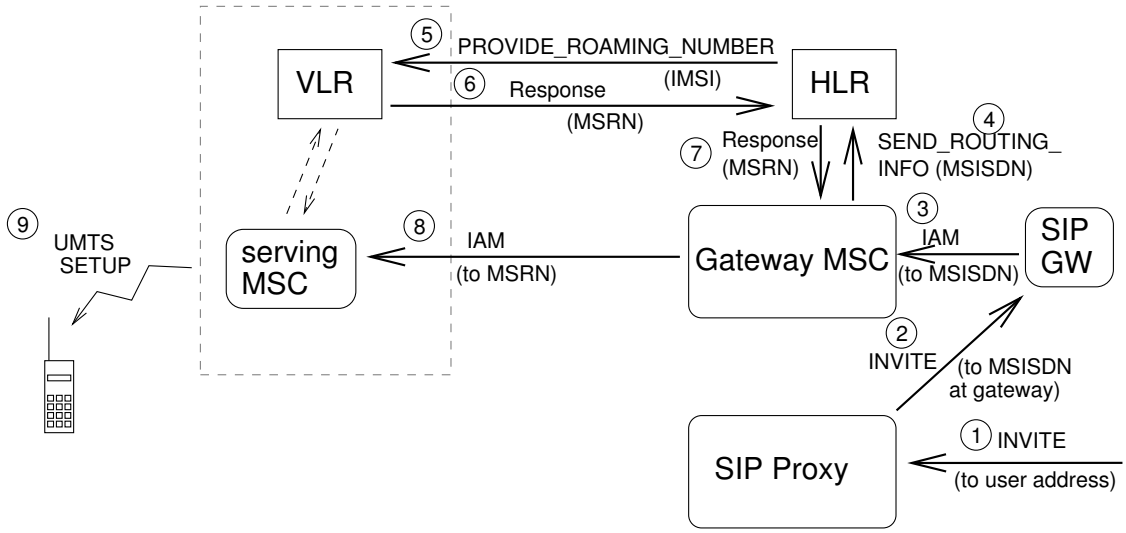

Figure 9.12: Call Setup Procedure for Proposal 1 — Non-IP-enabled Visited Network

each of the three proposals, and analyzes their relative signalling performance.

Non-IP-enabled Visited Networks with Modified Registration

The first proposal, modified registration, requires the serving MSC in the visited network to alter its registration procedure. The HLR and the SIP proxy server, in this case, are each unmodified.

In the interoperation case, however, we must assume the serving MSC is a standard UMTS device. In this case, therefore, the "modified registration" scenario does *not* actually involve a modified registration. Registration will simply be the standard UMTS registration procedure described in Section 9.2. No devices at all remain that have special knowledge of SIP and UMTS interworking, and so we must fall back to SIP–PSTN and PSTN–UMTS interworking.

In this scenario, when a SIP call is initiated, the SIP proxy discovers that the user is not at any SIP-enabled location. It does not know whether the user is at a non-SIP-enabled location, or is simply unreachable. To attempt to reach the user, it routes the call toward the user's MSISDN in the PSTN through an appropriate SIP gateway, and the PSTN then routes the call to a gateway MSC. The SIP gateway can either be discovered through TRIP, or pre-configured.

Thus, as shown in Figure 9.12, the call setup procedure for this procedure consists of a SIP INVITE message for the MSISDN at a SIP gateway, followed by the standard UMTS call

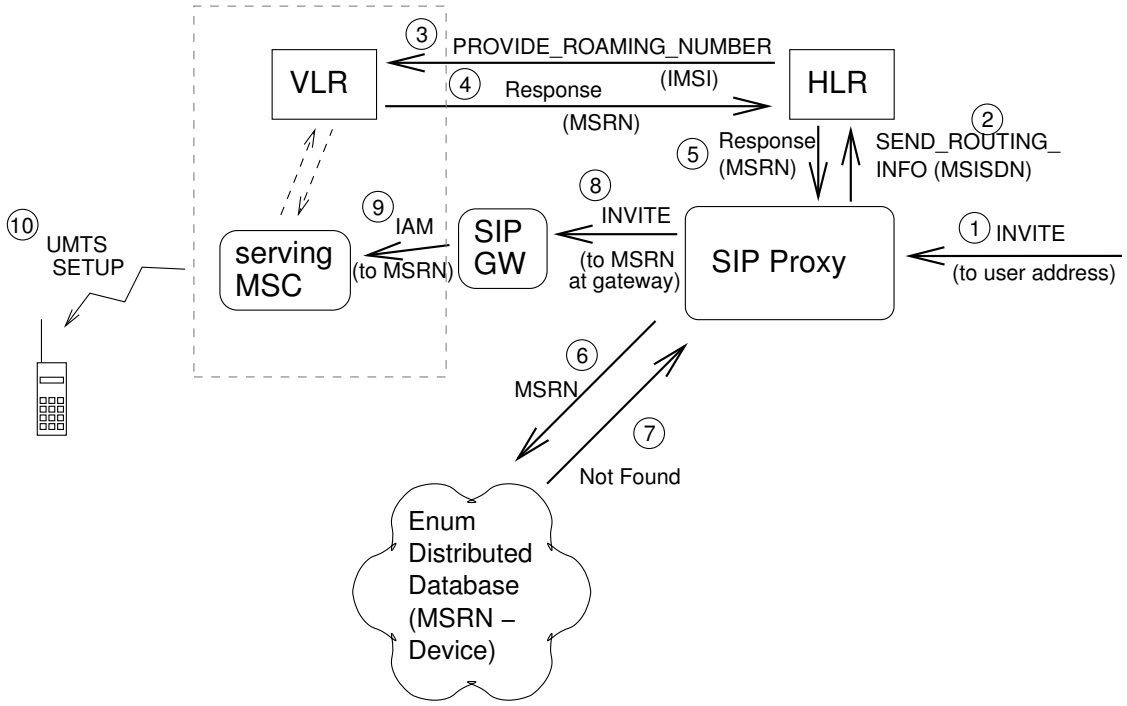

Figure 9.13: Call Setup Procedure for Proposal 2 — Non-IP-enabled Visited Network

setup procedure. Because the call must be directed to the MSISDN via the PSTN, connections to non-IP-enabled visited networks, under this proposal, do not avoid triangular routing.

In the non-IP-enabled visited network case, this proposal uses the standard eight or ten UMTS MAP messages for registration. Call setup requires one SIP message, two ISUP messages, and four MAP messages. We can assume that the SIP proxy has only a small number of SIP gateways which it wants to use to reach gateway MSCs, and therefore the DNS lookup for the SIP gateway can be amortized widely over all the users and therefore be ignored.

Non-IP-enabled Visited Networks with Modified Call Setup

In the modified call setup proposal, the SIP Proxy discovers that a serving MSC does not support SIP. As shown in figure 9.13, this occurs at call setup time, when the ENUM MSRN mapping database does not return a mapping from the MSRN to a SIP address.

In this case, the SIP proxy knows the MSRN to use to reach the user. Using TRIP, the proxy can thus locate a SIP gateway close to the serving MSC. Assuming that such a gateway

is available, therefore, this proposal therefore largely eliminates triangular routing even when visited networks do not support IP.

However, interoperation with non-IP-enabled visited networks makes this scenario's primary disadvantage, slow call setup, even worse. In this case, the lookup may potentially require *four* round trips between the originating and serving systems — the MSRN lookup; the failing ENUM lookup; potentially, the DNS lookup of the SIP gateway; and finally the SIP INVITE message to the SIP gateway. If we assume the SIP gateway is close to the serving MSC, however, the ISUP message sent from the SIP gateway to the serving MSC does not require another round trip.

This proposal uses the standard eight or ten UMTS MAP messages for registration. Call setup involves four MAP messages, six DNS messages, one SIP message, and one ISUP message.

Non-IP-enabled Visited Networks with Modified HLR

Finally, the proposal to modify the UMTS HLR is different from the other two proposals in that it can detect non-IP-enabled visited networks at registration time. As shown in Figure 9.14, when the modified HLR attempts to determine the serving MSC's SIP domain based on its E.164 address, it discovers that there is no such domain available. It therefore knows that calls for this user must be handled in a circuit-compatible manner.

Figure 9.15 shows the resulting call setup procedure. Because the call must reach the serving MSC through UMTS means, the HLR must initiate the standard MSRN lookup procedure. Once a MSRN has been assigned, a SIP gateway can be located for it, using TRIP. (This TRIP lookup can be done either by the HLR or by the SIP Proxy.) The call is then placed through the SIP proxy to the serving MSC.

Registration in this proposal requires eight or ten MAP messages and two DNS messages. Call setup requires two MAP messages, four DNS messages, one SIP message, and one ISUP message. As in the case when serving MSCs are IP-enabled, communication between the SIP proxy and the HLR can be considered to be "free."

Because this proposal discovers early on, at registration time, that visited networks do not support IP, in this environment this proposal is better than the other two both for the call setup

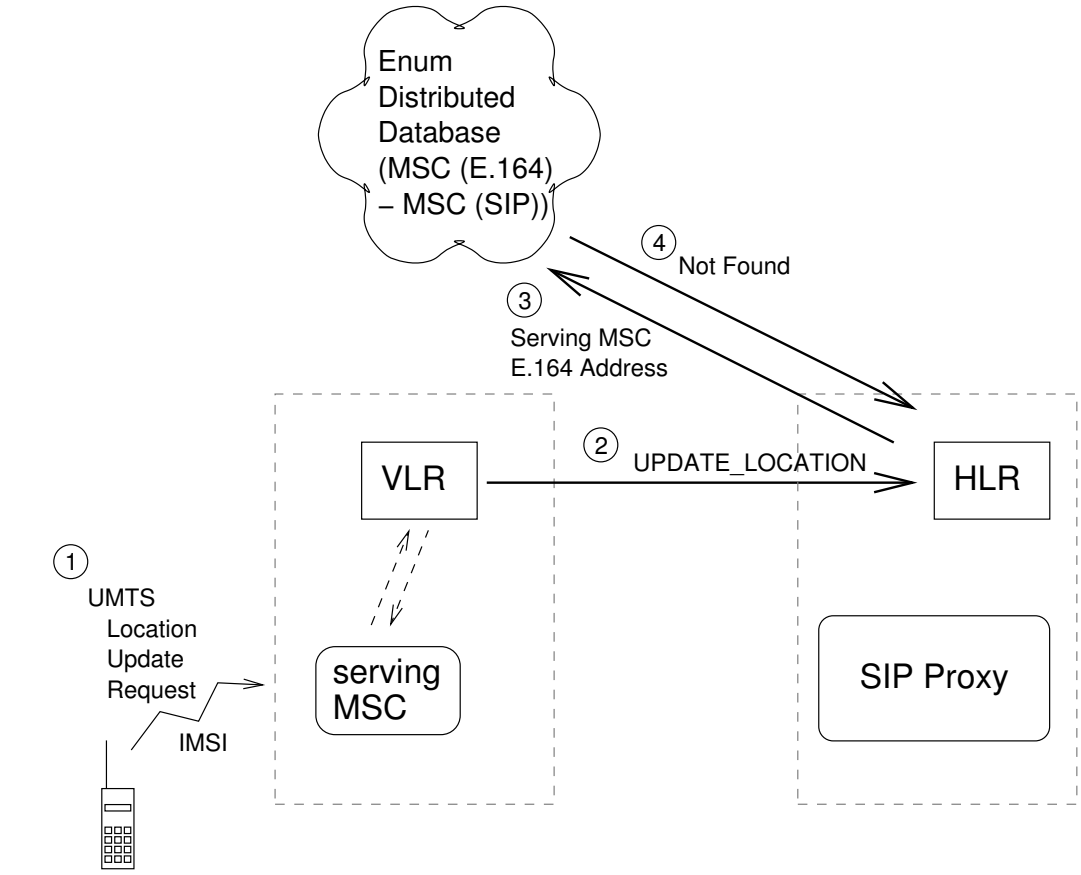

Figure 9.14: Registration Procedure for Proposal 3 — Non-IP-enabled Visited Network

