Network Working Group
Request for Comments: 5057
Category: Informational

R. Sparks
Estacado Systems

November 2007

Multiple Dialog Usages in the Session Initiation Protocol

Status of This Memo

Abstract

Several methods in the Session Initiation Protocol (SIP) can create an association between endpoints known as a dialog. Some of these methods can also create a different, but related, association within an existing dialog. These multiple associations, or dialog usages, require carefully coordinated processing as they have independent life-cycles, but share common dialog state. Processing multiple dialog usages correctly is not completely understood. What is understood is difficult to implement.

This memo argues that multiple dialog usages should be avoided. It discusses alternatives to their use and clarifies essential behavior for elements that cannot currently avoid them.

This is an informative document and makes no normative statements of any kind.

Table of Contents

1. Overview

This is an informative document. It makes no normative statements of any kind. This document refines the concept of a dialog usage in the Session Initiation Protocol (SIP [1]), and discusses what led to its existence. It explores ambiguity associated with processing multiple dialog usages that share a dialog. In particular, it surveys the effect of SIP failure responses on transaction, dialog usage, and dialog state. This document will help the implementer understand what is required to process multiple dialog usages correctly, and will provide information for future standards-track work that will clarify RFC 3261 and other related documents. Finally, the document explores single-usage dialog alternatives (using SIP extensions) to multiple dialog usages.

2. Introduction

Several methods in SIP can establish a dialog. When they do so, they also establish an association between the endpoints within that dialog. This association has been known for some time as a "dialog usage" in the developer community. A dialog initiated with an INVITE request has an invite usage. A dialog initiated with a SUBSCRIBE

request has a subscribe usage. A dialog initiated with a REFER request has a subscribe usage.

Dialogs with multiple usages arise when a usage-creating action occurs inside an existing dialog. Such actions include accepting a REFER or SUBSCRIBE issued inside a dialog established with an INVITE request. Multiple REFERs within a dialog create multiple subscriptions, each of which is a new dialog usage sharing common dialog state. (Note that any REFER issued utilizing the subscription-suppression mechanism specified in [2] creates no new usage.) Similarly, an endpoint in a dialog established with an INVITE might subscribe to its peer's Key Press Markup Language (KPML) [3] and later issue a REFER, resulting in three dialog usages sharing common dialog state.

The common state in the dialog shared by any usages is exactly:

- the Call-ID
- the local Tag
- the remote Tag
- the local CSeq
- the remote CSeq
- the Route-set
- the local contact
- the remote target
- the secure flag

Usages have state that is not shared in the dialog. For example, a subscription has a duration, along with other usage-specific state. Multiple subscriptions in the same dialog each have their own duration.

A dialog comes into existence with the creation of the first usage, and continues to exist until the last usage is terminated (reference counting). Unfortunately, many of the usage management aspects of SIP, such as authentication, were originally designed with the implicit assumption that there was one usage per dialog. The resulting mechanisms have mixed effects, some influencing the usage, and some influencing the entire dialog.

The current specifications define two usages, invite and subscribe. A dialog can share up to one invite usage and arbitrarily many subscribe usages.

Because RFC 3261 [1] states that user-agents should reuse Call-ID and increment CSeq across a series of registration requests (and that to-tags appear in register responses in some of the examples), some implementations have treated REGISTER as if it were in a dialog. However, RFC 3261 explicitly calls out that REGISTER does not create a dialog. A series of REGISTER requests does not create any usage or dialog. Similarly, PUBLISH [4] does not create any usage or dialog.

3. Examples of Multiple Usages

3.1. Transfer

In Figure 1, Alice transfers a call she received from Bob to Carol. A dialog (and an invite dialog usage) between Alice and Bob comes into being with the 200 OK labeled F1. A second usage (a subscription to event refer) comes into being with the NOTIFY labeled F2. This second usage ends when the subscription is terminated by the NOTIFY transaction labeled F3. The dialog still has one usage (the invite usage), which lasts until the BYE transaction labeled F4. At this point, the dialog has no remaining usages, so it ceases to exist. Details of each of these messages are shown in Figure 2.

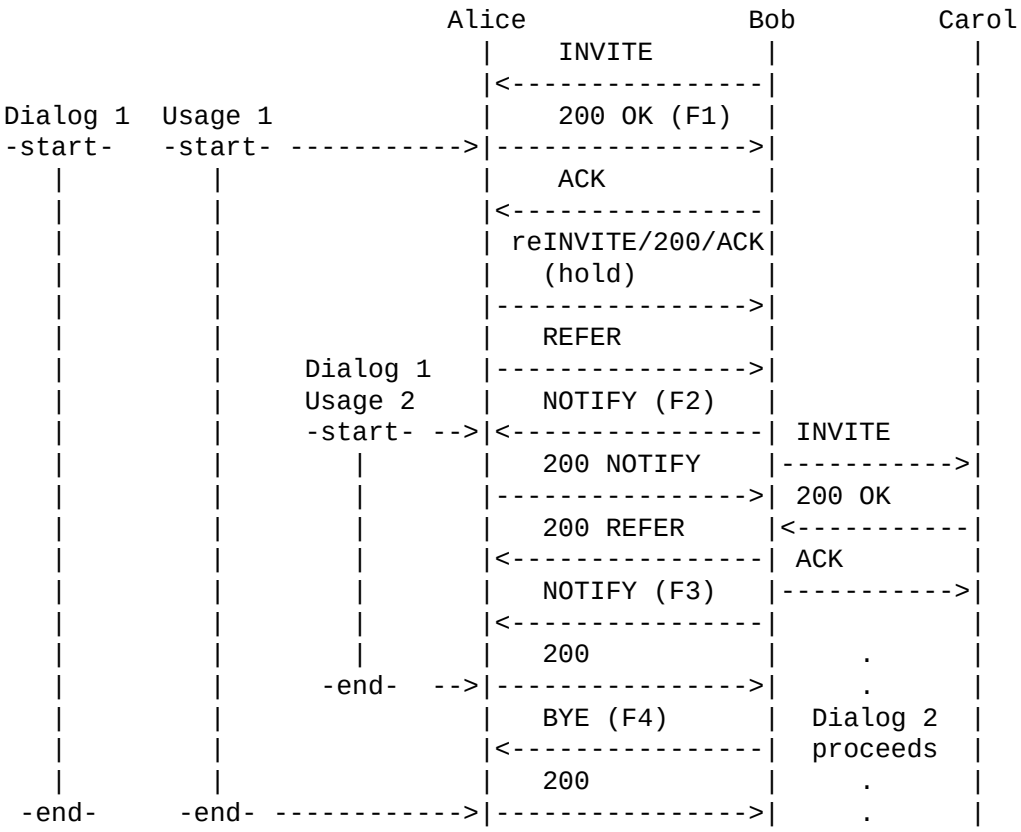


Figure 1

Message Details (abridged to show only dialog or usage details)

