Internet Engineering Task Force (IETF)
Request for Comments: 6121
Obsoletes: 3921
Category: Standards Track
ISSN: 2070-1721

P. Saint-Andre
Cisco

March 2011

Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence

Abstract

This document defines extensions to core features of the Extensible Messaging and Presence Protocol (XMPP) that provide basic instant messaging (IM) and presence functionality in conformance with the requirements in RFC 2779. This document obsoletes RFC 3921.

Status of this Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6121.

Table of Contents

1. Introduction

1.1. Overview

The Extensible Messaging and Presence Protocol (XMPP) is an application profile of the Extensible Markup Language [XML] that enables the near-real-time exchange of structured yet extensible data between any two or more network entities. The core features of XMPP defined in [XMPP-CORE] provide the building blocks for many types of near-real-time applications, which can be layered on top of the core by sending application-specific data qualified by particular XML namespaces (refer to [XML-NAMES]). This document defines XMPP extensions that provide the basic functionality expected of an instant messaging (IM) and presence application as described in [IMP-REQS].

1.2. History

The basic syntax and semantics of XMPP were developed originally within the Jabber open-source community, mainly in 1999. In late 2002, the XMPP Working Group was chartered with developing an adaptation of the core Jabber protocol that would be suitable as an IETF IM and presence technology in accordance with [IMP-REQS]. In October 2004, [RFC3920] and [RFC3921] were published, representing the most complete definition of XMPP at that time.

Since 2004 the Internet community has gained extensive implementation and deployment experience with XMPP, including formal interoperability testing carried out under the auspices of the XMPP Standards Foundation (XSF). This document incorporates comprehensive feedback from software developers and service providers, including a number of backward-compatible modifications summarized under Appendix E. As a result, this document reflects the rough consensus of the Internet community regarding the IM and presence features of XMPP 1.0, thus obsoleting RFC 3921.

1.3. Requirements

Traditionally, IM applications have combined the following factors:

1. The central point of focus is a list of one’s contacts or "buddies" (in XMPP this list is called a "roster").

2. The purpose of using such an application is to exchange relatively brief text messages with particular contacts in close to real time -- often relatively large numbers of such messages in rapid succession, in the form of a one-to-one "chat session" as described under Section 5.1.

3. The catalyst for exchanging messages is "presence" -- i.e., information about the network availability of particular contacts (thus knowing who is online and available for a one-to-one chat session).

4. Presence information is provided only to contacts that one has authorized by means of an explicit agreement called a "presence subscription".

Thus at a high level this document assumes that a user needs to be able to complete the following use cases:

- Manage items in one's contact list

- Exchange messages with one's contacts

- Exchange presence information with one's contacts

- Manage presence subscriptions to and from one's contacts

Detailed definitions of these functionality areas are contained in RFC 2779 [IMP-REQS], and the interested reader is referred to that document regarding in-depth requirements. Although the XMPP IM and presence extensions specified herein meet the requirements of RFC 2779, they were not designed explicitly with that specification in mind, since the base protocol evolved through an open development process within the Jabber open-source community before RFC 2779 was written. Although XMPP protocol extensions addressing many other functionality areas have been defined in the XMPP Standards Foundation's XEP series (e.g., multi-user text chat as specified in [XEP-0045]), such extensions are not specified in this document because they are not mandated by RFC 2779.

> Implementation Note: RFC 2779 stipulates that presence services must be separable from IM services and vice-versa; i.e., it must be possible to use the protocol to provide a presence service, a messaging service, or both. Although the text of this document assumes that implementations and deployments will want to offer a unified IM and presence service, it is not mandatory for an XMPP service to offer both a presence service and a messaging service, and the protocol makes it possible to offer separate and distinct

services for presence and for messaging. (For example, a presence-only service could return a <service-unavailable/> stanza error if a client attempts to send a <message/> stanza.)

1.4. Functional Summary

This non-normative section provides a developer-friendly, functional summary of XMPP-based IM and presence features; consult the sections that follow for a normative definition of these features.

[XMPP-CORE] specifies how an XMPP client connects to an XMPP server. In particular, it specifies the preconditions that need to be fulfilled before a client is allowed to send XML stanzas (the basic unit of meaning in XMPP) to other entities on an XMPP network. These preconditions comprise negotiation of the XML stream and include exchange of XML stream headers, optional channel encryption via Transport Layer Security [TLS], mandatory authentication via Simple Authentication and Security Layer [SASL], and binding of a resource to the stream for client addressing. The reader is referred to [XMPP-CORE] for details regarding these preconditions, and knowledge of [XMPP-CORE] is assumed herein.

Interoperability Note: [RFC3921] specified one additional precondition: formal establishment of an instant messaging and presence session. Implementation and deployment experience has shown that this additional step is unnecessary. However, for backward compatibility an implementation MAY still offer that feature. This enables older software to connect while letting newer software save a round trip.

Upon fulfillment of the preconditions specified in [XMPP-CORE], an XMPP client has a long-lived XML stream with an XMPP server, which enables the user controlling that client to send and receive a potentially unlimited number of XML stanzas over the stream. Such a stream can be used to exchange messages, share presence information, and engage in structured request-response interactions in close to real time. After negotiation of the XML stream, the typical flow for an instant messaging and presence session is as follows:

1. Retrieve one’s roster. (See Section 2.2.)

2. Send initial presence to the server for broadcast to all subscribed contacts, thus "going online" from the perspective of XMPP communication. (See Section 4.2.)

3. Exchange messages, manage presence subscriptions, perform roster updates, and in general process and generate other XML stanzas with particular semantics throughout the life of the session. (See Sections 5, 3, 2, and 6.)

4. Terminate the session when desired by sending unavailable presence and closing the underlying XML stream. (See Section 4.5.)

1.5. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [KEYWORDS].

This document inherits the terminology defined in [XMPP-CORE].

The terms "automated client" and "interactive client" are to be understood in the sense defined in [TLS-CERTS].

For convenience, this document employs the term "user" to refer to the owner of an XMPP account; however, account owners need not be humans and can be bots, devices, or other automated applications.

Several other terms, such as "interested resource", are defined within the body of this document.

Following the "XML Notation" used in [IRI] to represent characters that cannot be rendered in ASCII-only documents, some examples in this document use the form "&#x...." as a notational device to represent [UNICODE] characters (e.g., the string "ř" stands for the Unicode character LATIN SMALL LETTER R WITH CARON); this form is definitely not to be sent over the wire in XMPP systems.

In examples, lines have been wrapped for improved readability, "[...]" means elision, and the following prepended strings are used (these prepended strings are not to be sent over the wire):

- C: = client
- CC: = contact’s client
- CS: = contact’s server
- S: = server
- UC: = user’s client

- US: = user's server

Readers need to be aware that the examples are not exhaustive and that, in examples for some protocol flows, the alternate steps shown would not necessarily be triggered by the exact data sent in the previous step; in all cases, the protocol definitions specified in this document or in normatively referenced documents rule over any examples provided here. All examples are fictional and the information exchanged (e.g., usernames and passwords) does not represent any existing users or servers.

2. Managing the Roster

In XMPP, a user's roster contains any number of specific contacts. A user's roster is stored by the user's server on the user's behalf so that the user can access roster information from any device. When the user adds items to the roster or modifies existing items, if an error does not occur then the server SHOULD store that data unmodified if at all possible and MUST return the data it has stored when an authorized client requests the roster.

> Security Warning: Because the user's roster can contain confidential data, the server MUST restrict access to this data so that only authorized entities (typically limited to the account owner) are able to retrieve, modify, or delete it.

RFC 3921 assumed that the only place where a user stores their roster is the server where the user's account is registered and at which the user authenticates for access to the XMPP network. This specification removes that strict coupling of roster storage to account registration and network authentication, with the result that a user could store their roster at another location, or could have multiple rosters that are stored in multiple locations. However, in the absence of implementation and deployment experience with a more flexible roster storage model, this specification retains the terminology of RFC 3921 by using the terms "client" and "server" (and "the roster" instead of "a roster"), rather than coining a new term for "a place where a user stores a roster". Future documents might provide normative rules for non-server roster storage or for the management of multiple rosters, but such rules are out of scope for this document.

2.1. Syntax and Semantics

Rosters are managed using <iq/> stanzas (see Section 8.2.3 of [XMPP-CORE]), specifically by means of a <query/> child element qualified by the 'jabber:iq:roster' namespace. The detailed syntax and semantics are defined in the following sections.

2.1.1. Ver Attribute

The 'ver' attribute is a string that identifies a particular version of the roster information. The value MUST be generated only by the server and MUST be treated by the client as opaque. The server can use any appropriate method for generating the version ID, such as a hash of the roster data or a strictly increasing sequence number.

Inclusion of the 'ver' attribute is RECOMMENDED.

Use of the 'ver' attribute is described more fully under Section 2.6.

> Interoperability Note: The 'ver' attribute of the <query/> element was not defined in RFC 3921 and is newly defined in this specification.

2.1.2. Roster Items

The <query/> element inside a roster set (Section 2.1.5) contains one <item/> child, and a roster result (Section 2.1.4) typically contains multiple <item/> children. Each <item/> element describes a unique "roster item" (sometimes also called a "contact").

The syntax of the <item/> element is described in the following sections.

2.1.2.1. Approved Attribute

The boolean 'approved' attribute with a value of "true" is used to signal subscription pre-approval as described under Section 3.4 (the default is "false", in accordance with [XML-DATATYPES]).

A server SHOULD include the 'approved' attribute to inform the client of subscription pre-approvals. A client MUST NOT include the 'approved' attribute in the roster sets it sends to the server, but instead MUST use presence stanzas of type "subscribed" and "unsubscribed" to manage pre-approvals as described under Section 3.4.

> Interoperability Note: The 'approved' attribute of the <item/> element was not defined in RFC 3921 and is newly defined in this specification.

2.1.2.2. Ask Attribute

The 'ask' attribute of the <item/> element with a value of "subscribe" is used to signal various subscription sub-states that include a "Pending Out" aspect as described under Section 3.1.2.

A server SHOULD include the 'ask' attribute to inform the client of "Pending Out" sub-states. A client MUST NOT include the 'ask' attribute in the roster sets it sends to the server, but instead MUST use presence stanzas of type "subscribe" and "unsubscribe" to manage such sub-states as described under Section 3.1.2.

2.1.2.3. JID Attribute

The 'jid' attribute of the <item/> element specifies the Jabber Identifier (JID) that uniquely identifies the roster item.

The 'jid' attribute is REQUIRED whenever a client or server adds, updates, deletes, or returns a roster item.

2.1.2.4. Name Attribute

The 'name' attribute of the <item/> element specifies the "handle" to be associated with the JID, as determined by the user (not the contact). Although the value of the 'name' attribute MAY have meaning to a human user, it is opaque to the server. However, the 'name' attribute MAY be used by the server for matching purposes within the context of various XMPP extensions (one possible comparison method is that described for XMPP resourceparts in [XMPP-ADDR]).

It is OPTIONAL for a client to include the 'name' attribute when adding or updating a roster item.

2.1.2.5. Subscription Attribute

The state of the presence subscription is captured in the 'subscription' attribute of the <item/> element. The defined subscription-related values are:

none: the user does not have a subscription to the contact's presence, and the contact does not have a subscription to the user's presence; this is the default value, so if the subscription attribute is not included then the state is to be understood as "none"

to: the user has a subscription to the contact's presence, but the contact does not have a subscription to the user's presence

from: the contact has a subscription to the user's presence, but the user does not have a subscription to the contact's presence

both: the user and the contact have subscriptions to each other's presence (also called a "mutual subscription")

In a roster result (Section 2.1.4), the client MUST ignore values of the 'subscription' attribute other than "none", "to", "from", or "both".

In a roster push (Section 2.1.6), the client MUST ignore values of the 'subscription' attribute other than "none", "to", "from", "both", or "remove".

In a roster set (Section 2.1.5), the 'subscription' attribute MAY be included with a value of "remove", which indicates that the item is to be removed from the roster; in a roster set the server MUST ignore all values of the 'subscription' attribute other than "remove".

Inclusion of the 'subscription' attribute is OPTIONAL.

2.1.2.6. Group Element

The <group/> child element specifies a category or "bucket" into which the roster item is to be grouped by a client. An <item/> element MAY contain more than one <group/> element, which means that roster groups are not exclusive. Although the XML character data of the <group/> element MAY have meaning to a human user, it is opaque to the server. However, the <group/> element MAY be used by the server for matching purposes within the context of various XMPP extensions (one possible comparison method is that described for XMPP resourceparts in [XMPP-ADDR]).

It is OPTIONAL for a client to include the <group/> element when adding or updating a roster item. If a roster set (Section 2.1.5) includes no <group/> element, then the item is to be interpreted as being affiliated with no group.

2.1.3. Roster Get

A "roster get" is a client's request for the server to return the roster; syntactically it is an IQ stanza of type "get" sent from client to server and containing a <query/> element qualified by the 'jabber:iq:roster' namespace, where the <query/> element MUST NOT contain any <item/> child elements.

```
C: <iq from='juliet@example.com/balcony'
       id='bv1bs71f'
       type='get'>
     <query xmlns='jabber:iq:roster'/>
   </iq>
```

The expected outcome of sending a roster get is for the server to return a roster result.

2.1.4. Roster Result

A "roster result" is the server's response to a roster get; syntactically it is an IQ stanza of type "result" sent from server to client and containing a <query/> element qualified by the 'jabber:iq:roster' namespace.

The <query/> element in a roster result contains one <item/> element for each contact and therefore can contain more than one <item/> element.

```
S: <iq id='bv1bs71f'
       to='juliet@example.com/chamber'
       type='result'>
    <query xmlns='jabber:iq:roster' ver='ver7'>
      <item jid='nurse@example.com'/>
      <item jid='romeo@example.net'/>
    </query>
  </iq>
```

If the roster exists but there are no contacts in the roster, then the server MUST return an IQ-result containing a child <query/> element that in turn contains no <item/> children (i.e., the server MUST NOT return an empty <iq/> stanza of type "error").

```
S: <iq id='bv1bs71f'
       to='juliet@example.com/chamber'
       type='result'>
    <query xmlns='jabber:iq:roster' ver='ver9'/>
  </iq>
```

If the roster does not exist, then the server MUST return a stanza error with a condition of <item-not-found/>.

```
S: <iq id='bv1bs71f'
       to='juliet@example.com/chamber'
       type='error'>
    <error type='cancel'>
      <item-not-found
          xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
    </error>
  </iq>
```

2.1.5. Roster Set

A "roster set" is a client's request for the server to modify (i.e., create, update, or delete) a roster item; syntactically it is an IQ stanza of type "set" sent from client to server and containing a <query/> element qualified by the 'jabber:iq:roster' namespace.

The following rules apply to roster sets:

1. The <query/> element MUST contain one and only one <item/> element.

2. The server MUST ignore any value of the 'subscription' attribute other than "remove" (see Section 2.1.2.5).

Security Warning: Traditionally, the IQ stanza of the roster set included no 'to' address, with the result that all roster sets were sent from an authenticated resource (full JID) of the account whose roster was being updated. Furthermore, RFC 3921 required a server to perform special-case checking of roster sets to ignore the 'to' address; however, this specification has removed that special-casing, which means that a roster set might include a 'to' address other than that of the sender. Therefore, the entity that processes a roster set MUST verify that the sender of the roster set is authorized to update the roster, and if not return a <forbidden/> error.

```
C: <iq from='juliet@example.com/balcony'
       id='rs1'
       type='set'>
    <query xmlns='jabber:iq:roster'>
      <item jid='nurse@example.com'/>
    </query>
  </iq>
```

2.1.6. Roster Push

A "roster push" is a newly created, updated, or deleted roster item that is sent from the server to the client; syntactically it is an IQ stanza of type "set" sent from server to client and containing a <query/> element qualified by the 'jabber:iq:roster' namespace.

The following rules apply to roster pushes:

1. The <query/> element in a roster push MUST contain one and only one <item/> element.

2. A receiving client MUST ignore the stanza unless it has no 'from' attribute (i.e., implicitly from the bare JID of the user's account) or it has a 'from' attribute whose value matches the user's bare JID <user@domainpart>.

```
S: <iq id='a78b4q6ha463'
       to='juliet@example.com/chamber'
       type='set'>
    <query xmlns='jabber:iq:roster'>
      <item jid='nurse@example.com'/>
    </query>
  </iq>
```

As mandated by the semantics of the IQ stanza as defined in [XMPP-CORE], each resource that receives a roster push from the server is supposed to reply with an IQ stanza of type "result" or "error" (however, it is known that many existing clients do not reply to roster pushes).

```
C: <iq from='juliet@example.com/balcony'
       id='a78b4q6ha463'
       type='result'/>

C: <iq from='juliet@example.com/chamber'
       id='a78b4q6ha463'
       type='result'/>
```

Security Warning: Traditionally, a roster push included no 'from' address, with the result that all roster pushes were sent implicitly from the bare JID of the account itself. However, this specification allows entities other than the user's server to maintain roster information, which means that a roster push might include a 'from' address other than the bare JID of the user's account. Therefore, the client MUST check the 'from' address to verify that the sender of the roster push is authorized to update the roster. If the client receives a roster push from an unauthorized entity, it MUST NOT process the pushed data; in addition, the client can either return a stanza error of <service-unavailable/> error or refuse to return a stanza error at all (the latter behavior overrides a MUST-level requirement from [XMPP-CORE] for the purpose of preventing a presence leak).

Implementation Note: There is no error case for client processing of roster pushes; if the server receives an IQ of type "error" in response to a roster push then it SHOULD ignore the error.

2.2. Retrieving the Roster on Login

Upon authenticating with a server and binding a resource (thus becoming a connected resource as defined in [XMPP-CORE]), a client SHOULD request the roster before sending initial presence (however, because receiving the roster is not necessarily desirable for all resources, e.g., a connection with limited bandwidth, the client's request for the roster is not mandatory). After a connected resource sends initial presence (see Section 4.2), it is referred to as an "available resource". If a connected resource or available resource requests the roster, it is referred to as an "interested resource". The server MUST send roster pushes to all interested resources.

> Implementation Note: Presence subscription requests are sent to available resources, whereas the roster pushes associated with subscription state changes are sent to interested resources. Therefore, if a resource wishes to receive both subscription requests and roster pushes, it MUST both send initial presence and request the roster.

A client requests the roster by sending a roster get over its stream with the server.

```
C: <iq from='juliet@example.com/balcony'
       id='hu2bac18'
       type='get'>
     <query xmlns='jabber:iq:roster'/>
   </iq>


S: <iq id='hu2bac18'
       to='juliet@example.com/balcony'
       type='result'>
     <query xmlns='jabber:iq:roster' ver='ver11'>
       <item jid='romeo@example.net'
             name='Romeo'
             subscription='both'>
         <group>Friends</group>
       </item>
       <item jid='mercutio@example.com'
             name='Mercutio'
             subscription='from'/>
       <item jid='benvolio@example.net'
             name='Benvolio'
             subscription='both'/>
     </query>
   </iq>
```

If the server cannot process the roster get, it MUST return an appropriate stanza error as described in [XMPP-CORE] (such as <service-unavailable/> if the roster namespace is not supported or <internal-server-error/> if the server experiences trouble processing or returning the roster).