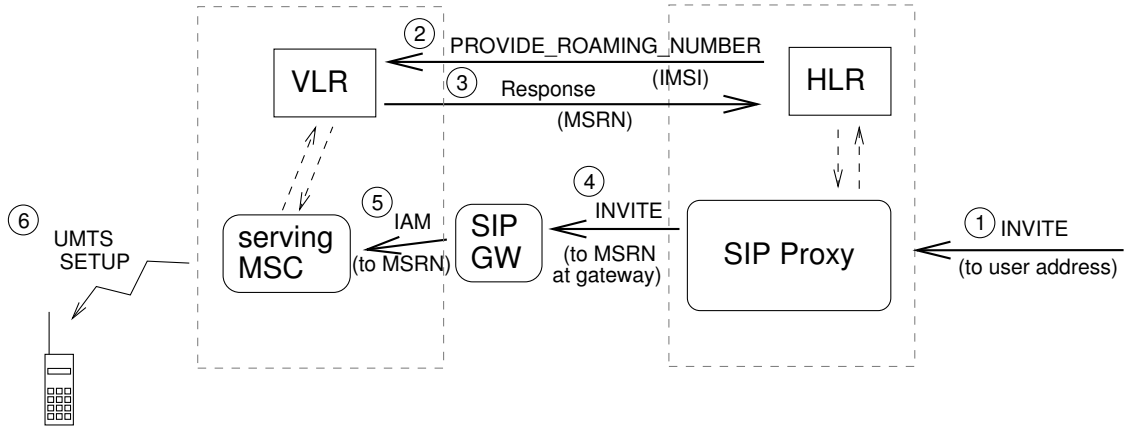

Figure 9.15: Call Setup Procedure for Proposal 3 — Non-IP-enabled Visited Network

Case	Formula
Modified Registration	
Registration	$r_{\mathrm{bc}}\left(8 + 2/c_{\mathrm{auth}}\right) w_{\mathrm{map}}$
Call setup	$r_{\mathrm{in}}\left(4w_{\mathrm{map}} + 1w_{\mathrm{sip}} + 2w_{\mathrm{isup}}\right)$
Modified Call Setup	
Registration	$r_{\mathrm{bc}}\left(8 + 2/c_{\mathrm{auth}}\right) w_{\mathrm{map}}$
Call setup	$r_{\mathrm{in}}\left(4w_{\mathrm{map}} + 6P_{\mathrm{us}}w_{\mathrm{dns}} + 1w_{\mathrm{sip}} + 1w_{\mathrm{isup}}\right)$
Modified HLR	
Registration	$r_{\mathrm{bc}}((8 + 2/c_{\mathrm{auth}})\, w_{\mathrm{map}} + 2P_{\mathrm{us}}w_{\mathrm{dns}})$
Call setup	$r_{\mathrm{in}}\left(2w_{\mathrm{map}} + 4P_{\mathrm{us}}w_{\mathrm{dns}} + 1w_{\mathrm{sip}} + 1w_{\mathrm{isup}}\right)$

Table 9.6: Weighted Packet Counts for Each Proposal: Non-IP-enabled Visited Network

delay and for the total message load. Additionally, as with the second scenario but in contrast to the first, triangular routing is still largely avoided. Because of the need for MSRN lookup, however, call setup for non-IP-enabled visited networks is still significantly heavier-weight than it is with IP-enabled networks.

9.5.2 Analysis of Non-IP-enabled Scenarios

Section 9.4 analyzed the performance of the three proposals in the ordinary cases, by assigning weights to every message (Table 9.2) and considering the total signalling load each protocol imposes on the network under a range of possible user behaviors (Table 9.4).

The behavior of the non-IP-enabled scenarios for the three protocols can be analyzed similarly. Table 9.6 shows the equations for the weighted signalling load for the three proposals in this case.

Figure 9.16 graphs Table 9.6 given the same assumptions as used in Figure 9.7, and Figure 9.17 gives a cross section in the same manner as Figure 9.8. The graph shows that when the visited network is not IP-enabled, the signalling load of the modified registration and modified call setup procedures are nearly equal. Indeed, analysis of the equations quickly shows that in this scenario the load of modified registration exceeds that of modified call setup by only $r_{\mathrm{bc}}\left(w_{\mathrm{isup}} - 6P_{\mathrm{us}}w_{\mathrm{dns}}\right)$, or $0.4r_{\mathrm{bc}}$ given the parameter values used for the graph. (Because this is a constant factor, the weights of modified registration and modified call setup never cross in this

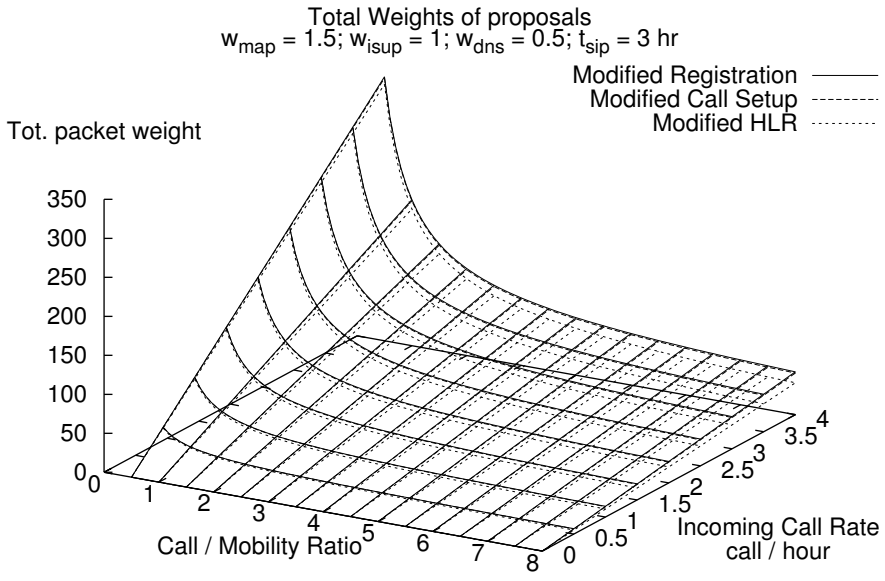

Figure 9.16: Weighted Signalling Load of the Three Proposals: Non-IP-enabled Visited Network: Call Rate and Call / Mobility Ratio Both Vary

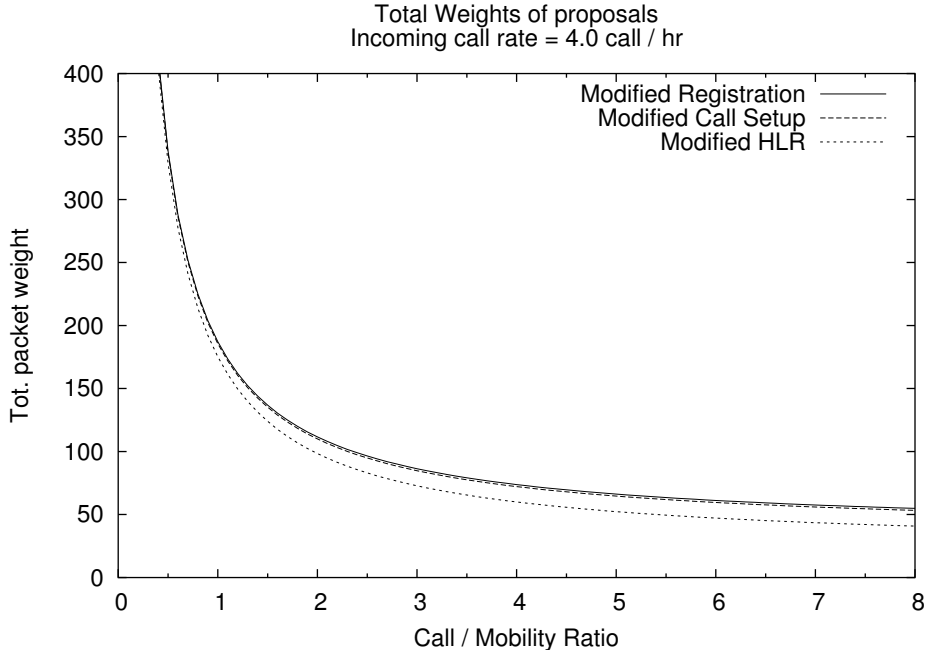

Figure 9.17: Weighted Signalling Load of the Three Proposals: Non-IP-enabled Visited Network: Call / Mobility Ratio Varies

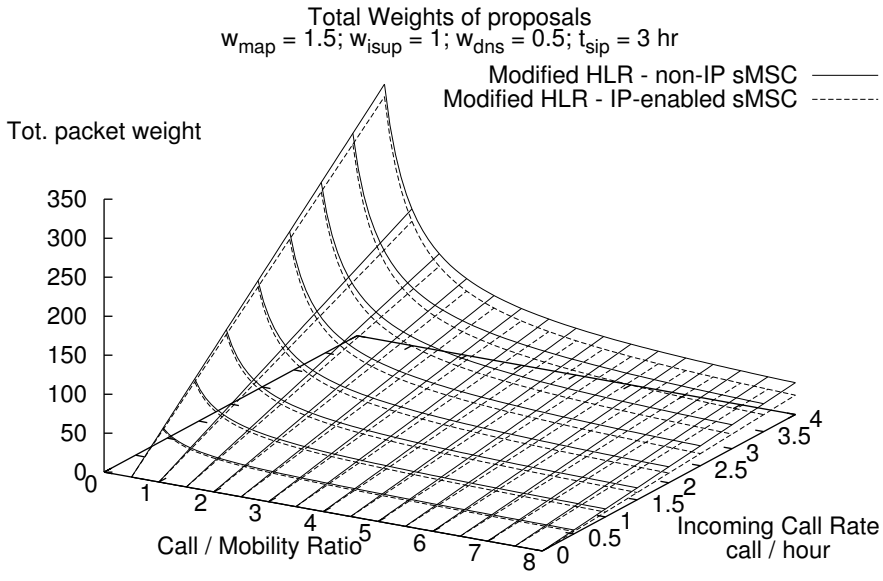

Figure 9.18: Comparison of Modified HLR Signalling Load With and Without IP-enabled Visited Network: Call Rate and Call / Mobility Ratio Both Vary

graph, so no line of intersection is shown in Figure 9.16.)

