

Network Working Group  
Request for Comments: 3921  
Category: Standards Track

P. Saint-Andre, Ed.  
Jabber Software Foundation  
October 2004

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

Status of this Memo

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

Copyright Notice

Copyright (C) The Internet Society (2004).

Abstract

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

## Table of Contents

1.	Introduction . . . . .	2
2.	Syntax of XML Stanzas . . . . .	4
3.	Session Establishment . . . . .	10
4.	Exchanging Messages . . . . .	13
5.	Exchanging Presence Information . . . . .	16
6.	Managing Subscriptions . . . . .	26
7.	Roster Management . . . . .	27
8.	Integration of Roster Items and Presence Subscriptions . . . .	32
9.	Subscription States . . . . .	56
10.	Blocking Communication . . . . .	62
11.	Server Rules for Handling XML Stanzas . . . . .	85
12.	IM and Presence Compliance Requirements . . . . .	88
13.	Internationalization Considerations . . . . .	89
14.	Security Considerations . . . . .	89
15.	IANA Considerations . . . . .	90
16.	References . . . . .	91
A.	vCards . . . . .	93
B.	XML Schemas . . . . .	93
C.	Differences Between Jabber IM/Presence Protocols and XMPP. .	105
	Contributors . . . . .	106
	Acknowledgements . . . . .	106
	Author's Address . . . . .	106
	Full Copyright Statement . . . . .	107

## 1. Introduction

## 1.1. Overview

The Extensible Messaging and Presence Protocol (XMPP) is a protocol for streaming XML [XML] elements in order to exchange messages and presence information in close to real time. The core features of XMPP are defined in Extensible Messaging and Presence Protocol (XMPP): Core [XMPP-CORE]. These features -- mainly XML streams, use of TLS and SASL, 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 [XML-NAMES]. This memo describes extensions to and applications of the core features of XMPP that provide the basic functionality expected of an instant messaging (IM) and presence application as defined in RFC 2779 [IMP-REQS].

## 1.2. Requirements

For the purposes of this memo, the requirements of a basic instant messaging and presence application are defined by [IMP-REQS], which at a high level stipulates that a user must be able to complete the following use cases:

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

Detailed definitions of these functionality areas are contained in [IMP-REQS], and the interested reader is directed to that document regarding the requirements addressed herein.

[IMP-REQS] also stipulates that presence services must be separable from instant messaging services; i.e., it must be possible to use the protocol to provide a presence service, an instant messaging service, or both. Although the text of this memo assumes that implementations and deployments will want to offer a unified instant messaging and presence service, there is no requirement that a service must offer both a presence service and an instant messaging service, and the protocol makes it possible to offer separate and distinct services for presence and for instant messaging.

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

## 1.3. Terminology

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

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 BCP 14, RFC 2119 [TERMS].

## 2. Syntax of XML Stanzas

The basic semantics and common attributes of XML stanzas qualified by the 'jabber:client' and 'jabber:server' namespaces are defined in [XMPP-CORE]. However, these namespaces also define various child elements, as well as values for the common 'type' attribute, that are specific to instant messaging and presence applications. Thus, before addressing particular "use cases" for such applications, we here further describe the syntax of XML stanzas, thereby supplementing the discussion in [XMPP-CORE].

### 2.1. Message Syntax

Message stanzas qualified by the 'jabber:client' or 'jabber:server' namespace are used to "push" information to another entity. Common uses in instant messaging applications include single messages, messages sent in the context of a chat conversation, messages sent in the context of a multi-user chat room, headlines and other alerts, and errors.

#### 2.1.1. Types of Message

The 'type' attribute of a message stanza is RECOMMENDED; if included, it specifies the conversational context of the message, thus providing a hint regarding presentation (e.g., in a GUI). If included, the 'type' attribute MUST have one of the following values:

- o chat -- The message is sent in the context of a one-to-one chat conversation. A compliant client SHOULD present the message in an interface enabling one-to-one chat between the two parties, including an appropriate conversation history.
- o error -- An error has occurred related to a previous message sent by the sender (for details regarding stanza error syntax, refer to [XMPP-CORE]). A compliant client SHOULD present an appropriate interface informing the sender of the nature of the error.
- o groupchat -- The message is sent in the context of a multi-user chat environment (similar to that of [IRC]). A compliant client SHOULD present the message in an interface enabling many-to-many chat between the parties, including a roster of parties in the chatroom and an appropriate conversation history. Full definition of XMPP-based groupchat protocols is out of scope for this memo.
- o 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 the message in 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).

