Network Working Group
Internet-Draft
Expires: January 26, 2004

P. Saint-Andre
J. Miller
Jabber Software Foundation
July 28, 2003

XMPP Instant Messaging
draft-ietf-xmpp-im-15

Status of this Memo

Copyright Notice

Abstract

This document describes specific extensions to and applications of the Extensible Messaging and Presence Protocol (XMPP) that provide the basic instant messaging and presence functionality defined in RFC 2779.

Table of Contents

1. Introduction

1.1 Overview

The core features of the Extensible Messaging and Presence Protocol are defined in XMPP Core [1]. These features -- specifically XML streams, stream authentication and encryption, and the <message/>, <presence/>, and <iq/> children of the stream root -- provide the building blocks for many types of near-real-time applications, which may be layered on top of the core by sending application-specific data qualified by particular XML namespaces. This document describes extensions to and applications of XMPP Core that provide the basic functionality expected of an instant messaging (IM) and presence application as defined in RFC 2779 [2].

1.2 Requirements

For the purposes of this document, the requirements of a basic instant messaging and presence application are defined by RFC 2779 [2]. At a high level, RFC 2779 stipulates that a user must be able to complete the following use cases:

- Exchange messages with other users
- Exchange presence information with other users
- Manage subscriptions to and from other users
- Manage items in a contact list (in XMPP this is called a "roster")
- Block communications to or from specific other users

Detailed definitions of these functionality areas are contained in RFC 2779, and the interested reader is directed to that document regarding the requirements addressed herein.

Note: while XMPP-based instant messaging and presence meets the requirements of RFC 2779, it was not designed explicitly with RFC 2779 in mind, since the base protocol evolved through an open development process within the Jabber open-source community before RFC 2779 was written. Note also that although protocols addressing many other functionality areas have been defined in the Jabber community, such protocols are not included in this document because they are not required by RFC 2779 [2].

1.3 Terminology

This document inherits the terminology defined in XMPP Core [1].

The capitalized key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [3].

1.4 Discussion Venue

The authors welcome discussion and comments related to the topics presented in this document. The preferred forum is the <xmppwg@jabber.org> mailing list, for which archives and subscription information are available at <http://www.jabber.org/cgi-bin/mailman/listinfo/xmppwg/>.

1.5 Intellectual Property Notice

This document is in full compliance with all provisions of Section 10 of RFC 2026. Parts of this specification use the term "jabber" for identifying namespaces and other protocol syntax. Jabber[tm] is a registered trademark of Jabber, Inc. Jabber, Inc. grants permission to the IETF for use of the Jabber trademark in association with this specification and its successors, if any.

2. Establishing a Session

Most instant messaging and presence applications based on XMPP are implemented via a client-server architecture that requires a user to establish a session on a server in order to engage in the expected instant messaging and presence activities. However, there are several pre-conditions that must be met before a user may establish such a

session. These include:

1. Account Provisioning -- methods for account provisioning include account creation by a server administrator as well as in-band account registration using the 'jabber:iq:register' namespace; the latter method is documented by the Jabber Software Foundation [4] at <http://www.jabber.org/protocol/> but is out of scope for this document.

2. Authentication and Resource Authorization -- methods for completing these pre-conditions are documented in XMPP Core [1]; note that client authentication with a server MUST include an authorization identity that specifies the full JID (<user@somedomain/resource>) associated with the connection for addressing purposes.

Once a client has authenticated with a server and has authorized a full JID, it SHOULD request that the server activate an instant messaging session for the client. This is accomplished by means of the 'urn:ietf:params:xml:ns:xmpp-session' namespace:

Step 1: Client requests session with server:

```
<iq type='set' id='sess_1'>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
</iq>
```

Step 2: Server informs client that session has been created:

```
<iq type='result' id='sess_1'/>
```

Several error conditions are possible. For example, the server may encounter an internal condition that prevents it from creating the session, the username or authorization identity may lack permissions to create a session, or there may already be an active session associated with an authzid of the same name.

If the server encounters an internal condition that prevents it from creating the session, it MUST return an error.

Step 2 (alt): Server responds with error (internal server error):

```
<iq type='error' id='sess_1'>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
  <error type='wait'>
    <internal-server-error
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

If the username or authorization identity is not allowed to create a session, the server MUST return an error.

Step 2 (alt): Server responds with error (username or authzid not allowed to create session):

```
<iq type='error' id='sess_1'>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
  <error type='auth'>
    <not-allowed
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

If there is already an active session associated with an authzid of the same name, the server MUST either (1) terminate the active session and allow the newly-requested session, or (2) disallow the newly-requested session and maintain the existing session. Which of these the server does is up to the implementation, although it is RECOMMENDED to implement (1).

Step 2 (alt): Server informs client of resource conflict (the desired resource name is already in use by another active connection):

```
<iq type='error' id='sess_1'>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
  <error type='cancel'>
    <conflict
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

After establishing a session, a client SHOULD send initial presence and request its roster as described below, although these actions are NOT REQUIRED.

3. Exchanging Messages

Exchanging messages is a basic use of XMPP and is effected when a user generates a message stanza that is addressed to another user (or, more generally, another entity). As defined under Section 10, the sender's server is responsible for delivering the message to the intended recipient (if the recipient is on the same server) or for routing the message to the recipient's server (if the recipient is on a different server).

For information regarding the syntax of message stanzas as well as their defined attributes and child elements, refer to XMPP Core [1].

3.1 Specifying an Intended Recipient

An instant messaging client SHOULD specify an intended recipient for a message by providing the JID of an entity other than the sender in the 'to' attribute of the <message/> stanza. If the message is being sent in reply to a message previously received from an address of the form <user@somedomain/resource> (e.g., within the context of a chat session), the value of the 'to' address SHOULD be the full JID (<user@somedomain/resource>) rather than merely <user@somedomain> unless the sender has knowledge (via presence) that the intended recipient's resource is no longer available. 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 <user@somedomain> rather than <user@somedomain/resource>.

3.2 Specifying a Message Type

As mentioned in XMPP Core [1], there are several defined types of messages (specified by means of a 'type' attribute within the <message/> element). In the context of an instant messaging application, a client SHOULD include a message type in order to capture the conversational context of the message, thus providing a hint regarding presentation (e.g., in a GUI). If the 'type' attribute is included, it SHOULD have one of the following values (any other value MAY be ignored):

- chat -- The message is sent in the context of a one-to-one chat conversation. A compliant client SHOULD present an interface enabling one-to-one chat between the two parties, including an appropriate conversation history.

- error -- An error has occurred related to a previous message sent by the sender (for details regarding stanza error syntax, refer to XMPP Core [1]). A compliant client SHOULD present an appropriate interface informing the sender of the nature of the error.

- groupchat -- The message is sent in the context of a multi-user chat environment. A compliant client SHOULD present an interface enabling many-to-many chat between the parties, including a roster of parties in the chatroom and an appropriate conversation history.

- headline -- The message is probably generated by an automated service that delivers or broadcasts content (news, sports, market information, RSS feeds, etc.). No reply to the message is expected, and a compliant client SHOULD present an interface that appropriately differentiates the message from standalone messages, chat sessions, or groupchat sessions (e.g., by not providing the recipient with the ability to reply).

- normal -- The message is a standalone message to which the recipient MAY reply if desired. This is the default type.

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".

Although the 'type' attribute is NOT REQUIRED, 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').

3.3 Specifying a Message Body

A message stanza MAY (and often will) contain a child <body/> element specifying the primary meaning of the message. The content of the body element MUST be XML character data and the element MUST NOT contain mixed content. If it is necessary to provide the primary meaning in an alternate form (e.g., formatted using XHTML), the alternate form MUST be contained in some other child of the message

stanza. However, multiple <body/> elements MAY be included to provide the primary meaning in different languages, as long as each such element possesses an 'xml:lang' attribute with a distinct value.

Example: A message with a body:

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

3.4 Specifying a Message Subject

A message stanza MAY contain one or more child <subject/> elements specifying the topic of the message. The content of the subject element MUST be XML character data and the element MUST NOT contain mixed content. Multiple <subject/> elements MAY be included, as long as each such element possesses an 'xml:lang' attribute with a distinct value.

Example: A message with a subject:

```
<message
    to='romeo@example.net'
    from='juliet@example.com/balcony'
    xml:lang='en'>
  <subject>I implore you!</subject>
  <subject xml:lang='cz'>
    &#x00DA;p&#x011B;nliv&#x011B; prosim!
```

```
  </subject>
  <body>Wherefore art thou, Romeo?</body>
  <body xml:lang='cz'>
    Pro&#x010D;e&#x017D; jsi ty, Romeo?
  </body>
</message>
```

3.5 Specifying a Conversation Thread

A message stanza MAY contain a child <thread/> element specifying the conversation thread in which the message is situated, for the purpose of tracking the conversation. The content of the <thread/> element is a random string that is generated by the sender in accordance with the algorithm specified in XMPP Core [1]; this string SHOULD be copied back to the sender in subsequent replies.

Example: A threaded conversation:

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

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

<message
    to='romeo@example.net/orchard'
    from='juliet@example.com/balcony'
    type='chat'>
```

```
  <body>How cam'st thou hither, tell me, and wherefore?</body>
  <thread>e0ffe42b28561960c6b12b944a092794b9683a38</thread>
</message>
```


4. Exchanging Presence Information

Exchanging presence information is made relatively straightforward within XMPP by using presence stanzas. However, we see here a contrast to the handling of messages: although a client MAY send directed presence information to another entity, normally presence information is sent from a client to its server (with no 'to' address) and then broadcasted by the server to any entities that are subscribed to the presence of the sending entity. (Note: in the terminology of RFC 2778 [5], the only watchers in XMPP are subscribers.)

For information regarding the syntax of presence stanzas as well as their defined attributes and child elements, refer to XMPP Core [1].

4.1 Client and Server Responsibilities

When a client connects to its server, it SHOULD (but is NOT REQUIRED to) send initial presence to the server in order to signal its availability for communications. As defined herein, the initial presence stanza (1) MUST possess no 'to' address (signalling that it is meant to be handled by the server on behalf of the user) and (2) MUST possess no 'type' attribute (signalling the user's availability).

Upon receiving initial presence from a client, the user's server MUST do the following:

1. Send presence probes (i.e., presence stanzas whose 'type' attribute is set to a value of "probe") from the full JID (<user@somedomain/resource>) of the user to the bare JID (<contact@otherdomain>) of any contacts to which the user is subscribed in order to determine if they are available; such contacts are those which are present in the user's roster with the 'subscription' attribute set to a value of "to" or "both". (Note: a user or client SHOULD NOT send presence probes.)

2. Broadcast initial presence from the full JID (<user@somedomain/resource>) of the user to the bare JID (<contact@otherdomain>) of any contacts that are subscribed to the user's presence; such contacts are those which are present in the user's roster with the 'subscription' attribute set to a value of "from" or "both".

Upon receiving a presence probe from the user, the contact's server MUST send to the user the last known availability information (i.e., the full XML of the last presence stanza) provided by each of the contact's active sessions (if there exist no active sessions, the server SHOULD NOT reply to the presence probe). The server MUST send

this information subject to domain-specific access rules, and only if the user is in the contact's roster with a subscription state of "from" or "both" and the contact has not blocked outbound presence notifications to the user's bare or full JID (as defined in Section 9.11). (Note: if the server receives a presence probe from a subdomain of the server's hostname or another such trusted service, it MAY provide presence information about the user to that entity.)

Upon receiving initial presence from the user, the contact's server MUST deliver the user's presence stanza to the full JIDs (<contact@otherdomain/resource>) associated with all of the contact's active sessions, but only if the user is in the contact's roster with a subscription state of "to" or "both" and the contact has not blocked inbound presence notifications from the user's bare or full JID (as defined in Section 9.10).