The modified HLR procedure is consistently better than the other two proposals in this environment as well. The amount by which modified HLR outperforms the other proposals depends strongly on the degree to which call setup dominates the weight, since the three proposals have very similar registration procedures in these scenarios. The signalling load of modified HLR is lower by a factor of only 2% when the call-mobility ratio is very low (0.5), but is 20% lower with a moderate call-mobility ratio (4.0) and 30% lower with a high call-mobility ration (8.0).

Figures 9.18 and 9.19 compare the weights of the modified HLR proposal with and without an IP-enabled visited network. We can see that the IP-enabled case is significantly more efficient than the non-IP-enabled case. As would be expected, since the registration procedure uses the same number of messages in both cases, the relative benefit of the IP-enabled case depends on how much the message flow is dominated by call setup. The load advantage of the IP-enabled case varies, from approximately 5% when the call-mobility ratio is very low (0.5), through 36% for a moderate ratio (4.0), to approximately 65% when the ratio is high (8.0). The

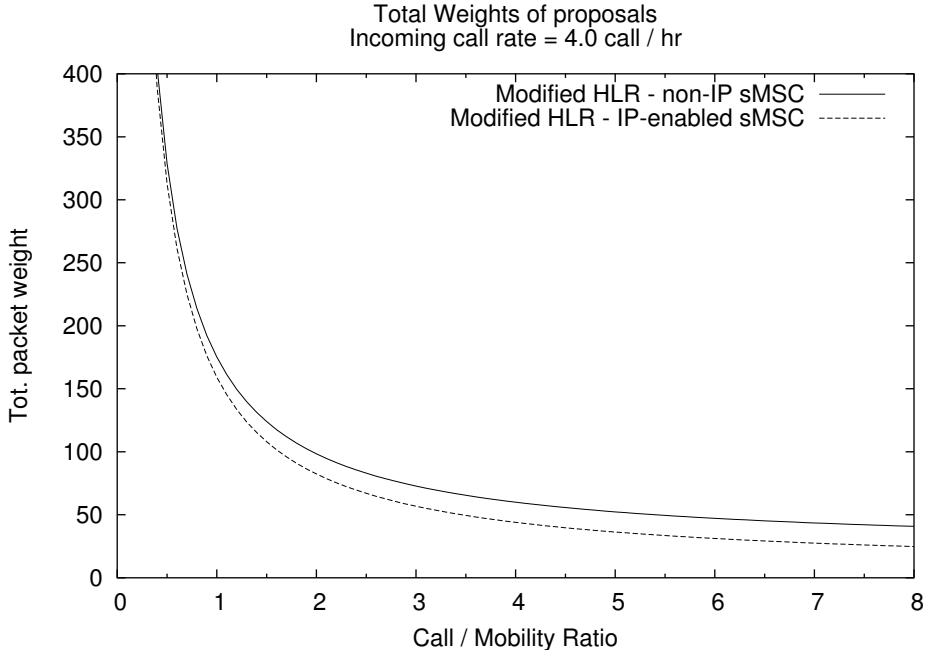

Figure 9.19: Comparison of Modified HLR Signalling Load With and Without IP-enabled Visited Network: Call / Mobility Ratio Varies

relative loads of the other two proposals are not shown, but are generally similar.

The comparative merits of the three proposals in the case of a non-IP-enabled visited network are therefore relatively similar to what they are in the case of the IP-enabled visited network described in Sections 9.3 and 9.4. Modified registration and modified call setup are roughly similar, and their relative merits depend on the exact assumptions made about packet weights and network characteristics. The modified HLR case is significantly better, though again it requires fairly invasive modifications of HLRs.

9.6 Discussion

The three proposed schemes to interconnect UMTS mobile and SIP Internet telephony impose different signalling burdens on the network. The modified HLR scheme (3) always imposes the least signalling burden, typically $20 - 30\%$ less than the other schemes. The efficiency of the other two proposals, modified registration (1) and modified call setup (2), depends on the

traffic parameters. When the incoming call rate and call / mobility ratio are both high, modified registration is more efficient. Modified call setup performs better otherwise.

In the case when we must interoperate with visited networks that do not support IP, the total signalling burden is higher, by about 36% in a typical case. The modified HLR scheme is still the most efficient in this scenario, with typically 20% less load than the other two proposals. The modified call setup and modified registration schemes result in nearly identical signalling load.

The modified HLR case therefore appears to be the most efficient of the three proposed scenarios that have been studied. However, it requires significantly greater modification to UMTS equipment. The other two proposals are roughly similar in efficiency. There relative merits depend on the environment in which they would be deployed.

9.6.1 Further Work

Our work addresses the issue of how calls can be set up to a SIP-enabled serving MSC. Full support of SIP-UMTS interconnection will require another issue to be resolved: interworking in-call handovers, in which a terminal moves during a call.

As explained in Section 9.2, there are two categories of in-call handover: intra-MSC and inter-MSC. Intra-MSC handover does not need to be treated specially for SIP-UMTS interworking. Because this happens between the serving MSC and the base stations, the network beyond the serving MSC is not affected. As an optimization, however, a serving MSC could use different IP addresses corresponding to different base stations under its control. In this case, a mechanism for SIP mobility as described before could be used to change the media endpoint address in mid-call.

Inter-MSC handover does affect SIP-UMTS interworking, and this issue remains for future study. We anticipate that a mechanism similar to that of [94], as described in the introduction, could be adapted to SIP for this purpose.

9.7 Conclusion

Three novel schemes were proposed to directly interconnect UMTS mobile and SIP Internet telephony systems. Compared with the conventional approach of routing a call through the PSTN, direct interconnection prevents triangular routing and eliminates unnecessary transcodings along its path. An analysis was made of the signalling message load of three proposals under a wide range of call and mobility conditions. The modified HLR scheme always imposes less signalling burden, although it requires significantly greater modification to UMTS equipment. The efficiency of the other two proposals, modified registration and modified call setup, depends on the traffic parameters. In the case when we must interoperate with visited networks that do not support IP, the total signalling burden is higher. The modified HLR scheme is still the most efficient in this scenario. We therefore conclude that the modified HLR scheme is the best of the three proposals.

The Unified Mobility Manager, as described by Haase, Murakami, and La Porta [95], is an implementation of (among other things) a modified HLR, inspired by the work in this chapter.

Chapter 10

Feature Interaction in Internet Telephony

Internet telephony services aim to replicate existing services from traditional telephony, to extend and generalize these services, and to implement entirely new services made possible by the Internet context. The architecture of Internet telephony is sufficiently different from that of traditional telephony to make it necessary to revisit the issue of feature interaction — the unexpected and undesirable interaction of multiple services and features — in this context. While many basic feature interaction problems remain the same, Internet telephony adds additional complications. Complications arise since functionality tends to be more distributed, users can program the behavior of end systems and signalling systems, the distinction between end systems and network equipment largely vanishes and the trust model implicit in the PSTN architecture no longer holds. On the other hand, Internet telephony makes end point addresses plentiful and its signalling makes it easy to specify in detail the desired network behavior. Many techniques for resolving interactions in the PSTN are no longer easily applied, but several new techniques, *explicitness*, *authentication*, and *verification testing*, become possible in the Internet environment.

10.1 Introduction

In telephone networks, feature interaction occurs when several features or services, operating simultaneously, interact in such a way as to interfere with the desired operation of some of the features. This problem of feature interaction still exists in Internet telephony, and it will become an increasingly pressing problem as more, and more sophisticated, services are created and deployed in this environment. The large amount of work that has been done to understand and resolve feature interactions in traditional telephone networks will help us to understand and control interactions in Internet telephony.

Internet telephony, however, is different in many ways from the PSTN. Some of these differences help resolve or prevent feature interaction problems, as the design of new protocols, and the characteristics of the underlying network, eliminate problems associated with legacy networks and systems. However, Internet telephony also introduces some new types of interactions; it also makes several techniques for preventing or resolving interactions more difficult or impossible.

This paper generally discusses Internet telephony in terms of the Internet Engineering Task Force's (IETF's) architecture for it [109], centered around the Session Initiation Protocol [1, 110], described in detail in Chapter 2. Many of the discussions and observation's also apply to H.323 [14], an alternative protocol developed by the International Telecommunications Union (ITU). However, the IETF architecture is generally better developed in areas such as inter-provider communications, areas in which Internet telephony's differences from the PSTN in feature interaction issues are more pronounced.

This chapter is organized as follows. Section 10.2 presents an overview of the architecture and component devices of Internet telephony. Section 10.3 details many of the differences between the PSTN and Internet telephony, both those that simplify service creation and allow new services, and those that make the feature interaction problem more difficult to resolve. Section 10.4 then discusses the applicability of existing approaches to solving feature interaction problems to the new environment. Section 10.5 gives some examples of new feature interactions that can occur in Internet telephony. Section 10.6 discusses some new approaches for resolving feature interactions in the Internet. The paper looks ahead to future work in section 10.7.

Internet Telephony	PSTN
End system	Customer-premises equipment, private branch exchange
Gateway	Signalling gateway
Signalling server	Service Control Point (SCP), Service Switching Point (SSP)
Router	Service Transfer Point (STP)

Table 10.1: Comparable Components of Internet Telephony and the PSTN

10.2 Internet Telephony Architectural Model

The architecture of Internet telephony is similar to traditional telephone networks in many ways, of course, but it also has some significant differences. Most fundamentally, Internet telephony is different from traditional telephone networks in that it, naturally, runs over the Internet, or more generally over IP networks. The most significant consequence of having this underlying network is that it provides transparent connectivity between any two devices on the network. Whereas devices in traditional networks are restricted to communicating with those devices to which they are directly connected, and the telephony protocols themselves must handle all location and routing features, Internet telephony can rely on an underlying infrastructure which provides all these capabilities automatically.

Specific entities in Internet telephony networks — *end systems*, *signalling servers*, and *gateways* — are discussed in Section 2.2. Table 10.1 lists the Internet telephony devices and their analogous devices in the PSTN.