2.3. Adding a Roster Item

2.3.1. Request

At any time, a client can add an item to the roster. This is done by sending a roster set containing a new item.

```
C: <iq from='juliet@example.com/balcony'
       id='ph1xaz53'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             name='Nurse'>
         <group>Servants</group>
       </item>
     </query>
   </iq>
```

2.3.2. Success Case

If the server can successfully process the roster set for the new item (i.e., if no error occurs), it MUST create the item in the user's roster and proceed as follows.

The server MUST return an IQ stanza of type "result" to the connected resource that sent the roster set.

```
S: <iq id='ph1xaz53'
       to='juliet@example.com/balcony'
       type='result'/>
```

The server MUST also send a roster push containing the new roster item to all of the user's interested resources, including the resource that generated the roster set.

```
S: <iq to='juliet@example.com/balcony'
       id='a78b4q6ha463'
       type='set'>
     <query xmlns='jabber:iq:roster' ver='ver13'>
       <item jid='nurse@example.com'
             name='Nurse'
             subscription='none'>
         <group>Servants</group>
       </item>
     </query>
   </iq>

S: <iq to='juliet@example.com/chamber'
       id='x81g3bdy4n19'
       type='set'>
     <query xmlns='jabber:iq:roster' ver='ver13'>
       <item jid='nurse@example.com'
             name='Nurse'
             subscription='none'>
         <group>Servants</group>
       </item>
     </query>
   </iq>
```

As mandated by the semantics of the IQ stanza as defined in [XMPP-CORE], each resource that receives a roster push from the server is supposed to reply with an IQ stanza of type "result" or "error" (however, it is known that many existing clients do not reply to roster pushes).

```
C: <iq from='juliet@example.com/balcony'
       id='a78b4q6ha463'
       type='result'/>

C: <iq from='juliet@example.com/chamber'
       id='x81g3bdy4n19'
       type='result'/>
```

2.3.3. Error Cases

If the server cannot successfully process the roster set, it MUST return a stanza error. The following error cases are defined. Naturally, other stanza errors can occur, such as <internal-server-error/> if the server experiences an internal problem with processing the roster get, or even <not-allowed/> if the server only allows roster modifications by means of a non-XMPP method such as a web interface.

The server MUST return a <forbidden/> stanza error to the client if the sender of the roster set is not authorized to update the roster (where typically only an authenticated resource of the account itself is authorized).

The server MUST return a <bad-request/> stanza error to the client if the roster set contains any of the following violations:

1. The <query/> element contains more than one <item/> child element.

2. The <item/> element contains more than one <group/> element, but there are duplicate groups (one possible comparison method for determining duplicates is that described for XMPP resourceparts in [XMPP-ADDR]).

The server MUST return a <not-acceptable/> stanza error to the client if the roster set contains any of the following violations:

1. The length of the 'name' attribute is greater than a server-configured limit.

2. The XML character data of the <group/> element is of zero length (to remove an item from all groups, the client instead needs to exclude any <group/> element from the roster set).

3. The XML character data of the <group/> element is larger than a server-configured limit.

Error: Roster set initiated by unauthorized entity

```
C: <iq from='juliet@example.com/balcony'
       id='ix7s53v2'
       to='romeo@example.net'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'/>
     </query>
   </iq>

S: <iq id='ix7s53v2'
       to='juliet@example.com/balcony'
       type='error'>
    <error type='auth'>
      <forbidden xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
    </error>
  </iq>
```

Error: Roster set contains more than one item

```
C: <iq from='juliet@example.com/balcony'
       id='nw83vcj4'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             name='Nurse'>
         <group>Servants</group>
       </item>
       <item jid='mother@example.com'
             name='Mom'>
         <group>Family</group>
       </item>
     </query>
   </iq>

S: <iq id='nw83vcj4'
       to='juliet@example.com/balcony'
       type='error'>
    <error type='modify'>
      <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
    </error>
  </iq>
```

Error: Roster set contains item with oversized handle

```
C: <iq from='juliet@example.com/balcony'
       id='yl491b3d'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             name='[ ... some-very-long-handle ... ]'>
         <group>Servants</group>
       </item>
     </query>
   </iq>

S: <iq id='yl491b3d'
       to='juliet@example.com/balcony'
       type='error'>
     <error type='modify'>
       <not-acceptable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

Error: Roster set contains duplicate groups

```
C: <iq from='juliet@example.com/balcony'
       id='tk3va749'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             name='Nurse'>
         <group>Servants</group>
         <group>Servants</group>
       </item>
     </query>
   </iq>

S: <iq id='tk3va749'
       to='juliet@example.com/balcony'
       type='error'>
     <error type='modify'>
       <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

Error: Roster set contains empty group

```
C: <iq from='juliet@example.com/balcony'
       id='fl3b486u'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             name='Nurse'>
         <group></group>
       </item>
     </query>
   </iq>

S: <iq id='fl3b486u'
       to='juliet@example.com/balcony'
       type='error'>
     <error type='modify'>
       <not-acceptable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

Error: Roster set contains oversized group name

```
C: <iq from='juliet@example.com/balcony'
       id='qh3b4v19'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             name='Nurse'>
         <group>[ ... some-very-long-group-name ... ]</group>
       </item>
     </query>
   </iq>

S: <iq id='qh3b4v19'
       to='juliet@example.com/balcony'
       type='error'>
     <error type='modify'>
       <not-acceptable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

Interoperability Note: Some servers return a <not-allowed/> stanza error to the client if the value of the <item/> element's 'jid' attribute matches the bare JID <localpart@domainpart> of the user's account.

2.4. Updating a Roster Item

2.4.1. Request

Updating an existing roster item is done in the same way as adding a new roster item, i.e., by sending a roster set to the server. Because a roster item is atomic, the item MUST be updated exactly as provided in the roster set.

There are several reasons why a client might update a roster item:

1. Adding a group
2. Deleting a group
3. Changing the handle
4. Deleting the handle

Consider a roster item that is defined as follows:

```
    <item jid='romeo@example.net'
          name='Romeo'>
      <group>Friends</group>
    </item>
```

The user who has this item in her roster might want to add the item to another group.

```
C: <iq from='juliet@example.com/balcony'
       id='di43b2x9'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='romeo@example.net'
             name='Romeo'>
         <group>Friends</group>
         <group>Lovers</group>
       </item>
     </query>
   </iq>
```

Sometime later, the user might want to remove the item from the original group.

```
C: <iq from='juliet@example.com/balcony'
       id='lf72v157'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='romeo@example.net'
             name='Romeo'>
         <group>Lovers</group>
       </item>
     </query>
   </iq>
```

The user might want to remove the item from all groups.

```
C: <iq from='juliet@example.com/balcony'
       id='ju4b62a5'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='romeo@example.net'/>
     </query>
   </iq>
```

The user might also want to change the handle for the item.

```
C: <iq from='juliet@example.com/balcony'
       id='gb3sv487'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='romeo@example.net'
             name='MyRomeo'/>
     </query>
   </iq>
```

The user might then want to remove the handle altogether.

```
C: <iq from='juliet@example.com/balcony'
       id='o3bx66s5'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='romeo@example.net'
             name=''/>
     </query>
   </iq>
```

Implementation Note: Including an empty 'name' attribute is equivalent to including no 'name' attribute; both actions set the name to the empty string.

2.4.2. Success Case

As with adding a roster item, if the roster item can be successfully processed then the server MUST update the item in the user's roster, send a roster push to all of the user's interested resources, and send an IQ result to the initiating resource; details are provided under Section 2.3.

2.4.3. Error Cases

The error cases described under Section 2.3.3 also apply to updating a roster item.

2.5. Deleting a Roster Item

2.5.1. Request

At any time, a client can delete an item from his or her roster by sending a roster set and specifying a value of "remove" for the 'subscription' attribute.

```
C: <iq from='juliet@example.com/balcony'
       id='hm4hs97y'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='nurse@example.com'
             subscription='remove'/>
     </query>
   </iq>
```

2.5.2. Success Case

As with adding a roster item, if the server can successfully process the roster set then it MUST update the item in the user's roster, send a roster push to all of the user's interested resources (with the 'subscription' attribute set to a value of "remove"), and send an IQ result to the initiating resource; details are provided under Section 2.3.

In addition, the user's server might need to generate one or more subscription-related presence stanzas, as follows:

1. If the user has a presence subscription to the contact, then the user's server MUST send a presence stanza of type "unsubscribe" to the contact (in order to unsubscribe from the contact's presence).

2. If the contact has a presence subscription to the user, then the user's server MUST send a presence stanza of type "unsubscribed" to the contact (in order to cancel the contact's subscription to the user).

3. If the presence subscription is mutual, then the user's server MUST send both a presence stanza of type "unsubscribe" and a presence stanza of type "unsubscribed" to the contact.

```
S: <presence from='juliet@example.com'
             id='lm3ba81g'
             to='nurse@example.com'
             type='unsubscribe'/>

S: <presence from='juliet@example.com'
             id='xb2c1v4k'
             to='nurse@example.com'
             type='unsubscribed'/>
```

2.5.3. Error Cases

If the value of the 'jid' attribute specifies an item that is not in the roster, then the server MUST return an <item-not-found/> stanza error.

Error: Roster item not found

```
C: <iq from='juliet@example.com/balcony'
       id='uj4b1ca8'
       type='set'>
     <query xmlns='jabber:iq:roster'>
       <item jid='[ ... non-existent-jid ... ]'
             subscription='remove'/>
     </query>
   </iq>

S: <iq id='uj4b1ca8'
       to='juliet@example.com/balcony'
       type='error'>
     <error type='modify'>
       <item-not-found
           xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
     </error>
   </iq>
```

2.6. Roster Versioning

2.6.1. Stream Feature

If a server supports roster versioning, then it MUST advertise the following stream feature during stream negotiation.

```
<ver xmlns='urn:xmpp:features:rosterver'/>
```

The roster versioning stream feature is merely informative and therefore is never mandatory-to-negotiate.

2.6.2. Request

If a client supports roster versioning and the server to which it has connected advertises support for roster versioning as described in the foregoing section, then the client SHOULD include the 'ver' element in its request for the roster. If the server does not advertise support for roster versioning, the client MUST NOT include the 'ver' attribute. If the client includes the 'ver' attribute in its roster get, it sets the attribute's value to the version ID associated with its last cache of the roster.

```
C: <iq from='romeo@example.net/home'
       id='r1h3vzp7'
       to='romeo@example.net'
       type='get'>
     <query xmlns='jabber:iq:roster' ver='ver14'/>
   </iq>
```

If the client has not yet cached the roster or the cache is lost or corrupted, but the client wishes to bootstrap the use of roster versioning, it MUST set the 'ver' attribute to the empty string (i.e., ver="").

Naturally, if the client does not support roster versioning or does not wish to bootstrap the use of roster versioning, it will not include the 'ver' attribute.

2.6.3. Success Case

Whether or not the roster has been modified since the version ID enumerated by the client, the server MUST either return the complete roster as described under Section 2.1.4 (including a 'ver' attribute that signals the latest version) or return an empty IQ-result (thus indicating that any roster modifications will be sent via roster pushes, as described below). In general, unless returning the complete roster would (1) use less bandwidth than sending individual roster pushes to the client (e.g., if the roster contains only a few items) or (2) the server cannot associate the version ID with any previous version it has on file, the server SHOULD send an empty IQ-result and then send the modifications (if any) via roster pushes.

```
S: <iq from='romeo@example.net'
       id='r1h3vzp7'
       to='romeo@example.net/home'
       type='result'/>
```

Implementation Note: This empty IQ-result is different from an empty <query/> element, thus disambiguating this usage from an empty roster.

If roster versioning is enabled and the roster has not been modified since the version ID enumerated by the client, the server will simply not send any roster pushes to the client (until and unless some relevant event triggers a roster push during the lifetime of the client's session).

If the roster has been modified since the version ID enumerated by the client, the server MUST then send one roster push to the client for each roster item that has been modified since the version ID enumerated by the client. (We call a roster push that is sent for purposes of roster version synchronization an "interim roster push".)

Definition: A "roster modification" is any change to the roster data that would result in a roster push to a connected client. Therefore, internal states related to roster processing within the server that would not result in a roster push to a connected client do not necessitate a change to the version.

```
S: <iq from='romeo@example.net'
       id='ah382g67'
       to='romeo@example.net/home'
       type='set'>
     <query xmlns='jabber:iq:roster' ver='ver34'>
       <item jid='tybalt@example.org' subscription='remove'/>
     </query>
   </iq>

S: <iq from='romeo@example.net'
       id='b2gs90j5'
       to='romeo@example.net/home'
       type='set'>
     <query xmlns='jabber:iq:roster' ver='ver42'>
       <item jid='bill@example.org' subscription='both'/>
     </query>
   </iq>

S: <iq from='romeo@example.net'
       id='c73gs419'
       to='romeo@example.net/home'
       type='set'>
     <query xmlns='jabber:iq:roster' ver='ver72'>
       <item jid='nurse@example.org'
             name='Nurse'
             subscription='to'>
         <group>Servants</group>
       </item>
     </query>
   </iq>
```

```
S: <iq from='romeo@example.net'
       id='dh361f35'
       to='romeo@example.net/home'
       type='set'>
     <query xmlns='jabber:iq:roster' ver='ver96'>
       <item jid='juliet@example.org'
             name='Juliet'
             subscription='both'>
         <group>VIPs</group>
       </item>
     </query>
   </iq>
```

These "interim roster pushes" can be understood as follows:

1. Imagine that the client had an active presence session for the entire time between its cached roster version (say, "ver14") and the new roster version (say, "ver96").

2. During that time, the client might have received roster pushes related to various roster versions (which might have been, say, "ver51" and "ver79"). However, some of those roster pushes might have contained intermediate updates to the same roster item (e.g., modifications to the subscription state for bill@example.org from "none" to "to" and from "to" to "both").

3. The interim roster pushes would not include all of the intermediate steps, only the final result of all modifications applied to each item while the client was in fact offline (which might have been, say, "ver34", "ver42", "ver72", and "ver96").

The client MUST handle an "interim roster push" in the same way it handles any roster push (indeed, from the client's perspective it cannot tell the difference between an "interim" roster push and a "live" roster push and therefore it has no way of knowing when it has received all of the interim roster pushes). When requesting the roster after reconnection, the client SHOULD request the version associated with the last roster push it received during its previous session, not the version associated with the roster result it received at the start of its previous session.

When roster versioning is enabled, the server MUST include the updated roster version with each roster push. Roster pushes MUST occur in order of modification and the version contained in a roster push MUST be unique. Even if the client has not included the 'ver' attribute in its roster gets or sets, the server SHOULD include the 'ver' attribute on all roster pushes and results that it sends to the client.

> Implementation Note: Guidelines and more detailed examples for roster versioning are provided in [XEP-0237].

3. Managing Presence Subscriptions

In order to protect the privacy of XMPP users, presence information is disclosed only to other entities that a user has approved. When a user has agreed that another entity is allowed to view its presence, the entity is said to have a "subscription" to the user's presence. An entity that has a subscription to a user's presence or to which a user has a presence subscription is called a "contact" (in this document the term "contact" is also used in a less strict sense to refer to a potential contact or any item in a user's roster).

In XMPP, a subscription lasts across presence sessions; indeed, it lasts until the contact unsubscribes or the user cancels the previously granted subscription. (This model is different from that used for presence subscriptions in the Session Initiation Protocol (SIP), as defined in [SIP-PRES].)

Subscriptions are managed within XMPP by sending presence stanzas containing specially defined attributes ("subscribe", "unsubscribe", "subscribed", and "unsubscribed").

> Implementation Note: When a server processes or generates an outbound presence stanza of type "subscribe", "subscribed", "unsubscribe", or "unsubscribed", the server MUST stamp the outgoing presence stanza with the bare JID <localpart@domainpart> of the sending entity, not the full JID <localpart@domainpart/resourcepart>. Enforcement of this rule simplifies the presence subscription model and helps to prevent presence leaks; for information about presence leaks, refer to the security considerations of [XMPP-CORE].

Subscription states are reflected in the rosters of both the user and the contact. This section does not cover every possible case related to presence subscriptions, and mainly narrates the protocol flows for bootstrapping a mutual subscription between a user and a contact. Complete details regarding subscription states can be found under Appendix A.

3.1. Requesting a Subscription

A "subscription request" is a request from a user for authorization to permanently subscribe to a contact's presence information; syntactically it is a presence stanza whose 'type' attribute has a value of "subscribe". A subscription request is generated by a

user's client, processed by the (potential) contact's server, and acted on by the contact via the contact's client. The workflow is described in the following sections.

> Implementation Note: Presence subscription requests are sent to available resources, whereas the roster pushes associated with subscription state changes are sent to interested resources. Therefore, if a resource wishes to receive both subscription requests and roster pushes, it MUST both send initial presence and request the roster.

3.1.1. Client Generation of Outbound Subscription Request

A user's client generates a subscription request by sending a presence stanza of type "subscribe" and specifying a 'to' address of the potential contact's bare JID <contact@domainpart>.

```
UC: <presence id='xk3h1v69'
              to='juliet@example.com'
              type='subscribe'/>
```

When a user sends a presence subscription request to a potential instant messaging and presence contact, the value of the 'to' attribute MUST be a bare JID <contact@domainpart> rather than a full JID <contact@domainpart/resourcepart>, since the desired result is for the user to receive presence from all of the contact's resources, not merely the particular resource specified in the 'to' attribute. Use of bare JIDs also simplifies subscription processing, presence probes, and presence notifications by the user's server and the contact's server.

For tracking purposes, a client SHOULD include an 'id' attribute in a presence subscription request.

> Implementation Note: Many XMPP clients prompt the user for information about the potential contact (e.g., "handle" and desired roster group) when generating an outbound presence subscription request and therefore send a roster set before sending the outbound presence subscription request. This behavior is OPTIONAL, because a client MAY instead wait until receiving the initial roster push from the server before uploading user-provided information about the contact. A server MUST process a roster set and outbound presence subscription request in either order (i.e., in whatever order generated by the client).

3.1.2. Server Processing of Outbound Subscription Request

Upon receiving the outbound presence subscription request, the user's server MUST proceed as follows.

1. Before processing the request, the user's server MUST check the syntax of the JID contained in the 'to' attribute (however, it is known that some existing implementations do not perform this check). If the JID is of the form <contact@domainpart/resourcepart> instead of <contact@domainpart>, the user's server SHOULD treat it as if the request had been directed to the contact's bare JID and modify the 'to' address accordingly. The server MAY also verify that the JID adheres to the format defined in [XMPP-ADDR] and possibly return a <jid-malformed/> stanza error.

2. If the potential contact is hosted on the same server as the user, then the server MUST adhere to the rules specified under Section 3.1.3 when processing the subscription request and delivering it to the (local) contact.

3. If the potential contact is hosted on a remote server, subject to local service policies the user's server MUST then route the stanza to that remote domain in accordance with core XMPP stanza processing rules. (This can result in returning an appropriate stanza error to the user, such as <remote-server-timeout/>.)

