

SIP-Specific Event Notification
draft-roach-sipcore-rfc3265bis-00

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/1id-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on September 4, 2009.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process.

Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Abstract

This document describes an extension to the Session Initiation Protocol (SIP). The purpose of this extension is to provide an extensible framework by which SIP nodes can request notification from remote nodes indicating that certain events have occurred.

Note that the event notification mechanisms defined herein are NOT intended to be a general-purpose infrastructure for all classes of event subscription and notification.

Table of Contents

1.	Introduction	6
1.1.	Overview of Operation	6
1.2.	Documentation Conventions	7
2.	Definitions	7
3.	SIP Methods for Event Notification	8
3.1.	SUBSCRIBE	8
3.1.1.	Subscription Duration	8
3.1.2.	Identification of Subscribed Events and Event Classes	9
3.1.3.	Additional SUBSCRIBE Header Values	10
3.2.	NOTIFY	10
3.2.1.	Identification of Reported Events, Event Classes, and Current State	10
4.	Node Behavior	11
4.1.	Subscriber Behavior	11
4.1.1.	Detecting Support for SIP Events	11
4.1.2.	Creating and Maintaining Subscriptions	11
4.1.3.	Receiving and Processing State Information	14
4.1.4.	Forking of SUBSCRIBE Messages	17
4.2.	Notifier Behavior	17
4.2.1.	Subscription Establishment and Maintenance	17
4.2.2.	Sending State Information to Subscribers	21
4.2.3.	PINT Compatibility	23
4.3.	Proxy Behavior	23
4.4.	Common Behavior	24
4.4.1.	Dialog Creation and Termination	24
4.4.2.	Notifier Migration	24
4.4.3.	Polling Resource State	25
4.4.4.	Allow-Events header field usage	25
4.5.	Targeting Subscriptions at Devices	26
4.5.1.	Using GRUUs to Route to Devices	26
4.5.2.	Sharing Dialogs	27
4.6.	CANCEL Requests for SUBSCRIBE and NOTIFY	28
5.	Event Packages	28
5.1.	Appropriateness of Usage	29
5.2.	Event Template-packages	29
5.3.	Amount of State to be Conveyed	30
5.3.1.	Complete State Information	30
5.3.2.	State Deltas	30
5.4.	Event Package Responsibilities	31
5.4.1.	Event Package Name	31
5.4.2.	Event Package Parameters	31
5.4.3.	SUBSCRIBE Bodies	31
5.4.4.	Subscription Duration	32
5.4.5.	NOTIFY Bodies	32
5.4.6.	Notifier processing of SUBSCRIBE requests	32

Roach

Expires September 4, 2009

[Page 3]

5.4.7.	Notifier generation of NOTIFY requests	32
5.4.8.	Subscriber processing of NOTIFY requests	33
5.4.9.	Handling of forked requests	33
5.4.10.	Rate of notifications	33
5.4.11.	State Aggregation	34
5.4.12.	Examples	34
5.4.13.	Use of URIs to Retrieve State	34
6.	Security Considerations	34
6.1.	Access Control	34
6.2.	Notifier Privacy Mechanism	35
6.3.	Denial-of-Service attacks	35
6.4.	Replay Attacks	35
6.5.	Man-in-the middle attacks	36
6.6.	Confidentiality	36
7.	IANA Considerations	36
7.1.	Event Packages	37
7.1.1.	Registration Information	37
7.1.2.	Registration Template	38
7.2.	Reason Codes	38
7.3.	Header Field Names	39
7.4.	Response Codes	40
8.	Syntax	40
8.1.	New Methods	40
8.1.1.	SUBSCRIBE method	42
8.1.2.	NOTIFY method	42
8.2.	New Header Fields	42
8.2.1.	"Event" Header Field	42
8.2.2.	"Allow-Events" Header Field	43
8.2.3.	"Subscription-State" Header Field	43
8.3.	New Response Codes	43
8.3.1.	"202 Accepted" Response Code	43
8.3.2.	"489 Bad Event" Response Code	43
8.4.	Augmented BNF Definitions	43
9.	References	44
9.1.	Normative References	44
9.2.	Informative References	45
Appendix A.	Acknowledgements	46
Appendix B.	Open Issues	46
B.1.	Bug 711: Allow-Events can't express template support	46
B.2.	Remove 202 Response Code?	47
B.3.	Timer L and Resubscribes	47
Appendix C.	Changes	47
C.1.	Changes since RFC 3265	47
C.1.1.	Bug 666: Clarify use of expires=xxx with terminated	47
C.1.2.	Bug 667: Reason code for unsub/poll not clearly spelled out	47
C.1.3.	Bug 669: Clarify: SUBSCRIBE for a duration might be answered with a NOTIFY/expires=0	48

Roach

Expires September 4, 2009

[Page 4]

C.1.4.	Bug 670: Dialog State Machine needs clarification . .	48
C.1.5.	Bug 671: Clarify timeout-based removal of subscriptions	48
C.1.6.	Bug 672: Mandate expires= in NOTIFY	48
C.1.7.	Bug 673: INVITE 481 response effect clarification . .	48
C.1.8.	Bug 677: SUBSCRIBE response matching text in error . .	48
C.1.9.	Bug 695: Document is not explicit about response to NOTIFY at subscription termination	48
C.1.10.	Bug 696: Subscription state machine needs clarification	48
C.1.11.	Bug 697: Unsubscription behavior could be clarified .	49
C.1.12.	Bug 699: NOTIFY and SUBSCRIBE are target refresh requests	49
C.1.13.	Bug 722: Inconsistent 423 reason phrase text	49
C.1.14.	Bug 741: guidance needed on when to not include Allow-Events	49
C.1.15.	Bug 744: 5xx to NOTIFY terminates a subscription, but should not	49
C.1.16.	Bug 752: Detection of forked requests is incorrect . .	49
C.1.17.	Bug 773: Reason code needs IANA registry	49
C.1.18.	Bug 774: Need new reason for terminating subscriptions to resources that never change	49
C.1.19.	Clarify handling of Route/Record-Route in NOTIFY . . .	50
C.1.20.	Eliminate implicit subscriptions	50
C.1.21.	Deprecate dialog re-use	50
C.1.22.	Rationalize dialog creation	50
C.1.23.	Refactor behavior sections	50
C.1.24.	Clarify sections that need to be present in event packages	50
C.1.25.	Make CANCEL handling more explicit	50
C.1.26.	Remove State Agent Terminology	51
Author's Address		51

1. Introduction

The ability to request asynchronous notification of events proves useful in many types of SIP services for which cooperation between end-nodes is required. Examples of such services include automatic callback services (based on terminal state events), buddy lists (based on user presence events), message waiting indications (based on mailbox state change events), and PSTN and Internet Internetworking (PINT) [[RFC2848](#)] status (based on call state events).

The methods described in this document provide a framework by which notification of these events can be ordered.

The event notification mechanisms defined herein are NOT intended to be a general-purpose infrastructure for all classes of event subscription and notification. Meeting requirements for the general problem set of subscription and notification is far too complex for a single protocol. Our goal is to provide a SIP-specific framework for event notification which is not so complex as to be unusable for simple features, but which is still flexible enough to provide powerful services. Note, however, that event packages based on this framework may define arbitrarily elaborate rules which govern the subscription and notification for the events or classes of events they describe.

This document does not describe an extension which may be used directly; it must be extended by other documents (herein referred to as "event packages"). In object-oriented design terminology, it may be thought of as an abstract base class which must be derived into an instantiatable class by further extensions. Guidelines for creating these extensions are described in [Section 5](#).

1.1. Overview of Operation

The general concept is that entities in the network can subscribe to resource or call state for various resources or calls in the network, and those entities (or entities acting on their behalf) can send notifications when those states change.

A typical flow of messages would be:

Subscriber	Notifier
-----SUBSCRIBE----->	Request state subscription
<-----200-----	Acknowledge subscription
<-----NOTIFY-----	Return current state information
-----200----->	
<-----NOTIFY-----	Return current state information
-----200----->	

Subscriptions are expired and must be refreshed by subsequent SUBSCRIBE messages.

1.2. Documentation Conventions

There are several paragraphs throughout this document which provide motivational or clarifying text. Such passages are non-normative, and are provided only to assist with reader comprehension. These passages are set off from the remainder of the text by being indented thus:

This is an example of non-normative explanatory text. It does not form part of the specification, and is used only for clarification.

The all-capital terms "MUST", "SHOULD", "MAY", "SHOULD NOT", "MUST NOT", and "RECOMMENDED" are used as defined in [[RFC2119](#)]. In particular, implementors need to take careful note of the meaning of "SHOULD" defined in [RFC 2119](#). To rephrase: violation of SHOULD-strength requirements requires careful analysis and clearly enumerable reasons. It is inappropriate to fail to comply with "SHOULD"-strength requirements whimsically or for ease of implementation.

The use of quotation marks next to periods and commas follows the convention used by the American Mathematical Society; although contrary to traditional American English convention, this usage lends clarity to certain passages.

2. Definitions

Event Package: An event package is an additional specification which defines a set of state information to be reported by a notifier to a subscriber. Event packages also define further syntax and semantics based on the framework defined by this document required to convey such state information.

Event Template-Package: An event template-package is a special kind of event package which defines a set of states which may be applied to all possible event packages, including itself.

Notification: Notification is the act of a notifier sending a NOTIFY message to a subscriber to inform the subscriber of the state of a resource.

Notifier: A notifier is a user agent which generates NOTIFY requests for the purpose of notifying subscribers of the state of a resource. Notifiers typically also accept SUBSCRIBE requests to create subscriptions.

Subscriber: A subscriber is a user agent which receives NOTIFY requests from notifiers; these NOTIFY requests contain information about the state of a resource in which the subscriber is interested. Subscribers typically also generate SUBSCRIBE requests and send them to notifiers to create subscriptions.

Subscription: A subscription is a set of application state associated with a dialog. This application state includes a pointer to the associated dialog, the event package name, and possibly an identification token. Event packages will define additional subscription state information. By definition, subscriptions exist in both a subscriber and a notifier.

Subscription Migration: Subscription migration is the act of moving a subscription from one notifier to another notifier.