If the user's server receives a presence stanza of type "error" in response to the initial presence that it forwarded to a contact on behalf of the user, it SHOULD NOT send further presence updates to that contact (until and unless it receives a presence probe from the contact).

After sending initial presence, the user MAY update and broadcast its presence information at any time during its active session by sending a presence stanza with no 'to' address and either no 'type' attribute or a 'type' attribute with a value of "unavailable". (Note: a user's client SHOULD NOT send a presence update to broadcast information that changes independently of the user's presence and availability.) If the presence stanza lacks a 'type' attribute (i.e., expresses availability), the user's server MUST broadcast the full XML of that presence stanza to all contacts (1) that are in the user's roster with a subscription type of "from" or "both", (2) to whom the user has not blocked outbound presence, and (3) from whom the server has not received a presence error during the user's session. If the presence stanza has a 'type' attribute set to a value of "unavailable", the user's server MUST broadcast the full XML of that presence stanza to all contacts meeting the three conditions just mentioned, as well as to any entities to which the user has sent directed available presence during the user's session (if the user has not yet sent directed unavailable presence to that entity).

A user MAY send directed presence to another entity (i.e., a presence stanza with a 'to' attribute whose value is the JID of the other entity and with either no 'type' attribute or a 'type' attribute whose value is "unavailable"). There are three possible cases:

1. If the user sends directed 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, the user's server MUST route or deliver the full XML of that presence stanza (subject to privacy rules) 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, the user's server MUST route or deliver 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 connected resource from which the user sent the directed presence become unavailable, the user's server MUST broadcast 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, the user's server MUST treat the entities to which the user sends directed presence in the same way that it treats the entities listed in Case 2 above.

Before ending its session with a server, a client SHOULD gracefully become unavailable by sending a final presence stanza that possesses no 'to' attribute and that possesses a 'type' attribute whose value is "unavailable" (optionally, the final presence stanza MAY contain one or more <status/> elements specifying the reason why the user is no longer available). However, the user's server MUST NOT depend on receiving final presence from an available resource, since the resource may become unavailable unexpectedly. If the user's server detects that one of the user's resources has become unavailable for any reason (either gracefully or ungracefully), it MUST broadcast unavailable presence to all contacts (1) that are in the user's roster with a subscription type of "from" or "both", (2) to whom the user has not blocked outbound presence, and (3) from whom the server has not received a presence error during the user's session; the user's server MUST also send that unavailable presence stanza to any entities to which the user has sent directed presence during the user's session for that resource (if the user has not yet sent directed unavailable presence to that entity). Any presence stanza with no 'type' attribute and no 'to' attribute that is sent after sending directed unavailable presence or broadcasted unavailable presence MUST be broadcasted by the server to all subscribers.

4.2 Specifying Availability Status

A client MAY provide further information about its availability status by using the <show/> element. As mentioned in XMPP Core [1], the recognized values for the show element are:

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

Example: Availability status:

```
<presence>
  <show>dnd</show>
</presence>
```

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

4.3 Specifying Detailed Status Information

In conjunction with the <show/> element, a client MAY provide detailed status information by using the <status/> element. The content of this element is a natural-language description of the user's current availability status. The content of the status element MUST be XML character data and the element MUST NOT contain mixed content. Multiple <status/> elements MAY be included, as long as each such element possesses an 'xml:lang' attribute with a distinct value.

Example: Detailed status information:

```
<presence xml:lang='en'>
  <show>dnd</show>
  <status>Wooing Juliet</status>
  <status xml:lang='cz'>Ja dvo&#x0159;&#x00ED;m Juliet</status>
</presence>
```

4.4 Specifying Presence Priority

A client MAY provide a priority for its resource by using the

<priority/> element. The content of this element is an integer whose value is between -128 and +127. If a client does not provide the

priority element in a presence stanza, its server SHOULD assume that the priority value is zero.

Example: Presence priority:

```
<presence xml:lang='en'>
  <show>dnd</show>
  <status>Wooing Juliet</status>
  <status xml:lang='cz'>Ja dvo&#x0159;&#x00ED;m Juliet</status>
  <priority>1</priority>
</presence>
```

4.5 Determining When a Contact Went Offline

The server MUST maintain a record of the time at which a user became unavailable (whether gracefully or ungracefully). An authorized subscriber to that user's presence MAY request the time of last activity by sending an IQ stanza to the user's bare JID (<user@somedomain>) containing an empty <query/> element qualified by the 'jabber:iq:last' namespace:

Example: Requesting the last active time of an offline user:

```
<iq type='get' to='user@somedomain'>
  <query xmlns='jabber:iq:last'/>
</iq>
```

If the entity requesting the time of last activity is an authorized subscriber to the user's presence (i.e., exists in the user's roster with the 'subscription' attribute set to a value of "from" or "both") and the user is not blocking IQ stanzas to and from the entity (as defined in Section 9.12), the server SHOULD return an IQ stanza of type "result" with the number of seconds since the user was last active:

Example: Returning the last active time of an offline user:

```
<iq from='user@somedomain' type='result' to='contact@otherdomain/resource'>
```

```
  <query seconds='76490' xmlns='jabber:iq:last'/>
</iq>
```

If the entity requesting the time of last activity is not an authorized subscriber to the user's presence (i.e., does not exist in the user's roster with the 'subscription' attribute set to a value of "from" or "both"), the server MUST return an IQ stanza of type "error" with an error condition of forbidden:

Example: Requester is forbidden to view the last active time of an offline user:

```
<iq from='user@somedomain' type='error' to='contact@otherdomain/resource'>
  <query xmlns='jabber:iq:last'/>
  <error type='auth'>
    <forbidden
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

Note: this document defines responses to requests for last active time only with regard to JIDs of the form <user@somedomain>, and only with regard to JIDs that correspond to an (offline) instant messaging user. The behavior of other JID forms (e.g., <user@somedomain/resource> or <somedomain>) and entity types (e.g., online user or host) is out of scope and undefined.

4.6 Presence Examples

The examples in this section illustrate the presence-related protocols described above. The user is romeo@example.net, he has authorized a resource "orchard", and he has the following individuals in his roster:

- juliet@example.com (subscription="both" and she has two active sessions, one whose resource is "chamber" and another whose resource is "balcony")
- benvolio@example.org (subscription="to")
- mercutio@shakespeare.lit (subscription="from")

Example 1: User sends initial presence:

```
<presence/>
```


Example 2: User's server sends presence probe to contacts with subscription="to" and subscription="both" on behalf of the user's connected resource:

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

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

Example 3: User's server sends initial presence to contacts with subscription="from" and subscription="both" on behalf of the user's connected resource:

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

<presence
    from='romeo@example.net/orchard'
    to='mercutio@shakespeare.lit'/>
```

Example 4: Contacts' server replies to presence probe on behalf of all of the contact's available resources:

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

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

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

Example 5: Contact's server delivers user's initial presence to all of the contact's available resources or returns error to user:

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

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

<presence
    type='error'
    from='mercutio@shakespeare.lit'
    to='romeo@example.net/orchard'>
  <error type='cancel'>
    <remote-server-not-found
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</presence>
```


Example 6: User sends directed presence to another user not in his roster:

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

Example 7: User sends updated available presence information for broadcasting:

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

Example 8: Updated presence information is delivered only to one contact (not those from whom an error was received or to whom the user sent directed presence):

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

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

Example 9: One of the contact's resources sends final presence:

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


Example 10: Contact's server sends unavailable presence information to user:

```
<presence
```

```
    type='unavailable'
    from='juliet@example.com/balcony'
    to='romeo@example.net/orchard'/>
```

Example 11: User sends final presence:

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

Example 12: Unavailable presence information is delivered to contact's one remaining resource as well as to the person to whom the user sent directed presence:

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

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

5. Managing Subscriptions

In order to protect the privacy of instant messaging users and any other entities, presence and availability information is disclosed only to other entities that the user has approved. When a user has agreed that another entity may view its presence, the entity is said to have a subscription to the user's presence information. A subscription lasts across sessions; indeed, it lasts until the subscriber unsubscribes or the subscribee cancels the previously-granted subscription. Subscriptions are managed within XMPP by sending presence stanzas containing specially-defined attributes.

Note: there are important interactions between subscriptions and rosters; these are defined under Integration of Roster Items and Presence Subscriptions (Section 7), and the reader must refer to that section for a complete understanding of presence subscriptions.

5.1 Requesting a Subscription

A request to subscribe to another entity's presence is made by sending a presence stanza of type "subscribe".

Example: Sending a subscription request:

```
<presence to='juliet@example.com' type='subscribe'/>
```

If the subscription request is being sent to another instant messaging user, the JID supplied in the 'to' attribute SHOULD be of the form <contact@otherdomain> rather than <contact@otherdomain/resource>.

A user's server MUST NOT automatically accept subscription requests on the user's behalf. All subscription requests MUST be directed to the user's client. If there is no available resource associated with the user when the subscription request is received by the server, the user's server MUST store the subscription request offline for delivery when the user next becomes available. (Note: if a resource has authorized a session but has not provided initial presence, the server SHOULD NOT consider it to be available and therefore SHOULD NOT send subscription requests to it.)

5.2 Handling a Subscription Request

When a client receives a subscription request from another entity, it MUST either accept the request by sending a presence stanza of type "subscribed" or decline the request by sending a presence stanza of

type "unsubscribed".

Example: Accepting a subscription request:

```
<presence to='romeo@example.net' type='subscribed'/>
```

Example: Denying a presence subscription request:

```
<presence to='romeo@example.net' type='unsubscribed'/>
```

5.3 Cancelling a Subscription from Another Entity

If a user would like to cancel a previously-granted subscription request, it sends a presence stanza of type "unsubscribed".

Example: Cancelling a previously granted subscription request:

```
<presence to='romeo@example.net' type='unsubscribed'/>
```

5.4 Unsubscribing from Another Entity's Presence

If a user would like to unsubscribe from the presence of another entity, it sends a presence stanza of type "unsubscribe".

Example: Unsubscribing from an entity's presence:

```
<presence to='juliet@example.com' type='unsubscribe'/>
```

6. Managing One's Roster

In XMPP, one's contact list is called a roster, which consists of any number of specific roster items, each roster item being identified by a unique JID (usually of the form <contact@otherdomain>). A user's roster is stored by the user's server on the user's behalf so that the user may access roster information from any available resource.

Note: there are important interactions between rosters and subscriptions; these are defined under Integration of Roster Items and Presence Subscriptions (Section 7), and the reader must refer to that section for a complete understanding of roster management.

Note: a server MUST ignore any 'to' address on a roster "set", and MUST treat any roster "set" as applying to the sender. For added safety, a client SHOULD check the "from" address of a roster "push" to ensure that it is from a trusted source; specifically, the stanza should have no 'from' attribute (i.e., implicitly from the server) or the JID contained in the 'from' attribute should match the user's bare JID or full JID; otherwise, the client SHOULD ignore the roster "push".

6.1 Retrieving One's Roster on Login

Upon connecting to the server, a client SHOULD request the roster (however, because receiving the roster may not be desirable for all resources, e.g., a connection with limited bandwidth, the client's request for the roster is NOT REQUIRED). If an available resource does not request the roster during a session, the server SHOULD NOT send it presence subscriptions and associated "roster pushes".

Example: Client requests current roster from server:

```
<iq type='get' id='roster_1'>
  <query xmlns='jabber:iq:roster'/>
</iq>
```


Example: Client receives roster from the server:

```
<iq id='roster_1' type='result'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='romeo@example.net'
        name='Romeo'
        subscription='both'>
      <group>Friends</group>
    </item>
    <item
        jid='mercutio@shakespeare.lit'
        name='Mercutio'
        subscription='both'>
      <group>Friends</group>
    </item>
    <item
        jid='benvolio@example.org'
        name='Benvolio'
        subscription='both'>
      <group>Friends</group>
    </item>
  </query>
</iq>
```

6.2 Adding a Roster Item

At any time, a user MAY add an item to his or her roster.

Example: Client adds a new item:

```
<iq type='set' id='roster_2'>
  <query xmlns='jabber:iq:roster'>
    <item
        name='Nurse'
        jid='nurse@example.com'>
      <group>Servants</group>
    </item>
  </query>
</iq>
```

The value of the 'jid' attribute SHOULD be of the form <user@somedomain>, especially if the item is associated with another (human) instant messaging user.

The server MUST update the roster information in persistent storage, and also push the change out to all of the user's available resources

that have requested the roster. This "roster push" consists of an IQ set from the server to the client and enables all available resources to remain in sync with the server-based roster information.

Example: Server (1) pushes the updated roster information to all available resources and (2) replies with an IQ result to the sending resource:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        name='Nurse'
        jid='nurse@example.com'
        subscription='none'>
      <group>Servants</group>
    </item>
  </query>
</iq>
```

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        name='Nurse'
        jid='nurse@example.com'
        subscription='none'>
      <group>Servants</group>
    </item>
  </query>
</iq>

<iq type='result' id='roster_2'/>
```

Example: Connected resources reply with an IQ result to the server:

```
<iq
    from='juliet@example.com/balcony'
    to='example.com'
    type='result'/>
<iq
    from='juliet@example.com/chamber'
    to='example.com'
    type='result'/>
```

6.3 Updating a Roster Item

Updating an existing roster item (e.g., changing the group) is done in the same way as adding a new roster item, i.e., by sending the

roster item in an IQ set to the server.

Example: User updates roster item (added group):

```
<iq type='set' id='roster_3'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='romeo@example.net'
        name='Romeo'
        subscription='both'>
      <group>Friends</group>
      <group>Lovers</group>
```

```
    </item>
  </query>
</iq>
```

As with adding a roster item, when updating a roster item the server MUST update the roster information in persistent storage, and also initiate a "roster push" to all of the user's available resources that have requested the roster.

6.4 Deleting a Roster Item

At any time, a user MAY delete an item from its roster by doing an IQ set and making sure that the value of the 'subscription' attribute is "remove" (a compliant server MUST ignore any other values of the 'subscription' attribute when received from a client).

Example: Client removes an item:

```
<iq type='set' id='roster_4'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='nurse@example.com'
        subscription='remove'/>
  </query>
</iq>
```

As with adding a roster item, when deleting a roster item the server MUST update the roster information in persistent storage, initiate a "roster push" to all of the user's available resources that have requested the roster (with the 'subscription' attribute set to a value of "remove"), and send an IQ result to the initiating resource.

For further information about the implications of this command, see Section 7.6.

7. Integration of Roster Items and Presence Subscriptions

7.1 Overview

Some level of integration between roster items and presence

subscriptions is normally expected by an instant messaging user regarding the user's subscriptions to and from other contacts. This section describes the level of integration that must be supported within XMPP IM.

There are four primary subscription states:

- None -- Neither the user nor the contact is subscribed to the other's presence
- To -- The user is subscribed to the contact's presence but there is no subscription from the contact to the user
- From -- There is a subscription from the contact to the user, but the user has not subscribed to the contact's presence
- Both -- Both the user and the contact are subscribed to each other's presence (i.e., the union of 'from' and 'to')

Each of these states is reflected in the roster of both the user and the contact, thus resulting in durable subscription states. A detailed explanation of how these subscription states interact with roster items is provided in the following sub-sections.

If a connected resource does not both send initial presence and request the roster, the server SHOULD NOT send it presence subscription requests or "roster pushes".

The 'from' and 'to' addresess are OPTIONAL in roster pushes; if included, their values SHOULD be the full JID of the resource for that session. A client MUST acknowledge each "roster push" with an IQ stanza of type "result" (for the sake of brevity, these stanzas are not shown in the following examples but are required by XMPP Core [1]).

7.2 User Subscribes to Contact

The process by which a user subscribes to a contact, including the interaction between roster items and subscription states, is defined below.

1. In preparation for being able to render the contact in the user's client interface and for the server to keep track of the

subscription, the user's client SHOULD perform a "roster set" for the new roster item. This request consists of an IQ stanza of type='set' containing a <query/> element in the 'jabber:iq:roster' namespace, which in turn contains an <item/> element that defines the new roster item; the <item/> element MUST possess a 'jid' attribute, MAY possess a 'name' attribute, MUST NOT possess a 'subscription' attribute, and MAY contain one or more <group/> child elements:

```
<iq type='set' id='int1'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>
```

2. As a result, the user's server (1) MUST initiate a "roster push" for the new roster item to all available resources associated with this user that have requested the roster, setting the 'subscription' attribute to a value of "none"; and (2) MUST reply with an IQ stanza of type='result':

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='none'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<iq type='result' id='int1'/>
```

3. If the user wants to request a subscription to the contact's presence, the user's client MUST send a presence stanza of type='subscribe' to the contact:

```
<presence to='contact@otherdomain' type='subscribe'/>
```

4. As a result, the user's server MUST initiate a second "roster push" to all of the user's available resources that have requested the roster, setting the contact to the pending sub-state of the 'none' subscription state; this pending

sub-state is denoted by the inclusion of the ask='subscribe' attribute in the roster item:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='none'
        ask='subscribe'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>
```

Note: if the user did not create a roster item before sending the subscription request, the server MUST now create one and send a "roster push" to all of the user's available resources that have requested the roster, absent the 'name' attribute and the <group/> child.

5. The user's server MUST also stamp the presence stanza of type "subscribe" with the user's bare JID (i.e., <user@somedomain>) as the 'from' address. If the contact is served by a different host than the user, the user's server MUST route the presence stanza to the contact's server for delivery to the contact (this case is assumed throughout; however, if the contact is served by the same host, then the server can simply deliver the presence stanza directly):

```
<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='subscribe'/>
```

6. Upon receiving the presence stanza of type "subscribe" addressed to the contact, the contact's server must determine if there is at least one active session in which the contact has sent available presence and has requested the roster. If so, it MUST deliver the subscription request to the contact (if not, the contact's server MUST store the subscription request offline for delivery when this condition is next met). No matter when the

subscription request is delivered, the contact must decide whether or not to accept it (subject to configured preferences, the contact's client MAY accept or deny the subscription request without presenting it to the contact). Here we assume the "happy path" that the contact accepts the subscription request (the alternate flow of declining the subscription request is defined

in Section 7.2.1). In this case, the contact's client (1) SHOULD perform a roster set specifying the desired nickname and group for the user; and (2) MUST send a presence stanza of type "subscribed" to the user in order to accept the subscription request.

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence to='user@somedomain' type='subscribed'/>
```

7. As a result, the contact's server (1) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, containing a roster item for the user with the subscription state set to 'from'; (2) MUST route the presence stanza of type "subscribed" to the user; and (3) MUST send available presence from all of the contact's available resources to the user:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='from'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
```

```
</iq>

<presence
    from='contact@otherdomain/resource'
    to='user@somedomain'
    type='subscribed'/>

<presence
    from='contact@otherdomain/resource'
    to='user@somedomain'/>
```

8. Upon receiving the presence stanza of type "subscribed" addressed to the user, the user's server MUST first verify that the contact

is in the user's roster with either of the following states: (a) subscription='none', ask='subscribe' or (b) subscription='from', ask='subscribe'. If the contact is not in the user's roster with either of those states, the user's server MUST silently ignore the presence stanza of type "subscribed" (i.e., it MUST NOT route it to the user, modify the user's roster, or generate a roster push to the user's available resources). If the contact is in the user's roster with either of those states, the user's server (1) MUST deliver the presence stanza of type "subscribed" from the contact to the user; (2) MUST initiate a "roster push" to all of the user's available resources that have requested the roster, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "to"; and (3) MUST deliver the available presence stanza received from each of the contact's available resources to each of the user's available resources:

```
<presence
    to='user@somedomain'
    from='contact@otherdomain'
    type='subscribed'/>

<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='to'
        name='MyContact'>
```

```
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<presence
    from='contact@otherdomain/resource'
    to='user@somedomain/resource'/>
```

9. Upon receiving the presence stanza of type "subscribed", the user SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "subscribe" to the contact or denying it by sending a presence stanza of type "unsubscribe" to the contact; this step lets the user's server know that it must no longer send notification of the subscription state change to the user (see Section 8.6).

From the perspective of the user, there now exists a subscription to the contact; from the perspective of the contact, there now exists a

subscription from the user. (Note: If at this point the user sends another subscription request to the contact, the user's server MUST silently ignore that request.)

7.2.1 Alternate Flow: Contact Declines Subscription Request

The above activity flow represents the "happy path" related to the user's subscription request to the contact. The main alternate flow occurs if the contact denies the user's subscription request.

1. If the contact wants to deny the request, the contact's client MUST send a presence stanza of type "unsubscribed" to the user (instead of the presence stanza of type "subscribed" sent in Step 6 of Section 7.2):

```
<presence to='user@somedomain' type='unsubscribed'/>
```

2. As a result, the contact's server MUST route the presence stanza of type "unsubscribed" to the user, first stamping the 'from' address as the bare JID (<contact@otherdomain>) of the contact:

```
<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>
```

3. Upon receiving the presence stanza of type "unsubscribed" addressed to the user, the user's server (1) MUST deliver that presence stanza to the user and (2) MUST initiate a "roster push" to all of the user's available resources that have requested the roster, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "none" and with no 'ask' attribute:

```
<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='none'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
```



```
</iq>
```

4. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribe" to the contact or denying it by sending a presence stanza of type "subscribe" to the contact; this step lets the user's server know that it must no longer send notification of the subscription state change to the user (see Section 8.6).

As a result of this activity, the contact is now in the user's roster with a subscription state of "none", whereas the user is not in the contact's roster at all.

7.3 Creating a Mutual Subscription

The user and contact can build on the foregoing to create a mutual subscription (i.e., a subscription of type "both"). The process is defined below.

1. If the contact wants to create a mutual subscription, the contact MUST send a subscription request to the user (subject to configured preferences, the contact's client MAY send this automatically):

```
<presence to='user@somedomain' type='subscribe'/>
```

2. As a result, the contact's server (1) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, with the user still in the 'from' subscription state but with a pending 'to' subscription denoted by the inclusion of the ask='subscribe' attribute in the roster item; and (2) MUST route the presence stanza of type "subscribe" to the user, first stamping the 'from' address as the bare JID (<contact@otherdomain>) of the contact:


```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='from'
        ask='subscribe'
        name='SomeUser'>
```

```
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='subscribe'/>
```

3. Upon receiving the presence stanza of type "subscribe" addressed to the user, the user's server must determine if there is at least one active session in which the user has sent available presence and has requested the roster. If so, the user's server MUST deliver the subscription request to the user (if not, it MUST store the subscription request offline for delivery when this condition is next met). No matter when the subscription request is delivered, the user must then decide whether or not to accept it (subject to configured preferences, the user's client MAY accept or deny the subscription request without presenting it to the user). Here we assume the "happy path" that the user accepts the subscription request (the alternate flow of declining the subscription request is defined in Section 7.3.1). In this case, the user's client MUST send a presence stanza of type "subscribed" to the contact in order to accept the subscription request.