As mentioned, before locally delivering or remotely routing the presence subscription request, the user's server MUST stamp the outbound subscription request with the bare JID <user@domainpart> of the user.

```
US: <presence from='romeo@example.net'
              id='xk3h1v69'
              to='juliet@example.com'
              type='subscribe'/>
```

If the presence subscription request cannot be locally delivered or remotely routed (e.g., because the request is malformed, the local contact does not exist, the remote server does not exist, an attempt to contact the remote server times out, or any other error is determined or experienced by the user's server), then the user's server MUST return an appropriate error stanza to the user. An example follows.

```
US: <presence from='juliet@example.com'
              id='xk3h1v69'
              to='romeo@example.net'
              type='error'>
      <error type='modify'>
        <remote-server-not-found
            xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
      </error>
    </presence>
```

After locally delivering or remotely routing the presence subscription request, the user's server MUST then send a roster push to all of the user's interested resources, containing the potential contact with a subscription state of "none" and with notation that the subscription is pending (via an 'ask' attribute whose value is "subscribe").

```
US: <iq id='b89c5r7ib574'
        to='romeo@example.net/foo'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item ask='subscribe'
              jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>

US: <iq id='b89c5r7ib575'
        to='romeo@example.net/bar'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item ask='subscribe'
              jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>
```

If a remote contact does not approve or deny the subscription request within some configurable amount of time, the user's server SHOULD resend the subscription request to the contact based on an implementation-specific algorithm (e.g., whenever a new resource becomes available for the user, or after a certain amount of time has elapsed); this helps to recover from transient, silent errors that might have occurred when the original subscription request was routed to the remote domain. When doing so, it is RECOMMENDED for the server to include an 'id' attribute so that it can track responses to the resent subscription request.

3.1.3. Server Processing of Inbound Subscription Request

Before processing the inbound presence subscription request, the contact's server SHOULD check the syntax of the JID contained in the 'to' attribute. If the JID is of the form <contact@domainpart/resourcepart> instead of <contact@domainpart>, the contact's server SHOULD treat it as if the request had been directed to the contact's bare JID and modify the 'to' address accordingly. The server MAY also verify that the JID adheres to the format defined in [XMPP-ADDR] and possibly return a <jid-malformed/> stanza error.

When processing the inbound presence subscription request, the contact's server MUST adhere to the following rules:

1. Above all, the contact's server MUST NOT automatically approve subscription requests on the contact's behalf -- unless the contact has (a) pre-approved subscription requests from the user as described under Section 3.4, (b) configured its account to automatically approve subscription requests, or (c) accepted an agreement with its service provider that allows automatic approval (for instance, via an employment agreement within an enterprise deployment). Instead, if a subscription request requires approval then the contact's server MUST deliver that request to the contact's available resource(s) for approval or denial by the contact.

2. If the contact exists and the user already has a subscription to the contact's presence, then the contact's server MUST auto-reply on behalf of the contact by sending a presence stanza of type "subscribed" from the contact's bare JID to the user's bare JID. Likewise, if the contact previously sent a presence stanza of type "subscribed" and the contact's server treated that as indicating "pre-approval" for the user's presence subscription (see Section 3.4), then the contact's server SHOULD also auto-reply on behalf of the contact.

```
CS: <presence from='juliet@example.com'
              id='xk3h1v69'
              to='romeo@example.net'
              type='subscribed'/>
```

3. Otherwise, if there is at least one available resource associated with the contact when the subscription request is received by the contact's server, then the contact's server MUST send that subscription request to all available resources in accordance with Section 8. As a way of acknowledging receipt of the presence subscription request, the contact's server MAY send a

presence stanza of type "unavailable" from the bare JID of the contact to the bare JID of the user (the user's client MUST NOT assume that this acknowledgement provides presence information about the contact, since it comes from the contact's bare JID and is received before the subscription request has been approved).

4. Otherwise, if the contact has no available resources when the subscription request is received by the contact's server, then the contact's server MUST keep a record of the complete presence stanza comprising the subscription request, including any extended content contained therein (see Section 8.4 of [XMPP-CORE]), and then deliver the request when the contact next has an available resource. The contact's server MUST continue to deliver the subscription request whenever the contact creates an available resource, until the contact either approves or denies the request. (The contact's server MUST NOT deliver more than one subscription request from any given user when the contact next has an available resource; e.g., if the user sends multiple subscription requests to the contact while the contact is offline, the contact's server SHOULD store only one of those requests, such as the first request or last request, and MUST deliver only one of the requests when the contact next has an available resource; this helps to prevent "subscription request spam".)

Security Warning: Until and unless the contact approves the subscription request as described under Section 3.1.4, the contact's server MUST NOT add an item for the user to the contact's roster.

Security Warning: The mandate for the contact's server to store the complete stanza of the presence subscription request introduces the possibility of an application resource exhaustion attack (see Section 2.1.2 of [DOS]), for example, by a rogue server or a coordinated group of users (e.g., a botnet) against the contact's server or particular contact. Server implementers are advised to consider the possibility of such attacks and provide tools for counteracting it, such as enabling service administrators to set limits on the number or size of inbound presence subscription requests that the server will store in aggregate or for any given contact.

3.1.4. Client Processing of Inbound Subscription Request

When an interactive client receives a subscription request, it MUST present the request to the natural person controlling the client (i.e., the "contact") for approval, unless the contact has explicitly configured the client to automatically approve or deny some or all

subscription requests as described above. An automated client that is not controlled by a natural person will have its own application-specific rules for approving or denying subscription requests.

A client approves a subscription request by sending a presence stanza of type "subscribed", which is processed as described under Section 3.1.5 for the contact's server and Section 3.1.6 for the user's server.

```
CC: <presence id='h4v1c4kj'
              to='romeo@example.net'
              type='subscribed'/>
```

A client denies a subscription request by sending a presence stanza of type "unsubscribed", which is processed as described under Section 3.2 for both the contact's server and the user's server.

```
CC: <presence id='tb2m1b59'
              to='romeo@example.net'
              type='unsubscribed'/>
```

For tracking purposes, a client SHOULD include an 'id' attribute in a subscription approval or subscription denial; this 'id' attribute MUST NOT mirror the 'id' attribute of the subscription request.

3.1.5. Server Processing of Outbound Subscription Approval

When the contact's client sends the subscription approval, the contact's server MUST stamp the outbound stanza with the bare JID <contact@domainpart> of the contact and locally deliver or remotely route the stanza to the user.

```
CS: <presence from='juliet@example.com'
              id='h4v1c4kj'
              to='romeo@example.net'
              type='subscribed'/>
```

The contact's server then MUST send an updated roster push to all of the contact's interested resources, with the 'subscription' attribute set to a value of "from". (Here we assume that the contact does not already have a subscription to the user; if that were the case, the 'subscription' attribute would be set to a value of "both", as explained under Appendix A.)

```
CS: <iq id='a78b4q6ha463'
        to='juliet@example.com/balcony'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='romeo@example.net'
              subscription='from'/>
      </query>
    </iq>

CS: <iq id='x81g3bdy4n19'
        to='juliet@example.com/chamber'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='romeo@example.net'
              subscription='from'/>
      </query>
    </iq>
```

From the perspective of the contact, there now exists a subscription from the user, which is why the 'subscription' attribute is set to a value of "from". (Here we assume that the contact does not already have a subscription to the user; if that were the case, the 'subscription' attribute would be set to a value of "both", as explained under Appendix A.)

The contact's server MUST then also send current presence to the user from each of the contact's available resources.

```
CS: <presence from='juliet@example.com/balcony'
              id='pw72bc5j'
              to='romeo@example.net'/>

CS: <presence from='juliet@example.com/chamber'
              id='ux31da4q'
              to='romeo@example.net'/>
```

In order to subscribe to the user's presence, the contact would then need to send a subscription request to the user. (XMPP clients will often automatically send the subscription request instead of requiring the contact to initiate the subscription request, since it is assumed that the desired end state is a mutual subscription.) Naturally, when the contact sends a subscription request to the user, the subscription states will be different from those shown in the foregoing examples (see Appendix A) and the roles will be reversed.

3.1.6. Server Processing of Inbound Subscription Approval

When the user's server receives a subscription approval, it MUST first check if the contact is in the user's roster with subscription='none' or subscription='from' and the 'ask' flag set to "subscribe" (i.e., a subscription state of "None + Pending Out", "None + Pending Out+In", or "From + Pending Out"; see Appendix A). If this check is successful, then the user's server MUST:

1. Deliver the inbound subscription approval to all of the user's interested resources (this helps to give the user's client(s) proper context regarding the subscription approval so that they can differentiate between a roster push originated by another of the user's resources and a subscription approval received from the contact). This MUST occur before sending the roster push described in the next step.

```
US: <presence from='juliet@example.com'
              id='h4v1c4kj'
              to='romeo@example.net'
              type='subscribed'/>
```

2. Initiate a roster push to all of the user's interested resources, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "to" (if the subscription state was "None + Pending Out" or "None + Pending Out+In") or "both" (if the subscription state was "From + Pending Out").

```
US: <iq id='b89c5r7ib576'
        to='romeo@example.net/foo'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              subscription='to'/>
      </query>
    </iq>

US: <iq id='b89c5r7ib577'
        to='romeo@example.net/bar'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              subscription='to'/>
      </query>
    </iq>
```

3. The user's server MUST also deliver the available presence stanza received from each of the contact's available resources to each of the user's available resources.

```
[ ... to resource1 ... ]

US: <presence from='juliet@example.com/balcony'
              id='pw72bc5j'
              to='romeo@example.net'/>

[ ... to resource2 ... ]

US: <presence from='juliet@example.com/balcony'
              id='pw72bc5j'
              to='romeo@example.net'/>

[ ... to resource1 ... ]

US: <presence from='juliet@example.com/chamber'
              id='ux31da4q'
              to='romeo@example.net'/>

[ ... to resource2 ... ]

US: <presence from='juliet@example.com/chamber'
              id='ux31da4q'
              to='romeo@example.net'/>
```

Implementation Note: If the user's account has no available resources when the inbound subscription approval notification is received, the user's server MAY keep a record of the notification (ideally the complete presence stanza) and then deliver the notification when the account next has an available resource. This behavior provides more complete signaling to the user regarding the reasons for the roster change that occurred while the user was offline.

Otherwise -- that is, if the user does not exist, if the contact is not in the user's roster, or if the contact is in the user's roster with a subscription state other than those described in the foregoing check -- then the user's server MUST silently ignore the subscription approval notification by not delivering it to the user, not modifying the user's roster, and not generating a roster push to the user's interested resources.

From the perspective of the user, there now exists a subscription to the contact's presence (which is why the 'subscription' attribute is set to a value of "to").

3.2. Canceling a Subscription

3.2.1. Client Generation of Subscription Cancellation

If a contact would like to cancel a subscription that it has previously granted to a user, to cancel a subscription pre-approval (Section 3.4), or to deny a subscription request, it sends a presence stanza of type "unsubscribed".

```
CC: <presence id='ij5b1v7g'
              to='romeo@example.net'
              type='unsubscribed'/>
```

3.2.2. Server Processing of Outbound Subscription Cancellation

Upon receiving the outbound subscription cancellation, the contact's server MUST proceed as follows.

1. If the user's bare JID is not yet in the contact's roster or is in the contact's roster with a state of "None", "None + Pending Out", or "To", the contact's server SHOULD NOT route or deliver the presence stanza of type "unsubscribed" to the user and MUST NOT send presence notifications of type "unavailable" to the user as described below.

2. If the user's bare JID is in the contact's roster with a state of "None", "None + Pending Out", or "To" and the 'approved' flag is set to "true" (thus signaling a subscription pre-approval as described under Section 3.4), the contact's server MUST remove the pre-approval and MUST NOT route or deliver the presence stanza of type "unsubscribed" to the user.

3. Otherwise, as shown in the following examples, the contact's server MUST route or deliver both presence notifications of type "unavailable" and presence stanzas of type "unsubscribed" to the user and MUST send a roster push to the contact.

While the user is still subscribed to the contact's presence (i.e., before the contact's server routes or delivers the presence stanza of type "unsubscribed" to the user), the contact's server MUST send a presence stanza of type "unavailable" from all of the contact's online resources to the user.

```
CS: <presence from='juliet@example.com/balcony'
              id='i8bsg3h3'
              type='unavailable'/>

CS: <presence from='juliet@example.com/chamber'
              id='bvx2c9mk'
              type='unavailable'/>
```

Then the contact's server MUST route or deliver the presence stanza of type "unsubscribed" to the user, making sure to stamp the outbound subscription cancellation with the bare JID <contact@domainpart> of the contact.

```
CS: <presence from='juliet@example.com'
              id='ij5b1v7g'
              to='romeo@example.net'
              type='unsubscribed'/>
```

The contact's server then MUST send a roster push with the updated roster item to all of the contact's interested resources, where the subscription state is now either "none" or "to" (see Appendix A).

```
CS: <iq id='pw3f2v175b34'
        to='juliet@example.com/balcony'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='romeo@example.net'
              subscription='none'/>
      </query>
    </iq>

CS: <iq id='zu2y3f571v35'
        to='juliet@example.com/chamber'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='romeo@example.net'
              subscription='none'/>
      </query>
    </iq>
```

3.2.3. Server Processing of Inbound Subscription Cancellation

When the user's server receives the inbound subscription cancellation, it MUST first check if the contact is in the user's roster with subscription='to' or subscription='both' (see Appendix A). If this check is successful, then the user's server MUST:

1. Deliver the inbound subscription cancellation to all of the user's interested resources (this helps to give the user's client(s) proper context regarding the subscription cancellation so that they can differentiate between a roster push originated by another of the user's resources and a subscription cancellation received from the contact). This MUST occur before sending the roster push described in the next step.

```
US: <presence from='juliet@example.com'
              id='ij5b1v7g'
              to='romeo@example.net'
              type='unsubscribed'/>
```

2. Initiate a roster push to all of the user's interested resources, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "none" (if the subscription state was "To" or "To + Pending In") or "from" (if the subscription state was "Both").

```
US: <iq id='h37h3u1bv400'
        to='romeo@example.net/foo'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>

US: <iq id='h37h3u1bv401'
        to='romeo@example.net/bar'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>
```

The user's server MUST also deliver the inbound presence stanzas of type "unavailable".

Implementation Note: If the user's account has no available resources when the inbound unsubscribed notification is received, the user's server MAY keep a record of the notification (ideally the complete presence stanza) and then deliver the notification when the account next has an available resource. This behavior provides more complete signaling to the user regarding the reasons for the roster change that occurred while the user was offline.

Otherwise -- that is, if the user does not exist, if the contact is not in the user's roster, or if the contact is in the user's roster with a subscription state other than those described in the foregoing check -- then the user's server MUST silently ignore the unsubscribed notification by not delivering it to the user, not modifying the user's roster, and not generating a roster push to the user's interested resources.

3.3. Unsubscribing

3.3.1. Client Generation of Unsubscribe

If a user would like to unsubscribe from a contact's presence, it sends a presence stanza of type "unsubscribe".

```
UC: <presence id='ul4bs71n'
              to='juliet@example.com'
              type='unsubscribe'/>
```

3.3.2. Server Processing of Outbound Unsubscribe

Upon receiving the outbound unsubscribe, the user's server MUST proceed as follows.

1. If the contact is hosted on the same server as the user, then the server MUST adhere to the rules specified under Section 3.3.3 when processing the subscription request.

2. If the contact is hosted on a remote server, subject to local service policies the user's server MUST then route the stanza to that remote domain in accordance with core XMPP stanza processing rules. (This can result in returning an appropriate stanza error to the user, such as <remote-server-timeout/>.)

As mentioned, before locally delivering or remotely routing the unsubscribe, the user's server MUST stamp the stanza with the bare JID <user@domainpart> of the user.

```
US: <presence from='romeo@example.net'
              id='ul4bs71n'
              to='juliet@example.com'
              type='unsubscribe'/>
```

The user's server then MUST send a roster push with the updated roster item to all of the user's interested resources, where the subscription state is now either "none" or "from" (see Appendix A).

```
US: <iq id='h37h3u1bv402'
        to='romeo@example.net/foo'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>

US: <iq to='romeo@example.net/bar'
        type='set'
        id='h37h3u1bv403'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>
```

3.3.3. Server Processing of Inbound Unsubscribe

When the contact's server receives the unsubscribe notification, it MUST first check if the user's bare JID is in the contact's roster with subscription='from' or subscription='both' (i.e., a subscription state of "From", "From + Pending Out", or "Both"; see Appendix A). If this check is successful, then the contact's server MUST:

1. Deliver the inbound unsubscribe to all of the contact's interested resources (this helps to give the contact's client(s) proper context regarding the unsubscribe so that they can differentiate between a roster push originated by another of the contact's resources and an unsubscribe received from the user). This MUST occur before sending the roster push described in the next step.

```
CS: <presence from='romeo@example.net'
              id='ul4bs71n'
              to='juliet@example.com'
              type='unsubscribe'/>
```

2. Initiate a roster push to all of the contact's interested resources, containing an updated roster item for the user with the 'subscription' attribute set to a value of "none" (if the subscription state was "From" or "From + Pending Out") or "to" (if the subscription state was "Both").

```
CS: <iq id='tn2b5893g1s4'
        to='juliet@example.com/balcony'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='romeo@example.net'
              subscription='none'/>
      </query>
    </iq>

CS: <iq id='sp3b56n27hrp'
        to='juliet@example.com/chamber'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item jid='romeo@example.net'
              subscription='none'/>
      </query>
    </iq>
```

3. Generate an outbound presence stanza of type "unavailable" from each of the contact's available resources to the user.

```
CS: <presence from='juliet@example.com/balcony'
              id='o5v91w49'
              to='romeo@example.net'
              type='unavailable'/>

CS: <presence from='juliet@example.com/chamber'
              id='n6b1c37k'
              to='romeo@example.net'
              type='unavailable'/>
```

Implementation Note: If the contact's account has no available resources when the inbound unsubscribe notification is received, the contact's server MAY keep a record of the notification (ideally the complete presence stanza) and then deliver the notification when the account next has an available resource. This behavior provides more complete signaling to the user regarding the reasons for the roster change that occurred while the user was offline.

Otherwise -- that is, if the contact does not exist, if the user is not in the contact's roster, or if the user's bare JID is in the contact's roster with a subscription state other than those described in the foregoing check -- then the contact's server MUST silently ignore the unsubscribe stanza by not delivering it to the contact, not modifying the contact's roster, and not generating a roster push to the contact's interested resources. However, if the contact's server is keeping track of an inbound presence subscription request

from the user to the contact but the user is not yet in the contact's roster (functionally equivalent to a subscription state of "None + Pending In" where the contact never added the user to the contact's roster), then the contact's server MUST simply remove any record of the inbound presence subscription request (it cannot remove the user from the contact's roster because the user was never added to the contact's roster).

> Implementation Note: The user's client MUST NOT depend on receiving the unavailable presence notification from the contact, since it MUST consider its presence subscription to the contact, and its presence information about the contact, to be null and void when it sends the presence stanza of type "unsubscribe" or when it receives the roster push triggered by the unsubscribe request.

3.4. Pre-Approving a Subscription Request

If a user has not received a subscription request from a contact, the user can "pre-approve" such a request so that it will be automatically approved by the user's server.

Support for subscription pre-approvals is OPTIONAL on the part of clients and servers. If a server supports subscription pre-approvals, then it MUST advertise the following stream feature during stream negotiation.

```
<sub xmlns='urn:xmpp:features:pre-approval'/>
```

The subscription pre-approval stream feature is merely informative and therefore is never mandatory-to-negotiate.