3. SIP Methods for Event Notification

3.1. SUBSCRIBE

The SUBSCRIBE method is used to request current state and state updates from a remote node. SUBSCRIBE is a target refresh request, as that term is defined in SIP [[RFC3261](#)].

3.1.1. Subscription Duration

SUBSCRIBE requests SHOULD contain an "Expires" header field (defined in SIP [[RFC3261](#)]). This expires value indicates the duration of the subscription. In order to keep subscriptions effective beyond the duration communicated in the "Expires" header field, subscribers need to refresh subscriptions on a periodic basis using a new SUBSCRIBE message on the same dialog as defined in SIP [[RFC3261](#)].

If no "Expires" header field is present in a SUBSCRIBE request, the implied default is defined by the event package being used.

200-class responses to SUBSCRIBE requests also MUST contain an "Expires" header field. The period of time in the response MAY be shorter but MUST NOT be longer than specified in the request. The notifier is explicitly allowed to shorten the duration to zero. The period of time in the response is the one which defines the duration of the subscription.

An "expires" parameter on the "Contact" header field has no semantics for SUBSCRIBE and is explicitly not equivalent to an "Expires" header field in a SUBSCRIBE request or response.

A natural consequence of this scheme is that a SUBSCRIBE with an "Expires" of 0 constitutes a request to unsubscribe from an event.

In addition to being a request to unsubscribe, a SUBSCRIBE message with "Expires" of 0 also causes a fetch of state; see [Section 4.4.3](#).

Notifiers may also wish to cancel subscriptions to events; this is useful, for example, when the resource to which a subscription refers is no longer available. Further details on this mechanism are discussed in [Section 4.2.2](#).

[3.1.2](#). Identification of Subscribed Events and Event Classes

Identification of events is provided by three pieces of information: Request URI, Event Type, and (optionally) message body.

The Request URI of a SUBSCRIBE request, most importantly, contains enough information to route the request to the appropriate entity per the request routing procedures outlined in SIP [[RFC3261](#)]. It also contains enough information to identify the resource for which event notification is desired, but not necessarily enough information to uniquely identify the nature of the event (e.g., "sip:adam@example.com" would be an appropriate URI to subscribe to for my presence state; it would also be an appropriate URI to subscribe to the state of my voice mailbox).

Subscribers MUST include exactly one "Event" header field in SUBSCRIBE requests, indicating to which event or class of events they are subscribing. The "Event" header field will contain a token which indicates the type of state for which a subscription is being requested. This token will be registered with the IANA and will correspond to an event package which further describes the semantics of the event or event class.

If the event package to which the event token corresponds defines behavior associated with the body of its SUBSCRIBE requests, those semantics apply.

Event packages may also define parameters for the Event header field; if they do so, they must define the semantics for such parameters.

3.1.3. Additional SUBSCRIBE Header Values

Because SUBSCRIBE requests create a dialog as defined in SIP [[RFC3261](#)], they MAY contain an "Accept" header field. This header field, if present, indicates the body formats allowed in subsequent NOTIFY requests. Event packages MUST define the behavior for SUBSCRIBE requests without "Accept" header fields; usually, this will connote a single, default body type.

Header values not described in this document are to be interpreted as described in SIP [[RFC3261](#)].

3.2. NOTIFY

NOTIFY messages are sent to inform subscribers of changes in state to which the subscriber has a subscription. Subscriptions are typically put in place using the SUBSCRIBE method; however, it is possible that other means have been used.

NOTIFY is a target refresh request, as that term is defined in SIP [[RFC3261](#)].

A NOTIFY does not terminate its corresponding subscription; in other words, a single SUBSCRIBE request may trigger several NOTIFY requests.

3.2.1. Identification of Reported Events, Event Classes, and Current State

Identification of events being reported in a notification is very similar to that described for subscription to events (see [Section 3.1.2](#)).

As in SUBSCRIBE requests, NOTIFY "Event" header fields will contain a single event package name for which a notification is being generated. The package name in the "Event" header field MUST match the "Event" header field in the corresponding SUBSCRIBE message.

Event packages may define semantics associated with the body of their NOTIFY requests; if they do so, those semantics apply. NOTIFY bodies are expected to provide additional details about the nature of the event which has occurred and the resultant resource state.

When present, the body of the NOTIFY request MUST be formatted into one of the body formats specified in the "Accept" header field of the corresponding SUBSCRIBE request. This body will contain either the state of the subscribed resource or a pointer to such state in the form of a URI (see [Section 5.4.13](#)).

4. Node Behavior

4.1. Subscriber Behavior

4.1.1. Detecting Support for SIP Events

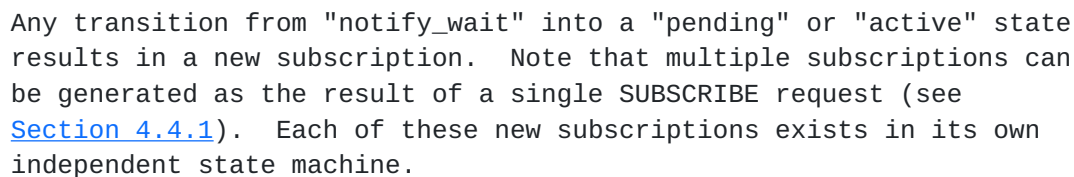
The extension described in this document does not make use of the use of "Require" or "Proxy-Require" header fields; similarly, there is no token defined for "Supported" header fields. Potential subscribers may probe for the support of SIP Events using the OPTIONS request defined in SIP [[RFC3261](#)].

The presence of "SUBSCRIBE" in the "Allow" header field of any request or response indicates support for SIP Events; further, in the absence of an "Allow" header field, the simple presence of an "Allow-Events" header field is sufficient to indicate that the node that sent the message is capable of acting as a notifier (see [Section 4.4.4](#)).

The "methods" parameter for Contact may also be used to specifically announce support for SUBSCRIBE and NOTIFY messages when registering. (See [[RFC3840](#)] for details on the "methods" parameter).

4.1.2. Creating and Maintaining Subscriptions

From the subscriber's perspective, a subscription proceeds according to the following state diagram:



When a subscriber wishes to subscribe to a particular state for a resource, it forms a SUBSCRIBE message. If the initial SUBSCRIBE represents a request outside of a dialog (as it typically will), its construction follows the procedures outlined in SIP [[RFC3261](#)] for UAC

Roach

Expires September 4, 2009

[Page 12]

request generation outside of a dialog.

This SUBSCRIBE request will be confirmed with a final response. 200-class responses indicate that the subscription has been accepted, and that a NOTIFY will be sent immediately. A 200 response indicates that the subscription has been accepted and that the user is authorized to subscribe to the requested resource. A 202 response merely indicates that the subscription has been understood, and that authorization may or may not have been granted.

The "Expires" header field in a 200-class response to SUBSCRIBE indicates the actual duration for which the subscription will remain active (unless refreshed).

Non-200 class final responses indicate that no subscription or dialog has been created, and no subsequent NOTIFY message will be sent. All non-200 class responses (with the exception of "489", described herein) have the same meanings and handling as described in SIP [[RFC3261](#)].

4.1.2.2. Refreshing of Subscriptions

At any time before a subscription expires, the subscriber may refresh the timer on such a subscription by sending another SUBSCRIBE request on the same dialog as the existing subscription. The handling for such a request is the same as for the initial creation of a subscription except as described below.

If a SUBSCRIBE request to refresh a subscription receives a 404, 405, 410, 416, 480-485, 489, 501, or 604 response, the subscriber should consider the subscription terminated. (See [[RFC5057](#)] for further details and notes about the effect of error codes on dialogs and usages within dialog, such as subscriptions). If the subscriber wishes to re-subscribe to the state, he does so by composing an unrelated initial SUBSCRIBE request with a freshly-generated Call-ID and a new, unique "From" tag (see [Section 4.1.2.1](#).)

If a SUBSCRIBE request to refresh a subscription fails with any error code other than those listed above, the original subscription is still considered valid for the duration of the most recently known "Expires" value as negotiated by SUBSCRIBE and its response, or as communicated by NOTIFY in the "Subscription-State" header field "expires" parameter.

Note that many such errors indicate that there may be a problem with the network or the notifier such that no further NOTIFY messages will be received.

4.1.2.3. Unsubscribing

Unsubscribing is handled in the same way as refreshing of a subscription, with the "Expires" header field set to "0". Note that a successful unsubscription will also trigger a final NOTIFY message.

The final NOTIFY may or may not contain information about the state of the resource; subscribers need to be prepared to receive final NOTIFY messages both with and without state.

4.1.2.4. Confirmation of Subscription Creation

The subscriber can expect to receive a NOTIFY message from each node which has processed a successful subscription or subscription refresh. To ensure that subscribers do not wait indefinitely for a subscription to be established, a subscriber starts a Timer L, set to $64 * T1$. If this Timer L expires prior to the receipt of a NOTIFY message, the subscriber considers the subscription failed, and cleans up any state associated with the subscription attempt.

Until Timer L expires, several NOTIFY messages may arrive from different destinations (see [Section 4.4.1](#)). Each of these messages establish a new dialog and a new subscription. After the expiration of Timer L, the subscriber SHOULD reject any such NOTIFY messages that would otherwise establish a new dialog with a "481" response code.

Until the first NOTIFY message arrives, the subscriber should consider the state of the subscribed resource to be in a neutral state. Documents which define new event packages MUST define this "neutral state" in such a way that makes sense for their application (see [Section 5.4.7](#)).

Due to the potential for both out-of-order messages and forking, the subscriber MUST be prepared to receive NOTIFY messages before the SUBSCRIBE transaction has completed.

Except as noted above, processing of this NOTIFY is the same as in [Section 4.1.3](#).

4.1.3. Receiving and Processing State Information

Subscribers receive information about the state of a resource to which they have subscribed in the form of NOTIFY requests.

Upon receiving a NOTIFY request, the subscriber should check that it matches at least one of its outstanding subscriptions; if not, it MUST return a "481 Subscription does not exist" response unless another 400- or 500-class response is more appropriate. The rules for matching NOTIFY requests with subscriptions that create a new dialog are described in [Section 4.4.1](#). Notifications for subscriptions which were created inside an existing dialog match if they are in the same dialog and the "Event" header fields match (as described in [Section 8.2.1](#)).

