faqs.org



RFC 3261 - SIP: Session Initiation Protocol

Internet RFC Index
Usenet FAQ Index
Other FAQs
Documents
Tools

Search
Search FAQs
Search RFCs

IFC Home
Cities
Countries
Hospitals
Web Hosting Ratings

Search the RFC Archives
Search
Or Display the document by number
Display RFC By Number

[RFC Index | Usenet FAQs | Web FAQs | Documents | Cities | SEC Filings | Patents | Neighborhoods]

Network Working Group
Request for Comments: 3261
Obsoletes: 2543
Category: Standards Track

J. Rosenberg
dynamicsoft
H. Schulzrinne
Columbia U.
G. Camarillo
Ericsson
A. Johnston
WorldCom
J. Peterson
Neustar
R. Sparks
dynamicsoft
M. Handley
ICIR
E. Schooler
AT&T

June 2002

SIP: Session Initiation Protocol

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2002). All Rights Reserved.

Abstract

This document describes Session Initiation Protocol (SIP), an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia

APPLE 1026

distribution, and multimedia conferences.

SIP invitations used to create sessions carry session descriptions that allow participants to agree on a set of compatible media types. SIP makes use of elements called proxy servers to help route requests to the user's current location, authenticate and authorize users for services, implement provider call-routing policies, and provide features to users. SIP also provides a registration function that allows users to upload their current locations for use by proxy servers. SIP runs on top of several different transport protocols.

Table of Contents

1 Introduction 8
2 Overview of SIP Functionality 9
3 Terminology 10
4 Overview of Operation 10
5 Structure of the Protocol 18
6 Definitions 20
7 SIP Messages 26
7.1 Requests 27
7.2 Responses 28
7.3 Header Fields 29
7.3.1 Header Field Format 30
7.3.2 Header Field Classification 32
7.3.3 Compact Form 32
7.4 Bodies 33
7.4.1 Message Body Type 33
7.4.2 Message Body Length 33
7.5 Framing SIP Messages 34
8 General User Agent Behavior 34
8.1 UAC Behavior 35
8.1.1 Generating the Request 35
8.1.1.1 Request-URI 35
8.1.1.2 To 36
8.1.1.3 From 37
8.1.1.4 Call-ID 37
8.1.1.5 CSeq 38
8.1.1.6 Max-Forwards 38
8.1.1.7 Via 39
8.1.1.8 Contact 40
8.1.1.9 Supported and Require 40
8.1.1.10 Additional Message Components 41
8.1.2 Sending the Request 41
8.1.3 Processing Responses 42
8.1.3.1 Transaction Layer Errors 42
8.1.3.2 Unrecognized Responses 42
8.1.3.3 Vias 43
8.1.3.4 Processing 3xx Responses 43
8.1.3.5 Processing 4xx Responses 45
8.2 UAS Behavior 46
8.2.1 Method Inspection 46
8.2.2 Header Inspection 46
8.2.2.1 To and Request-URI 46
8.2.2.2 Merged Requests 47
8.2.2.3 Require 47
8.2.3 Content Processing 48

8.2.4 Applying Extensions 49
8.2.5 Processing the Request 49

8.2.6 Generating the Response 49
8.2.6.1 Sending a Provisional Response 49
8.2.6.2 Headers and Tags 50
8.2.7 Stateless UAS Behavior 50
8.3 Redirect Servers 51
9 Canceling a Request 53
9.1 Client Behavior 53
9.2 Server Behavior 55
10 Registrations 56
10.1 Overview 56
10.2 Constructing the REGISTER Request 57
10.2.1 Adding Bindings 59
10.2.1.1 Setting the Expiration Interval of Contact Addresses 60
10.2.1.2 Preferences among Contact Addresses 61
10.2.2 Removing Bindings 61
10.2.3 Fetching Bindings 61
10.2.4 Refreshing Bindings 61
10.2.5 Setting the Internal Clock 62
10.2.6 Discovering a Registrar 62
10.2.7 Transmitting a Request 62
10.2.8 Error Responses 63
10.3 Processing REGISTER Requests 63
11 Querying for Capabilities 66
11.1 Construction of OPTIONS Request 67
11.2 Processing of OPTIONS Request 68
12 Dialogs 69
12.1 Creation of a Dialog 70
12.1.1 UAS behavior 70
12.1.2 UAC Behavior 71
12.2 Requests within a Dialog 72
12.2.1 UAC Behavior 73
12.2.1.1 Generating the Request 73
12.2.1.2 Processing the Responses 75
12.2.2 UAS Behavior 76
12.3 Termination of a Dialog 77
13 Initiating a Session 77
13.1 Overview 77
13.2 UAC Processing 78
13.2.1 Creating the Initial INVITE 78
13.2.2 Processing INVITE Responses 81
13.2.2.1 1xx Responses 81
13.2.2.2 3xx Responses 81
13.2.2.3 4xx, 5xx and 6xx Responses 81
13.2.2.4 2xx Responses 82
13.3 UAS Processing 83
13.3.1 Processing of the INVITE 83
13.3.1.1 Progress 84
13.3.1.2 The INVITE is Redirected 84

13.3.1.3 The INVITE is Rejected 85
13.3.1.4 The INVITE is Accepted 85
14 Modifying an Existing Session 86
14.1 UAC Behavior 86
14.2 UAS Behavior 88

15 Terminating a Session 89
15.1 Terminating a Session with a BYE Request 90
15.1.1 UAC Behavior 90
15.1.2 UAS Behavior 91
16 Proxy Behavior 91
16.1 Overview 91
16.2 Stateful Proxy 92
16.3 Request Validation 94
16.4 Route Information Preprocessing 96
16.5 Determining Request Targets 97
16.6 Request Forwarding 99
16.7 Response Processing 107
16.8 Processing Timer C 114
16.9 Handling Transport Errors 115
16.10 CANCEL Processing 115
16.11 Stateless Proxy 116
16.12 Summary of Proxy Route Processing 118
16.12.1 Examples 118
16.12.1.1 Basic SIP Trapezoid 118
16.12.1.2 Traversing a Strict-Routing Proxy 120
16.12.1.3 Rewriting Record-Route Header Field Values 121
17 Transactions 122
17.1 Client Transaction 124
17.1.1 INVITE Client Transaction 125
17.1.1.1 Overview of INVITE Transaction 125
17.1.1.2 Formal Description 125
17.1.1.3 Construction of the ACK Request 129
17.1.2 Non-INVITE Client Transaction 130
17.1.2.1 Overview of the non-INVITE Transaction 130
17.1.2.2 Formal Description 131
17.1.3 Matching Responses to Client Transactions 132
17.1.4 Handling Transport Errors 133
17.2 Server Transaction 134
17.2.1 INVITE Server Transaction 134
17.2.2 Non-INVITE Server Transaction 137
17.2.3 Matching Requests to Server Transactions 138
17.2.4 Handling Transport Errors 141
18 Transport 141
18.1 Clients 142
18.1.1 Sending Requests 142
18.1.2 Receiving Responses 144
18.2 Servers 145
18.2.1 Receiving Requests 145

18.2.2 Sending Responses 146
18.3 Framing 147
18.4 Error Handling 147
19 Common Message Components 147
19.1 SIP and SIPS Uniform Resource Indicators 148
19.1.1 SIP and SIPS URI Components 148
19.1.2 Character Escaping Requirements 152
19.1.3 Example SIP and SIPS URIs 153
19.1.4 URI Comparison 153
19.1.5 Forming Requests from a URI 156
19.1.6 Relating SIP URIs and tel URLs 157
19.2 Option Tags 158
19.3 Tags 159

20 Header Fields 159
20.1 Accept 161
20.2 Accept-Encoding 163
20.3 Accept-Language 164
20.4 Alert-Info 164
20.5 Allow 165
20.6 Authentication-Info 165
20.7 Authorization 165
20.8 Call-ID 166
20.9 Call-Info 166
20.10 Contact 167
20.11 Content-Disposition 168
20.12 Content-Encoding 169
20.13 Content-Language 169
20.14 Content-Length 169
20.15 Content-Type 170
20.16 CSeq 170
20.17 Date 170
20.18 Error-Info 171
20.19 Expires 171
20.20 From 172
20.21 In-Reply-To 172
20.22 Max-Forwards 173
20.23 Min-Expires 173
20.24 MIME-Version 173
20.25 Organization 174
20.26 Priority 174
20.27 Proxy-Authenticate 174
20.28 Proxy-Authorization 175
20.29 Proxy-Require 175
20.30 Record-Route 175
20.31 Reply-To 176
20.32 Require 176
20.33 Retry-After 176
20.34 Route 177

20.35 Server 177
20.36 Subject 177
20.37 Supported 178
20.38 Timestamp 178
20.39 To 178
20.40 Unsupported 179
20.41 User-Agent 179
20.42 Via 179
20.43 Warning 180
20.44 WWW-Authenticate 182
21 Response Codes 182
21.1 Provisional 1xx 182
21.1.1 100 Trying 183
21.1.2 180 Ringing 183
21.1.3 181 Call Is Being Forwarded 183
21.1.4 182 Queued 183
21.1.5 183 Session Progress 183
21.2 Successful 2xx 183
21.2.1 200 OK 183
21.3 Redirection 3xx 184
21.3.1 300 Multiple Choices 184

21.3.2 301 Moved Permanently 184
21.3.3 302 Moved Temporarily 184
21.3.4 305 Use Proxy 185
21.3.5 380 Alternative Service 185
21.4 Request Failure 4xx 185
21.4.1 400 Bad Request 185
21.4.2 401 Unauthorized 185
21.4.3 402 Payment Required 186
21.4.4 403 Forbidden 186
21.4.5 404 Not Found 186
21.4.6 405 Method Not Allowed 186
21.4.7 406 Not Acceptable 186
21.4.8 407 Proxy Authentication Required 186
21.4.9 408 Request Timeout 186
21.4.10 410 Gone 187
21.4.11 413 Request Entity Too Large 187
21.4.12 414 Request-URI Too Long 187
21.4.13 415 Unsupported Media Type 187
21.4.14 416 Unsupported URI Scheme 187
21.4.15 420 Bad Extension 187
21.4.16 421 Extension Required 188
21.4.17 423 Interval Too Brief 188
21.4.18 480 Temporarily Unavailable 188
21.4.19 481 Call/Transaction Does Not Exist 188
21.4.20 482 Loop Detected 188
21.4.21 483 Too Many Hops 189
21.4.22 484 Address Incomplete 189

21.4.23 485 Ambiguous 189
21.4.24 486 Busy Here 189
21.4.25 487 Request Terminated 190
21.4.26 488 Not Acceptable Here 190
21.4.27 491 Request Pending 190
21.4.28 493 Undecipherable 190
21.5 Server Failure 5xx 190
21.5.1 500 Server Internal Error 190
21.5.2 501 Not Implemented 191
21.5.3 502 Bad Gateway 191
21.5.4 503 Service Unavailable 191
21.5.5 504 Server Time-out 191
21.5.6 505 Version Not Supported 192
21.5.7 513 Message Too Large 192
21.6 Global Failures 6xx 192
21.6.1 600 Busy Everywhere 192
21.6.2 603 Decline 192
21.6.3 604 Does Not Exist Anywhere 192
21.6.4 606 Not Acceptable 192
22 Usage of HTTP Authentication 193
22.1 Framework 193
22.2 User-to-User Authentication 195
22.3 Proxy-to-User Authentication 197
22.4 The Digest Authentication Scheme 199
23 S/MIME 201
23.1 S/MIME Certificates 201
23.2 S/MIME Key Exchange 202
23.3 Securing MIME bodies 205
23.4 SIP Header Privacy and Integrity using S/MIME:

Tunneling SIP 207
23.4.1 Integrity and Confidentiality Properties of SIP Headers 207
23.4.1.1 Integrity 207
23.4.1.2 Confidentiality 208
23.4.2 Tunneling Integrity and Authentication 209
23.4.3 Tunneling Encryption 211
24 Examples 213
24.1 Registration 213
24.2 Session Setup 214
25 Augmented BNF for the SIP Protocol 219
25.1 Basic Rules 219
26 Security Considerations: Threat Model and Security Usage Recommendations 232
26.1 Attacks and Threat Models 233
26.1.1 Registration Hijacking 233
26.1.2 Impersonating a Server 234
26.1.3 Tampering with Message Bodies 235
26.1.4 Tearing Down Sessions 235
26.1.5 Denial of Service and Amplification 236
26.2 Security Mechanisms 237
26.2.1 Transport and Network Layer Security 238
26.2.2 SIPS URI Scheme 239
26.2.3 HTTP Authentication 240
26.2.4 S/MIME 240
26.3 Implementing Security Mechanisms 241
26.3.1 Requirements for Implementers of SIP 241
26.3.2 Security Solutions 242
26.3.2.1 Registration 242
26.3.2.2 Interdomain Requests 243
26.3.2.3 Peer-to-Peer Requests 245
26.3.2.4 DoS Protection 246
26.4 Limitations 247
26.4.1 HTTP Digest 247
26.4.2 S/MIME 248
26.4.3 TLS 249
26.4.4 SIPS URIs 249
26.5 Privacy 251
27 IANA Considerations 252
27.1 Option Tags 252
27.2 Warn-Codes 252
27.3 Header Field Names 253
27.4 Method and Response Codes 253
27.5 The "message/sip" MIME type. 254
27.6 New Content-Disposition Parameter Registrations 255
28 Changes From RFC 2543 255
28.1 Major Functional Changes 255
28.2 Minor Functional Changes 260
29 Normative References 261
30 Informative References 262
A Table of Timer Values 265
Acknowledgments 266
Authors' Addresses 267
Full Copyright Statement 269

1 Introduction

There are many applications of the Internet that require the creation and management of a session, where a session is considered an exchange of data between an association of participants. The implementation of these applications is complicated by the practices of participants: users may move between endpoints, they may be addressable by multiple names, and they may communicate in several different media - sometimes simultaneously. Numerous protocols have been authored that carry various forms of real-time multimedia session data such as voice, video, or text messages. The Session Initiation Protocol (SIP) works in concert with these protocols by

enabling Internet endpoints (called user agents) to discover one another and to agree on a characterization of a session they would like to share. For locating prospective session participants, and for other functions, SIP enables the creation of an infrastructure of network hosts (called proxy servers) to which user agents can send registrations, invitations to sessions, and other requests. SIP is an agile, general-purpose tool for creating, modifying, and terminating sessions that works independently of underlying transport protocols and without dependency on the type of session that is being established.

2 Overview of SIP Functionality

SIP is an application-layer control protocol that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls. SIP can also invite participants to already existing sessions, such as multicast conferences. Media can be added to (and removed from) an existing session. SIP transparently supports name mapping and redirection services, which supports personal mobility [27] - users can maintain a single externally visible identifier regardless of their network location.

SIP supports five facets of establishing and terminating multimedia communications:

- User location: determination of the end system to be used for communication;
- User availability: determination of the willingness of the called party to engage in communications;
- User capabilities: determination of the media and media parameters to be used;
- Session setup: "ringing", establishment of session parameters at both called and calling party;
- Session management: including transfer and termination of sessions, modifying session parameters, and invoking services.

SIP is not a vertically integrated communications system. SIP is rather a component that can be used with other IETF protocols to build a complete multimedia architecture. Typically, these architectures will include protocols such as the Real-time Transport

Protocol (RTP) (RFC 1889 [28]) for transporting real-time data and providing QoS feedback, the Real-Time streaming protocol (RTSP) (RFC 2326 [29]) for controlling delivery of streaming media, the Media

Gateway Control Protocol (MEGACO) (RFC 3015 [30]) for controlling gateways to the Public Switched Telephone Network (PSTN), and the Session Description Protocol (SDP) (RFC 2327 [1]) for describing multimedia sessions. Therefore, SIP should be used in conjunction with other protocols in order to provide complete services to the users. However, the basic functionality and operation of SIP does not depend on any of these protocols.

SIP does not provide services. Rather, SIP provides primitives that can be used to implement different services. For example, SIP can locate a user and deliver an opaque object to his current location. If this primitive is used to deliver a session description written in SDP, for instance, the endpoints can agree on the parameters of a session. If the same primitive is used to deliver a photo of the caller as well as the session description, a "caller ID" service can be easily implemented. As this example shows, a single primitive is typically used to provide several different services.

SIP does not offer conference control services such as floor control or voting and does not prescribe how a conference is to be managed. SIP can be used to initiate a session that uses some other conference control protocol. Since SIP messages and the sessions they establish can pass through entirely different networks, SIP cannot, and does not, provide any kind of network resource reservation capabilities.

The nature of the services provided make security particularly important. To that end, SIP provides a suite of security services, which include denial-of-service prevention, authentication (both user to user and proxy to user), integrity protection, and encryption and privacy services.

SIP works with both IPv4 and IPv6.

3 Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [2] and indicate requirement levels for compliant SIP implementations.

4 Overview of Operation

This section introduces the basic operations of SIP using simple examples. This section is tutorial in nature and does not contain any normative statements.

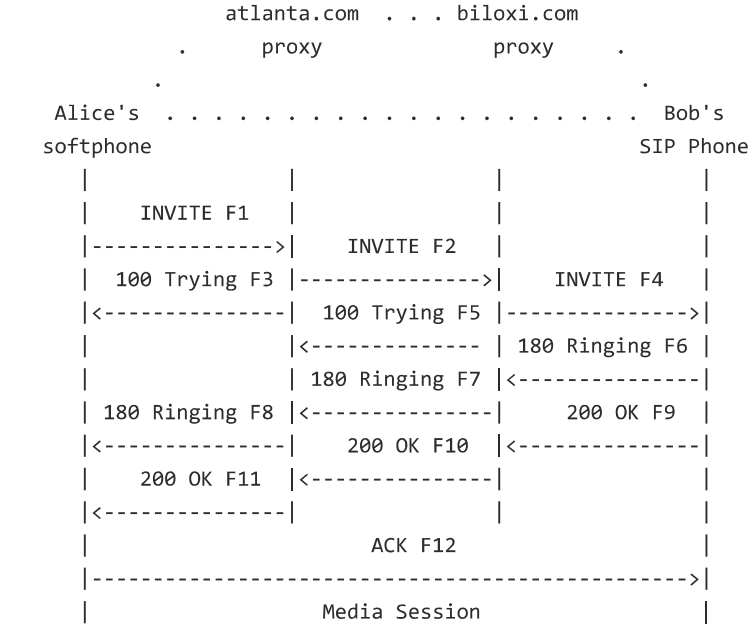
The first example shows the basic functions of SIP: location of an end point, signal of a desire to communicate, negotiation of session parameters to establish the session, and teardown of the session once established.

Figure 1 shows a typical example of a SIP message exchange between

two users, Alice and Bob. (Each message is labeled with the letter "F" and a number for reference by the text.) In this example, Alice uses a SIP application on her PC (referred to as a softphone) to call Bob on his SIP phone over the Internet. Also shown are two SIP proxy servers that act on behalf of Alice and Bob to facilitate the session establishment. This typical arrangement is often referred to as the "SIP trapezoid" as shown by the geometric shape of the dotted lines in Figure 1.

Alice "calls" Bob using his SIP identity, a type of Uniform Resource Identifier (URI) called a SIP URI. SIP URIs are defined in Section 19.1. It has a similar form to an email address, typically containing a username and a host name. In this case, it is sip:bob@biloxi.com, where biloxi.com is the domain of Bob's SIP service provider. Alice has a SIP URI of sip:alice@atlanta.com. Alice might have typed in Bob's URI or perhaps clicked on a hyperlink or an entry in an address book. SIP also provides a secure URI, called a SIPS URI. An example would be sips:bob@biloxi.com. A call made to a SIPS URI guarantees that secure, encrypted transport (namely TLS) is used to carry all SIP messages from the caller to the domain of the callee. From there, the request is sent securely to the callee, but with security mechanisms that depend on the policy of the domain of the callee.

SIP is based on an HTTP-like request/response transaction model. Each transaction consists of a request that invokes a particular method, or function, on the server and at least one response. In this example, the transaction begins with Alice's softphone sending an INVITE request addressed to Bob's SIP URI. INVITE is an example of a SIP method that specifies the action that the requestor (Alice) wants the server (Bob) to take. The INVITE request contains a number of header fields. Header fields are named attributes that provide additional information about a message. The ones present in an INVITE include a unique identifier for the call, the destination address, Alice's address, and information about the type of session that Alice wishes to establish with Bob. The INVITE (message F1 in Figure 1) might look like this:



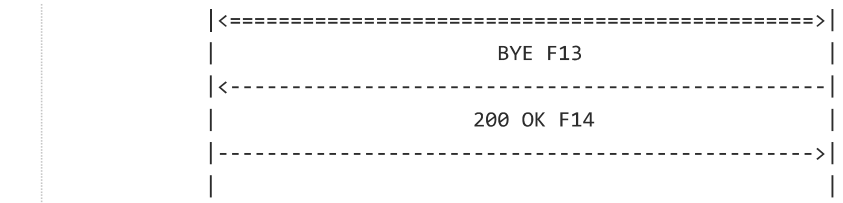


Figure 1: SIP session setup example with SIP trapezoid

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/sdp
Content-Length: 142

(Alice's SDP not shown)
```

The first line of the text-encoded message contains the method name (INVITE). The lines that follow are a list of header fields. This example contains a minimum required set. The header fields are briefly described below:

Via contains the address (pc33.atlanta.com) at which Alice is expecting to receive responses to this request. It also contains a branch parameter that identifies this transaction.

To contains a display name (Bob) and a SIP or SIPS URI (sip:bob@biloxi.com) towards which the request was originally directed. Display names are described in RFC 2822 [3].

From also contains a display name (Alice) and a SIP or SIPS URI (sip:alice@atlanta.com) that indicate the originator of the request. This header field also has a tag parameter containing a random string (1928301774) that was added to the URI by the softphone. It is used for identification purposes.

Call-ID contains a globally unique identifier for this call, generated by the combination of a random string and the softphone's host name or IP address. The combination of the To tag, From tag, and Call-ID completely defines a peer-to-peer SIP relationship between Alice and Bob and is referred to as a dialog.

CSeq or Command Sequence contains an integer and a method name. The CSeq number is incremented for each new request within a dialog and is a traditional sequence number.

Contact contains a SIP or SIPS URI that represents a direct route to contact Alice, usually composed of a username at a fully qualified domain name (FQDN). While an FQDN is preferred, many end systems do not have registered domain names, so IP addresses are permitted. While the Via header field tells other elements where to send the response, the Contact header field tells other elements where to send

future requests.

Max-Forwards serves to limit the number of hops a request can make on the way to its destination. It consists of an integer that is decremented by one at each hop.

Content-Type contains a description of the message body (not shown).

Content-Length contains an octet (byte) count of the message body.

The complete set of SIP header fields is defined in Section 20.

The details of the session, such as the type of media, codec, or sampling rate, are not described using SIP. Rather, the body of a SIP message contains a description of the session, encoded in some other protocol format. One such format is the Session Description Protocol (SDP) (RFC 2327 [1]). This SDP message (not shown in the

example) is carried by the SIP message in a way that is analogous to a document attachment being carried by an email message, or a web page being carried in an HTTP message.

Since the softphone does not know the location of Bob or the SIP server in the biloxi.com domain, the softphone sends the INVITE to the SIP server that serves Alice's domain, atlanta.com. The address of the atlanta.com SIP server could have been configured in Alice's softphone, or it could have been discovered by DHCP, for example.

The atlanta.com SIP server is a type of SIP server known as a proxy server. A proxy server receives SIP requests and forwards them on behalf of the requestor. In this example, the proxy server receives the INVITE request and sends a 100 (Trying) response back to Alice's softphone. The 100 (Trying) response indicates that the INVITE has been received and that the proxy is working on her behalf to route the INVITE to the destination. Responses in SIP use a three-digit code followed by a descriptive phrase. This response contains the same To, From, Call-ID, CSeq and branch parameter in the Via as the INVITE, which allows Alice's softphone to correlate this response to the sent INVITE. The atlanta.com proxy server locates the proxy server at biloxi.com, possibly by performing a particular type of DNS (Domain Name Service) lookup to find the SIP server that serves the biloxi.com domain. This is described in [4]. As a result, it obtains the IP address of the biloxi.com proxy server and forwards, or proxies, the INVITE request there. Before forwarding the request, the atlanta.com proxy server adds an additional Via header field value that contains its own address (the INVITE already contains Alice's address in the first Via). The biloxi.com proxy server receives the INVITE and responds with a 100 (Trying) response back to the atlanta.com proxy server to indicate that it has received the INVITE and is processing the request. The proxy server consults a database, generically called a location service, that contains the current IP address of Bob. (We shall see in the next section how this database can be populated.) The biloxi.com proxy server adds another Via header field value with its own address to the INVITE and proxies it to Bob's SIP phone.

Bob's SIP phone receives the INVITE and alerts Bob to the incoming

call from Alice so that Bob can decide whether to answer the call, that is, Bob's phone rings. Bob's SIP phone indicates this in a 180 (Ringing) response, which is routed back through the two proxies in the reverse direction. Each proxy uses the Via header field to determine where to send the response and removes its own address from the top. As a result, although DNS and location service lookups were required to route the initial INVITE, the 180 (Ringing) response can be returned to the caller without lookups or without state being

maintained in the proxies. This also has the desirable property that each proxy that sees the INVITE will also see all responses to the INVITE.

When Alice's softphone receives the 180 (Ringing) response, it passes this information to Alice, perhaps using an audio ringback tone or by displaying a message on Alice's screen.

In this example, Bob decides to answer the call. When he picks up the handset, his SIP phone sends a 200 (OK) response to indicate that the call has been answered. The 200 (OK) contains a message body with the SDP media description of the type of session that Bob is willing to establish with Alice. As a result, there is a two-phase exchange of SDP messages: Alice sent one to Bob, and Bob sent one back to Alice. This two-phase exchange provides basic negotiation capabilities and is based on a simple offer/answer model of SDP exchange. If Bob did not wish to answer the call or was busy on another call, an error response would have been sent instead of the 200 (OK), which would have resulted in no media session being established. The complete list of SIP response codes is in Section 21. The 200 (OK) (message F9 in Figure 1) might look like this as Bob sends it out:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP server10.biloxi.com
   ;branch=z9hG4bKnashds8;received=192.0.2.3
Via: SIP/2.0/UDP bigbox3.site3.atlanta.com
   ;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via: SIP/2.0/UDP pc33.atlanta.com
   ;branch=z9hG4bK776asdhds ;received=192.0.2.1
To: Bob <sip:bob@biloxi.com>;tag=a6c85cf
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:bob@192.0.2.4>
Content-Type: application/sdp
Content-Length: 131

(Bob's SDP not shown)
```

The first line of the response contains the response code (200) and the reason phrase (OK). The remaining lines contain header fields. The Via, To, From, Call-ID, and CSeq header fields are copied from the INVITE request. (There are three Via header field values - one added by Alice's SIP phone, one added by the atlanta.com proxy, and one added by the biloxi.com proxy.) Bob's SIP phone has added a tag parameter to the To header field. This tag will be incorporated by both endpoints into the dialog and will be included in all future

requests and responses in this call. The Contact header field contains a URI at which Bob can be directly reached at his SIP phone. The Content-Type and Content-Length refer to the message body (not shown) that contains Bob's SDP media information.

In addition to DNS and location service lookups shown in this example, proxy servers can make flexible "routing decisions" to decide where to send a request. For example, if Bob's SIP phone returned a 486 (Busy Here) response, the biloxi.com proxy server could proxy the INVITE to Bob's voicemail server. A proxy server can also send an INVITE to a number of locations at the same time. This type of parallel search is known as forking.

In this case, the 200 (OK) is routed back through the two proxies and is received by Alice's softphone, which then stops the ringback tone and indicates that the call has been answered. Finally, Alice's softphone sends an acknowledgement message, ACK, to Bob's SIP phone to confirm the reception of the final response (200 (OK)). In this example, the ACK is sent directly from Alice's softphone to Bob's SIP phone, bypassing the two proxies. This occurs because the endpoints have learned each other's address from the Contact header fields through the INVITE/200 (OK) exchange, which was not known when the initial INVITE was sent. The lookups performed by the two proxies are no longer needed, so the proxies drop out of the call flow. This completes the INVITE/200/ACK three-way handshake used to establish SIP sessions. Full details on session setup are in Section 13.