3.4.1. Client Generation of Subscription Pre-Approval

If the server to which a client connects has advertised support for subscription pre-approvals, the client MAY generate a subscription pre-approval by sending a presence stanza of type "subscribed" to the contact.

```
UC: <presence id='pg81vx64'
              to='juliet@example.com'
              type='subscribed'/>
```

If the server does not advertise support for subscription pre-approvals, the client MUST NOT attempt to pre-approve subscription requests from potential or actual contacts.

3.4.2. Server Processing of Subscription Pre-Approval

Upon receiving the presence stanza of type "subscribed", the user's server MUST proceed as follows if it supports subscription pre-approvals.

1. If the contact is in the user's roster with a state of "Both", "From", or "From + Pending Out", the user's server MUST silently ignore the stanza.

2. If the contact is in the user's roster with a state of "To + Pending In", "None + Pending In", or "None + Pending Out+In", the user's server MUST handle the stanza as a normal subscription approval (see under Section 3.1.5) by updating the existing roster item to a state of "Both", "From", or "From + Pending Out" (respectively), pushing the modified roster item to all of the user's interested resources, and routing the presence stanza of type "subscribed" to the contact.

3. If the contact is in the user's roster with a state of "To", "None", or "None + Pending Out", the user's server MUST note the subscription pre-approval by setting the 'approved' flag to a value of "true", then push the modified roster item to all of the user's interested resources. However, the user's server MUST NOT route the presence stanza of type "subscribed" to the contact.

4. If the contact is not yet in the user's roster, the user's server MUST create a roster item for the contact with a state of "None" and set the 'approved' flag to a value of "true", then push the roster item to all of the user's interested resources. However, the user's server MUST NOT route the presence stanza of type "subscribed" to the contact.

An example of the roster push follows.

```
US: <iq id='h3bs81vs763f'
        to='romeo@example.net/bar'
        type='set'>
      <query xmlns='jabber:iq:roster'>
        <item approved='true'
              jid='juliet@example.com'
              subscription='none'/>
      </query>
    </iq>
```

When the 'approved' flag is set to "true", the user's server MUST NOT deliver a presence stanza of type "subscribe" from the contact to the user, but instead MUST automatically respond to such a stanza on

behalf of the user by returning a presence stanza of type "subscribed" from the bare JID of the user to the bare JID of the contact.

Implementation Note: It is a matter of implementation or local service policy whether the server maintains a record of the subscription approval after it has received a presence subscription request from the contact. If the server does not maintain such a record, upon receiving the subscription request it will not include the 'approved' attribute in the roster item for the contact (i.e., in subsequent roster pushes and roster results). If the server maintains such a record, it will always include the 'approved' attribute (set to "true") in the roster item for the contact, until and unless the user sends a presence stanza of type "unsubscribed" to the contact (or removes the contact from the roster entirely).

Implementation Note: A client can cancel a pre-approval by sending a presence stanza of type "unsubscribed", as described more fully under Section 3.2. In this case, the user's server would send a roster push to all of the user's interested resources with the 'approved' attribute removed. (Alternatively, the client can simply remove the roster item entirely.)

4. Exchanging Presence Information

4.1. Presence Fundamentals

The concept of presence refers to an entity's availability for communication over a network. At the most basic level, presence is a boolean "on/off" variable that signals whether an entity is available or unavailable for communication (the terms "online" and "offline" are also used). In XMPP, an entity's availability is signaled when its client generates a <presence/> stanza with no 'type' attribute, and an entity's lack of availability is signaled when its client generates a <presence/> stanza whose 'type' attribute has a value of "unavailable".

XMPP presence typically follows a "publish-subscribe" or "observer" pattern, wherein an entity sends presence to its server, and its server then broadcasts that information to all of the entity's contacts who have a subscription to the entity's presence (in the terminology of [IMP-MODEL], an entity that generates presence is a "presentity" and the entities that receive presence are "subscribers"). A client generates presence for broadcast to all subscribed entities by sending a presence stanza to its server with no 'to' address, where the presence stanza has either no 'type' attribute or a 'type' attribute whose value is "unavailable". This

kind of presence is called "broadcast presence". (A client can also send "directed presence", i.e., a presence stanza with a 'to' address; this is less common but is sometimes used to send presence to entities that are not subscribed to the user's presence; see Section 4.6.)

After a client completes the preconditions specified in [XMPP-CORE], it can establish a "presence session" at its server by sending initial presence (Section 4.2), where the presence session is terminated by sending unavailable presence (Section 4.5). For the duration of its presence session, a connected resource (in the terminology of [XMPP-CORE]) is said to be an "available resource".

In XMPP, applications that combine messaging and presence functionality, the default type of communication for which presence signals availability is messaging; however, it is not necessary for XMPP applications to combine messaging and presence functionality, and they can provide standalone presence features without messaging (in addition, XMPP servers do not require information about network availability in order to successfully route message and IQ stanzas).

> Informational Note: In the examples that follow, the user is <juliet@example.com>, she has two available resources ("balcony" and "chamber"), and she has three contacts in her roster with a subscription state of "from" or "both": <romeo@example.net>, <mercutio@example.com>, and <benvolio@example.net>.

4.2. Initial Presence

4.2.1. Client Generation of Initial Presence

After completing the preconditions described in [XMPP-CORE] (REQUIRED) and requesting the roster (RECOMMENDED), a client signals its availability for communication by sending "initial presence" to its server, i.e., a presence stanza with no 'to' address (indicating that it is meant to be broadcast by the server on behalf of the client) and no 'type' attribute (indicating the user's availability).

UC: <presence/>

The initial presence stanza MAY contain the <priority/> element, the <show/> element, and one or more instances of the <status/> element, as well as extended content; details are provided under Section 4.7.

4.2.2. Server Processing of Outbound Initial Presence

Upon receiving initial presence from a client, the user's server MUST send the initial presence stanza from the full JID <user@domainpart/resourcepart> of the user to all contacts that are subscribed to the user's presence; such contacts are those for which a JID is present in the user's roster with the 'subscription' attribute set to a value of "from" or "both".

```
US: <presence from='juliet@example.com/balcony'
              to='romeo@example.net'/>

US: <presence from='juliet@example.com/balcony'
              to='mercutio@example.com'/>

US: <presence from='juliet@example.com/balcony'
              to='benvolio@example.net'/>
```

The user's server MUST also broadcast initial presence from the user's newly available resource to all of the user's available resources, including the resource that generated the presence notification in the first place (i.e., an entity is implicitly subscribed to its own presence).

```
[... to the "balcony" resource ...]

US: <presence from='juliet@example.com/balcony'
              to='juliet@example.com'/>

[... to the "chamber" resource ...]

US: <presence from='juliet@example.com/balcony'
              to='juliet@example.com'/>
```

In the absence of presence information about the user's contacts, the user's server MUST also send presence probes to the user's contacts on behalf of the user as specified under Section 4.3.

4.2.3. Server Processing of Inbound Initial Presence

Upon receiving presence from the user, the contact's server MUST deliver the user's presence stanza to all of the contact's available resources.

```
[ ... to resource1 ... ]

CS: <presence from='juliet@example.com/balcony'
              to='romeo@example.net'/>
```

```
[ ... to resource2 ... ]

CS: <presence from='juliet@example.com/balcony'
              to='romeo@example.net'/>
```

4.2.4. Client Processing of Initial Presence

When the contact's client receives presence from the user, the following behavior is suggested for interactive clients:

1. If the user's bare JID is in the contact's roster, display the presence information in an appropriate roster interface.

2. If the user is not in the contact's roster but the contact and the user are actively exchanging message or IQ stanzas, display the presence information in the user interface for that communication session (see also Section 4.6 and Section 5.1).

3. Otherwise, ignore the presence information and do not display it to the contact.

4.3. Presence Probes

A "presence probe" is a request for a contact's current presence information, sent on behalf of a user by the user's server; syntactically it is a presence stanza whose 'type' attribute has a value of "probe". In the context of presence subscriptions, the value of the 'from' address MUST be the bare JID of the subscribed user and the value of the 'to' address MUST be the bare JID of the contact to which the user is subscribed, since presence subscriptions are based on the bare JID.

```
US: <presence from='juliet@example.com'
              id='ign291v5'
              to='romeo@example.net'
              type='probe'/>
```

> Interoperability Note: RFC 3921 specified that probes are sent from the full JID, not the bare JID (a rule that was changed because subscriptions are based on the bare JID). Some existing implementations send from the full JID instead of the bare JID.

Probes can also be sent by an entity that has received presence outside the context of a presence subscription, typically when the contact has sent directed presence as described under Section 4.6; in this case the value of the 'from' or 'to' address can be a full JID instead of a bare JID. See Section 4.6 for a complete discussion.

Presence probes SHOULD NOT be sent by a client, because in general a client will not need to send them since the task of gathering presence from a user's contacts is managed by the user's server. However, if a user's client generates an outbound presence probe then the user's server SHOULD route the probe (if the contact is at another server) or process the probe (if the contact is at the same server) and MUST NOT use its receipt of the presence probe from a connected client as the sole cause for returning a stanza or stream error to the client.

4.3.1. Server Generation of Outbound Presence Probe

When a server needs to discover the availability of a user's contact, it sends a presence probe from the bare JID <user@domainpart> of the user to the bare JID <contact@domainpart> of the contact.

> Implementation Note: Although presence probes are intended for sending to contacts (i.e., entities to which a user is subscribed), a server MAY send a presence probe to the full JID of an entity from which the user has received presence information during the current session.

The user's server SHOULD send a presence probe whenever the user starts a new presence session by sending initial presence; however, the server MAY choose not to send the probe at that point if it has what it deems to be reliable and up-to-date presence information about the user's contacts (e.g., because the user has another available resource or because the user briefly logged off and on before the new presence session began). In addition, a server MAY periodically send a presence probe to a contact if it has not received presence information or other traffic from the contact in some configurable amount of time; this can help to prevent "ghost" contacts who appear to be online but in fact are not.

```
US: <presence from='juliet@example.com'
              id='ign291v5'
              to='romeo@example.net'
              type='probe'/>

US: <presence from='juliet@example.com'
              id='xv291f38'
              to='mercutio@example.com'
              type='probe'/>
```

Naturally, the user's server does not need to send a presence probe to a contact if the contact's account resides on the same server as the user, since the server possesses the contact's information locally.

4.3.2. Server Processing of Inbound Presence Probe

Upon receiving a presence probe to the contact's bare JID from the user's server on behalf of the user, the contact's server MUST reply as follows:

1. If the contact account does not exist or the user's bare JID is in the contact's roster with a subscription state other than "From", "From + Pending Out", or "Both" (as explained under Appendix A), then the contact's server SHOULD return a presence stanza of type "unsubscribed" in response to the presence probe (this will trigger a protocol flow for canceling the user's subscription to the contact as described under Section 3.2; however, this MUST NOT result in cancellation of a subscription pre-approval as described under Section 3.4). Here the 'from' address MUST be the bare JID of the contact, since specifying a full JID would constitute a presence leak as described in [XMPP-CORE].

```
CS: <presence from='mercutio@example.com'
              id='xv291f38'
              to='juliet@example.com'
              type='unsubscribed'/>
```

However, if a server receives a presence probe from a configured domain of the server itself or another such trusted service, it MAY provide presence information about the user to that entity.

2. Else, if the contact has moved temporarily or permanently to another address, then the server SHOULD return a presence stanza of type "error" with a stanza error condition of <redirect/> (temporary) or <gone/> (permanent) that includes the new address of the contact.

```
CS: <presence from='mercutio@example.com'
              id='xv291f38'
              to='juliet@example.com'
              type='error'>
      <error type='modify'>
        <gone xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
          xmpp:la-mer@example.com
        </gone>
      </error>
    </presence>
```

3. Else, if the contact has no available resources, then the server SHOULD reply to the presence probe by sending to the user a presence stanza of type "unavailable" (although sending

unavailable presence here is preferable because it results in a deterministic answer to the probe, it is not mandatory because it can greatly increase the number of presence notifications generated by the contact's server). Here the 'from' address is the bare JID because there is no available resource associated with the contact. If appropriate in accordance with local security policies this presence notification MAY include the full XML of the last unavailable presence stanza that the server received from the contact (including the 'id' of the original stanza), but if not then the presence notification SHOULD simply indicate that the contact is unavailable without any of the details originally provided. In any case, the presence notification returned to the probing entity SHOULD include information about the time when the last unavailable presence stanza was generated (formatted using the XMPP delayed delivery extension [DELAY]).

```
CS: <presence from='mercutio@example.com'
              id='xv291f38'
              to='juliet@example.com'
              type='unavailable'>
      <delay xmlns='urn:xmpp:delay'
             stamp='2002-09-10T23:41:07Z'/>
    </presence>
```

4. Else, if the contact has at least one available resource, then the server MUST reply to the presence probe by sending to the user the full XML of the last presence stanza with no 'to' attribute received by the server from each of the contact's available resources. Here the 'from' addresses are the full JIDs of each available resource.

```
CS: <presence from='romeo@example.net/foo'
              id='hzf1v27k'
              to='juliet@example.com'/>

CS: <presence from='romeo@example.net/bar'
              id='ps6t1fu3'
              to='juliet@example.com'>
      <show>away</show>
    </presence>
```

Implementation Note: By "full XML" is meant the complete stanza from the opening <presence> tag to the closing </presence> tag, including all elements and attributes whether qualified by the content namespace or extended namespaces; however, in accordance

with [XMPP-CORE], the contact's server will need to transform the content namespace from 'jabber:client' to 'jabber:server' if it sends the complete stanza over a server-to-server stream.

If the contact's server receives a presence probe addressed to a full JID of the contact, the server MUST NOT return presence information about any resource except the resource specified by the 'to' address of the probe. Rules #1 and #2 for a bare JID probe apply equally to the case of a full JID probe. If there is a resource matching the full JID and the probing entity has authorization via a presence subscription to see the contact's presence, then the server MUST return an available presence notification, which SHOULD communicate only the fact that the resource is available (not detailed information such as the <show/>, <status/>, <priority/>, or presence extensions).

```
CS: <presence from='romeo@example.net/bar'
              to='lobby@chat.example.com'/>
```

Implementation Note: See Section 4.6 regarding rules that supplement the foregoing for handling of directed presence.

4.3.2.1. Handling of the 'id' Attribute

The handling of the 'id' attribute in relation to presence probes was unspecified in RFC 3921. Although the pattern of "send a probe and receive a reply" might seem like a request-response protocol similar to the XMPP <iq/> stanza, in fact it is not because the response to a probe might consist of multiple presence stanzas (one for each available resource currently active for the contact). For this reason, if the contact currently has available resources then the contact's server SHOULD preserve the 'id' attribute of the contact's original presence stanza (if any) when sending those presence notifications to the probing entity. By contrast, if the contact currently has no available resources, the probing entity is not authorized (via presence subscription) to see the contact's presence, or an error occurs in relation to the probe, then the contact's server SHOULD mirror the 'id' of the user's presence probe when replying to the probing entity.

The following examples illustrate the difference.

In the first scenario, Juliet sends presence from her "chamber" resource.

```
CC: <presence from='juliet@example.com/chamber' id='pres1'>
      <show>dnd</show>
      <status>busy!</status>
    </presence>
```

She also sends presence from her "balcony" resource.

```
CC: <presence from='juliet@example.com/balcony' id='pres2'>
      <show>away</show>
      <status>stepped away</status>
    </presence>
```

Romeo's server then sends a probe to Juliet.

```
US: <presence from='romeo@example.net' id='probe1' type='probe'/>
```

Juliet's server then sends both of her presence notifications to Romeo, preserving the 'id' attributes included in the stanzas that her client has sent.

```
CS: <presence from='juliet@example.com/chamber' id='pres1'>
      <show>dnd</show>
      <status>busy!</status>
    </presence>

CS: <presence from='juliet@example.com/balcony' id='pres2'>
      <show>away</show>
      <status>stepped away</status>
    </presence>
```

In the second scenario, Juliet is offline when Romeo's server sends a probe.

```
US: <presence from='romeo@example.net'
              id='probe2'
              type='probe'/>
```

Juliet's server replies with an unavailable notification, mirroring the 'id' of Rome's presence probe because there is no 'id' to preserve from an available notification that her client has sent.

```
CS: <presence from='juliet@example.com'
              id='probe2'
              type='unavailable'/>
```

4.4. Subsequent Presence Broadcast

4.4.1. Client Generation of Subsequent Presence Broadcast

After sending initial presence, at any time during its session the user's client can update its availability for broadcast by sending a presence stanza with no 'to' address and no 'type' attribute.

```
UC: <presence>
      <show>away</show>
    </presence>
```

The presence broadcast MAY contain the <priority/> element, the <show/> element, and one or more instances of the <status/> element, as well as extended content; details are provided under Section 4.7.

However, a user SHOULD send a presence update only to broadcast information that is relevant to the user's availability for communication or the communication capabilities of the resource. Information that is not relevant in this way might be of interest to the user's contacts but SHOULD be sent via other means, such as the "publish-subscribe" method described in [XEP-0163].

4.4.2. Server Processing of Subsequent Outbound Presence

Upon receiving a presence stanza expressing updated availability, the user's server MUST broadcast the full XML of that presence stanza to the contacts who are in the user's roster with a subscription type of "from" or "both".

> Interoperability Note: RFC 3921 specified that the user's server would check to make sure that it had not received a presence error from the contact before sending subsequent presence notifications. That rule has been removed because this specification uses presence stanzas of type "unsubscribe" (not "error") to solve subscription synchronization problems, in part because such stanzas change the contact's subscription state in the user's roster to either "none" or "to" (see Section 3.3 and Appendix A), thus obviating the need for the error check.
>
> Interoperability Note: If the subscription type is "both", some existing server implementations send subsequent presence notifications to a contact only if the contact is online according to the user's server (that is, if the user's server never received a positive indication that the contact is online in response to the presence probe it sent to the contact, the user's server does not send subsequent presence notifications from the user to the

contact). This behavior is perceived to save bandwidth, since most presence subscriptions are bidirectional and many contacts will not be online at any given time.

```
US: <presence from='juliet@example.com/balcony'
            to='romeo@example.net'>
      <show>away</show>
    </presence>

US: <presence from='juliet@example.com/balcony'
            to='benvolio@example.net'>
      <show>away</show>
    </presence>

US: <presence from='juliet@example.com/balcony'
            to='mercutio@example.com'>
      <show>away</show>
    </presence>
```

Implementation Note: See Section 4.6 regarding rules that supplement the foregoing for handling of directed presence.

The user's server MUST also send the presence stanza to all of the user's available resources (including the resource that generated the presence notification in the first place).

```
US: <presence from='juliet@example.com/balcony'
            to='juliet@example.com/chamber'>
      <show>away</show>
    </presence>

US: <presence from='juliet@example.com/balcony'
            to='juliet@example.com/balcony'>
      <show>away</show>
    </presence>
```

4.4.3. Server Processing of Subsequent Inbound Presence

Upon receiving presence from the user, the contact's server MUST deliver the user's presence stanza to all of the contact's available resources.

```
[ ... to resource1 ... ]

CS: <presence from='juliet@example.com/balcony'
            to='romeo@example.net'>
      <show>away</show>
    </presence>
```

```
[ ... to resource2 ... ]

CS: <presence from='juliet@example.com/balcony'
            to='romeo@example.net'>
      <show>away</show>
    </presence>
```

4.4.4. Client Processing of Subsequent Presence

From the perspective of the contact's client, there is no significant difference between initial presence broadcast and subsequent presence, so the contact's client follows the rules for processing of inbound presence defined under Section 4.4.3.