If, for some reason, the event package designated in the "Event" header field of the NOTIFY request is not supported, the subscriber will respond with a "489 Bad Event" response.

To prevent spoofing of events, NOTIFY requests SHOULD be authenticated, using any defined SIP authentication mechanism.

NOTIFY requests MUST contain "Subscription-State" header fields which indicate the status of the subscription.

If the "Subscription-State" header field value is "active", it means that the subscription has been accepted and (in general) has been authorized. If the header field also contains an "expires" parameter, the subscriber SHOULD take it as the authoritative subscription duration and adjust accordingly. The "retry-after" and "reason" parameters have no semantics for "active".

If the "Subscription-State" value is "pending", the subscription has been received by the notifier, but there is insufficient policy information to grant or deny the subscription yet. If the header field also contains an "expires" parameter, the subscriber SHOULD take it as the authoritative subscription duration and adjust accordingly. No further action is necessary on the part of the subscriber. The "retry-after" and "reason" parameters have no semantics for "pending".

If the "Subscription-State" value is "terminated", the subscriber should consider the subscription terminated. The "expires" parameter has no semantics for "terminated" -- notifiers SHOULD NOT include an "expires" parameter on a "Subscription-State" header field with a value of "terminated," and subscribers MUST ignore any such parameter, if present. If a reason code is present, the client should behave as described below. If no reason code or an unknown reason code is present, the client MAY attempt to re-subscribe at any time (unless a "retry-after" parameter is present, in which case the client SHOULD NOT attempt re-subscription until after the number of seconds specified by the "retry-after" parameter). The reason codes defined by this document are:

deactivated: The subscription has been terminated, but the subscriber SHOULD retry immediately with a new subscription. One primary use of such a status code is to allow migration of subscriptions between nodes. The "retry-after" parameter has no semantics for "deactivated".

probation: The subscription has been terminated, but the client SHOULD retry at some later time. If a "retry-after" parameter is also present, the client SHOULD wait at least the number of seconds specified by that parameter before attempting to re-subscribe.

rejected: The subscription has been terminated due to change in authorization policy. Clients SHOULD NOT attempt to re-subscribe. The "retry-after" parameter has no semantics for "rejected".

timeout: The subscription has been terminated because it was not refreshed before it expired. Clients MAY re-subscribe immediately. The "retry-after" parameter has no semantics for "timeout". This reason code is also associated with polling of resource state, as detailed in [Section 4.4.3](#)

giveup: The subscription has been terminated because the notifier could not obtain authorization in a timely fashion. If a "retry-after" parameter is also present, the client SHOULD wait at least the number of seconds specified by that parameter before attempting to re-subscribe; otherwise, the client MAY retry immediately, but will likely get put back into pending state.

noresource: The subscription has been terminated because the resource state which was being monitored no longer exists. Clients SHOULD NOT attempt to re-subscribe. The "retry-after" parameter has no semantics for "noresource".

invariant: The subscription has been terminated because the resource state is guaranteed not to change for the foreseeable future. This may be the case, for example, when subscribing to the location information of a fixed-location land-line telephone. When using this reason code, notifiers are advised to include a "retry-after" parameter with a large value (for example, 31536000 -- or one year) to prevent older, [RFC 3265](#)-compliant clients from periodically resubscribing. Clients SHOULD NOT attempt to resubscribe after receiving a reason code of "invariant," regardless of the presence of or value of a "retry-after" parameter.

Other specifications may define new reason codes for use with the "Subscription-State" header field.

Once the notification is deemed acceptable to the subscriber, the subscriber SHOULD return a 200 response. In general, it is not expected that NOTIFY responses will contain bodies; however, they MAY, if the NOTIFY request contained an "Accept" header field.

Other responses defined in SIP [[RFC3261](#)] may also be returned, as appropriate. In no case should a NOTIFY transaction extend for any longer than the time necessary for automated processing. In particular, subscribers MUST NOT wait for a user response before returning a final response to a NOTIFY request.

[4.1.4.](#) Forking of SUBSCRIBE Messages

In accordance with the rules for proxying non-INVITE requests as defined in SIP [[RFC3261](#)], successful SUBSCRIBE requests will receive only one 200-class response; however, due to forking, the subscription may have been accepted by multiple nodes. The subscriber MUST therefore be prepared to receive NOTIFY requests with "From:" tags which differ from the "To:" tag received in the SUBSCRIBE 200-class response.

If multiple NOTIFY messages are received in different dialogs in response to a single SUBSCRIBE message, each dialog represents a different destination to which the SUBSCRIBE request was forked. Subscriber handling in such situations varies by event package; see [Section 5.4.9](#) for details.

[4.2.](#) Notifier Behavior

[4.2.1.](#) Subscription Establishment and Maintenance

Notifiers learn about subscription requests by receiving SUBSCRIBE requests from interested parties. Notifiers MUST NOT create subscriptions except upon receipt of a SUBSCRIBE message. However, for historical reasons, the implicit creation of subscriptions as defined in [[RFC3515](#)] is still permitted.

[RFC3265] allowed the creation of subscriptions using means other than SUBSCRIBE. The only standardized use of this mechanism is the REFER method [[RFC3515](#)]. Implementation experience with REFER has shown that the implicit creation of a subscription has a number of undesirable effects, such as the inability to signal the success of a REFER while signaling a problem with the subscription; and difficulty performing one action without the other. Additionally, the proper exchange of dialog identifiers is difficult without dialog re-use (which has its own set of problems; see [Section 4.5](#)).

4.2.1.1. Initial SUBSCRIBE Transaction Processing

In no case should a SUBSCRIBE transaction extend for any longer than the time necessary for automated processing. In particular, notifiers MUST NOT wait for a user response before returning a final response to a SUBSCRIBE request.

This requirement is imposed primarily to prevent the non-INVITE transaction timeout timer F (see [[RFC3261](#)]) from firing during the SUBSCRIBE transaction, since interaction with a user would often exceed $64 \cdot T1$ seconds.

The notifier SHOULD check that the event package specified in the "Event" header field is understood. If not, the notifier SHOULD return a "489 Bad Event" response to indicate that the specified event/event class is not understood.

The notifier SHOULD also perform any necessary authentication and authorization per its local policy. See [Section 4.2.1.3](#).

The notifier MAY also check that the duration in the "Expires" header field is not too small. If and only if the expiration interval is greater than zero AND smaller than one hour AND less than a notifier-configured minimum, the notifier MAY return a "423 Interval Too Brief" error which contains a "Min-Expires" header field. The "Min-Expires" header field is described in SIP [[RFC3261](#)].

If the notifier is able to immediately determine that it understands the event package, that the authenticated subscriber is authorized to subscribe, and that there are no other barriers to creating the subscription, it creates the subscription and a dialog (if necessary), and returns a "200 OK" response (unless doing so would reveal authorization policy in an undesirable fashion; see [Section 6.2](#)).

If the notifier cannot immediately create the subscription (e.g., it needs to wait for user input for authorization, or is acting for another node which is not currently reachable), or wishes to mask authorization policy, it will return a "202 Accepted" response. This response indicates that the request has been received and understood, but does not necessarily imply that the subscription has been authorized yet.

When a subscription is created in the notifier, it stores the event package name as part of the subscription information.

The "Expires" values present in SUBSCRIBE 200-class responses behave in the same way as they do in REGISTER responses: the server MAY

shorten the interval, but MUST NOT lengthen it.

If the duration specified in a SUBSCRIBE message is unacceptably short, the notifier may be able to send a 423 response, as described earlier in this section.

200-class responses to SUBSCRIBE requests will not generally contain any useful information beyond subscription duration; their primary purpose is to serve as a reliability mechanism. State information will be communicated via a subsequent NOTIFY request from the notifier.

The other response codes defined in SIP [[RFC3261](#)] may be used in response to SUBSCRIBE requests, as appropriate.

4.2.1.2. Confirmation of Subscription Creation/Refreshing

Upon successfully accepting or refreshing a subscription, notifiers MUST send a NOTIFY message immediately to communicate the current resource state to the subscriber. This NOTIFY message is sent on the same dialog as created by the SUBSCRIBE response. If the resource has no meaningful state at the time that the SUBSCRIBE message is processed, this NOTIFY message MAY contain an empty or neutral body. See [Section 4.2.2](#) for further details on NOTIFY message generation.

Note that a NOTIFY message is always sent immediately after any 200-class response to a SUBSCRIBE request, regardless of whether the subscription has already been authorized.

4.2.1.3. Authentication/Authorization of SUBSCRIBE requests

Privacy concerns may require that notifiers apply policy to determine whether a particular subscriber is authorized to subscribe to a certain set of events. Such policy may be defined by mechanisms such as access control lists or real-time interaction with a user. In general, authorization of subscribers prior to authentication is not particularly useful.

SIP authentication mechanisms are discussed in SIP [[RFC3261](#)]. Note that, even if the notifier node typically acts as a proxy, authentication for SUBSCRIBE requests will always be performed via a "401" response, not a "407;" notifiers always act as a user agents when accepting subscriptions and sending notifications.

Of course, when acting as a proxy, a node will perform normal proxy authentication (using 407). The foregoing explanation is a reminder that notifiers are always UAs, and as such perform UA authentication.

If authorization fails based on an access list or some other automated mechanism (i.e., it can be automatically authoritatively determined that the subscriber is not authorized to subscribe), the notifier SHOULD reply to the request with a "403 Forbidden" or "603 Decline" response, unless doing so might reveal information that should stay private; see [Section 6.2](#).

If the notifier owner is interactively queried to determine whether a subscription is allowed, a "202 Accept" response is returned immediately. Note that a NOTIFY message is still formed and sent under these circumstances, as described in the previous section.

If subscription authorization was delayed and the notifier wishes to convey that such authorization has been declined, it may do so by sending a NOTIFY message containing a "Subscription-State" header field with a value of "terminated" and a reason parameter of "rejected".