- o normal -- The message is a single message that is sent outside the context of a one-to-one conversation or groupchat, and to which it is expected that the recipient will reply. A compliant client SHOULD present the message in an interface enabling the recipient to reply, but without a conversation history.

An IM application SHOULD support all of the foregoing message types; if an application receives a message with no 'type' attribute or the application does not understand the value of the 'type' attribute provided, it MUST consider the message to be of type "normal" (i.e., "normal" is the default). The "error" type MUST be generated only in response to an error related to a message received from another entity.

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

#### 2.1.2. Child Elements

As described under extended namespaces (Section 2.4), a message stanza MAY contain any properly-namespaced child element.

In accordance with the default namespace declaration, by default a message stanza is qualified by the 'jabber:client' or 'jabber:server' namespace, which defines certain allowable children of message stanzas. If the message stanza is of type "error", it MUST include an <error/> child; for details, see [XMPP-CORE]. Otherwise, the message stanza MAY contain any of the following child elements without an explicit namespace declaration:

1. <subject/>
2. <body/>
3. <thread/>

##### 2.1.2.1. Subject

The <subject/> element contains human-readable XML character data that specifies the topic of the message. The <subject/> element MUST NOT possess any attributes, with the exception of the 'xml:lang' attribute. Multiple instances of the <subject/> element MAY be included for the purpose of providing alternate versions of the same

subject, but only if each instance possesses an 'xml:lang' attribute with a distinct language value. The <subject/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

#### 2.1.2.2. Body

The <body/> element contains human-readable XML character data that specifies the textual contents of the message; this child element is normally included but is OPTIONAL. The <body/> element MUST NOT possess any attributes, with the exception of the 'xml:lang' attribute. Multiple instances of the <body/> element MAY be included but only if each instance possesses an 'xml:lang' attribute with a distinct language value. The <body/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

#### 2.1.2.3. Thread

The <thread/> element contains non-human-readable XML character data specifying an identifier that is used for tracking a conversation thread (sometimes referred to as an "instant messaging session") between two entities. The value of the <thread/> element is generated by the sender and SHOULD be copied back in any replies. If used, it MUST be unique to that conversation thread within the stream and MUST be consistent throughout that conversation (a client that receives a message from the same full JID but with a different thread ID MUST assume that the message in question exists outside the context of the existing conversation thread). The use of the <thread/> element is OPTIONAL and is not used to identify individual messages, only conversations. A message stanza MUST NOT contain more than one <thread/> element. The <thread/> element MUST NOT possess any attributes. The value of the <thread/> element MUST be treated as opaque by entities; no semantic meaning may be derived from it, and only exact comparisons may be made against it. The <thread/> element MUST NOT contain mixed content (as defined in Section 3.2.2 of [XML]).

### 2.2. Presence Syntax

Presence stanzas are used qualified by the 'jabber:client' or 'jabber:server' namespace to express an entity's current network availability (offline or online, along with various sub-states of the latter and optional user-defined descriptive text), and to notify other entities of that availability. Presence stanzas are also used to negotiate and manage subscriptions to the presence of other entities.

### 2.2.1. Types of Presence

The 'type' attribute of a presence stanza is OPTIONAL. A presence stanza that does not possess a 'type' attribute is used to signal to the server that the sender is online and available for communication. If included, the 'type' attribute specifies a lack of availability, a request to manage a subscription to another entity's presence, a request for another entity's current presence, or an error related to a previously-sent presence stanza. If included, the 'type' attribute MUST have one of the following values:

- o unavailable -- Signals that the entity is no longer available for communication.
- o subscribe -- The sender wishes to subscribe to the recipient's presence.
- o subscribed -- The sender has allowed the recipient to receive their presence.
- o unsubscribe -- The sender is unsubscribing from another entity's presence.
- o unsubscribed -- The subscription request has been denied or a previously-granted subscription has been cancelled.
- o probe -- A request for an entity's current presence; SHOULD be generated only by a server on behalf of a user.
- o error -- An error has occurred regarding processing or delivery of a previously-sent presence stanza.