4.5. Unavailable Presence

4.5.1. Client Generation of Unavailable Presence

Before ending its presence session with a server, the user's client SHOULD gracefully become unavailable by sending "unavailable presence", i.e., a presence stanza that possesses no 'to' attribute and that possesses a 'type' attribute whose value is "unavailable".

```
UC: <presence type='unavailable'/>
```

Optionally, the unavailable presence stanza MAY contain one or more <status/> elements specifying the reason why the user is no longer available.

```
UC: <presence type='unavailable'>
      <status>going on vacation</status>
    </presence>
```

However, the unavailable presence stanza MUST NOT contain the <priority/> element or the <show/> element, since these elements apply only to available resources.

4.5.2. Server Processing of Outbound Unavailable Presence

The user's server MUST NOT depend on receiving unavailable presence from an available resource, since the resource might become unavailable ungracefully (e.g., the resource's XML stream might be closed with or without a stream error for any of the reasons described in [XMPP-CORE]).

If an available resource becomes unavailable for any reason (either gracefully or ungracefully), the user's server MUST broadcast unavailable presence to all contacts that are in the user's roster with a subscription type of "from" or "both".

Interoperability Note: RFC 3921 specified that the user's server would check to make sure that it had not received a presence error from the contact before sending unavailable presence notifications. That rule has been removed because this specification uses presence stanzas of type "unsubscribe" (not "error") to solve subscription synchronization problems, in part because such stanzas change the contact's subscription state in the user's roster to either "none" or "to" (see Section 3.3 and Appendix A), thus obviating the need for the error check.

Implementation Note: Even if the user's server does not broadcast the user's subsequent presence notifications to contacts who are offline (as described under Section 4.4.2), it MUST broadcast the user's unavailable presence notification; if it did not do so, the last presence received by the contact's server would be the user's initial presence for the presence session, with the result that the contact would consider the user to be online.

Implementation Note: See Section 4.6 regarding rules that supplement the foregoing for handling of directed presence.

If the unavailable notification was gracefully received from the client, then the server MUST broadcast the full XML of the presence stanza.

```
US: <presence from='juliet@example.com/balcony'
            to='romeo@example.net'
            type='unavailable'>
      <status>going on vacation</status>
    </presence>

US: <presence from='juliet@example.com/balcony'
            to='benvolio@example.net'
            type='unavailable'>
      <status>going on vacation</status>
    </presence>

US: <presence from='juliet@example.com/balcony'
            to='mercutio@example.com'
            type='unavailable'>
      <status>going on vacation</status>
    </presence>
```

The user's server MUST also send the unavailable notification to all of the user's available resources (as well as to the resource that generated the unavailable presence in the first place).

```
US: <presence from='juliet@example.com/balcony'
              to='juliet@example.com/chamber'
              type='unavailable'>
      <status>going on vacation</status>
    </presence>
```

If the server detects that the user has gone offline ungracefully, then the server MUST generate the unavailable presence broadcast on the user's behalf.

> Implementation Note: Any presence stanza with no 'type' attribute and no 'to' attribute that the client sends after the server broadcasts or generates an unavailable presence notification MUST be routed or delivered by the user's server to all subscribers (i.e., MUST be treated as equivalent to initial presence for a new presence session).

4.5.3. Server Processing of Inbound Unavailable Presence

Upon receiving an unavailable notification from the user, the contact's server MUST deliver the user's presence stanza to all of the contact's available resources.

```
[ ... to resource1 ... ]

CS: <presence from='juliet@example.com/balcony'
              to='romeo@example.net'
              type='unavailable'>
      <status>going on vacation</status>
    </presence>

[ ... to resource2 ... ]

CS: <presence from='juliet@example.com/balcony'
              to='romeo@example.net'
              type='unavailable'>
      <status>going on vacation</status>
    </presence>
```

> Implementation Note: If the contact's server does not broadcast subsequent presence notifications to users who are offline (as described under Section 4.4.2), it MUST also update its internal representation of which entities are online by noting that the user is unavailable.

4.5.4. Client Processing of Unavailable Presence

From the perspective of the contact's client, there is no significant difference between available presence broadcast and unavailable presence broadcast, so in general the contact's client follows the rules for processing of inbound presence defined under Section 4.4.3.

However, if the contact receives an unavailable notification from the bare JID of the user (rather than the full JID of a particular available resource), the contact's client SHOULD treat the unavailable notification as applying to all resources.

4.6. Directed Presence

This section supplements the rules for client and server processing of presence notifications and presence probes, but only for the special case of directed presence.

4.6.1. General Considerations

In general, a client sends directed presence when it wishes to share availability information with an entity that is not subscribed to its presence, typically on a temporary basis. Common uses of directed presence include casual one-to-one chat sessions as described under Section 5.1 and multi-user chat rooms as described in [XEP-0045].

The temporary relationship established by sharing directed presence with another entity is secondary to the permanent relationship established through a presence subscription. Therefore, the acts of creating, modifying, or canceling a presence subscription MUST take precedence over the rules specified in the following subsections. For example, if a user shares directed presence with a contact but then adds the contact to the user's roster by completing the presence subscription "handshake", the user's server MUST treat the contact just as it would any normal subscriber as described under Section 3, for example, by sending subsequent presence broadcasts to the contact. As another example, if the user then cancels the contact's subscription to the user's presence, the user's server MUST handle the cancellation just as it normally would as described under Section 3.2, which includes sending unavailable presence to the contact even if the user has sent directed presence to the contact.

XMPP servers typically implement directed presence by keeping a list of the entities (bare JIDs or full JIDs) to which a user has sent directed presence during the user's current session for a given resource (full JID), then clearing the list when the user goes offline (e.g., by sending a broadcast presence stanza of type "unavailable"). The server MUST remove from the directed presence

list (or its functional equivalent) any entity to which the user sends directed unavailable presence and SHOULD remove any entity that sends unavailable presence to the user.

4.6.2. Client Generation of Directed Presence

As noted, directed presence is a client-generated presence stanza with a 'to' attribute whose value is the bare JID or full JID of the other entity and with either no 'type' attribute (indicating availability) or a 'type' attribute whose value is "unavailable".

4.6.3. Server Processing of Outbound Directed Presence

When the user's server receives a directed presence stanza, it SHOULD process it according to the following rules.

1. If the user sends directed available or unavailable presence to a contact that is in the user's roster with a subscription type of "from" or "both" after having sent initial presence and before sending unavailable presence broadcast (i.e., during the user's presence session), the user's server MUST locally deliver or remotely route the full XML of that presence stanza but SHOULD NOT otherwise modify the contact's status regarding presence broadcast (i.e., it SHOULD include the contact's JID in any subsequent presence broadcasts initiated by the user).

2. If the user sends directed presence to an entity that is not in the user's roster with a subscription type of "from" or "both" after having sent initial presence and before sending unavailable presence broadcast (i.e., during the user's presence session), the user's server MUST locally deliver or remotely route the full XML of that presence stanza to the entity but MUST NOT modify the contact's status regarding available presence broadcast (i.e., it MUST NOT include the entity's JID in any subsequent broadcasts of available presence initiated by the user); however, if the available resource from which the user sent the directed presence becomes unavailable, the user's server MUST route that unavailable presence to the entity (if the user has not yet sent directed unavailable presence to that entity).

3. If the user sends directed presence without first sending initial presence or after having sent unavailable presence broadcast (i.e., the resource is connected but not available), the user's server MUST treat the entity to which the user sends directed presence as in case #2 above.

4.6.4. Server Processing of Inbound Directed Presence

From the perspective of the contact's server, there is no significant difference between presence broadcast and directed presence, so the contact's server follows the rules for processing of inbound presence defined under Sections 4.3.2, 4.4.3, and 4.5.3.

4.6.5. Client Processing of Inbound Directed Presence

From the perspective of the contact's client, there is no significant difference between presence broadcast and directed presence, so the contact's client follows the rules for processing of inbound presence defined under Section 4.4.3.

4.6.6. Server Processing of Presence Probes

If a user's client has sent directed presence to another entity (e.g., a one-to-one chat partner or a multi-user chat room), after some time the entity or its server might want to know if the client is still online. This scenario is especially common in the case of multi-user chat rooms, in which the user might be a participant for a long period of time. If the user's client goes offline without the chat room being informed (either by the client or the client's server), the user's representation in the room might become a "ghost" that appears to be participating but that in fact is no longer present in the room. To detect such "ghosts", some multi-user chat room implementations send presence probes to users that have joined the room.

In the case of directed presence, the probing entity SHOULD send the probe from the JID that received directed presence (whether a full JID or a bare JID). The probe SHOULD be sent to the user's full JID, not the user's bare JID without a resourcepart, because the temporary "authorization" involved with directed presence is based on the full JID from which the user sent directed presence to the probing entity. When the user's server receives a probe, it MUST first apply any logic associated with presence subscriptions as described under Section 4.3.2. If the probing entity does not have a subscription to the user's presence, then the server MUST check if the user has sent directed presence to the entity during its current session; if so, the server SHOULD answer the probe with only mere presence of type "available" or "unavailable" (i.e., not including child elements) and only for that full JID (i.e., not for any other resources that might be currently associated with the user's bare JID).

4.7. Presence Syntax

4.7.1. Type Attribute

The absence of a 'type' attribute signals that the relevant entity is available for communication (see Section 4.2 and Section 4.4).

A 'type' attribute with a value of "unavailable" signals that the relevant entity is not available for communication (see Section 4.5).

The XMPP presence stanza is also used to negotiate and manage subscriptions to the presence of other entities. These tasks are completed via presence stanzas of type "subscribe", "unsubscribe", "subscribed", and "unsubscribed" as described under Section 3.

If a user and contact are associated with different XMPP servers, those servers also use a special presence stanza of type "probe" in order to determine the availability of the entity on the peer server; details are provided under Section 4.3. Clients SHOULD NOT send presence stanzas of type "probe".

The values of the 'type' attribute can be summarized as follows:

- error -- An error has occurred regarding processing of a previously sent presence stanza; if the presence stanza is of type "error", it MUST include an <error/> child element (refer to [XMPP-CORE]).
- probe -- A request for an entity's current presence; SHOULD be generated only by a server on behalf of a user.
- subscribe -- The sender wishes to subscribe to the recipient's presence.
- subscribed -- The sender has allowed the recipient to receive their presence.
- unavailable -- The sender is no longer available for communication.
- unsubscribe -- The sender is unsubscribing from the receiver's presence.
- unsubscribed -- The subscription request has been denied or a previously granted subscription has been canceled.

If the value of the 'type' attribute is not one of the foregoing values, the recipient or an intermediate router SHOULD return a stanza error of <bad-request/>.

Implementation Note: There is no default value for the 'type' attribute of the <presence/> element.

Implementation Note: There is no value of "available" for the 'type' attribute of the <presence/> element.

4.7.2. Child Elements

In accordance with the default namespace declaration, a presence stanza is qualified by the 'jabber:client' or 'jabber:server' namespace, which defines certain child elements of presence stanzas, in particular the <show/>, <status/>, and <priority/> elements. These child elements are used to provide more detailed information about an entity's availability. Typically these child elements are included only if the presence stanza possesses no 'type' attribute, although exceptions are noted in the text that follows.

4.7.2.1. Show Element

The OPTIONAL <show/> element specifies the particular availability sub-state of an entity or a specific resource thereof. A presence stanza MUST NOT contain more than one <show/> element. There are no attributes defined for the <show/> element. The <show/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]). The XML character data of the <show/> element is not meant for presentation to a human user. The XML character data MUST be one of the following (additional availability states could be defined through extended content elements):

- away -- The entity or resource is temporarily away.
- chat -- The entity or resource is actively interested in chatting.
- dnd -- The entity or resource is busy (dnd = "Do Not Disturb").
- xa -- The entity or resource is away for an extended period (xa = "eXtended Away").

If no <show/> element is provided, the entity is assumed to be online and available.

Any specialized processing of availability states by recipients and intermediate routers is up to the implementation (e.g., incorporation of availability states into stanza routing and delivery logic).

4.7.2.2. Status Element

The OPTIONAL <status/> element contains human-readable XML character data specifying a natural-language description of an entity's availability. It is normally used in conjunction with the show element to provide a detailed description of an availability state (e.g., "In a meeting") when the presence stanza has no 'type' attribute.

```
<presence from='romeo@example.net/orchard'
          xml:lang='en'>
  <show>dnd</show>
  <status>Wooing Juliet</status>
</presence>
```

There are no attributes defined for the <status/> element, with the exception of the 'xml:lang' attribute inherited from [XML]. The <status/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]). Multiple instances of the <status/> element MAY be included, but only if each instance possesses an 'xml:lang' attribute with a distinct language value (either explicitly or by inheritance from the 'xml:lang' value of an element farther up in the XML hierarchy, which from the sender's perspective can include the XML stream header as described in [XMPP-CORE]).

```
<presence from='romeo@example.net/orchard'
          id='jx62vs97'
          xml:lang='en'>
  <show>dnd</show>
  <status>Wooing Juliet</status>
  <status xml:lang='cs'>Dvo&#x0159;&#x00ED;m se Julii</status>
</presence>
```

A presence stanza of type "unavailable" MAY also include a <status/> element to provide detailed information about why the entity is going offline.

```
<presence from='romeo@example.net/orchard'
          id='oy6sb241'
          type='unavailable'
          xml:lang='en'>
  <status>Busy IRL</status>
</presence>
```

The <status/> child MAY also be sent in a subscription-related presence stanza (i.e., type "subscribe", "subscribed", "unsubscribe", or "unsubscribed") to provide a description of the action. An interactive client MAY present this <status/> information to a human user (see Section 11).

```
<presence from='romeo@example.net'
          id='uc51xs63'
          to='nurse@example.com'
          type='subscribe'>
  <status>Hi, Juliet told me to add you to my buddy list.</status>
</presence>
```

4.7.2.3. Priority Element

The OPTIONAL <priority/> element contains non-human-readable XML character data that specifies the priority level of the resource. The value MUST be an integer between -128 and +127. A presence stanza MUST NOT contain more than one <priority/> element. There are no attributes defined for the <priority/> element. The <priority/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

```
<presence xml:lang='en'>
  <show>dnd</show>
  <status>Wooing Juliet</status>
  <status xml:lang='cs'>Dvo&#x0159;&#x00ED;m se Julii</status>
  <priority>1</priority>
</presence>
```

If no priority is provided, the processing server or client MUST consider the priority to be zero ("0").

The client's server MAY override the priority value provided by the client (e.g., in order to impose a message handling rule of delivering a message intended for the account's bare JID to all of the account's available resources). If the server does so, it MUST communicate the modified priority value when it echoes the client's presence back to itself and sends the presence notification to the user's contacts (because this modified priority value is typically the default value of zero, communicating the modified priority value can be done by not including the <priority/> child element).

For information regarding the semantics of priority values in stanza processing within instant messaging and presence applications, refer to Section 8.

4.7.3. Extended Content

As described in [XMPP-CORE], an XML stanza MAY contain any child element that is qualified by a namespace other than the default namespace; this applies to the presence stanza as well.

(In the following example, the presence stanza includes entity capabilities information as defined in [XEP-0115].)

```
<presence from='romeo@example.net'>
  <c xmlns='http://jabber.org/protocol/caps'
     hash='sha-1'
     node='http://psi-im.org'
     ver='q07IKJEyjvHSyhy//CH0CxmKi8w='/>
</presence>
```

Any extended content included in a presence stanza SHOULD represent aspects of an entity's availability for communication or provide information about communication-related capabilities.

5. Exchanging Messages

Once a client has authenticated with a server and bound a resource to an XML stream as described in [XMPP-CORE], an XMPP server will route XML stanzas to and from that client. One kind of stanza that can be exchanged is <message/> (if, that is, messaging functionality is enabled on the server). Exchanging messages is a basic use of XMPP and occurs when a user generates a message stanza that is addressed to another entity. As defined under Section 8, the sender's server is responsible for delivering the message to the intended recipient (if the recipient is on the same local server) or for routing the message to the recipient's server (if the recipient is on a remote server). Thus a message stanza is used to "push" information to another entity.

5.1. One-to-One Chat Sessions

In practice, instant messaging activity between human users tends to occur in the form of a conversational burst that we call a "chat session": the exchange of multiple messages between two parties in relatively rapid succession within a relatively brief period of time.

When a human user intends to engage in such a chat session with a contact (rather than sending a single message to which no reply is expected), the message type generated by the user's client SHOULD be "chat" and the contact's client SHOULD preserve that message type in subsequent replies. The user's client also SHOULD include a

<thread/> element with its initial message, which the contact's client SHOULD also preserve during the life of the chat session (see Section 5.2.5).

The user's client SHOULD address the initial message in a chat session to the bare JID <contact@domainpart> of the contact (rather than attempting to guess an appropriate full JID <contact@domainpart/resourcepart> based on the <show/>, <status/>, or <priority/> value of any presence notifications it might have received from the contact). Until and unless the user's client receives a reply from the contact, it SHOULD send any further messages to the contact's bare JID. The contact's client SHOULD address its replies to the user's full JID <user@domainpart/resourcepart> as provided in the 'from' address of the initial message. Once the user's client receives a reply from the contact's full JID, it SHOULD address its subsequent messages to the contact's full JID as provided in the 'from' address of the contact's replies, thus "locking in" on that full JID. A client SHOULD "unlock" after having received a <message/> or <presence/> stanza from any other resource controlled by the peer (or a presence stanza from the locked resource); as a result, it SHOULD address its next message(s) in the chat session to the bare JID of the peer (thus "unlocking" the previous "lock") until it receives a message from one of the peer's full JIDs.

When two parties engage in a chat session but do not share presence with each other based on a presence subscription, they SHOULD send directed presence to each other so that either party can easily discover if the peer goes offline during the course of the chat session. However, a client MUST provide a way for a user to disable such presence sharing globally or to enable it only with particular entities. Furthermore, a party SHOULD send directed unavailable presence to the peer when it has reason to believe that the chat session is over (e.g., if, after some reasonable amount of time, no subsequent messages have been exchanged between the parties).

An example of a chat session is provided under Section 7.

5.2. Message Syntax

The following sections describe the syntax of the <message/> stanza.

5.2.1. To Attribute

An instant messaging client specifies an intended recipient for a message by providing the JID of the intended recipient in the 'to' attribute of the <message/> stanza.