```
<presence to='contact@otherdomain' type='subscribed'/>
```

4. As a result, the user's server (1) MUST initiate a "roster push" to all of the user's available resources that have requested the roster, containing a roster item for the contact with the 'subscription' attribute set to a value of "both"; (2) MUST route the presence stanza of type "subscribed" to the contact, first stamping the 'from' address as the bare JID (<user@somedomain>) of the user; and (3) MUST send available presence from each of the user's available resources to the contact:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='both'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='subscribed'/>

<presence
    from='user@somedomain/resource'
    to='contact@otherdomain'/>
```

5. Upon receiving the presence stanza of type "subscribed" addressed to the contact, the contact's server MUST first verify that the user is in the contact's roster with either of the following states: (a) subscription='none', ask='subscribe' or (b) subscription='from', ask='subscribe'. If the user is not in the contact's roster with either of those states, the contact's server MUST silently ignore the presence stanza of type "subscribed" (i.e., it MUST NOT route it to the contact, modify the contact's roster, or generate a roster push to the contact's available resources). If the user is in the contact's roster with either of those states, the contact's server (1) MUST deliver the presence stanza of type "subscribed" from the user to the contact; (2) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "both"; and (3) MUST deliver the available presence stanza received from each of the user's available resources to each of the contact's available resources:

```
<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='subscribed'/>

<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='both'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain/resource'
    to='contact@otherdomain/resource'/>
```

6. Upon receiving the presence stanza of type "subscribed", the contact SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "subscribe" to the user or denying it by sending a presence stanza of type "unsubscribe" to the user; this step lets the contact's server know that it must no longer send notification of the subscription state change to the contact (see Section 8.6).

The user and the contact now have a mutual subscription to each other's presence -- i.e., the subscription is of type "both". The user's server MUST now send the user's current presence information to the contact. (Note: If at this point the user sends a subscription request to the contact or the contact sends a subscription request to the user, the sending user's server MUST silently ignore that request and not route it to the intended recipient.)

7.3.1 Alternate Flow: User Declines Subscription Request

The above activity flow represents the "happy path" related to the contact's subscription request to the user. The main alternate flow occurs if the user denies the contact's subscription request.

1. If the user wants to deny the request, the user's client MUST send a presence stanza of type "unsubscribed" to the contact (instead of the presence stanza of type "subscribed" sent in Step 3 of Section 7.3):


```
<presence to='contact@otherdomain' type='unsubscribed'/>
```

2. As a result, the user's server MUST route the presence stanza of type "unsubscribed" to the contact, first stamping the 'from' address as the bare JID (<user@somedomain>) of the user:

```
<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribed'/>
```

3. Upon receiving the presence stanza of type "unsubscribed" addressed to the contact, the contact's server (1) MUST deliver that presence stanza to the contact; and (2) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "from" and with no 'ask' attribute:

```
<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribed'/>

<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='from'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>
```

4. Upon receiving the presence stanza of type "unsubscribed", the

contact SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribe" to the user or denying it by sending a presence stanza of type "subscribe" to the user; this step lets the contact's server know that it must no longer send notification of the subscription state change to the contact (see Section 8.6).

As a result of this activity, there has been no change in the subscription state; i.e., the contact is in the user's roster with a subscription state of "to" and the user is in the contact's roster with a subscription state of "from".

7.4 Unsubscribing

At any time after subscribing to a contact's presence, a user MAY unsubscribe. While the XML that the user sends to make this happen is the same in all instances, the subsequent subscription state is different depending on the subscription state obtaining when the unsubscribe "command" was sent. Both possible scenarios are defined below.

7.4.1 Case #1: Unsubscribing When Subscription is Not Mutual

In the first case, the user has a subscription to the contact but the contact does not have a subscription to the user (i.e., the subscription is not yet mutual).

1. If the user wants to unsubscribe from the contact's presence, the user MUST send a presence stanza of type "unsubscribe" to the contact:

```
<presence to='contact@otherdomain' type='unsubscribe'/>
```

2. As a result, the user's server (1) MUST send a "roster push" to all of the user's available resources that have requested the roster, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "none"; and (2) MUST route the presence stanza of type "unsubscribe" to the contact, first stamping the 'from' address as the bare JID (<user@somedomain>) of the user:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='none'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribe'/>
```

3. Upon receiving the presence stanza of type "unsubscribe" addressed to the contact, the contact's server (1) MUST initiate a "roster push" to all available resources associated with the

contact that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "none" (if the contact is offline, the contact's server MUST modify the roster item and send that modified item the next time the contact requests the roster); and (2) MUST deliver the "unsubscribe" state change notification to the contact:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='none'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
```

```
    type='unsubscribe'/>
```

4. Upon receiving the presence stanza of type "unsubscribe", the contact SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribed" to the user or denying it by sending a presence stanza of type "subscribed" to the user; this step lets the contact's server know that it must no longer send notification of the subscription state change to the contact (see Section 8.6).

5. The contact's server then (1) MUST send a presence stanza of type "unsubscribed" to the user; and (2) SHOULD send unavailable presence from the contact to the user:

```
<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```


6. When the user's server receives a presence stanza of type "unsubscribed" and/or unavailable presence, it MUST deliver them to the user:

```
<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```

7. Upon receiving the presence stanza of type "unsubscribed", the

user SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribe" to the contact or denying it by sending a presence stanza of type "subscribe" to the contact; this step lets the user's server know that it must no longer send notification of the subscription state change to the user (see Section 8.6).

7.4.2 Case #2: Unsubscribing When Subscription is Mutual

In the second case, the user has a subscription to the contact and the contact also has a subscription to the user (i.e., the subscription is mutual).

1. If the user wants to unsubscribe from the contact's presence, the user MUST send a presence stanza of type "unsubscribe" to the contact:

```
<presence to='contact@otherdomain' type='unsubscribe'/>
```

2. As a result, the user's server (1) MUST send a "roster push" to all of the user's available resources that have requested the roster, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "from"; and (2) MUST route the presence stanza of type "unsubscribe" to the contact, first stamping the 'from' address as the bare JID (<user@somedomain>) of the user:


```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='from'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
```

```
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribe'/>
```

3. Upon receiving the presence stanza of type "unsubscribe" addressed to the contact, the contact's server (1) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "to" (if the contact is offline, the contact's server MUST modify the roster item and send that modified item the next time the contact requests the roster); and (2) MUST deliver the "unsubscribe" state change notification to the contact:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='to'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribe'/>
```

4. Upon receiving the presence stanza of type "unsubscribe", the contact SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribed" to the user or denying it by sending a presence stanza of type "subscribed" to the user; this step lets the contact's server know that it must no longer send

notification of the subscription state change to the contact (see

Section 8.6).

5. The contact's server then (1) MUST send a presence stanza of type "unsubscribed" to the user; and (2) SHOULD send unavailable presence from the contact to the user:

```
<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```

6. When the user's server receives a presence stanza of type "unsubscribed" and/or unavailable presence, it MUST deliver them to the user:

```
<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```

7. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribe" to the contact or denying it by sending a presence stanza of type "subscribe" to the contact; this step lets the user's server know that it must no longer send notification of the subscription state change to the user (see Section 8.6).

Note: Obviously this does not result in removal of the roster item from the user's roster, and the contact still has a subscription to the user's presence. In order to both completely cancel a mutual subscription and fully remove the roster item from the user's roster, the user should update the roster item with subscription='remove' as defined in Section 7.6.

7.5 Cancelling a Subscription

At any time after approving a subscription request from a user, a contact MAY cancel that subscription. While the XML that the contact sends to make this happen is the same in all instances, the subsequent subscription state is different depending on the subscription state obtaining when the cancellation was sent. Both possible scenarios are defined below.

7.5.1 Case #1: Cancelling When Subscription is Not Mutual

In the first case, the user has a subscription to the contact but the contact does not have a subscription to the user (i.e., the subscription is not yet mutual).

1. If the contact wants to cancel the user's subscription, the contact MUST send a presence stanza of type "unsubscribed" to the user:

```
<presence to='user@somedomain' type='unsubscribed'/>
```

2. As a result, the contact's server (1) MUST send a "roster push" to all of the contact's available resources that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "none"; (2) MUST route the presence stanza of type "unsubscribed" to the user, first stamping the 'from' address as the bare JID (<contact@otherdomain>) of the contact; and (3) SHOULD send unavailable presence from the contact to the user:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='none'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```

3. Upon receiving the presence stanza of type "unsubscribed" addressed to the user, the user's server (1) MUST initiate a "roster push" to all of the user's available resources that have requested the roster, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "none" (if the user is offline, the user's server MUST modify the roster item and send that modified item the next time the user requests the roster); (2) MUST deliver the "unsubscribed" state change notification to the user; and (3) MUST deliver the unavailable presence to the user:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='none'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```

4. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribe" to the contact or denying it by sending a presence stanza of type "subscribe" to the contact; this step lets the user's server know that it must no longer send notification of the subscription state change to the user (see Section 8.6).

7.5.2 Case #2: Cancelling When Subscription is Mutual

In the second case, the user has a subscription to the contact and the contact also has a subscription to the user (i.e., the subscription is mutual).

1. If the contact wants to cancel the user's subscription, the contact MUST send a presence stanza of type "unsubscribed" to the user:

```
<presence to='user@somedomain' type='unsubscribed'/>
```

2. As a result, the contact's server (1) MUST send a "roster push" to all of the contact's available resources that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "to"; (2) MUST route the presence stanza of type "unsubscribed" to the user,

first stamping the 'from' address as the bare JID (<contact@otherdomain>) of the contact; and (3) SHOULD send unavailable presence from the contact to the user:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='to'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unavailable'/>
```

3. Upon receiving the presence stanza of type "unsubscribed" addressed to the user, the user's server (1) MUST initiate a "roster push" to all of the user's available resources that have requested the roster, containing an updated roster item for the contact with the 'subscription' attribute set to a value of "from" (if the user is offline, the user's server MUST modify the roster item and send that modified item the next time the user requests the roster); and (2) MUST deliver the "unsubscribed" state change notification to the user; and (3) MUST deliver the unavailable presence to the user:


```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='from'
        name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<presence
    from='contact@otherdomain'
    to='user@somedomain'
    type='unsubscribed'/>

<presence
    from='contact@otherdomain'
```

```
    to='user@somedomain'
    type='unavailable'/>
```

4. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either accepting it by sending a presence stanza of type "unsubscribe" to the contact or denying it by sending a presence stanza of type "subscribe" to the contact; this step lets the user's server know that it must no longer send notification of the subscription state change to the user (see Section 8.6).

Note: Obviously this does not result in removal of the roster item from the contact's roster, and the contact still has a subscription to the user's presence. In order to both completely cancel a mutual subscription and fully remove the roster item from the contact's roster, the contact should update the roster item with subscription='remove' as defined in Section 7.6.