For detailed information regarding presence semantics and the subscription model used in the context of XMPP-based instant messaging and presence applications, refer to Exchanging Presence Information (Section 5) and Managing Subscriptions (Section 6).

### 2.2.2. Child Elements

As described under extended namespaces (Section 2.4), a presence stanza MAY contain any properly-namespaced child element.

In accordance with the default namespace declaration, by default a presence stanza is qualified by the 'jabber:client' or 'jabber:server' namespace, which defines certain allowable children of presence stanzas. If the presence stanza is of type "error", it MUST include an <error/> child; for details, see [XMPP-CORE]. If the presence stanza possesses no 'type' attribute, it MAY contain any of

the following child elements (note that the <status/> child MAY be sent in a presence stanza of type "unavailable" or, for historical reasons, "subscribe"):

1. <show/>
2. <status/>
3. <priority/>

#### 2.2.2.1. Show

The OPTIONAL <show/> element contains non-human-readable XML character data that specifies the particular availability status of an entity or specific resource. A presence stanza MUST NOT contain more than one <show/> element. The <show/> element MUST NOT possess any attributes. If provided, the XML character data value MUST be one of the following (additional availability types could be defined through a properly-namespaced child element of the presence stanza):

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

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

#### 2.2.2.2. Status

The OPTIONAL <status/> element contains XML character data specifying a natural-language description of availability status. It is normally used in conjunction with the show element to provide a detailed description of an availability state (e.g., "In a meeting"). The <status/> element MUST NOT possess any attributes, with the exception of the 'xml:lang' attribute. Multiple instances of the <status/> element MAY be included but only if each instance possesses an 'xml:lang' attribute with a distinct language value.

#### 2.2.2.3. Priority

The OPTIONAL <priority/> element contains non-human-readable XML character data that specifies the priority level of the resource. The value MUST be an integer between -128 and +127. A presence stanza MUST NOT contain more than one <priority/> element. The <priority/> element MUST NOT possess any attributes. If no priority is provided,



a server SHOULD consider the priority to be zero. For information regarding the semantics of priority values in stanza routing within instant messaging and presence applications, refer to Server Rules for Handling XML Stanzas (Section 11).

### 2.3. IQ Syntax

IQ stanzas provide a structured request-response mechanism. The basic semantics of that mechanism (e.g., that the 'id' attribute is REQUIRED) are defined in [XMPP-CORE], whereas the specific semantics required to complete particular use cases are defined in all cases by an extended namespace (Section 2.4) (note that the 'jabber:client' and 'jabber:server' namespaces do not define any children of IQ stanzas other than the common <error/>). This memo defines two such extended namespaces, one for Roster Management (Section 7) and the other for Blocking Communication (Section 10); however, an IQ stanza MAY contain structured information qualified by any extended namespace.

### 2.4. Extended Namespaces

While the three XML stanza kinds defined in the "jabber:client" or "jabber:server" namespace (along with their attributes and child elements) provide a basic level of functionality for messaging and presence, XMPP uses XML namespaces to extend the stanzas for the purpose of providing additional functionality. Thus a message or presence stanza MAY contain one or more optional child elements specifying content that extends the meaning of the message (e.g., an XHTML-formatted version of the message body), and an IQ stanza MAY contain one such child element. This child element MAY have any name and MUST possess an 'xmlns' namespace declaration (other than "jabber:client", "jabber:server", or "http://etherx.jabber.org/streams") that defines all data contained within the child element.

Support for any given extended namespace is OPTIONAL on the part of any implementation (aside from the extended namespaces defined herein). If an entity does not understand such a namespace, the entity's expected behavior depends on whether the entity is (1) the recipient or (2) an entity that is routing the stanza to the recipient:

Recipient: If a recipient receives a stanza that contains a child element it does not understand, it SHOULD ignore that specific XML data, i.e., it SHOULD not process it or present it to a user or associated application (if any). In particular:

- \* If an entity receives a message or presence stanza that contains XML data qualified by a namespace it does not understand, the portion of the stanza that is in the unknown namespace SHOULD be ignored.
- \* If an entity receives a message stanza whose only child element is qualified by a namespace it does not understand, it MUST ignore the entire stanza.
- \* If an entity receives an IQ stanza of type "get" or "set" containing a child element qualified by a namespace it does not understand, the entity SHOULD return an IQ stanza of type "error" with an error condition of <service-unavailable/>.