If the message is being sent outside the context of any existing chat session or received message, the value of the 'to' address SHOULD be of the form <localpart@domainpart> rather than of the form <localpart@domainpart/resourcepart> (see Section 5.1).

```
<message
    from='juliet@example.com/balcony'
    id='ktx72v49'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <body>Art thou not Romeo, and a Montague?</body>
</message>
```

If the message is being sent in reply to a message previously received from an address of the form <localpart@domainpart/resourcepart> (e.g., within the context of a one-to-one chat session as described under Section 5.1), the value of the 'to' address SHOULD be of the form <localpart@domainpart/resourcepart> rather than of the form <localpart@domainpart> unless the sender has knowledge (e.g., via presence) that the intended recipient's resource is no longer available.

```
<message
    from='romeo@example.net/orchard'
    id='sl3nx51f'
    to='juliet@example.com/balcony'
    type='chat'
    xml:lang='en'>
  <body>Neither, fair saint, if either thee dislike.</body>
</message>
```

5.2.2. Type Attribute

Common uses of the message stanza in instant messaging applications include: single messages; messages sent in the context of a one-to-one chat session; messages sent in the context of a multi-user chat room; alerts, notifications, or other information to which no reply is expected; and errors. These uses are differentiated via the 'type' attribute. Inclusion of the 'type' attribute is RECOMMENDED. If included, the 'type' attribute MUST have one of the following values:

- chat -- The message is sent in the context of a one-to-one chat session. Typically an interactive client will present a message of type "chat" in an interface that enables one-to-one chat between the two parties, including an appropriate conversation history. Detailed recommendations regarding one-to-one chat sessions are provided under Section 5.1.

- error -- The message is generated by an entity that experiences an error when processing a message received from another entity (for details regarding stanza error syntax, refer to [XMPP-CORE]). A client that receives a message of type "error" SHOULD present an appropriate interface informing the original sender regarding the nature of the error.

- groupchat -- The message is sent in the context of a multi-user chat environment (similar to that of [IRC]). Typically a receiving client will present a message of type "groupchat" in an interface that enables many-to-many chat between the parties, including a roster of parties in the chatroom and an appropriate conversation history. For detailed information about XMPP-based groupchat, refer to [XEP-0045].

- headline -- The message provides an alert, a notification, or other transient information to which no reply is expected (e.g., news headlines, sports updates, near-real-time market data, or syndicated content). Because no reply to the message is expected, typically a receiving client will present a message of type "headline" in an interface that appropriately differentiates the message from standalone messages, chat messages, and groupchat messages (e.g., by not providing the recipient with the ability to reply). If the 'to' address is the bare JID, the receiving server SHOULD deliver the message to all of the recipient's available resources with non-negative presence priority and MUST deliver the message to at least one of those resources; if the 'to' address is a full JID and there is a matching resource, the server MUST deliver the message to that resource; otherwise the server MUST either silently ignore the message or return an error (see Section 8).

- normal -- The message is a standalone message that is sent outside the context of a one-to-one conversation or groupchat, and to which it is expected that the recipient will reply. Typically a receiving client will present a message of type "normal" in an interface that enables the recipient to reply, but without a conversation history. The default value of the 'type' attribute is "normal".

An IM application SHOULD support all of the foregoing message types. If an application receives a message with no 'type' attribute or the application does not understand the value of the 'type' attribute provided, it MUST consider the message to be of type "normal" (i.e., "normal" is the default).

Guidelines for server handling of different message types is provided under Section 8.

Although the 'type' attribute is OPTIONAL, it is considered polite to mirror the type in any replies to a message; furthermore, some specialized applications (e.g., a multi-user chat service) MAY at their discretion enforce the use of a particular message type (e.g., type='groupchat').

5.2.3. Body Element

The <body/> element contains human-readable XML character data that specifies the textual contents of the message; this child element is normally included but is OPTIONAL.

```
<message
    from='juliet@example.com/balcony'
    id='b4vs9km4'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <body>Wherefore art thou, Romeo?</body>
</message>
```

There are no attributes defined for the <body/> element, with the exception of the 'xml:lang' attribute. Multiple instances of the <body/> element MAY be included in a message stanza for the purpose of providing alternate versions of the same body, but only if each instance possesses an 'xml:lang' attribute with a distinct language value (either explicitly or by inheritance from the 'xml:lang' value of an element farther up in the XML hierarchy, which from the sender's perspective can include the XML stream header as described in [XMPP-CORE]).

```
<message
    from='juliet@example.com/balcony'
    id='z94nb37h'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <body>Wherefore art thou, Romeo?</body>
  <body xml:lang='cs'>
     Pro&#x010D;e&#x017D; jsi ty, Romeo?
   </body>
</message>
```

The <body/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

5.2.4. Subject Element

The <subject/> element contains human-readable XML character data that specifies the topic of the message.

```
<message
    from='juliet@example.com/balcony'
    id='c8xg3nf8'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <subject>I implore you!</subject>
  <body>Wherefore art thou, Romeo?</body>
</message>
```

There are no attributes defined for the <subject/> element, with the exception of the 'xml:lang' attribute inherited from [XML]. Multiple instances of the <subject/> element MAY be included for the purpose of providing alternate versions of the same subject, but only if each instance possesses an 'xml:lang' attribute with a distinct language value (either explicitly or by inheritance from the 'xml:lang' value of an element farther up in the XML hierarchy, which from the sender's perspective can include the XML stream header as described in [XMPP-CORE]).

```
<message
    from='juliet@example.com/balcony'
    id='jk3v47gw'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <subject>I implore you!</subject>
  <subject xml:lang='cs'>
    &#x00DA;p&#x011B;nliv&#x011B; pros&#x00ED;m!
  </subject>
  <body>Wherefore art thou, Romeo?</body>
  <body xml:lang='cs'>
     Pro&#x010D;e&#x017E; jsi ty, Romeo?
   </body>
</message>
```

The <subject/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

5.2.5. Thread Element

The primary use of the XMPP <thread/> element is to uniquely identify a conversation thread or "chat session" between two entities instantiated by <message/> stanzas of type 'chat'. However, the XMPP <thread/> element MAY also be used to uniquely identify an analogous thread between two entities instantiated by <message/> stanzas of type 'headline' or 'normal', or among multiple entities in the context of a multi-user chat room instantiated by <message/> stanzas of type 'groupchat'. It MAY also be used for <message/> stanzas not related to a human conversation, such as a game session or an interaction between plugins. The <thread/> element is not used to identify individual messages, only conversations or messaging sessions.

The inclusion of the <thread/> element is OPTIONAL. Because the <thread/> element identifies the particular conversation thread to which a message belongs, a message stanza MUST NOT contain more than one <thread/> element.

The <thread/> element MAY possess a 'parent' attribute that identifies another thread of which the current thread is an offshoot or child. The 'parent' attribute MUST conform to the syntax of the <thread/> element itself and its value MUST be different from the XML character data of the <thread/> element on which the 'parent' attribute is included.

Implementation Note: The ability to specify both a parent thread and a child thread introduces the possibility of conflicts between thread identifiers for overlapping threads. For example, one <thread/> element might contain XML character data of "foo" and a 'parent' attribute whose value is "bar", a second <thread/> element might contain XML character data of "bar" and a 'parent' attribute whose value is "baz", and a third <thread/> element might contain XML character data of "baz" and a 'parent' attribute whose value is once again "foo". It is up to the implementation how it will treat conflicts between such overlapping thread identifiers (e.g., whether it will "chain together" thread identifiers by showing "foo" as both a parent and grandchild of "baz" in a multi-level user interface, or whether it will show only one level of dependency at a time).

The value of the <thread/> element is not human-readable and MUST be treated as opaque by entities; no semantic meaning can be derived from it, and only exact comparisons can be made against it. The value of the <thread/> element MUST uniquely identify the conversation thread either between the conversation partners or more generally (one way to ensure uniqueness is by generating a universally unique identifier (UUID) as described in [UUID]).

Security Warning: An application that generates a ThreadID MUST ensure that it does not reveal identifying information about the entity (e.g., the MAC address of the device on which the XMPP application is running).

The <thread/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

```
<message
    from='juliet@example.com/balcony'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <subject>I implore you!</subject>
  <subject xml:lang='cs'>
    &#x00DA;p&#x011B;nliv&#x011B; pros&#x00ED;m!
  </subject>
  <body>Wherefore art thou, Romeo?</body>
  <body xml:lang='cs'>
     Pro&#x010D;e&#x017E; jsi ty, Romeo?
  </body>
  <thread parent='e0ffe42b28561960c6b12b944a092794b9683a38'>
    0e3141cd80894871a68e6fe6b1ec56fa
  </thread>
</message>
```

For detailed recommendations regarding use of the <thread/> element, refer to [XEP-0201].

5.3. Extended Content

As described in [XMPP-CORE], an XML stanza MAY contain any child element that is qualified by a namespace other than the default namespace; this applies to the message stanza as well. Guidelines for handling extended content on the part of both routing servers and end recipients are provided in Section 8.4 of [XMPP-CORE].

(In the following example, the message stanza includes an XHTML-formatted version of the message as defined in [XEP-0071]).

```
<message
    from='juliet@example.com/balcony'
    to='romeo@example.net'
    type='chat'
    xml:lang='en'>
  <body>Wherefore art thou, Romeo?</body>
  <html xmlns='http://jabber.org/protocol/xhtml-im'>
    <body xmlns='http://www.w3.org/1999/xhtml'>
      <p>Wherefore <span style='font-style: italic'>art</span>
      thou, <span style='color:red'>Romeo</span>?</p>
    </body>
  </html>
</message>
```

6. Exchanging IQ Stanzas

As described in [XMPP-CORE], IQ stanzas provide a structured request-response mechanism. The basic semantics of that mechanism (e.g., that the 'id' attribute is mandatory) are defined in [XMPP-CORE], whereas the specific semantics needed to complete particular use cases are defined in all instances by the extended namespace that qualifies the direct child element of an IQ stanza of type "get" or "set". The 'jabber:client' and 'jabber:server' namespaces do not define any children of IQ stanzas other than the <error/> element common to all stanza types. This document defines one such extended namespace, for Managing the Roster (Section 2). However, an IQ stanza MAY contain structured information qualified by any extended namespace.

7. A Sample Session

The examples in this section illustrate a possible instant messaging and presence session. The user is <romeo@example.net>, he has an available resource whose resourcepart is "orchard", and he has the following individuals in his roster:

- <juliet@example.com> (subscription="both" and she has two available resources, "chamber" and "balcony")
- <benvolio@example.net> (subscription="to")
- <mercutio@example.org> (subscription="from")

First, the user completes the preconditions (stream establishment, TLS and SASL negotiation, and resource binding) described in [XMPP-CORE]; those protocol flows are not reproduced here.

Next, the user requests his roster.

Example 1: User requests current roster from server

```
UC: <iq from='romeo@example.net/orchard'
        id='hf61v3n7'
        type='get'>
      <query xmlns='jabber:iq:roster'/>
    </iq>
```

Example 2: User receives roster from server

```
US: <iq id='hf61v3n7'
        to='romeo@example.net/orchard'
        type='result'>
      <query xmlns='jabber:iq:roster'>
        <item jid='juliet@example.com'
              name='Juliet'
              subscription='both'>
          <group>Friends</group>
        </item>
        <item jid='benvolio@example.org'
              name='Benvolio'
              subscription='to'/>
        <item jid='mercutio@example.org'
              name='Mercutio'
              subscription='from'/>
      </query>
    </iq>
```

Now the user begins a presence session.

Example 3: User sends initial presence

```
UC: <presence/>
```

Example 4: User's server sends presence probes to contacts with subscription="to" and subscription="both" on behalf of the user

```
US: <presence
        from='romeo@example.net'
        to='juliet@example.com'
        type='probe'/>

US: <presence
        from='romeo@example.net'
        to='benvolio@example.org'
        type='probe'/>
```

Example 5: User's server sends initial presence to contacts with subscription="from" and subscription="both" on behalf of the user's available resource, as well as to user

```
US: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'/>

US: <presence
        from='romeo@example.net/orchard'
        to='mercutio@example.org'/>

US: <presence
        from='romeo@example.net/orchard'
        to='romeo@example.net'/>
```

Example 6: Contacts' servers reply to presence probe on behalf of all available resources

```
CS: <presence
        from='juliet@example.com/balcony'
        to='romeo@example.net'
        xml:lang='en'>
      <show>away</show>
      <status>be right back</status>
      <priority>0</priority>
    </presence>
```

```
CS: <presence
        from='juliet@example.com/chamber'
        to='romeo@example.net'>
      <priority>1</priority>
    </presence>

CS: <presence
        from='benvolio@example.org/pda'
        to='romeo@example.net'
        xml:lang='en'>
      <show>dnd</show>
      <status>gallivanting</status>
    </presence>
```

Example 7: Contacts' servers deliver user's initial presence to all available resources

```
CS: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'/>

CS: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'/>

CS: <presence
        from='romeo@example.net/orchard'
        to='mercutio@example.org'/>
```

Example 8: User sends directed presence to another user not in his roster

```
UC: <presence
        from='romeo@example.net/orchard'
        to='nurse@example.com'
        xml:lang='en'>
      <show>dnd</show>
      <status>courting Juliet</status>
      <priority>0</priority>
    </presence>
```

Now the user engages in a chat session with one of his contacts.

Example 9: A threaded conversation

```
CC: <message
        from='juliet@example.com/balcony'
        to='romeo@example.net'
        type='chat'
        xml:lang='en'>
      <body>My ears have not yet drunk a hundred words</body>
      <thread>e0ffe42b28561960c6b12b944a092794b9683a38</thread>
    </message>

CC: <message
        from='juliet@example.com/balcony'
        to='romeo@example.net'
        type='chat'
        xml:lang='en'>
      <body>Of that tongue's utterance, yet I know the sound:</body>
      <thread>e0ffe42b28561960c6b12b944a092794b9683a38</thread>
    </message>

CC: <message
        from='juliet@example.com/balcony'
        to='romeo@example.net'
        type='chat'
        xml:lang='en'>
      <body>Art thou not Romeo, and a Montague?</body>
      <thread>e0ffe42b28561960c6b12b944a092794b9683a38</thread>
    </message>

UC: <message
        from='romeo@example.net/orchard'
        to='juliet@example.com/balcony'
        type='chat'
        xml:lang='en'>
      <body>Neither, fair saint, if either thee dislike.</body>
      <thread>e0ffe42b28561960c6b12b944a092794b9683a38</thread>
    </message>

CC: <message
        from='juliet@example.com/balcony'
        to='romeo@example.net/orchard'
        type='chat'
        xml:lang='en'>
      <body>How cam'st thou hither, tell me, and wherefore?</body>
      <thread>e0ffe42b28561960c6b12b944a092794b9683a38</thread>
    </message>
```

And so on.

The user can also send subsequent presence broadcast.

Example 10: User sends updated available presence for broadcast

```
UC: <presence xml:lang='en'>
      <show>away</show>
      <status>I shall return!</status>
      <priority>1</priority>
    </presence>
```

Example 11: User's server broadcasts updated presence to the contacts who have a subscription of type "both" or "from" (but not to the entity to which the user sent directed presence)

```
US: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'
        xml:lang='en'>
      <show>away</show>
      <status>I shall return!</status>
      <priority>1</priority>
    </presence>

US: <presence
        from='romeo@example.net/orchard'
        to='mercutio@example.org'
        xml:lang='en'>
      <show>away</show>
      <status>I shall return!</status>
      <priority>1</priority>
    </presence>
```

Example 12: Contacts' servers deliver updated presence

```
CS: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'
        xml:lang='en'>
      <show>away</show>
      <status>I shall return!</status>
      <priority>1</priority>
    </presence>
```

```
CS: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'
        xml:lang='en'>
      <show>away</show>
      <status>I shall return!</status>
      <priority>1</priority>
    </presence>

CS: <presence
        from='romeo@example.net/orchard'
        to='mercutio@example.org'
        xml:lang='en'>
      <show>away</show>
      <status>I shall return!</status>
      <priority>1</priority>
    </presence>
```

Example 13: One of the contact's resources broadcasts unavailable notification

```
CC: <presence from='juliet@example.com/chamber' type='unavailable'/>
```

Example 14: Contact's server sends unavailable notification to user

```
CS: <presence
        from='juliet@example.com/chamber'
        to='romeo@example.net'
        type='unavailable'/>
```

Now the user ends his presence session.

Example 15: User sends unavailable notification

```
UC: <presence type='unavailable' xml:lang='en'>
      <status>gone home</status>
    </presence>
```

Example 16: User's server broadcasts unavailable notification to contacts as well as to the entity to whom the user sent directed presence

```
US: <presence
        from='romeo@example.net/orchard'
        to='juliet@example.com'
        type='unavailable'
        xml:lang='en'>
      <status>gone home</status>
    </presence>

US: <presence
        from='romeo@example.net/orchard'
        to='mercutio@example.org'
        type='unavailable'
        xml:lang='en'>
      <status>gone home</status>
    </presence>

US: <presence
        from='romeo@example.net/orchard'
        to='nurse@example.com'
        type='unavailable'
        xml:lang='en'>
      <status>gone home</status>
    </presence>
```

Finally the user closes his stream and the server responds in kind.

Example 17: User closes stream

```
UC: </stream:stream>
```

Example 18: User's server closes stream

```
US: </stream:stream>
```

THE END

8. Server Rules for Processing XML Stanzas

Basic server rules for processing XML stanzas are defined in [XMPP-CORE], and the reader is referred to that specification for underlying rules and security implications. This section defines supplementary rules for XMPP instant messaging and presence servers.

Some delivery rules defined in this section specify the use of "offline storage", i.e., the server's act of storing a message stanza on behalf of the user and then delivering it when the user next becomes available. For recommendations regarding offline message storage, refer to [XEP-0160].

8.1. General Considerations

[XMPP-CORE] discusses general considerations for stanza delivery, in particular the tradeoffs between (i) providing an acceptable level of service regarding stanza delivery and (ii) preventing directory harvesting attacks and presence leaks. However, the concept of a directory harvesting attack does not apply if a contact is known to and trusted by a user (because the contact is in the user's roster as described under Section 2). Similarly, the concept of a presence leak does not apply if a contact is authorized to know a user's presence (by means of a presence subscription as described under Section 3) or if the user has voluntarily sent presence to an entity (by means of directed presence as described under Section 4.6). Therefore, in cases where the following sections guard against directory harvesting attacks and presence leaks by providing an alternative of (a) silently ignoring a stanza or (b) returning an error, a server SHOULD return an error if the originating entity is in the user's roster (when the error would reveal whether the user's account exists) or is authorized to receive presence from the user or has received directed presence from the user (when the error would reveal the presence of a user's resource).

> Security Warning: All of the stanza processing rules described below are defined with the understanding that they will be applied subject to enforcement of relevant privacy and security policies, such as those deployed by means of [XEP-0016] or [XEP-0191]. The conformance language (MUST, SHOULD, etc.) in the following sections is not meant to override any such local service policies.

8.2. No 'to' Address

If the stanza possesses no 'to' attribute, the rules defined in [XMPP-CORE] apply.

8.3. Remote Domain

If the domainpart of the address contained in the 'to' attribute of an outbound stanza does not match a configured domain of the server itself, then the rules provided in Section 10.4 of [XMPP-CORE] apply.

> Interoperability Note: RFC 3921 specified how to use the _im._xmpp and _pres._xmpp SRV records [IMP-SRV] as a fallback method for discovering whether a remote instant messaging and presence service communicates via XMPP. Because those SRV records have not been widely deployed, this document no longer specifies their use, and new implementations are not encouraged.

8.4. Local Domain

If the domainpart of the JID contained in the 'to' attribute matches one of the configured domains of the server, the domain is serviced by the server itself (not by a specialized local service), and the JID is of the form <domainpart> or <domainpart/resourcepart>, the rules defined in [XMPP-CORE] apply.

8.5. Local User

If the 'to' address specifies a bare JID <localpart@domainpart> or full JID <localpart@domainpart/resourcepart> where the domainpart of the JID matches a configured domain that is serviced by the server itself, the server MUST proceed as follows.