10.3 Differences From the PSTN

Because of the effects of the Internet environment, Internet telephony has a number of differences from the traditional telephone networks; many of these differences will effect what sorts of features are possible, how these features are created, and how their interactions are managed. In general, the new flexibility the Internet gives telephony allows a wide range of new possibilities; however, this flexibility also introduces new challenges.

10.3.1 Advantages of Internet Telephony for Handling Feature Interaction

The advantages of Internet telephony, for handling feature interaction issues, can be broadly divided into three categories, which will be discussed in detail in the subsequent sections. First of all, this section summarizes the advantages that arise due to the design of Internet telephony protocols. Since there was an opportunity to design protocols "from scratch," a number of the difficulties present in traditional networks have been avoided by altering the underlying protocol architecture. Secondly, the section lists the advantages that arise from the infrastructure of the Internet itself. The Internet has been developed over the past decades to support a wide variety of types of services; many of these can be leveraged to provide powerful new abilities for the telephony environment. Finally, it describes those attributes of the Internet that are not as much its technical as its conceptual developments; the social and commercial evolution of the Internet has been substantially different from that of the PSTN, and this difference carries over to the social and commercial environment of Internet telephony.

Protocol Issues

Internet telephony signalling protocols are significantly more expressive than those of the PSTN. This is particularly true compared to the limited signalling of tones and hook signals available to two-wire analog telephones. Rich signalling in Internet telephony eliminates many previous limitations on feature development. For example, an end system no longer needs to indicate its desire to transfer a call through an elaborate sequence of switchhook and DTMF tones; it can explicitly indicate to its partner the party to which the call should be transfered.

Furthermore, Internet telephony signalling is extensible, and can be extended while maintaining compatibility. As new signalling properties or events are invented, they can be added to the existing protocol in ways which can interoperate cleanly with existing implementations, either by providing richer information about the signalling information or by allowing fine-grained control over what features are required to be understood in order to understand a signalling message successfully. Internet telephony devices can also query each other to determine what properties and parameters they support. As new signalling elements and capabilities are developed, the network will be able to evolve gracefully to support advanced features without needing to undergo

painful universal upgrades of an entire system.

In the past, difficulties have arisen in particular when trying to add new kinds of signalling capability, such as voice-mail control. In analog systems, only DTMF can be used, while even in ISDN, all such signalling would have to carried in user-to-user elements within existing signalling messages. In an Internet context, adding another control protocol, for example, RTSP [68] for voice mail or a presence protocol, can be done independently of the telephony signalling protocol.

Internet telephony enables the creation of new services that integrate telephone services with existing Internet protocols and services. Since Internet telephony addresses are URLs, the Internet telephony protocols have been designed so that "forwarding" or "transferring" a call to an e-mail address or a web page is not conceptually different than forwarding or transferring it to another telephone. Similarly, a signalling request can carry an arbitrary payload in its body — any media type which can be carried in MIME, the payload description mechanism of the web and e-mail, can also be carried in an Internet telephony request.

In addition, the real-time communications streams of Internet telephony sessions, while they can encompass traditional multimedia such as audio and video, are not limited to such types of communications. Because Internet telephony's signalling protocols separate the type of event (the beginning of a session, for instance) from the description of the stream, it is possible to use these same protocols to invite someone, for instance, to a multi-player game, or indeed to simultaneously invite to a game and voice communication.

One major difference of the Internet's telephony protocols from those of PSTN or ISDN networks is that the protocols the user's device uses to talk to the network (user-network interface or UNI) and the protocols that network devices use (network-network interfaces or NNI) to talk to each other are identical. Indeed, Internet telephony does not make a strong distinction between user devices and network devices; a device sending a request typically is not aware (and does not need to be aware) of whether it is communicating with to a signalling server or an end system. Because of this unification, Internet telephony deployment can scale from a few individuals running their own end systems, to a giant organization providing elaborate services and user location features; and these two organizations can interoperate cleanly. What's more, this means that even a customer of a large provider can choose to bypass the provider if his current needs don't require

Internet Telephony	PSTN
MAC address	Circuit identifier
IP address	Routing number (E.164)
SIP URL, H.323 alias	Telephone number, including 800/900 numbers

Table 10.2: Comparable Addressing Concepts in Internet Telephony and the PSTN

its services; for simplicity, flexibility, reliability, or privacy reasons, users can choose to communicate with each other directly end-to-end rather than through intermediate servers, without any need to modify their end systems.

Internet telephony protocols allow for capability labeling of end systems. In traditional networks, one often encounters the problem of a voice caller accidentally reaching a fax machine or modem, or vice-versa. Internet telephony, by contrast, prevents this in two ways: first, since the media type specifications for voice and fax differ, a voice-only end system will immediately reject the call with an "unsupported media type" error. On a broader scale, an end system can identify itself by the type of communication it supports; when a caller is searching for a destination, it can specify the type of communication desired in the call, and thus network devices can automatically resolve and prevent incompatible calls.

The Internet model eliminates user-level address scarcity. SIP and H.323 can use logical names (in the form of e-mail style identifiers) for telephone addresses. Thus, though the underlying routing numbers, IP addresses, are a scarce resource, Internet telephone addresses can be created in practically infinite quantity by any organization which possesses a DNS domain. PSTN telephone numbers, in contrast, are used both for routing calls and for identifying terminals or users; and as such, are a scare resource. In the PSTN, it is not generally possible to obtain "throw-away" identifiers, except by brief and temporary allocation as with an MSRN (see Chapter 9). When numbers in a certain geographic area are exhausted, an expensive and intrusive re-numbering is usually required. Table 10.2 lists comparable addressing concepts between Internet telephony and the PSTN.

This lack of address scarcity has a number of important secondary consequences. Telephone numbers have become more than just identifiers of telephone end points, but have been

overloaded to indicate a variety of network and end system properties. First, numbers can refer to a user, to a device, to a connection to a switch, a temporary routing number for a mobile call, or to a distribution point for a complex service such as a phone bank. They are used as lookup keys for a caller's location for emergency services (911) or commercial call centers. They also in some circumstances indicate carrier selection, which party is paying, or (in some regions) whether a device is a fixed-line device, a mobile phone, or a pager. Because Internet telephony addresses are "cheap," however, such overloading can be separated and eliminated. Thus, it is possible for each resident of a house to have his or her own address; for someone to maintain separate addresses for his general reachability, for each role that he has (home and work, for instance), and for each device that he owns; or for addresses to be assigned dynamically for temporary use, and discarded afterwards — all without imposing any more burden on the network or the numbering plan than a single telephone number would.

Network Issues

The nature of the Internet itself engenders a number of advantages that Internet telephony has over traditional circuit-switched telephone networks.

By their nature, circuit-switched networks, if they are to enable communication among huge numbers of people, require some sort of parallel signalling mechanism which enables circuits to be established. Because communication channels cannot be constantly maintained between every pair of stations that might wish to communicate, this parallel mechanism must be "self-routed" — an originating node specifies the destination of its signalling request, and the network sees to it that the request arrives at its destination; a circuit is established while this process takes place. The Internet, however, is inherently self-routing. Both signalling and media are sent off into the network through the same mechanism; thus there is no need for two parallel infrastructures to be maintained.

Additionally, because of the end-to-end nature of the Internet, the paths by which signalling and media traverse the network can be widely disparate. While in the PSTN signalling and media can indeed travel by separate routes, the architecture of that network still requires the two types of data to traverse the same key switches and administrative domains. In the Internet,

by contrast, the routes which signalling and media traverse can be entirely disparate — only the end points of the two paths need to be the same. Media packets are normally sent end-to-end — thus traveling over the "natural" route the Internet's low-level routing protocols have established between the endpoints — whereas signalling can travel across many servers which can provide elaborate third-party services.

Because IP is entirely packet-based, media communication is not limited to a single fixed-rate communications channel as it is in circuit-switched network. Internet telephony can, as appropriate for the environment in which it is being used, use very-low-bitrate speech encodings, or high-bandwidth video. Multiple media sessions can also be used in a single call, and these media sessions will inherently multiplex the communications channel between the endpoints. Bandwidth usage can even vary dynamically within a call depending on network conditions, with end systems stepping down to a lower-bandwidth encoding as a network becomes more loaded, then restoring higher quality once resources are again available.

Finally, the Internet environment supports a number of means of strong encryption and authentication, such as the IPsec suite of protocols [111]. These tools can secure communications and reliably guarantee that false information is not injected into end systems. Using the sophisticated algorithms and design techniques that have been developed in recent years in the fields of computer and network security, and by taking advantage of increases in processing power, communications can be made secure from eavesdroppers in manners never before possible. Security in the PSTN, by contrast, relies on the physical security of network cables and equipment; this is generally both more expensive to accomplish, and less reliable in the long run.

Conceptual Issues

The conceptual framework of Internet services also gives rise a number of new characteristics of the Internet telephony environment. First of all, whereas the PSTN is gradually moving to an increasingly distributed environment where multiple providers must interwork and compete on an fine-grained level, the Internet is already at such a level, and shows no signs of moving away from it. Thus, services can be provided by third parties — organizations dedicated only to providing services, with no intention of providing actual voice or multimedia transport — as easily as they

can be by the original provider, and indeed providers may well specialize into service provision or data transport, as these are rather separate tasks.

The broadly distributed environment also introduces some new possibilities in terms of trust models for Internet telephony. It is relatively easy in Internet telephony for a customer to proxy all his calls through a service which, for example, automatically blocks calls from known telemarketers. A traditional telephone company does not have much interest in providing such a service — and few customers would likely trust a telephone company to provide it reliably, as telemarketing calls provide the company with revenue. The introduction of the distributed network allows users to have trust relationships with organizations other than their service provider.

Additionally, the Internet environment enables programmability on a scale not seen in the telephone network. Following the precedent of web services, we see that the Internet's distributed nature will give rise to programmability on a scale unprecedented in PSTN networks. This has several causes. First of all, the rich communications media and sophisticated processing possible for even low-end users allow complex feature descriptions to be passed in real time. For example, the Call Processing Language, described in Chapter 5 allows users to design and upload scripts to network signalling servers. Real-time control of PSTN services, by contrast, is generally not terribly powerful; a user can typically at best set either a single parameter, or turn the feature on or off. Even when a user is specifying features off-line to his provider, he usually has only a checklist of possible features available; sophisticated controls which allow loops, branches, or user-settable timers are not possible.

The wide variety of providers available, and the fact that users will be able to use any provider of services regardless of who their data connections come from, will give service providers a strong motivation to create services which will distinguish them from their competitors. A single, standardized list of enumerated features which customers can choose among does not give service providers much to distinguish themselves from the pack, so we envision that providers will quickly develop more sophisticated, distinctive features instead.