F1

```
SIP/2.0 200 OK
Call-ID: dialog1@bob.example.com
CSeq: 100 INVITE
To: <sip:Alice@alice.example.com>;tag=alicetag1
From: <sip:Bob@bob.example.com>;tag=bobtag1
Contact: <sip:aliceinstance@alice.example.com>
```

F2

```
NOTIFY sip:aliceinstance@alice.example.com SIP/2.0
Event: refer
Call-ID: dialog1@bob.example.com
CSeq: 101 NOTIFY
To: <sip:Alice@alice.example.com>;tag=alicetag1
From: <sip:Bob@bob.example.com>;tag=bobtag1
Contact: <sip:bobinstance@bob.example.com>
```

```
F3
  NOTIFY sip:aliceinstance@alice.example.com SIP/2.0
  Event: refer
  Subscription-State: terminated;reason=noresource
  Call-ID: dialog1@bob.example.com
  CSeq: 102 NOTIFY
  To: <sip:Alice@alice.example.com>;tag=alicetag1
  From: <sip:Bob@bob.example.com>;tag=bobtag1
  Contact: <sip:bobinstance@bob.example.com>
  Content-Type: message/sipfrag

  SIP/2.0 200 OK

F4
  BYE sip:aliceinstance@alice.example.com SIP/2.0
  Call-ID: dialog1@bob.example.com
  CSeq: 103 BYE
  To: <sip:Alice@alice.example.com>;tag=alicetag1
  From: <sip:Bob@bob.example.com>;tag=bobtag1
  Contact: <sip:bobinstance@bob.example.com>
```

Figure 2

3.2. Reciprocal Subscription

In Figure 3, Alice subscribes to Bob's presence. For simplicity, assume Bob and Alice are both serving their presence from their endpoints instead of a presence server. To focus on the essential points, the figure leaves out any rendezvous signaling through which Alice discovers Bob's endpoint.

Bob is interested in Alice's presence too, so he subscribes to Alice (in most deployed presence/IM systems, people watch each other). He decides to skip the rendezvous step since he's already in a dialog with Alice, and sends his SUBSCRIBE inside that dialog (a few early SIMPLE clients behaved exactly this way).

The dialog and its first usage comes into being at F1, which establishes Alice's subscription to Bob. Its second usage begins at F2, which establishes Bob's subscription to Alice. These two subscriptions are independent - they have distinct and different expirations, but they share all the dialog state.

The first usage ends when Alice decides to unsubscribe at F3. Bob's subscription to Alice, and thus the dialog, continues to exist. Alice's UA must maintain this dialog state even though the subscription that caused it to exist in the first place is now over. The second usage ends when Alice decides to terminate Bob's

subscription at F4 (she's probably going to reject any attempt on Bob's part to resubscribe until she's ready to subscribe to Bob again). Since this was the last usage, the dialog also terminates. Details of these messages are shown in Figure 4.