Router: If a routing entity (usually a server) handles a stanza that contains a child element it does not understand, it SHOULD ignore the associated XML data by passing it on untouched to the recipient.

### 3. Session Establishment

Most instant messaging and presence applications based on XMPP are implemented via a client-server architecture that requires a client 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 client can establish an instant messaging and presence session. These are:

1. Stream Authentication -- a client MUST complete stream authentication as documented in [XMPP-CORE] before attempting to establish a session or send any XML stanzas.
2. Resource Binding -- after completing stream authentication, a client MUST bind a resource to the stream so that the client's address is of the form <user@domain/resource>, after which the entity is now said to be a "connected resource" in the terminology of [XMPP-CORE].

If a server supports sessions, it MUST include a <session/> element qualified by the 'urn:ietf:params:xml:ns:xmpp-session' namespace in the stream features it advertises to a client after the completion of stream authentication as defined in [XMPP-CORE]:

Server advertises session establishment feature to client:

```
<stream:stream
  xmlns='jabber:client'
  xmlns:stream='http://etherx.jabber.org/streams'
  id='c2s_345'
  from='example.com'
  version='1.0'>
<stream:features>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'/>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session'/>
</stream:features>
```

Upon being so informed that session establishment is required (and after completing resource binding), the client **MUST** establish a session if it desires to engage in instant messaging and presence functionality; it completes this step by sending to the server an IQ stanza of type "set" containing an empty <session/> child element qualified by the 'urn:ietf:params:xml:ns:xmpp-session' namespace:

Step 1: Client requests session with server:

```
<iq to='example.com'
  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 from='example.com'
  type='result'
  id='sess_1'/>
```

Upon establishing a session, a connected resource (in the terminology of [XMPP-CORE]) is said to be an "active resource".

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 resource associated with a resource identifier 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 from='example.com' 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 resource is not allowed to create a session, the server MUST return an error (e.g., forbidden).

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

```
<iq from='example.com' type='error' id='sess_1'>
  <session xmlns='urn:ietf:params:xml:ns:xmpp-session' />
  <error type='auth'>
    <forbidden
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

If there is already an active resource of the same name, the server MUST either (1) terminate the active resource and allow the newly-requested session, or (2) disallow the newly-requested session and maintain the active resource. Which of these the server does is up to the implementation, although it is RECOMMENDED to implement case #1. In case #1, the server SHOULD send a <conflict/> stream error to the active resource, terminate the XML stream and underlying TCP connection for the active resource, and return a IQ stanza of type "result" (indicating success) to the newly-requested session. In case #2, the server SHOULD send a <conflict/> stanza error to the newly-requested session but maintain the XML stream for that connection so that the newly-requested session has an opportunity to negotiate a non-conflicting resource identifier before sending another request for session establishment.

Step 2 (alt): Server informs existing active resource of resource conflict (case #1):

```
<stream:error>
  <conflict xmlns='urn:ietf:params:xml:ns:xmpp-streams' />
</stream:error>
</stream:stream>
```

Step 2 (alt): Server informs newly-requested session of resource conflict (case #2):

```
<iq from='example.com' 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 OPTIONAL.

Note: Before allowing the creation of instant messaging and presence sessions, a server MAY require prior account provisioning. Possible 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 out of scope for this memo, but is documented in [JEP-0077], published by the Jabber Software Foundation [JSF].

## 4. Exchanging Messages

Exchanging messages is a basic use of XMPP and is brought about when a user generates a message stanza that is addressed to another entity. As defined under Server Rules for Handling XML Stanzas (Section 11), 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 Message Syntax (Section 2.1).

### 4.1. 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@domain/resource> (e.g., within the context of a chat session), the value of the 'to' address SHOULD be of the form <user@domain/resource> rather than of the form <user@domain> 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@domain> rather than of the form <user@domain/resource>.

#### 4.2. Specifying a Message Type

As noted, it is RECOMMENDED for a message stanza to possess a 'type' attribute whose value captures the conversational context (if any) of the message (see Type (Section 2.1.1)).