[4.2.1.4](#). Refreshing of Subscriptions

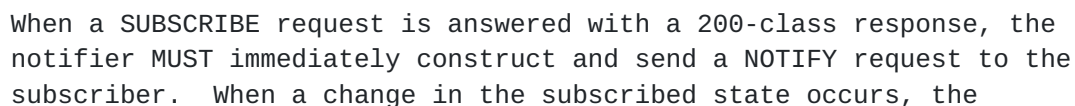
When a notifier receives a subscription refresh, assuming that the subscriber is still authorized, the notifier updates the expiration time for subscription. As with the initial subscription, the server MAY shorten the amount of time until expiration, but MUST NOT increase it. The final expiration time is placed in the "Expires" header field in the response. If the duration specified in a SUBSCRIBE message is unacceptably short, the notifier SHOULD respond with a "423 Interval Too Brief" message.

If no refresh for a notification address is received before its expiration time, the subscription is removed. When removing a subscription, the notifier SHOULD send a NOTIFY message with a "Subscription-State" value of "terminated" to inform it that the subscription is being removed. If such a message is sent, the "Subscription-State" header field SHOULD contain a "reason=timeout" parameter.

Clients can cause a subscription to be terminated immediately by sending a SUBSCRIBE with an "Expires" header field set to '0'. Notifiers largely treat this the same way as any other subscription expiration: they send a NOTIFY message containing a "Subscription-State" of "terminated", with a reason code of "timeout." For consistency with state polling (see [Section 4.4.3](#)) and subscription

The sending of a NOTIFY when a subscription expires allows the corresponding dialog to be terminated, if appropriate.

Notifiers use the NOTIFY method to send information about the state of a resource to subscribers. The notifier's view of a subscription is shown in the following state diagram:



notifier SHOULD immediately construct and send a NOTIFY request, subject to authorization, local policy, and throttling considerations.

If the NOTIFY request fails due to expiration of SIP Timer F (transaction timeout), the notifier SHOULD remove the subscription.

This behavior prevents unnecessary transmission of state information for subscribers who have crashed or disappeared from the network. Because such transmissions will be sent multiple times, per the retransmission algorithm defined in SIP [[RFC3261](#)] (instead of the typical single transmission for functioning clients), continuing to service them when no client is available to acknowledge them could place undue strain on a network. Upon client restart or reestablishment of a network connection, it is expected that clients will send SUBSCRIBE messages to refresh potentially stale state information; such messages will re-install subscriptions in all relevant nodes.

If the NOTIFY transaction fails due to the receipt of a 404, 405, 410, 416, 480-485, 489, 501, or 604 response to the NOTIFY, the notifier MUST remove the corresponding subscription. See [[RFC5057](#)] for further details and notes about the effect of error codes on dialogs and usages within dialog (such as subscriptions).

A notify error response would generally indicate that something has gone wrong with the subscriber or with some proxy on the way to the subscriber. If the subscriber is in error, it makes the most sense to allow the subscriber to rectify the situation (by re-subscribing) once the error condition has been handled. If a proxy is in error, the periodic SUBSCRIBE refreshes will re-install subscription state once the network problem has been resolved.

NOTIFY requests MUST contain a "Subscription-State" header field with a value of "active", "pending", or "terminated". The "active" value indicates that the subscription has been accepted and has been authorized (in most cases; see [Section 6.2](#)). The "pending" value indicates that the subscription has been received, but that policy information is insufficient to accept or deny the subscription at this time. The "terminated" value indicates that the subscription is not active.

If the value of the "Subscription-State" header field is "active" or "pending", the notifier MUST also include in the "Subscription-State" header field an "expires" parameter which indicates the time remaining on the subscription. The notifier MAY use this mechanism to shorten a subscription; however, this mechanism MUST NOT be used

to lengthen a subscription.

Including expiration information for active and pending subscriptions is necessary in case the SUBSCRIBE request forks, since the response to a forked SUBSCRIBE may not be received by the subscriber. [RFC3265] allowed the notifier some discretion in the inclusion of this parameter, so subscriber implementations are warned to handle the lack of an "expires" parameter gracefully. Note well that this "expires" value is a parameter on the "Subscription-State" header field, NOT an "Expires" header field.

The period of time for a subscription can be shortened to zero by the notifier. In other words, it is perfectly valid for a SUBSCRIBE with a non-zero expires to be answered with a NOTIFY that contains "Subscription-Status: terminated;reason=expired". This merely means that the notifier has shortened the subscription timeout to zero, and the subscription has expired instantaneously. The body may contain valid state, or it may contain a neutral state (see [Section 5.4.7](#)).

If the value of the "Subscription-State" header field is "terminated", the notifier SHOULD also include a "reason" parameter. The notifier MAY also include a "retry-after" parameter, where appropriate. For details on the value and semantics of the "reason" and "retry-after" parameters, see [Section 4.1.3](#).

[4.2.3](#). PINT Compatibility

The "Event" header field is considered mandatory for the purposes of this document. However, to maintain compatibility with PINT (see [RFC2848](#)), notifiers MAY interpret a SUBSCRIBE request with no "Event" header field as requesting a subscription to PINT events. If a notifier does not support PINT, it SHOULD return "489 Bad Event" to any SUBSCRIBE messages without an "Event" header field.

[4.3](#). Proxy Behavior

Proxies need no additional behavior beyond that described in SIP [RFC3261](#) to support SUBSCRIBE and NOTIFY. If a proxy wishes to see all of the SUBSCRIBE and NOTIFY requests for a given dialog, it MUST add a Record-Route header field to the initial SUBSCRIBE request and all NOTIFY requests. It MAY choose to include Record-Route in subsequent SUBSCRIBE messages; however, these requests cannot cause the dialog's route set to be modified.

Note that subscribers and notifiers may elect to use S/MIME encryption of SUBSCRIBE and NOTIFY requests; consequently, proxies cannot rely on being able to access any information that is not explicitly required to be proxy-readable by SIP [[RFC3261](#)].

4.4. Common Behavior

4.4.1. Dialog Creation and Termination

Dialogs are created upon completion of a NOTIFY transaction for a new subscription, unless the NOTIFY contains a "Subscription-State" of "terminated."

NOTIFY requests are matched to such SUBSCRIBE requests if they contain the same "Call-ID", a "To" header field "tag" parameter which matches the "From" header field "tag" parameter of the SUBSCRIBE, and the same "Event" header field. Rules for comparisons of the "Event" header fields are described in [Section 8.2.1](#).

A subscription is destroyed after a notifier sends a NOTIFY request with a "Subscription-State" of "terminated." The subscriber will generally answer such final requests with a "200 OK" response (unless a condition warranting an alternate response has arisen). Except when the mechanism described in [Section 4.5.2](#) is used, the destruction of a subscription results in the termination of its associated dialog.

A subscriber may send a SUBSCRIBE request with an "Expires" header field of 0 in order to trigger the sending of such a NOTIFY request; however, for the purposes of subscription and dialog lifetime, the subscription is not considered terminated until the NOTIFY transaction with a "Subscription-State" of "terminated" completes.

4.4.2. Notifier Migration

It is often useful to allow migration of subscriptions between notifiers. Such migration may be effected by sending a NOTIFY message with a "Subscription-State" header field of "terminated", and a reason parameter of "deactivated". This NOTIFY request is otherwise normal, and is formed as described in [Section 4.2.2](#).

Upon receipt of this NOTIFY message, the subscriber SHOULD attempt to re-subscribe (as described in the preceding sections). Note that this subscription is established on a new dialog, and does not re-use the route set from the previous subscription dialog.

The actual migration is effected by making a change to the policy

(such as routing decisions) of one or more servers to which the SUBSCRIBE request will be sent in such a way that a different node ends up responding to the SUBSCRIBE request. This may be as simple as a change in the local policy in the notifier from which the subscription is migrating so that it serves as a proxy or redirect server instead of a notifier.

Whether, when, and why to perform notifier migrations may be described in individual event packages; otherwise, such decisions are a matter of local notifier policy, and are left up to individual implementations.

4.4.3. Polling Resource State

A natural consequence of the behavior described in the preceding sections is that an immediate fetch without a persistent subscription may be effected by sending a SUBSCRIBE with an "Expires" of 0.

Of course, an immediate fetch while a subscription is active may be effected by sending a SUBSCRIBE with an "Expires" equal to the number of seconds remaining in the subscription.

Upon receipt of this SUBSCRIBE request, the notifier (or notifiers, if the SUBSCRIBE request was forked) will send a NOTIFY request containing resource state in the same dialog.

Note that the NOTIFY messages triggered by SUBSCRIBE messages with "Expires" header fields of 0 will contain a "Subscription-State" value of "terminated", and a "reason" parameter of "timeout".

Polling of event state can cause significant increases in load on the network and notifiers; as such, it should be used only sparingly. In particular, polling SHOULD NOT be used in circumstances in which it will typically result in more network messages than long-running subscriptions.

When polling is used, subscribers SHOULD attempt to cache authentication credentials between polls so as to reduce the number of messages sent.

4.4.4. Allow-Events header field usage

The "Allow-Events" header field, if present, includes a list of tokens which indicates the event packages supported by a notifier. In other words, a user agent sending an "Allow-Events" header field is advertising that it can process SUBSCRIBE requests and generate NOTIFY requests for all of the event packages listed in that header field.

Any user agent that can act as a notifier for one or more event packages SHOULD include an appropriate "Allow-Events" header field indicating all supported events in all methods which initiate dialogs and their responses (such as INVITE) and OPTIONS responses.

This information is very useful, for example, in allowing user agents to render particular interface elements appropriately according to whether the events required to implement the features they represent are supported by the appropriate nodes.

On the other hand, it doesn't necessarily make much sense to indicate supported events inside a NOTIFY-established dialog if the only event package supported is the one associated with that subscription.

Note that "Allow-Events" header fields MUST NOT be inserted by proxies.

4.5. Targeting Subscriptions at Devices

[RFC3265] defined a mechanism by which subscriptions could share dialogs with invite usages and with other subscriptions. The purpose of this behavior was to allow subscribers to ensure that a subscription arrived at the same device as an established dialog. Unfortunately, the re-use of dialogs has proven to be exceedingly confusing. [RFC5057] attempted to clarify proper behavior in a variety of circumstances; however, the ensuing rules remain confusing and prone to implementation error. At the same time, the mechanism described in [I-D.ietf-sip-gruu] now provides a far more elegant and unambiguous means to achieve the same goal.