Alice and Bob's media session has now begun, and they send media packets using the format to which they agreed in the exchange of SDP. In general, the end-to-end media packets take a different path from the SIP signaling messages.

During the session, either Alice or Bob may decide to change the characteristics of the media session. This is accomplished by sending a re-INVITE containing a new media description. This re-INVITE references the existing dialog so that the other party knows that it is to modify an existing session instead of establishing a new session. The other party sends a 200 (OK) to accept the change. The requestor responds to the 200 (OK) with an ACK. If the other party does not accept the change, he sends an error response such as 488 (Not Acceptable Here), which also receives an ACK. However, the failure of the re-INVITE does not cause the existing call to fail - the session continues using the previously negotiated characteristics. Full details on session modification are in Section 14.

At the end of the call, Bob disconnects (hangs up) first and generates a BYE message. This BYE is routed directly to Alice's softphone, again bypassing the proxies. Alice confirms receipt of the BYE with a 200 (OK) response, which terminates the session and the BYE transaction. No ACK is sent - an ACK is only sent in response to a response to an INVITE request. The reasons for this special handling for INVITE will be discussed later, but relate to the reliability mechanisms in SIP, the length of time it can take for a ringing phone to be answered, and forking. For this reason, request handling in SIP is often classified as either INVITE or non-

INVITE, referring to all other methods besides INVITE. Full details on session termination are in Section 15.

Section 24.2 describes the messages shown in Figure 1 in full.

In some cases, it may be useful for proxies in the SIP signaling path to see all the messaging between the endpoints for the duration of the session. For example, if the biloxi.com proxy server wished to remain in the SIP messaging path beyond the initial INVITE, it would add to the INVITE a required routing header field known as Record-Route that contained a URI resolving to the hostname or IP address of the proxy. This information would be received by both Bob's SIP phone and (due to the Record-Route header field being passed back in the 200 (OK)) Alice's softphone and stored for the duration of the dialog. The biloxi.com proxy server would then receive and proxy the ACK, BYE, and 200 (OK) to the BYE. Each proxy can independently decide to receive subsequent messages, and those messages will pass through all proxies that elect to receive it. This capability is frequently used for proxies that are providing mid-call features.

Registration is another common operation in SIP. Registration is one way that the biloxi.com server can learn the current location of Bob. Upon initialization, and at periodic intervals, Bob's SIP phone sends REGISTER messages to a server in the biloxi.com domain known as a SIP registrar. The REGISTER messages associate Bob's SIP or SIPS URI (sip:bob@biloxi.com) with the machine into which he is currently logged (conveyed as a SIP or SIPS URI in the Contact header field). The registrar writes this association, also called a binding, to a database, called the location service, where it can be used by the proxy in the biloxi.com domain. Often, a registrar server for a domain is co-located with the proxy for that domain. It is an important concept that the distinction between types of SIP servers is logical, not physical.

Bob is not limited to registering from a single device. For example, both his SIP phone at home and the one in the office could send registrations. This information is stored together in the location

service and allows a proxy to perform various types of searches to locate Bob. Similarly, more than one user can be registered on a single device at the same time.

The location service is just an abstract concept. It generally contains information that allows a proxy to input a URI and receive a set of zero or more URIs that tell the proxy where to send the request. Registrations are one way to create this information, but not the only way. Arbitrary mapping functions can be configured at the discretion of the administrator.

Finally, it is important to note that in SIP, registration is used for routing incoming SIP requests and has no role in authorizing outgoing requests. Authorization and authentication are handled in SIP either on a request-by-request basis with a challenge/response mechanism, or by using a lower layer scheme as discussed in Section 26.

The complete set of SIP message details for this registration example

is in Section 24.1.

Additional operations in SIP, such as querying for the capabilities of a SIP server or client using OPTIONS, or canceling a pending request using CANCEL, will be introduced in later sections.

5 Structure of the Protocol

SIP is structured as a layered protocol, which means that its behavior is described in terms of a set of fairly independent processing stages with only a loose coupling between each stage. The protocol behavior is described as layers for the purpose of presentation, allowing the description of functions common across elements in a single section. It does not dictate an implementation in any way. When we say that an element "contains" a layer, we mean it is compliant to the set of rules defined by that layer.

Not every element specified by the protocol contains every layer. Furthermore, the elements specified by SIP are logical elements, not physical ones. A physical realization can choose to act as different logical elements, perhaps even on a transaction-by-transaction basis.

The lowest layer of SIP is its syntax and encoding. Its encoding is specified using an augmented Backus-Naur Form grammar (BNF). The complete BNF is specified in Section 25; an overview of a SIP message's structure can be found in Section 7.

The second layer is the transport layer. It defines how a client sends requests and receives responses and how a server receives requests and sends responses over the network. All SIP elements contain a transport layer. The transport layer is described in Section 18.

The third layer is the transaction layer. Transactions are a fundamental component of SIP. A transaction is a request sent by a client transaction (using the transport layer) to a server transaction, along with all responses to that request sent from the server transaction back to the client. The transaction layer handles application-layer retransmissions, matching of responses to requests, and application-layer timeouts. Any task that a user agent client (UAC) accomplishes takes place using a series of transactions. Discussion of transactions can be found in Section 17. User agents contain a transaction layer, as do stateful proxies. Stateless proxies do not contain a transaction layer. The transaction layer has a client component (referred to as a client transaction) and a server component (referred to as a server transaction), each of which are represented by a finite state machine that is constructed to process a particular request.

The layer above the transaction layer is called the transaction user (TU). Each of the SIP entities, except the stateless proxy, is a transaction user. When a TU wishes to send a request, it creates a client transaction instance and passes it the request along with the destination IP address, port, and transport to which to send the request. A TU that creates a client transaction can also cancel it. When a client cancels a transaction, it requests that the server stop further processing, revert to the state that existed before the

transaction was initiated, and generate a specific error response to that transaction. This is done with a CANCEL request, which constitutes its own transaction, but references the transaction to be cancelled (Section 9).

The SIP elements, that is, user agent clients and servers, stateless and stateful proxies and registrars, contain a core that distinguishes them from each other. Cores, except for the stateless proxy, are transaction users. While the behavior of the UAC and UAS cores depends on the method, there are some common rules for all methods (Section 8). For a UAC, these rules govern the construction of a request; for a UAS, they govern the processing of a request and generating a response. Since registrations play an important role in SIP, a UAS that handles a REGISTER is given the special name registrar. Section 10 describes UAC and UAS core behavior for the REGISTER method. Section 11 describes UAC and UAS core behavior for the OPTIONS method, used for determining the capabilities of a UA.

Certain other requests are sent within a dialog. A dialog is a peer-to-peer SIP relationship between two user agents that persists for some time. The dialog facilitates sequencing of messages and proper routing of requests between the user agents. The INVITE method is the only way defined in this specification to establish a dialog. When a UAC sends a request that is within the context of a dialog, it follows the common UAC rules as discussed in Section 8 but also the rules for mid-dialog requests. Section 12 discusses dialogs and presents the procedures for their construction and maintenance, in addition to construction of requests within a dialog.

The most important method in SIP is the INVITE method, which is used to establish a session between participants. A session is a collection of participants, and streams of media between them, for the purposes of communication. Section 13 discusses how sessions are initiated, resulting in one or more SIP dialogs. Section 14 discusses how characteristics of that session are modified through the use of an INVITE request within a dialog. Finally, section 15 discusses how a session is terminated.

The procedures of Sections 8, 10, 11, 12, 13, 14, and 15 deal entirely with the UA core (Section 9 describes cancellation, which applies to both UA core and proxy core). Section 16 discusses the proxy element, which facilitates routing of messages between user agents.

6 Definitions

The following terms have special significance for SIP.

Address-of-Record: An address-of-record (AOR) is a SIP or SIPS URI that points to a domain with a location service that can map the URI to another URI where the user might be available. Typically, the location service is populated through registrations. An AOR is frequently thought of as the "public address" of the user.

Back-to-Back User Agent: A back-to-back user agent (B2BUA) is a logical entity that receives a request and processes it as a

user agent server (UAS). In order to determine how the request should be answered, it acts as a user agent client (UAC) and generates requests. Unlike a proxy server, it maintains dialog state and must participate in all requests sent on the dialogs it has established. Since it is a concatenation of a UAC and UAS, no explicit definitions are needed for its behavior.

Call: A call is an informal term that refers to some communication between peers, generally set up for the purposes of a multimedia conversation.

Call Leg: Another name for a dialog [31]; no longer used in this specification.

Call Stateful: A proxy is call stateful if it retains state for a dialog from the initiating INVITE to the terminating BYE request. A call stateful proxy is always transaction stateful, but the converse is not necessarily true.

Client: A client is any network element that sends SIP requests and receives SIP responses. Clients may or may not interact directly with a human user. User agent clients and proxies are clients.

Conference: A multimedia session (see below) that contains multiple participants.

Core: Core designates the functions specific to a particular type of SIP entity, i.e., specific to either a stateful or stateless proxy, a user agent or registrar. All cores, except those for the stateless proxy, are transaction users.

Dialog: A dialog is a peer-to-peer SIP relationship between two UAs that persists for some time. A dialog is established by SIP messages, such as a 2xx response to an INVITE request. A dialog is identified by a call identifier, local tag, and a remote tag. A dialog was formerly known as a call leg in RFC 2543.

Downstream: A direction of message forwarding within a transaction that refers to the direction that requests flow from the user agent client to user agent server.

Final Response: A response that terminates a SIP transaction, as opposed to a provisional response that does not. All 2xx, 3xx, 4xx, 5xx and 6xx responses are final.

Header: A header is a component of a SIP message that conveys information about the message. It is structured as a sequence of header fields.

Header Field: A header field is a component of the SIP message header. A header field can appear as one or more header field rows. Header field rows consist of a header field name and zero or more header field values. Multiple header field values on a

given header field row are separated by commas. Some header

fields can only have a single header field value, and as a result, always appear as a single header field row.

Header Field Value: A header field value is a single value; a header field consists of zero or more header field values.

Home Domain: The domain providing service to a SIP user. Typically, this is the domain present in the URI in the address-of-record of a registration.

Informational Response: Same as a provisional response.

Initiator, Calling Party, Caller: The party initiating a session (and dialog) with an INVITE request. A caller retains this role from the time it sends the initial INVITE that established a dialog until the termination of that dialog.

Invitation: An INVITE request.

Invitee, Invited User, Called Party, Callee: The party that receives an INVITE request for the purpose of establishing a new session. A callee retains this role from the time it receives the INVITE until the termination of the dialog established by that INVITE.

Location Service: A location service is used by a SIP redirect or proxy server to obtain information about a callee's possible location(s). It contains a list of bindings of address-of-record keys to zero or more contact addresses. The bindings can be created and removed in many ways; this specification defines a REGISTER method that updates the bindings.

Loop: A request that arrives at a proxy, is forwarded, and later arrives back at the same proxy. When it arrives the second time, its Request-URI is identical to the first time, and other header fields that affect proxy operation are unchanged, so that the proxy would make the same processing decision on the request it made the first time. Looped requests are errors, and the procedures for detecting them and handling them are described by the protocol.

Loose Routing: A proxy is said to be loose routing if it follows the procedures defined in this specification for processing of the Route header field. These procedures separate the destination of the request (present in the Request-URI) from

the set of proxies that need to be visited along the way (present in the Route header field). A proxy compliant to these mechanisms is also known as a loose router.

Message: Data sent between SIP elements as part of the protocol. SIP messages are either requests or responses.

Method: The method is the primary function that a request is meant to invoke on a server. The method is carried in the request message itself. Example methods are INVITE and BYE.

Outbound Proxy: A proxy that receives requests from a client, even
   though it may not be the server resolved by the Request-URI.
   Typically, a UA is manually configured with an outbound proxy,
   or can learn about one through auto-configuration protocols.

Parallel Search: In a parallel search, a proxy issues several
   requests to possible user locations upon receiving an incoming
   request. Rather than issuing one request and then waiting for
   the final response before issuing the next request as in a
   sequential search, a parallel search issues requests without
   waiting for the result of previous requests.

Provisional Response: A response used by the server to indicate
   progress, but that does not terminate a SIP transaction. 1xx
   responses are provisional, other responses are considered
   final.

Proxy, Proxy Server: An intermediary entity that acts as both a
   server and a client for the purpose of making requests on
   behalf of other clients. A proxy server primarily plays the
   role of routing, which means its job is to ensure that a
   request is sent to another entity "closer" to the targeted
   user. Proxies are also useful for enforcing policy (for
   example, making sure a user is allowed to make a call). A
   proxy interprets, and, if necessary, rewrites specific parts of
   a request message before forwarding it.

Recursion: A client recurses on a 3xx response when it generates a
   new request to one or more of the URIs in the Contact header
   field in the response.

Redirect Server: A redirect server is a user agent server that
   generates 3xx responses to requests it receives, directing the
   client to contact an alternate set of URIs.

Registrar: A registrar is a server that accepts REGISTER requests
   and places the information it receives in those requests into
   the location service for the domain it handles.

Regular Transaction: A regular transaction is any transaction with
   a method other than INVITE, ACK, or CANCEL.

Request: A SIP message sent from a client to a server, for the
   purpose of invoking a particular operation.

Response: A SIP message sent from a server to a client, for
   indicating the status of a request sent from the client to the
   server.

Ringback: Ringback is the signaling tone produced by the calling
   party's application indicating that a called party is being
   alerted (ringing).

Route Set: A route set is a collection of ordered SIP or SIPS URI
   which represent a list of proxies that must be traversed when
   sending a particular request. A route set can be learned,
   through headers like Record-Route, or it can be configured.

Server: A server is a network element that receives requests in order to service them and sends back responses to those requests. Examples of servers are proxies, user agent servers, redirect servers, and registrars.

Sequential Search: In a sequential search, a proxy server attempts each contact address in sequence, proceeding to the next one only after the previous has generated a final response. A 2xx or 6xx class final response always terminates a sequential search.

Session: From the SDP specification: "A multimedia session is a set of multimedia senders and receivers and the data streams flowing from senders to receivers. A multimedia conference is an example of a multimedia session." (RFC 2327 [1]) (A session as defined for SDP can comprise one or more RTP sessions.) As defined, a callee can be invited several times, by different calls, to the same session. If SDP is used, a session is defined by the concatenation of the SDP user name, session id, network type, address type, and address elements in the origin field.

SIP Transaction: A SIP transaction occurs between a client and a server and comprises all messages from the first request sent from the client to the server up to a final (non-1xx) response

sent from the server to the client. If the request is INVITE and the final response is a non-2xx, the transaction also includes an ACK to the response. The ACK for a 2xx response to an INVITE request is a separate transaction.

Spiral: A spiral is a SIP request that is routed to a proxy, forwarded onwards, and arrives once again at that proxy, but this time differs in a way that will result in a different processing decision than the original request. Typically, this means that the request's Request-URI differs from its previous arrival. A spiral is not an error condition, unlike a loop. A typical cause for this is call forwarding. A user calls joe@example.com. The example.com proxy forwards it to Joe's PC, which in turn, forwards it to bob@example.com. This request is proxied back to the example.com proxy. However, this is not a loop. Since the request is targeted at a different user, it is considered a spiral, and is a valid condition.

Stateful Proxy: A logical entity that maintains the client and server transaction state machines defined by this specification during the processing of a request, also known as a transaction stateful proxy. The behavior of a stateful proxy is further defined in Section 16. A (transaction) stateful proxy is not the same as a call stateful proxy.

Stateless Proxy: A logical entity that does not maintain the client or server transaction state machines defined in this specification when it processes requests. A stateless proxy forwards every request it receives downstream and every

response it receives upstream.

Strict Routing: A proxy is said to be strict routing if it follows the Route processing rules of RFC 2543 and many prior work in progress versions of this RFC. That rule caused proxies to destroy the contents of the Request-URI when a Route header field was present. Strict routing behavior is not used in this specification, in favor of a loose routing behavior. Proxies that perform strict routing are also known as strict routers.

Target Refresh Request: A target refresh request sent within a dialog is defined as a request that can modify the remote target of the dialog.

Transaction User (TU): The layer of protocol processing that resides above the transaction layer. Transaction users include the UAC core, UAS core, and proxy core.

Upstream: A direction of message forwarding within a transaction that refers to the direction that responses flow from the user agent server back to the user agent client.

URL-encoded: A character string encoded according to RFC 2396, Section 2.4 [5].

User Agent Client (UAC): A user agent client is a logical entity that creates a new request, and then uses the client transaction state machinery to send it. The role of UAC lasts only for the duration of that transaction. In other words, if a piece of software initiates a request, it acts as a UAC for the duration of that transaction. If it receives a request later, it assumes the role of a user agent server for the processing of that transaction.

UAC Core: The set of processing functions required of a UAC that reside above the transaction and transport layers.

User Agent Server (UAS): A user agent server is a logical entity that generates a response to a SIP request. The response accepts, rejects, or redirects the request. This role lasts only for the duration of that transaction. In other words, if a piece of software responds to a request, it acts as a UAS for the duration of that transaction. If it generates a request later, it assumes the role of a user agent client for the processing of that transaction.

UAS Core: The set of processing functions required at a UAS that resides above the transaction and transport layers.

User Agent (UA): A logical entity that can act as both a user agent client and user agent server.

The role of UAC and UAS, as well as proxy and redirect servers, are defined on a transaction-by-transaction basis. For example, the user agent initiating a call acts as a UAC when sending the initial INVITE request and as a UAS when receiving a BYE request from the callee. Similarly, the same software can act as a proxy server for one

request and as a redirect server for the next request.

Proxy, location, and registrar servers defined above are logical entities; implementations MAY combine them into a single application.

7 SIP Messages

SIP is a text-based protocol and uses the UTF-8 charset (RFC 2279 [7]).

A SIP message is either a request from a client to a server, or a response from a server to a client.

Both Request (section 7.1) and Response (section 7.2) messages use the basic format of RFC 2822 [3], even though the syntax differs in character set and syntax specifics. (SIP allows header fields that would not be valid RFC 2822 header fields, for example.) Both types of messages consist of a start-line, one or more header fields, an empty line indicating the end of the header fields, and an optional message-body.

```
generic-message  =  start-line
                    *message-header
                    CRLF
                    [ message-body ]
start-line       =  Request-Line / Status-Line
```

The start-line, each message-header line, and the empty line MUST be terminated by a carriage-return line-feed sequence (CRLF). Note that the empty line MUST be present even if the message-body is not.

Except for the above difference in character sets, much of SIP's message and header field syntax is identical to HTTP/1.1. Rather than repeating the syntax and semantics here, we use [HX.Y] to refer to Section X.Y of the current HTTP/1.1 specification (RFC 2616 [8]).

However, SIP is not an extension of HTTP.

7.1 Requests

SIP requests are distinguished by having a Request-Line for a start-line. A Request-Line contains a method name, a Request-URI, and the protocol version separated by a single space (SP) character.

The Request-Line ends with CRLF. No CR or LF are allowed except in the end-of-line CRLF sequence. No linear whitespace (LWS) is allowed in any of the elements.

```
Request-Line  =  Method SP Request-URI SP SIP-Version CRLF
```

Method: This specification defines six methods: REGISTER for registering contact information, INVITE, ACK, and CANCEL for setting up sessions, BYE for terminating sessions, and OPTIONS for querying servers about their capabilities. SIP extensions, documented in standards track RFCs, may define additional methods.

Request-URI: The Request-URI is a SIP or SIPS URI as described in Section 19.1 or a general URI (RFC 2396 [5]). It indicates the user or service to which this request is being addressed. The Request-URI MUST NOT contain unescaped spaces or control characters and MUST NOT be enclosed in "<>".

SIP elements MAY support Request-URIs with schemes other than "sip" and "sips", for example the "tel" URI scheme of RFC 2806 [9]. SIP elements MAY translate non-SIP URIs using any mechanism at their disposal, resulting in SIP URI, SIPS URI, or some other scheme.

SIP-Version: Both request and response messages include the version of SIP in use, and follow [H3.1] (with HTTP replaced by SIP, and HTTP/1.1 replaced by SIP/2.0) regarding version ordering, compliance requirements, and upgrading of version numbers. To be compliant with this specification, applications sending SIP messages MUST include a SIP-Version of "SIP/2.0". The SIP-Version string is case-insensitive, but implementations MUST send upper-case.

Unlike HTTP/1.1, SIP treats the version number as a literal string. In practice, this should make no difference.

7.2 Responses

SIP responses are distinguished from requests by having a Status-Line as their start-line. A Status-Line consists of the protocol version followed by a numeric Status-Code and its associated textual phrase, with each element separated by a single SP character.

No CR or LF is allowed except in the final CRLF sequence.

```
Status-Line  =  SIP-Version SP Status-Code SP Reason-Phrase CRLF
```

The Status-Code is a 3-digit integer result code that indicates the outcome of an attempt to understand and satisfy a request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata, whereas the Reason-Phrase is intended for the human user. A client is not required to examine or display the Reason-Phrase.

While this specification suggests specific wording for the reason phrase, implementations MAY choose other text, for example, in the language indicated in the Accept-Language header field of the request.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. For this reason, any response with a status code between 100 and 199 is referred to as a "1xx response", any response with a status code between 200 and 299 as a "2xx response", and so on. SIP/2.0 allows six values for the first digit:

1xx: Provisional -- request received, continuing to process the request;

2xx: Success -- the action was successfully received, understood, and accepted;

3xx: Redirection -- further action needs to be taken in order to complete the request;

4xx: Client Error -- the request contains bad syntax or cannot be fulfilled at this server;

5xx: Server Error -- the server failed to fulfill an apparently valid request;

6xx: Global Failure -- the request cannot be fulfilled at any server.

Section 21 defines these classes and describes the individual codes.

7.3 Header Fields

SIP header fields are similar to HTTP header fields in both syntax and semantics. In particular, SIP header fields follow the [H4.2] definitions of syntax for the message-header and the rules for extending header fields over multiple lines. However, the latter is specified in HTTP with implicit whitespace and folding. This specification conforms to RFC 2234 [10] and uses only explicit whitespace and folding as an integral part of the grammar.

[H4.2] also specifies that multiple header fields of the same field name whose value is a comma-separated list can be combined into one header field. That applies to SIP as well, but the specific rule is different because of the different grammars. Specifically, any SIP header whose grammar is of the form

```
header  =  "header-name" HCOLON header-value *(COMMA header-value)
```

allows for combining header fields of the same name into a comma-separated list. The Contact header field allows a comma-separated list unless the header field value is "*".

7.3.1 Header Field Format

Header fields follow the same generic header format as that given in Section 2.2 of RFC 2822 [3]. Each header field consists of a field name followed by a colon (":") and the field value.

```
field-name: field-value
```

The formal grammar for a message-header specified in Section 25 allows for an arbitrary amount of whitespace on either side of the colon; however, implementations should avoid spaces between the field name and the colon and use a single space (SP) between the colon and the field-value.

```
Subject:            lunch
Subject      :      lunch
Subject            :lunch
Subject: lunch
```

Thus, the above are all valid and equivalent, but the last is the preferred form.

Header fields can be extended over multiple lines by preceding each extra line with at least one SP or horizontal tab (HT). The line break and the whitespace at the beginning of the next line are treated as a single SP character. Thus, the following are equivalent:

```
Subject: I know you're there, pick up the phone and talk to me!
Subject: I know you're there,
         pick up the phone
         and talk to me!
```

The relative order of header fields with different field names is not significant. However, it is RECOMMENDED that header fields which are needed for proxy processing (Via, Route, Record-Route, Proxy-Require, Max-Forwards, and Proxy-Authorization, for example) appear towards the top of the message to facilitate rapid parsing. The relative order of header field rows with the same field name is important. Multiple header field rows with the same field-name MAY be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list (that is, if follows the grammar defined in Section 7.3). It MUST be possible to combine the multiple header field rows into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma. The exceptions to this rule are the WWW-Authenticate, Authorization, Proxy-Authenticate, and Proxy-Authorization header fields. Multiple header

field rows with these names MAY be present in a message, but since their grammar does not follow the general form listed in Section 7.3, they MUST NOT be combined into a single header field row.

Implementations MUST be able to process multiple header field rows with the same name in any combination of the single-value-per-line or comma-separated value forms.

The following groups of header field rows are valid and equivalent:

```
Route: <sip:alice@atlanta.com>
Subject: Lunch
Route: <sip:bob@biloxi.com>
Route: <sip:carol@chicago.com>

Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>
Route: <sip:carol@chicago.com>
Subject: Lunch

Subject: Lunch
Route: <sip:alice@atlanta.com>, <sip:bob@biloxi.com>,
       <sip:carol@chicago.com>
```

Each of the following blocks is valid but not equivalent to the others:

```
   Route: <sip:alice@atlanta.com>
   Route: <sip:bob@biloxi.com>
   Route: <sip:carol@chicago.com>

   Route: <sip:bob@biloxi.com>
   Route: <sip:alice@atlanta.com>
   Route: <sip:carol@chicago.com>

   Route: <sip:alice@atlanta.com>,<sip:carol@chicago.com>,
          <sip:bob@biloxi.com>
```

The format of a header field-value is defined per header-name. It will always be either an opaque sequence of TEXT-UTF8 octets, or a combination of whitespace, tokens, separators, and quoted strings. Many existing header fields will adhere to the general form of a value followed by a semi-colon separated sequence of parameter-name, parameter-value pairs:

```
     field-name: field-value *(;parameter-name=parameter-value)
```

Even though an arbitrary number of parameter pairs may be attached to a header field value, any given parameter-name MUST NOT appear more than once.

When comparing header fields, field names are always case-insensitive. Unless otherwise stated in the definition of a particular header field, field values, parameter names, and parameter values are case-insensitive. Tokens are always case-insensitive. Unless specified otherwise, values expressed as quoted strings are case-sensitive. For example,

```
   Contact: <sip:alice@atlanta.com>;expires=3600
```

is equivalent to

```
   CONTACT: <sip:alice@atlanta.com>;ExPiReS=3600
```

and

```
   Content-Disposition: session;handling=optional
```

is equivalent to

```
   content-disposition: Session;HANDLING=OPTIONAL
```

The following two header fields are not equivalent:

```
   Warning: 370 devnull "Choose a bigger pipe"
   Warning: 370 devnull "CHOOSE A BIGGER PIPE"
```

7.3.2 Header Field Classification

Some header fields only make sense in requests or responses. These are called request header fields and response header fields, respectively. If a header field appears in a message not matching its category (such as a request header field in a response), it MUST be ignored. Section 20 defines the classification of each header

field.

7.3.3 Compact Form

SIP provides a mechanism to represent common header field names in an abbreviated form. This may be useful when messages would otherwise become too large to be carried on the transport available to it (exceeding the maximum transmission unit (MTU) when using UDP, for example). These compact forms are defined in Section 20. A compact form MAY be substituted for the longer form of a header field name at any time without changing the semantics of the message. A header

field name MAY appear in both long and short forms within the same message. Implementations MUST accept both the long and short forms of each header name.

7.4 Bodies

Requests, including new requests defined in extensions to this specification, MAY contain message bodies unless otherwise noted. The interpretation of the body depends on the request method.

For response messages, the request method and the response status code determine the type and interpretation of any message body. All responses MAY include a body.

7.4.1 Message Body Type

The Internet media type of the message body MUST be given by the Content-Type header field. If the body has undergone any encoding such as compression, then this MUST be indicated by the Content-Encoding header field; otherwise, Content-Encoding MUST be omitted. If applicable, the character set of the message body is indicated as part of the Content-Type header-field value.

The "multipart" MIME type defined in RFC 2046 [11] MAY be used within the body of the message. Implementations that send requests containing multipart message bodies MUST send a session description as a non-multipart message body if the remote implementation requests this through an Accept header field that does not contain multipart.

SIP messages MAY contain binary bodies or body parts. When no explicit charset parameter is provided by the sender, media subtypes of the "text" type are defined to have a default charset value of "UTF-8".

7.4.2 Message Body Length

The body length in bytes is provided by the Content-Length header field. Section 20.14 describes the necessary contents of this header field in detail.

The "chunked" transfer encoding of HTTP/1.1 MUST NOT be used for SIP. (Note: The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator.)