8.5.1. No Such User

If the user account identified by the 'to' attribute does not exist, how the stanza is processed depends on the stanza type.

- For an IQ stanza, the server MUST return a <service-unavailable/> stanza error to the sender.
- For a message stanza, the server MUST either (a) silently ignore the message or (b) return a <service-unavailable/> stanza error to the sender.
- For a presence stanza with no 'type' attribute or a 'type' attribute of "unavailable", the server MUST silently ignore the stanza.
- For a presence stanza of type "subscribe", "subscribed", "unsubscribe", or "unsubscribed", the server MUST silently ignore the stanza.
- For a presence stanza of type "probe", the server MUST either (a) silently ignore the stanza or (b) return a presence stanza of type "unsubscribed".

8.5.2. localpart@domainpart

If the JID contained in the 'to' attribute is of the form <localpart@domainpart>, then the server MUST adhere to the following rules.

8.5.2.1. Available or Connected Resources

If there is at least one available resource or connected resource, how the stanza is processed depends on the stanza type.

8.5.2.1.1. Message

For a message stanza of type "normal":

- If all of the available resources have a negative presence priority then the server SHOULD either (a) store the message offline for later delivery or (b) return a stanza error to the sender, which SHOULD be <service-unavailable/>.

- If there is one available resource with a non-negative presence priority then the server MUST deliver the message to that resource.

- If there is more than one resource with a non-negative presence priority then the server MUST either (a) deliver the message to the "most available" resource or resources (according to the server's implementation-specific algorithm, e.g., treating the resource or resources with the highest presence priority as "most available") or (b) deliver the message to all of the non-negative resources.

For a message stanza of type "chat":

- If the only available resource has a negative presence priority then the server SHOULD either (a) store the message offline for later delivery or (b) return a stanza error to the sender, which SHOULD be <service-unavailable/>.

- If the only available resource has a non-negative presence priority then the server MUST deliver the message to that resource.

- If there is more than one resource with a non-negative presence priority then the server MUST either (a) deliver the message to the "most available" resource or resources (according to the server's implementation-specific algorithm, e.g., treating the resource or resources with the highest presence priority as "most available") or (b) deliver the message to all of the non-negative resources that have opted in to receive chat messages.

For a message stanza of type "groupchat", the server MUST NOT deliver the stanza to any of the available resources but instead MUST return a stanza error to the sender, which SHOULD be <service-unavailable/>.

For a message stanza of type "headline":

- If the only available resource has a negative presence priority then the server MUST silently ignore the stanza.
- If the only available resource has a non-negative presence priority then the server MUST deliver the message to that resource.
- If there is more than one resource with a non-negative presence priority then the server MUST deliver the message to all of the non-negative resources.

For a message stanza of type "error", the server MUST silently ignore the message.

However, for any message type the server MUST NOT deliver the stanza to any available resource with a negative priority; if the only available resource has a negative priority, the server SHOULD handle the message as if there were no available resources or connected resources as described under Section 8.5.2.2.

In all cases, the server MUST NOT rewrite the 'to' attribute (i.e., it MUST leave it as <localpart@domainpart> rather than change it to <localpart@domainpart/resourcepart>).

8.5.2.1.2. Presence

For a presence stanza with no type or of type "unavailable", the server MUST deliver it to all available resources.

For a presence stanza of type "subscribe", "subscribed", "unsubscribe", or "unsubscribed", the server MUST adhere to the rules defined under Section 3 and summarized under Appendix A.

For a presence stanza of type "probe", the server MUST handle it directly as described under Section 4.3.

In all cases, the server MUST NOT rewrite the 'to' attribute (i.e., it MUST leave it as <localpart@domainpart> rather than change it to <localpart@domainpart/resourcepart>).

8.5.2.1.3. IQ

For an IQ stanza, the server itself MUST reply on behalf of the user with either an IQ result or an IQ error, and MUST NOT deliver the IQ stanza to any of the user's available resources. Specifically, if the semantics of the qualifying namespace define a reply that the

server can provide on behalf of the user, then the server MUST reply to the stanza on behalf of the user by returning either an IQ stanza of type "result" or an IQ stanza of type "error" that is appropriate to the original payload; if not, then the server MUST reply with a <service-unavailable/> stanza error.

8.5.2.2. No Available or Connected Resources

If there are no available resources or connected resources associated with the user, how the stanza is processed depends on the stanza type.

8.5.2.2.1. Message

For a message stanza of type "normal" or "chat", the server SHOULD either (a) add the message to offline storage or (b) return a stanza error to the sender, which SHOULD be <service-unavailable/>.

For a message stanza of type "groupchat", the server MUST return an error to the sender, which SHOULD be <service-unavailable/>.

For a message stanza of type "headline" or "error", the server MUST silently ignore the message.

8.5.2.2.2. Presence

For a presence stanza with no type or of type "unavailable", the server SHOULD silently ignore the stanza by not storing it for later delivery and not replying to it on behalf of the user.

For a presence stanza of type "subscribe", "subscribed", "unsubscribe", or "unsubscribed", the server MUST adhere to the rules defined under Section 3 and summarized under Appendix A.

For a presence stanza of type "probe", the server MUST handle it directly as described under Section 4.3.

8.5.2.2.3. IQ

For an IQ stanza, the server itself MUST reply on behalf of the user with either an IQ result or an IQ error. Specifically, if the semantics of the qualifying namespace define a reply that the server can provide on behalf of the user, then the server MUST reply to the stanza on behalf of the user by returning either an IQ stanza of type "result" or an IQ stanza of type "error" that is appropriate to the original payload; if not, then the server MUST reply with a <service-unavailable/> stanza error.

8.5.3. localpart@domainpart/resourcepart

If the domainpart of the JID contained in the 'to' attribute of an inbound stanza matches one of the configured domains of the server itself and the JID contained in the 'to' attribute is of the form <localpart@domainpart/resourcepart>, then the server MUST adhere to the following rules.

8.5.3.1. Resource Matches

If an available resource or connected resource exactly matches the full JID, how the stanza is processed depends on the stanza type.

- For an IQ stanza of type "get" or "set", if the intended recipient does not share presence with the requesting entity either by means of a presence subscription of type "both" or "from" or by means of directed presence, then the server SHOULD NOT deliver the IQ stanza but instead SHOULD return a <service-unavailable/> stanza error to the requesting entity. This policy helps to prevent presence leaks (see Section 11).

- For an IQ stanza of type "result" or "error", the server MUST deliver the stanza to the resource.

- For a message stanza, the server MUST deliver the stanza to the resource.

- For a presence stanza with no 'type' attribute or a 'type' attribute of "unavailable", the server MUST deliver the stanza to the resource.

- For a presence stanza of type "subscribe", "subscribed", "unsubscribe", or "unsubscribed", the server MUST follow the guidelines provided under Section 3.

- For a presence stanza of type "probe", the server MUST follow the guidelines provided under Section 4.3.

8.5.3.2. No Resource Matches

If no available resource or connected resource exactly matches the full JID, how the stanza is processed depends on the stanza type.

8.5.3.2.1. Message

For a message stanza of type "normal", "groupchat", or "headline", the server MUST either (a) silently ignore the stanza or (b) return an error stanza to the sender, which SHOULD be <service-unavailable/>.

For a message stanza of type "chat":

- If there is no available or connected resource, the server MUST either (a) store the message offline for later delivery or (b) return an error stanza to the sender, which SHOULD be <service-unavailable/>.
- If all of the available resources have a negative presence priority then the server SHOULD (a) store the message offline for later delivery or (b) return a stanza error to the sender, which SHOULD be <service-unavailable/>.
- If there is one available resource with a non-negative presence priority then the server MUST deliver the message to that resource.
- If there is more than one resource with a non-negative presence priority then the server MUST either (a) deliver the message to the "most available" resource or resources (according to the server's implementation-specific algorithm, e.g., treating the resource or resources with the highest presence priority as "most available") or (b) deliver the message to all of the non-negative resources that have opted in to receive chat messages.

For a message stanza of type "error", the server MUST silently ignore the stanza.

8.5.3.2.2. Presence

For a presence stanza with no 'type' attribute or a 'type' attribute of "unavailable", the server MUST silently ignore the stanza.

For a presence stanza of type "subscribe", the server MUST follow the guidelines provided under Section 3.1.3.

For a presence stanza of type "subscribed", "unsubscribe", or "unsubscribed", the server MUST ignore the stanza.

For a presence stanza of type "probe", the server MUST follow the guidelines provided under Section 4.3.

8.5.3.2.3. IQ

For an IQ stanza, the server MUST return a <service-unavailable/> stanza error to the sender.

8.5.4. Summary of Message Delivery Rules

The following table summarizes the message (not stanza) delivery rules described earlier in this section. The left column shows various combinations of conditions (non-existent account, no active resources, only one resource and it has a negative presence priority, only one resource and it has a non-negative presence priority, or more than one resource and each one has a non-negative presence priority) and 'to' addresses (bare JID, full JID matching an available resource, or full JID matching no available resource). The subsequent columns list the four primary message types (normal, chat, groupchat, or headline) along with six possible delivery options: storing the message offline (O), bouncing the message with a stanza error (E), silently ignoring the message (S), delivering the message to the resource specified in the 'to' address (D), delivering the message to the "most available" resource or resources according to the server's implementation-specific algorithm, e.g., treating the resource or resources with the highest presence priority as "most available" (M), or delivering the message to all resources with non-negative presence priority (A -- where for chat messages "all resources" can mean the set of resources that have explicitly opted in to receiving every chat message). The '/' character stands for "exclusive or". The server SHOULD observe the rules given in section 8.1 when choosing which action to take for a particular message.

Table 1: Message Delivery Rules

Condition	Normal	Chat	Groupchat	Headline
ACCOUNT DOES NOT EXIST				
bare	S/E	S/E	E	S
full	S/E	S/E	S/E	S/E
ACCOUNT EXISTS, BUT NO ACTIVE RESOURCES				
bare	O/E	O/E	E	S
full (no match)	S/E	O/E	S/E	S/E
1+ NEGATIVE RESOURCES BUT ZERO NON-NEGATIVE RESOURCES				
bare	O/E	O/E	E	S
full match	D	D	D	D
full no match	S/E	O/E	S/E	S/E
1 NON-NEGATIVE RESOURCE				
bare	D	D	E	D
full match	D	D	D	D
full no match	S/E	D	S/E	S/E
1+ NON-NEGATIVE RESOURCES				
bare	M/A	M/A*	E	A
full match	D	D/A*	D	D
full no match	S/E	M/A*	S/E	S/E

* For messages of type "chat", a server SHOULD NOT act in accordance with option (A) unless clients can explicitly opt in to receiving all chat messages; however, methods for opting in are outside the scope of this specification.

9. Handling of URIs

The addresses of XMPP entities as used in communication over an XMPP network (e.g., in the 'from' and 'to' addresses of an XML stanza) MUST NOT be prepended with a Uniform Resource Identifier [URI] scheme.

However, an application that is external to XMPP itself (e.g., a page on the World Wide Web) might need to identify an XMPP entity either as a URI or as an Internationalized Resource Identifier [IRI], and an XMPP client might need to interact with such an external application (for example, an XMPP client might be invoked by clicking a link provided on a web page). In the context of such interactions, XMPP clients are encouraged to handle addresses that are encoded as

"xmpp:" URIs and IRIs as specified in [XMPP-URI] and further described in [XEP-0147]. Although XMPP clients are also encouraged to handle addresses that are encoded as "im:" URIs as specified in [CPIM] and "pres:" URIs as specified in [CPP], they can do so by removing the "im:" or "pres:" scheme and entrusting address resolution to the server as specified under Section 8.3.

10. Internationalization Considerations

For internationalization considerations, refer to the relevant section of [XMPP-CORE].

11. Security Considerations

Core security considerations for XMPP are provided in Section 13 of [XMPP-CORE], including discussion of channel encryption, authentication, information leaks, denial-of-service attacks, and interdomain federation.

Section 13.1 of [XMPP-CORE] outlines the architectural roles of clients and servers in typical deployments of XMPP, and discusses the security properties associated with those roles. These roles have an impact on the security of instant messages, presence subscriptions, and presence notifications as described in this document. In essence, an XMPP user registers (or has provisioned) an account on an XMPP server and therefore places some level of trust in the server to complete various tasks on the user's behalf, enforce security policies, etc. Thus it is the server's responsibility to:

1. Preferably mandate the use of channel encryption for communication with local clients and remote servers.

2. Authenticate any client that wishes to access the user's account.

3. Process XML stanzas to and from clients that have authenticated as the user (specifically with regard to instant messaging and presence functionality, store the user's roster, process inbound and outbound subscription requests and responses, generate and handle presence probes, broadcast outbound presence notifications, route outbound messages, and deliver inbound messages and presence notifications).

As discussed in Sections 13.1 and 13.4 of [XMPP-CORE], even if the server fulfills the foregoing responsibilities, the client does not have any assurance that stanzas it might exchange with other clients (whether on the same server or a remote server) are protected for all hops along the XMPP communication path, or within the server itself. It is the responsibility of the client to use an appropriate

technology for encryption and signing of XML stanzas if it wishes to ensure end-to-end confidentiality and integrity of its communications.

Additional considerations that apply only to instant messaging and presence applications of XMPP are defined in several places within this document; specifically:

- When a server processes an inbound presence stanza of type "probe" whose intended recipient is a user associated with one of the server's configured domains, the server MUST NOT reveal the user's presence if the sender is an entity that is not authorized to receive that information as determined by presence subscriptions (see Section 4).

- A user's server MUST NOT leak the user's network availability to entities who are not authorized to know the user's presence. In XMPP itself, authorization takes the form of an explicit subscription from a contact to the user (as described under Section 3). However, some XMPP deployments might consider an entity to be authorized if there is an existing trust relationship between the entity and the user who is generating presence information (as an example, a corporate deployment of XMPP might automatically add the user's presence information to a private directory of employees if the organization mandates the sharing of presence information as part of an employment agreement).

- When a server processes an outbound presence stanza with no type or of type "unavailable", it MUST follow the rules defined under Section 4 in order to ensure that such presence information is not sent to entities that are not authorized to know such information.

- A client MAY ignore the <status/> element when contained in a presence stanza of type "subscribe", "unsubscribe", "subscribed", or "unsubscribed"; this can help prevent "presence subscription spam".

12. Conformance Requirements

This section describes a protocol feature set that summarizes the conformance requirements of this specification. This feature set is appropriate for use in software certification, interoperability testing, and implementation reports. For each feature, this section provides the following information:

- A human-readable name

- An informational description

- A reference to the particular section of this document that normatively defines the feature

- Whether the feature applies to the Client role, the Server role, or both (where "N/A" signifies that the feature is not applicable to the specified role)

- Whether the feature MUST or SHOULD be implemented, where the capitalized terms are to be understood as described in [KEYWORDS]

The feature set specified here attempts to adhere to the concepts and formats proposed by Larry Masinter within the IETF's NEWTRK Working Group in 2005, as captured in [INTEROP]. Although this feature set is more detailed than called for by [REPORTS], it provides a suitable basis for the generation of implementation reports to be submitted in support of advancing this specification from Proposed Standard to Draft Standard in accordance with [PROCESS].

Feature: message-body
Description: Support the <body/> child element of the <message/> stanza.
Section: Section 5.2.3
Roles: Client MUST, Server N/A.

Feature: message-subject
Description: Support the <subject/> child element of the <message/> stanza.
Section: Section 5.2.4
Roles: Client SHOULD, Server N/A.

Feature: message-thread
Description: Support the <thread/> child element of the <message/> stanza.
Section: Section 5.2.5
Roles: Client SHOULD, Server N/A.

Feature: message-type-support
Description: Support reception of messages of type "normal", "chat", "groupchat", "headline", and "error".
Section: Section 5.2.2
Roles: Client SHOULD, Server N/A.

Feature: message-type-deliver
Description: Appropriately deliver messages of type "normal", "chat", "groupchat", "headline", and "error".
Section: Section 8
Roles: Client N/A, Server SHOULD.

Feature: presence-notype
Description: Treat a presence stanza with no 'type' attribute as indicating availability.
Section: Section 4.7.1
Roles: Client MUST, Server MUST.

Feature: presence-probe
Description: Send and receive presence stanzas with a 'type' attribute of "probe" for the discovery of presence information.
Section: Section 4.7.1
Roles: Client N/A, Server MUST.

Feature: presence-sub-approval
Description: Treat an outbound presence stanza of type "subscribed" as the act of approving a presence subscription request previously received from another entity, and treat an inbound presence stanza of type "subscribed" as a subscription approval from another entity.
Section: Section 3.1
Roles: Client MUST, Server MUST.

Feature: presence-sub-cancel
Description: Treat an outbound presence stanza of type "unsubscribed" as the act of denying a subscription request received from another entity or canceling a subscription approval previously granted to another entity, and treat an inbound presence stanza of type "unsubscribed" as an subscription denial or cancellation from another entity.
Section: Section 3.2
Roles: Client MUST, Server MUST.

Feature: presence-sub-preapproval
Description: Treat an outbound presence stanza of type "subscribed" in certain circumstances as the act of pre-approving a subscription request received from another entity; this includes support for the 'approved' attribute of the <item/> element within the 'jabber:iq:roster' namespace.
Section: Section 3.4
Roles: Client MAY, Server MAY.

Feature: presence-sub-request
Description: Treat an outbound presence stanza of type "subscribe" as the act of requesting a subscription to the presence information of another entity, and treat an inbound presence stanza of type "subscribe" as a presence subscription request from another entity.
Section: Section 3.1
Roles: Client MUST, Server MUST.

Feature: presence-sub-unsubscribe
Description: Treat an outbound presence stanza of type "unsubscribe" as the act of unsubscribing from another entity, and treat an inbound presence stanza of type "unsubscribe" as an unsubscribe notification from another entity.
Section: Section 3.3
Roles: Client MUST, Server MUST.

Feature: presence-unavailable
Description: Treat a presence stanza with a 'type' attribute of "unavailable" as indicating lack of availability.
Section: Section 4.7.1
Roles: Client MUST, Server MUST.

Feature: roster-get
Description: Treat an IQ stanza of type "get" containing an empty <query/> element qualified by the 'jabber:iq:roster' namespace as a request to retrieve the roster information associated with an account on a server.
Section: Section 2.1.3
Roles: Client MUST, Server MUST.

Feature: roster-set
Description: Treat an IQ stanza of type "set" containing a <query/> element qualified by the 'jabber:iq:roster' namespace as a request to add or update the item contained in the <query/> element.
Section: Section 2.1.5
Roles: Client MUST, Server MUST.

Feature: roster-push
Description: Send a roster push to each interested resource whenever the server-side representation of the roster information materially changes, or handle such a push when received from the server.
Section: Section 2.1.6
Roles: Client MUST, Server MUST.

Feature: roster-version
Description: Treat the 'ver' attribute of the <query/> element qualified by the 'jabber:iq:roster' namespace as an identifier of the particular version of roster information being sent or received.
Section: Section 2.1.1
Roles: Client SHOULD, Server MUST.

13. References

13.1. Normative References

[DELAY] Saint-Andre, P., "Delayed Delivery", XSF XEP 0203, September 2009.

[KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[XML] Maler, E., Yergeau, F., Sperberg-McQueen, C., Paoli, J., and T. Bray, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, November 2008, <http://www.w3.org/TR/2008/REC-xml-20081126>.