7.6 Removing a Roster Item and Cancelling All Subscriptions

Because there may be many steps involved in completely removing a roster item and cancelling subscriptions in both directions, XMPP IM includes a "shortcut" method for doing so. The process may be initiated no matter what the current subscription state is by sending a roster set containing an item for the contact with the 'subscription' attribute set to a value of "remove":


```
<iq type='set' id='remove1'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='remove'/>
  </query>
</iq>
```

When the user removes a contact from his or her roster by setting the 'subscription' attribute to a value of "remove", the user's server (1) MUST automatically cancel any existing presence subscription

between the user and the contact (both 'to' and 'from' as appropriate); (2) MUST remove the roster item from the user's roster and inform all of the user's available resources of the roster item removal; (3) MUST inform the resource that initiated the removal of success; and (4) SHOULD send unavailable presence to the contact:

```
<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribe'/>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribed'/>

<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@otherdomain'
        subscription='remove'/>
  </query>
</iq>

<iq type='result' id='remove1'/>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unavailable'/>
```

Upon receiving the presence stanza of type "unsubscribe", the contact's server (1) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "to" (if the contact is offline, the contact's server MUST modify the roster item and send

that modified item the next time the contact requests the roster); and (2) MUST also deliver the "unsubscribe" state change notification to the contact:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='to'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribe'/>
```

Upon receiving the presence stanza of type "unsubscribed", the contact's server (1) MUST initiate a "roster push" to all available resources associated with the contact that have requested the roster, containing an updated roster item for the user with the 'subscription' attribute set to a value of "none" (if the contact is offline, the contact's server MUST modify the roster item and send that modified item the next time the contact requests the roster); and (2) MUST also deliver the "unsubscribe" state change notification to the contact:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@somedomain'
        subscription='none'
        name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unsubscribed'/>
```

Upon receiving the presence stanza of type "unavailable" addressed to the contact, the contact's server MUST deliver the unavailable

presence to the user:

```
<presence
    from='user@somedomain'
    to='contact@otherdomain'
    type='unavailable'/>
```

Note that when the user removes the contact from the user's roster, the end state of the contact's roster is that the user is still in the contact's roster with a subscription state of "none"; in order to completely remove the roster item for the user, the contact needs to also send a roster removal request.

8. Subscription States

This section summarizes information about subscription states.

8.1 Defined States

There are nine possible subscription states:

1. "None" = you and I are not subscribed to each other, and neither of us has requested a subscription from the other

2. "None + Pending Out" = you and I are not subscribed to each other, and I've sent you a subscription request but you have not responded yet

3. "None + Pending In" = you and I are not subscribed to each other, and you've sent me a subscription request but I have not responded yet

4. "None + Pending Out/In" = you and I are not subscribed to each other, you've sent me a subscription request but I have not responded yet, and I've sent you a subscription request but you have not responded yet

5. "To" = I'm subscribed to you (one-way)

6. "To + Pending In" = I'm subscribed to you, and you've send me a subscription request but I have not responded yet

7. "From" = you're subscribed to me (one-way)

8. "From + Pending Out" = you're subscribed to me, and I've sent you a subscription request but you have not responded yet

9. "Both" = we're subscribed to each other (two-way)

8.2 Server Handling of Outbound Presence, Categorized by Subscription State

This section defines how a server MUST handle an outbound presence stanza of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed" (i.e., route it to the intended recipient and/or make a change to the subscription state), categorized by the current subscription state. The general rule is that a server MUST route the stanza to the intended recipient if it would change the subscription state, and MUST NOT route the stanza if it would not change the subscription state. Detailed definitions are contained in the

following sections. Naturally, if the stanza changes the subscription state, the server MUST also change the subscription state.

8.2.1 Subscription State = None

STANZA TYPE	ROUTE?	NEW STATE
subscribe	yes	"None + Pending Out"
subscribed	no	no state change
unsubscribe	no	no state change
unsubscribed	no	no state change

8.2.2 Subscription State = None + Pending Out

STANZA TYPE	ROUTE?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"None"
unsubscribed	no	no state change

8.2.3 Subscription State = None + Pending In

STANZA TYPE	ROUTE?	NEW STATE
subscribe	yes	"None + Pending Out/In"

subscribed	yes	"From"
unsubscribe	no	no state change
unsubscribed	yes	"None"

+---+

8.2.4 Subscription State = None + Pending Out/In

STANZA TYPE	ROUTE?	NEW STATE
subscribe	no	no state change
subscribed	yes	"From + Pending Out"
unsubscribe	yes	"None + Pending In"
unsubscribed	yes	"None + Pending Out"

8.2.5 Subscription State = To

STANZA TYPE	ROUTE?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"None"
unsubscribed	no	no state change

8.2.6 Subscription State = To + Pending In

STANZA TYPE	ROUTE?	NEW STATE
subscribe	no	no state change
subscribed	yes	"Both"
unsubscribe	yes	"None + Pending In"
unsubscribed	yes	"To"

8.2.7 Subscription State = From

STANZA TYPE	ROUTE?	NEW STATE
subscribe	yes	"From + Pending Out"
subscribed	no	no state change
unsubscribe	no	no state change
unsubscribed	yes	"None"

8.2.8 Subscription State = From + Pending Out

STANZA TYPE	ROUTE?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change

| unsubscribe | yes | "From" |
| unsubscribed | yes | "None + Pending Out" |

8.2.9 Subscription State = Both

STANZA TYPE	ROUTE?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"From"
unsubscribed	yes	"To"

8.3 Server Handling of Outbound Presence, Categorized by Presence Type

This section defines how a server MUST handle an outbound presence stanza of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed" (i.e., route it to the intended recipient and/or make a change to the subscription state), categorized by presence type.

8.3.1 Subscribe

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

| "From + Pending Out" | no | no state change |
| "Both" | no | no state change |

8.3.2 Subscribed

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

8.3.3 Unsubscribe

| EXISTING STATE | ROUTE? | NEW STATE |

"None"	no	no state change
"None + Pending Out"	no	no state change
"None + Pending In"	no	no state change
"None + Pending Out/In"	yes	"None + Pending In"
"To"	yes	"None"
"To + Pending In"	yes	"None + Pending In"
"From"	no	no state change
"From + Pending Out"	yes	"From"
"Both"	yes	"From"

Note: When a user sends an outbound presence stanza of type "unsubscribe" that results in a subscription state change, the contact's server SHOULD auto-reply by sending a presence stanza of type "unsubscribed" to the user on behalf of the contact and MUST deliver that presence stanza to the contact.

8.3.4 Unsubscribed

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

8.4 Server Handling of Inbound Presence, Categorized by Subscription State

This section defines how a server MUST handle an inbound presence stanza of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed" (i.e., deliver it to the intended recipient and/or make a change to the subscription state), categorized by subscription state. (Note: some of the presence stanza type should never be received as inbound stanzas, since the sender's server MUST NOT route

them to the intended recipient; however, these stanza types are included for the sake of completeness.)

8.4.1 Subscription State = None

STANZA TYPE	DELIVER?	NEW STATE
subscribe	yes	"None + Pending In"
subscribed	no	no state change
unsubscribe	no	no state change
unsubscribed	no	no state change

8.4.2 Subscription State = None + Pending Out

STANZA TYPE	DELIVER?	NEW STATE
subscribe	yes	"None + Pending Out/In"
subscribed	yes	"To"
unsubscribe	no	no state change
unsubscribed	yes	"None"

8.4.3 Subscription State = None + Pending In

STANZA TYPE	DELIVER?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"None"
unsubscribed	no	no state change

8.4.4 Subscription State = None + Pending Out/In

STANZA TYPE	DELIVER?	NEW STATE
subscribe	no	no state change
subscribed	yes	"To + Pending In"
unsubscribe	yes	"None + Pending Out"
unsubscribed	yes	"None + Pending In"

8.4.5 Subscription State = To

STANZA TYPE	DELIVER?	NEW STATE
subscribe	yes	"To + Pending In"
subscribed	no	no state change
unsubscribe	no	no state change
unsubscribed	yes	"None"

8.4.6 Subscription State = To + Pending In

STANZA TYPE	DELIVER?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"To"
unsubscribed	yes	"None + Pending In"

8.4.7 Subscription State = From

STANZA TYPE	DELIVER?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"None"
unsubscribed	no	no state change

8.4.8 Subscription State = From + Pending Out

STANZA TYPE	DELIVER?	NEW STATE
subscribe	no	no state change
subscribed	yes	"Both"
unsubscribe	yes	"None + Pending Out"
unsubscribed	yes	"From"

8.4.9 Subscription State = Both

STANZA TYPE	DELIVER?	NEW STATE
subscribe	no	no state change
subscribed	no	no state change
unsubscribe	yes	"To"
unsubscribed	yes	"From"

8.5 Server Handling of Inbound Presence, Categorized by Presence Type

This section defines how a server MUST handle an inbound presence stanza of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed" (i.e., deliver it to the intended recipient and/or make a change to the subscription state), categorized by presence type.

8.5.1 Subscribe

EXISTING STATE	DELIVER?	NEW STATE
"None"	yes	"None + Pending In"
"None + Pending Out"	yes	"None + Pending Out/In"
"None + Pending In"	no	no state change
"None + Pending Out/In"	no	no state change
"To"	yes	"To + Pending In"
"To + Pending In"	no	no state change
"From"	no	no state change
"From + Pending Out"	no	no state change
"Both"	no	no state change

8.5.2 Subscribed

EXISTING STATE	DELIVER?	NEW STATE
"None"	no	no state change
"None + Pending Out"	yes	"To"
"None + Pending In"	no	no state change
"None + Pending Out/In"	yes	"To + Pending In"
"To"	no	no state change
"To + Pending In"	no	no state change

"From"	no	no state change
"From + Pending Out"	yes	"Both"
"Both"	no	no state change

8.5.3 Unsubscribe

EXISTING STATE	DELIVER?	NEW STATE
"None"	no	no state change
"None + Pending Out"	no	no state change
"None + Pending In"	yes	"None"
"None + Pending Out/In"	yes	"None + Pending Out"
"To"	no	no state change
"To + Pending In"	yes	"To"
"From"	yes	"None"
"From + Pending Out"	yes	"None + Pending Out
"Both"	yes	"To"

8.5.4 Unsubscribed

EXISTING STATE	DELIVER?	NEW STATE

"None"	no	no state change
"None + Pending Out"	yes	"None"
"None + Pending In"	no	no state change
"None + Pending Out/In"	yes	"None + Pending In"
"To"	yes	"None"
"To + Pending In"	yes	"None + Pending In"
"From"	no	no state change
"From + Pending Out"	yes	"From"
"Both"	yes	"From"

8.6 Server Delivery and Client Acknowledgement of Subscription State Change Notifications

When a server receives an inbound presence of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed" that consists of a subscription state change notification, in addition to sending the appropriate "roster push" (or updated roster when the roster is next requested), it MUST deliver the notification to the intended recipient at least once.

A server MUST require the recipient to approve or deny a subscription request (i.e., an inbound presence stanza of type "subscribe") and MAY require the recipient to acknowledge receipt of the state change notification. In order to require acknowledgement, a server SHOULD

send the notification to the recipient each time the recipient logs in, until the recipient acknowledges receipt of the notification by accepting or denying the relevant notification. Acknowledgement is sent by either accepting or denying the notification, as shown in the following table:

NOTIFICATION	ACCEPT	DENY
subscribe	subscribed	unsubscribed
subscribed	subscribe	unsubscribe
unsubscribe	unsubscribed	subscribed
unsubscribed	unsubscribe	subscribe

Obviously, given the foregoing subscription state charts, some the acknowledgement and denial stanzas will be routed to the contact and result in subscription state changes, while others will not. However, any such stanzas MUST result in the server's no longer sending the subscription state notification to the user.

Because the sender's server MUST automatically generate outbound presence stanzas of type "unsubscribe" and "unsubscribed" upon receiving a roster set with the 'subscription' attribute set to a value of "remove" (see Section 7.6), the server MUST treat a roster remove request as equivalent to sending those presence stanzas for purposes of determining whether to continue sending subscription state change notifications of type "subscribe" or "subscribed" to the user.

9. Blocking Communication

Most instant messaging systems have found it necessary to implement some method for users to block communications from particular other users (this is also required by sections 5.1.5, 5.1.15, 5.3.2, and 5.4.10 of RFC 2779 [2]). In XMPP this is done using the 'jabber:iq:privacy' namespace by managing one's privacy lists.