7.5 Framing SIP Messages

Unlike HTTP, SIP implementations can use UDP or other unreliable datagram protocols. Each such datagram carries one request or response. See Section 18 on constraints on usage of unreliable transports.

Implementations processing SIP messages over stream-oriented transports MUST ignore any CRLF appearing before the start-line [H4.1].

> The Content-Length header field value is used to locate the end of each SIP message in a stream. It will always be present when SIP messages are sent over stream-oriented transports.

8 General User Agent Behavior

A user agent represents an end system. It contains a user agent client (UAC), which generates requests, and a user agent server (UAS), which responds to them. A UAC is capable of generating a request based on some external stimulus (the user clicking a button, or a signal on a PSTN line) and processing a response. A UAS is capable of receiving a request and generating a response based on user input, external stimulus, the result of a program execution, or some other mechanism.

When a UAC sends a request, the request passes through some number of proxy servers, which forward the request towards the UAS. When the UAS generates a response, the response is forwarded towards the UAC.

UAC and UAS procedures depend strongly on two factors. First, based on whether the request or response is inside or outside of a dialog, and second, based on the method of a request. Dialogs are discussed thoroughly in Section 12; they represent a peer-to-peer relationship between user agents and are established by specific SIP methods, such as INVITE.

In this section, we discuss the method-independent rules for UAC and UAS behavior when processing requests that are outside of a dialog. This includes, of course, the requests which themselves establish a dialog.

Security procedures for requests and responses outside of a dialog are described in Section 26. Specifically, mechanisms exist for the UAS and UAC to mutually authenticate. A limited set of privacy features are also supported through encryption of bodies using S/MIME.

8.1 UAC Behavior

This section covers UAC behavior outside of a dialog.

8.1.1 Generating the Request

A valid SIP request formulated by a UAC MUST, at a minimum, contain the following header fields: To, From, CSeq, Call-ID, Max-Forwards, and Via; all of these header fields are mandatory in all SIP

requests. These six header fields are the fundamental building blocks of a SIP message, as they jointly provide for most of the critical message routing services including the addressing of messages, the routing of responses, limiting message propagation, ordering of messages, and the unique identification of transactions. These header fields are in addition to the mandatory request line, which contains the method, Request-URI, and SIP version.

Examples of requests sent outside of a dialog include an INVITE to establish a session (Section 13) and an OPTIONS to query for capabilities (Section 11).

8.1.1.1 Request-URI

The initial Request-URI of the message SHOULD be set to the value of the URI in the To field. One notable exception is the REGISTER method; behavior for setting the Request-URI of REGISTER is given in Section 10. It may also be undesirable for privacy reasons or convenience to set these fields to the same value (especially if the originating UA expects that the Request-URI will be changed during transit).

In some special circumstances, the presence of a pre-existing route set can affect the Request-URI of the message. A pre-existing route set is an ordered set of URIs that identify a chain of servers, to which a UAC will send outgoing requests that are outside of a dialog. Commonly, they are configured on the UA by a user or service provider manually, or through some other non-SIP mechanism. When a provider wishes to configure a UA with an outbound proxy, it is RECOMMENDED that this be done by providing it with a pre-existing route set with a single URI, that of the outbound proxy.

When a pre-existing route set is present, the procedures for populating the Request-URI and Route header field detailed in Section 12.2.1.1 MUST be followed (even though there is no dialog), using the desired Request-URI as the remote target URI.

8.1.1.2 To

The To header field first and foremost specifies the desired "logical" recipient of the request, or the address-of-record of the user or resource that is the target of this request. This may or may not be the ultimate recipient of the request. The To header field MAY contain a SIP or SIPS URI, but it may also make use of other URI schemes (the tel URL (RFC 2806 [9]), for example) when appropriate. All SIP implementations MUST support the SIP URI scheme. Any implementation that supports TLS MUST support the SIPS URI scheme. The To header field allows for a display name.

A UAC may learn how to populate the To header field for a particular request in a number of ways. Usually the user will suggest the To header field through a human interface, perhaps inputting the URI manually or selecting it from some sort of address book. Frequently, the user will not enter a complete URI, but rather a string of digits or letters (for example, "bob"). It is at the discretion of the UA to choose how to interpret this input. Using the string to form the user part of a SIP URI implies that the UA wishes the name to be

resolved in the domain to the right-hand side (RHS) of the at-sign in the SIP URI (for instance, sip:bob@example.com). Using the string to form the user part of a SIPS URI implies that the UA wishes to communicate securely, and that the name is to be resolved in the domain to the RHS of the at-sign. The RHS will frequently be the home domain of the requestor, which allows for the home domain to process the outgoing request. This is useful for features like "speed dial" that require interpretation of the user part in the home domain. The tel URL may be used when the UA does not wish to specify the domain that should interpret a telephone number that has been input by the user. Rather, each domain through which the request passes would be given that opportunity. As an example, a user in an airport might log in and send requests through an outbound proxy in the airport. If they enter "411" (this is the phone number for local directory assistance in the United States), that needs to be interpreted and processed by the outbound proxy in the airport, not the user's home domain. In this case, tel:411 would be the right choice.

A request outside of a dialog MUST NOT contain a To tag; the tag in the To field of a request identifies the peer of the dialog. Since no dialog is established, no tag is present.

For further information on the To header field, see Section 20.39. The following is an example of a valid To header field:

```
To: Carol <sip:carol@chicago.com>
```

8.1.1.3 From

The From header field indicates the logical identity of the initiator of the request, possibly the user's address-of-record. Like the To header field, it contains a URI and optionally a display name. It is used by SIP elements to determine which processing rules to apply to a request (for example, automatic call rejection). As such, it is very important that the From URI not contain IP addresses or the FQDN of the host on which the UA is running, since these are not logical names.

The From header field allows for a display name. A UAC SHOULD use the display name "Anonymous", along with a syntactically correct, but otherwise meaningless URI (like sip:thisis@anonymous.invalid), if the identity of the client is to remain hidden.

Usually, the value that populates the From header field in requests generated by a particular UA is pre-provisioned by the user or by the administrators of the user's local domain. If a particular UA is used by multiple users, it might have switchable profiles that include a URI corresponding to the identity of the profiled user. Recipients of requests can authenticate the originator of a request in order to ascertain that they are who their From header field claims they are (see Section 22 for more on authentication).

The From field MUST contain a new "tag" parameter, chosen by the UAC. See Section 19.3 for details on choosing a tag.

For further information on the From header field, see Section 20.20.

Examples:

From: "Bob" <sips:bob@biloxi.com> ;tag=a48s
From: sip:+12125551212@phone2net.com;tag=887s
From: Anonymous <sip:c8oqz84zk7z@privacy.org>;tag=hyh8

8.1.1.4 Call-ID

The Call-ID header field acts as a unique identifier to group together a series of messages. It MUST be the same for all requests and responses sent by either UA in a dialog. It SHOULD be the same in each registration from a UA.

In a new request created by a UAC outside of any dialog, the Call-ID header field MUST be selected by the UAC as a globally unique identifier over space and time unless overridden by method-specific behavior. All SIP UAs must have a means to guarantee that the Call-ID header fields they produce will not be inadvertently generated by any other UA. Note that when requests are retried after certain

failure responses that solicit an amendment to a request (for example, a challenge for authentication), these retried requests are not considered new requests, and therefore do not need new Call-ID header fields; see Section 8.1.3.5.

Use of cryptographically random identifiers (RFC 1750 [12]) in the generation of Call-IDs is RECOMMENDED. Implementations MAY use the form "localid@host". Call-IDs are case-sensitive and are simply compared byte-by-byte.

Using cryptographically random identifiers provides some protection against session hijacking and reduces the likelihood of unintentional Call-ID collisions.

No provisioning or human interface is required for the selection of the Call-ID header field value for a request.

For further information on the Call-ID header field, see Section 20.8.

Example:

Call-ID: f81d4fae-7dec-11d0-a765-00a0c91e6bf6@foo.bar.com

8.1.1.5 CSeq

The CSeq header field serves as a way to identify and order transactions. It consists of a sequence number and a method. The method MUST match that of the request. For non-REGISTER requests outside of a dialog, the sequence number value is arbitrary. The sequence number value MUST be expressible as a 32-bit unsigned integer and MUST be less than 2**31. As long as it follows the above guidelines, a client may use any mechanism it would like to select CSeq header field values.

Section 12.2.1.1 discusses construction of the CSeq for requests within a dialog.

Example:

CSeq: 4711 INVITE

8.1.1.6 Max-Forwards

The Max-Forwards header field serves to limit the number of hops a request can transit on the way to its destination. It consists of an integer that is decremented by one at each hop. If the Max-Forwards value reaches 0 before the request reaches its destination, it will be rejected with a 483(Too Many Hops) error response.

A UAC MUST insert a Max-Forwards header field into each request it originates with a value that SHOULD be 70. This number was chosen to be sufficiently large to guarantee that a request would not be dropped in any SIP network when there were no loops, but not so large as to consume proxy resources when a loop does occur. Lower values should be used with caution and only in networks where topologies are known by the UA.

8.1.1.7 Via

The Via header field indicates the transport used for the transaction and identifies the location where the response is to be sent. A Via header field value is added only after the transport that will be used to reach the next hop has been selected (which may involve the usage of the procedures in [4]).

When the UAC creates a request, it MUST insert a Via into that request. The protocol name and protocol version in the header field MUST be SIP and 2.0, respectively. The Via header field value MUST contain a branch parameter. This parameter is used to identify the transaction created by that request. This parameter is used by both the client and the server.

The branch parameter value MUST be unique across space and time for all requests sent by the UA. The exceptions to this rule are CANCEL and ACK for non-2xx responses. As discussed below, a CANCEL request will have the same value of the branch parameter as the request it cancels. As discussed in Section 17.1.1.3, an ACK for a non-2xx response will also have the same branch ID as the INVITE whose response it acknowledges.

> The uniqueness property of the branch ID parameter, to facilitate its use as a transaction ID, was not part of RFC 2543.

The branch ID inserted by an element compliant with this specification MUST always begin with the characters "z9hG4bK". These 7 characters are used as a magic cookie (7 is deemed sufficient to ensure that an older RFC 2543 implementation would not pick such a value), so that servers receiving the request can determine that the branch ID was constructed in the fashion described by this

specification (that is, globally unique). Beyond this requirement, the precise format of the branch token is implementation-defined.

The Via header maddr, ttl, and sent-by components will be set when the request is processed by the transport layer (Section 18).

Via processing for proxies is described in Section 16.6 Item 8 and Section 16.7 Item 3.

8.1.1.8 Contact

The Contact header field provides a SIP or SIPS URI that can be used to contact that specific instance of the UA for subsequent requests. The Contact header field MUST be present and contain exactly one SIP or SIPS URI in any request that can result in the establishment of a dialog. For the methods defined in this specification, that includes only the INVITE request. For these requests, the scope of the Contact is global. That is, the Contact header field value contains the URI at which the UA would like to receive requests, and this URI MUST be valid even if used in subsequent requests outside of any dialogs.

If the Request-URI or top Route header field value contains a SIPS URI, the Contact header field MUST contain a SIPS URI as well.

For further information on the Contact header field, see Section 20.10.

8.1.1.9 Supported and Require

If the UAC supports extensions to SIP that can be applied by the server to the response, the UAC SHOULD include a Supported header field in the request listing the option tags (Section 19.2) for those extensions.

The option tags listed MUST only refer to extensions defined in standards-track RFCs. This is to prevent servers from insisting that clients implement non-standard, vendor-defined features in order to receive service. Extensions defined by experimental and informational RFCs are explicitly excluded from usage with the Supported header field in a request, since they too are often used to document vendor-defined extensions.

If the UAC wishes to insist that a UAS understand an extension that the UAC will apply to the request in order to process the request, it MUST insert a Require header field into the request listing the option tag for that extension. If the UAC wishes to apply an extension to the request and insist that any proxies that are

traversed understand that extension, it MUST insert a Proxy-Require header field into the request listing the option tag for that extension.

As with the Supported header field, the option tags in the Require and Proxy-Require header fields MUST only refer to extensions defined in standards-track RFCs.

8.1.1.10 Additional Message Components

After a new request has been created, and the header fields described

above have been properly constructed, any additional optional header fields are added, as are any header fields specific to the method.

SIP requests MAY contain a MIME-encoded message-body. Regardless of the type of body that a request contains, certain header fields must be formulated to characterize the contents of the body. For further information on these header fields, see Sections 20.11 through 20.15.

8.1.2 Sending the Request

The destination for the request is then computed. Unless there is local policy specifying otherwise, the destination MUST be determined by applying the DNS procedures described in [4] as follows. If the first element in the route set indicated a strict router (resulting in forming the request as described in Section 12.2.1.1), the procedures MUST be applied to the Request-URI of the request. Otherwise, the procedures are applied to the first Route header field value in the request (if one exists), or to the request's Request-URI if there is no Route header field present. These procedures yield an ordered set of address, port, and transports to attempt. Independent of which URI is used as input to the procedures of [4], if the Request-URI specifies a SIPS resource, the UAC MUST follow the procedures of [4] as if the input URI were a SIPS URI.

Local policy MAY specify an alternate set of destinations to attempt. If the Request-URI contains a SIPS URI, any alternate destinations MUST be contacted with TLS. Beyond that, there are no restrictions on the alternate destinations if the request contains no Route header field. This provides a simple alternative to a pre-existing route set as a way to specify an outbound proxy. However, that approach for configuring an outbound proxy is NOT RECOMMENDED; a pre-existing route set with a single URI SHOULD be used instead. If the request contains a Route header field, the request SHOULD be sent to the locations derived from its topmost value, but MAY be sent to any server that the UA is certain will honor the Route and Request-URI policies specified in this document (as opposed to those in RFC 2543). In particular, a UAC configured with an outbound proxy SHOULD

attempt to send the request to the location indicated in the first Route header field value instead of adopting the policy of sending all messages to the outbound proxy.

> This ensures that outbound proxies that do not add Record-Route header field values will drop out of the path of subsequent requests. It allows endpoints that cannot resolve the first Route URI to delegate that task to an outbound proxy.

The UAC SHOULD follow the procedures defined in [4] for stateful elements, trying each address until a server is contacted. Each try constitutes a new transaction, and therefore each carries a different topmost Via header field value with a new branch parameter. Furthermore, the transport value in the Via header field is set to whatever transport was determined for the target server.

8.1.3 Processing Responses

Responses are first processed by the transport layer and then passed

up to the transaction layer. The transaction layer performs its processing and then passes the response up to the TU. The majority of response processing in the TU is method specific. However, there are some general behaviors independent of the method.

8.1.3.1 Transaction Layer Errors

In some cases, the response returned by the transaction layer will not be a SIP message, but rather a transaction layer error. When a timeout error is received from the transaction layer, it MUST be treated as if a 408 (Request Timeout) status code has been received. If a fatal transport error is reported by the transport layer (generally, due to fatal ICMP errors in UDP or connection failures in TCP), the condition MUST be treated as a 503 (Service Unavailable) status code.

8.1.3.2 Unrecognized Responses

A UAC MUST treat any final response it does not recognize as being equivalent to the x00 response code of that class, and MUST be able to process the x00 response code for all classes. For example, if a UAC receives an unrecognized response code of 431, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 (Bad Request) response code. A UAC MUST treat any provisional response different than 100 that it does not recognize as 183 (Session Progress). A UAC MUST be able to process 100 and 183 responses.

8.1.3.3 Vias

If more than one Via header field value is present in a response, the UAC SHOULD discard the message.

> The presence of additional Via header field values that precede the originator of the request suggests that the message was misrouted or possibly corrupted.

8.1.3.4 Processing 3xx Responses

Upon receipt of a redirection response (for example, a 301 response status code), clients SHOULD use the URI(s) in the Contact header field to formulate one or more new requests based on the redirected request. This process is similar to that of a proxy recursing on a 3xx class response as detailed in Sections 16.5 and 16.6. A client starts with an initial target set containing exactly one URI, the Request-URI of the original request. If a client wishes to formulate new requests based on a 3xx class response to that request, it places the URIs to try into the target set. Subject to the restrictions in this specification, a client can choose which Contact URIs it places into the target set. As with proxy recursion, a client processing 3xx class responses MUST NOT add any given URI to the target set more than once. If the original request had a SIPS URI in the Request-URI, the client MAY choose to recurse to a non-SIPS URI, but SHOULD inform the user of the redirection to an insecure URI.

> Any new request may receive 3xx responses themselves containing the original URI as a contact. Two locations can be configured to

redirect to each other. Placing any given URI in the target set only once prevents infinite redirection loops.

As the target set grows, the client MAY generate new requests to the URIs in any order. A common mechanism is to order the set by the "q" parameter value from the Contact header field value. Requests to the URIs MAY be generated serially or in parallel. One approach is to process groups of decreasing q-values serially and process the URIs in each q-value group in parallel. Another is to perform only serial processing in decreasing q-value order, arbitrarily choosing between contacts of equal q-value.

If contacting an address in the list results in a failure, as defined in the next paragraph, the element moves to the next address in the list, until the list is exhausted. If the list is exhausted, then the request has failed.

Failures SHOULD be detected through failure response codes (codes greater than 399); for network errors the client transaction will report any transport layer failures to the transaction user. Note that some response codes (detailed in 8.1.3.5) indicate that the request can be retried; requests that are reattempted should not be considered failures.

When a failure for a particular contact address is received, the client SHOULD try the next contact address. This will involve creating a new client transaction to deliver a new request.

In order to create a request based on a contact address in a 3xx response, a UAC MUST copy the entire URI from the target set into the Request-URI, except for the "method-param" and "header" URI parameters (see Section 19.1.1 for a definition of these parameters). It uses the "header" parameters to create header field values for the new request, overwriting header field values associated with the redirected request in accordance with the guidelines in Section 19.1.5.

Note that in some instances, header fields that have been communicated in the contact address may instead append to existing request header fields in the original redirected request. As a general rule, if the header field can accept a comma-separated list of values, then the new header field value MAY be appended to any existing values in the original redirected request. If the header field does not accept multiple values, the value in the original redirected request MAY be overwritten by the header field value communicated in the contact address. For example, if a contact address is returned with the following value:

sip:user@host?Subject=foo&Call-Info=<http://www.foo.com>

Then any Subject header field in the original redirected request is overwritten, but the HTTP URL is merely appended to any existing Call-Info header field values.

It is RECOMMENDED that the UAC reuse the same To, From, and Call-ID used in the original redirected request, but the UAC MAY also choose to update the Call-ID header field value for new requests, for

example.

Finally, once the new request has been constructed, it is sent using a new client transaction, and therefore MUST have a new branch ID in the top Via field as discussed in Section 8.1.1.7.

In all other respects, requests sent upon receipt of a redirect response SHOULD re-use the header fields and bodies of the original request.

In some instances, Contact header field values may be cached at UAC temporarily or permanently depending on the status code received and the presence of an expiration interval; see Sections 21.3.2 and 21.3.3.

8.1.3.5 Processing 4xx Responses

Certain 4xx response codes require specific UA processing, independent of the method.

If a 401 (Unauthorized) or 407 (Proxy Authentication Required) response is received, the UAC SHOULD follow the authorization procedures of Section 22.2 and Section 22.3 to retry the request with credentials.

If a 413 (Request Entity Too Large) response is received (Section 21.4.11), the request contained a body that was longer than the UAS was willing to accept. If possible, the UAC SHOULD retry the request, either omitting the body or using one of a smaller length.

If a 415 (Unsupported Media Type) response is received (Section 21.4.13), the request contained media types not supported by the UAS. The UAC SHOULD retry sending the request, this time only using content with types listed in the Accept header field in the response, with encodings listed in the Accept-Encoding header field in the response, and with languages listed in the Accept-Language in the response.

If a 416 (Unsupported URI Scheme) response is received (Section 21.4.14), the Request-URI used a URI scheme not supported by the server. The client SHOULD retry the request, this time, using a SIP URI.

If a 420 (Bad Extension) response is received (Section 21.4.15), the request contained a Require or Proxy-Require header field listing an option-tag for a feature not supported by a proxy or UAS. The UAC SHOULD retry the request, this time omitting any extensions listed in the Unsupported header field in the response.

In all of the above cases, the request is retried by creating a new request with the appropriate modifications. This new request constitutes a new transaction and SHOULD have the same value of the Call-ID, To, and From of the previous request, but the CSeq should contain a new sequence number that is one higher than the previous.

With other 4xx responses, including those yet to be defined, a retry may or may not be possible depending on the method and the use case.

8.2 UAS Behavior

When a request outside of a dialog is processed by a UAS, there is a set of processing rules that are followed, independent of the method. Section 12 gives guidance on how a UAS can tell whether a request is inside or outside of a dialog.

Note that request processing is atomic. If a request is accepted, all state changes associated with it MUST be performed. If it is rejected, all state changes MUST NOT be performed.

UASs SHOULD process the requests in the order of the steps that follow in this section (that is, starting with authentication, then inspecting the method, the header fields, and so on throughout the remainder of this section).

8.2.1 Method Inspection

Once a request is authenticated (or authentication is skipped), the UAS MUST inspect the method of the request. If the UAS recognizes but does not support the method of a request, it MUST generate a 405 (Method Not Allowed) response. Procedures for generating responses are described in Section 8.2.6. The UAS MUST also add an Allow header field to the 405 (Method Not Allowed) response. The Allow header field MUST list the set of methods supported by the UAS generating the message. The Allow header field is presented in Section 20.5.

If the method is one supported by the server, processing continues.

8.2.2 Header Inspection

If a UAS does not understand a header field in a request (that is, the header field is not defined in this specification or in any supported extension), the server MUST ignore that header field and continue processing the message. A UAS SHOULD ignore any malformed header fields that are not necessary for processing requests.

8.2.2.1 To and Request-URI

The To header field identifies the original recipient of the request designated by the user identified in the From field. The original recipient may or may not be the UAS processing the request, due to call forwarding or other proxy operations. A UAS MAY apply any policy it wishes to determine whether to accept requests when the To

header field is not the identity of the UAS. However, it is RECOMMENDED that a UAS accept requests even if they do not recognize the URI scheme (for example, a tel: URI) in the To header field, or if the To header field does not address a known or current user of this UAS. If, on the other hand, the UAS decides to reject the request, it SHOULD generate a response with a 403 (Forbidden) status code and pass it to the server transaction for transmission.

However, the Request-URI identifies the UAS that is to process the request. If the Request-URI uses a scheme not supported by the UAS,

it SHOULD reject the request with a 416 (Unsupported URI Scheme) response. If the Request-URI does not identify an address that the UAS is willing to accept requests for, it SHOULD reject the request with a 404 (Not Found) response. Typically, a UA that uses the REGISTER method to bind its address-of-record to a specific contact address will see requests whose Request-URI equals that contact address. Other potential sources of received Request-URIs include the Contact header fields of requests and responses sent by the UA that establish or refresh dialogs.

8.2.2.2 Merged Requests

If the request has no tag in the To header field, the UAS core MUST check the request against ongoing transactions. If the From tag, Call-ID, and CSeq exactly match those associated with an ongoing transaction, but the request does not match that transaction (based on the matching rules in Section 17.2.3), the UAS core SHOULD generate a 482 (Loop Detected) response and pass it to the server transaction.

> The same request has arrived at the UAS more than once, following different paths, most likely due to forking. The UAS processes the first such request received and responds with a 482 (Loop Detected) to the rest of them.

8.2.2.3 Require

Assuming the UAS decides that it is the proper element to process the request, it examines the Require header field, if present.

The Require header field is used by a UAC to tell a UAS about SIP extensions that the UAC expects the UAS to support in order to process the request properly. Its format is described in Section 20.32. If a UAS does not understand an option-tag listed in a Require header field, it MUST respond by generating a response with status code 420 (Bad Extension). The UAS MUST add an Unsupported header field, and list in it those options it does not understand amongst those in the Require header field of the request.

Note that Require and Proxy-Require MUST NOT be used in a SIP CANCEL request, or in an ACK request sent for a non-2xx response. These header fields MUST be ignored if they are present in these requests.

An ACK request for a 2xx response MUST contain only those Require and Proxy-Require values that were present in the initial request.

Example:

```
UAC->UAS:   INVITE sip:watson@bell-telephone.com SIP/2.0
            Require: 100rel

UAS->UAC:   SIP/2.0 420 Bad Extension
            Unsupported: 100rel
```

> This behavior ensures that the client-server interaction will proceed without delay when all options are understood by both sides, and only slow down if options are not understood (as in the

example above). For a well-matched client-server pair, the interaction proceeds quickly, saving a round-trip often required by negotiation mechanisms. In addition, it also removes ambiguity when the client requires features that the server does not understand. Some features, such as call handling fields, are only of interest to end systems.

8.2.3 Content Processing

Assuming the UAS understands any extensions required by the client, the UAS examines the body of the message, and the header fields that describe it. If there are any bodies whose type (indicated by the Content-Type), language (indicated by the Content-Language) or encoding (indicated by the Content-Encoding) are not understood, and that body part is not optional (as indicated by the Content-Disposition header field), the UAS MUST reject the request with a 415 (Unsupported Media Type) response. The response MUST contain an Accept header field listing the types of all bodies it understands, in the event the request contained bodies of types not supported by the UAS. If the request contained content encodings not understood by the UAS, the response MUST contain an Accept-Encoding header field listing the encodings understood by the UAS. If the request contained content with languages not understood by the UAS, the response MUST contain an Accept-Language header field indicating the languages understood by the UAS. Beyond these checks, body handling depends on the method and type. For further information on the processing of content-specific header fields, see Section 7.4 as well as Section 20.11 through 20.15.

8.2.4 Applying Extensions

A UAS that wishes to apply some extension when generating the response MUST NOT do so unless support for that extension is indicated in the Supported header field in the request. If the desired extension is not supported, the server SHOULD rely only on baseline SIP and any other extensions supported by the client. In rare circumstances, where the server cannot process the request without the extension, the server MAY send a 421 (Extension Required) response. This response indicates that the proper response cannot be generated without support of a specific extension. The needed extension(s) MUST be included in a Require header field in the response. This behavior is NOT RECOMMENDED, as it will generally break interoperability.

Any extensions applied to a non-421 response MUST be listed in a Require header field included in the response. Of course, the server MUST NOT apply extensions not listed in the Supported header field in the request. As a result of this, the Require header field in a response will only ever contain option tags defined in standards-track RFCs.

8.2.5 Processing the Request

Assuming all of the checks in the previous subsections are passed, the UAS processing becomes method-specific. Section 10 covers the REGISTER request, Section 11 covers the OPTIONS request, Section 13 covers the INVITE request, and Section 15 covers the BYE request.

8.2.6 Generating the Response

When a UAS wishes to construct a response to a request, it follows the general procedures detailed in the following subsections. Additional behaviors specific to the response code in question, which are not detailed in this section, may also be required.

Once all procedures associated with the creation of a response have been completed, the UAS hands the response back to the server transaction from which it received the request.

8.2.6.1 Sending a Provisional Response

One largely non-method-specific guideline for the generation of responses is that UASs SHOULD NOT issue a provisional response for a non-INVITE request. Rather, UASs SHOULD generate a final response to a non-INVITE request as soon as possible.

When a 100 (Trying) response is generated, any Timestamp header field present in the request MUST be copied into this 100 (Trying) response. If there is a delay in generating the response, the UAS SHOULD add a delay value into the Timestamp value in the response. This value MUST contain the difference between the time of sending of the response and receipt of the request, measured in seconds.

8.2.6.2 Headers and Tags

The From field of the response MUST equal the From header field of the request. The Call-ID header field of the response MUST equal the Call-ID header field of the request. The CSeq header field of the response MUST equal the CSeq field of the request. The Via header field values in the response MUST equal the Via header field values in the request and MUST maintain the same ordering.

If a request contained a To tag in the request, the To header field in the response MUST equal that of the request. However, if the To header field in the request did not contain a tag, the URI in the To header field in the response MUST equal the URI in the To header field; additionally, the UAS MUST add a tag to the To header field in the response (with the exception of the 100 (Trying) response, in which a tag MAY be present). This serves to identify the UAS that is responding, possibly resulting in a component of a dialog ID. The same tag MUST be used for all responses to that request, both final and provisional (again excepting the 100 (Trying)). Procedures for the generation of tags are defined in Section 19.3.