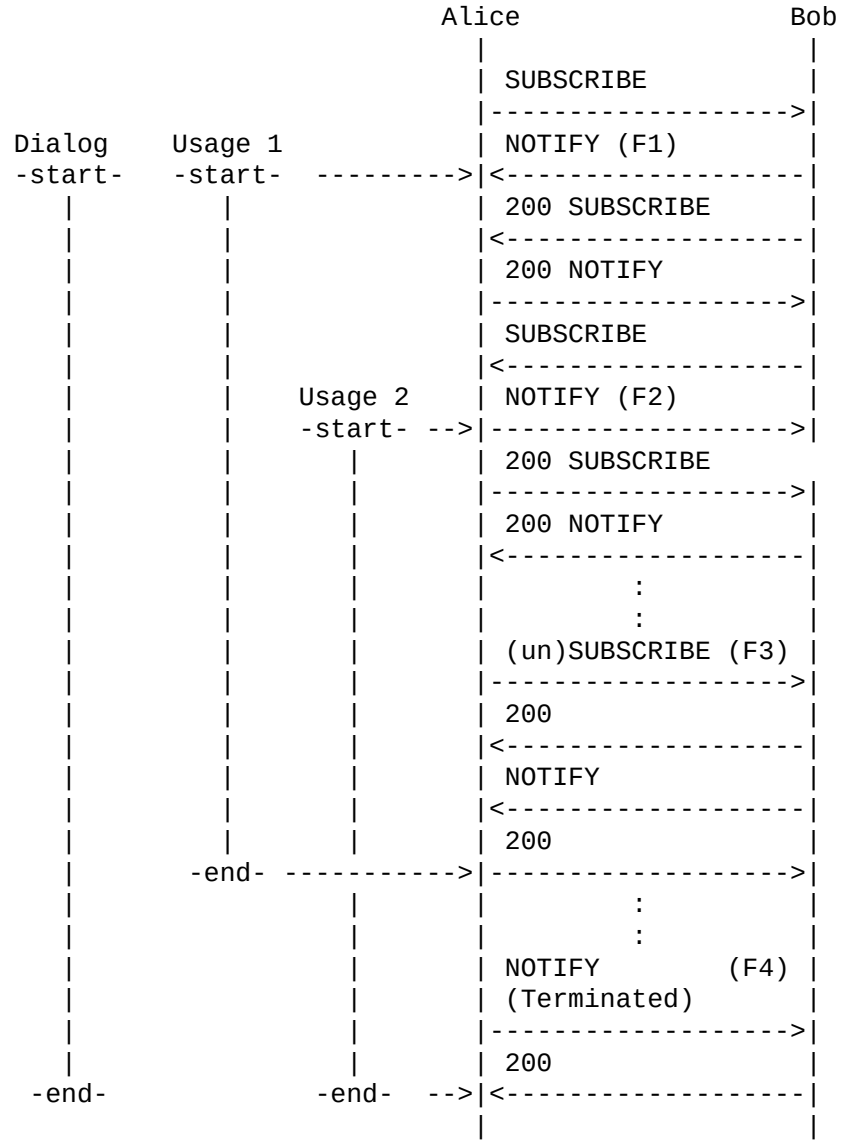


Figure 3

```
Message Details (abridged to show only dialog or usage details)
F1
  NOTIFY sip:aliceinstance@alice.example.com SIP/2.0
  Event: presence
  Subscription-State: active;expires=600
  Call-ID: alicecallid1@alice.example.com
  From: <sip:Bob@bob.example.com>;tag=bobtag2
  To: <sip:Alice@alice.example.com>;tag=alicetag2
  CSeq: 100 NOTIFY
  Contact: <sip:bobinstance@bob.example.com>

F2
  NOTIFY sip:bobinstance@bob.example.com SIP/2.0
  Event: presence
  Subscription-State: active;expires=1200
  Call-ID: alicecallid1@alice.example.com
  To: <sip:Bob@bob.example.com>;tag=bobtag2
  From: <sip:Alice@alice.example.com>;tag=alicetag2
  CSeq: 500 NOTIFY
  Contact: <sip:aliceinstance@alice.example.com>

F3
  SUBSCRIBE sip:bobinstance@bob.example.com SIP/2.0
  Event: presence
  Expires: 0
  Call-ID: alicecallid1@alice.example.com
  To: <sip:Bob@bob.example.com>;tag=bobtag2
  From: <sip:Alice@alice.example.com>;tag=alicetag2
  CSeq: 501 SUBSCRIBE
  Contact: <sip:aliceinstance@alice.example.com>

F4
  NOTIFY sip:bobinstance@bob.example.com SIP/2.0
  Event: presence
  Subscription-State: terminated;reason=deactivated
  Call-ID: alicecallid1@alice.example.com
  To: <sip:Bob@bob.example.com>;tag=bobtag2
  From: <sip:Alice@alice.example.com>;tag=alicetag2
  CSeq: 502 NOTIFY
  Contact: <sip:aliceinstance@alice.example.com>
```

Figure 4

4. Usage Creation and Destruction

Dialogs come into existence along with their first usage. Dialogs terminate when their last usage is destroyed. The messages that create and destroy usages vary per usage. This section provides a high-level categorization of those messages. The section does not attempt to explore the REGISTER pseudo-dialog.

4.1. Invite Usages

Created by: non-100 provisional responses to INVITE; 200 response to INVITE

Destroyed by: 200 responses to BYE; certain failure responses to INVITE, UPDATE, PRACK, INFO, or BYE; anything that destroys a dialog and all its usages

4.2. Subscribe usages

Created by: 200 class responses to SUBSCRIBE; 200 class responses to REFER; NOTIFY requests

Destroyed by: 200 class responses to NOTIFY-terminated; NOTIFY or refresh-SUBSCRIBE request timeout; certain failure responses to NOTIFY or SUBSCRIBE; expiration without refresh if network issues prevent the terminal NOTIFY from arriving; anything that destroys a dialog and all its usages

5. Proper Handling of Multiple Usages

The examples in Section 3 show straightforward cases where it is fairly obvious when the dialog begins and ends. Unfortunately, there are many scenarios where such clarity is not present. For instance, in Figure 1, what would it mean if the response to the NOTIFY (F2) were a 481? Does that simply terminate the refer subscription, or does it destroy the entire dialog? This section explores the problem areas with multiple usages that have been identified to date.

5.1. A Survey of the Effect of Failure Responses on Usages and Dialogs

For this survey, consider a subscribe usage inside a dialog established with an invite usage. Unless stated otherwise, we'll discuss the effect on each usage and the dialog when a client issuing a NOTIFY inside the subscribe usage receives a failure response (such as a transferee issuing a NOTIFY to event refer). Further, unless otherwise stated, the conclusions apply to arbitrary multiple usages. This survey is written from the perspective of a client receiving the

error response. The effect on dialogs and usages at the server issuing the response is the same.

3xx responses: Redirection mid-dialog is not well understood in SIP, but whatever effect it has impacts the entire dialog and all of its usages equally. In our example scenario, both the subscription and the invite usage would be redirected by this single response.

For the failure responses with code 400 and greater, there are three common ways the failure can affect the transaction, usage, and dialog state.

Transaction Only The error affects only the transaction, not the usage or dialog the transaction occurs in (beyond affecting the local CSeq). Any other usage of the dialog is unaffected. The error is a complaint about this transaction, not the usage or dialog that the transaction occurs in.

Destroys Usage The error destroys the usage, but not the dialog. Any other usages sharing this dialog are not affected.

Destroys Dialog The error destroys the dialog and all usages sharing it.

Table 1 and Table 2 display how the various codes affect transaction, usage, or dialog state. Response code specific comments or exceptions follow the table.

Transaction Only	Destroys Usage	Destroys Dialog
400 (or unknown 4xx)	405, 480	404, 410, 416
401, 402, 403, 406	481, 489	482, 483
407, 408, 412-415	501	484, 485
417, 420, 421, 422		502, 604
423, 428, 429		
436-438, 486, 487		
488, 491, 493, 494		
500 (or unknown 5xx)		
503, 504, 505		
513, 580		
600 (or unknown 6xx)		
603, 606		