Server-side privacy lists enable successful completion of the following use cases:

- Retrieving one's privacy lists.
- Adding, removing, and editing one's privacy lists.
- Setting, changing, or declining active lists.
- Setting, changing, or declining the default list (i.e., the list that is active by default).
- Allowing or denying messages based on JID, group, or subscription type (or globally).
- Allowing or denying inbound presence notifications based on JID, group, or subscription type (or globally).
- Allowing or denying outbound presence notifications based on JID, group, or subscription type (or globally).
- Allowing or denying IQs based on JID, group, or subscription type (or globally).
- Allowing or denying all communications based on JID, group, or subscription type (or globally).

Note: presence notifications do not include presence subscriptions, only presence information that is broadcasted to entities that are subscribed to a user's presence information. Thus this includes presence stanzas with no 'type' attribute or of type='unavailable' only.

9.1 Syntax

A user MAY define one or more privacy lists, which are stored by the user's server. Each <list/> element contains one or more rules in the form of <item/> elements, and each <item/> element uses attributes to define a privacy rule type, a specific value to which the rules applies, the relevant action, and the place of the item in the

processing order.

The syntax is as follows:

```
<iq>
  <query xmlns='jabber:iq:privacy'>
    <list name='foo'>
      <item
          type='[jid|group|subscription]'
          value='bar'
          action='[accept|deny]'
          order='unsignedInt'/>
    </list>
  </query>
</iq>
```

If the type is "jid", then the 'value' attribute MUST contain a valid Jabber ID. JIDs are matched in the following order: <user@somedomain/resource>, then <user@somedomain>, then <somedomain/resource>, then <somedomain>. If the value is <user@somedomain>, then any connected resource for that user@somedomain matches. If the value is <somedomain/resource>, then only that resource matches. If the value is <somedomain>, then any user@somedomain (or subdomain) matches.

If the type is "group", then the 'value' attribute SHOULD contain the name of a group in the user's roster. (If a client attempts to update, create, or delete a list item with a group that is not in the user's roster, the server SHOULD return to the client an <item-not-found/> stanza error.)

If the type is "subscription", then the 'value' attribute MUST be one of "both", "to", "from", or "none" as defined in XMPP Core [1].

If no 'type' attribute is included, the rule provides the "fall-through" case.

The 'action' attribute MUST be included and its value MUST be either "accept" or "deny".

The 'order' attribute MUST be included and its value MUST be a non-negative integer that is unique among all items in the list. (If a client attempts to create or update a list with non-unique order values, the server MUST return to the client a <bad-request/> stanza error.

Within the 'jabber:iq:privacy' namespace, the <query/> child of a client-generated IQ stanza of type "set" MUST NOT include more than one child element (i.e., the stanza must contain only one <active/>

element, one <default/> element, or one <list/> element); if a client violates this rule, the server MUST return to the client a <bad-request/> stanza error.)

When a client adds or updates a privacy list, the <list/> element SHOULD contain at least one <item/> child element; when a client removes a privacy list, the <list/> element SHOULD contain no one <item/> child element.

When a client updates a privacy list, it must include all of the desired items (i.e., not a "delta").

9.2 Business Rules

1. If there is an active list set for a session, it affects only the session for which it is activated, and only for the duration of the session. Only the active list for that session is processed (i.e., the default list is ignored).

2. The default list applies to the user as a whole, and is processed if there is no active list set for the target session/resource to which a stanza is addressed, or if there are no current sessions for the user.

3. If there is no active list set for a session (or there are no current sessions for the user), and there is no default list, then all stanzas SHOULD BE accepted or appropriately processed by the server on behalf of the user.

4. Privacy lists SHOULD be the first routing and delivery rule applied by a server, trumping the other rules specified in Section 10.

5. The order in which privacy list items are processed by the server is important. List items MUST be processed in ascending order determined by the values of the 'order' attribute for each <item/>.

6. As soon as a stanza is matched against a privacy list, the server SHOULD appropriately handle the stanza and cease processing.

7. If no fall-through item is provided in a list, the fall-through action is assumed to be "accept".

8. If a user updates the definition for an active list, subsequent processing based on that active list MUST use the updated definition (for all resources to which that active list currently applies).

9. If a change to the subscription state or roster group of a roster item defined in an active list occurs during a user's session, subsequent processing based on that active list MUST take into account the changed state or group (for all resources to which that active list currently applies).

9.3 Retrieving One's Privacy Lists

Example: Client requests names of privacy lists from server:

```
<iq type='get' id='getlist1'>
  <query xmlns='jabber:iq:privacy'/>
</iq>
```

Example: Server sends names of privacy lists to client, preceded by active list and default list:

```
<iq type='result' id='getlist1' to='romeo@example.net/orchard'>
  <query xmlns='jabber:iq:privacy'>
    <active name='private'/>
    <default name='public'/>
    <list name='public'/>
    <list name='private'/>
    <list name='special'/>
  </query>
</iq>
```

Example: Client requests a privacy list from server:

```
<iq type='get' id='getlist2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public'/>
  </query>
</iq>
```

Example: Server sends a privacy list to client:

```
<iq type='result' id='getlist2' to='romeo@example.net/orchard'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public'>
      <item type='jid'
            value='tybalt@example.com'
            action='deny'
            order='1'/>
      <item action='allow' order='2'/>
    </list>
  </query>
</iq>
```

Example: Client requests another privacy list from server:

```
<iq type='get' id='getlist3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='private'/>
  </query>
</iq>
```

Example: Server sends another privacy list to client:

```
<iq type='result' id='getlist3' to='romeo@example.net/orchard'>
  <query xmlns='jabber:iq:privacy'>
    <list name='private'>
      <item type='subscription' value='both'
            action='allow' order='10'/>
```

```
      <item action='deny' order='15'/>
    </list>
  </query>
</iq>
```

Example: Client requests yet another privacy list from server:

```
<iq type='get' id='getlist4'>
  <query xmlns='jabber:iq:privacy'>
    <list name='special'/>
  </query>
</iq>
```


Example: Server sends yet another privacy list to client:

```
<iq type='result' id='getlist4' to='romeo@example.net/orchard'>
  <query xmlns='jabber:iq:privacy'>
    <list name='special'>
      <item
          type='jid'
          value='juliet@example.com'
          action='allow'
          order='6'/>
      <item
          type='jid'
          value='benvolio@example.org'
          action='allow'
          order='7'/>
      <item
          type='jid'
          value='mercutio@shakespeare.lit'
          action='allow'
          order='42'/>
      <item action='deny' order='666'/>
    </list>
```

```
  </query>
</iq>
```

In this example, the user has three lists: (1) 'public', which allows communications from everyone except one specific entity (this is the default list); (2) 'private', which allows communications only with contacts who have a bidirectional subscription with the user (this is the active list); and (3) 'special', which allows communications only with three specific entities.

If the user attempts to retrieve a list but a list by that name does not exist, the server MUST return an <item-not-found> stanza error to the user:

Example: Client attempts to retrieve non-existent list:

```
<iq type='error' id='getlist5'>
  <query xmlns='jabber:iq:privacy'>
    <list name='The Empty Set'/>
  </query>
  <error type='cancel'>
    <item-not-found
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```


The user is allowed to retrieve only one list at a time. If the user attempts to retrieve more than one list in the same request, the server MUST return a <bad request> stanza error to the user:

Example: Client attempts to retrieve more than one list:

```
<iq type='error' id='getlist6'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public'/>
    <list name='private'/>
    <list name='special'/>
  </query>
  <error type='modify'>
    <bad-request
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
```

```
  </error>
</iq>
```

9.4 Managing Active Lists

In order to set or change the active list currently being applied by the server, the user MUST send an IQ stanza of type "set" with a <query/> element qualified by the 'jabber:iq:privacy' namespace that contains an empty <active/> child element possessing a 'name' attribute whose value is set to the desired list name.

Example: Client requests change of active list:

```
<iq type='set' id='active1'>
  <query xmlns='jabber:iq:privacy'>
    <active name='special'/>
  </query>
</iq>
```

The server MUST activate and apply the requested list before sending the result back to the client.

Example: Server acknowledges success of active list change:

```
<iq type='result' id='active1' to='juliet@example.com/balcony'/>
```

If the user attempts to set an active list but a list by that name does not exist, the server MUST return an <item-not-found> stanza error to the user:

Example: Client attempts to set a non-existent list as active:

```
<iq type='error' id='active2'>
  <query xmlns='jabber:iq:privacy'>
    <active name='The Empty Set'/>
  </query>
  <error type='cancel'>
    <item-not-found
```

```
          xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
    </error>
  </iq>
```

In order to decline the use of any active list, a user MUST send an empty <active/> element with no name.

Example: Client declines the use of active lists:

```
<iq type='set' id='active2'>
  <query xmlns='jabber:iq:privacy'>
    <active/>
  </query>
</iq>
```

9.5 Managing the Default List

In order to change its default list, the user MUST send an IQ stanza of type "set" with a <query/> element qualified by the 'jabber:iq:privacy' namespace that contains an empty <default/> child element possessing a 'name' attribute whose value is set to the desired list name.

Example: Client requests change of default list:

```
<iq type='set' id='default1'>
  <query xmlns='jabber:iq:privacy'>
    <default name='special'/>
  </query>
</iq>
```

Example: Server acknowledges success of default list change:

```
<iq type='result' id='default1' to='juliet@example.com/balcony'/>
```

If the user attempts to set a default list but a list by that name does not exist, the server MUST return an <item-not-found> stanza error to the user:

Example: Client attempts to set a non-existent list as default:

```
<iq type='error' id='default2'>
  <query xmlns='jabber:iq:privacy'>
    <default name='The Empty Set'/>
  </query>
  <error type='cancel'>
    <item-not-found
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
  </error>
</iq>
```

In order to decline the use of a default list (i.e., to use the domain's stanza routing rules at all times), a user MUST send an empty <default/> element with no name.

Example: Client declines the use of the default list:

```
<iq type='set' id='default2'>
  <query xmlns='jabber:iq:privacy'>
    <default/>
  </query>
</iq>
```

9.6 Editing a Privacy List

In order to edit a privacy list, the user MUST send an IQ stanza of type "set" with a <query/> element qualified by the 'jabber:iq:privacy' namespace that contains one <list/> child element possessing a 'name' attribute whose value is set to the list name the user would like to edit. The <list/> element MUST contain one or more <item/> elements, which specify the user's desired changes to the list by including all elements in the list (not the "delta").

Example: Client edits a privacy list:

```
<iq type='set' id='edit1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public'>
      <item type='jid' value='tybalt@example.com'
            action='deny' order='3'/>
      <item type='jid' value='paris@example.org'
            action='deny' order='5'/>
      <item action='allow' order='68'/>
    </list>
  </query>
</iq>
```

Note: The value of the 'order' attribute for any given item is not fixed. Thus in the foregoing example if the user would like to add 4 items between the "tybalt@example.com" item and the "paris@example.org" item, the user's client MUST renumber the relevant items before submitting the list to the server.

Example: Server acknowledges success of list edit:

```
<iq type='result' id='edit1' to='juliet@example.com/balcony'/>
```

9.7 Adding a New Privacy List

The same protocol used to edit an existing list is used to create a new list. If the list name matches that of an existing list, the request to add a new list will overwrite the old one.

9.8 Removing a Privacy List

In order to remove a privacy list, the user MUST send an IQ stanza of type "set" with a <query/> element qualified by the 'jabber:iq:privacy' namespace that contains one empty <list/> child element possessing a 'name' attribute whose value is set to the list name the user would like to remove.

Example: Client removes a privacy list:

```
<iq type='set' id='remove1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='private'/>
  </query>
</iq>
```

Example: Server acknowledges success of list removal:

```
<iq type='result' id='remove1' to='juliet@example.com/balcony'/>
```

If a user attempts to remove an active list or the default list, the server MUST return a <conflict/> stanza error to the user. The user MUST first set another list to active or default before removing it.

If the user attempts to remove a list but a list by that name does not exist, the server MUST return an <item-not-found> stanza error to the user:

If the user attempts to remove more than one list in the same

request, the server MUST return a <bad request> stanza error to the user.

9.9 Blocking Messages

Server-side privacy lists enable a user to block incoming messages from other users based on the other user's JID, roster group, or subscription status (or globally). The following examples illustrate the protocol.

Example: User blocks based on JID:

```
<iq type='set' id='msg1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='message-jid-example'>
      <item type='jid' value='tybalt@example.com'
            action='deny' order='3'>
        <message/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive messages from the user with the specified JID.