The following example shows a valid value of the 'type' attribute:

Example: A message of a defined type:

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

#### 4.3. Specifying a Message Body

A message stanza MAY (and often will) contain a child <body/> element whose XML character data specifies the primary meaning of the message (see Body (Section 2.1.2.2)).

Example: A message with a body:

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

#### 4.4. Specifying a Message Subject

A message stanza MAY contain one or more child <subject/> elements specifying the topic of the message (see Subject (Section 2.1.2.1)).

Example: A message with a subject:

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

#### 4.5. 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 (see Thread (Section 2.1.2.3)).

Example: A threaded conversation:

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

## 5. 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 by including a 'to' address, normally presence notifications (i.e., presence stanzas with no 'type' or of type "unavailable" and with no 'to' address) are sent from a client to its server and then broadcasted by the server to any entities that are subscribed to the presence of the sending entity (in the terminology of RFC 2778 [IMP-MODEL], these entities are subscribers). This broadcast model does not apply to subscription-related presence stanzas or presence stanzas of type "error", but to presence notifications only as defined above. (Note: While presence information MAY be provided on a user's behalf by an automated service, normally it is provided by the user's client.)

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

### 5.1. Client and Server Presence Responsibilities

#### 5.1.1. Initial Presence

After establishing a session, a client SHOULD 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 broadcasted by the server on behalf of the client) and (2) MUST possess no 'type' attribute (signalling the user's availability). After sending initial presence, an active resource is said to be an "available resource".

Upon receiving initial presence from a client, the user's server MUST do the following if there is not already one or more available resources for the user (if there is already one or more available resources for the user, the server obviously does not need to send the presence probes, since it already possesses the requisite information):

1. Send presence probes (i.e., presence stanzas whose 'type' attribute is set to a value of "probe") from the full JID (e.g., <user@example.com/resource>) of the user to all contacts to which the user is subscribed in order to determine if they are available; such contacts are those for which a JID is present in the user's roster with the 'subscription' attribute set to a value of "to" or "both". (Note: The user's server MUST NOT send presence probes to contacts from which the user is blocking



inbound presence notifications, as described under Blocking Inbound Presence Notifications (Section 10.10).)

2. Broadcast initial presence from the full JID (e.g., <user@example.com/resource>) of the user to all contacts that are subscribed to the user's presence information; such contacts are those for which a JID is present in the user's roster with the 'subscription' attribute set to a value of "from" or "both". (Note: The user's server MUST NOT broadcast initial presence to contacts to which the user is blocking outbound presence notifications, as described under Blocking Outbound Presence Notifications (Section 10.11).)

In addition, the user's server MUST broadcast initial presence from the user's new available resource to any of the user's existing available resources (if any).

Upon receiving initial presence from the user, the contact's server MUST deliver the user's presence stanza to the full JIDs (<contact@example.org/resource>) associated with all of the contact's available resources, 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 under Blocking Inbound Presence Notifications (Section 10.10)).

If the user's server receives a presence stanza of type "error" in response to the initial presence that it sent 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 stanza from the contact).

#### 5.1.2. Presence Broadcast

After sending initial presence, the user MAY update its presence information for broadcasting at any time during its 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 notifications, and (3) from whom the server has not received a presence error during the user's session (as well as to any of the user's other available resources).

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 entities that fit the above description, 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).

#### 5.1.3. Presence Probes

Upon receiving a presence probe from the user, the contact's server SHOULD reply as follows:

1. If the user is not in the contact's roster with a subscription state of "From", "From + Pending Out", or "Both" (as defined under Subscription States (Section 9)), the contact's server MUST return a presence stanza of type "error" in response to the presence probe (however, if a 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). Specifically:
  - \* if the user is in the contact's roster with a subscription state of "None", "None + Pending Out", or "To" (or is not in the contact's roster at all), the contact's server MUST return a <forbidden/> stanza error in response to the presence probe.
  - \* if the user is in the contact's roster with a subscription state of "None + Pending In", "None + Pending Out/In", or "To + Pending In", the contact's server MUST return a <not-authorized/> stanza error in response to the presence probe.
2. Else, if the contact is blocking presence notifications to the user's bare JID or full JID (using either a default list or active list as defined under Blocking Outbound Presence Notifications (Section 10.11)), the server MUST NOT reply to the presence probe.
3. Else, if the contact has no available resources, the server MUST either (1) reply to the presence probe by sending to the user the full XML of the last presence stanza of type "unavailable" received by the server from the contact, or (2) not reply at all.