Table 1

Code	Reason	Impact	Notes
400/4xx	Bad Request	Transaction	
401	Unauthorized	Transaction	
402	Payment Required	Transaction	(1)
403	Forbidden	Transaction	
404	Not Found	Dialog	(2)
405	Method Not Allowed	Usage	(3)
406	Not Acceptable	Transaction	
407	Proxy Authentication Required	Transaction	
408	Request Timeout	Transaction	(4)
410	Gone	Dialog	(2)
412	Conditional Request Failed	Transaction	
413	Request Entity Too Large	Transaction	
414	Request-URI Too Long	Transaction	
415	Unsupported Media Type	Transaction	
416	Unsupported URI Scheme	Dialog	(2)
417	Unknown Resource-Priority	Transaction	
420	Bad Extension	Transaction	
421	Extension Required	Transaction	
422	Session Interval Too Small	Transaction	(5)
423	Interval Too Brief	Transaction	
428	Use Identity Header	Transaction	
429	Provide Referrer Identity	Transaction	(6)
436	Bad Identity-Info	Transaction	
437	Unsupported Certificate	Transaction	
438	Invalid Identity Header	Transaction	
480	Temporarily Unavailable	Usage	(7)
481	Call/Transaction Does Not Exist	Usage	(8)
482	Loop Detected	Dialog	(9)
483	Too Many Hops	Dialog	(10)
484	Address Incomplete	Dialog	(2)
485	Ambiguous	Dialog	(2)
486	Busy Here	Transaction	(11)
487	Request Terminated	Transaction	
488	Not Acceptable Here	Transaction	
489	Bad Event	Usage	(12)
491	Request Pending	Transaction	
493	Undecipherable	Transaction	
494	Security Agreement Required	Transaction	
500/5xx	Server Internal Error	Transaction	(13)
501	Not Implemented	Usage	(3)
502	Bad Gateway	Dialog	(14)
503	Service Unavailable	Transaction	(15)
504	Server Time-Out	Transaction	(16)
505	Version Not Supported	Transaction	
513	Message Too Large	Transaction	

580	Precondition Failure	Transaction	
600/6xx	Busy Everywhere	Transaction	(17)
603	Decline	Transaction	
604	Does Not Exist Anywhere	Dialog	(2)
606	Not Acceptable	Transaction	

Table 2

(1) 402 Payment Required: This is a reserved response code. If encountered, it should be treated as an unrecognized 4xx.

(2) 404 Not Found:

410 Gone:

416 Unsupported URI Scheme:

484 Address Incomplete:

485 Ambiguous:

604 Does Not Exist Anywhere:

The Request-URI that is being rejected is the remote target set by the Contact provided by the peer. Getting this response means that something has gone fundamentally wrong with the dialog state.

(3) 405 Method Not Allowed:

501 Not Implemented:

Either of these responses would be aberrant in our example scenario since support for the NOTIFY method is required by the usage. In this case, the UA knows the condition is unrecoverable and should stop sending NOTIFYs on the usage. Any refresh subscriptions should be rejected. In general, these errors will affect at most the usage. If the request was not integral to the usage (it used an unknown method, or was an INFO inside an INVITE usage, for example), only the transaction will be affected.

(4) 408 Request Timeout: Receiving a 408 will have the same effect on usages and dialogs as a real transaction timeout as described in Section 5.2.

(5) 422 Session Interval Too Small: This response does not make sense for any mid-usage request. If it is received, an element in the path of the request is violating protocol, and the recipient should treat this as it would an unknown 4xx response.

(6) 429 Provide Referrer Identity: This response won't be returned to a NOTIFY as in our example scenario, but when it is returned to a REFER, it is objecting only to the REFER request itself.

(7) 480 Temporarily Unavailable: RFC 3261 is unclear on what this response means for mid-usage requests. Future updates to that specification are expected to clarify that this response affects only the usage in which the request occurs. No other usages are affected. If the response included a Retry-After header field, further requests in that usage should not be sent until the indicated time has past. Requests in other usages may still be sent at any time.

(8) 481 Call/Transaction Does Not Exist: This response indicates that the peer has lost its copy of the dialog usage state. The dialog itself should not be destroyed unless this was the last usage.

The effects of a 481 on a dialog and its usages are the most ambiguous of any final response. There are implementations that have chosen the meaning recommended here, and others that destroy the entire dialog without regard to the number of outstanding usages. Going forward with this clarification will allow those deployed implementations that assumed only the usage was destroyed to work with a wider number of implementations. Existing implementations that destroy all other usages in the dialog will continue to function as they do now, except that peers following the recommendation will attempt to do things with the other usages and this element will return 481s for each of them until they are all gone. However, the necessary clarification to RFC 3261 needs to make it very clear that the ability to terminate usages independently from the overall dialog using a 481 is not justification for designing new applications that count on multiple usages in a dialog.

The 481 response to a CANCEL request has to be treated differently. For CANCEL, a 481 means the UAS can't find a matching transaction. A 481 response to a CANCEL affects only the CANCEL transaction. The usage associated with the INVITE is not affected.

(9) 482 Loop Detected: This response is aberrant mid-dialog. It will only occur if the Record-Route header field were improperly constructed by the proxies involved in setting up the dialog's initial usage, or if a mid-dialog request forks and merges (which should never happen). Future requests using this dialog state will also fail.

An edge condition exists during RFC 3263 failover at the element sending a request, where the request effectively forks to multiple destinations from the client. Some implementations increase risk entering this edge condition by trying the next potential location as determined by RFC 3263 very rapidly if the first does not immediately respond. In any situation where a client sends the same request to more than one endpoint, it must be prepared to receive a response from each branch (and should choose a "best" response to act on following the same guidelines as a forking proxy). In this particular race condition, if multiple branches respond, all but one will most likely return a 482 Merged Request. The client should select the remaining non-482 response as the "best" response.

(10) 483 Too Many Hops: Similar to 482, receiving this mid-dialog is aberrant. Unlike 482, recovery may be possible by increasing Max-Forwards (assuming that the requester did something strange like using a smaller value for Max-Forwards in mid-dialog requests than it used for an initial request). If the request isn't tried with an increased Max-Forwards, then the agent should follow the Destroy Dialog actions.

(11) 486 Busy Here: This response is nonsensical in our example scenario, or in any scenario where this response comes inside an established usage. If it occurs in that context, it should be treated as an unknown 4xx response.

(12) 489 Bad Event: In our example scenario, [5] declares that the subscription usage in which the NOTIFY is sent is terminated. This response is only valid in the context of SUBSCRIBE and NOTIFY. UAC behavior for receiving this response to other methods is not specified, but treating it as an unknown 4xx is a reasonable practice.