[XML-NAMES] Bray, T., Hollander, D., and A. Layman, "Namespaces in XML", W3C REC-xml-names, January 1999, <http://www.w3.org/TR/REC-xml-names>.

[XMPP-CORE] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.

13.2. Informative References

[CPIM] Peterson, J., "Common Profile for Instant Messaging (CPIM)", RFC 3860, August 2004.

[CPP] Peterson, J., "Common Profile for Presence (CPP)", RFC 3859, August 2004.

[DOS] Handley, M., Rescorla, E., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, December 2006.

[IMP-MODEL] Day, M., Rosenberg, J., and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000.

[IMP-REQS] Day, M., Aggarwal, S., and J. Vincent, "Instant Messaging / Presence Protocol Requirements", RFC 2779, February 2000.

[IMP-SRV] Peterson, J., "Address Resolution for Instant Messaging and Presence", RFC 3861, August 2004.

[INTEROP] Masinter, L., "Formalizing IETF Interoperability Reporting", Work in Progress, October 2005.

[IRC] Kalt, C., "Internet Relay Chat: Architecture", RFC 2810, April 2000.

[IRI] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, January 2005.

[PROCESS] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[REPORTS] Dusseault, L. and R. Sparks, "Guidance on Interoperation and Implementation Reports for Advancement to Draft Standard", BCP 9, RFC 5657, September 2009.

[RFC3920] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 3920, October 2004.

[RFC3921] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", RFC 3921, October 2004.

[SASL] Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.

[SIP-PRES] Rosenberg, J., "A Presence Event Package for the Session Initiation Protocol (SIP)", RFC 3856, August 2004.

[TLS] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[TLS-CERTS] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, March 2011.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version 6.0", 2010, <http://www.unicode.org/versions/Unicode6.0.0/>.

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[UUID] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

[XEP-0016] Millard, P. and P. Saint-Andre, "Privacy Lists", XSF XEP 0016, February 2007.

[XEP-0045] Saint-Andre, P., "Multi-User Chat", XSF XEP 0045, July 2008.

[XEP-0054] Saint-Andre, P., "vcard-temp", XSF XEP 0054, July 2008.

[XEP-0071] Saint-Andre, P., "XHTML-IM", XSF XEP 0071, September 2008.

[XEP-0115] Hildebrand, J., Saint-Andre, P., and R. Troncon, "Entity Capabilities", XSF XEP 0115, February 2008.

[XEP-0147] Saint-Andre, P., "XMPP URI Scheme Query Components", XSF XEP 0147, September 2006.

[XEP-0160] Saint-Andre, P., "Best Practices for Handling Offline Messages", XSF XEP 0160, January 2006.

[XEP-0163] Saint-Andre, P. and K. Smith, "Personal Eventing Protocol", XSF XEP 0163, July 2010.

[XEP-0191] Saint-Andre, P., "Simple Communications Blocking", XSF XEP 0191, February 2007.

[XEP-0201] Saint-Andre, P., Paterson, I., and K. Smith, "Best Practices for Message Threads", XSF XEP 0201, November 2010.

[XEP-0237] Saint-Andre, P. and D. Cridland, "Roster Versioning", XSF XEP 0237, March 2010.

[XML-DATATYPES]
Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition", W3C REC-xmlschema-2, October 2004, <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>.

[XML-SCHEMA]
Thompson, H., Maloney, M., Mendelsohn, N., and D. Beech, "XML Schema Part 1: Structures Second Edition", World Wide Web Consortium Recommendation REC-xmlschema-1-20041028, October 2004, <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028>.

[XMPP-ADDR]
Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Address Format", RFC 6122, March 2011.

[XMPP-URI]
Saint-Andre, P., "Internationalized Resource Identifiers (IRIs) and Uniform Resource Identifiers (URIs) for the Extensible Messaging and Presence Protocol (XMPP)", RFC 5122, February 2008.

[VCARD] Dawson, F. and T. Howes, "vCard MIME Directory Profile", RFC 2426, September 1998.

Appendix A. Subscription States

This section provides detailed information about subscription states and server processing of subscription-related presence stanzas (i.e., presence stanzas of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed").

A.1. Defined States

There are four primary subscription states (these states are described from the perspective of the user, not the contact):

None: The user does not have a subscription to the contact's presence, and the contact does not have a subscription to the user's presence.

To: The user has a subscription to the contact's presence, but the contact does not have a subscription to the user's presence.

From: The contact has a subscription to the user's presence, but the user does not have a subscription to the contact's presence.

Both: Both the user and the contact have subscriptions to each other's presence (i.e., the union of 'from' and 'to').

Implementation Note: For the purpose of processing subscription-related presence stanzas as described in the following sections, a subscription state of "None" includes the case of the contact not being in the user's roster at all, i.e., an unknown entity from the perspective of the user's roster.

The foregoing states are supplemented by various sub-states related to pending inbound and outbound subscriptions, thus yielding nine possible subscription states:

1. "None" = Contact and user are not subscribed to each other, and neither has requested a subscription from the other; this is reflected in the user's roster by subscription='none'.

2. "None + Pending Out" = Contact and user are not subscribed to each other, and user has sent contact a subscription request but contact has not replied yet; this is reflected in the user's roster by subscription='none' and ask='subscribe'.

3. "None + Pending In" = Contact and user are not subscribed to each other, and contact has sent user a subscription request but user has not replied yet. This state might or might not be reflected in the user's roster, as follows: if the user has created a

roster item for the contact then the server MUST maintain that roster item and also note the existence of the inbound presence subscription request, whereas if the user has not created a roster item for the contact then the user's server MUST note the existence of the inbound presence subscription request but MUST NOT create a roster item for the contact (instead, the server MUST wait until the user has approved the subscription request before adding the contact to the user's roster).

4. "None + Pending Out+In" = Contact and user are not subscribed to each other, contact has sent user a subscription request but user has not replied yet, and user has sent contact a subscription request but contact has not replied yet; this is reflected in the user's roster by subscription='none' and ask='subscribe'.

5. "To" = User is subscribed to contact (one-way); this is reflected in the user's roster by subscription='to'.

6. "To + Pending In" = User is subscribed to contact, and contact has sent user a subscription request but user has not replied yet; this is reflected in the user's roster by subscription='to'.

7. "From" = Contact is subscribed to user (one-way); this is reflected in the user's roster by subscription='from'.

8. "From + Pending Out" = Contact is subscribed to user, and user has sent contact a subscription request but contact has not replied yet; this is reflected in the user's roster by subscription='from' and ask='subscribe'.

9. "Both" = User and contact are subscribed to each other (two-way); this is reflected in the user's roster by subscription='both'.

A.2. Server Processing of Outbound Presence Subscription Stanzas

Outbound presence subscription stanzas enable the user to manage his or her subscription to the contact's presence (via the "subscribe" and "unsubscribe" types), and to manage the contact's access to the user's presence (via the "subscribed" and "unsubscribed" types).

The following rules apply to outbound routing of the stanza as well as changes to the user's roster. (These rules are described from the perspective of the user, not the contact. In addition, "S.N." stands for SHOULD NOT and "M.N." stands for MUST NOT.)

A.2.1. Subscribe

Table 2: Processing of outbound "subscribe" stanzas

EXISTING STATE	ROUTE?	NEW STATE
"None"	MUST [1]	"None + Pending Out"
"None + Pending Out"	MUST	no state change
"None + Pending In"	MUST [1]	"None + Pending Out+In"
"None + Pending Out+In"	MUST	no state change
"To"	MUST	no state change
"To + Pending In"	MUST	no state change
"From"	MUST [1]	"From + Pending Out"
"From + Pending Out"	MUST	no state change
"Both"	MUST	no state change

[1] A state change to "pending out" includes setting the ’ask’ flag to a value of "subscribe" in the user’s roster.

A.2.2. Unsubscribe

Table 3: Processing of outbound "unsubscribe" stanzas

EXISTING STATE	ROUTE?	NEW STATE
"None"	MUST	no state change
"None + Pending Out"	MUST	"None"
"None + Pending In"	MUST	no state change
"None + Pending Out+In"	MUST	"None + Pending In"
"To"	MUST	"None"
"To + Pending In"	MUST	"None + Pending In"
"From"	MUST	no state change
"From + Pending Out"	MUST	"From"
"Both"	MUST	"From"

A.2.3. Subscribed

Table 4: Processing of outbound "subscribed" stanzas

EXISTING STATE	ROUTE?	NEW STATE
"None"	M.N.	pre-approval [1]
"None + Pending Out"	M.N.	pre-approval [1]
"None + Pending In"	MUST	"From"
"None + Pending Out+In"	MUST	"From + Pending Out"
"To"	M.N.	pre-approval [1]
"To + Pending In"	MUST	"Both"
"From"	M.N.	no state change
"From + Pending Out"	M.N.	no state change
"Both"	M.N.	no state change

[1] Detailed information regarding subscription pre-approval is provided under Section 3.4.

A.2.4. Unsubscribed

Table 5: Processing of outbound "unsubscribed" stanzas

EXISTING STATE	ROUTE?	NEW STATE
"None"	S.N.	no state change [1]
"None + Pending Out"	S.N.	no state change [1]
"None + Pending In"	MUST	"None"
"None + Pending Out+In"	MUST	"None + Pending Out"
"To"	S.N.	no state change [1]
"To + Pending In"	MUST	"To"
"From"	MUST	"None"
"From + Pending Out"	MUST	"None + Pending Out"
"Both"	MUST	"To"

[1] This event can result in cancellation of a subscription pre-approval, as described under Section 3.4.

A.3. Server Processing of Inbound Presence Subscription Stanzas

Inbound presence subscription stanzas request a subscription-related action from the user (via the "subscribe" type), inform the user of subscription-related actions taken by the contact (via the

"unsubscribe" type), or enable the user to manage the contact's access to the user's presence information (via the "subscribed" and "unsubscribed" types).

The following rules apply to delivery of the inbound stanza as well as changes to the user's roster. (These rules for server processing of inbound presence subscription stanzas are described from the perspective of the user, not the contact. In addition, "S.N." stands for SHOULD NOT.)

A.3.1. Subscribe

Table 6: Processing of inbound "subscribe" stanzas

EXISTING STATE	DELIVER?	NEW STATE
"None"	MUST [1]	"None + Pending In"
"None + Pending Out"	MUST	"None + Pending Out+In"
"None + Pending In"	S.N.	no state change
"None + Pending Out+In"	S.N.	no state change
"To"	MUST	"To + Pending In"
"To + Pending In"	S.N.	no state change
"From"	S.N. [2]	no state change
"From + Pending Out"	S.N. [2]	no state change
"Both"	S.N. [2]	no state change

[1] If the user previously sent presence of type "subscribed" as described under Appendix A.2.3 and Section 3.4, then the server MAY auto-reply with "subscribed" and change the state to "From" rather than "None + Pending In".

[2] Server SHOULD auto-reply with "subscribed".

A.3.2. Unsubscribe

When the user's server receives a presence stanza of type "unsubscribe" for the user from the contact, if the stanza results in a subscription state change from the user's perspective then the user's server MUST change the state, MUST deliver the presence stanza from the contact to the user, and SHOULD auto-reply by sending a presence stanza of type "unsubscribed" to the contact on behalf of the user. Otherwise the user's server MUST NOT change the state and (because there is no state change) SHOULD NOT deliver the stanza. These rules are summarized in the following table.

Table 7: Processing of inbound "unsubscribe" stanzas

EXISTING STATE	DELIVER?	NEW STATE
"None"	S.N.	no state change
"None + Pending Out"	S.N.	no state change
"None + Pending In"	MUST [1]	"None"
"None + Pending Out+In"	MUST [1]	"None + Pending Out"
"To"	S.N.	no state change
"To + Pending In"	MUST [1]	"To"
"From"	MUST [1]	"None"
"From + Pending Out"	MUST [1]	"None + Pending Out"
"Both"	MUST [1]	"To"

[1] Server SHOULD auto-reply with "unsubscribed".

A.3.3. Subscribed

When the user's server receives a presence stanza of type "subscribed" for the user from the contact, if there is no pending outbound request for access to the contact's presence information, then it MUST NOT change the subscription state and (because there is no state change) SHOULD NOT deliver the stanza to the user. If there is a pending outbound request for access to the contact's presence information and the inbound presence stanza of type "subscribed" results in a subscription state change, then the user's server MUST change the subscription state and MUST deliver the stanza to the user. If the user already is subscribed to the contact's presence information, the inbound presence stanza of type "subscribed" does not result in a subscription state change; therefore the user's server MUST NOT change the subscription state and (because there is no state change) SHOULD NOT deliver the stanza to the user. These rules are summarized in the following table.

Table 8: Processing of inbound "subscribed" stanzas

EXISTING STATE	DELIVER?	NEW STATE
"None"	S.N.	no state change
"None + Pending Out"	MUST	"To"
"None + Pending In"	S.N.	no state change
"None + Pending Out+In"	MUST	"To + Pending In"
"To"	S.N.	no state change
"To + Pending In"	S.N.	no state change
"From"	S.N.	no state change
"From + Pending Out"	MUST	"Both"
"Both"	S.N.	no state change

A.3.4. Unsubscribed

When the user's server receives a presence stanza of type "unsubscribed" for the user from the contact, if there is a pending outbound request for access to the contact's presence information or if the user currently is subscribed to the contact's presence information, then the user's server MUST change the subscription state and MUST deliver the stanza to the user. Otherwise, the user's server MUST NOT change the subscription state and (because there is no state change) SHOULD NOT deliver the stanza. These rules are summarized in the following table.

Table 9: Processing of inbound "unsubscribed" stanzas

EXISTING STATE	DELIVER?	NEW STATE
"None"	S.N.	no state change
"None + Pending Out"	MUST	"None"
"None + Pending In"	S.N.	no state change
"None + Pending Out+In"	MUST	"None + Pending In"
"To"	MUST	"None"
"To + Pending In"	MUST	"None + Pending In"
"From"	S.N.	no state change
"From + Pending Out"	MUST	"From"
"Both"	MUST	"From"

Appendix B. Blocking Communication

Sections 2.3.5 and 5.4.10 of [IMP-REQS] require that a compliant instant messaging and presence technology needs to enable a user to block communications from selected users. Protocols for doing so are specified in [XEP-0016] and [XEP-0191].

Appendix C. vCards

Sections 3.1.3 and 4.1.4 of [IMP-REQS] require that it be possible to retrieve out-of-band contact information for other users (e.g., telephone number or email address). An XML representation of the vCard specification defined in RFC 2426 [VCARD] is in common use within the XMPP community to provide such information but is out of scope for this specification (documentation of this protocol is contained in [XEP-0054]).

Appendix D. XML Schema for jabber:iq:roster

The following schema formally defines the 'jabber:iq:roster' namespace used in this document, in conformance with [XML-SCHEMA]. Because validation of XML streams and stanzas is optional, this schema is not normative and is provided for descriptive purposes only. For schemas defining core XMPP namespaces, refer to [XMPP-CORE].

```
<?xml version='1.0' encoding='UTF-8'?>

<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='jabber:iq:roster'
    xmlns='jabber:iq:roster'
    elementFormDefault='qualified'>

  <xs:element name='query'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='item'
                    minOccurs='0'
                    maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:attribute name='ver'
                    type='xs:string'
                    use='optional'/>
    </xs:complexType>
  </xs:element>
```

```
    <xs:element name='item'>
      <xs:complexType>
        <xs:sequence>
          <xs:element ref='group'
                      minOccurs='0'
                      maxOccurs='unbounded'/>
        </xs:sequence>
        <xs:attribute name='approved'
                      type='xs:boolean'
                      use='optional'/>
        <xs:attribute name='ask'
                      use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:NMTOKEN'>
              <xs:enumeration value='subscribe'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name='jid'
                      type='xs:string'
                      use='required'/>
        <xs:attribute name='name'
                      type='xs:string'
                      use='optional'/>
        <xs:attribute name='subscription'
                      use='optional'
                      default='none'>
          <xs:simpleType>
            <xs:restriction base='xs:NMTOKEN'>
              <xs:enumeration value='both'/>
              <xs:enumeration value='from'/>
              <xs:enumeration value='none'/>
              <xs:enumeration value='remove'/>
              <xs:enumeration value='to'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:element name='group' type='xs:string'/>

  </xs:schema>
```

Appendix E. Differences From RFC 3921

Based on consensus derived from implementation and deployment experience as well as formal interoperability testing, the following substantive modifications were made from [RFC3921] (in addition to numerous changes of an editorial nature).

- The protocol for session establishment was determined to be unnecessary and therefore the content previously defined in Section 3 of RFC 3921 was removed. However, for the sake of backward-compatibility server implementations are encouraged to advertise support for the feature, even though session establishment is a "no-op".

- In order to more seamlessly repair lack of synchronization in subscription states between rosters located at different servers, clarified and modified error handling related to presence subscription requests, presence probes and presence notifications.

- Changed the 'from' address for presence probes so that it is the bare JID, not the full JID.

- Adjusted and clarified stanza delivery rules based on implementation and deployment experience.

- Explicitly specified that a server is allowed to deliver a message stanza of type "normal" or "chat" to all resources if it has a method for allowing resources to opt in to such behavior.

- Allowed a server to use its own algorithm for determining the "most available" resource for the purpose of message delivery, but mentioned the recommended algorithm from RFC 3921 (based on presence priority) as one possible algorithm.

- Added optional versioning of roster information to save bandwidth in cases where the roster has not changed (or has changed very little) between sessions; the relevant protocol interactions were originally described in [XEP-0237].

- Added optional server support for pre-approved presence subscriptions via presence stanzas of type "subscribed", including a new 'approved' attribute that can be set to "true" (for a pre-approved subscription) or "false" (the default).

- Added optional 'parent' attribute to <thread/> element.

- Moved the protocol for communications blocking (specified in Section 10 of RFC 3921) back to [XEP-0016], from which it was originally taken.

- Recommended returning presence unavailable in response to probes.

- Clarified handling of presence probes sent to full JIDs.

- Explicitly specified that the default value for the presence <priority/> element is zero.

- Removed recommendation to support the "_im" and "_pres" SRV records.

Appendix F. Acknowledgements

This document is an update to, and derived from, RFC 3921. This document would have been impossible without the work of the contributors and commenters acknowledged there.

Hundreds of people have provided implementation feedback, bug reports, requests for clarification, and suggestions for improvement since publication of RFC 3921. Although the document editor has endeavored to address all such feedback, he is solely responsible for any remaining errors and ambiguities.

Some of the text about roster versioning was borrowed from [XEP-0237], and some of the text about message threads was borrowed from [XEP-0201].

Special thanks are due to Kevin Smith, Matthew Wild, Dave Cridland, Waqas Hussain, Philipp Hancke, Florian Zeitz, Jonas Lindberg, Jehan Pages, Tory Patnoe, and others for their comments during Working Group Last Call.

Thanks also to Richard Barnes for his review on behalf of the Security Directorate.

The Working Group chairs were Ben Campbell and Joe Hildebrand. The responsible Area Director was Gonzalo Camarillo.

Author's Address

Peter Saint-Andre
Cisco
1899 Wyknoop Street, Suite 600
Denver, CO 80202
USA

Phone: +1-303-308-3282
EMail: psaintan@cisco.com