Consequently, the dialog re-use technique described in [RFC 3265](#) is now deprecated.

4.5.1. Using GRUUs to Route to Devices

Notifiers MUST implement the GRUU extension defined in [I-D.ietf-sip-gruu], and MUST use a GRUU as their local target. This allows subscribers to explicitly target desired devices.

If a subscriber wishes to subscribe to a resource on the same device as an established dialog, it should check whether the remote contact in that dialog is a GRUU (i.e., whether it contains a "gr" URI parameter). If so, the subscriber creates a new dialog, using the GRUU as the request URI for the new SUBSCRIBE.

Because GRUUs are guaranteed to route to a specific device, this ensures that the subscription will be routed to the same place as the established dialog.

4.5.2. Sharing Dialogs

For compatibility with older clients, subscriber and notifier implementations may choose to allow dialog sharing. The behavior of multiple usages within a dialog are described in [[RFC5057](#)].

Subscribers MUST NOT attempt to re-use dialogs whose remote target is a GRUU.

Note that the techniques described in this section are included for backwards compatibility purposes only. Because subscribers cannot re-use dialogs with a GRUU for their remote target, and because notifiers must use GRUUs as their local target, any two implementations that conform to this specification will automatically use the mechanism described in [Section 4.5.1](#).

If a subscriber wishes to subscribe to a resource on the same device as an established dialog and the remote contact is not a GRUU, it MAY revert to dialog sharing behavior. Alternately, it MAY choose to treat the remote party as incapable of servicing the subscription (i.e., the same way it would behave if the remote party did not support SIP events at all).

If a notifier receives a SUBSCRIBE request for a new subscription on an existing dialog, it MAY choose to implement dialog sharing behavior. Alternately, it may choose to fail the SUBSCRIBE request with a 403 response. The error text of such 403 responses SHOULD indicate that dialog sharing is not supported.

To implement dialog sharing, subscribers and notifiers perform the following additional processing:

- o When subscriptions exist in dialogs associated with INVITE-created application state and/or other subscriptions, these sets of application state do not interact beyond the behavior described for a dialog (e.g., route set handling). In particular, multiple subscriptions within a dialog are expire independently, and require independent SUBSCRIBE refreshes.
- o If a subscription's destruction leaves no other application state associated with the dialog, the dialog terminates. The destruction of other application state (such as that created by an INVITE) will not terminate the dialog if a subscription is still associated with that dialog. This means that, when dialog are re-

used, then a dialog created with an INVITE does not necessarily terminate upon receipt of a BYE. Similarly, in the case that several subscriptions are associated with a single dialog, the dialog does not terminate until all the subscriptions in it are destroyed.

- o Subscribers MAY include an "id" parameter in SUBSCRIBE request "Event" header field to allow differentiation between multiple subscriptions in the same dialog. This "id" parameter, if present, contains an opaque token which identifies the specific subscription within a dialog. An "id" parameter is only valid within the scope of a single dialog.
- o If an "id" parameter is present in the SUBSCRIBE message used to establish a subscription, that "id" parameter MUST also be present in all corresponding NOTIFY messages.
- o When a subscriber refreshes a the subscription timer, the SUBSCRIBE MUST contain the same "Event" header field "id" parameter as was present in the initial subscription. (Otherwise, the notifier will interpret the SUBSCRIBE message as a request for a new subscription in the same dialog).
- o When a subscription is created in the notifier, it stores the any "Event" header field "id" parameter as part of the subscription information (along with the event package name).
- o If an initial SUBSCRIBE is sent on a pre-existing dialog, a matching NOTIFY request merely creates a new subscription associated with that dialog.

4.6. CANCEL Requests for SUBSCRIBE and NOTIFY

Neither SUBSCRIBE nor NOTIFY messages can be canceled. If a UAS receives a CANCEL request that matches a known SUBSCRIBE or NOTIFY transaction, it MUST respond to the CANCEL request, but otherwise ignore it. In particular, the CANCEL request MUST NOT affect processing of the SUBSCRIBE or NOTIFY request in any way.

UACs SHOULD NOT send CANCEL requests for SUBSCRIBE or NOTIFY transactions.

5. Event Packages

This section covers several issues which should be taken into consideration when event packages based on SUBSCRIBE and NOTIFY are

proposed. Event package definitions contain sections addressing each of these issues, ideally in the same order and with the same titles as the following sections.

5.1. Appropriateness of Usage

When designing an event package using the methods described in this document for event notification, it is important to consider: is SIP an appropriate mechanism for the problem set? Is SIP being selected because of some unique feature provided by the protocol (e.g., user mobility), or merely because "it can be done?" If you find yourself defining event packages for notifications related to, for example, network management or the temperature inside your car's engine, you may want to reconsider your selection of protocols.

Those interested in extending the mechanism defined in this document are urged to follow the development of "Guidelines for Authors of SIP Extensions" [[RFC4485](#)] for further guidance regarding appropriate uses of SIP.

Further, it is expected that this mechanism is not to be used in applications where the frequency of reportable events is excessively rapid (e.g., more than about once per second). A SIP network is generally going to be provisioned for a reasonable signaling volume; sending a notification every time a user's GPS position changes by one hundredth of a second could easily overload such a network.

5.2. Event Template-packages

Normal event packages define a set of state applied to a specific type of resource, such as user presence, call state, and messaging mailbox state.

Event template-packages are a special type of package which define a set of state applied to other packages, such as statistics, access policy, and subscriber lists. Event template-packages may even be applied to other event template-packages.

To extend the object-oriented analogy made earlier, event template-packages can be thought of as templated C++ packages which must be applied to other packages to be useful.

The name of an event template-package as applied to a package is formed by appending a period followed by the event template-package name to the end of the package. For example, if a template-package called "winfo" were being applied to a package called "presence", the event token used in "Event" and "Allow-Events" would be "presence.winfo".

Event template-packages must be defined so that they can be applied to any arbitrary package. In other words, event template-packages cannot be specifically tied to one or a few "parent" packages in such a way that they will not work with other packages.

5.3. Amount of State to be Conveyed

When designing event packages, it is important to consider the type of information which will be conveyed during a notification.

A natural temptation is to convey merely the event (e.g., "a new voice message just arrived") without accompanying state (e.g., "7 total voice messages"). This complicates implementation of subscribing entities (since they have to maintain complete state for the entity to which they have subscribed), and also is particularly susceptible to synchronization problems.

There are two possible solutions to this problem that event packages may choose to implement.

5.3.1. Complete State Information

For packages which typically convey state information that is reasonably small (on the order of 1 KB or so), it is suggested that event packages are designed so as to send complete state information when an event occurs.

In some circumstances, conveying the current state alone may be insufficient for a particular class of events. In these cases, the event packages should include complete state information along with the event that occurred. For example, conveying "no customer service representatives available" may not be as useful as conveying "no customer service representatives available; representative sip:46@cs.xyz.int just logged off".

5.3.2. State Deltas

In the case that the state information to be conveyed is large, the event package may choose to detail a scheme by which NOTIFY messages contain state deltas instead of complete state.

Such a scheme would work as follows: any NOTIFY sent in immediate response to a SUBSCRIBE contains full state information. NOTIFY messages sent because of a state change will contain only the state information that has changed; the subscriber will then merge this information into its current knowledge about the state of the resource.

Any event package that supports delta changes to states **MUST** include a version number that increases by exactly one for each NOTIFY transaction in a subscription. Note that the state version number appears in the body of the message, not in a SIP header field.

If a NOTIFY arrives that has a version number that is incremented by more than one, the subscriber knows that a state delta has been missed; it ignores the NOTIFY message containing the state delta (except for the version number, which it retains to detect message loss), and re-sends a SUBSCRIBE to force a NOTIFY containing a complete state snapshot.

5.4. Event Package Responsibilities

Event packages are not required to reiterate any of the behavior described in this document, although they may choose to do so for clarity or emphasis. In general, though, such packages are expected to describe only the behavior that extends or modifies the behavior described in this document.

Note that any behavior designated with "SHOULD" or "MUST" in this document is not allowed to be weakened by extension documents; however, such documents may elect to strengthen "SHOULD" requirements to "MUST" strength if required by their application.

In addition to the normal sections expected in standards-track RFCs and SIP extension documents, authors of event packages need to address each of the issues detailed in the following subsections, whenever applicable.

5.4.1. Event Package Name

This section, which **MUST** be present, defines the token name to be used to designate the event package. It **MUST** include the information which appears in the IANA registration of the token. For information on registering such types, see [Section 7](#).

5.4.2. Event Package Parameters

If parameters are to be used on the "Event" header field to modify the behavior of the event package, the syntax and semantics of such header fields **MUST** be clearly defined.

5.4.3. SUBSCRIBE Bodies

It is expected that most, but not all, event packages will define syntax and semantics for SUBSCRIBE method bodies; these bodies will typically modify, expand, filter, throttle, and/or set thresholds for

the class of events being requested. Designers of event packages are strongly encouraged to re-use existing MIME types for message bodies where practical.

This mandatory section of an event package defines what type or types of event bodies are expected in SUBSCRIBE requests (or specify that no event bodies are expected). It should point to detailed definitions of syntax and semantics for all referenced body types.

5.4.4. Subscription Duration

It is RECOMMENDED that event packages give a suggested range of times considered reasonable for the duration of a subscription. Such packages MUST also define a default "Expires" value to be used if none is specified.

5.4.5. NOTIFY Bodies

The NOTIFY body is used to report state on the resource being monitored. Each package MUST define what type or types of event bodies are expected in NOTIFY requests. Such packages MUST specify or cite detailed specifications for the syntax and semantics associated with such event body.

Event packages also MUST define which MIME type is to be assumed if none are specified in the "Accept" header field of the SUBSCRIBE request.

5.4.6. Notifier processing of SUBSCRIBE requests

This section describes the processing to be performed by the notifier upon receipt of a SUBSCRIBE request. Such a section is required.