8.2.7 Stateless UAS Behavior

A stateless UAS is a UAS that does not maintain transaction state. It replies to requests normally, but discards any state that would ordinarily be retained by a UAS after a response has been sent. If a stateless UAS receives a retransmission of a request, it regenerates the response and resends it, just as if it were replying to the first instance of the request. A UAS cannot be stateless unless the request processing for that method would always result in the same response if the requests are identical. This rules out stateless registrars,

for example. Stateless UASs do not use a transaction layer; they receive requests directly from the transport layer and send responses directly to the transport layer.

The stateless UAS role is needed primarily to handle unauthenticated requests for which a challenge response is issued. If unauthenticated requests were handled statefully, then malicious floods of unauthenticated requests could create massive amounts of

transaction state that might slow or completely halt call processing in a UAS, effectively creating a denial of service condition; for more information see Section 26.1.5.

The most important behaviors of a stateless UAS are the following:

- A stateless UAS MUST NOT send provisional (1xx) responses.
- A stateless UAS MUST NOT retransmit responses.
- A stateless UAS MUST ignore ACK requests.
- A stateless UAS MUST ignore CANCEL requests.
- To header tags MUST be generated for responses in a stateless manner - in a manner that will generate the same tag for the same request consistently. For information on tag construction see Section 19.3.

In all other respects, a stateless UAS behaves in the same manner as a stateful UAS. A UAS can operate in either a stateful or stateless mode for each new request.

8.3 Redirect Servers

In some architectures it may be desirable to reduce the processing load on proxy servers that are responsible for routing requests, and improve signaling path robustness, by relying on redirection.

Redirection allows servers to push routing information for a request back in a response to the client, thereby taking themselves out of the loop of further messaging for this transaction while still aiding in locating the target of the request. When the originator of the request receives the redirection, it will send a new request based on the URI(s) it has received. By propagating URIs from the core of the network to its edges, redirection allows for considerable network scalability.

A redirect server is logically constituted of a server transaction layer and a transaction user that has access to a location service of some kind (see Section 10 for more on registrars and location services). This location service is effectively a database containing mappings between a single URI and a set of one or more alternative locations at which the target of that URI can be found.

A redirect server does not issue any SIP requests of its own. After receiving a request other than CANCEL, the server either refuses the request or gathers the list of alternative locations from the

location service and returns a final response of class 3xx. For well-formed CANCEL requests, it SHOULD return a 2xx response. This response ends the SIP transaction. The redirect server maintains transaction state for an entire SIP transaction. It is the responsibility of clients to detect forwarding loops between redirect servers.

When a redirect server returns a 3xx response to a request, it populates the list of (one or more) alternative locations into the Contact header field. An "expires" parameter to the Contact header field values may also be supplied to indicate the lifetime of the Contact data.

The Contact header field contains URIs giving the new locations or user names to try, or may simply specify additional transport parameters. A 301 (Moved Permanently) or 302 (Moved Temporarily) response may also give the same location and username that was targeted by the initial request but specify additional transport parameters such as a different server or multicast address to try, or a change of SIP transport from UDP to TCP or vice versa.

However, redirect servers MUST NOT redirect a request to a URI equal to the one in the Request-URI; instead, provided that the URI does not point to itself, the server MAY proxy the request to the destination URI, or MAY reject it with a 404.

> If a client is using an outbound proxy, and that proxy actually redirects requests, a potential arises for infinite redirection loops.

Note that a Contact header field value MAY also refer to a different resource than the one originally called. For example, a SIP call connected to PSTN gateway may need to deliver a special informational announcement such as "The number you have dialed has been changed."

A Contact response header field can contain any suitable URI indicating where the called party can be reached, not limited to SIP URIs. For example, it could contain URIs for phones, fax, or irc (if they were defined) or a mailto: (RFC 2368 [32]) URL. Section 26.4.4 discusses implications and limitations of redirecting a SIPS URI to a non-SIPS URI.

The "expires" parameter of a Contact header field value indicates how long the URI is valid. The value of the parameter is a number indicating seconds. If this parameter is not provided, the value of the Expires header field determines how long the URI is valid. Malformed values SHOULD be treated as equivalent to 3600.

> This provides a modest level of backwards compatibility with RFC 2543, which allowed absolute times in this header field. If an absolute time is received, it will be treated as malformed, and then default to 3600.

Redirect servers MUST ignore features that are not understood (including unrecognized header fields, any unknown option tags in Require, or even method names) and proceed with the redirection of

the request in question.

9 Canceling a Request

The previous section has discussed general UA behavior for generating requests and processing responses for requests of all methods. In this section, we discuss a general purpose method, called CANCEL.

The CANCEL request, as the name implies, is used to cancel a previous request sent by a client. Specifically, it asks the UAS to cease processing the request and to generate an error response to that request. CANCEL has no effect on a request to which a UAS has already given a final response. Because of this, it is most useful to CANCEL requests to which it can take a server long time to respond. For this reason, CANCEL is best for INVITE requests, which can take a long time to generate a response. In that usage, a UAS that receives a CANCEL request for an INVITE, but has not yet sent a final response, would "stop ringing", and then respond to the INVITE with a specific error response (a 487).

CANCEL requests can be constructed and sent by both proxies and user agent clients. Section 15 discusses under what conditions a UAC would CANCEL an INVITE request, and Section 16.10 discusses proxy usage of CANCEL.

A stateful proxy responds to a CANCEL, rather than simply forwarding a response it would receive from a downstream element. For that reason, CANCEL is referred to as a "hop-by-hop" request, since it is responded to at each stateful proxy hop.

9.1 Client Behavior

A CANCEL request SHOULD NOT be sent to cancel a request other than INVITE.

> Since requests other than INVITE are responded to immediately, sending a CANCEL for a non-INVITE request would always create a race condition.

The following procedures are used to construct a CANCEL request. The Request-URI, Call-ID, To, the numeric part of CSeq, and From header fields in the CANCEL request MUST be identical to those in the request being cancelled, including tags. A CANCEL constructed by a client MUST have only a single Via header field value matching the top Via value in the request being cancelled. Using the same values for these header fields allows the CANCEL to be matched with the request it cancels (Section 9.2 indicates how such matching occurs). However, the method part of the CSeq header field MUST have a value of CANCEL. This allows it to be identified and processed as a transaction in its own right (See Section 17).

If the request being cancelled contains a Route header field, the CANCEL request MUST include that Route header field's values.

> This is needed so that stateless proxies are able to route CANCEL requests properly.

The CANCEL request MUST NOT contain any Require or Proxy-Require header fields.

Once the CANCEL is constructed, the client SHOULD check whether it has received any response (provisional or final) for the request being cancelled (herein referred to as the "original request").

If no provisional response has been received, the CANCEL request MUST NOT be sent; rather, the client MUST wait for the arrival of a provisional response before sending the request. If the original request has generated a final response, the CANCEL SHOULD NOT be sent, as it is an effective no-op, since CANCEL has no effect on requests that have already generated a final response. When the client decides to send the CANCEL, it creates a client transaction for the CANCEL and passes it the CANCEL request along with the destination address, port, and transport. The destination address, port, and transport for the CANCEL MUST be identical to those used to send the original request.

> If it was allowed to send the CANCEL before receiving a response for the previous request, the server could receive the CANCEL before the original request.

Note that both the transaction corresponding to the original request and the CANCEL transaction will complete independently. However, a UAC canceling a request cannot rely on receiving a 487 (Request Terminated) response for the original request, as an RFC 2543-compliant UAS will not generate such a response. If there is no final response for the original request in 64*T1 seconds (T1 is

defined in Section 17.1.1.1), the client SHOULD then consider the original transaction cancelled and SHOULD destroy the client transaction handling the original request.

9.2 Server Behavior

The CANCEL method requests that the TU at the server side cancel a pending transaction. The TU determines the transaction to be cancelled by taking the CANCEL request, and then assuming that the request method is anything but CANCEL or ACK and applying the transaction matching procedures of Section 17.2.3. The matching transaction is the one to be cancelled.

The processing of a CANCEL request at a server depends on the type of server. A stateless proxy will forward it, a stateful proxy might respond to it and generate some CANCEL requests of its own, and a UAS will respond to it. See Section 16.10 for proxy treatment of CANCEL.

A UAS first processes the CANCEL request according to the general UAS processing described in Section 8.2. However, since CANCEL requests are hop-by-hop and cannot be resubmitted, they cannot be challenged by the server in order to get proper credentials in an Authorization header field. Note also that CANCEL requests do not contain a Require header field.

If the UAS did not find a matching transaction for the CANCEL according to the procedure above, it SHOULD respond to the CANCEL

with a 481 (Call Leg/Transaction Does Not Exist). If the transaction for the original request still exists, the behavior of the UAS on receiving a CANCEL request depends on whether it has already sent a final response for the original request. If it has, the CANCEL request has no effect on the processing of the original request, no effect on any session state, and no effect on the responses generated for the original request. If the UAS has not issued a final response for the original request, its behavior depends on the method of the original request. If the original request was an INVITE, the UAS SHOULD immediately respond to the INVITE with a 487 (Request Terminated). A CANCEL request has no impact on the processing of transactions with any other method defined in this specification.

Regardless of the method of the original request, as long as the CANCEL matched an existing transaction, the UAS answers the CANCEL request itself with a 200 (OK) response. This response is constructed following the procedures described in Section 8.2.6 noting that the To tag of the response to the CANCEL and the To tag in the response to the original request SHOULD be the same. The response to CANCEL is passed to the server transaction for transmission.

10 Registrations

10.1 Overview

SIP offers a discovery capability. If a user wants to initiate a session with another user, SIP must discover the current host(s) at which the destination user is reachable. This discovery process is frequently accomplished by SIP network elements such as proxy servers and redirect servers which are responsible for receiving a request, determining where to send it based on knowledge of the location of the user, and then sending it there. To do this, SIP network elements consult an abstract service known as a location service, which provides address bindings for a particular domain. These address bindings map an incoming SIP or SIPS URI, sip:bob@biloxi.com, for example, to one or more URIs that are somehow "closer" to the desired user, sip:bob@engineering.biloxi.com, for example. Ultimately, a proxy will consult a location service that maps a received URI to the user agent(s) at which the desired recipient is currently residing.

Registration creates bindings in a location service for a particular domain that associates an address-of-record URI with one or more contact addresses. Thus, when a proxy for that domain receives a request whose Request-URI matches the address-of-record, the proxy will forward the request to the contact addresses registered to that address-of-record. Generally, it only makes sense to register an address-of-record at a domain's location service when requests for that address-of-record would be routed to that domain. In most cases, this means that the domain of the registration will need to match the domain in the URI of the address-of-record.

There are many ways by which the contents of the location service can be established. One way is administratively. In the above example, Bob is known to be a member of the engineering department through access to a corporate database. However, SIP provides a mechanism

for a UA to create a binding explicitly. This mechanism is known as registration.

Registration entails sending a REGISTER request to a special type of UAS known as a registrar. A registrar acts as the front end to the location service for a domain, reading and writing mappings based on the contents of REGISTER requests. This location service is then typically consulted by a proxy server that is responsible for routing requests for that domain.

An illustration of the overall registration process is given in Figure 2. Note that the registrar and proxy server are logical roles that can be played by a single device in a network; for purposes of

clarity the two are separated in this illustration. Also note that UAs may send requests through a proxy server in order to reach a registrar if the two are separate elements.

SIP does not mandate a particular mechanism for implementing the location service. The only requirement is that a registrar for some domain MUST be able to read and write data to the location service, and a proxy or a redirect server for that domain MUST be capable of reading that same data. A registrar MAY be co-located with a particular SIP proxy server for the same domain.

10.2 Constructing the REGISTER Request

REGISTER requests add, remove, and query bindings. A REGISTER request can add a new binding between an address-of-record and one or more contact addresses. Registration on behalf of a particular address-of-record can be performed by a suitably authorized third party. A client can also remove previous bindings or query to determine which bindings are currently in place for an address-of-record.

Except as noted, the construction of the REGISTER request and the behavior of clients sending a REGISTER request is identical to the general UAC behavior described in Section 8.1 and Section 17.1.

A REGISTER request does not establish a dialog. A UAC MAY include a Route header field in a REGISTER request based on a pre-existing route set as described in Section 8.1. The Record-Route header field has no meaning in REGISTER requests or responses, and MUST be ignored if present. In particular, the UAC MUST NOT create a new route set based on the presence or absence of a Record-Route header field in any response to a REGISTER request.

The following header fields, except Contact, MUST be included in a REGISTER request. A Contact header field MAY be included:

Request-URI: The Request-URI names the domain of the location service for which the registration is meant (for example, "sip:chicago.com"). The "userinfo" and "@" components of the SIP URI MUST NOT be present.

To: The To header field contains the address of record whose registration is to be created, queried, or modified. The To

header field and the Request-URI field typically differ, as the former contains a user name. This address-of-record MUST be a SIP URI or SIPS URI.

From: The From header field contains the address-of-record of the person responsible for the registration. The value is the same as the To header field unless the request is a third-party registration.

Call-ID: All registrations from a UAC SHOULD use the same Call-ID header field value for registrations sent to a particular registrar.

If the same client were to use different Call-ID values, a registrar could not detect whether a delayed REGISTER request might have arrived out of order.

CSeq: The CSeq value guarantees proper ordering of REGISTER requests. A UA MUST increment the CSeq value by one for each REGISTER request with the same Call-ID.

Contact: REGISTER requests MAY contain a Contact header field with zero or more values containing address bindings.

UAs MUST NOT send a new registration (that is, containing new Contact header field values, as opposed to a retransmission) until they have received a final response from the registrar for the previous one or the previous REGISTER request has timed out.

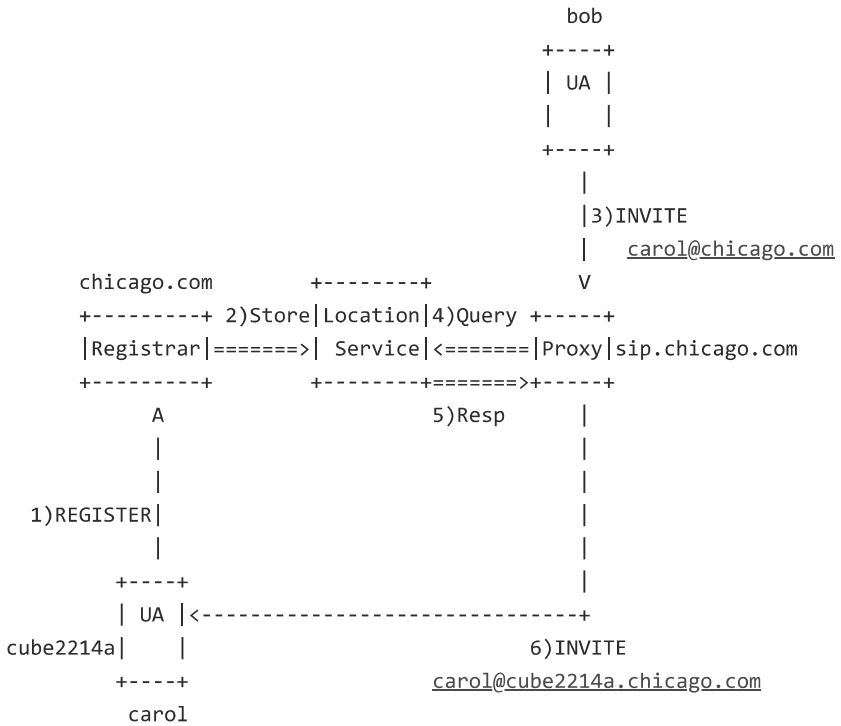


Figure 2: REGISTER example

The following Contact header parameters have a special meaning in REGISTER requests:

action: The "action" parameter from RFC 2543 has been deprecated. UACs SHOULD NOT use the "action" parameter.

expires: The "expires" parameter indicates how long the UA would like the binding to be valid. The value is a number indicating seconds. If this parameter is not provided, the value of the Expires header field is used instead. Implementations MAY treat values larger than 2**32-1 (4294967295 seconds or 136 years) as equivalent to 2**32-1. Malformed values SHOULD be treated as equivalent to 3600.

10.2.1 Adding Bindings

The REGISTER request sent to a registrar includes the contact address(es) to which SIP requests for the address-of-record should be forwarded. The address-of-record is included in the To header field of the REGISTER request.

The Contact header field values of the request typically consist of SIP or SIPS URIs that identify particular SIP endpoints (for example, "sip:carol@cube2214a.chicago.com"), but they MAY use any URI scheme. A SIP UA can choose to register telephone numbers (with the tel URL, RFC 2806 [9]) or email addresses (with a mailto URL, RFC 2368 [32]) as Contacts for an address-of-record, for example.

For example, Carol, with address-of-record "sip:carol@chicago.com", would register with the SIP registrar of the domain chicago.com. Her registrations would then be used by a proxy server in the chicago.com domain to route requests for Carol's address-of-record to her SIP endpoint.

Once a client has established bindings at a registrar, it MAY send subsequent registrations containing new bindings or modifications to existing bindings as necessary. The 2xx response to the REGISTER request will contain, in a Contact header field, a complete list of bindings that have been registered for this address-of-record at this registrar.

If the address-of-record in the To header field of a REGISTER request is a SIPS URI, then any Contact header field values in the request SHOULD also be SIPS URIs. Clients should only register non-SIPS URIs under a SIPS address-of-record when the security of the resource represented by the contact address is guaranteed by other means. This may be applicable to URIs that invoke protocols other than SIP, or SIP devices secured by protocols other than TLS.

Registrations do not need to update all bindings. Typically, a UA only updates its own contact addresses.

10.2.1.1 Setting the Expiration Interval of Contact Addresses

When a client sends a REGISTER request, it MAY suggest an expiration interval that indicates how long the client would like the registration to be valid. (As described in Section 10.3, the registrar selects the actual time interval based on its local policy.)

There are two ways in which a client can suggest an expiration interval for a binding: through an Expires header field or an "expires" Contact header parameter. The latter allows expiration intervals to be suggested on a per-binding basis when more than one binding is given in a single REGISTER request, whereas the former suggests an expiration interval for all Contact header field values that do not contain the "expires" parameter.

If neither mechanism for expressing a suggested expiration time is present in a REGISTER, the client is indicating its desire for the server to choose.

10.2.1.2 Preferences among Contact Addresses

If more than one Contact is sent in a REGISTER request, the registering UA intends to associate all of the URIs in these Contact header field values with the address-of-record present in the To field. This list can be prioritized with the "q" parameter in the Contact header field. The "q" parameter indicates a relative preference for the particular Contact header field value compared to other bindings for this address-of-record. Section 16.6 describes how a proxy server uses this preference indication.

10.2.2 Removing Bindings

Registrations are soft state and expire unless refreshed, but can also be explicitly removed. A client can attempt to influence the expiration interval selected by the registrar as described in Section 10.2.1. A UA requests the immediate removal of a binding by specifying an expiration interval of "0" for that contact address in a REGISTER request. UAs SHOULD support this mechanism so that bindings can be removed before their expiration interval has passed.

The REGISTER-specific Contact header field value of "*" applies to all registrations, but it MUST NOT be used unless the Expires header field is present with a value of "0".

> Use of the "*" Contact header field value allows a registering UA to remove all bindings associated with an address-of-record without knowing their precise values.

10.2.3 Fetching Bindings

A success response to any REGISTER request contains the complete list of existing bindings, regardless of whether the request contained a Contact header field. If no Contact header field is present in a REGISTER request, the list of bindings is left unchanged.

10.2.4 Refreshing Bindings

Each UA is responsible for refreshing the bindings that it has previously established. A UA SHOULD NOT refresh bindings set up by other UAs.

The 200 (OK) response from the registrar contains a list of Contact fields enumerating all current bindings. The UA compares each contact address to see if it created the contact address, using

comparison rules in Section 19.1.4. If so, it updates the expiration time interval according to the expires parameter or, if absent, the Expires field value. The UA then issues a REGISTER request for each of its bindings before the expiration interval has elapsed. It MAY combine several updates into one REGISTER request.

A UA SHOULD use the same Call-ID for all registrations during a single boot cycle. Registration refreshes SHOULD be sent to the same network address as the original registration, unless redirected.

10.2.5 Setting the Internal Clock

If the response for a REGISTER request contains a Date header field, the client MAY use this header field to learn the current time in order to set any internal clocks.

10.2.6 Discovering a Registrar

UAs can use three ways to determine the address to which to send registrations: by configuration, using the address-of-record, and multicast. A UA can be configured, in ways beyond the scope of this specification, with a registrar address. If there is no configured registrar address, the UA SHOULD use the host part of the address-of-record as the Request-URI and address the request there, using the normal SIP server location mechanisms [4]. For example, the UA for the user "sip:carol@chicago.com" addresses the REGISTER request to "sip:chicago.com".

Finally, a UA can be configured to use multicast. Multicast registrations are addressed to the well-known "all SIP servers" multicast address "sip.mcast.net" (224.0.1.75 for IPv4). No well-known IPv6 multicast address has been allocated; such an allocation will be documented separately when needed. SIP UAs MAY listen to that address and use it to become aware of the location of other local users (see [33]); however, they do not respond to the request.

> Multicast registration may be inappropriate in some environments, for example, if multiple businesses share the same local area network.

10.2.7 Transmitting a Request

Once the REGISTER method has been constructed, and the destination of the message identified, UACs follow the procedures described in Section 8.1.2 to hand off the REGISTER to the transaction layer.

If the transaction layer returns a timeout error because the REGISTER yielded no response, the UAC SHOULD NOT immediately re-attempt a registration to the same registrar.

> An immediate re-attempt is likely to also timeout. Waiting some reasonable time interval for the conditions causing the timeout to be corrected reduces unnecessary load on the network. No specific interval is mandated.

10.2.8 Error Responses

If a UA receives a 423 (Interval Too Brief) response, it MAY retry the registration after making the expiration interval of all contact addresses in the REGISTER request equal to or greater than the expiration interval within the Min-Expires header field of the 423 (Interval Too Brief) response.

10.3 Processing REGISTER Requests

A registrar is a UAS that responds to REGISTER requests and maintains a list of bindings that are accessible to proxy servers and redirect servers within its administrative domain. A registrar handles requests according to Section 8.2 and Section 17.2, but it accepts only REGISTER requests. A registrar MUST not generate 6xx responses.

A registrar MAY redirect REGISTER requests as appropriate. One common usage would be for a registrar listening on a multicast interface to redirect multicast REGISTER requests to its own unicast interface with a 302 (Moved Temporarily) response.

Registrars MUST ignore the Record-Route header field if it is included in a REGISTER request. Registrars MUST NOT include a Record-Route header field in any response to a REGISTER request.

> A registrar might receive a request that traversed a proxy which treats REGISTER as an unknown request and which added a Record-Route header field value.

A registrar has to know (for example, through configuration) the set of domain(s) for which it maintains bindings. REGISTER requests MUST be processed by a registrar in the order that they are received. REGISTER requests MUST also be processed atomically, meaning that a particular REGISTER request is either processed completely or not at all. Each REGISTER message MUST be processed independently of any other registration or binding changes.

When receiving a REGISTER request, a registrar follows these steps:

1. The registrar inspects the Request-URI to determine whether it has access to bindings for the domain identified in the Request-URI. If not, and if the server also acts as a proxy server, the server SHOULD forward the request to the addressed domain, following the general behavior for proxying messages described in Section 16.

2. To guarantee that the registrar supports any necessary extensions, the registrar MUST process the Require header field values as described for UASs in Section 8.2.2.

3. A registrar SHOULD authenticate the UAC. Mechanisms for the authentication of SIP user agents are described in Section 22. Registration behavior in no way overrides the generic authentication framework for SIP. If no authentication mechanism is available, the registrar MAY take the From address as the asserted identity of the originator of the request.

4. The registrar SHOULD determine if the authenticated user is authorized to modify registrations for this address-of-record.

For example, a registrar might consult an authorization database that maps user names to a list of addresses-of-record for which that user has authorization to modify bindings. If the authenticated user is not authorized to modify bindings, the registrar MUST return a 403 (Forbidden) and skip the remaining steps.

In architectures that support third-party registration, one entity may be responsible for updating the registrations associated with multiple addresses-of-record.

5. The registrar extracts the address-of-record from the To header field of the request. If the address-of-record is not valid for the domain in the Request-URI, the registrar MUST send a 404 (Not Found) response and skip the remaining steps. The URI MUST then be converted to a canonical form. To do that, all URI parameters MUST be removed (including the user-param), and any escaped characters MUST be converted to their unescaped form. The result serves as an index into the list of bindings.

6. The registrar checks whether the request contains the Contact header field. If not, it skips to the last step. If the Contact header field is present, the registrar checks if there is one Contact field value that contains the special value "*" and an Expires field. If the request has additional Contact fields or an expiration time other than zero, the request is invalid, and the server MUST return a 400 (Invalid Request) and skip the remaining steps. If not, the registrar checks whether the Call-ID agrees with the value stored for each binding. If not, it MUST remove the binding. If it does agree, it MUST remove the binding only if the CSeq in the request is higher than the value stored for that binding. Otherwise, the update MUST be aborted and the request fails.

7. The registrar now processes each contact address in the Contact header field in turn. For each address, it determines the expiration interval as follows:

   - If the field value has an "expires" parameter, that value MUST be taken as the requested expiration.

   - If there is no such parameter, but the request has an Expires header field, that value MUST be taken as the requested expiration.

   - If there is neither, a locally-configured default value MUST be taken as the requested expiration.

   The registrar MAY choose an expiration less than the requested expiration interval. If and only if the requested expiration interval is greater than zero AND smaller than one hour AND less than a registrar-configured minimum, the registrar MAY reject the registration with a response of 423 (Interval Too Brief). This response MUST contain a Min-Expires header field that states the minimum expiration interval the registrar is willing to honor. It then skips the remaining steps.

Allowing the registrar to set the registration interval protects it against excessively frequent registration refreshes while limiting the state that it needs to maintain and decreasing the likelihood of registrations going stale. The expiration interval of a registration is frequently used in the creation of services. An example is a follow-me service, where the user may only be available at a terminal for a brief period. Therefore, registrars should accept brief registrations; a request should only be rejected if the interval is so short that the refreshes would degrade registrar performance.

For each address, the registrar then searches the list of current bindings using the URI comparison rules. If the binding does not exist, it is tentatively added. If the binding does exist, the registrar checks the Call-ID value. If the Call-ID value in the existing binding differs from the Call-ID value in the request, the binding MUST be removed if the expiration time is zero and updated otherwise. If they are the same, the registrar compares the CSeq value. If the value is higher than that of the existing binding, it MUST update or remove the binding as above. If not, the update MUST be aborted and the request fails.

This algorithm ensures that out-of-order requests from the same UA are ignored.

Each binding record records the Call-ID and CSeq values from the request.

The binding updates MUST be committed (that is, made visible to the proxy or redirect server) if and only if all binding updates and additions succeed. If any one of them fails (for example, because the back-end database commit failed), the request MUST fail with a 500 (Server Error) response and all tentative binding updates MUST be removed.

8. The registrar returns a 200 (OK) response. The response MUST contain Contact header field values enumerating all current bindings. Each Contact value MUST feature an "expires" parameter indicating its expiration interval chosen by the registrar. The response SHOULD include a Date header field.

11 Querying for Capabilities

The SIP method OPTIONS allows a UA to query another UA or a proxy server as to its capabilities. This allows a client to discover information about the supported methods, content types, extensions, codecs, etc. without "ringing" the other party. For example, before a client inserts a Require header field into an INVITE listing an option that it is not certain the destination UAS supports, the client can query the destination UAS with an OPTIONS to see if this option is returned in a Supported header field. All UAs MUST support the OPTIONS method.

The target of the OPTIONS request is identified by the Request-URI, which could identify another UA or a SIP server. If the OPTIONS is

addressed to a proxy server, the Request-URI is set without a user part, similar to the way a Request-URI is set for a REGISTER request.

Alternatively, a server receiving an OPTIONS request with a Max-Forwards header field value of 0 MAY respond to the request regardless of the Request-URI.

> This behavior is common with HTTP/1.1. This behavior can be used as a "traceroute" functionality to check the capabilities of individual hop servers by sending a series of OPTIONS requests with incremented Max-Forwards values.