10.3.2 New Complications

The new features of the Internet introduce, however, a significant number of additional complications to the problem of creating and deploying features and resolving their interactions. Most of these problems are the "flip side" of new features described in the previous section; while the new characteristics of the Internet enable new possibilities, they also increase the complexity of creating features.

The most significant of these new complications is the *distributed nature of the Internet* itself. Features can be implemented and deployed at numerous network devices, both end systems and signalling servers. What's more, these systems may well be controlled by entirely separate organizations, which may be unaware of each other or even competing, and thus will not generally be inclined to co-operate to resolve feature interactions.

Additionally, because user programmability is now possible, the new phenomenon of *features created by amateur feature designers* arises. Because new services can be created and deployed with much the same level of ease that, for example, dynamic web pages can be created today — a simple service can be put together by a reasonably experienced programmer in a matter of hours — they may be created by programmers who may not consider feature interaction issues thoroughly, either through ignorance or expediency. Such distributed problems may be dismissed as the "just desserts" of customers of incompetent feature designers, but unfortunately other service providers will have to interoperate with such services.

On a network level, the characteristics of the Internet also introduce some new complications. First of all, the fact that *media packets travel end-to-end*, without being interceptable by intermediate servers, means that intermediate servers can no longer implement a number of features transparently. For instance, ordinary signalling servers cannot listen in on calls to collect digits ("press '#' for new call"); perhaps more significantly, they cannot perform "pipe-bending" services, where an intermediate system moves one endpoint of a call from one end system to another — for example, to transfer a call — without explicitly informing the end systems of the new locations to which they should send their media packets. (It should be noted that, architecturally, an Internet telephony server *can* forego this feature of the Internet, and instruct end systems to route their media packets through an intermediate media gateway, which can perform

these pipe-bending or media-stream-listening services. This is known as a "back-to-back user agent" (B2BUA). The overhead this implies, due to the re-introduction of triangular routing, makes it undesirable in most cases; however, some features, such as call anonymizers, require it.)

Another related complication is the fact that *end systems have control of call state*. While this introduces many new possibilities for general feature creation and deployment, it also complicates issues in situations when the network wants to be able to impose control contrary to the expressed desires of an end system. For example, in traditional telephone networks, 911 (emergency) calls are usually handled specially, so that end systems cannot hang them up; the emergency operator must hang up the call before the line is cleared. If the end system controls its own states, however, it is impossible for the network to enforce this without the end system's cooperation.

Several new features of Internet telephony protocols also have the potential for dramatic feature interaction consequences with existing protocols. Probably the most dramatic of these is what is known as the *forking proxy*. A signalling server, or proxy server, can take an existing call request and transmit it in parallel to several other devices. We discuss some examples of complex interactions that can occur with this feature in Section 10.5.1.

Another new feature is *request expiration*. A request, when it is placed, can specify how long it should be considered valid — a user might want a call to only ring for the equivalent of four rings, for example — but services on subsequent signalling servers may be programmed to do different things when the expiration time elapses.

The Internet's *lack of address scarcity* can also complicate some common features. In traditional telephone networks, where telephone numbers are difficult to obtain, a telephone number can be used, reasonably effectively, as a representative of a party's identity for such purposes as incoming or outgoing call screening. In the Internet, however, "throw-away" addresses become easy to use; someone wishing to evade a block on their address can switch to another one with minimal effort.

Related to this problem is the Internet's *trust model*. In the PSTN, telephone users generally assume that they can trust their telephone company to provide accurate information, that their telephone company will not reveal private information to third parties when inappropriate,

and that the wire leading out of their house indeed connects to the telephone company and no one else. Telephone carriers, meanwhile, can assume that the signals they get from a subscriber line are indeed coming from that subscriber; and signals they get from other telephone companies are reliable and secure. All these assumptions break down when end-to-end connectivity is introduced and anybody can become an Internet Service Provider. Forging communications becomes relatively straightforward when packets may be sent from any location on the network to any other, and intercepting them, while somewhat more difficult, is still significantly more tractable than on a telephone network, due to Internet characteristics such shared-bandwidth communications channels and dynamic routing protocols. While protocols for strong authentication and encryption have been developed, deployment of a key infrastructure which would enable large-scale trust is still a long way off.[1]

Additionally, features like "caller I-D blocking" become much more difficult when users cannot trust the network not to reveal calling information to recipients — and indeed cannot reliably distinguish whether they are communicating with a "network" or a "customer."

10.4 Applicability of Existing Feature Interaction Work

There has not been much work as yet on feature interactions in Internet telephony. One paper by Kenneth Turner [112] applies formal methods to CPL and VoiceXML scripts, and Nakamura *et al* [113] apply an automated detection approach to discover feature interactions in CPL scripts. (Both these papers draw upon an earlier version of this chapter presented at the Feature Interaction Workshop in May of 2000 [8].)

In terms of the categorization of feature interaction problems in the Internet environment, and the ways in which this environment differs from the PSTN, very little has been done. However, earlier work on PSTN feature interactions is applicable to the Internet environment in some circumstances. If we consider the framework of Cameron et al. [114], single-component interactions (those where all the interacting features are implemented on the same network component)

[1] In addition, many of the existing user-level certification services simply assure that the presenter of the signed request can indeed be reached by the (e-mail) address indicated, but do not associate a legal or civil identity with a key.

are largely the same in the Internet environment as they are in traditional telephone networks, and we expect the techniques developed to resolve these interactions to work in the new environment.

An example of single-component interaction that can be dealt with in the Internet as it is in the PSTN is Cameron et al.'s Example 1, the interaction between *Call Waiting* and *Answer Call*. These two features have conflicting definitions of what should occur when a call attempts to reach a busy line: to signal the user with a tone, or to connect the calling party to an answering service, respectively. If, in an Internet telephony environment, both these services are deployed in the same device, or in multiple devices controlled by the same organization, techniques for resolving their interaction would carry over naturally from the PSTN.

Multiple-component interactions, however, are much more complicated for Internet telephony. The problem arises as features are designed and deployed by providers who do not cooperate, and have no interest in doing so; therefore, feature interaction resolution techniques which depend on being able to describe features globally, and resolve their interactions at the time they are designed, are no longer practically applicable. (This is, of course, a growing problem in the PSTN as well, as increasing numbers of providers enter the market.)

10.5 Examples of New Interactions in Internet Telephony

Several varieties of new feature interactions appear in Internet telephony which either do not appear or are not as severe in traditional telephone networks. We categorize these into two types of interactions: *cooperative* interactions are those where all the parties who implement features would consider the others' actions reasonable, and would prefer to avoid an interaction if it were possible. *Adversarial* interactions, by contrast, are those where the parties involved in the call have conflicting desires, and one is trying to subvert the other's features. Roughly, cooperative interactions correspond with those that Cameron et al. [114] describe as single-user multiple-component (SUMC) interactions; adversarial interactions are more commonly multiple-user multiple-component (MUMC) or customer-system (CUSY) interactions.

10.5.1 Cooperative Interactions

"Cooperative" feature interactions are multiple-component feature interactions where all the components share a common goal — typically, allowing the caller to communicate with his or her intended called party — but have different and uncoordinated ways of achieving that goal. These conflicting implementations can interact in ways that can prevent the most desirable means of communication from occurring, even though it would be possible given the state of the parties involved; and can result in surprising or unpredictable consequences of deployed services.

Example 1 *Request Forking* and *Call Forward to Voicemail*

Request Forking allows an Internet telephony proxy server P to attempt to locate a user by forwarding a request to multiple destinations, A and B. The call will be connected to the first destination to pick up, and the call attempt to the others will be canceled. The interaction arises when the user to be reached is currently located at A, and another, B, has had its calls forwarded to a voicemail system. The call to B will be picked up first, as it is an automated system, and thus P will connect the call from B and cancel the call from A. The caller will never be able to reach the actual human.

Example 2 *Multiple Expiration Timers*

A SIP request may specify a length of time for which the request is valid. Difficulties arise, however, if several servers are programmed to have special behavior if the timeout elapses before the call has been definitively accepted or rejected. For example, one proxy server P_1 may be programmed to forward a call to a voicemail server when the expiration has elapsed, whereas another server P_2 may respond with a web page giving alternate ways of contacting the destination. If P_1 is earlier in the call path of P_2, the former server considers the latter server's response to be a definitive response to the call; and if P_2's response arrives at P_1 before its own timer expires, P_1 will forward that response back to the original caller rather than triggering its own expiration behavior. The two timers have the same nominal expiration period (the length of time specified in the request); which one executes first depends on factors such as processing time and the precision of the two servers' clocks. Therefore, there is a race condition of which of the two expiration-related services will be executed.

Example 3 *Camp-on* and *Call Forward on Busy*

Camp-on allows a caller who reaches a busy destination to continue to re-try that destination periodically until the line becomes free. However, if the destination has *Call Forward on Busy*, the call is forwarded to some alternate destination in this case, and the caller never receives the busy indication; thus there is no way to trigger the camp-on service. This is an interaction which can also arise in the PSTN, but it is more serious in Internet telephony for several reasons. First of all, because Internet telephony places so much additional power and call state knowledge into end systems, Call Forward on Busy is likely to be triggered by intelligent services implemented an end system, which may not be aware that the other party is attempting to camp on. PSTN switches which try to camp on will generally also be the location where Call Forward on Busy is implemented, and thus can resolve the interaction locally. Furthermore, camp-on services in the Internet will generally need to be globally usable; users will not accept camp-on services which work only within one provider's network, so state cannot be shared easily among servers in a private manner either.

10.5.2 Adversarial Interactions

"Adversarial" feature interactions, by contrast, are those where several of the parties involved — the caller, the destination, and/or either endpoint's administrator — disagree about something having to do with the call, typically about whether it should be allowed to be completed. These can be more difficult to resolve reliably than cooperative interactions, because generally parties attempting to subvert others will find ways to lie to them, or bypass them. They are also more complicated because users will generally be quite upset if the network allows their expectations about security or privacy to be violated.

Example 4 *Outgoing Call Screening* and *Call Forwarding*

Outgoing Call Screening blocks calls at an originating party based on the address to which a call is placed. However, even if a Call Screening service blocks calls to an address X, another signalling server, downstream from the location where the blocked is imposed, may forward calls originally directed to a non-blocked address Y to the blocked address X. This interaction also appears in the PSTN, of course (and this description is largely taken from [114]), but the ability

to easily change addresses and get easy call forwarding on the Internet makes this problem much more significant in the Internet environment.

Example 5 *Outgoing Call Screening* and *End-to-end Connectivity*

Because the Internet provides end-to-end connectivity, enforcement of *Outgoing Call Screening* policy is difficult for another reason. A signalling server cannot force calls to be placed through it; because the Internet telephony UNI and NNI protocols are identical, and because (in the absence of firewalls) any device can talk to any other, an end system can be programmed to communicate directly with the remote party, bypassing local administrative controls entirely.