Information in this section includes details of how to authenticate subscribers and authorization issues for the package. Such authorization issues may include, for example, whether all SUBSCRIBE requests for this package are answered with 202 responses (see [Section 6.2](#)).

5.4.7. Notifier generation of NOTIFY requests

This section of an event package describes the process by which the notifier generates and sends a NOTIFY request. This includes detailed information about what events cause a NOTIFY to be sent, how to compute the state information in the NOTIFY, how to generate neutral or fake state information to hide authorization delays and decisions from users, and whether state information is complete or deltas for notifications; see [Section 5.3](#). Such a section is

required.

This section may optionally describe the behavior used to process the subsequent response.

5.4.8. Subscriber processing of NOTIFY requests

This section of an event package describes the process followed by the subscriber upon receipt of a NOTIFY request, including any logic required to form a coherent resource state (if applicable).

5.4.9. Handling of forked requests

Each event package **MUST** specify whether forked SUBSCRIBE requests are allowed to install multiple subscriptions.

If such behavior is not allowed, the first potential dialog-establishing message will create a dialog. All subsequent NOTIFY messages which correspond to the SUBSCRIBE message (i.e., match "To", "From", "From" header field "tag" parameter, "Call-ID", "Event", and "Event" header field "id" parameter) but which do not match the dialog would be rejected with a 481 response. Note that the 200-class response to the SUBSCRIBE can arrive after a matching NOTIFY has been received; such responses might not correlate to the same dialog established by the NOTIFY. Except as required to complete the SUBSCRIBE transaction, such non-matching 200-class responses are ignored.

If installing of multiple subscriptions by way of a single forked SUBSCRIBE is allowed, the subscriber establishes a new dialog towards each notifier by returning a 200-class response to each NOTIFY. Each dialog is then handled as its own entity, and is refreshed independent of the other dialogs.

In the case that multiple subscriptions are allowed, the event package **MUST** specify whether merging of the notifications to form a single state is required, and how such merging is to be performed. Note that it is possible that some event packages may be defined in such a way that each dialog is tied to a mutually exclusive state which is unaffected by the other dialogs; this **MUST** be clearly stated if it is the case.

5.4.10. Rate of notifications

Each event package is expected to define a requirement (SHOULD or MUST strength) which defines an absolute maximum on the rate at which notifications are allowed to be generated by a single notifier.

Each package MAY further define a throttle mechanism which allows subscribers to further limit the rate of notification.

5.4.11. State Aggregation

Many event packages inherently work by collecting information about a resource from a number of other sources -- either through the use of PUBLISH [[RFC3903](#)], by subscribing to state information, or through other state gathering mechanisms.

Event packages that involve retrieval of state information for a single resource from more than one source need to consider how notifiers aggregate information into a single, coherent state. Such packages MUST specify how notifiers aggregate information and how they provide authentication and authorization.

5.4.12. Examples

Event packages SHOULD include several demonstrative message flow diagrams paired with several typical, syntactically correct, and complete messages.

It is RECOMMENDED that documents describing event packages clearly indicate that such examples are informative and not normative, with instructions that implementors refer to the main text of the document for exact protocol details.

5.4.13. Use of URIs to Retrieve State

Some types of event packages may define state information which is potentially too large to reasonably send in a SIP message. To alleviate this problem, event packages may include the ability to convey a URI instead of state information; this URI will then be used to retrieve the actual state information.

[RFC4483] defines a mechanism that can be used by event packages to convey information in such a fashion.

6. Security Considerations

6.1. Access Control

The ability to accept subscriptions should be under the direct control of the notifier's user, since many types of events may be considered sensitive for the purposes of privacy. Similarly, the notifier should have the ability to selectively reject subscriptions based on the subscriber identity (based on access control lists),

using standard SIP authentication mechanisms. The methods for creation and distribution of such access control lists is outside the scope of this document.

6.2. Notifier Privacy Mechanism

The mere act of returning a 200 or certain 4xx and 6xx responses to SUBSCRIBE requests may, under certain circumstances, create privacy concerns by revealing sensitive policy information. In these cases, the notifier SHOULD always return a 202 response. While the subsequent NOTIFY message may not convey true state, it MUST appear to contain a potentially correct piece of data from the point of view of the subscriber, indistinguishable from a valid response. Information about whether a user is authorized to subscribe to the requested state is never conveyed back to the original user under these circumstances.

Individual packages and their related documents for which such a mode of operation makes sense can further describe how and why to generate such potentially correct data. For example, such a mode of operation is mandated by [[RFC2779](#)] for user presence information.

6.3. Denial-of-Service attacks

The current model (one SUBSCRIBE request triggers a SUBSCRIBE response and one or more NOTIFY requests) is a classic setup for an amplifier node to be used in a smurf attack.

Also, the creation of state upon receipt of a SUBSCRIBE request can be used by attackers to consume resources on a victim's machine, rendering it unusable.

To reduce the chances of such an attack, implementations of notifiers SHOULD require authentication. Authentication issues are discussed in SIP [[RFC3261](#)].

6.4. Replay Attacks

Replaying of either SUBSCRIBE or NOTIFY can have detrimental effects.

In the case of SUBSCRIBE messages, attackers may be able to install any arbitrary subscription which it witnessed being installed at some point in the past. Replaying of NOTIFY messages may be used to spoof old state information (although a good versioning mechanism in the body of the NOTIFY messages may help mitigate such an attack). Note that the prohibition on sending NOTIFY messages to nodes which have not subscribed to an event also aids in mitigating the effects of such an attack.

To prevent such attacks, implementations SHOULD require authentication with anti-replay protection. Authentication issues are discussed in SIP [[RFC3261](#)].

6.5. Man-in-the middle attacks

Even with authentication, man-in-the-middle attacks using SUBSCRIBE may be used to install arbitrary subscriptions, hijack existing subscriptions, terminate outstanding subscriptions, or modify the resource to which a subscription is being made. To prevent such attacks, implementations SHOULD provide integrity protection across "Contact", "Route", "Expires", "Event", and "To" header fields of SUBSCRIBE messages, at a minimum. If SUBSCRIBE bodies are used to define further information about the state of the call, they SHOULD be included in the integrity protection scheme.

Man-in-the-middle attacks may also attempt to use NOTIFY messages to spoof arbitrary state information and/or terminate outstanding subscriptions. To prevent such attacks, implementations SHOULD provide integrity protection across the "Call-ID", "CSeq", and "Subscription-State" header fields and the bodies of NOTIFY messages.

Integrity protection of message header fields and bodies is discussed in SIP [[RFC3261](#)].

6.6. Confidentiality

The state information contained in a NOTIFY message has the potential to contain sensitive information. Implementations MAY encrypt such information to ensure confidentiality.

While less likely, it is also possible that the information contained in a SUBSCRIBE message contains information that users might not want to have revealed. Implementations MAY encrypt such information to ensure confidentiality.

To allow the remote party to hide information it considers sensitive, all implementations SHOULD be able to handle encrypted SUBSCRIBE and NOTIFY messages.

The mechanisms for providing confidentiality are detailed in SIP [[RFC3261](#)].

7. IANA Considerations

(This section is not applicable until this document is published as an RFC.)

7.1. Event Packages

This document defines an event-type namespace which requires a central coordinating body. The body chosen for this coordination is the Internet Assigned Numbers Authority (IANA).

There are two different types of event-types: normal event packages, and event template-packages; see [Section 5.2](#). To avoid confusion, template-package names and package names share the same namespace; in other words, an event template-package MUST NOT share a name with a package.

Following the policies outlined in "Guidelines for Writing an IANA Considerations Section in RFCs" [[RFC2434](#)], normal event package identification tokens are allocated as First Come First Served, and event template-package identification tokens are allocated on a IETF Consensus basis.

Registrations with the IANA MUST include the token being registered and whether the token is a package or a template-package. Further, packages MUST include contact information for the party responsible for the registration and/or a published document which describes the event package. Event template-package token registrations MUST include a pointer to the published RFC which defines the event template-package.

Registered tokens to designate packages and template-packages MUST NOT contain the character ".", which is used to separate template-packages from packages.

7.1.1. Registration Information

As this document specifies no package or template-package names, the initial IANA registration for event types will be empty. The remainder of the text in this section gives an example of the type of information to be maintained by the IANA; it also demonstrates all five possible permutations of package type, contact, and reference.

The table below lists the event packages and template-packages defined in "SIP-Specific Event Notification" [RFC xxxx]. Each name is designated as a package or a template-package under "Type".

Package Name	Type	Contact	Reference
-----	----	-----	-----
example1	package	[Roach]	
example2	package	[Roach]	[RFC xxxx]
example3	package		[RFC xxxx]
example4	template	[Roach]	[RFC xxxx]
example5	template		[RFC xxxx]

PEOPLE

[Roach] Adam Roach <adam.roach@tekelec.com>

REFERENCES

[RFC xxxx] A.B. Roach, "SIP-Specific Event Notification", RFC XXXX,
Monthname 20XX

7.1.2. Registration Template

To: ietf-sip-events@iana.org

Subject: Registration of new SIP event package

Package Name:

(Package names must conform to the syntax described in
[Section 8.2.1.](#))

Is this registration for a Template Package:

(indicate yes or no)

Published Specification(s):

(Template packages require a published RFC. Other packages may
reference a specification when appropriate).

Person & email address to contact for further information:

7.2. Reason Codes

This document further defines "reason" codes for use in the
"Subscription-State" header field (see [Section 4.1.3](#)).Following the policies outlined in "Guidelines for Writing an IANA
Considerations Section in RFCs" [[RFC2434](#)], new reason codes require a
Standards Action.

Registrations with the IANA include the reason code being registered and a reference to a published document which describes the event package. Insertion of such values takes place as part of the RFC publication process or as the result of inter-SDO liaison activity. New reason codes must conform to the syntax of the ABNF "token" element defined in SIP [[RFC3261](#)].

[RFC4660] defined a new reason code prior to the establishment of an IANA registry. We include its reason code ("badfilter") in the initial list of reason codes to ensure a complete registry.