As is the case for general UA behavior, the transaction layer can return a timeout error if the OPTIONS yields no response. This may indicate that the target is unreachable and hence unavailable.

An OPTIONS request MAY be sent as part of an established dialog to query the peer on capabilities that may be utilized later in the dialog.

11.1 Construction of OPTIONS Request

An OPTIONS request is constructed using the standard rules for a SIP request as discussed in Section 8.1.1.

A Contact header field MAY be present in an OPTIONS.

An Accept header field SHOULD be included to indicate the type of message body the UAC wishes to receive in the response. Typically, this is set to a format that is used to describe the media capabilities of a UA, such as SDP (application/sdp).

The response to an OPTIONS request is assumed to be scoped to the Request-URI in the original request. However, only when an OPTIONS is sent as part of an established dialog is it guaranteed that future requests will be received by the server that generated the OPTIONS response.

Example OPTIONS request:

```
OPTIONS sip:carol@chicago.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKhjhs8ass877
Max-Forwards: 70
To: <sip:carol@chicago.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710
CSeq: 63104 OPTIONS
Contact: <sip:alice@pc33.atlanta.com>
Accept: application/sdp
Content-Length: 0
```

11.2 Processing of OPTIONS Request

The response to an OPTIONS is constructed using the standard rules for a SIP response as discussed in Section 8.2.6. The response code chosen MUST be the same that would have been chosen had the request been an INVITE. That is, a 200 (OK) would be returned if the UAS is

ready to accept a call, a 486 (Busy Here) would be returned if the UAS is busy, etc. This allows an OPTIONS request to be used to determine the basic state of a UAS, which can be an indication of whether the UAS will accept an INVITE request.

An OPTIONS request received within a dialog generates a 200 (OK) response that is identical to one constructed outside a dialog and does not have any impact on the dialog.

This use of OPTIONS has limitations due to the differences in proxy handling of OPTIONS and INVITE requests. While a forked INVITE can result in multiple 200 (OK) responses being returned, a forked OPTIONS will only result in a single 200 (OK) response, since it is treated by proxies using the non-INVITE handling. See Section 16.7 for the normative details.

If the response to an OPTIONS is generated by a proxy server, the proxy returns a 200 (OK), listing the capabilities of the server. The response does not contain a message body.

Allow, Accept, Accept-Encoding, Accept-Language, and Supported header fields SHOULD be present in a 200 (OK) response to an OPTIONS request. If the response is generated by a proxy, the Allow header field SHOULD be omitted as it is ambiguous since a proxy is method agnostic. Contact header fields MAY be present in a 200 (OK) response and have the same semantics as in a 3xx response. That is, they may list a set of alternative names and methods of reaching the user. A Warning header field MAY be present.

A message body MAY be sent, the type of which is determined by the Accept header field in the OPTIONS request (application/sdp is the default if the Accept header field is not present). If the types include one that can describe media capabilities, the UAS SHOULD include a body in the response for that purpose. Details on the construction of such a body in the case of application/sdp are described in [13].

Example OPTIONS response generated by a UAS (corresponding to the request in Section 11.1):

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKhjhs8ass877
 ;received=192.0.2.4
To: <sip:carol@chicago.com>;tag=93810874
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710
CSeq: 63104 OPTIONS
Contact: <sip:carol@chicago.com>
Contact: <mailto:carol@chicago.com>
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE
Accept: application/sdp
Accept-Encoding: gzip
Accept-Language: en
Supported: foo
Content-Type: application/sdp
Content-Length: 274
```

(SDP not shown)

12 Dialogs

A key concept for a user agent is that of a dialog. A dialog represents a peer-to-peer SIP relationship between two user agents that persists for some time. The dialog facilitates sequencing of messages between the user agents and proper routing of requests between both of them. The dialog represents a context in which to interpret SIP messages. Section 8 discussed method independent UA processing for requests and responses outside of a dialog. This section discusses how those requests and responses are used to construct a dialog, and then how subsequent requests and responses are sent within a dialog.

A dialog is identified at each UA with a dialog ID, which consists of a Call-ID value, a local tag and a remote tag. The dialog ID at each UA involved in the dialog is not the same. Specifically, the local tag at one UA is identical to the remote tag at the peer UA. The tags are opaque tokens that facilitate the generation of unique dialog IDs.

A dialog ID is also associated with all responses and with any request that contains a tag in the To field. The rules for computing the dialog ID of a message depend on whether the SIP element is a UAC or UAS. For a UAC, the Call-ID value of the dialog ID is set to the Call-ID of the message, the remote tag is set to the tag in the To field of the message, and the local tag is set to the tag in the From

field of the message (these rules apply to both requests and responses). As one would expect for a UAS, the Call-ID value of the dialog ID is set to the Call-ID of the message, the remote tag is set to the tag in the From field of the message, and the local tag is set to the tag in the To field of the message.

A dialog contains certain pieces of state needed for further message transmissions within the dialog. This state consists of the dialog ID, a local sequence number (used to order requests from the UA to its peer), a remote sequence number (used to order requests from its peer to the UA), a local URI, a remote URI, remote target, a boolean flag called "secure", and a route set, which is an ordered list of URIs. The route set is the list of servers that need to be traversed to send a request to the peer. A dialog can also be in the "early" state, which occurs when it is created with a provisional response, and then transition to the "confirmed" state when a 2xx final response arrives. For other responses, or if no response arrives at all on that dialog, the early dialog terminates.

12.1 Creation of a Dialog

Dialogs are created through the generation of non-failure responses to requests with specific methods. Within this specification, only 2xx and 101-199 responses with a To tag, where the request was INVITE, will establish a dialog. A dialog established by a non-final response to a request is in the "early" state and it is called an early dialog. Extensions MAY define other means for creating dialogs. Section 13 gives more details that are specific to the

INVITE method. Here, we describe the process for creation of dialog state that is not dependent on the method.

UAs MUST assign values to the dialog ID components as described below.

12.1.1 UAS behavior

When a UAS responds to a request with a response that establishes a dialog (such as a 2xx to INVITE), the UAS MUST copy all Record-Route header field values from the request into the response (including the URIs, URI parameters, and any Record-Route header field parameters, whether they are known or unknown to the UAS) and MUST maintain the order of those values. The UAS MUST add a Contact header field to the response. The Contact header field contains an address where the UAS would like to be contacted for subsequent requests in the dialog (which includes the ACK for a 2xx response in the case of an INVITE). Generally, the host portion of this URI is the IP address or FQDN of the host. The URI provided in the Contact header field MUST be a SIP or SIPS URI. If the request that initiated the dialog contained a

SIPS URI in the Request-URI or in the top Record-Route header field value, if there was any, or the Contact header field if there was no Record-Route header field, the Contact header field in the response MUST be a SIPS URI. The URI SHOULD have global scope (that is, the same URI can be used in messages outside this dialog). The same way, the scope of the URI in the Contact header field of the INVITE is not limited to this dialog either. It can therefore be used in messages to the UAC even outside this dialog.

The UAS then constructs the state of the dialog. This state MUST be maintained for the duration of the dialog.

If the request arrived over TLS, and the Request-URI contained a SIPS URI, the "secure" flag is set to TRUE.

The route set MUST be set to the list of URIs in the Record-Route header field from the request, taken in order and preserving all URI parameters. If no Record-Route header field is present in the request, the route set MUST be set to the empty set. This route set, even if empty, overrides any pre-existing route set for future requests in this dialog. The remote target MUST be set to the URI from the Contact header field of the request.

The remote sequence number MUST be set to the value of the sequence number in the CSeq header field of the request. The local sequence number MUST be empty. The call identifier component of the dialog ID MUST be set to the value of the Call-ID in the request. The local tag component of the dialog ID MUST be set to the tag in the To field in the response to the request (which always includes a tag), and the remote tag component of the dialog ID MUST be set to the tag from the From field in the request. A UAS MUST be prepared to receive a request without a tag in the From field, in which case the tag is considered to have a value of null.

> This is to maintain backwards compatibility with RFC 2543, which did not mandate From tags.

The remote URI MUST be set to the URI in the From field, and the local URI MUST be set to the URI in the To field.

12.1.2 UAC Behavior

When a UAC sends a request that can establish a dialog (such as an INVITE) it MUST provide a SIP or SIPS URI with global scope (i.e., the same SIP URI can be used in messages outside this dialog) in the Contact header field of the request. If the request has a Request-URI or a topmost Route header field value with a SIPS URI, the Contact header field MUST contain a SIPS URI.

When a UAC receives a response that establishes a dialog, it constructs the state of the dialog. This state MUST be maintained for the duration of the dialog.

If the request was sent over TLS, and the Request-URI contained a SIPS URI, the "secure" flag is set to TRUE.

The route set MUST be set to the list of URIs in the Record-Route header field from the response, taken in reverse order and preserving all URI parameters. If no Record-Route header field is present in the response, the route set MUST be set to the empty set. This route set, even if empty, overrides any pre-existing route set for future requests in this dialog. The remote target MUST be set to the URI from the Contact header field of the response.

The local sequence number MUST be set to the value of the sequence number in the CSeq header field of the request. The remote sequence number MUST be empty (it is established when the remote UA sends a request within the dialog). The call identifier component of the dialog ID MUST be set to the value of the Call-ID in the request. The local tag component of the dialog ID MUST be set to the tag in the From field in the request, and the remote tag component of the dialog ID MUST be set to the tag in the To field of the response. A UAC MUST be prepared to receive a response without a tag in the To field, in which case the tag is considered to have a value of null.

> This is to maintain backwards compatibility with RFC 2543, which did not mandate To tags.

The remote URI MUST be set to the URI in the To field, and the local URI MUST be set to the URI in the From field.

12.2 Requests within a Dialog

Once a dialog has been established between two UAs, either of them MAY initiate new transactions as needed within the dialog. The UA sending the request will take the UAC role for the transaction. The UA receiving the request will take the UAS role. Note that these may be different roles than the UAs held during the transaction that established the dialog.

Requests within a dialog MAY contain Record-Route and Contact header fields. However, these requests do not cause the dialog's route set to be modified, although they may modify the remote target URI.

Specifically, requests that are not target refresh requests do not modify the dialog's remote target URI, and requests that are target refresh requests do. For dialogs that have been established with an

INVITE, the only target refresh request defined is re-INVITE (see Section 14). Other extensions may define different target refresh requests for dialogs established in other ways.

Note that an ACK is NOT a target refresh request.

Target refresh requests only update the dialog's remote target URI, and not the route set formed from the Record-Route. Updating the latter would introduce severe backwards compatibility problems with RFC 2543-compliant systems.

12.2.1 UAC Behavior

12.2.1.1 Generating the Request

A request within a dialog is constructed by using many of the components of the state stored as part of the dialog.

The URI in the To field of the request MUST be set to the remote URI from the dialog state. The tag in the To header field of the request MUST be set to the remote tag of the dialog ID. The From URI of the request MUST be set to the local URI from the dialog state. The tag in the From header field of the request MUST be set to the local tag of the dialog ID. If the value of the remote or local tags is null, the tag parameter MUST be omitted from the To or From header fields, respectively.

Usage of the URI from the To and From fields in the original request within subsequent requests is done for backwards compatibility with RFC 2543, which used the URI for dialog identification. In this specification, only the tags are used for dialog identification. It is expected that mandatory reflection of the original To and From URI in mid-dialog requests will be deprecated in a subsequent revision of this specification.

The Call-ID of the request MUST be set to the Call-ID of the dialog. Requests within a dialog MUST contain strictly monotonically increasing and contiguous CSeq sequence numbers (increasing-by-one) in each direction (excepting ACK and CANCEL of course, whose numbers equal the requests being acknowledged or cancelled). Therefore, if the local sequence number is not empty, the value of the local sequence number MUST be incremented by one, and this value MUST be placed into the CSeq header field. If the local sequence number is empty, an initial value MUST be chosen using the guidelines of Section 8.1.1.5. The method field in the CSeq header field value MUST match the method of the request.

With a length of 32 bits, a client could generate, within a single call, one request a second for about 136 years before needing to wrap around. The initial value of the sequence number is chosen so that subsequent requests within the same call will not wrap around. A non-zero initial value allows clients to use a time-based initial sequence number. A client could, for example,

choose the 31 most significant bits of a 32-bit second clock as an initial sequence number.

The UAC uses the remote target and route set to build the Request-URI and Route header field of the request.

If the route set is empty, the UAC MUST place the remote target URI into the Request-URI. The UAC MUST NOT add a Route header field to the request.

If the route set is not empty, and the first URI in the route set contains the lr parameter (see Section 19.1.1), the UAC MUST place the remote target URI into the Request-URI and MUST include a Route header field containing the route set values in order, including all parameters.

If the route set is not empty, and its first URI does not contain the lr parameter, the UAC MUST place the first URI from the route set into the Request-URI, stripping any parameters that are not allowed in a Request-URI. The UAC MUST add a Route header field containing the remainder of the route set values in order, including all parameters. The UAC MUST then place the remote target URI into the Route header field as the last value.

For example, if the remote target is sip:user@remoteua and the route set contains:

<sip:proxy1>,<sip:proxy2>,<sip:proxy3;lr>,<sip:proxy4>

The request will be formed with the following Request-URI and Route header field:

```
METHOD sip:proxy1
Route: <sip:proxy2>,<sip:proxy3;lr>,<sip:proxy4>,<sip:user@remoteua>
```

If the first URI of the route set does not contain the lr parameter, the proxy indicated does not understand the routing mechanisms described in this document and will act as specified in RFC 2543, replacing the Request-URI with the first Route header field value it receives while forwarding the message. Placing the Request-URI at the end of the Route header field preserves the

information in that Request-URI across the strict router (it will be returned to the Request-URI when the request reaches a loose-router).

A UAC SHOULD include a Contact header field in any target refresh requests within a dialog, and unless there is a need to change it, the URI SHOULD be the same as used in previous requests within the dialog. If the "secure" flag is true, that URI MUST be a SIPS URI. As discussed in Section 12.2.2, a Contact header field in a target refresh request updates the remote target URI. This allows a UA to provide a new contact address, should its address change during the duration of the dialog.

However, requests that are not target refresh requests do not affect the remote target URI for the dialog.

The rest of the request is formed as described in Section 8.1.1.

Once the request has been constructed, the address of the server is computed and the request is sent, using the same procedures for requests outside of a dialog (Section 8.1.2).

> The procedures in Section 8.1.2 will normally result in the request being sent to the address indicated by the topmost Route header field value or the Request-URI if no Route header field is present. Subject to certain restrictions, they allow the request to be sent to an alternate address (such as a default outbound proxy not represented in the route set).

12.2.1.2 Processing the Responses

The UAC will receive responses to the request from the transaction layer. If the client transaction returns a timeout, this is treated as a 408 (Request Timeout) response.

The behavior of a UAC that receives a 3xx response for a request sent within a dialog is the same as if the request had been sent outside a dialog. This behavior is described in Section 8.1.3.4.

> Note, however, that when the UAC tries alternative locations, it still uses the route set for the dialog to build the Route header of the request.

When a UAC receives a 2xx response to a target refresh request, it MUST replace the dialog's remote target URI with the URI from the Contact header field in that response, if present.

If the response for a request within a dialog is a 481 (Call/Transaction Does Not Exist) or a 408 (Request Timeout), the UAC SHOULD terminate the dialog. A UAC SHOULD also terminate a dialog if no response at all is received for the request (the client transaction would inform the TU about the timeout.)

> For INVITE initiated dialogs, terminating the dialog consists of sending a BYE.

12.2.2 UAS Behavior

Requests sent within a dialog, as any other requests, are atomic. If a particular request is accepted by the UAS, all the state changes associated with it are performed. If the request is rejected, none of the state changes are performed.

> Note that some requests, such as INVITEs, affect several pieces of state.

The UAS will receive the request from the transaction layer. If the request has a tag in the To header field, the UAS core computes the dialog identifier corresponding to the request and compares it with existing dialogs. If there is a match, this is a mid-dialog request. In that case, the UAS first applies the same processing rules for requests outside of a dialog, discussed in Section 8.2.

If the request has a tag in the To header field, but the dialog identifier does not match any existing dialogs, the UAS may have crashed and restarted, or it may have received a request for a different (possibly failed) UAS (the UASs can construct the To tags so that a UAS can identify that the tag was for a UAS for which it is providing recovery). Another possibility is that the incoming request has been simply misrouted. Based on the To tag, the UAS MAY either accept or reject the request. Accepting the request for acceptable To tags provides robustness, so that dialogs can persist even through crashes. UAs wishing to support this capability must take into consideration some issues such as choosing monotonically increasing CSeq sequence numbers even across reboots, reconstructing the route set, and accepting out-of-range RTP timestamps and sequence numbers.

If the UAS wishes to reject the request because it does not wish to recreate the dialog, it MUST respond to the request with a 481 (Call/Transaction Does Not Exist) status code and pass that to the server transaction.

Requests that do not change in any way the state of a dialog may be received within a dialog (for example, an OPTIONS request). They are processed as if they had been received outside the dialog.

If the remote sequence number is empty, it MUST be set to the value of the sequence number in the CSeq header field value in the request. If the remote sequence number was not empty, but the sequence number of the request is lower than the remote sequence number, the request is out of order and MUST be rejected with a 500 (Server Internal Error) response. If the remote sequence number was not empty, and the sequence number of the request is greater than the remote sequence number, the request is in order. It is possible for the CSeq sequence number to be higher than the remote sequence number by more than one. This is not an error condition, and a UAS SHOULD be prepared to receive and process requests with CSeq values more than one higher than the previous received request. The UAS MUST then set the remote sequence number to the value of the sequence number in the CSeq header field value in the request.

> If a proxy challenges a request generated by the UAC, the UAC has to resubmit the request with credentials. The resubmitted request will have a new CSeq number. The UAS will never see the first request, and thus, it will notice a gap in the CSeq number space. Such a gap does not represent any error condition.

When a UAS receives a target refresh request, it MUST replace the dialog's remote target URI with the URI from the Contact header field in that request, if present.

12.3 Termination of a Dialog

Independent of the method, if a request outside of a dialog generates a non-2xx final response, any early dialogs created through provisional responses to that request are terminated. The mechanism for terminating confirmed dialogs is method specific. In this specification, the BYE method terminates a session and the dialog

associated with it. See Section 15 for details.

13 Initiating a Session

13.1 Overview

When a user agent client desires to initiate a session (for example, audio, video, or a game), it formulates an INVITE request. The INVITE request asks a server to establish a session. This request may be forwarded by proxies, eventually arriving at one or more UAS that can potentially accept the invitation. These UASs will frequently need to query the user about whether to accept the

invitation. After some time, those UASs can accept the invitation (meaning the session is to be established) by sending a 2xx response. If the invitation is not accepted, a 3xx, 4xx, 5xx or 6xx response is sent, depending on the reason for the rejection. Before sending a final response, the UAS can also send provisional responses (1xx) to advise the UAC of progress in contacting the called user.

After possibly receiving one or more provisional responses, the UAC will get one or more 2xx responses or one non-2xx final response. Because of the protracted amount of time it can take to receive final responses to INVITE, the reliability mechanisms for INVITE transactions differ from those of other requests (like OPTIONS). Once it receives a final response, the UAC needs to send an ACK for every final response it receives. The procedure for sending this ACK depends on the type of response. For final responses between 300 and 699, the ACK processing is done in the transaction layer and follows one set of rules (See Section 17). For 2xx responses, the ACK is generated by the UAC core.

A 2xx response to an INVITE establishes a session, and it also creates a dialog between the UA that issued the INVITE and the UA that generated the 2xx response. Therefore, when multiple 2xx responses are received from different remote UAs (because the INVITE forked), each 2xx establishes a different dialog. All these dialogs are part of the same call.

This section provides details on the establishment of a session using INVITE. A UA that supports INVITE MUST also support ACK, CANCEL and BYE.

13.2 UAC Processing

13.2.1 Creating the Initial INVITE

Since the initial INVITE represents a request outside of a dialog, its construction follows the procedures of Section 8.1.1. Additional processing is required for the specific case of INVITE.

An Allow header field (Section 20.5) SHOULD be present in the INVITE. It indicates what methods can be invoked within a dialog, on the UA sending the INVITE, for the duration of the dialog. For example, a UA capable of receiving INFO requests within a dialog [34] SHOULD include an Allow header field listing the INFO method.

A Supported header field (Section 20.37) SHOULD be present in the INVITE. It enumerates all the extensions understood by the UAC.

An Accept (Section 20.1) header field MAY be present in the INVITE. It indicates which Content-Types are acceptable to the UA, in both the response received by it, and in any subsequent requests sent to it within dialogs established by the INVITE. The Accept header field is especially useful for indicating support of various session description formats.

The UAC MAY add an Expires header field (Section 20.19) to limit the validity of the invitation. If the time indicated in the Expires header field is reached and no final answer for the INVITE has been received, the UAC core SHOULD generate a CANCEL request for the INVITE, as per Section 9.

A UAC MAY also find it useful to add, among others, Subject (Section 20.36), Organization (Section 20.25) and User-Agent (Section 20.41) header fields. They all contain information related to the INVITE.

The UAC MAY choose to add a message body to the INVITE. Section 8.1.1.10 deals with how to construct the header fields -- Content-Type among others -- needed to describe the message body.

There are special rules for message bodies that contain a session description - their corresponding Content-Disposition is "session". SIP uses an offer/answer model where one UA sends a session description, called the offer, which contains a proposed description of the session. The offer indicates the desired communications means (audio, video, games), parameters of those means (such as codec types) and addresses for receiving media from the answerer. The other UA responds with another session description, called the answer, which indicates which communications means are accepted, the parameters that apply to those means, and addresses for receiving media from the offerer. An offer/answer exchange is within the context of a dialog, so that if a SIP INVITE results in multiple dialogs, each is a separate offer/answer exchange. The offer/answer model defines restrictions on when offers and answers can be made (for example, you cannot make a new offer while one is in progress). This results in restrictions on where the offers and answers can appear in SIP messages. In this specification, offers and answers can only appear in INVITE requests and responses, and ACK. The usage of offers and answers is further restricted. For the initial INVITE transaction, the rules are:

- The initial offer MUST be in either an INVITE or, if not there, in the first reliable non-failure message from the UAS back to the UAC. In this specification, that is the final 2xx response.

- If the initial offer is in an INVITE, the answer MUST be in a reliable non-failure message from UAS back to UAC which is correlated to that INVITE. For this specification, that is only the final 2xx response to that INVITE. That same exact answer MAY also be placed in any provisional responses sent prior to the answer. The UAC MUST treat the first session description it receives as the answer, and MUST ignore any

session descriptions in subsequent responses to the initial INVITE.

- If the initial offer is in the first reliable non-failure message from the UAS back to UAC, the answer MUST be in the acknowledgement for that message (in this specification, ACK for a 2xx response).

- After having sent or received an answer to the first offer, the UAC MAY generate subsequent offers in requests based on rules specified for that method, but only if it has received answers to any previous offers, and has not sent any offers to which it hasn't gotten an answer.

- Once the UAS has sent or received an answer to the initial offer, it MUST NOT generate subsequent offers in any responses to the initial INVITE. This means that a UAS based on this specification alone can never generate subsequent offers until completion of the initial transaction.

Concretely, the above rules specify two exchanges for UAs compliant to this specification alone - the offer is in the INVITE, and the answer in the 2xx (and possibly in a 1xx as well, with the same value), or the offer is in the 2xx, and the answer is in the ACK. All user agents that support INVITE MUST support these two exchanges.

The Session Description Protocol (SDP) (RFC 2327 [1]) MUST be supported by all user agents as a means to describe sessions, and its usage for constructing offers and answers MUST follow the procedures defined in [13].

The restrictions of the offer-answer model just described only apply to bodies whose Content-Disposition header field value is "session". Therefore, it is possible that both the INVITE and the ACK contain a body message (for example, the INVITE carries a photo (Content-Disposition: render) and the ACK a session description (Content-Disposition: session)).

If the Content-Disposition header field is missing, bodies of Content-Type application/sdp imply the disposition "session", while other content types imply "render".

Once the INVITE has been created, the UAC follows the procedures defined for sending requests outside of a dialog (Section 8). This results in the construction of a client transaction that will ultimately send the request and deliver responses to the UAC.

13.2.2 Processing INVITE Responses

Once the INVITE has been passed to the INVITE client transaction, the UAC waits for responses for the INVITE. If the INVITE client transaction returns a timeout rather than a response the TU acts as if a 408 (Request Timeout) response had been received, as described in Section 8.1.3.

13.2.2.1 1xx Responses

Zero, one or multiple provisional responses may arrive before one or more final responses are received. Provisional responses for an INVITE request can create "early dialogs". If a provisional response has a tag in the To field, and if the dialog ID of the response does not match an existing dialog, one is constructed using the procedures defined in Section 12.1.2.

The early dialog will only be needed if the UAC needs to send a request to its peer within the dialog before the initial INVITE transaction completes. Header fields present in a provisional response are applicable as long as the dialog is in the early state (for example, an Allow header field in a provisional response contains the methods that can be used in the dialog while this is in the early state).

13.2.2.2 3xx Responses

A 3xx response may contain one or more Contact header field values providing new addresses where the callee might be reachable. Depending on the status code of the 3xx response (see Section 21.3), the UAC MAY choose to try those new addresses.

13.2.2.3 4xx, 5xx and 6xx Responses

A single non-2xx final response may be received for the INVITE. 4xx, 5xx and 6xx responses may contain a Contact header field value indicating the location where additional information about the error can be found. Subsequent final responses (which would only arrive under error conditions) MUST be ignored.

All early dialogs are considered terminated upon reception of the non-2xx final response.

After having received the non-2xx final response the UAC core considers the INVITE transaction completed. The INVITE client transaction handles the generation of ACKs for the response (see Section 17).

13.2.2.4 2xx Responses

Multiple 2xx responses may arrive at the UAC for a single INVITE request due to a forking proxy. Each response is distinguished by the tag parameter in the To header field, and each represents a distinct dialog, with a distinct dialog identifier.

If the dialog identifier in the 2xx response matches the dialog identifier of an existing dialog, the dialog MUST be transitioned to the "confirmed" state, and the route set for the dialog MUST be recomputed based on the 2xx response using the procedures of Section 12.2.1.2. Otherwise, a new dialog in the "confirmed" state MUST be constructed using the procedures of Section 12.1.2.

> Note that the only piece of state that is recomputed is the route set. Other pieces of state such as the highest sequence numbers (remote and local) sent within the dialog are not recomputed. The route set only is recomputed for backwards compatibility. RFC 2543 did not mandate mirroring of the Record-Route header field in

a 1xx, only 2xx. However, we cannot update the entire state of the dialog, since mid-dialog requests may have been sent within the early dialog, modifying the sequence numbers, for example.

The UAC core MUST generate an ACK request for each 2xx received from the transaction layer. The header fields of the ACK are constructed in the same way as for any request sent within a dialog (see Section 12) with the exception of the CSeq and the header fields related to authentication. The sequence number of the CSeq header field MUST be the same as the INVITE being acknowledged, but the CSeq method MUST be ACK. The ACK MUST contain the same credentials as the INVITE. If the 2xx contains an offer (based on the rules above), the ACK MUST carry an answer in its body. If the offer in the 2xx response is not acceptable, the UAC core MUST generate a valid answer in the ACK and then send a BYE immediately.

Once the ACK has been constructed, the procedures of [4] are used to determine the destination address, port and transport. However, the request is passed to the transport layer directly for transmission, rather than a client transaction. This is because the UAC core handles retransmissions of the ACK, not the transaction layer. The ACK MUST be passed to the client transport every time a retransmission of the 2xx final response that triggered the ACK arrives.

The UAC core considers the INVITE transaction completed 64*T1 seconds after the reception of the first 2xx response. At this point all the early dialogs that have not transitioned to established dialogs are terminated. Once the INVITE transaction is considered completed by the UAC core, no more new 2xx responses are expected to arrive.

If, after acknowledging any 2xx response to an INVITE, the UAC does not want to continue with that dialog, then the UAC MUST terminate the dialog by sending a BYE request as described in Section 15.

13.3 UAS Processing

13.3.1 Processing of the INVITE

The UAS core will receive INVITE requests from the transaction layer. It first performs the request processing procedures of Section 8.2, which are applied for both requests inside and outside of a dialog.

Assuming these processing states are completed without generating a response, the UAS core performs the additional processing steps:

1. If the request is an INVITE that contains an Expires header field, the UAS core sets a timer for the number of seconds indicated in the header field value. When the timer fires, the invitation is considered to be expired. If the invitation expires before the UAS has generated a final response, a 487 (Request Terminated) response SHOULD be generated.

2. If the request is a mid-dialog request, the method-independent processing described in Section 12.2.2 is first applied. It might also modify the session; Section 14 provides details.

3. If the request has a tag in the To header field but the dialog identifier does not match any of the existing dialogs, the UAS may have crashed and restarted, or may have received a request for a different (possibly failed) UAS. Section 12.2.2 provides guidelines to achieve a robust behavior under such a situation.

Processing from here forward assumes that the INVITE is outside of a dialog, and is thus for the purposes of establishing a new session.

The INVITE may contain a session description, in which case the UAS is being presented with an offer for that session. It is possible that the user is already a participant in that session, even though the INVITE is outside of a dialog. This can happen when a user is invited to the same multicast conference by multiple other participants. If desired, the UAS MAY use identifiers within the session description to detect this duplication. For example, SDP

contains a session id and version number in the origin (o) field. If the user is already a member of the session, and the session parameters contained in the session description have not changed, the UAS MAY silently accept the INVITE (that is, send a 2xx response without prompting the user).

If the INVITE does not contain a session description, the UAS is being asked to participate in a session, and the UAC has asked that the UAS provide the offer of the session. It MUST provide the offer in its first non-failure reliable message back to the UAC. In this specification, that is a 2xx response to the INVITE.

The UAS can indicate progress, accept, redirect, or reject the invitation. In all of these cases, it formulates a response using the procedures described in Section 8.2.6.

13.3.1.1 Progress

If the UAS is not able to answer the invitation immediately, it can choose to indicate some kind of progress to the UAC (for example, an indication that a phone is ringing). This is accomplished with a provisional response between 101 and 199. These provisional responses establish early dialogs and therefore follow the procedures of Section 12.1.1 in addition to those of Section 8.2.6. A UAS MAY send as many provisional responses as it likes. Each of these MUST indicate the same dialog ID. However, these will not be delivered reliably.

If the UAS desires an extended period of time to answer the INVITE, it will need to ask for an "extension" in order to prevent proxies from canceling the transaction. A proxy has the option of canceling a transaction when there is a gap of 3 minutes between responses in a transaction. To prevent cancellation, the UAS MUST send a non-100 provisional response at every minute, to handle the possibility of lost provisional responses.

An INVITE transaction can go on for extended durations when the user is placed on hold, or when interworking with PSTN systems which allow communications to take place without answering the call. The latter is common in Interactive Voice Response (IVR)

systems.

13.3.1.2 The INVITE is Redirected

If the UAS decides to redirect the call, a 3xx response is sent. A 300 (Multiple Choices), 301 (Moved Permanently) or 302 (Moved Temporarily) response SHOULD contain a Contact header field

containing one or more URIs of new addresses to be tried. The response is passed to the INVITE server transaction, which will deal with its retransmissions.

13.3.1.3 The INVITE is Rejected

A common scenario occurs when the callee is currently not willing or able to take additional calls at this end system. A 486 (Busy Here) SHOULD be returned in such a scenario. If the UAS knows that no other end system will be able to accept this call, a 600 (Busy Everywhere) response SHOULD be sent instead. However, it is unlikely that a UAS will be able to know this in general, and thus this response will not usually be used. The response is passed to the INVITE server transaction, which will deal with its retransmissions.

A UAS rejecting an offer contained in an INVITE SHOULD return a 488 (Not Acceptable Here) response. Such a response SHOULD include a Warning header field value explaining why the offer was rejected.

13.3.1.4 The INVITE is Accepted

The UAS core generates a 2xx response. This response establishes a dialog, and therefore follows the procedures of Section 12.1.1 in addition to those of Section 8.2.6.

A 2xx response to an INVITE SHOULD contain the Allow header field and the Supported header field, and MAY contain the Accept header field. Including these header fields allows the UAC to determine the features and extensions supported by the UAS for the duration of the call, without probing.

If the INVITE request contained an offer, and the UAS had not yet sent an answer, the 2xx MUST contain an answer. If the INVITE did not contain an offer, the 2xx MUST contain an offer if the UAS had not yet sent an offer.

Once the response has been constructed, it is passed to the INVITE server transaction. Note, however, that the INVITE server transaction will be destroyed as soon as it receives this final response and passes it to the transport. Therefore, it is necessary to periodically pass the response directly to the transport until the ACK arrives. The 2xx response is passed to the transport with an interval that starts at T1 seconds and doubles for each retransmission until it reaches T2 seconds (T1 and T2 are defined in Section 17). Response retransmissions cease when an ACK request for the response is received. This is independent of whatever transport protocols are used to send the response.

Since 2xx is retransmitted end-to-end, there may be hops between

UAS and UAC that are UDP. To ensure reliable delivery across these hops, the response is retransmitted periodically even if the transport at the UAS is reliable.

If the server retransmits the 2xx response for 64*T1 seconds without receiving an ACK, the dialog is confirmed, but the session SHOULD be terminated. This is accomplished with a BYE, as described in Section 15.

14 Modifying an Existing Session

A successful INVITE request (see Section 13) establishes both a dialog between two user agents and a session using the offer-answer model. Section 12 explains how to modify an existing dialog using a target refresh request (for example, changing the remote target URI of the dialog). This section describes how to modify the actual session. This modification can involve changing addresses or ports, adding a media stream, deleting a media stream, and so on. This is accomplished by sending a new INVITE request within the same dialog that established the session. An INVITE request sent within an existing dialog is known as a re-INVITE.

Note that a single re-INVITE can modify the dialog and the parameters of the session at the same time.

Either the caller or callee can modify an existing session.

The behavior of a UA on detection of media failure is a matter of local policy. However, automated generation of re-INVITE or BYE is NOT RECOMMENDED to avoid flooding the network with traffic when there is congestion. In any case, if these messages are sent automatically, they SHOULD be sent after some randomized interval.

Note that the paragraph above refers to automatically generated BYEs and re-INVITEs. If the user hangs up upon media failure, the UA would send a BYE request as usual.

14.1 UAC Behavior

The same offer-answer model that applies to session descriptions in INVITEs (Section 13.2.1) applies to re-INVITEs. As a result, a UAC that wants to add a media stream, for example, will create a new offer that contains this media stream, and send that in an INVITE request to its peer. It is important to note that the full description of the session, not just the change, is sent. This supports stateless session processing in various elements, and supports failover and recovery capabilities. Of course, a UAC MAY

send a re-INVITE with no session description, in which case the first reliable non-failure response to the re-INVITE will contain the offer (in this specification, that is a 2xx response).

If the session description format has the capability for version numbers, the offerer SHOULD indicate that the version of the session description has changed.

The To, From, Call-ID, CSeq, and Request-URI of a re-INVITE are set

following the same rules as for regular requests within an existing dialog, described in Section 12.

A UAC MAY choose not to add an Alert-Info header field or a body with Content-Disposition "alert" to re-INVITEs because UASs do not typically alert the user upon reception of a re-INVITE.

Unlike an INVITE, which can fork, a re-INVITE will never fork, and therefore, only ever generate a single final response. The reason a re-INVITE will never fork is that the Request-URI identifies the target as the UA instance it established the dialog with, rather than identifying an address-of-record for the user.

Note that a UAC MUST NOT initiate a new INVITE transaction within a dialog while another INVITE transaction is in progress in either direction.

1. If there is an ongoing INVITE client transaction, the TU MUST wait until the transaction reaches the completed or terminated state before initiating the new INVITE.

2. If there is an ongoing INVITE server transaction, the TU MUST wait until the transaction reaches the confirmed or terminated state before initiating the new INVITE.

However, a UA MAY initiate a regular transaction while an INVITE transaction is in progress. A UA MAY also initiate an INVITE transaction while a regular transaction is in progress.

If a UA receives a non-2xx final response to a re-INVITE, the session parameters MUST remain unchanged, as if no re-INVITE had been issued. Note that, as stated in Section 12.2.1.2, if the non-2xx final response is a 481 (Call/Transaction Does Not Exist), or a 408 (Request Timeout), or no response at all is received for the re-INVITE (that is, a timeout is returned by the INVITE client transaction), the UAC will terminate the dialog.

If a UAC receives a 491 response to a re-INVITE, it SHOULD start a timer with a value T chosen as follows:

1. If the UAC is the owner of the Call-ID of the dialog ID (meaning it generated the value), T has a randomly chosen value between 2.1 and 4 seconds in units of 10 ms.

2. If the UAC is not the owner of the Call-ID of the dialog ID, T has a randomly chosen value of between 0 and 2 seconds in units of 10 ms.

When the timer fires, the UAC SHOULD attempt the re-INVITE once more, if it still desires for that session modification to take place. For example, if the call was already hung up with a BYE, the re-INVITE would not take place.

The rules for transmitting a re-INVITE and for generating an ACK for a 2xx response to re-INVITE are the same as for the initial INVITE (Section 13.2.1).

14.2 UAS Behavior

Section 13.3.1 describes the procedure for distinguishing incoming re-INVITEs from incoming initial INVITEs and handling a re-INVITE for an existing dialog.

A UAS that receives a second INVITE before it sends the final response to a first INVITE with a lower CSeq sequence number on the same dialog MUST return a 500 (Server Internal Error) response to the second INVITE and MUST include a Retry-After header field with a randomly chosen value of between 0 and 10 seconds.

A UAS that receives an INVITE on a dialog while an INVITE it had sent on that dialog is in progress MUST return a 491 (Request Pending) response to the received INVITE.

If a UA receives a re-INVITE for an existing dialog, it MUST check any version identifiers in the session description or, if there are no version identifiers, the content of the session description to see if it has changed. If the session description has changed, the UAS MUST adjust the session parameters accordingly, possibly after asking the user for confirmation.

> Versioning of the session description can be used to accommodate the capabilities of new arrivals to a conference, add or delete media, or change from a unicast to a multicast conference.

If the new session description is not acceptable, the UAS can reject it by returning a 488 (Not Acceptable Here) response for the re-INVITE. This response SHOULD include a Warning header field.

If a UAS generates a 2xx response and never receives an ACK, it SHOULD generate a BYE to terminate the dialog.

A UAS MAY choose not to generate 180 (Ringing) responses for a re-INVITE because UACs do not typically render this information to the user. For the same reason, UASs MAY choose not to use an Alert-Info header field or a body with Content-Disposition "alert" in responses to a re-INVITE.

A UAS providing an offer in a 2xx (because the INVITE did not contain an offer) SHOULD construct the offer as if the UAS were making a brand new call, subject to the constraints of sending an offer that updates an existing session, as described in [13] in the case of SDP. Specifically, this means that it SHOULD include as many media formats and media types that the UA is willing to support. The UAS MUST ensure that the session description overlaps with its previous session description in media formats, transports, or other parameters that require support from the peer. This is to avoid the need for the peer to reject the session description. If, however, it is unacceptable to the UAC, the UAC SHOULD generate an answer with a valid session description, and then send a BYE to terminate the session.

15 Terminating a Session

This section describes the procedures for terminating a session

established by SIP. The state of the session and the state of the dialog are very closely related. When a session is initiated with an INVITE, each 1xx or 2xx response from a distinct UAS creates a dialog, and if that response completes the offer/answer exchange, it also creates a session. As a result, each session is "associated" with a single dialog - the one which resulted in its creation. If an initial INVITE generates a non-2xx final response, that terminates all sessions (if any) and all dialogs (if any) that were created through responses to the request. By virtue of completing the transaction, a non-2xx final response also prevents further sessions from being created as a result of the INVITE. The BYE request is used to terminate a specific session or attempted session. In this case, the specific session is the one with the peer UA on the other side of the dialog. When a BYE is received on a dialog, any session associated with that dialog SHOULD terminate. A UA MUST NOT send a BYE outside of a dialog. The caller's UA MAY send a BYE for either confirmed or early dialogs, and the callee's UA MAY send a BYE on confirmed dialogs, but MUST NOT send a BYE on early dialogs.

However, the callee's UA MUST NOT send a BYE on a confirmed dialog until it has received an ACK for its 2xx response or until the server transaction times out. If no SIP extensions have defined other application layer states associated with the dialog, the BYE also terminates the dialog.

The impact of a non-2xx final response to INVITE on dialogs and sessions makes the use of CANCEL attractive. The CANCEL attempts to force a non-2xx response to the INVITE (in particular, a 487). Therefore, if a UAC wishes to give up on its call attempt entirely, it can send a CANCEL. If the INVITE results in 2xx final response(s) to the INVITE, this means that a UAS accepted the invitation while the CANCEL was in progress. The UAC MAY continue with the sessions established by any 2xx responses, or MAY terminate them with BYE.

> The notion of "hanging up" is not well defined within SIP. It is specific to a particular, albeit common, user interface. Typically, when the user hangs up, it indicates a desire to terminate the attempt to establish a session, and to terminate any sessions already created. For the caller's UA, this would imply a CANCEL request if the initial INVITE has not generated a final response, and a BYE to all confirmed dialogs after a final response. For the callee's UA, it would typically imply a BYE; presumably, when the user picked up the phone, a 2xx was generated, and so hanging up would result in a BYE after the ACK is received. This does not mean a user cannot hang up before receipt of the ACK, it just means that the software in his phone needs to maintain state for a short while in order to clean up properly. If the particular UI allows for the user to reject a call before its answered, a 403 (Forbidden) is a good way to express that. As per the rules above, a BYE can't be sent.

15.1 Terminating a Session with a BYE Request

15.1.1 UAC Behavior

A BYE request is constructed as would any other request within a dialog, as described in Section 12.

Once the BYE is constructed, the UAC core creates a new non-INVITE client transaction, and passes it the BYE request. The UAC MUST consider the session terminated (and therefore stop sending or listening for media) as soon as the BYE request is passed to the client transaction. If the response for the BYE is a 481 (Call/Transaction Does Not Exist) or a 408 (Request Timeout) or no

response at all is received for the BYE (that is, a timeout is returned by the client transaction), the UAC MUST consider the session and the dialog terminated.

15.1.2 UAS Behavior

A UAS first processes the BYE request according to the general UAS processing described in Section 8.2. A UAS core receiving a BYE request checks if it matches an existing dialog. If the BYE does not match an existing dialog, the UAS core SHOULD generate a 481 (Call/Transaction Does Not Exist) response and pass that to the server transaction.

> This rule means that a BYE sent without tags by a UAC will be rejected. This is a change from RFC 2543, which allowed BYE without tags.

A UAS core receiving a BYE request for an existing dialog MUST follow the procedures of Section 12.2.2 to process the request. Once done, the UAS SHOULD terminate the session (and therefore stop sending and listening for media). The only case where it can elect not to are multicast sessions, where participation is possible even if the other participant in the dialog has terminated its involvement in the session. Whether or not it ends its participation on the session, the UAS core MUST generate a 2xx response to the BYE, and MUST pass that to the server transaction for transmission.

The UAS MUST still respond to any pending requests received for that dialog. It is RECOMMENDED that a 487 (Request Terminated) response be generated to those pending requests.

16 Proxy Behavior

16.1 Overview

SIP proxies are elements that route SIP requests to user agent servers and SIP responses to user agent clients. A request may traverse several proxies on its way to a UAS. Each will make routing decisions, modifying the request before forwarding it to the next element. Responses will route through the same set of proxies traversed by the request in the reverse order.

Being a proxy is a logical role for a SIP element. When a request arrives, an element that can play the role of a proxy first decides if it needs to respond to the request on its own. For instance, the request may be malformed or the element may need credentials from the client before acting as a proxy. The element MAY respond with any

appropriate error code. When responding directly to a request, the

element is playing the role of a UAS and MUST behave as described in Section 8.2.

A proxy can operate in either a stateful or stateless mode for each new request. When stateless, a proxy acts as a simple forwarding element. It forwards each request downstream to a single element determined by making a targeting and routing decision based on the request. It simply forwards every response it receives upstream. A stateless proxy discards information about a message once the message has been forwarded. A stateful proxy remembers information (specifically, transaction state) about each incoming request and any requests it sends as a result of processing the incoming request. It uses this information to affect the processing of future messages associated with that request. A stateful proxy MAY choose to "fork" a request, routing it to multiple destinations. Any request that is forwarded to more than one location MUST be handled statefully.

In some circumstances, a proxy MAY forward requests using stateful transports (such as TCP) without being transaction-stateful. For instance, a proxy MAY forward a request from one TCP connection to another transaction statelessly as long as it places enough information in the message to be able to forward the response down the same connection the request arrived on. Requests forwarded between different types of transports where the proxy's TU must take an active role in ensuring reliable delivery on one of the transports MUST be forwarded transaction statefully.

A stateful proxy MAY transition to stateless operation at any time during the processing of a request, so long as it did not do anything that would otherwise prevent it from being stateless initially (forking, for example, or generation of a 100 response). When performing such a transition, all state is simply discarded. The proxy SHOULD NOT initiate a CANCEL request.

Much of the processing involved when acting statelessly or statefully for a request is identical. The next several subsections are written from the point of view of a stateful proxy. The last section calls out those places where a stateless proxy behaves differently.

16.2 Stateful Proxy

When stateful, a proxy is purely a SIP transaction processing engine. Its behavior is modeled here in terms of the server and client transactions defined in Section 17. A stateful proxy has a server transaction associated with one or more client transactions by a higher layer proxy processing component (see figure 3), known as a proxy core. An incoming request is processed by a server

transaction. Requests from the server transaction are passed to a proxy core. The proxy core determines where to route the request, choosing one or more next-hop locations. An outgoing request for each next-hop location is processed by its own associated client transaction. The proxy core collects the responses from the client transactions and uses them to send responses to the server transaction.

A stateful proxy creates a new server transaction for each new

request received. Any retransmissions of the request will then be handled by that server transaction per Section 17. The proxy core MUST behave as a UAS with respect to sending an immediate provisional on that server transaction (such as 100 Trying) as described in Section 8.2.6. Thus, a stateful proxy SHOULD NOT generate 100 (Trying) responses to non-INVITE requests.

This is a model of proxy behavior, not of software. An implementation is free to take any approach that replicates the external behavior this model defines.

For all new requests, including any with unknown methods, an element intending to proxy the request MUST:

1. Validate the request (Section 16.3)

2. Preprocess routing information (Section 16.4)

3. Determine target(s) for the request (Section 16.5)

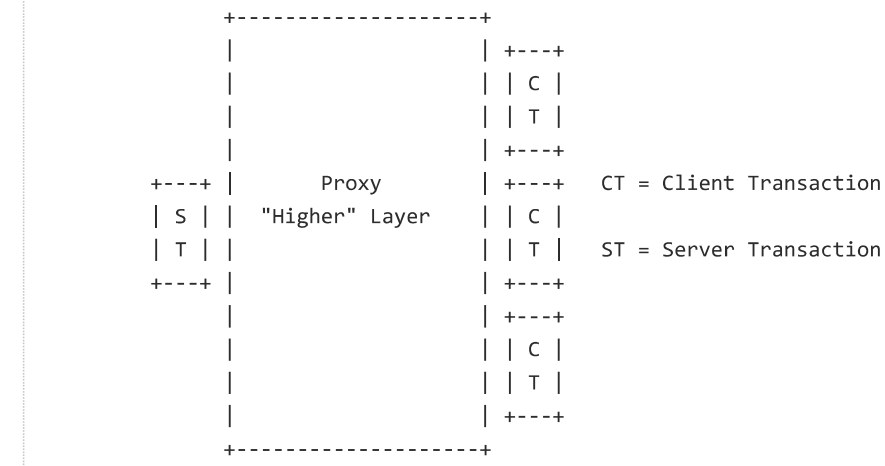


Figure 3: Stateful Proxy Model

4. Forward the request to each target (Section 16.6)

5. Process all responses (Section 16.7)

16.3 Request Validation

Before an element can proxy a request, it MUST verify the message's validity. A valid message must pass the following checks:

1. Reasonable Syntax

2. URI scheme

3. Max-Forwards

4. (Optional) Loop Detection

5. Proxy-Require

6. Proxy-Authorization

If any of these checks fail, the element MUST behave as a user agent server (see Section 8.2) and respond with an error code.

Notice that a proxy is not required to detect merged requests and MUST NOT treat merged requests as an error condition. The endpoints receiving the requests will resolve the merge as described in Section 8.2.2.2.

1. Reasonable syntax check

   The request MUST be well-formed enough to be handled with a server transaction. Any components involved in the remainder of these Request Validation steps or the Request Forwarding section MUST be well-formed. Any other components, well-formed or not, SHOULD be ignored and remain unchanged when the message is forwarded. For instance, an element would not reject a request because of a malformed Date header field. Likewise, a proxy would not remove a malformed Date header field before forwarding a request.

   This protocol is designed to be extended. Future extensions may define new methods and header fields at any time. An element MUST NOT refuse to proxy a request because it contains a method or header field it does not know about.

2. URI scheme check

   If the Request-URI has a URI whose scheme is not understood by the proxy, the proxy SHOULD reject the request with a 416 (Unsupported URI Scheme) response.

3. Max-Forwards check

   The Max-Forwards header field (Section 20.22) is used to limit the number of elements a SIP request can traverse.

   If the request does not contain a Max-Forwards header field, this check is passed.

   If the request contains a Max-Forwards header field with a field value greater than zero, the check is passed.

   If the request contains a Max-Forwards header field with a field value of zero (0), the element MUST NOT forward the request. If the request was for OPTIONS, the element MAY act as the final recipient and respond per Section 11. Otherwise, the element MUST return a 483 (Too many hops) response.

4. Optional Loop Detection check

   An element MAY check for forwarding loops before forwarding a request. If the request contains a Via header field with a sent-by value that equals a value placed into previous requests by the proxy, the request has been forwarded by this element before. The request has either looped or is legitimately spiraling through the element. To determine if the request has looped, the element MAY perform the branch parameter calculation described in Step 8 of

Section 16.6 on this message and compare it to the parameter received in that Via header field. If the parameters match, the request has looped. If they differ, the request is spiraling, and processing continues. If a loop is detected, the element MAY return a 482 (Loop Detected) response.

5. Proxy-Require check

Future extensions to this protocol may introduce features that require special handling by proxies. Endpoints will include a Proxy-Require header field in requests that use these features, telling the proxy not to process the request unless the feature is understood.

If the request contains a Proxy-Require header field (Section 20.29) with one or more option-tags this element does not understand, the element MUST return a 420 (Bad Extension) response. The response MUST include an Unsupported (Section 20.40) header field listing those option-tags the element did not understand.

6. Proxy-Authorization check

If an element requires credentials before forwarding a request, the request MUST be inspected as described in Section 22.3. That section also defines what the element must do if the inspection fails.

16.4 Route Information Preprocessing

The proxy MUST inspect the Request-URI of the request. If the Request-URI of the request contains a value this proxy previously placed into a Record-Route header field (see Section 16.6 item 4), the proxy MUST replace the Request-URI in the request with the last value from the Route header field, and remove that value from the Route header field. The proxy MUST then proceed as if it received this modified request.