(13) 500 and 5xx unrecognized responses: If the response contains a Retry-After header field value, the server thinks the condition is temporary, and the request can be retried after the indicated interval. If the response does not contain a Retry-After header field value, the UA may decide to retry after an interval of its choosing or attempt to gracefully terminate the usage. Whether or not to terminate other usages depends on the application. If the

UA receives a 500 (or unrecognized 5xx) in response to an attempt to gracefully terminate this usage, it can treat this usage as terminated. If this is the last usage sharing the dialog, the dialog is also terminated.

(14) 502 Bad Gateway: This response is aberrant mid-dialog. It will only occur if the Record-Route header field were improperly constructed by the proxies involved in setting up the dialog's initial usage. Future requests using this dialog state will also fail.

(15) 503 Service Unavailable: As per [6], the logic handling locating SIP servers for transactions may handle 503 requests (effectively, sequentially forking at the endpoint based on DNS results). If this process does not yield a better response, a 503 may be returned to the transaction user. Like a 500 response, the error is a complaint about this transaction, not the usage. Because this response occurred in the context of an established usage (hence an existing dialog), the route-set has already been formed and any opportunity to try alternate servers (as recommended in [1]) has been exhausted by the RFC3263 logic.

(16) 504 Server Time-out: It is not obvious under what circumstances this response would be returned to a request in an existing dialog.

(17) 600 and 6xx unrecognized responses: Unlike 400 Bad Request, a 600 response code says something about the recipient user, not the request that was made. This end user is stating an unwillingness to communicate. If the response contains a Retry-After header field value, the user is indicating willingness to communicate later and the request can be retried after the indicated interval. This usage, and any other usages sharing the dialog are unaffected. If the response does not contain a Retry-After header field value, the UA may decide to retry after an interval of its choosing or attempt to gracefully terminate the usage. Whether or not to terminate other usages depends on the application. If the UA receives a 600 (or unrecognized 6xx) in response to an attempt to gracefully terminate this usage, it can treat this usage as terminated. If this is the last usage sharing the dialog, the dialog is also terminated.

5.2. Transaction Timeouts

[1] states that a UAC should terminate a dialog (by sending a BYE) if no response is received for a request sent within a dialog. This recommendation should have been limited to the invite usage instead of the whole dialog. [5] states that a timeout for a NOTIFY removes a

subscription, but a SUBSCRIBE that fails with anything other than a 481 does not. Given these statements, it is unclear whether a refresh SUBSCRIBE issued in a dialog shared with an invite usage destroys either usage or the dialog if it times out.

Generally, a transaction timeout should affect only the usage in which the transaction occurred. Other uses sharing the dialog should not be affected. In the worst case of timeout due to total transport failure, it may require multiple failed messages to remove all usages from a dialog (at least one per usage).

There are some mid-dialog messages that never belong to any usage. If they timeout, they will have no effect on the dialog or its usages.

5.3. Matching Requests to Usages

For many mid-dialog requests, identifying the usage they belong to is obvious. A dialog can have at most one invite usage, so any INVITE, UPDATE, PRACK, ACK, CANCEL, BYE, or INFO requests belong to it. The usage (i.e. the particular subscription) SUBSCRIBE, NOTIFY, and REFER requests belong to can be determined from the Event header field of the request. REGISTER requests within a (pseudo)-dialog belong to the registration usage. (As mentioned before, implementations aren't mixing registration usages with other usages, so this document isn't exploring the consequences of that bad behavior).

According to [1], "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 that dialog". Thus, OPTIONS does not belong to any usage. Only those failures discussed in Section 5.1 and Section 5.2 that destroy entire dialogs will have any effect on the usages sharing the dialog with a failed OPTIONS request.

MESSAGE requests are discouraged inside a dialog. Implementations are restricted from creating a usage for the purpose of carrying a sequence of MESSAGE requests (though some implementations use it that way, against the standard recommendation). A failed MESSAGE occurring inside an existing dialog will have similar effects on the dialog and its usages as a failed OPTIONS request.

Mid-dialog requests with unknown methods cannot be matched with a usage. Servers will return a failure response (likely a 501). The effect on the dialog and its usages at either the client or the server should be similar to that of a failed OPTIONS request.

These guidelines for matching messages to usages (or determining there is no usage) apply equally when acting as a UAS, a UAC, or any third party tracking usage and dialog state by inspecting all messages between two endpoints.

5.4. Target Refresh Requests

Target refresh requests update the remote target of a dialog when they are successfully processed. The currently defined target refresh requests are INVITE, UPDATE, SUBSCRIBE, NOTIFY, and REFER [7]).

The remote target is part of the dialog state. When a target refresh request affects it, it affects it for ALL usages sharing that dialog. If a subscription and invite usage are sharing a dialog, sending a refresh SUBSCRIBE with a different contact will cause reINVITEs from the peer to go to that different contact.

A UAS will only update the remote target if it sends a 200 class response to a target refresh request. A UAC will only update the remote target if it receives a 200 class response to a target refresh request. Again, any update to a dialog's remote target affects all usages of that dialog.

There is known ambiguity around the effects of provisional responses on remote targets that a future specification will attempt to clarify. Furthermore, because the remote target is part of the dialog state, not any usage state, there is ambiguity in having target refresh requests in progress simultaneously on multiple usages in the same dialog. Implementation designers should consider these conditions with care.

5.5. Refreshing and Terminating Usages

Subscription and registration usages expire over time and must be refreshed (with a refresh SUBSCRIBE, for example). This expiration is usage state, not dialog state. If several subscriptions share a dialog, refreshing one of them has no effect on the expiration of the others.

Normal termination of a usage has no effect on other usages sharing the same dialog. For instance, terminating a subscription with a NOTIFY/Subscription-State: terminated will not terminate an invite usage sharing its dialog. Likewise, ending an invite usage with a BYE does not terminate any active Event: refer subscriptions established on that dialog.

5.6. Refusing New Usages

As the survey of the effect of failure responses shows, care must be taken when refusing a new usage inside an existing dialog. Choosing the wrong response code will terminate the dialog and all of its usages. Generally, returning a 603 Decline is the safest way to refuse a new usage.

5.7. Replacing Usages

[8] defines a mechanism through which one usage can replace another. It can be used, for example, to associate the two dialogs in which a transfer target is involved during an attended transfer. It is written using the term "dialog", but its intent was only to affect the invite usage of the dialog it targets. Any other usages inside that dialog are unaffected. For some applications, the other usages may no longer make sense, and the application may terminate them as well.