Example: User blocks based on roster group:

```
<iq type='set' id='msg2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='message-group-example'>
      <item type='group' value='Enemies' action='deny' order='4'>
        <message/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive messages from any users in the specified roster group.

Example: User blocks based on subscription type:

```
<iq type='set' id='msg3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='message-sub-example'>
      <item type='subscription' value='none' action='deny' order='5'>
        <message/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive messages from any users with the specified subscription type.

Example: User blocks globally:

```
<iq type='set' id='msg4'>
  <query xmlns='jabber:iq:privacy'>
    <list name='message-global-example'>
      <item action='deny' order='6'>
        <message/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive messages from any other users.

9.10 Blocking Inbound Presence Notifications

Server-side privacy lists enable a user to block incoming presence notifications from other users based on the other user's JID, roster group, or subscription status (or globally). The following examples illustrate the protocol.

Note: presence notifications do not include presence subscriptions, only presence information that is broadcasted to the user because the user previously subscribed to a contact's presence information. Thus this includes presence stanzas with no 'type' attribute or of type='unavailable' only.

Example: User blocks based on JID:

```
<iq type='set' id='presin1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presin-jid-example'>
      <item type='jid' value='tybalt@example.com'
            action='deny' order='7'>
        <presence-in/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive presence notifications from the user with the specified JID.

Example: User blocks based on roster group:

```
<iq type='set' id='presin2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presin-group-example'>
      <item type='group' value='Enemies' action='deny' order='8'>
        <presence-in/>
      </item>
```

```
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive presence notifications from any users in the specified roster group.

Example: User blocks based on subscription type:

```
<iq type='set' id='presin3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presin-sub-example'>
      <item type='subscription' value='to' action='deny' order='9'>
        <presence-in/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive presence notifications from any users with the specified subscription type.

Example: User blocks globally:

```
<iq type='set' id='presin4'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presin-global-example'>
      <item action='deny' order='11'>
        <presence-in/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive presence notifications from any other users.

9.11 Blocking Outbound Presence Notifications

Server-side privacy lists enable a user to block outgoing presence

notifications to other users based on the other user's JID, roster group, or subscription status (or globally). The following examples illustrate the protocol.

Note: presence notifications do not include presence subscriptions, only presence information that is broadcasted to contacts because those contacts previously subscribed to the user's presence information. Thus this includes presence stanzas with no 'type' attribute or of type='unavailable' only. Note also that because information about last activity MAY be requested by a contact (as defined in Section 4.5), a user SHOULD block both outbound presence and IQs in relation to any given entity.

Example: User blocks based on JID:

```
<iq type='set' id='presout1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presout-jid-example'>
      <item type='jid' value='tybalt@example.com'
            action='deny' order='13'>
        <presence-out/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not send presence notifications to the user with the specified JID.

Example: User blocks based on roster group:

```
<iq type='set' id='presout2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presout-group-example'>
      <item type='group' value='Enemies' action='deny' order='15'>
        <presence-out/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not send presence notifications to any users in the specified roster group.

Example: User blocks based on subscription type:

```
<iq type='set' id='presout3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presout-sub-example'>
      <item type='subscription' value='from'
            action='deny' order='17'>
        <presence-out/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not send presence notifications to any users with the specified subscription type.

Example: User blocks globally:

```
<iq type='set' id='presout4'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presout-global-example'>
      <item action='deny' order='23'>
        <presence-out/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not send presence notifications to any other users.

9.12 Blocking IQs

Server-side privacy lists enable a user to block incoming IQ requests of type "get" or "set" from other users based on the other user's

JID, roster group, or subscription status (or globally). The following examples illustrate the protocol.

Example: User blocks based on JID:

```
<iq type='set' id='iq1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='iq-jid-example'>
      <item type='jid' value='tybalt@example.com'
            action='deny' order='29'>
        <iq/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive IQ requests of type "get" or "set" from the user with the specified JID.

Example: User blocks based on roster group:

```
<iq type='set' id='iq2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='iq-group-example'>
      <item type='group' value='Enemies' action='deny' order='31'>
        <iq/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive IQ requests of type "get" or "set" from any users in the specified roster group.

Example: User blocks based on subscription type:

```
<iq type='set' id='iq3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='iq-sub-example'>
      <item type='subscription' value='none'
            action='deny' order='17'>
        <iq/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive IQ requests of type "get" or "set" from any users with the specified subscription type.

Example: User blocks globally:

```
<iq type='set' id='iq4'>
  <query xmlns='jabber:iq:privacy'>
    <list name='iq-global-example'>
      <item action='deny' order='1'>
        <iq/>
      </item>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive IQ requests of type "get" or "set" from any other users.

9.13 Blocking All Communication

Server-side privacy lists enable a user to block all communications from and presence to other users based on the other user's JID, roster group, or subscription status (or globally). The following examples illustrate the protocol.

Example: User blocks based on JID:

```
<iq type='set' id='all1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='all-jid-example'>
      <item type='jid' value='tybalt@example.com'
            action='deny' order='23'/>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive any communications from, nor send any stanzas to, the user with the specified JID.

Example: User blocks based on roster group:

```
<iq type='set' id='all2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='all-group-example'>
      <item type='group' value='Enemies' action='deny' order='13'/>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive any communications from, nor send any stanzas to, any users in the specified roster group.

Example: User blocks based on subscription type:

```
<iq type='set' id='all3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='all-sub-example'>
      <item type='subscription' value='none'
            action='deny' order='11'/>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive any communications from, nor send any stanzas to, any users with the specified subscription type.

Example: User blocks globally:

```
<iq type='set' id='all4'>
  <query xmlns='jabber:iq:privacy'>
    <list name='all-global-example'>
      <item action='deny' order='7'/>
    </list>
  </query>
</iq>
```

As a result of creating and applying the foregoing list, the user will not receive any communications from, nor send any stanzas to, any other users.

9.14 Blocked Entity Attempts to Communicate with User

If a blocked entity attempts to send messages or presence notifications to the user, the user's server SHOULD silently drop the stanza and MUST NOT return an error to the sending entity.

If a blocked entity attempts to send an IQ stanza of type "get" or "set" to the user, the user's server MUST return to the sending entity a <feature-not-implemented/> stanza error, since this is the standard error code sent from a client that does not understand the namespace of an IQ get or set. IQ stanzas of other types SHOULD be silently dropped by the server.

Example: Blocked entity attempts to send IQ get:

```
<iq
    type='get'
    to='romeo@example.net'
    from='tybalt@example.com/pda'
```

```
    id='probing1'>
  <query xmlns='jabber:iq:last'/>
</iq>
```


Example: Server returns error to blocked entity:

```
<iq
    type='error'
    from='romeo@example.net'
    to='tybalt@example.com/pda'
    id='probing1'>
  <query xmlns='jabber:iq:last'/>
  <error type='cancel'>
    <feature-not-implemented
        xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
  </error>
</iq>
```

9.15 Higher-Level Heuristics

When building a representation of a higher-level privacy heuristic, a client SHOULD use the simplest possible representation.

For example, the heuristic "block all communications with any user not in my roster" could be constructed in any of the following ways:

- accept communications from all JIDs in my roster (i.e., listing each JID as a separate list item), but deny communications with everyone else

- accept communications from any user who is in one of the groups that make up my roster (i.e., listing each group as a separate list item), but deny communications from everyone else

- accept communications from any user with whom I have a subscription of 'both' or 'to' or 'from' (i.e., listing each subscription value separately), but deny communications from everyone else

- deny communications from anyone whose subscription state is 'none'

The final representation is the simplest and SHOULD be used; here is the XML that would be sent in this case:

```
<iq type='set' id='heuristic1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='heuristic-example'>
      <item type='subscription' value='none'
            action='deny' order='437'/>
    </list>
  </query>
```



```
</iq>
```

10. Server Rules for Handling XML Stanzas

Each server implementation will contain its own "delivery tree" for handling stanzas it receives. Such a tree determines whether a stanza needs to be routed to another domain, processed internally, or delivered to a connected resource associated with a registered user. The following rules apply:

10.1 No 'to' Address

If the stanza possesses no 'to' attribute, the server SHOULD process it on behalf of the entity that sent it. Because all stanzas received

from other servers MUST possess a 'to' attribute, this rule applies only to stanzas received from an entity that is connected to the server (usually an active client session). If the server receives a presence stanza with no 'to' attribute, the server SHOULD broadcast it to the entities that are subscribed to the sending entity's presence as defined under Section 4.1. If the server receives an IQ stanza of type "get" or "set" with no 'to' attribute and it understands the namespace that qualifies the content of the stanza, it MUST process the stanza on behalf of sending entity (where the meaning of "process" is determined by the semantics of the qualifying namespace).

10.2 Foreign Domain

If the hostname of the domain identifier portion of the JID contained in the 'to' attribute does not match one of the configured hostnames of the server itself or a subdomain thereof, the server SHOULD route the stanza to the foreign domain (subject to local service provisioning and security policies regarding inter-domain communication). If routing to the recipient's server is unsuccessful, the sender's server MUST return an error to the sender; if the recipient's server can be contacted but delivery by the recipient's server to the recipient is unsuccessful, the recipient's server MUST return an error to the sender by way of the sender's server.

10.3 Subdomain

If the hostname of the domain identifier portion of the JID contained in the 'to' attribute matches a subdomain of one of the configured hostnames of the server itself, the server MAY process the stanza itself or MAY route the stanza to a specialized service that is responsible for that subdomain (if any).

10.4 Bare Domain or Specific Resource

If the hostname of the domain identifier portion of the JID contained

in the 'to' attribute matches the hostname of the server itself and the JID contained in the 'to' attribute is of the form <somedomain> or <somedomain/resource>, the server (or a defined resource thereof) SHOULD process the stanza as appropriate for the stanza type. If the stanza is an IQ stanza and the server understands the namespace that

qualifies the content of the stanza, the server SHOULD process the request according to the semantics of the qualifying namespace, and MUST reply with an IQ of type "result" or "error".

10.5 User in Same Domain

If the hostname of the domain identifier portion of the JID contained in the 'to' attribute matches the hostname of the server itself and the JID contained in the 'to' attribute is of the form <user@somedomain> or <user@somedomain/resource>, the server SHOULD first apply any privacy rules (Section 9) that are in force. If privacy rules allow the stanza, it SHOULD be routed or delivered to the intended recipient of the stanza as represented by the JID contained in the 'to' attribute. The following rules apply:

1. If the JID contains a resource identifier (i.e., is of the form <user@somedomain/resource>) and there is an available resource whose authzid matches the full JID, the recipient's server SHOULD deliver the stanza to the session that exactly matches the resource identifier.

2. If the JID contains a resource identifier and there is no available resource whose authzid matches the full JID, the recipient's server SHOULD return to the sender a <recipient-unavailable/> stanza error.

3. If the JID is of the form <user@somedomain> and there is at least one available resource available for the user, the recipient's server MUST follow these rules:

    1. For message stanzas, the server SHOULD deliver the stanza to the available resource that provided the highest value for the <priority/> element (if the resource did not provide a priority, the server SHOULD consider it to have provided a value of zero). If two resources have the same priority, the server MAY use some other rule (e.g., most recent connect time or activity time) to choose between them. However, the server MUST NOT deliver the stanza to an available resource that provided a negative value for the <priority/> element.