Example 6 *Incoming Call Screening* and *Polymorphic Identity*

Incoming Call Screening allows a called party — either in a signalling server or an end system — to reject calls from certain callers automatically. Because Internet telephony addresses are cheap, however, and because the caller can switch the identity he presents in his call request, he can easily alter the address he presents as his own in order to evade the screening lists the destination has programmed her phone to reject.

Example 7 *Incoming Call Screening* and *Anonymity*

Even in the absence of a malicious caller, *Incoming Call Screening* can be complicated by a caller's legitimate desire for anonymity. Because the trust model of the Internet does not allow a user to be sure that a network provider will hide the information like caller ID, if a user wishes to be anonymous he must avoid sending all identifying information in the signalling information in the first place — and for assured anonymity will likely have to use an anonymizing server run by a trusted third party, which will hide all information, including the sender's IP address for transmission of media packets and signalling. In the PSTN, a destination switch can easily apply *Incoming Call Screening* and *Caller I-D Blocking* services simultaneously; and both the caller and destination can trust this switch to apply their service reliably. In Internet telephony, however, there are not generally such mutually-trusted third parties, so for anonymous calls the critical information is simply not sent to the network. There is no reliable way to screen anonymized calls other than simply rejecting all of them.

10.6 New Approaches for Managing Internet Interactions

Though Internet telephony brings about new feature interactions, it also presents new possibilities for managing or resolving these interactions. The flexibility of the signalling protocols, and the underlying infrastructure of the Internet, can be exploited to resolve or prevent interactions in a manner which maintains and extends the powerful new characteristics of the Internet telephony architecture.

10.6.1 Explicitness

Many of the interactions which we have categorized as "cooperative" can be prevented or made less likely by making explicit the actions being taken, and their desired effects. Because the Internet telephony protocols are extensible, it is possible to add parameters which tell downstream servers what actual actions are desired; such parameters are currently being standardized [65]. If a call is intended to only reach a human, for instance, it is possible to specify that the call should not be forwarded to a station which has registered with a "voicemail" attribute; intelligent services which would otherwise forward a call to voicemail should know to return a "not currently available" status code instead. Similarly, a call wishing to camp on to the actual user to be contacted could specify "do-not-forward" so as to get back a "busy" response rather than have the call be forwarded against their wishes. Alternately, new and generalized protocols can be defined — "camp-on" can be generalized as "presence," and a caller could request notification when a caller's state switches from "busy" to "available." The difficulty with these solutions is that it can complicate the creation of services significantly; service creators need not only to determine what it is they wish to do, but to determine whether those actions are compatible with the preferences the caller specified with the call. Also, this explicitness requires that the receiver know about the attributes the caller desires; a call may specify "want to reach only the family goldfish," but the recipient is unlikely to be able to do anything useful with this if "goldfish" is not a recognized category.

10.6.2 Universal Authentication

Many of the problems introduced by polymorphic identities and identity forging can be resolved by insisting on strong authentication of requests. Whereas a generic address can easily be used once and thrown away, and indeed a user can claim to be someone else, the barrier toward obtaining certificates giving actual signed identity information is much higher, and presumably widely-trusted certification authorities can be relied upon to be sufficiently consistent in their identification of users that call screening services can use this information to block callers. Unfortunately, all of this infrastructure fails if users accept non-authenticated calls; and authentication is far from being sufficiently widespread enough for it to be practical to accept only authenticated ones. However, we hope that the growth of Internet telephony will help be a driving force for widespread authentication to finally become widely deployed on the Internet.

10.6.3 Network-level Administrative Restriction

Administrative restrictions in the Internet cannot generally be reliably applied at the application level. If users have end-to-end connectivity available, it is not generally possible to prevent them from taking advantage of this connectivity by imposing restrictions solely at the application layer. Therefore, network-layer administrative restrictions such as firewalls must be used to limit end-to-end connectivity in order to impose administrative controls; these restrictions also have the advantage that they automatically apply to *all* Internet services, not just a limited subset of them. Network-level and application-level restrictions can also be used in concert; for instance, an Internet telephony signalling server, if it decided to allow a call, could instruct a firewall to open up the appropriate ports to allow the media associated with the call to flow.

10.6.4 Verification Testing

Finally, the most direct way of ensuring correct operation of features is to test them directly. It is for third parties to establish services which automatically, at your request, place calls to you with various parameters or conditions enabled, to allow you to confirm explicitly that your features work the way you desire. As such providers gain more experience into the sorts of conditions that are likely to cause problems with services, they can expand their suites of testing tools to cover

more esoteric interaction conditions. Thus, it should be possible to verify features and resolve their interactions in the real environment in which they are deployed, rather than attempting to analyze and categorize all possible consequences of a feature beforehand.

10.7 Conclusion

Feature interactions in Internet telephony are a serious issue, which feature developers will need to consider as they develop services for this new environment. There is a temptation, as Internet telephony "re-invents" the telephone network, to discard the lessons learned from the experience of traditional networks; however, it is clear that feature creation in the Internet must learn from prior experience of creation of telephony services. If these lessons are learned, however, problems of feature interaction will be manageable and can be dealt with efficiently.

The architecture of the Internet makes some of the feature interaction management techniques developed for traditional circuit-switched networks impractical. The distributed nature of feature creation, in particular, means that it will not generally be possible to describe all features globally before designing and implementing them. The Internet also, however, makes new techniques for dealing with interactions possible. These new techniques, and new applications of existing techniques in the new environment, will be a fruitful area for future research.

Chapter 11

Conclusion

The Internet is different in a number of fundamental ways from earlier telecommunications networks. This difference makes it possible to introduce a large number of new services, in the broad sense: it allows the creation of a large number of new methods of communication, and new ways that communication can be customized.

This thesis has presented a number of these new services, for a specific application domain: Internet telephony. Because it operates over the Internet, Internet telephony allows a wide variety services beyond those of traditional telephone networks. This concluding chapter will review this work, demonstrate how it arises as a consequence of the Internet environment, and discuss how the work can be a foundation for further service development, both in Internet telephony and in other application domains.

11.1 Internet Services: Distributed Intelligence

Compared to earlier telecommunications networks, the Internet is different in two primary ways. First of all, communications are rich and flexible: Internet systems can communicate directly with most other systems on the network, and can do so through a wide variety of different mechanisms and media. This allows new types of services, much more flexible than previously possible. As one consequence of this, on the Internet, intelligence is distributed. Rather than having intelligence “in” the network, in one place, Internet services can be implemented in a variety of

locations, through a wide variety of mechanisms. The work presented in this thesis is possible because of this distribution and flexibility.

Chapters 4 and 5 presented new mechanisms for the creation of user-location services in Internet telephony. Chapter 4 introduced the SIP Common Gateway Interface (SIP CGI), an interface for language-independent low-level control of SIP server. Based on the widely-deployed HTTP CGI interface, SIP CGI allows administrators and trusted users to write programs which understand and process SIP requests and responses. Both SIP CGI and its model are examples of the distributed intelligence that distinguishes Internet services. Services can be under the control of independent providers, who can run their own network services; SIP CGI allows them to implement intelligence locally.

The Call Processing Language, presented in Chapter 5, is an alternate approach to the creation of user-location services. The CPL is designed to be executable by naïve or untrusted users, who cannot either be expected or trusted to write services in general-purpose programming languages. The Call Processing Language is a service environment that becomes possible due to the Internet's distributed intelligence and rich communication. In addition to those aspects of distributed intelligence that make SIP CGI possible, CPL also relies on several other factors. First of all, intelligence at the users' side makes script creation and editing environments possible, so users who would be uncomfortable with writing scripts in a textual form can still take advantage of the full power of the language. Secondly, the Internet's rich, flexible communications makes it possible for users to then transmit these scripts to servers in the network. Single-purpose communications networks did not expose mechanisms to allow users to communicate these sorts of configuration information.

Chapters 6 and 7 described the architecture of the server implementing these user-location services. Chapter 6 described an implementation framework to execute policies, by providing a generic API which is sufficiently flexible to support the needs of multiple user-location service execution environments. Chapter 7 illustrated how this server can be made efficient and robust. Both these sections provide an example of the distributed intelligence that becomes possible in the Internet environment.

Full-mesh distributed conferencing was described in Chapter 8. This presented a mech-

anism by which Internet telephony endpoints can reliably establish a tightly-coupled multi-party conference, on the conference call model, without needing a centralized server to mix or coordinate the call. This is very much an example of the merits of distributed intelligence. Each endpoint of the call maintains its own knowledge of the state of the call, and independently establishes communications channels with the others. In networks with centralized intelligence, conference calls by contrast inherently require centralized control.

Chapter 9 introduced a set of mechanisms to allow SIP networks to interwork with UMTS networks, in ways more efficient than simply routing calls via an intermediate PSTN network. This provided ways of connecting and unifying the two networks' rather disparate user-location and terminal-location procedures. The distributed intelligence and rich communications of the Internet makes it possible to connect disparate networks in sophisticated and complex ways. The work of this chapter presented a number of these, and also provided an analysis of how various forms of rich communication can have relative advantages and disadvantages in their efficiency.

Finally, Chapter 10 illustrated a disadvantage of the distributed intelligence and communication: additional complexity. Feature interaction is a problem that has been much studied in traditional telephone networks. When services are created and designed, they can interact in unexpected ways with other services in the same network. In networks where services were implemented on a reasonably centralized basis, these interactions could be controlled and managed by their implementors. Because of the Internet's distributed intelligence, however, the complexity of interactions can increase greatly, and existing techniques for managing interactions can fail — the creator of one service will often be completely ignorant of services being deployed by others. This introduces new levels of complexity to feature interaction problem. The chapter introduced mechanisms for understanding these issues.

11.2 Extensions, In and Beyond Telephony

Many of the issues addressed in this thesis have applications beyond just those described, and beyond just voice and multimedia telephony. The techniques they raise can be applied to other parts of the telephony network, other communications media, and other aspects of computer networks and architecture.

The user location services work of Chapters 4 and 5 can be extended to many other application domains. (SIP CGI, of course, is already based upon work from another application domain, HTTP CGI.) This work, first of all, can be applied to other communication mechanisms — instant messaging, presence, machine communication, e-mail, and so forth — and to other aspects of the Internet telephony network. (The Language for End System Services (LESS) [16] is already building upon the Call Processing Language to control the behavior of Internet telephony end systems.) Moreover, the Call Processing Language establishes a mechanism of representing policies and preferences as decision trees in XML. These trees are expressive and flexible, and easily parsed, created, and edited, but are deliberately restricted in their power so that they can be safely executed by service providers without their needing to worry about incorrect or malicious policies damaging the service environment.