However, the interactions between Replaces and multiple dialog usages have not been well explored. More discussion of this topic is needed. Implementers should avoid this scenario completely.

6. Avoiding Multiple Usages

Processing multiple usages correctly is not completely understood. What is understood is difficult to implement and is very likely to lead to interoperability problems. The best way to avoid the trouble that comes with such complexity is to avoid it altogether.

When designing new applications or features that use SIP dialogs, do not require endpoints to construct multiple usages to participate in the application or use the feature. When designing endpoints, address the existing multiple usage scenarios as best as possible. Outside those scenarios, if a peer attempts to create a second usage inside a dialog, refuse it.

Unfortunately, there are existing applications, like transfer, that currently entail multiple usages, so the simple solution of "don't do it" will require some transitional work. This section looks at the pressures that led to these existing multiple usages and suggests alternatives.

When executing a transfer, the transferor and transferee currently share an invite usage and a subscription usage within the dialog between them. This is a result of sending the REFER request within the dialog established by the invite usage. Implementations were led to this behavior by these primary problems:

1. There was no way to ensure that a REFER on a new dialog would reach the particular endpoint involved in a transfer. Many factors, including details of implementations and changes in proxy routing between an INVITE and a REFER could cause the REFER to be sent to the wrong place. Sending the REFER down the existing dialog ensured it got to the same endpoint with which the dialog was established.

2. It was unclear how to associate an existing invite usage with a REFER arriving on a new dialog, where it was completely obvious what the association was when the REFER came on the invite usage's dialog.

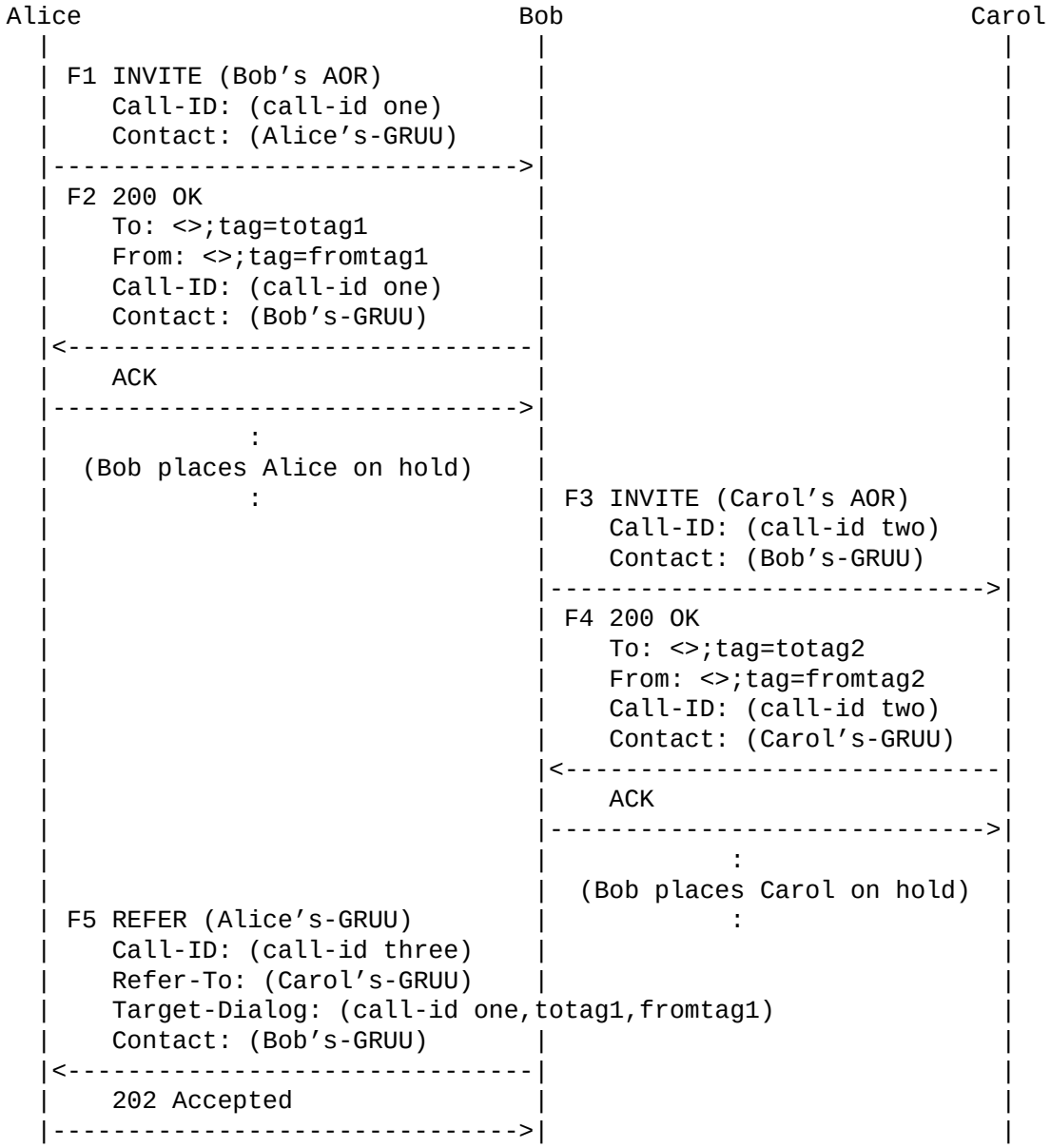
3. There were concerns with authorizing out-of-dialog REFERs. The authorization policy for REFER in most implementations piggybacks on the authorization policy for INVITE (which is, in most cases, based simply on "I placed or answered this call").

Globally Routable User Agent (UA) URIs (GRUUs) [9] have been defined specifically to address problem 1 by providing a URI that will reach one specific user-agent. The Target-Dialog header field [10] was created to address problems 2 and 3. This header field allows a request to indicate the dialog identifiers of some other dialog, providing association with the other dialog that can be used in an authorization decision.