    2. For presence stanzas other than those of type "probe", the server MUST deliver the stanza to all available resources, except that the server MUST NOT deliver the stanza to an

available resource that provided a negative value for the <priority/> element; for presence probes, the server SHOULD reply based on the rules defined in Section 4.1.

3. For IQ stanzas, the server SHOULD deliver the stanza to all available resources, except that the server MUST NOT deliver the stanza to an available resource that provided a negative value for the <priority/> element.

4. If the JID is of the form <user@somedomain> and there are no available resources associated with the user (e.g., an instant messaging user is offline), how the stanza is handled depends on the stanza type:

   1. For presence stanzas of type "subscribe", the server MUST maintain a record of the stanza, as specified under Section 4.1.

   2. For all other presence stanzas, the server SHOULD silently ignore the stanza by not storing it for later delivery or replying to it on behalf of the user.

   3. For message stanzas, the server MAY choose to store the stanza on behalf of the user and deliver it when the user next becomes available. However, if offline message storage is not enabled, the server MUST return to the sender a <service-unavailable/> stanza error. (Note: offline message storage is not defined in XMPP since it strictly is a matter of implementation and service provisioning.)

   4. For IQ stanzas, the server MUST reply 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, the server MUST reply to the stanza on behalf of the user (e.g., this is the case with the 'jabber:iq:last' protocol defined above); if not, the server MUST reply with a <service-unavailable/> stanza error.

11. IANA Considerations

For several related IANA considerations, refer to XMPP Core [1].

11.1 XML Namespace Name for Session Data

A URN sub-namespace for session-related data in the Extensible Messaging and Presence Protocol (XMPP) is defined as follows.

URI: urn:ietf:params:xml:ns:xmpp-session

Specification: [RFCXXXX]

Description: This is the XML namespace name for session-related data in the Extensible Messaging and Presence Protocol (XMPP) as defined by [RFCXXXX].

Registrant Contact: IETF, XMPP Working Group, <xmppwg@jabber.org>

12. Security Considerations

For security considerations, refer to the relevant section of XMPP Core [1].

Normative References

[1] Saint-Andre, P. and J. Miller, "XMPP Core", draft-ietf-xmpp-core-15 (work in progress), June 2003.

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

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

Informative References

[4] Jabber Software Foundation, "Jabber Software Foundation", August 2001, <http://www.jabber.org/>.

[5] Day, M., Rosenberg, J. and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000, <http://www.ietf.org/rfc/rfc2778.txt>.

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

Authors' Addresses

Peter Saint-Andre
Jabber Software Foundation

EMail: stpeter@jabber.org

Jeremie Miller
Jabber Software Foundation

EMail: jeremie@jabber.org

Appendix A. vCards

Sections 3.1.3 and 4.1.4 of RFC 2779 [2] require that it be possible to retrieve contact information for other users (e.g., telephone number or email address). An XML representation of the vCard specification defined in RFC 2426 [6] is in common use within the Jabber community to provide such information. Documentation of this protocol is maintained by the Jabber Software Foundation [4] at <http://www.jabber.org/protocol/> but is out of scope for this document.

Appendix B. XML Schemas

The following XML schemas are descriptive, not normative. For schemas defining the core features of XMPP, refer to XMPP Core [1].

B.1 session

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='urn:ietf:params:xml:ns:xmpp-session'
    xmlns='urn:ietf:params:xml:ns:xmpp-session'
    elementFormDefault='qualified'>

  <xs:element name='session' type='empty'/>

  <xs:simpleType name='empty'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value=''/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

B.2 jabber:iq:last

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='jabber:iq:last'
    xmlns='jabber:iq:last'
    elementFormDefault='qualified'>

  <xs:element name='query'>
    <xs:complexType>
      <xs:attribute name='seconds'
                    type='xs:unsignedLong'
                    use='optional'/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

B.3 jabber:iq:privacy

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

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

  <xs:element name='query'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='active'
                    minOccurs='0'
                    maxOccurs='1'/>
        <xs:element ref='default'
                    minOccurs='0'
                    maxOccurs='1'/>
        <xs:element ref='list'
                    minOccurs='0'
                    maxOccurs='unbounded'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name='active'>
    <xs:complexType>
      <xs:attribute name='name'
                    type='xs:string'
                    use='optional'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='default'>
    <xs:complexType>
      <xs:attribute name='name'
                    type='xs:string'
                    use='optional'/>
    </xs:complexType>
  </xs:element>

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

```
      </xs:complexType>
    </xs:element>

    <xs:element name='item'>
      <xs:complexType>
        <xs:sequence>
          <xs:element ref='iq'
                      minOccurs='0'
                      maxOccurs='1'/>
          <xs:element ref='message'
                      minOccurs='0'
                      maxOccurs='1'/>
          <xs:element ref='presence-in'
                      minOccurs='0'
                      maxOccurs='1'/>
          <xs:element ref='presence-out'
                      minOccurs='0'
                      maxOccurs='1'/>
        </xs:sequence>
        <xs:attribute name='order'
                      type='xs:unsignedInt'
                      use='required'/>
        <xs:attribute name='value'
                      type='xs:string'
                      use='optional'/>
        <xs:attribute name='action' use='required'>
          <xs:simpleType>
            <xs:restriction base='xs:NCName'>
              <xs:enumeration value='allow'/>
              <xs:enumeration value='deny'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name='type' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:NCName'>
              <xs:enumeration value='group'/>
              <xs:enumeration value='jid'/>
              <xs:enumeration value='subscription'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
```

```
    </xs:element>

    <xs:element name='iq' type='empty'/>
    <xs:element name='message' type='empty'/>
    <xs:element name='presence-in' type='empty'/>
```



```
    <xs:element name='presence-out' type='empty'/>

    <xs:simpleType name='empty'>
      <xs:restriction base='xs:string'>
        <xs:enumeration value=''/>
      </xs:restriction>
    </xs:simpleType>

  </xs:schema>
```

B.4 jabber:iq:roster

```
  <?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:complexType>
    </xs:element>

    <xs:element name='item'>
      <xs:complexType>
        <xs:sequence>
          <xs:element ref='group'
                      minOccurs='0'
                      maxOccurs='unbounded'/>
```

```
        </xs:sequence>
        <xs:attribute name='jid' type='xs:string' use='required'/>
        <xs:attribute name='name' type='xs:string' use='optional'/>
        <xs:attribute name='subscription' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:NCNAME'>
              <xs:enumeration value='to'/>
              <xs:enumeration value='from'/>
              <xs:enumeration value='both'/>
              <xs:enumeration value='none'/>
              <xs:enumeration value='remove'/>
            </xs:restriction>
```



```
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name='ask' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:NCNAME'>
              <xs:enumeration value='subscribe'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

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

  </xs:schema>
```

Appendix C. Revision History

Note to RFC Editor: please remove this entire appendix, and the corresponding entries in the table of contents, prior to publication.

C.1 Changes from draft-ietf-xmpp-im-14

- Added subscription state charts.
- Fixed several typographical errors in the privacy rules examples.
- Changed datatype of 'order' attribute in privacy rules from nonNegativeInteger to unsignedInt.

C.2 Changes from draft-ietf-xmpp-im-13

- Made one small change to privacy list syntax rules.

C.3 Changes from draft-ietf-xmpp-im-12

- Clarified meaning of the default message type as well as handling of unknown or unsupported types.
- Made several small editorial changes.

C.4 Changes from draft-ietf-xmpp-im-11

- Further clarified subscription syntax and semantics.
- Further clarified presence responsibilities for clients and servers.
- Added 'xml:lang' example to presence status.
- Added subsection on presence priority.
- Defined server handling of unsolicited presence stanzas of type "subscribed".
- Specified default resource priority if not provided.
- Corrected several errors in the schemas.
- Added privacy list business rule regarding roster changes.

- Removed the 'jabber:iq:privacy:error' namespace (not necessary).
- Documented message type='normal'.
- Made numerous small editorial changes throughout.

C.5 Changes from draft-ietf-xmpp-im-10

- Clarified presence responsibilities for servers and clients.
- Clarified the routing and delivery rules for servers.
- Made the 'xml:lang' examples more complete.

- Corrected several errors in the unsubscribe workflow.
- Made small editorial changes in several sections.

C.6 Changes from draft-ietf-xmpp-im-09

- Clarified rules regarding allowable JID types in rosters.
- Further clarified the semantics and routing implications of presence priorities.
- Removed several obsolete subsections.

C.7 Changes from draft-ietf-xmpp-im-08

- Removed authorization content (now addressed in XMPP Core).
- Added protocol for initiating an IM session, including schema and IANA registration template.
- Corrected <*-condition/> elements to be <condition/>.
- Made small editorial changes to address RFC Editor requirements.

C.8 Changes from draft-ietf-xmpp-im-07

- Added several error cases for resource authorization and updated relevant schema.

C.9 Changes from draft-ietf-xmpp-im-06

- Specified that IQ result stanzas are required in response to roster pushes.
- Changed stanza error namespace names to conform to the format defined in "The IETF XML Registry" as specified in XMPP Core.

- Removed note to RFC Editor regarding provisional namespace names.

C.10 Changes from draft-ietf-xmpp-im-05

- Removed use of ask='unsubscribe' per list discussion.
- Clarified handling of resource conflict during authorization.
- Added schemas for jabber:iq:auth, jabber:iq:auth:error, and jabber:iq:privacy:error.
- Corrected several small protocol errors in the examples.
- Clarified semantics of message types.

C.11 Changes from draft-ietf-xmpp-im-04

- Specified sending of unavailable presence after unsubscribe and subscription-cancellation actions.
- Further specified syntax and business rules for privacy lists.
- Brought error codes into line with definitions in draft-ietf-xmpp-core.
- Added note to RFC Editor regarding provisional namespace names.
- Removed vCard content and DTD, instead pointing to JSF documentation.

C.12 Changes from draft-ietf-xmpp-im-03

- Fixed order processing on privacy rules per list discussion.
- Made numerous small editorial changes.

C.13 Changes from draft-ietf-xmpp-im-02

- Added a great deal more detail to the narrative regarding server-side privacy rules as well as the interaction between rosters and subscriptions.

- Removed DTDs in favor of schemas (with the exception of vCard XML).

- Removed non-normative documentation of authentication using jabber:iq:auth and of in-band registration using jabber:iq:register, since these are maintained by the Jabber Software Foundation and are not part of the XMPP specification.

C.14 Changes from draft-ietf-xmpp-im-01

- Made numerous small editorial changes.

C.15 Changes from draft-ietf-xmpp-im-00

- Moved registration and authentication via jabber:iq:auth to non-normative appendices.

- Changed initial presence stanza from MUST be empty to SHOULD be empty.

- Specified that user or clients should not send presence stanzas of type='probe'.

- Specified the algorithm for digest passwords.

C.16 Changes from draft-miller-xmpp-im-02

- Added information about the 'jabber:iq:last' protocol to meet the requirement defined in section 3.2.4 of RFC 2779.

- Added information about the 'jabber:iq:privacy' protocol to meet the requirement defined in section 2.3.5 of RFC 2779.

- Added information about the vCard XML protocol to meet the requirement defined in sections 3.1.3 and 4.1.4 of RFC 2779.

- Changed the material describing authentication (but not resource authorization) with 'jabber:iq:auth' to non-normative.

- Noted that the only watchers are subscribers.

- Nomenclature changes: (1) from "chunks" to "stanzas"; (2) from "host" to "server"; (3) from "node" to "client" or "user" (as appropriate).

Intellectual Property Statement

Full Copyright Statement

English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assignees.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION

HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.