This will only happen when the element sending the request to the proxy (which may have been an endpoint) is a strict router. This rewrite on receive is necessary to enable backwards compatibility with those elements. It also allows elements following this specification to preserve the Request-URI through strict-routing proxies (see Section 12.2.1.1).

This requirement does not obligate a proxy to keep state in order to detect URIs it previously placed in Record-Route header fields. Instead, a proxy need only place enough information in those URIs to recognize them as values it provided when they later appear.

If the Request-URI contains a maddr parameter, the proxy MUST check to see if its value is in the set of addresses or domains the proxy is configured to be responsible for. If the Request-URI has a maddr parameter with a value the proxy is responsible for, and the request was received using the port and transport indicated (explicitly or by default) in the Request-URI, the proxy MUST strip the maddr and any non-default port or transport parameter and continue processing as if

those values had not been present in the request.

A request may arrive with a maddr matching the proxy, but on a port or transport different from that indicated in the URI. Such a request needs to be forwarded to the proxy using the indicated port and transport.

If the first value in the Route header field indicates this proxy, the proxy MUST remove that value from the request.

16.5 Determining Request Targets

Next, the proxy calculates the target(s) of the request. The set of targets will either be predetermined by the contents of the request or will be obtained from an abstract location service. Each target in the set is represented as a URI.

If the Request-URI of the request contains an maddr parameter, the Request-URI MUST be placed into the target set as the only target URI, and the proxy MUST proceed to Section 16.6.

If the domain of the Request-URI indicates a domain this element is not responsible for, the Request-URI MUST be placed into the target set as the only target, and the element MUST proceed to the task of Request Forwarding (Section 16.6).

There are many circumstances in which a proxy might receive a request for a domain it is not responsible for. A firewall proxy handling outgoing calls (the way HTTP proxies handle outgoing requests) is an example of where this is likely to occur.

If the target set for the request has not been predetermined as described above, this implies that the element is responsible for the domain in the Request-URI, and the element MAY use whatever mechanism it desires to determine where to send the request. Any of these mechanisms can be modeled as accessing an abstract Location Service. This may consist of obtaining information from a location service created by a SIP Registrar, reading a database, consulting a presence server, utilizing other protocols, or simply performing an algorithmic substitution on the Request-URI. When accessing the location service constructed by a registrar, the Request-URI MUST first be canonicalized as described in Section 10.3 before being used as an index. The output of these mechanisms is used to construct the target set.

If the Request-URI does not provide sufficient information for the proxy to determine the target set, it SHOULD return a 485 (Ambiguous) response. This response SHOULD contain a Contact header field containing URIs of new addresses to be tried. For example, an INVITE

to sip:John.Smith@company.com may be ambiguous at a proxy whose location service has multiple John Smiths listed. See Section 21.4.23 for details.

Any information in or about the request or the current environment of the element MAY be used in the construction of the target set. For instance, different sets may be constructed depending on contents or

the presence of header fields and bodies, the time of day of the request's arrival, the interface on which the request arrived, failure of previous requests, or even the element's current level of utilization.

As potential targets are located through these services, their URIs are added to the target set. Targets can only be placed in the target set once. If a target URI is already present in the set (based on the definition of equality for the URI type), it MUST NOT be added again.

A proxy MUST NOT add additional targets to the target set if the Request-URI of the original request does not indicate a resource this proxy is responsible for.

> A proxy can only change the Request-URI of a request during forwarding if it is responsible for that URI. If the proxy is not responsible for that URI, it will not recurse on 3xx or 416 responses as described below.

If the Request-URI of the original request indicates a resource this proxy is responsible for, the proxy MAY continue to add targets to the set after beginning Request Forwarding. It MAY use any information obtained during that processing to determine new targets. For instance, a proxy may choose to incorporate contacts obtained in a redirect response (3xx) into the target set. If a proxy uses a dynamic source of information while building the target set (for instance, if it consults a SIP Registrar), it SHOULD monitor that source for the duration of processing the request. New locations SHOULD be added to the target set as they become available. As above, any given URI MUST NOT be added to the set more than once.

> Allowing a URI to be added to the set only once reduces unnecessary network traffic, and in the case of incorporating contacts from redirect requests prevents infinite recursion.

For example, a trivial location service is a "no-op", where the target URI is equal to the incoming request URI. The request is sent to a specific next hop proxy for further processing. During request

forwarding of Section 16.6, Item 6, the identity of that next hop, expressed as a SIP or SIPS URI, is inserted as the top-most Route header field value into the request.

If the Request-URI indicates a resource at this proxy that does not exist, the proxy MUST return a 404 (Not Found) response.

If the target set remains empty after applying all of the above, the proxy MUST return an error response, which SHOULD be the 480 (Temporarily Unavailable) response.

16.6 Request Forwarding

As soon as the target set is non-empty, a proxy MAY begin forwarding the request. A stateful proxy MAY process the set in any order. It MAY process multiple targets serially, allowing each client transaction to complete before starting the next. It MAY start

client transactions with every target in parallel. It also MAY arbitrarily divide the set into groups, processing the groups serially and processing the targets in each group in parallel.

A common ordering mechanism is to use the qvalue parameter of targets obtained from Contact header fields (see Section 20.10). Targets are processed from highest qvalue to lowest. Targets with equal qvalues may be processed in parallel.

A stateful proxy must have a mechanism to maintain the target set as responses are received and associate the responses to each forwarded request with the original request. For the purposes of this model, this mechanism is a "response context" created by the proxy layer before forwarding the first request.

For each target, the proxy forwards the request following these steps:

1. Make a copy of the received request
2. Update the Request-URI
3. Update the Max-Forwards header field
4. Optionally add a Record-route header field value
5. Optionally add additional header fields
6. Postprocess routing information
7. Determine the next-hop address, port, and transport
8. Add a Via header field value
9. Add a Content-Length header field if necessary
10. Forward the new request
11. Set timer C

Each of these steps is detailed below:

1. Copy request

   The proxy starts with a copy of the received request. The copy MUST initially contain all of the header fields from the received request. Fields not detailed in the processing described below MUST NOT be removed. The copy SHOULD maintain the ordering of the header fields as in the received request. The proxy MUST NOT reorder field values with a common field name (See Section 7.3.1). The proxy MUST NOT add to, modify, or remove the message body.

   An actual implementation need not perform a copy; the primary requirement is that the processing for each next hop begin with the same request.

2. Request-URI

   The Request-URI in the copy's start line MUST be replaced with the URI for this target. If the URI contains any parameters not allowed in a Request-URI, they MUST be removed.

   This is the essence of a proxy's role. This is the mechanism through which a proxy routes a request toward its destination.

   In some circumstances, the received Request-URI is placed into the target set without being modified. For that target, the replacement above is effectively a no-op.

3. Max-Forwards

   If the copy contains a Max-Forwards header field, the proxy MUST decrement its value by one (1).

   If the copy does not contain a Max-Forwards header field, the proxy MUST add one with a field value, which SHOULD be 70.

   Some existing UAs will not provide a Max-Forwards header field in a request.

4. Record-Route

   If this proxy wishes to remain on the path of future requests in a dialog created by this request (assuming the request creates a dialog), it MUST insert a Record-Route header field value into the copy before any existing Record-Route header field values, even if a Route header field is already present.

   Requests establishing a dialog may contain a preloaded Route header field.

   If this request is already part of a dialog, the proxy SHOULD insert a Record-Route header field value if it wishes to remain on the path of future requests in the dialog. In normal endpoint operation as described in Section 12, these Record-Route header field values will not have any effect on the route sets used by the endpoints.

   The proxy will remain on the path if it chooses to not insert a Record-Route header field value into requests that are already part of a dialog. However, it would be removed from the path when an endpoint that has failed reconstitutes the dialog.

   A proxy MAY insert a Record-Route header field value into any request. If the request does not initiate a dialog, the endpoints will ignore the value. See Section 12 for details on how endpoints use the Record-Route header field values to construct Route header fields.

   Each proxy in the path of a request chooses whether to add a Record-Route header field value independently - the presence of a Record-Route header field in a request does not obligate this proxy to add a value.

The URI placed in the Record-Route header field value MUST be a SIP or SIPS URI. This URI MUST contain an lr parameter (see Section 19.1.1). This URI MAY be different for each destination the request is forwarded to. The URI SHOULD NOT contain the transport parameter unless the proxy has knowledge (such as in a private network) that the next downstream element that will be in the path of subsequent requests supports that transport.

The URI this proxy provides will be used by some other element to make a routing decision. This proxy, in general, has no way of knowing the capabilities of that element, so it must restrict itself to the mandatory elements of a SIP implementation: SIP URIs and either the TCP or UDP transports.

The URI placed in the Record-Route header field MUST resolve to the element inserting it (or a suitable stand-in) when the server location procedures of [4] are applied to it, so that subsequent requests reach the same SIP element. If the Request-URI contains a SIPS URI, or the topmost Route header field value (after the post processing of bullet 6) contains a SIPS URI, the URI placed into the Record-Route header field MUST be a SIPS URI. Furthermore, if the request was not received over TLS, the proxy MUST insert a Record-Route header field. In a similar fashion, a proxy that receives a request over TLS, but generates a request without a SIPS URI in the Request-URI or topmost Route header field value (after the post processing of bullet 6), MUST insert a Record-Route header field that is not a SIPS URI.

A proxy at a security perimeter must remain on the perimeter throughout the dialog.

If the URI placed in the Record-Route header field needs to be rewritten when it passes back through in a response, the URI MUST be distinct enough to locate at that time. (The request may spiral through this proxy, resulting in more than one Record-Route header field value being added). Item 8 of Section 16.7 recommends a mechanism to make the URI sufficiently distinct.

The proxy MAY include parameters in the Record-Route header field value. These will be echoed in some responses to the request such as the 200 (OK) responses to INVITE. Such parameters may be useful for keeping state in the message rather than the proxy.

If a proxy needs to be in the path of any type of dialog (such as one straddling a firewall), it SHOULD add a Record-Route header field value to every request with a method it does not understand since that method may have dialog semantics.

The URI a proxy places into a Record-Route header field is only valid for the lifetime of any dialog created by the transaction in which it occurs. A dialog-stateful proxy, for example, MAY refuse to accept future requests with that value in the

Request-URI after the dialog has terminated. Non-dialog-stateful proxies, of course, have no concept of when the dialog has terminated, but they MAY encode enough information in the value to compare it against the dialog identifier of future requests and MAY reject requests not matching that information. Endpoints MUST NOT use a URI obtained from a Record-Route header field outside the dialog in which it was provided. See

Section 12 for more information on an endpoint's use of Record-Route header fields.

Record-routing may be required by certain services where the proxy needs to observe all messages in a dialog. However, it slows down processing and impairs scalability and thus proxies should only record-route if required for a particular service.

The Record-Route process is designed to work for any SIP request that initiates a dialog. INVITE is the only such request in this specification, but extensions to the protocol MAY define others.

5. Add Additional Header Fields

The proxy MAY add any other appropriate header fields to the copy at this point.

6. Postprocess routing information

A proxy MAY have a local policy that mandates that a request visit a specific set of proxies before being delivered to the destination. A proxy MUST ensure that all such proxies are loose routers. Generally, this can only be known with certainty if the proxies are within the same administrative domain. This set of proxies is represented by a set of URIs (each of which contains the lr parameter). This set MUST be pushed into the Route header field of the copy ahead of any existing values, if present. If the Route header field is absent, it MUST be added, containing that list of URIs.

If the proxy has a local policy that mandates that the request visit one specific proxy, an alternative to pushing a Route value into the Route header field is to bypass the forwarding logic of item 10 below, and instead just send the request to the address, port, and transport for that specific proxy. If the request has a Route header field, this alternative MUST NOT be used unless it is known that next hop proxy is a loose router. Otherwise, this approach MAY be used, but the Route insertion mechanism above is preferred for its robustness, flexibility, generality and consistency of operation. Furthermore, if the Request-URI contains a SIPS URI, TLS MUST be used to communicate with that proxy.

If the copy contains a Route header field, the proxy MUST inspect the URI in its first value. If that URI does not contain an lr parameter, the proxy MUST modify the copy as follows:

- The proxy MUST place the Request-URI into the Route header field as the last value.

- The proxy MUST then place the first Route header field value into the Request-URI and remove that value from the Route header field.

Appending the Request-URI to the Route header field is part of a mechanism used to pass the information in that Request-URI through strict-routing elements. "Popping" the first Route header field value into the Request-URI formats the message the way a strict-routing element expects to receive it (with its own URI in the Request-URI and the next location to visit in the first Route header field value).

7. Determine Next-Hop Address, Port, and Transport

The proxy MAY have a local policy to send the request to a specific IP address, port, and transport, independent of the values of the Route and Request-URI. Such a policy MUST NOT be used if the proxy is not certain that the IP address, port, and transport correspond to a server that is a loose router. However, this mechanism for sending the request through a specific next hop is NOT RECOMMENDED; instead a Route header field should be used for that purpose as described above.

In the absence of such an overriding mechanism, the proxy applies the procedures listed in [4] as follows to determine where to send the request. If the proxy has reformatted the request to send to a strict-routing element as described in step 6 above, the proxy MUST apply those procedures to the Request-URI of the request. Otherwise, the proxy MUST apply the procedures to the first value in the Route header field, if present, else the Request-URI. The procedures will produce an ordered set of (address, port, transport) tuples. Independently of which URI is being used as input to the procedures of [4], if the Request-URI specifies a SIPS resource, the proxy MUST follow the procedures of [4] as if the input URI were a SIPS URI.

As described in [4], the proxy MUST attempt to deliver the message to the first tuple in that set, and proceed through the set in order until the delivery attempt succeeds.

For each tuple attempted, the proxy MUST format the message as appropriate for the tuple and send the request using a new client transaction as detailed in steps 8 through 10.

Since each attempt uses a new client transaction, it represents a new branch. Thus, the branch parameter provided with the Via header field inserted in step 8 MUST be different for each attempt.

If the client transaction reports failure to send the request or a timeout from its state machine, the proxy continues to the next address in that ordered set. If the ordered set is exhausted, the request cannot be forwarded to this element in

the target set. The proxy does not need to place anything in the response context, but otherwise acts as if this element of the target set returned a 408 (Request Timeout) final response.

8. Add a Via header field value

   The proxy MUST insert a Via header field value into the copy before the existing Via header field values. The construction of this value follows the same guidelines of Section 8.1.1.7. This implies that the proxy will compute its own branch parameter, which will be globally unique for that branch, and contain the requisite magic cookie. Note that this implies that the branch parameter will be different for different instances of a spiraled or looped request through a proxy.

   Proxies choosing to detect loops have an additional constraint in the value they use for construction of the branch parameter. A proxy choosing to detect loops SHOULD create a branch parameter separable into two parts by the implementation. The first part MUST satisfy the constraints of Section 8.1.1.7 as described above. The second is used to perform loop detection and distinguish loops from spirals.

   Loop detection is performed by verifying that, when a request returns to a proxy, those fields having an impact on the processing of the request have not changed. The value placed in this part of the branch parameter SHOULD reflect all of those fields (including any Route, Proxy-Require and Proxy-Authorization header fields). This is to ensure that if the request is routed back to the proxy and one of those fields changes, it is treated as a spiral and not a loop (see Section 16.3). A common way to create this value is to compute a cryptographic hash of the To tag, From tag, Call-ID header field, the Request-URI of the request received (before translation), the topmost Via header, and the sequence number from the CSeq header field, in addition to any Proxy-Require and Proxy-Authorization header fields that may be present. The

   algorithm used to compute the hash is implementation-dependent, but MD5 (RFC 1321 [35]), expressed in hexadecimal, is a reasonable choice. (Base64 is not permissible for a token.)

   If a proxy wishes to detect loops, the "branch" parameter it supplies MUST depend on all information affecting processing of a request, including the incoming Request-URI and any header fields affecting the request's admission or routing. This is necessary to distinguish looped requests from requests whose routing parameters have changed before returning to this server.

   The request method MUST NOT be included in the calculation of the branch parameter. In particular, CANCEL and ACK requests (for non-2xx responses) MUST have the same branch value as the corresponding request they cancel or acknowledge. The branch parameter is used in correlating those requests at the server handling them (see Sections 17.2.3 and 9.2).

9. Add a Content-Length header field if necessary

   If the request will be sent to the next hop using a stream-based transport and the copy contains no Content-Length header field, the proxy MUST insert one with the correct value for the body of the request (see Section 20.14).

10. Forward Request

   A stateful proxy MUST create a new client transaction for this request as described in Section 17.1 and instructs the transaction to send the request using the address, port and transport determined in step 7.

11. Set timer C

   In order to handle the case where an INVITE request never generates a final response, the TU uses a timer which is called timer C. Timer C MUST be set for each client transaction when an INVITE request is proxied. The timer MUST be larger than 3 minutes. Section 16.7 bullet 2 discusses how this timer is updated with provisional responses, and Section 16.8 discusses processing when it fires.

16.7 Response Processing

When a response is received by an element, it first tries to locate a client transaction (Section 17.1.3) matching the response. If none is found, the element MUST process the response (even if it is an informational response) as a stateless proxy (described below). If a match is found, the response is handed to the client transaction.

   Forwarding responses for which a client transaction (or more generally any knowledge of having sent an associated request) is not found improves robustness. In particular, it ensures that "late" 2xx responses to INVITE requests are forwarded properly.

As client transactions pass responses to the proxy layer, the following processing MUST take place:

1. Find the appropriate response context

2. Update timer C for provisional responses

3. Remove the topmost Via

4. Add the response to the response context

5. Check to see if this response should be forwarded immediately

6. When necessary, choose the best final response from the response context

If no final response has been forwarded after every client transaction associated with the response context has been terminated, the proxy must choose and forward the "best" response from those it has seen so far.

The following processing MUST be performed on each response that is forwarded. It is likely that more than one response to each request will be forwarded: at least each provisional and one final response.

7. Aggregate authorization header field values if necessary

8. Optionally rewrite Record-Route header field values

9. Forward the response

10. Generate any necessary CANCEL requests

Each of the above steps are detailed below:

1. Find Context

   The proxy locates the "response context" it created before forwarding the original request using the key described in Section 16.6. The remaining processing steps take place in this context.

2. Update timer C for provisional responses

   For an INVITE transaction, if the response is a provisional response with status codes 101 to 199 inclusive (i.e., anything but 100), the proxy MUST reset timer C for that client transaction. The timer MAY be reset to a different value, but this value MUST be greater than 3 minutes.

3. Via

   The proxy removes the topmost Via header field value from the response.

   If no Via header field values remain in the response, the response was meant for this element and MUST NOT be forwarded. The remainder of the processing described in this section is not performed on this message, the UAC processing rules described in Section 8.1.3 are followed instead (transport layer processing has already occurred).

   This will happen, for instance, when the element generates CANCEL requests as described in Section 10.

4. Add response to context

   Final responses received are stored in the response context until a final response is generated on the server transaction associated with this context. The response may be a candidate for the best final response to be returned on that server transaction. Information from this response may be needed in forming the best response, even if this response is not chosen.

   If the proxy chooses to recurse on any contacts in a 3xx response by adding them to the target set, it MUST remove them from the response before adding the response to the response

context. However, a proxy SHOULD NOT recurse to a non-SIPS URI if the Request-URI of the original request was a SIPS URI. If

the proxy recurses on all of the contacts in a 3xx response, the proxy SHOULD NOT add the resulting contactless response to the response context.

Removing the contact before adding the response to the response context prevents the next element upstream from retrying a location this proxy has already attempted.

3xx responses may contain a mixture of SIP, SIPS, and non-SIP URIs. A proxy may choose to recurse on the SIP and SIPS URIs and place the remainder into the response context to be returned, potentially in the final response.

If a proxy receives a 416 (Unsupported URI Scheme) response to a request whose Request-URI scheme was not SIP, but the scheme in the original received request was SIP or SIPS (that is, the proxy changed the scheme from SIP or SIPS to something else when it proxied a request), the proxy SHOULD add a new URI to the target set. This URI SHOULD be a SIP URI version of the non-SIP URI that was just tried. In the case of the tel URL, this is accomplished by placing the telephone-subscriber part of the tel URL into the user part of the SIP URI, and setting the hostpart to the domain where the prior request was sent. See Section 19.1.6 for more detail on forming SIP URIs from tel URLs.

As with a 3xx response, if a proxy "recurses" on the 416 by trying a SIP or SIPS URI instead, the 416 response SHOULD NOT be added to the response context.

5. Check response for forwarding

Until a final response has been sent on the server transaction, the following responses MUST be forwarded immediately:

- Any provisional response other than 100 (Trying)

- Any 2xx response

If a 6xx response is received, it is not immediately forwarded, but the stateful proxy SHOULD cancel all client pending transactions as described in Section 10, and it MUST NOT create any new branches in this context.

This is a change from RFC 2543, which mandated that the proxy was to forward the 6xx response immediately. For an INVITE transaction, this approach had the problem that a 2xx response could arrive on another branch, in which case the proxy would

have to forward the 2xx. The result was that the UAC could receive a 6xx response followed by a 2xx response, which should never be allowed to happen. Under the new rules, upon receiving a 6xx, a proxy will issue a CANCEL request, which will generally result in 487 responses from all outstanding

client transactions, and then at that point the 6xx is forwarded upstream.

After a final response has been sent on the server transaction, the following responses MUST be forwarded immediately:

- Any 2xx response to an INVITE request

A stateful proxy MUST NOT immediately forward any other responses. In particular, a stateful proxy MUST NOT forward any 100 (Trying) response. Those responses that are candidates for forwarding later as the "best" response have been gathered as described in step "Add Response to Context".

Any response chosen for immediate forwarding MUST be processed as described in steps "Aggregate Authorization Header Field Values" through "Record-Route".

This step, combined with the next, ensures that a stateful proxy will forward exactly one final response to a non-INVITE request, and either exactly one non-2xx response or one or more 2xx responses to an INVITE request.

6. Choosing the best response

A stateful proxy MUST send a final response to a response context's server transaction if no final responses have been immediately forwarded by the above rules and all client transactions in this response context have been terminated.

The stateful proxy MUST choose the "best" final response among those received and stored in the response context.

If there are no final responses in the context, the proxy MUST send a 408 (Request Timeout) response to the server transaction.

Otherwise, the proxy MUST forward a response from the responses stored in the response context. It MUST choose from the 6xx class responses if any exist in the context. If no 6xx class responses are present, the proxy SHOULD choose from the lowest response class stored in the response context. The proxy MAY select any response within that chosen class. The proxy SHOULD

give preference to responses that provide information affecting resubmission of this request, such as 401, 407, 415, 420, and 484 if the 4xx class is chosen.

A proxy which receives a 503 (Service Unavailable) response SHOULD NOT forward it upstream unless it can determine that any subsequent requests it might proxy will also generate a 503. In other words, forwarding a 503 means that the proxy knows it cannot service any requests, not just the one for the Request-URI in the request which generated the 503. If the only response that was received is a 503, the proxy SHOULD generate a 500 response and forward that upstream.

The forwarded response MUST be processed as described in steps "Aggregate Authorization Header Field Values" through "Record-Route".

For example, if a proxy forwarded a request to 4 locations, and received 503, 407, 501, and 404 responses, it may choose to forward the 407 (Proxy Authentication Required) response.

1xx and 2xx responses may be involved in the establishment of dialogs. When a request does not contain a To tag, the To tag in the response is used by the UAC to distinguish multiple responses to a dialog creating request. A proxy MUST NOT insert a tag into the To header field of a 1xx or 2xx response if the request did not contain one. A proxy MUST NOT modify the tag in the To header field of a 1xx or 2xx response.

Since a proxy may not insert a tag into the To header field of a 1xx response to a request that did not contain one, it cannot issue non-100 provisional responses on its own. However, it can branch the request to a UAS sharing the same element as the proxy. This UAS can return its own provisional responses, entering into an early dialog with the initiator of the request. The UAS does not have to be a discreet process from the proxy. It could be a virtual UAS implemented in the same code space as the proxy.

3-6xx responses are delivered hop-by-hop. When issuing a 3-6xx response, the element is effectively acting as a UAS, issuing its own response, usually based on the responses received from downstream elements. An element SHOULD preserve the To tag when simply forwarding a 3-6xx response to a request that did not contain a To tag.

A proxy MUST NOT modify the To tag in any forwarded response to a request that contains a To tag.

While it makes no difference to the upstream elements if the proxy replaced the To tag in a forwarded 3-6xx response, preserving the original tag may assist with debugging.

When the proxy is aggregating information from several responses, choosing a To tag from among them is arbitrary, and generating a new To tag may make debugging easier. This happens, for instance, when combining 401 (Unauthorized) and 407 (Proxy Authentication Required) challenges, or combining Contact values from unencrypted and unauthenticated 3xx responses.

7. Aggregate Authorization Header Field Values

If the selected response is a 401 (Unauthorized) or 407 (Proxy Authentication Required), the proxy MUST collect any WWW-Authenticate and Proxy-Authenticate header field values from all other 401 (Unauthorized) and 407 (Proxy Authentication Required) responses received so far in this response context and add them to this response without modification before forwarding. The resulting 401 (Unauthorized) or 407 (Proxy

Authentication Required) response could have several WWW-Authenticate AND Proxy-Authenticate header field values.

This is necessary because any or all of the destinations the request was forwarded to may have requested credentials. The client needs to receive all of those challenges and supply credentials for each of them when it retries the request. Motivation for this behavior is provided in Section 26.

8. Record-Route

If the selected response contains a Record-Route header field value originally provided by this proxy, the proxy MAY choose to rewrite the value before forwarding the response. This allows the proxy to provide different URIs for itself to the next upstream and downstream elements. A proxy may choose to use this mechanism for any reason. For instance, it is useful for multi-homed hosts.

If the proxy received the request over TLS, and sent it out over a non-TLS connection, the proxy MUST rewrite the URI in the Record-Route header field to be a SIPS URI. If the proxy received the request over a non-TLS connection, and sent it out over TLS, the proxy MUST rewrite the URI in the Record-Route header field to be a SIP URI.

The new URI provided by the proxy MUST satisfy the same constraints on URIs placed in Record-Route header fields in requests (see Step 4 of Section 16.6) with the following modifications:

The URI SHOULD NOT contain the transport parameter unless the proxy has knowledge that the next upstream (as opposed to downstream) element that will be in the path of subsequent requests supports that transport.

When a proxy does decide to modify the Record-Route header field in the response, one of the operations it performs is locating the Record-Route value that it had inserted. If the request spiraled, and the proxy inserted a Record-Route value in each iteration of the spiral, locating the correct value in the response (which must be the proper iteration in the reverse direction) is tricky. The rules above recommend that a proxy wishing to rewrite Record-Route header field values insert sufficiently distinct URIs into the Record-Route header field so that the right one may be selected for rewriting. A RECOMMENDED mechanism to achieve this is for the proxy to append a unique identifier for the proxy instance to the user portion of the URI.

When the response arrives, the proxy modifies the first Record-Route whose identifier matches the proxy instance. The modification results in a URI without this piece of data appended to the user portion of the URI. Upon the next iteration, the same algorithm (find the topmost Record-Route header field value with the parameter) will correctly extract the next Record-Route header field value inserted by that

proxy.

Not every response to a request to which a proxy adds a Record-Route header field value will contain a Record-Route header field. If the response does contain a Record-Route header field, it will contain the value the proxy added.

9. Forward response

After performing the processing described in steps "Aggregate Authorization Header Field Values" through "Record-Route", the proxy MAY perform any feature specific manipulations on the selected response. The proxy MUST NOT add to, modify, or remove the message body. Unless otherwise specified, the proxy MUST NOT remove any header field values other than the Via header field value discussed in Section 16.7 Item 3. In particular, the proxy MUST NOT remove any "received" parameter

it may have added to the next Via header field value while processing the request associated with this response. The proxy MUST pass the response to the server transaction associated with the response context. This will result in the response being sent to the location now indicated in the topmost Via header field value. If the server transaction is no longer available to handle the transmission, the element MUST forward the response statelessly by sending it to the server transport. The server transaction might indicate failure to send the response or signal a timeout in its state machine. These errors would be logged for diagnostic purposes as appropriate, but the protocol requires no remedial action from the proxy.

The proxy MUST maintain the response context until all of its associated transactions have been terminated, even after forwarding a final response.

10. Generate CANCELs

If the forwarded response was a final response, the proxy MUST generate a CANCEL request for all pending client transactions associated with this response context. A proxy SHOULD also generate a CANCEL request for all pending client transactions associated with this response context when it receives a 6xx response. A pending client transaction is one that has received a provisional response, but no final response (it is in the proceeding state) and has not had an associated CANCEL generated for it. Generating CANCEL requests is described in Section 9.1.

The requirement to CANCEL pending client transactions upon forwarding a final response does not guarantee that an endpoint will not receive multiple 200 (OK) responses to an INVITE. 200 (OK) responses on more than one branch may be generated before the CANCEL requests can be sent and processed. Further, it is reasonable to expect that a future extension may override this requirement to issue CANCEL requests.

16.8 Processing Timer C

If timer C should fire, the proxy MUST either reset the timer with any value it chooses, or terminate the client transaction. If the client transaction has received a provisional response, the proxy MUST generate a CANCEL request matching that transaction. If the client transaction has not received a provisional response, the proxy MUST behave as if the transaction received a 408 (Request Timeout) response.

Allowing the proxy to reset the timer allows the proxy to dynamically extend the transaction's lifetime based on current conditions (such as utilization) when the timer fires.

16.9 Handling Transport Errors

If the transport layer notifies a proxy of an error when it tries to forward a request (see Section 18.4), the proxy MUST behave as if the forwarded request received a 503 (Service Unavailable) response.

If the proxy is notified of an error when forwarding a response, it drops the response. The proxy SHOULD NOT cancel any outstanding client transactions associated with this response context due to this notification.

> If a proxy cancels its outstanding client transactions, a single malicious or misbehaving client can cause all transactions to fail through its Via header field.

16.10 CANCEL Processing

A stateful proxy MAY generate a CANCEL to any other request it has generated at any time (subject to receiving a provisional response to that request as described in section 9.1). A proxy MUST cancel any pending client transactions associated with a response context when it receives a matching CANCEL request.

A stateful proxy MAY generate CANCEL requests for pending INVITE client transactions based on the period specified in the INVITE's Expires header field elapsing. However, this is generally unnecessary since the endpoints involved will take care of signaling the end of the transaction.

While a CANCEL request is handled in a stateful proxy by its own server transaction, a new response context is not created for it. Instead, the proxy layer searches its existing response contexts for the server transaction handling the request associated with this CANCEL. If a matching response context is found, the element MUST immediately return a 200 (OK) response to the CANCEL request. In this case, the element is acting as a user agent server as defined in Section 8.2. Furthermore, the element MUST generate CANCEL requests for all pending client transactions in the context as described in Section 16.7 step 10.

If a response context is not found, the element does not have any knowledge of the request to apply the CANCEL to. It MUST statelessly forward the CANCEL request (it may have statelessly forwarded the

associated request previously).

16.11 Stateless Proxy

When acting statelessly, a proxy is a simple message forwarder. Much of the processing performed when acting statelessly is the same as when behaving statefully. The differences are detailed here.

A stateless proxy does not have any notion of a transaction, or of the response context used to describe stateful proxy behavior. Instead, the stateless proxy takes messages, both requests and responses, directly from the transport layer (See section 18). As a result, stateless proxies do not retransmit messages on their own. They do, however, forward all retransmissions they receive (they do not have the ability to distinguish a retransmission from the original message). Furthermore, when handling a request statelessly, an element MUST NOT generate its own 100 (Trying) or any other provisional response.

A stateless proxy MUST validate a request as described in Section 16.3

A stateless proxy MUST follow the request processing steps described in Sections 16.4 through 16.5 with the following exception:

- A stateless proxy MUST choose one and only one target from the target set. This choice MUST only rely on fields in the message and time-invariant properties of the server. In particular, a retransmitted request MUST be forwarded to the same destination each time it is processed. Furthermore, CANCEL and non-Routed ACK requests MUST generate the same choice as their associated INVITE.

A stateless proxy MUST follow the request processing steps described in Section 16.6 with the following exceptions:

- The requirement for unique branch IDs across space and time applies to stateless proxies as well. However, a stateless proxy cannot simply use a random number generator to compute the first component of the branch ID, as described in Section 16.6 bullet 8. This is because retransmissions of a request need to have the same value, and a stateless proxy cannot tell a retransmission from the original request. Therefore, the component of the branch parameter that makes it unique MUST be the same each time a retransmitted request is forwarded. Thus for a stateless proxy, the branch parameter MUST be computed as a combinatoric function of message parameters which are invariant on retransmission.

  The stateless proxy MAY use any technique it likes to guarantee uniqueness of its branch IDs across transactions. However, the following procedure is RECOMMENDED. The proxy examines the branch ID in the topmost Via header field of the received request. If it begins with the magic cookie, the first component of the branch ID of the outgoing request is computed as a hash of the received branch ID. Otherwise, the first component of the branch ID is computed as a hash of the topmost

Via, the tag in the To header field, the tag in the From header field, the Call-ID header field, the CSeq number (but not method), and the Request-URI from the received request. One of these fields will always vary across two different transactions.

- All other message transformations specified in Section 16.6 MUST result in the same transformation of a retransmitted request. In particular, if the proxy inserts a Record-Route value or pushes URIs into the Route header field, it MUST place the same values in retransmissions of the request. As for the Via branch parameter, this implies that the transformations MUST be based on time-invariant configuration or retransmission-invariant properties of the request.

- A stateless proxy determines where to forward the request as described for stateful proxies in Section 16.6 Item 10. The request is sent directly to the transport layer instead of through a client transaction.

  Since a stateless proxy must forward retransmitted requests to the same destination and add identical branch parameters to each of them, it can only use information from the message itself and time-invariant configuration data for those calculations. If the configuration state is not time-invariant (for example, if a routing table is updated) any requests that could be affected by the change may not be forwarded statelessly during an interval equal to the transaction timeout window before or after the change. The method of processing the affected requests in that interval is an implementation decision. A common solution is to forward them transaction statefully.

Stateless proxies MUST NOT perform special processing for CANCEL requests. They are processed by the above rules as any other requests. In particular, a stateless proxy applies the same Route header field processing to CANCEL requests that it applies to any other request.

Response processing as described in Section 16.7 does not apply to a proxy behaving statelessly. When a response arrives at a stateless proxy, the proxy MUST inspect the sent-by value in the first (topmost) Via header field value. If that address matches the proxy, (it equals a value this proxy has inserted into previous requests) the proxy MUST remove that header field value from the response and forward the result to the location indicated in the next Via header field value. The proxy MUST NOT add to, modify, or remove the message body. Unless specified otherwise, the proxy MUST NOT remove any other header field values. If the address does not match the proxy, the message MUST be silently discarded.

16.12 Summary of Proxy Route Processing

In the absence of local policy to the contrary, the processing a proxy performs on a request containing a Route header field can be summarized in the following steps.

1. The proxy will inspect the Request-URI. If it indicates a resource owned by this proxy, the proxy will replace it with the results of running a location service. Otherwise, the proxy will not change the Request-URI.

2. The proxy will inspect the URI in the topmost Route header field value. If it indicates this proxy, the proxy removes it from the Route header field (this route node has been reached).

3. The proxy will forward the request to the resource indicated by the URI in the topmost Route header field value or in the Request-URI if no Route header field is present. The proxy determines the address, port and transport to use when forwarding the request by applying the procedures in [4] to that URI.

If no strict-routing elements are encountered on the path of the request, the Request-URI will always indicate the target of the request.

16.12.1 Examples

16.12.1.1 Basic SIP Trapezoid

This scenario is the basic SIP trapezoid, U1 -> P1 -> P2 -> U2, with both proxies record-routing. Here is the flow.

U1 sends:

```
INVITE sip:callee@domain.com SIP/2.0
Contact: sip:caller@u1.example.com
```

to P1. P1 is an outbound proxy. P1 is not responsible for domain.com, so it looks it up in DNS and sends it there. It also adds a Record-Route header field value:

```
INVITE sip:callee@domain.com SIP/2.0
Contact: sip:caller@u1.example.com
Record-Route: <sip:p1.example.com;lr>
```

P2 gets this. It is responsible for domain.com so it runs a location service and rewrites the Request-URI. It also adds a Record-Route header field value. There is no Route header field, so it resolves the new Request-URI to determine where to send the request:

```
INVITE sip:callee@u2.domain.com SIP/2.0
Contact: sip:caller@u1.example.com
Record-Route: <sip:p2.domain.com;lr>
Record-Route: <sip:p1.example.com;lr>
```

The callee at u2.domain.com gets this and responds with a 200 OK:

```
SIP/2.0 200 OK
Contact: sip:callee@u2.domain.com
Record-Route: <sip:p2.domain.com;lr>
Record-Route: <sip:p1.example.com;lr>
```

The callee at u2 also sets its dialog state's remote target URI to sip:caller@u1.example.com and its route set to:

```
(<sip:p2.domain.com;lr>,<sip:p1.example.com;lr>)
```

This is forwarded by P2 to P1 to U1 as normal. Now, U1 sets its dialog state's remote target URI to sip:callee@u2.domain.com and its route set to:

```
(<sip:p1.example.com;lr>,<sip:p2.domain.com;lr>)
```

Since all the route set elements contain the lr parameter, U1 constructs the following BYE request:

```
BYE sip:callee@u2.domain.com SIP/2.0
Route: <sip:p1.example.com;lr>,<sip:p2.domain.com;lr>
```

As any other element (including proxies) would do, it resolves the URI in the topmost Route header field value using DNS to determine where to send the request. This goes to P1. P1 notices that it is not responsible for the resource indicated in the Request-URI so it doesn't change it. It does see that it is the first value in the Route header field, so it removes that value, and forwards the request to P2:

```
BYE sip:callee@u2.domain.com SIP/2.0
Route: <sip:p2.domain.com;lr>
```

P2 also notices it is not responsible for the resource indicated by the Request-URI (it is responsible for domain.com, not u2.domain.com), so it doesn't change it. It does see itself in the first Route header field value, so it removes it and forwards the following to u2.domain.com based on a DNS lookup against the Request-URI:

```
BYE sip:callee@u2.domain.com SIP/2.0
```

16.12.1.2 Traversing a Strict-Routing Proxy

In this scenario, a dialog is established across four proxies, each of which adds Record-Route header field values. The third proxy implements the strict-routing procedures specified in RFC 2543 and many works in progress.

```
U1->P1->P2->P3->P4->U2
```

The INVITE arriving at U2 contains:

```
INVITE sip:callee@u2.domain.com SIP/2.0
Contact: sip:caller@u1.example.com
Record-Route: <sip:p4.domain.com;lr>
Record-Route: <sip:p3.middle.com>
Record-Route: <sip:p2.example.com;lr>
Record-Route: <sip:p1.example.com;lr>
```

Which U2 responds to with a 200 OK. Later, U2 sends the following

BYE request to P4 based on the first Route header field value.

```
BYE sip:caller@u1.example.com SIP/2.0
Route: <sip:p4.domain.com;lr>
Route: <sip:p3.middle.com>
Route: <sip:p2.example.com;lr>
Route: <sip:p1.example.com;lr>
```

P4 is not responsible for the resource indicated in the Request-URI so it will leave it alone. It notices that it is the element in the first Route header field value so it removes it. It then prepares to send the request based on the now first Route header field value of sip:p3.middle.com, but it notices that this URI does not contain the lr parameter, so before sending, it reformats the request to be:

```
BYE sip:p3.middle.com SIP/2.0
Route: <sip:p2.example.com;lr>
Route: <sip:p1.example.com;lr>
Route: <sip:caller@u1.example.com>
```

P3 is a strict router, so it forwards the following to P2:

```
BYE sip:p2.example.com;lr SIP/2.0
Route: <sip:p1.example.com;lr>
Route: <sip:caller@u1.example.com>
```

P2 sees the request-URI is a value it placed into a Record-Route header field, so before further processing, it rewrites the request to be:

```
BYE sip:caller@u1.example.com SIP/2.0
Route: <sip:p1.example.com;lr>
```

P2 is not responsible for u1.example.com, so it sends the request to P1 based on the resolution of the Route header field value.

P1 notices itself in the topmost Route header field value, so it removes it, resulting in:

```
BYE sip:caller@u1.example.com SIP/2.0
```

Since P1 is not responsible for u1.example.com and there is no Route header field, P1 will forward the request to u1.example.com based on the Request-URI.

16.12.1.3 Rewriting Record-Route Header Field Values

In this scenario, U1 and U2 are in different private namespaces and they enter a dialog through a proxy P1, which acts as a gateway between the namespaces.

```
U1->P1->U2
```

U1 sends:

```
INVITE sip:callee@gateway.leftprivatespace.com SIP/2.0
Contact: <sip:caller@u1.leftprivatespace.com>
```

P1 uses its location service and sends the following to U2:

```
INVITE sip:callee@rightprivatespace.com SIP/2.0
Contact: <sip:caller@u1.leftprivatespace.com>
Record-Route: <sip:gateway.rightprivatespace.com;lr>
```

U2 sends this 200 (OK) back to P1:

```
SIP/2.0 200 OK
Contact: <sip:callee@u2.rightprivatespace.com>
Record-Route: <sip:gateway.rightprivatespace.com;lr>
```

P1 rewrites its Record-Route header parameter to provide a value that U1 will find useful, and sends the following to U1:

```
SIP/2.0 200 OK
Contact: <sip:callee@u2.rightprivatespace.com>
Record-Route: <sip:gateway.leftprivatespace.com;lr>
```

Later, U1 sends the following BYE request to P1:

```
BYE sip:callee@u2.rightprivatespace.com SIP/2.0
Route: <sip:gateway.leftprivatespace.com;lr>
```

which P1 forwards to U2 as:

```
BYE sip:callee@u2.rightprivatespace.com SIP/2.0
```

17 Transactions

SIP is a transactional protocol: interactions between components take place in a series of independent message exchanges. Specifically, a SIP transaction consists of a single request and any responses to that request, which include zero or more provisional responses and one or more final responses. In the case of a transaction where the request was an INVITE (known as an INVITE transaction), the transaction also includes the ACK only if the final response was not a 2xx response. If the response was a 2xx, the ACK is not considered part of the transaction.

> The reason for this separation is rooted in the importance of delivering all 200 (OK) responses to an INVITE to the UAC. To deliver them all to the UAC, the UAS alone takes responsibility
>
> for retransmitting them (see Section 13.3.1.4), and the UAC alone takes responsibility for acknowledging them with ACK (see Section 13.2.2.4). Since this ACK is retransmitted only by the UAC, it is effectively considered its own transaction.

Transactions have a client side and a server side. The client side is known as a client transaction and the server side as a server transaction. The client transaction sends the request, and the server transaction sends the response. The client and server transactions are logical functions that are embedded in any number of elements. Specifically, they exist within user agents and stateful proxy servers. Consider the example in Section 4. In this example,

the UAC executes the client transaction, and its outbound proxy executes the server transaction. The outbound proxy also executes a client transaction, which sends the request to a server transaction in the inbound proxy. That proxy also executes a client transaction, which in turn sends the request to a server transaction in the UAS. This is shown in Figure 4.

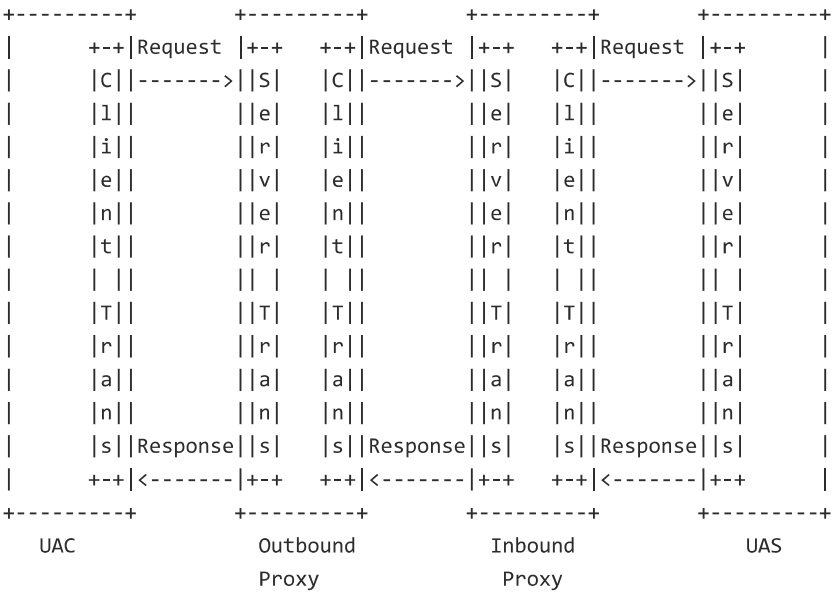


Figure 4: Transaction relationships

A stateless proxy does not contain a client or server transaction. The transaction exists between the UA or stateful proxy on one side, and the UA or stateful proxy on the other side. As far as SIP transactions are concerned, stateless proxies are effectively transparent. The purpose of the client transaction is to receive a request from the element in which the client is embedded (call this element the "Transaction User" or TU; it can be a UA or a stateful proxy), and reliably deliver the request to a server transaction.

The client transaction is also responsible for receiving responses and delivering them to the TU, filtering out any response retransmissions or disallowed responses (such as a response to ACK). Additionally, in the case of an INVITE request, the client transaction is responsible for generating the ACK request for any final response accepting a 2xx response.

Similarly, the purpose of the server transaction is to receive requests from the transport layer and deliver them to the TU. The server transaction filters any request retransmissions from the network. The server transaction accepts responses from the TU and delivers them to the transport layer for transmission over the network. In the case of an INVITE transaction, it absorbs the ACK request for any final response excepting a 2xx response.

The 2xx response and its ACK receive special treatment. This response is retransmitted only by a UAS, and its ACK generated only by the UAC. This end-to-end treatment is needed so that a caller knows the entire set of users that have accepted the call. Because of this special handling, retransmissions of the 2xx response are

handled by the UA core, not the transaction layer. Similarly, generation of the ACK for the 2xx is handled by the UA core. Each proxy along the path merely forwards each 2xx response to INVITE and its corresponding ACK.

17.1 Client Transaction

The client transaction provides its functionality through the maintenance of a state machine.

The TU communicates with the client transaction through a simple interface. When the TU wishes to initiate a new transaction, it creates a client transaction and passes it the SIP request to send and an IP address, port, and transport to which to send it. The client transaction begins execution of its state machine. Valid responses are passed up to the TU from the client transaction.

There are two types of client transaction state machines, depending on the method of the request passed by the TU. One handles client transactions for INVITE requests. This type of machine is referred to as an INVITE client transaction. Another type handles client transactions for all requests except INVITE and ACK. This is referred to as a non-INVITE client transaction. There is no client transaction for ACK. If the TU wishes to send an ACK, it passes one directly to the transport layer for transmission.

The INVITE transaction is different from those of other methods because of its extended duration. Normally, human input is required in order to respond to an INVITE. The long delays expected for sending a response argue for a three-way handshake. On the other hand, requests of other methods are expected to complete rapidly. Because of the non-INVITE transaction's reliance on a two-way handshake, TUs SHOULD respond immediately to non-INVITE requests.

17.1.1 INVITE Client Transaction

17.1.1.1 Overview of INVITE Transaction

The INVITE transaction consists of a three-way handshake. The client transaction sends an INVITE, the server transaction sends responses, and the client transaction sends an ACK. For unreliable transports (such as UDP), the client transaction retransmits requests at an interval that starts at T1 seconds and doubles after every retransmission. T1 is an estimate of the round-trip time (RTT), and it defaults to 500 ms. Nearly all of the transaction timers described here scale with T1, and changing T1 adjusts their values. The request is not retransmitted over reliable transports. After receiving a 1xx response, any retransmissions cease altogether, and the client waits for further responses. The server transaction can send additional 1xx responses, which are not transmitted reliably by the server transaction. Eventually, the server transaction decides to send a final response. For unreliable transports, that response is retransmitted periodically, and for reliable transports, it is sent once. For each final response that is received at the client transaction, the client transaction sends an ACK, the purpose of which is to quench retransmissions of the response.

17.1.1.2 Formal Description

The state machine for the INVITE client transaction is shown in Figure 5. The initial state, "calling", MUST be entered when the TU initiates a new client transaction with an INVITE request. The client transaction MUST pass the request to the transport layer for transmission (see Section 18). If an unreliable transport is being used, the client transaction MUST start timer A with a value of T1. If a reliable transport is being used, the client transaction SHOULD NOT start timer A (Timer A controls request retransmissions). For any transport, the client transaction MUST start timer B with a value of 64*T1 seconds (Timer B controls transaction timeouts).

When timer A fires, the client transaction MUST retransmit the request by passing it to the transport layer, and MUST reset the timer with a value of 2*T1. The formal definition of retransmit

within the context of the transaction layer is to take the message previously sent to the transport layer and pass it to the transport layer once more.

When timer A fires 2*T1 seconds later, the request MUST be retransmitted again (assuming the client transaction is still in this state). This process MUST continue so that the request is retransmitted with intervals that double after each transmission. These retransmissions SHOULD only be done while the client transaction is in the "calling" state.

The default value for T1 is 500 ms. T1 is an estimate of the RTT between the client and server transactions. Elements MAY (though it is NOT RECOMMENDED) use smaller values of T1 within closed, private networks that do not permit general Internet connection. T1 MAY be chosen larger, and this is RECOMMENDED if it is known in advance (such as on high latency access links) that the RTT is larger. Whatever the value of T1, the exponential backoffs on retransmissions described in this section MUST be used.

If the client transaction is still in the "Calling" state when timer B fires, the client transaction SHOULD inform the TU that a timeout has occurred. The client transaction MUST NOT generate an ACK. The value of 64*T1 is equal to the amount of time required to send seven requests in the case of an unreliable transport.

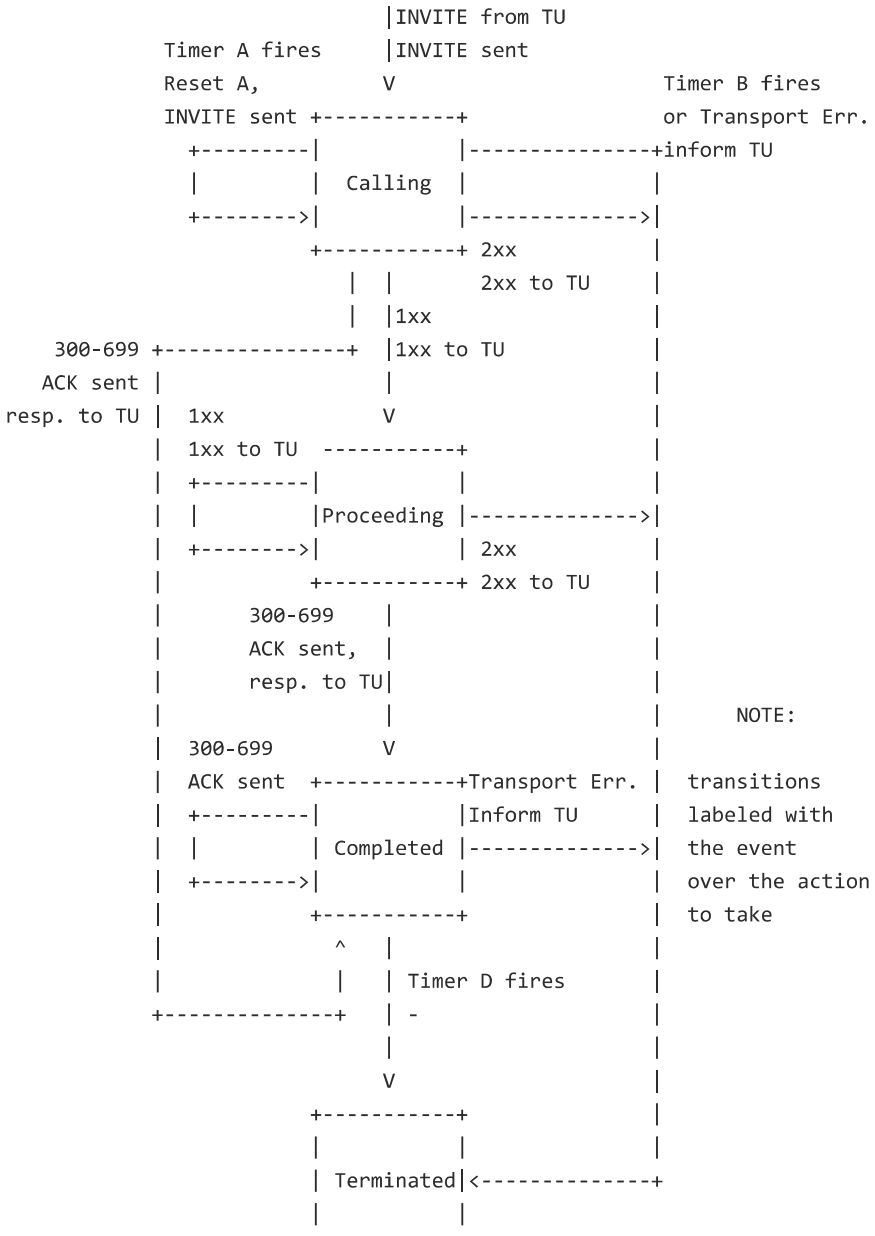
If the client transaction receives a provisional response while in the "Calling" state, it transitions to the "Proceeding" state. In the "Proceeding" state, the client transaction SHOULD NOT retransmit the request any longer. Furthermore, the provisional response MUST be passed to the TU. Any further provisional responses MUST be passed up to the TU while in the "Proceeding" state.

When in either the "Calling" or "Proceeding" states, reception of a response with status code from 300-699 MUST cause the client transaction to transition to "Completed". The client transaction MUST pass the received response up to the TU, and the client transaction MUST generate an ACK request, even if the transport is reliable (guidelines for constructing the ACK from the response are given in Section 17.1.1.3) and then pass the ACK to the transport

layer for transmission. The ACK MUST be sent to the same address, port, and transport to which the original request was sent. The client transaction SHOULD start timer D when it enters the "Completed" state, with a value of at least 32 seconds for unreliable transports, and a value of zero seconds for reliable transports. Timer D reflects the amount of time that the server transaction can remain in the "Completed" state when unreliable transports are used. This is equal to Timer H in the INVITE server transaction, whose

default is 64*T1. However, the client transaction does not know the value of T1 in use by the server transaction, so an absolute minimum of 32s is used instead of basing Timer D on T1.

Any retransmissions of the final response that are received while in the "Completed" state MUST cause the ACK to be re-passed to the transport layer for retransmission, but the newly received response MUST NOT be passed up to the TU. A retransmission of the response is defined as any response which would match the same client transaction based on the rules of Section 17.1.3.



```
                          +-----------+
```

Figure 5: INVITE client transaction

If timer D fires while the client transaction is in the "Completed" state, the client transaction MUST move to the terminated state.

When in either the "Calling" or "Proceeding" states, reception of a 2xx response MUST cause the client transaction to enter the "Terminated" state, and the response MUST be passed up to the TU. The handling of this response depends on whether the TU is a proxy

core or a UAC core. A UAC core will handle generation of the ACK for this response, while a proxy core will always forward the 200 (OK) upstream. The differing treatment of 200 (OK) between proxy and UAC is the reason that handling of it does not take place in the transaction layer.

The client transaction MUST be destroyed the instant it enters the "Terminated" state. This is actually necessary to guarantee correct operation. The reason is that 2xx responses to an INVITE are treated differently; each one is forwarded by proxies, and the ACK handling in a UAC is different. Thus, each 2xx needs to be passed to a proxy core (so that it can be forwarded) and to a UAC core (so it can be acknowledged). No transaction layer processing takes place. Whenever a response is received by the transport, if the transport layer finds no matching client transaction (using the rules of Section 17.1.3), the response is passed directly to the core. Since the matching client transaction is destroyed by the first 2xx, subsequent 2xx will find no match and therefore be passed to the core.

17.1.1.3 Construction of the ACK Request

This section specifies the construction of ACK requests sent within the client transaction. A UAC core that generates an ACK for 2xx MUST instead follow the rules described in Section 13.

The ACK request constructed by the client transaction MUST contain values for the Call-ID, From, and Request-URI that are equal to the values of those header fields in the request passed to the transport by the client transaction (call this the "original request"). The To header field in the ACK MUST equal the To header field in the response being acknowledged, and therefore will usually differ from the To header field in the original request by the addition of the tag parameter. The ACK MUST contain a single Via header field, and this MUST be equal to the top Via header field of the original request. The CSeq header field in the ACK MUST contain the same value for the sequence number as was present in the original request, but the method parameter MUST be equal to "ACK".

If the INVITE request whose response is being acknowledged had Route header fields, those header fields MUST appear in the ACK. This is to ensure that the ACK can be routed properly through any downstream stateless proxies.

Although any request MAY contain a body, a body in an ACK is special

since the request cannot be rejected if the body is not understood. Therefore, placement of bodies in ACK for non-2xx is NOT RECOMMENDED, but if done, the body types are restricted to any that appeared in the INVITE, assuming that the response to the INVITE was not 415. If it was, the body in the ACK MAY be any type listed in the Accept header field in the 415.

For example, consider the following request:

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKkjshdyff
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=88sja8x
Max-Forwards: 70
Call-ID: 987asjd97y7atg
CSeq: 986759 INVITE
```

The ACK request for a non-2xx final response to this request would look like this:

```
ACK sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKkjshdyff
To: Bob <sip:bob@biloxi.com>;tag=99sa0xk
From: Alice <sip:alice@atlanta.com>;tag=88sja8x
Max-Forwards: 70
Call-ID: 987asjd97y7atg
CSeq: 986759 ACK
```

17.1.2 Non-INVITE Client Transaction

17.1.2.1 Overview of the non-INVITE Transaction

Non-INVITE transactions do not make use of ACK. They are simple request-response interactions. For unreliable transports, requests are retransmitted at an interval which starts at T1 and doubles until it hits T2. If a provisional response is received, retransmissions continue for unreliable transports, but at an interval of T2. The server transaction retransmits the last response it sent, which can be a provisional or final response, only when a retransmission of the request is received. This is why request retransmissions need to continue even after a provisional response; they are to ensure reliable delivery of the final response.

Unlike an INVITE transaction, a non-INVITE transaction has no special handling for the 2xx response. The result is that only a single 2xx response to a non-INVITE is ever delivered to a UAC.

17.1.2.2 Formal Description

The state machine for the non-INVITE client transaction is shown in Figure 6. It is very similar to the state machine for INVITE.

The "Trying" state is entered when the TU initiates a new client transaction with a request. When entering this state, the client transaction SHOULD set timer F to fire in 64*T1 seconds. The request MUST be passed to the transport layer for transmission. If an unreliable transport is in use, the client transaction MUST set timer

E to fire in T1 seconds. If timer E fires while still in this state, the timer is reset, but this time with a value of MIN(2*T1, T2). When the timer fires again, it is reset to a MIN(4*T1, T2). This process continues so that retransmissions occur with an exponentially increasing interval that caps at T2. The default value of T2 is 4s, and it represents the amount of time a non-INVITE server transaction will take to respond to a request, if it does not respond immediately. For the default values of T1 and T2, this results in intervals of 500 ms, 1 s, 2 s, 4 s, 4 s, 4 s, etc.

If Timer F fires while the client transaction is still in the "Trying" state, the client transaction SHOULD inform the TU about the timeout, and then it SHOULD enter the "Terminated" state. If a provisional response is received while in the "Trying" state, the response MUST be passed to the TU, and then the client transaction SHOULD move to the "Proceeding" state. If a final response (status codes 200-699) is received while in the "Trying" state, the response MUST be passed to the TU, and the client transaction MUST transition to the "Completed" state.

If Timer E fires while in the "Proceeding" state, the request MUST be passed to the transport layer for retransmission, and Timer E MUST be reset with a value of T2 seconds. If timer F fires while in the "Proceeding" state, the TU MUST be informed of a timeout, and the client transaction MUST transition to the terminated state. If a final response (status codes 200-699) is received while in the "Proceeding" state, the response MUST be passed to the TU, and the client transaction MUST transition to the "Completed" state.

Once the client transaction enters the "Completed" state, it MUST set Timer K to fire in T4 seconds for unreliable transports, and zero seconds for reliable transports. The "Completed" state exists to buffer any additional response retransmissions that may be received (which is why the client transaction remains there only for

unreliable transports). T4 represents the amount of time the network will take to clear messages between client and server transactions. The default value of T4 is 5s. A response is a retransmission when it matches the same transaction, using the rules specified in Section 17.1.3. If Timer K fires while in this state, the client transaction MUST transition to the "Terminated" state.

Once the transaction is in the terminated state, it MUST be destroyed immediately.

17.1.3 Matching Responses to Client Transactions

When the transport layer in the client receives a response, it has to determine which client transaction will handle the response, so that the processing of Sections 17.1.1 and 17.1.2 can take place. The branch parameter in the top Via header field is used for this purpose. A response matches a client transaction under two conditions:

1. If the response has the same value of the branch parameter in the top Via header field as the branch parameter in the top Via header field of the request that created the transaction.