The Join [11] and Replaces [8] mechanisms can also be used to address problem 1. When using this technique, a new request is sent outside any dialog with the expectation that it will fork to possibly many endpoints, including the one we're interested in. This request contains a header field listing the dialog identifiers of a dialog in progress. Only the endpoint holding a dialog matching those identifiers will accept the request. The other endpoints the request may have forked to will respond with an error. This mechanism is reasonably robust, failing only when the routing logic for out-of-dialog requests changes such that the new request does not arrive at the endpoint holding the dialog of interest.

The reachability aspects of using a GRUU to address problem 1 can be combined with the association-with-other-dialogs aspects of the Join/Replaces and Target-Dialog mechanisms. A REFER request sent out-of-dialog can be sent towards a GRUU, and identify an existing dialog as part of the context the receiver should use. The Target-Dialog header field can be included in the REFER listing the dialog this REFER is associated with. Figure 5 sketches how this could be used to achieve transfer without reusing a dialog. For simplicity, the diagram and message details do not show the server at example.com

that will be involved in routing the GRUU. Refer to [9] for those details.



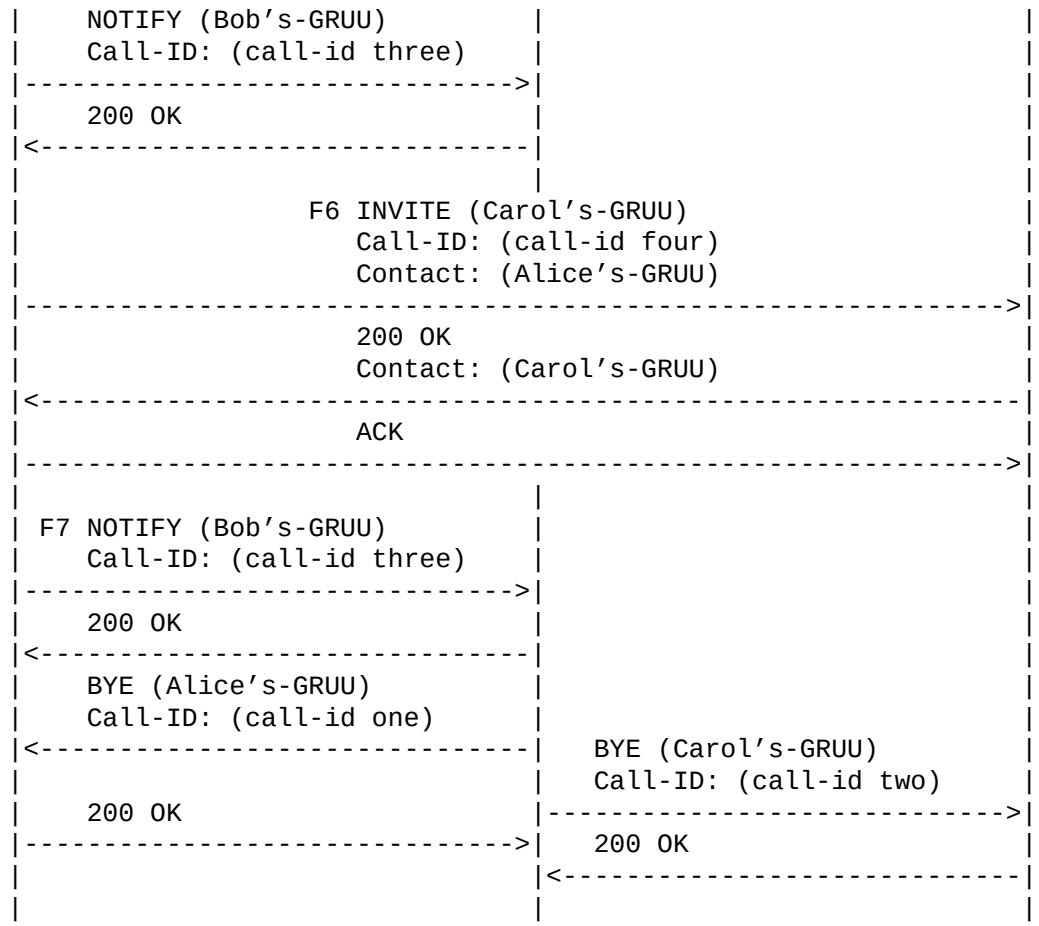


Figure 5: Transfer without dialog reuse

In message F1, Alice invites Bob indicating support for GRUUs (and offering a GRUU for herself):

Message F1 (abridged, detailing pertinent fields)

```
INVITE sip:bob@example.com SIP/2.0
Call-ID: 13jfdwer230jsdw@alice.example.com
Supported: gruu
Contact: <sip:alice@example.com;gr=urn:uuid:(Alice's UA's bits)>
```

Message F2 carries Bob's GRUU to Alice.

Message F2 (abridged, detailing pertinent fields)

```
SIP/2.0 200 OK
Supported: gruu
To: <sip:bob@example.com>;tag=totag1
From: <sip:alice@example.com>;tag=fromtag1
Contact: <sip:bob@example.com;gr=urn:uuid:(Bob's UA's bits)>
```

Bob decides to try to transfer Alice to Carol, so he puts Alice on hold and sends an INVITE to Carol. Carol and Bob negotiate GRUU support similar to what happened in F1 and F2.

Message F3 (abridged, detailing pertinent fields)

```
INVITE sip:carol@example.com SIP/2.0
Supported: gruu
Call-ID: 23rasdnfoa39i4jnasdf@bob.example.com
Contact: <sip:bob@example.com;gr=urn:uuid:(Bob's UA's bits)>
```

Message F4 (abridged, detailing pertinent fields)

```
SIP/2.0 200 OK
Supported: gruu
To: <sip:carol@example.com>;tag=totag2
From: <sip:bob@example.com>;tag=fromtag2
Call-ID: 23rasdnfoa39i4jnasdf@bob.example.com
Contact: <sip:carol@example.com;gr=urn:uuid:(Carol's UA's bits)>
```