The IANA registry for reason code will be initialized with the following values:

Reason Code	Reference
-----	-----
deactivated	[RFC xxxx]
probation	[RFC xxxx]
rejected	[RFC xxxx]
timeout	[RFC xxxx]
giveup	[RFC xxxx]
noresource	[RFC xxxx]
invariant	[RFC xxxx]
badfilter	[RFC 4660]

REFERENCES

-
- [RFC xxxx] A.B. Roach, "SIP-Specific Event Notification", RFC XXXX, Monthname 20XX
- [RFC 4660] Khartabil, H., Leppanen, E., Lonnfors, M., and J. Costa-Requena, "Functional Description of Event Notification Filtering", September 2006.

[7.3.](#) Header Field Names

This document registers three new header field names, described elsewhere in this document. These header fields are defined by the following information, which is to be added to the header field sub-registry under <http://www.iana.org/assignments/sip-parameters>.

Header Name: Allow-Events
Compact Form: u

Header Name: Subscription-State
Compact Form: (none)

Header Name: Event

Compact Form: 0

7.4. Response Codes

This document registers two new response codes. These response codes are defined by the following information, which is to be added to the method and response-code sub-registry under <http://www.iana.org/assignments/sip-parameters>.

Response Code Number: 202
Default Reason Phrase: Accepted

Response Code Number: 489
Default Reason Phrase: Bad Event

8. Syntax

This section describes the syntax extensions required for event notification in SIP. Semantics are described in [Section 4](#). Note that the formal syntax definitions described in this document are expressed in the ABNF format used in SIP [[RFC3261](#)], and contain references to elements defined therein.

8.1. New Methods

This document describes two new SIP methods: SUBSCRIBE and NOTIFY.

This table expands on tables 2 and 3 in SIP [[RFC3261](#)].

Header	Where	SUB	NOT
-----	-----	---	---
Accept	R	o	o
Accept	2xx	-	-
Accept	415	o	o
Accept-Encoding	R	o	o
Accept-Encoding	2xx	-	-
Accept-Encoding	415	o	o
Accept-Language	R	o	o
Accept-Language	2xx	-	-
Accept-Language	415	o	o
Alert-Info	R	-	-
Alert-Info	180	-	-
Allow	R	o	o
Allow	2xx	o	o
Allow	r	o	o
Allow	405	m	m
Authentication-Info	2xx	o	o

Authorization	R	o	o
Call-ID	c	m	m
Contact	R	m	m
Contact	1xx	o	o
Contact	2xx	m	o
Contact	3xx	m	m
Contact	485	o	o
Content-Disposition		o	o
Content-Encoding		o	o
Content-Language		o	o
Content-Length		t	t
Content-Type		*	*
CSeq	c	m	m
Date		o	o
Error-Info	300-699	o	o
Expires		o	-
Expires	2xx	m	-
From	c	m	m
In-Reply-To	R	-	-
Max-Forwards	R	m	m
Min-Expires	423	m	-
MIME-Version		o	o
Organization		o	-
Priority	R	o	-
Proxy-Authenticate	407	m	m
Proxy-Authorization	R	o	o
Proxy-Require	R	o	o
RAck	R	-	-
Record-Route	R	o	o
Record-Route	2xx, 401, 484	o	o
Reply-To		-	-
Require		o	o
Retry-After	404, 413, 480, 486	o	o
Retry-After	500, 503	o	o
Retry-After	600, 603	o	o
Route	R	c	c
RSeq	1xx	o	o
Server	r	o	o
Subject	R	-	-
Supported	R	o	o
Supported	2xx	o	o
Timestamp		o	o
To	c(1)	m	m
Unsupported	420	o	o
User-Agent		o	o
Via	c	m	m
Warning	R	-	o
Warning	r	o	o

WWW-Authenticate 401 m m

8.1.1. SUBSCRIBE method

"SUBSCRIBE" is added to the definition of the element "Method" in the SIP message grammar.

Like all SIP method names, the SUBSCRIBE method name is case sensitive. The SUBSCRIBE method is used to request asynchronous notification of an event or set of events at a later time.

8.1.2. NOTIFY method

"NOTIFY" is added to the definition of the element "Method" in the SIP message grammar.

The NOTIFY method is used to notify a SIP node that an event which has been requested by an earlier SUBSCRIBE method has occurred. It may also provide further details about the event.

8.2. New Header Fields

This table expands on tables 2 and 3 in SIP [[RFC3261](#)], as amended by the changes described in [Section 8.1](#).

Header field	where	proxy	ACK	BYE	CAN	INV	OPT	REG	PRA	SUB	NOT
Allow-Events	R		0	0	-	0	0	0	0	0	0
Allow-Events	2xx		-	0	-	0	0	0	0	0	0
Allow-Events	489		-	-	-	-	-	-	-	m	m
Event	R		-	-	-	-	-	-	-	m	m
Subscription-State	R		-	-	-	-	-	-	-	-	m

8.2.1. "Event" Header Field

Event is added to the definition of the element "message-header field" in the SIP message grammar.

For the purposes of matching NOTIFY messages with SUBSCRIBE messages, the event-type portion of the "Event" header field is compared byte-by-byte, and the "id" parameter token (if present) is compared byte-by-byte. An "Event" header field containing an "id" parameter never matches an "Event" header field without an "id" parameter. No other parameters are considered when performing a comparison. SUBSCRIBE responses are matched per the transaction handling rules in SIP [[RFC3261](#)].

Note that the forgoing text means that "Event: foo; id=1234" would match "Event: foo; param=abcd; id=1234", but not "Event: foo" (id does not match) or "Event: Foo; id=1234" (event portion does not match).

This document does not define values for event-types. These values will be defined by individual event packages, and MUST be registered with the IANA.

There MUST be exactly one event type listed per event header field. Multiple events per message are disallowed.

8.2.2. "Allow-Events" Header Field

Allow-Events is added to the definition of the element "general-header field" in the SIP message grammar. Its usage is described in [Section 4.4.4](#).

8.2.3. "Subscription-State" Header Field

Subscription-State is added to the definition of the element "request-header field" in the SIP message grammar. Its usage is described in [Section 4.1.3](#).

8.3. New Response Codes

8.3.1. "202 Accepted" Response Code

The 202 response is added to the "Success" header field definition. "202 Accepted" has the same meaning as that defined in HTTP/1.1 [[RFC2616](#)].

8.3.2. "489 Bad Event" Response Code

The 489 event response is added to the "Client-Error" header field definition. "489 Bad Event" is used to indicate that the server did not understand the event package specified in a "Event" header field.

8.4. Augmented BNF Definitions

The Augmented BNF definitions for the various new and modified syntax elements follows. The notation is as used in SIP [[RFC3261](#)], and any elements not defined in this section are as defined in SIP and the documents to which it refers.

SUBSCRIBE_m = %x53.55.42.53.43.52.49.42.45 ; SUBSCRIBE in caps
NOTIFY_m = %x4E.4F.54.49.46.59 ; NOTIFY in caps
extension-method = SUBSCRIBE_m / NOTIFY_m / token

Event = ("Event" / "o") HCOLON event-type
*(SEMI event-param)
event-type = event-package *("." event-template)
event-package = token-nodot
event-template = token-nodot
token-nodot = 1*(alphanum / "-" / "!" / "%" / "*" / "_" / "+" / "`" / "'" / "~")

; The use of the "id" parameter is deprecated; it is included
; for backwards compatibility purposes only.

event-param = generic-param / ("id" EQUAL token)

Allow-Events = ("Allow-Events" / "u") HCOLON event-type
*(COMMA event-type)

Subscription-State = "Subscription-State" HCOLON substate-value
*(SEMI subexp-params)
substate-value = "active" / "pending" / "terminated"
/ extension-substate
extension-substate = token
subexp-params = ("reason" EQUAL event-reason-value)
/ ("expires" EQUAL delta-seconds)
/ ("retry-after" EQUAL delta-seconds)
/ generic-param
event-reason-value = "deactivated"
/ "probation"
/ "rejected"
/ "timeout"
/ "giveup"
/ "noresource"
/ "invariant"
/ event-reason-extension
event-reason-extension = token

9. References

9.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.
- [RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", [BCP 26](#), [RFC 2434](#),

October 1998.

- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", [RFC 2616](#), June 1999.
- [RFC2779] Day, M., Aggarwal, S., Mohr, G., and J. Vincent, "Instant Messaging / Presence Protocol Requirements", [RFC 2779](#), February 2000.
- [RFC2848] Petrack, S. and L. Conroy, "The PINT Service Protocol: Extensions to SIP and SDP for IP Access to Telephone Call Services", [RFC 2848](#), June 2000.
- [RFC3261] 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.
- [RFC3265] Roach, A., "Session Initiation Protocol (SIP)-Specific Event Notification", [RFC 3265](#), June 2002.
- [RFC4483] Burger, E., "A Mechanism for Content Indirection in Session Initiation Protocol (SIP) Messages", [RFC 4483](#), May 2006.
- [I-D.ietf-sip-gruu]
Rosenberg, J., "Obtaining and Using Globally Routable User Agent (UA) URIs (GRUU) in the Session Initiation Protocol (SIP)", [draft-ietf-sip-gruu-15](#) (work in progress), October 2007.

9.2. Informative References

- [RFC3515] Sparks, R., "The Session Initiation Protocol (SIP) Refer Method", [RFC 3515](#), April 2003.
- [RFC3840] Rosenberg, J., Schulzrinne, H., and P. Kyzivat, "Indicating User Agent Capabilities in the Session Initiation Protocol (SIP)", [RFC 3840](#), August 2004.
- [RFC3903] Niemi, A., "Session Initiation Protocol (SIP) Extension for Event State Publication", [RFC 3903](#), October 2004.
- [RFC4485] Rosenberg, J. and H. Schulzrinne, "Guidelines for Authors of Extensions to the Session Initiation Protocol (SIP)", [RFC 4485](#), May 2006.

- [RFC4660] Khartabil, H., Leppanen, E., Lonnfors, M., and J. Costa-Requena, "Functional Description of Event Notification Filtering", [RFC 4660](#), September 2006.
- [RFC5057] Sparks, R., "Multiple Dialog Usages in the Session Initiation Protocol", [RFC 5057](#), November 2007.