The implementation models described in Chapters 6 and 7 have applicability well beyond the particular server described in those chapters. Chapter 6's policy framework is a model by which extension mechanisms to message-based service policies can be implemented on a common API. Even more generally applicable are the reactive systems described in Chapter 7. Reactive systems provide a flexible and efficient mechanism by which multiple independent transactions can be carried out, in a circumstance when each transaction might persist for a long period of time. As shown in the chapter, this approach can be significantly more efficient than one based naïvely on threads, which can be quite wasteful of system resources.

Reliable decentralized communications, of the sort described in Chapter 8 is useful in many application domains beyond voice or multimedia communications. Any number of applications require a group of entities to communicate with each other. For human use, the text-based chat room is long-established. For network servers — web servers, databases, service location servers, and the like — replication and mutual communication is of course a long-established practice for reliability and enhanced performance. Combined with a neighbor-discovery mechanism, the protocols described in the chapter could be generalized to allow new server instances to quickly enter and leave a distributed network of servers. More speculatively, networks of sensor devices could use the full mesh mechanism to communicate their results to each other, or systems for distributed computation could allocate and redistribute their tasks as nodes in the network

become available and drop out.

In both the Internet and other networks, there are any number of legacy protocols and architectures which need to be supported while new architectures and solutions are developed and deployed. Chapter 9 offers an example of how old and new networks can be interconnected and deployed, and also illustrates how various approaches to these interconnections can be analyzed and compared. The rapid development of computer and network technology will inevitably produce new "legacy" systems; techniques similar to those employed in this chapter will frequently be applicable in the future.

Finally, the complexity of current and future networks, and the service and features deployed in them, will inevitably cause unexpected, and often undesirable, interactions and emergent behavior. Chapter 10 shows how these interactions can be understood, analyzed, and controlled, in the specific context of Internet telephony. Such interactions will arise in many domains beyond those of Internet telephony, of course, and the techniques from this chapter will provide a framework by which they too can be understood and controlled.

11.3 Conclusion

Because of the nature of the Internet, Internet telephony makes possible many new services beyond those possible in traditional telephone networks. These services include both those that extend and generalize existing services, those that create new approaches and architectures for implementing them, and those that are entirely new and made possible because of the Internet environment. These services generally arise because of the unique characteristics of the Internet, particularly its distributed intelligence and rich, flexible communications. The work of this thesis both establishes a number of new services and services mechanisms, and provides a foundation upon which further work can be done in a number of application domains.

Bibliography

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: session initiation protocol," RFC 3261, Internet Engineering Task Force, June 2002.

[2] J. Lennox, H. Schulzrinne, and J. Rosenberg, "Common gateway interface for SIP," RFC 3050, Internet Engineering Task Force, Jan. 2001.

[3] J. Rosenberg, J. Lennox, and H. Schulzrinne, "Programming Internet telephony services," *IEEE Network*, Vol. 13, pp. 42–49, May/June 1999.

[4] J. Lennox, X. Wu, and H. Schulzrinne, "CPL: a language for user control of Internet telephony services," internet draft, Internet Engineering Task Force, Aug. 2003. Work in progress.

[5] J. Lennox and H. Schulzrinne, "Call processing language framework and requirements," RFC 2824, Internet Engineering Task Force, May 2000.

[6] J. Lennox and H. Schulzrinne, "A protocol for reliable decentralized conferencing," in *ACM NOSSDAV 2003*, June 2003.

[7] J. Lennox, K. Murakami, M. Karaul, and T. F. L. Porta, "Interworking Internet telephony and wireless telecommunications networks," *ACM Computer Communication Review*, Vol. 31, pp. 25–36, Oct. 2001.

[8] J. Lennox and H. Schulzrinne, "Feature interaction in Internet telephony," in *Feature Interaction in Telecommunications and Software Systems VI*, (Glasgow, United Kingdom), May 2000.

[9] H. Schulzrinne, "Re-engineering the telephone system," in *IEEE Singapore International Conference on Networks (SICON)*, (Singapore), Apr. 1997.

[10] J. Lennox, H. Schulzrinne, and T. F. L. Porta, "Implementing intelligent network services with the session initiation protocol," Technical Report CUCS-002-99, Columbia University, New York, New York, Jan. 1999.

[11] X. Wu and H. Schulzrinne, "Where should services reside in Internet telephony systems?," in *IP Telecom Services Workshop*, (Atlanta, Georgia), Sept. 2000.

[12] International Telecommunication Union, "Principles of intelligent network architecture," Recommendation Q.1201, International Telecommunication Union, Geneva, Switzerland, Oct. 1992.

[13] International Telecommunication Union, "General recommendations on telephone switching and signaling – intelligent network: Introduction to intelligent network capability set 1," Recommendation Q.1211, International Telecommunication Union, Geneva, Switzerland, Mar. 1993.

[14] International Telecommunication Union, "Packet based multimedia communication systems," Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, Nov. 2000.

[15] J. Rosenberg, "Indicating user agent capabilities in the session initiation protocol (SIP)," Internet Draft draft-ietf-sip-callee-caps-03, Internet Engineering Task Force, Jan. 2004. Work in progress.

[16] X. Wu and H. Schulzrinne, "Programmable end system services using SIP," in *Conference Record of the International Conference on Communications (ICC)*, May 2003.

[17] S. Petrack and L. Conroy, "The PINT service protocol: Extensions to SIP and SDP for IP access to telephone call services," RFC 2848, Internet Engineering Task Force, June 2000.

[18] H. Schulzrinne and J. Rosenberg, "Signaling for Internet telephony," in *International Conference on Network Protocols (ICNP)*, (Austin, Texas), Oct. 1998.

[19] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2616, Internet Engineering Task Force, June 1999.

[20] N. Freed and N. Borenstein, "Multipurpose Internet mail extensions (MIME) part one: Format of Internet message bodies," RFC 2045, Internet Engineering Task Force, Nov. 1996.

[21] M. Handley and V. Jacobson, "SDP: session description protocol," RFC 2327, Internet Engineering Task Force, Apr. 1998.

[22] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifiers (URI): generic syntax," RFC 2396, Internet Engineering Task Force, Aug. 1998.

[23] P. Hoffman, L. Masinter, and J. Zawinski, "The mailto URL scheme," RFC 2368, Internet Engineering Task Force, July 1998.

[24] A. Vaha-Sipila, "URLs for telephone calls," RFC 2806, Internet Engineering Task Force, Apr. 2000.

[25] P. Zolzettich and A. R. Ephrath, "Customized service creation: A new order for telecommunications services," *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, pp. 1014–1019, 1992.

[26] A. R. Ephrath, "Human computer interaction in the advanced intelligent network," in *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1915–1917, Oct. 1995.

[27] International Telecommunication Union, "Intelligent network interfaces," Recommendation Q.1218, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, 1995.

[28] R. Willner and L. Lee, "IN service creation elements: variations on the meaning of a SIB," in *Intelligent Network Workshop*, May 1997.

[29] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (SOAP) 1.1," tech. rep., World Wide Web Consortium, W3C, May 2000.

[30] D. Robinson and K. Coar, "The common gateway interface (CGI) version 1.1," Internet Draft draft-coar-cgi-v11-04.txt,.ps,, Internet Engineering Task Force, Oct. 2003. Work in progress.

[31] Sun Microsystems, *Server-Side JavaScript 1.4: iPlanet Web Server, Enterprise Edition Server-Side JavaScript Guide*. Santa Clara, CA: Sun Microsystems, Aug. 2000.

[32] J. Davidson and D. Coward, "Java servlet specification v2.2," Dec. 1999.

[33] S. S. Bakken, A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lerdorf, A. Zmievski, and J. Ahto, *PHP Manual*. The PHP Documentation Group, Aug. 2003.

[34] Microsoft, "Active server pages," Aug. 2003. http://msdn.microsoft.com/asp/.

[35] M. R. Brown, *FastCGI Specification*, Apr. 1996. http://www.fastcgi.com/devkit/doc/fcgi-spec.html.

[36] S. R. van den Berg and P. Guenther, "Welcome to procmail.org," June 2002.

[37] J. Myers and M. P. Rose, "Post office protocol - version 3," RFC 1939, Internet Engineering Task Force, May 1996.

[38] M. R. Crispin, "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1," RFC 3501, Internet Engineering Task Force, Mar. 2003.

[39] T. Showalter, "Sieve: A mail filtering language," RFC 3028, Internet Engineering Task Force, Jan. 2001.

[40] D. Wetherall, U. Legedza, and J. Guttag, "Introducing new Internet services: Why and how," *IEEE Network*, Vol. 12, pp. 12–19, May 1998.

[41] Y. Yemini and S. da Silva, "Towards programmable networks," in *7th IFIP/IEEE International Workshop on Distributed Systems Operations and Management (DSOM)*, (L'Aquila, Italy), Oct. 1996.

[42] Sun Microsystems, "JAIN APIs for integrated networks." Available at http://java.sun.com/products/jain/.

[43] R. Jain, F. M. Anjum, P. Missier, and S. Shastry, "Java call control, coordination, and transactions," *IEEE Communications Magazine*, Vol. 38, Jan. 2000.

[44] T. P. Group, "Parlay API business benefits white paper," Jan. 2000. Available at http://www.parlay.org/20event/White_paperv2.01.doc.

[45] J. C. Process, "SIP servlet API," Java Specification Requests JSR 116, Java Community Process, May 2002.

[46] S. McGlashan, D. Burnett, J. Carter, S. Tryphonas, J. Ferrans, A. Hunt, B. Lucas, and B. Porter, "Voice extensible markup language (voicexml) version 2.0," tech. rep., World Wide Web Consortium (W3C), Feb. 2003. http://www.w3.org/TR/voicexml20/.

[47] R. Auburn, M. Cafarella, D. Jackson, J. Peck, P. Sharma, S. Shanmughan, C. Stohs, and Y. Zhang, "Voice browser call control: CCXML version 1.0," w3c working draft, World Wide Web Consortium (W3C), June 2003. http://www.w3.org/TR/ccxml/.

[48] J.-L. Bakker and R. Jain, "Next generation service creation using XML scripting languages," in *Conference Record of the International Conference on Communications (ICC)*, (New York, NY, USA), pp. 2001–2007, Apr. 2002.

[49] P. Resnick, ed., "Internet message format," RFC 2822, Internet Engineering Task Force, Apr. 2001.

[50] D. Kristol and L. Montulli, "HTTP state management mechanism," RFC 2965, Internet Engineering Task Force, Oct. 2000.

[51] L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl*. Sebastopol, California: O'Reilly, 2nd ed., 1996.

[52] N. Freed and N. Borenstein, "Multipurpose Internet mail extensions (MIME) part two: Media types," RFC 2046, Internet Engineering Task Force, Nov. 1996.

[53] M. S. Johns, "Identification protocol," RFC 1413, Internet Engineering Task Force, Feb. 1993.