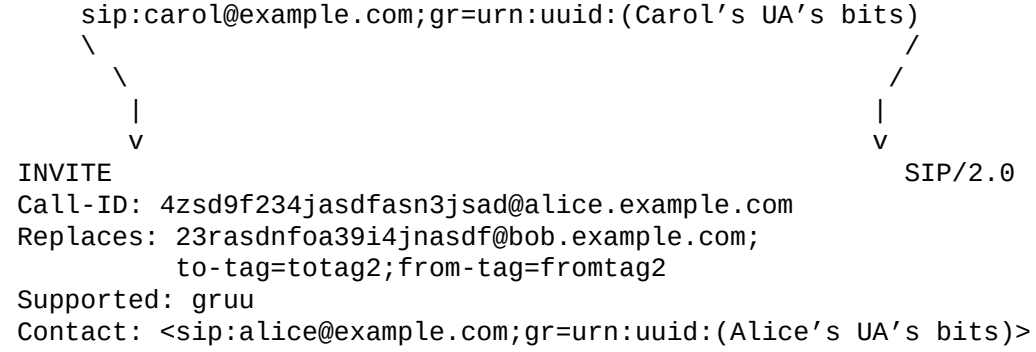
After consulting Carol, Bob places her on hold and refers Alice to her using message F5. Notice that the Refer-To URI is Carol's GRUU, and that this is on a different Call-ID than message F1. (The URI in the Refer-To header is line-broken for readability in this document; it would not be valid to break the URI this way in a real message.)

Message F5 (abridged, detailing pertinent fields)

```
REFER sip:aanewmr203raswdf@example.com SIP/2.0
Call-ID: 39fa99r0329493asdsf3n@bob.example.com
Refer-To: <sip:carol@example.com;g=urn:uid:(Carol's UA's bits)
           ?Replaces=23rasdnfoa39i4jnasdf@bob.example.com;
            to-tag=totag2;from-tag=fromtag2>
Target-Dialog: 13jfdwer230jsdw@alice.example.com;
               local-tag=fromtag1;remote-tag=totag1
Supported: gruu
Contact: <sip:bob@example.com;gr=urn:uuid:(Bob's UA's bits)>
```

Alice uses the information in the Target-Dialog header field to determine that this REFER is associated with the dialog she already has in place with Bob. Alice is now in a position to use the same admission policy she used for in-dialog REFERs: "Do I have a call with this person?". She accepts the REFER, sends Bob the obligatory immediate NOTIFY, and proceeds to INVITE Carol with message F6.

Message F6 (abridged, detailing pertinent fields)



Carol accepts Alice's invitation to replace her dialog (invite usage) with Bob, and notifies him that the REFERenced INVITE succeeded with F7:

Message F7 (abridged, detailing pertinent fields)

```
NOTIFY sip:boaiidfjjereis@example.com SIP/2.0
Subscription-State: terminated;reason=noresource
Call-ID: 39fa99r0329493asdsf3n@bob.example.com
Contact: <sip:alice@example.com;gr=urn:uuid:(Alice's UA's bits)>
Content-Type: message/sipfrag

SIP/2.0 200 OK
```

Bob then ends his invite usages with both Alice and Carol using BYEs.

7. Security Considerations

Handling multiple usages within a single dialog is complex and introduces scenarios where the right thing to do is not clear. The ambiguities described here can result in unexpected disruption of communication if response codes are chosen carelessly. Furthermore, these ambiguities could be exploited, particularly by third-parties injecting unauthenticated requests or inappropriate responses. Implementations choosing to create or accept multiple usages within a dialog should give extra attention to the security considerations in

[1], especially those concerning the authenticity of requests and processing of responses.

Service implementations should carefully consider the effects on their service of peers making different choices in these areas of ambiguity. A service that requires multiple usages needs to pay particular attention to the effect on service and network utilization when a client fails to destroy a dialog the service believes should be destroyed. A service that disallows multiple usages should consider the effect on clients that, for instance, destroy the entire dialog when only a usage should be torn down. In the worst case of a service deployed into a network with a large number of misbehaving clients trying to create multiple usages in an automated fashion, a retry storm similar to an avalanche restart could be induced.

8. Conclusion

Handling multiple usages within a single dialog is complex and introduces scenarios where the right thing to do is not clear. Implementations should avoid entering into multiple usages whenever possible. New applications should be designed to never introduce multiple usages.

There are some accepted SIP practices, including transfer, that currently require multiple usages. Recent work, most notably GRUU, makes those practices unnecessary. The standardization of those practices and the implementations should be revised as soon as possible to use only single-usage dialogs.

9. Acknowledgments

The ideas in this document have been refined over several IETF meetings with many participants. Significant contribution was provided by Adam Roach, Alan Johnston, Ben Campbell, Cullen Jennings, Jonathan Rosenberg, Paul Kyzivat, and Rohan Mahy. Members of the reSIProcate project also shared their difficulties and discoveries while implementing multiple-usage dialog handlers.

10. Informative References

[1] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[2] Levin, O., "Suppression of Session Initiation Protocol (SIP) REFER Method Implicit Subscription", RFC 4488, May 2006.

[3] Burger, E. and M. Dolly, "A Session Initiation Protocol (SIP) Event Package for Key Press Stimulus (KPML)", RFC 4730, November 2006.

[4] Niemi, A., "Session Initiation Protocol (SIP) Extension for Event State Publication", RFC 3903, October 2004.

[5] Roach, A., "Session Initiation Protocol (SIP)-Specific Event Notification", RFC 3265, June 2002.

[6] Rosenberg, J. and H. Schulzrinne, "Session Initiation Protocol (SIP): Locating SIP Servers", RFC 3263, June 2002.

[7] Sparks, R., "The Session Initiation Protocol (SIP) Refer Method", RFC 3515, April 2003.

[8] Mahy, R., Biggs, B., and R. Dean, "The Session Initiation Protocol (SIP) "Replaces" Header", RFC 3891, September 2004.

[9] Rosenberg, J., "Obtaining and Using Globally Routable User Agent (UA) URIs (GRUU) in the Session Initiation Protocol (SIP)", Work in Progress, June 2006.

[10] Rosenberg, J., "Request Authorization through Dialog Identification in the Session Initiation Protocol (SIP)", RFC 4538, June 2006.

[11] Mahy, R. and D. Petrie, "The Session Initiation Protocol (SIP) "Join" Header", RFC 3911, October 2004.

Author's Address

Robert J. Sparks
Estacado Systems

EMail: RjS@estacado.net

Full Copyright Statement

Intellectual Property