[Appendix A](#). Acknowledgements

Thanks to the participants in the Events BOF at the 48th IETF meeting in Pittsburgh, as well as those who gave ideas and suggestions on the SIP Events mailing list. In particular, I wish to thank Henning Schulzrinne of Columbia University for coming up with the final three-tiered event identification scheme, Sean Olson for miscellaneous guidance, Jonathan Rosenberg for a thorough scrubbing of the -00 draft, and the authors of the "SIP Extensions for Presence" document for their input to SUBSCRIBE and NOTIFY request semantics.

I also owe a debt of gratitude to all the implementors who have provided feedback on areas of confusion or difficulty in the original specification. In particular, Robert Sparks' Herculean efforts organizing, running, and collecting data from the SIPit events have proven invaluable in shaking out specification bugs. Robert Sparks is also responsible for untangling the dialog usage mess, in the form of [RFC 5057](#).

[Appendix B](#). Open Issues

[B.1](#). Bug 711: Allow-Events can't express template support

OPEN ISSUE: There are several things we can do here. I have not proposed on in particular; I would prefer to solicit initial feedback from implementors regarding what has been developed and deployed so far.

The key problem is that support of template event packages can't be expressed in a complete yet bounded fashion. It would not be reasonable, for example, to require notifiers that support winfo on arbitrary packages to include an "Allow-Events" header field with contents like "presence, presence.winfo, presence.winfo.winfo, presence.winfo.winfo.winfo, presence.winfo.winfo.winfo.winfo, presence.winfo.winfo.winfo.winfo.winfo...."

One alternative would be to list event packages and template event packages, without explicitly indicating which templates can be

applied to which other packages. In such a case, the preceding example would be collapsed to "Allow-Events: presence, winfo". The notifier may have local policy that limits how they can be combined -- but we have plenty of other places where protocol allows something, but policy forbids it.

B.2. Remove 202 Response Code?

In practice, the 202 response code defined in [RFC 3265](#) has proven to be nearly useless, due to its redundancy with the "pending" state, and its interaction with the HERFP problem. Given that 202 must be treated as 200 if an implementation does not understand it: would removing the 202 response code cause any issues for current implementations?

B.3. Timer L and Resubscribes

[Section 4.1.2.4](#) defines a new Timer L that is used upon initial subscription to bound the amount of time that a subscriber needs to wait for a NOTIFY. Should this also apply to resubscribes? On one hand, the mechanism is not as necessary, since the subscriber already has a negotiated expiration time associated with the subscription. On the other hand, if no NOTIFY arrives in $64 \cdot T1$, it is highly likely that the notifier has gone off the rails, which means that the subscriber can safely clean up state associated with that subscription. The key question involved in applying Timer L to resubscriptions is whether doing so makes subscriptions unnecessarily brittle.

Appendix C. Changes

This section, and all of its subsections, will be consolidated into a single "Changes Since [RFC 3265](#)" section prior to publication. Bug numbers refer to the identifiers for the bug reports kept on file at <http://bugs.sipit.net/>.

C.1. Changes since [RFC 3265](#)

C.1.1. Bug 666: Clarify use of expires=xxx with terminated

Strengthened language in [Section 4.1.3](#) to clarify that expires should not be sent with terminated, and must be ignored if received.

C.1.2. Bug 667: Reason code for unsub/poll not clearly spelled out

Clarified description of "timeout" in [Section 4.1.3](#). (n.b., the text in [Section 4.4.3](#) is actually pretty clear about this).

[C.1.3.](#) Bug 669: Clarify: SUBSCRIBE for a duration might be answered with a NOTIFY/expires=0

Added clarifying text to [Section 4.2.2](#) explaining that shortening a subscription to zero seconds is valid. Also added sentence to [Section 3.1.1](#) explicitly allowing shortening to zero.

[C.1.4.](#) Bug 670: Dialog State Machine needs clarification

The issues associated with the bug deal exclusively with the handling of multiple usages with a dialog. This behavior has been deprecated and moved to [Section 4.5.2](#). This section, in turn, cites [[RFC5057](#)], which addresses all of the issues in Bug 670.

[C.1.5.](#) Bug 671: Clarify timeout-based removal of subscriptions

Changed [Section 4.2.2](#) to specifically cite Timer F (so as to avoid ambiguity between transaction timeouts and retransmission timeouts).

[C.1.6.](#) Bug 672: Mandate expires= in NOTIFY

Changed strength of including of "expires" in a NOTIFY from SHOULD to MUST in [Section 4.2.2](#).

[C.1.7.](#) Bug 673: INVITE 481 response effect clarification

This bug was addressed in [[RFC5057](#)].

[C.1.8.](#) Bug 677: SUBSCRIBE response matching text in error

Fixed [Section 8.2.1](#) to remove incorrect "...responses and..." -- explicitly pointed to SIP for transaction response handling.

[C.1.9.](#) Bug 695: Document is not explicit about response to NOTIFY at subscription termination

Added text to [Section 4.4.1](#) indicating that the typical response to a terminal NOTIFY is a "200 OK".

[C.1.10.](#) Bug 696: Subscription state machine needs clarification

Added state machine diagram to [Section 4.1.2](#) with explicit handling of what to do when a SUBSCRIBE never shows up. Added definition of and handling for new Timer L to [Section 4.1.2.4](#). Added state machine to [Section 4.2.2](#) to reinforce text.

[C.1.11.](#) Bug 697: Unsubscription behavior could be clarified

Added text to [Section 4.2.1.4](#) encouraging (but not requiring) full state in final NOTIFY message. Also added text to [Section 4.1.2.3](#) warning subscribers that full state may or may not be present in the final NOTIFY.

[C.1.12.](#) Bug 699: NOTIFY and SUBSCRIBE are target refresh requests

Added text to both [Section 3.1](#) and [Section 3.2](#) explicitly indicating that SUBSCRIBE and NOTIFY are target refresh methods.

[C.1.13.](#) Bug 722: Inconsistent 423 reason phrase text

Changed reason code to "Interval Too Brief" in [Section 4.2.1.1](#) and [Section 4.2.1.4](#), to match 423 reason code in SIP [[RFC3261](#)].

[C.1.14.](#) Bug 741: guidance needed on when to not include Allow-Events

Added non-normative clarification to [Section 4.4.4](#) regarding inclusion of Allow-Events in a NOTIFY for the one-and-only package supported by the notifier.

[C.1.15.](#) Bug 744: 5xx to NOTIFY terminates a subscription, but should not

Issue of subscription (usage) termination versus dialog termination is handled in [[RFC5057](#)]. The text in [Section 4.2.2](#) has been updated to summarize the behavior described by 5057, and cites it for additional detail and rationale.

[C.1.16.](#) Bug 752: Detection of forked requests is incorrect

Removed erroneous "CSeq" from list of matching criteria in [Section 5.4.9](#).

[C.1.17.](#) Bug 773: Reason code needs IANA registry

Added [Section 7.2](#) to create and populate IANA registry.

[C.1.18.](#) Bug 774: Need new reason for terminating subscriptions to resources that never change

Added new "invariant" reason code to [Section 4.1.3](#), ABNF syntax.

[C.1.19.](#) Clarify handling of Route/Record-Route in NOTIFY

Changed text in [Section 4.3](#) mandating Record-Route in initial SUBSCRIBE and all NOTIFY messages, and adding "MAY" level statements for subsequent SUBSCRIBE messages.

[C.1.20.](#) Eliminate implicit subscriptions

Added text to [Section 4.2.1](#) explaining some of the problems associated with implicit subscriptions, normative language prohibiting them. Removed language from [Section 3.2](#) describing "non-SUBSCRIBE" mechanisms for creating subscriptions. Simplified language in [Section 4.2.2](#), now that the soft-state/non-soft-state distinction is unnecessary.

[C.1.21.](#) Deprecate dialog re-use

Moved handling of dialog re-use and "id" handling to [Section 4.5.2](#). It is documented only for backwards-compatibility purposes.

[C.1.22.](#) Rationalize dialog creation

[Section 4.4.1](#) has been updated to specify that dialogs should be created when the NOTIFY arrives. Previously, the dialog was established by the SUBSCRIBE 200, or by the NOTIFY transaction. This was unnecessarily complicated; the newer rules are easier to implement (and result in effectively the same behavior on the wire).

[C.1.23.](#) Refactor behavior sections

Reorganized [Section 4](#) to consolidate behavior along role lines (subscriber/notifier/proxy) instead of method lines.

[C.1.24.](#) Clarify sections that need to be present in event packages

Added sentence to [Section 5](#) clarifying that event packages are expected to include explicit sections covering the issues discussed in this section.

[C.1.25.](#) Make CANCEL handling more explicit

Text in [Section 4.6](#) now clearly calls out behavior upon receipt of a CANCEL. We also echo the "...SHOULD NOT send..." requirement from [\[RFC3261\]](#).

C.1.26. Remove State Agent Terminology

As originally planned, we anticipated a fairly large number of event packages that would move back and forth between end-user devices and servers in the network. In practice, this has ended up not being the case. Certain events, like dialog state, are inherently hosted at end-user devices; others, like presence, are almost always hosted in the network (due to issues like composition, and the ability to deliver information when user devices are offline). Further, the concept of State Agents is the most misunderstood by event package authors. In my expert review of event packages, I have yet to find one that got the concept of State Agents completely correct -- and most of them start out with the concept being 100% backwards from the way [RFC 3265](#) described it.

Rather than remove the ability to perform the actions previously attributed to the widely misunderstood term "State Agent," we have simply eliminated this term. Instead, we talk about the behaviors required to create state agents (state aggregation, subscription notification) without defining a formal term to describe the servers that exhibit these behaviors. In effect, this is an editorial change to make life easier for event package authors; the actual protocol does not change as a result.

The definition of "State Agent" has been removed from [Section 2](#). [Section 4.4.2](#) has been retooled to discuss migration of subscription in general, without calling out the specific example of state agents. [Section 5.4.11](#) has been focused on state aggregation in particular, instead of state aggregation as an aspect of state agents.

Author's Address

Adam Roach
Tekelec
17210 Campbell Rd.
Suite 250
Dallas, TX 75252
US

Email: adam@nostrum.com