[54] L. H. Bui, K. Singh, and H. Schulzrinne, "SIP application level gateway," Nov. 2002. http://www.cs.columbia.edu/ kns10/projects/fall2002/sipalg/.

[55] J. Lee, K. Singh, and H. Schulzrinne, "SIP 911 implementation," May 2002. http://www.cs.columbia.edu/ kns10/projects/spring2002/911/.

[56] L. Gong, M. E. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2," in *USENIX Symposium on Internet Technologies and Systems*, (Monterey, CA), Dec 1997.

[57] J. Y. Levy, L. Demailly, J. Ousterhout, and B. B. Welch, "The safe-tcl security model," in *Proc. of Usenix Annual Technical Conference*, (New Orleans, Louisiana), June 1998.

[58] J. Ousterhout, *Tcl and the Tk Toolkit*. Reading, Massachusetts: Addison-Wesley, 1994.

[59] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0 (second edition)," W3C Recommendation REC-xml-20001006, World Wide Web Consortium (W3C), Oct. 2000. Available at http://www.w3.org/XML/.

[60] D. Raggett, A. Le Hors, and I. Jacobs, "HTML 4.0 specification," W3C Recommendation REC-html40-19980424, World Wide Web Consortium (W3C), Apr. 1998. Available at http://www.w3.org/TR/REC-html40/.

[61] ISO (International Organization for Standardization), "Information processing — text and office systems — standard generalized markup language (SGML)," ISO Standard ISO 8879:1986(E), International Organization for Standardization, Geneva, Switzerland, Oct. 1986.

[62] D. C. Fallside, "XML schema part 0: Primer," W3C Candidate Recommendation CR-xmlschema-0-20001024, World Wide Web Consortium (W3C), Oct. 2000. Available at http://www.w3.org/TR/xmlschema-0/.

[63] H. Alvestrand, "Tags for the identification of languages," RFC 3066, Internet Engineering Task Force, Jan. 2001.

[64] F. Dawson and D. Stenerson, "Internet calendaring and scheduling core object specification (icalendar)," RFC 2445, Internet Engineering Task Force, Nov. 1998.

[65] J. Rosenberg, H. Schulzrinne, and P. Kyzivat, "Caller preferences for the session initiation protocol (SIP)," Internet Draft draft-ietf-sip-callerprefs-10, Internet Engineering Task Force, Oct. 2003. Work in progress.

[66] T. Bray, D. Hollander, and A. Layman, "Namespaces in XML," W3C Recommendation REC-xml-names-19900114, World Wide Web Consortium (W3C), Jan. 1999. Available at http://www.w3.org/TR/REC-xml-names/.

[67] E. W. Dijkstra, "Go to statement considered harmful," *Communications ACM*, Vol. 11, pp. 147–148, Mar. 1968.

[68] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (RTSP)," RFC 2326, Internet Engineering Task Force, Apr. 1998.

[69] "The apache project home page." http://www.apache.org/.

[70] The Apache Software Foundation, "Apache HTTP server version 1.3: Apache API notes." http://httpd.apache.org/docs/misc/API.html.

[71] V. K. Gurbani and V. Rastogi, "Accessing IN services from SIP networks," internet draft, Internet Engineering Task Force, Sept. 2001. Work in progress.

[72] H. Schulzrinne and S. Petrack, "RTP payload for DTMF digits, telephony tones and telephony signals," RFC 2833, Internet Engineering Task Force, May 2000.

[73] The Open Group, "Standard for information technology: Portable operating system interface (POSIX)," IEEE Std 1003.1-2001, IEEE, New York, NY, Dec. 2001.

[74] J. Trivedi, "Overcoming the limit of thread creation," ACCESS1.SUN.COM technical article, Sun Microsystems, 2002. http://access1.sun.com/techarticles/limit.html.

[75] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable Internet services," in *Symposium on Operating Systems Principles (SOSP)*, (Chateau Lake Louise, Canada), ACM, Oct. 2001.

[76] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag, 1991.

[77] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: session initiation protocol," RFC 2543, Internet Engineering Task Force, Mar. 1999.

[78] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle, "SIPstone - benchmarking SIP server performance," Technical Report CUCS-005-02, Department of Computer Science, Columbia University, New York, New York, Mar. 2002.

[79] C. Elliott, "A 'sticky' conference control protocol," *Internetworking: Research and Experience*, Vol. 5, pp. 97–119, 1994.

[80] W. Zhao and H. Schulzrinne, "mSLP - mesh-enhanced service location protocol," Technical Report CUCS-013-00, Columbia University, New York, May 2000.

[81] R. Boivie *et al.*, "Explicit multicast (xcast) basic specification," Internet Draft draft-ooms-xcast-basic-spec-05, Internet Engineering Task Force, Sept. 2003. Work in progress.

[82] H. Schulzrinne and J. Rosenberg, "SIP call control services," internet draft, Internet Engineering Task Force, June 1999. Work in progress.

[83] J. Rosenberg and H. Schulzrinne, "Models for multi party conferencing in SIP," internet draft, Internet Engineering Task Force, July 2002. Work in progress.

[84] S. Bhattacharyya and I. W. Ed., "An overview of source-specific multicast (SSM)," RFC 3569, Internet Engineering Task Force, July 2003.

[85] S. Bradner, A. Mankin, and J. I. Schiller, "A framework for purpose-built keys (PBK)," internet draft, Internet Engineering Task Force, June 2003. Work in progress.

[86] J. Rosenberg, "The session initiation protocol (SIP) UPDATE method," RFC 3311, Internet Engineering Task Force, Oct. 2002.

[87] O. Levin and R. K. Even, "High level requirements for tightly coupled SIP conferencing," internet draft, Internet Engineering Task Force, Mar. 2003. Work in progress.

[88] European Telecommunications Standards Institute, "Digital cellular telecommunications system, network architecture," GSM 03.02 version 7.1.0, European Telecommunications Standards Institute, Sophia Antipolis, France, Feb. 2000.

[89] European Telecommunications Standards Institute, "Universal mobile telecommunications system (UMTS), general UMTS architecture," 3G TS 23.101 version 3.0.1, European Telecommunications Standards Institute, Sophia Antipolis, France, Jan. 2000.

[90] B. Jabbari, "Special issue on wideband CDMA," *IEEE Communications Magazine*, Vol. 36, Sept. 1998.

[91] H. Rao, Y.-B. Lin, and S.-L. Cho, "igsm: VoIP service for mobile networks," *IEEE Communications Magazine*, Vol. 38, Apr. 2000.

[92] W. T. Liao, "Mobile Internet telephony: Mobile extensions to H.323," in *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, (New York), Mar. 1999.

[93] W. Liao, "Mobile Internet telephony protocol: An application layer protocol for mobile Internet telephony services," in *Conference Record of the International Conference on Communications (ICC)*, (Vancouver, British Columbia), June 1999.

[94] W. Liao and J.-C. Liu, "VoIP mobility in IP/Cellular network internetworking," *IEEE Communications Magazine*, Vol. 38, Apr. 2000.

[95] O. Haase, K. Murakami, and T. F. L. Porta, "Unified mobility manager: Enabling efficient SIP/UMTS mobile network control," *IEEE Wireless Communications*, Vol. 10, pp. 66–75, Aug. 2003.

[96] Telecommunications Industry Association and Electronics Industry Association, "Cellular radiotelecommunications intersystem operations," TIA/EIA ANSI-41-D, Telecommunications Industry Association, Arlington, Virginia, Dec. 1997.

[97] Y. S. Cho, Y. Lin, and H. Rao, "Reducing the network cost of call delivery to GSM roamers," *IEEE Network*, Vol. 11, pp. 19–25, Sept. 1997.

[98] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: a transport protocol for real-time applications," RFC 3550, Internet Engineering Task Force, July 2003.

[99] H. Schulzrinne and S. Casner, "RTP profile for audio and video conferences with minimal control," RFC 3551, Internet Engineering Task Force, July 2003.

[100] European Telecommunications Standards Institute, "Digital cellular telecommunications system, full rate speech," GSM 06.10 version 5.0.1, European Telecommunications Standards Institute, Sophia Antipolis, France, May 1997.

[101] T. F. L. Porta, K. Murakami, and R. Ramjee, "RIMA: router for integrated mobile access," in *11th IEEE International Symposium on Personal, Indoor and Mobile Radio Communication (PIMRC)*, (London, United Kingdom), Sept. 2000.

[102] International Telecommunication Union, "The international public telecommunication numbering plan," Recommendation E.164, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, May 1997.

[103] E. Wedlund and H. Schulzrinne, "Mobility support using SIP," in *2nd ACM/IEEE International Conference on Wireless and Mobile Multimedia (WoWMoM)*, (Seattle, Washington), Aug. 1999.

[104] C. E. Perkins, "IP mobility support for IPv4," RFC 3344, Internet Engineering Task Force, Aug. 2002.

[105] R. J. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, and M. Kalla, "Stream control transmission protocol," RFC 2960, Internet Engineering Task Force, Oct. 2000.

[106] P. Faltstrom, "E.164 number and DNS," RFC 2916, Internet Engineering Task Force, Sept. 2000.

[107] J. Rosenberg and H. Schulzrinne, "A framework for telephony routing over IP," RFC 2871, Internet Engineering Task Force, June 2000.

[108] J. Rosenberg, H. F. Salama, and M. Squire, "Telephony routing over IP (TRIP)," RFC 3219, Internet Engineering Task Force, Jan. 2002.

[109] H. Schulzrinne and J. Rosenberg, "Internet telephony: Architecture and protocols – an IETF perspective," *Computer Networks and ISDN Systems*, Vol. 31, pp. 237–255, Feb. 1999.

[110] H. Schulzrinne and J. Rosenberg, "The session initiation protocol: Providing advanced telephony services across the Internet," *Bell Labs Technical Journal*, Vol. 3, pp. 144–160, October-December 1998.

[111] S. A. Kent and R. Atkinson, "Security architecture for the Internet protocol," RFC 2401, Internet Engineering Task Force, Nov. 1998.

[112] K. Turner, "Representing new voice services and their features," in *Feature Interactions in Telecommunication Networks*, (Ottawa, Canada), IOS Press, June 2003.

[113] M. Nakamura, P. Leelaprute, K. ichi Matsumoto, and T. Kikuno, "Detecting script-to-script interactions in call processing language," in *Feature Interactions in Telecommunication Networks*, (Ottawa, Canada), IOS Press, June 2003.

[114] E. J. Cameron, N. Griffeth, Y. Lin, M. E. Nilson, W. K. Schure, and H. Velthuijsen, "A feature interaction benchmark for IN and beyond," in *Feature Interactions in Telecommunications Systems*, (Amsterdam, Netherlands), pp. 1–23, 1994.