4. Else, if the contact has at least one available resource, the server MUST reply to the presence probe by sending to the user the full XML of the last presence stanza with no 'to' attribute received by the server from each of the contact's available resources (again, subject to privacy lists in force for each session).

#### 5.1.4. Directed Presence

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 lists) 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 available 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 (i.e., the resource is active but not available), 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.

#### 5.1.5. Unavailable Presence

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 or may be timed out by the server. If one of the user's resources becomes unavailable for any reason (either gracefully or ungracefully), the user's server 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 of the user's other available resources, as well as 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.

#### 5.1.6. Presence Subscriptions

A subscription request is a presence stanza whose 'type' attribute has a value of "subscribe". If the subscription request is being sent to an instant messaging contact, the JID supplied in the 'to' attribute SHOULD be of the form <contact@example.org> rather than <contact@example.org/resource>, since the desired result is normally for the user to receive presence from all of the contact's resources, not merely the particular resource specified in the 'to' attribute.

A user's server MUST NOT automatically approve subscription requests on the user's behalf. All subscription requests MUST be directed to the user's client, specifically to one or more available resources associated with the user. If there is no available resource associated with the user when the subscription request is received by the user's server, the user's server MUST keep a record of the subscription request and deliver the request when the user next creates an available resource, until the user either approves or denies the request. If there is more than one available resource associated with the user when the subscription request is received by the user's server, the user's server MUST broadcast that subscription request to all available resources in accordance with Server Rules for Handling XML Stanzas (Section 11). (Note: If an active resource

has not provided initial presence, the server MUST NOT consider it to be available and therefore MUST NOT send subscription requests to it.) However, if the user receives a presence stanza of type "subscribe" from a contact to whom the user has already granted permission to see the user's presence information (e.g., in cases when the contact is seeking to resynchronize subscription states), the user's server SHOULD auto-reply on behalf of the user. In addition, the user's server MAY choose to re-send an unapproved pending subscription request to the contact based on an implementation-specific algorithm (e.g., whenever a new resource becomes available for the user, or after a certain amount of time has elapsed); this helps to recover from transient, silent errors that may have occurred in relation to the original subscription request.

## 5.2. Specifying Availability Status

A client MAY provide further information about its availability status by using the <show/> element (see Show (Section 2.2.2.1)).

Example: Availability status:

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

## 5.3. Specifying Detailed Status Information

In conjunction with the <show/> element, a client MAY provide detailed status information by using the <status/> element (see Status (Section 2.2.2.2)).

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>
```

#### 5.4. Specifying Presence Priority

A client MAY provide a priority for its resource by using the `<priority/>` element (see Priority (Section 2.2.2.3)).

Example: Presence priority:

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

#### 5.5. Presence Examples

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

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

Example 1: User sends initial presence:

```
<presence/>
```

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

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

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

Example 4: Contacts' servers reply to presence probe on behalf of all 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: Contacts' servers deliver user's initial presence to all available resources or return 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@example.org'
  to='romeo@example.net/orchard'>
  <error type='cancel'>
    <gone 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: User's server broadcasts updated presence information 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'
  xml:lang='en'>
  <show>away</show>
  <status>I shall return!</status>
  <priority>1</priority>
</presence>
```



Example 9: Contact's server delivers updated presence information to all of the contact's available resources:

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

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

Example 10: One of the contact's resources broadcasts final presence:

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

Example 11: Contact's server sends unavailable presence information to user:

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

Example 12: User sends final presence:

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

Example 13: User's server broadcasts unavailable presence information to contact 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'
  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>
```

## 6. 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 8), and the reader must refer to that section for a complete understanding of presence subscriptions.

### 6.1. 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' />
```

For client and server responsibilities regarding presence subscription requests, refer to Presence Subscriptions (Section 5.1.6).

## 6.2. Handling a Subscription Request

When a client receives a subscription request from another entity, it MUST either approve the request by sending a presence stanza of type "subscribed" or refuse the request by sending a presence stanza of type "unsubscribed".

Example: Approving a subscription request:

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

Example: Refusing a presence subscription request:

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

## 6.3. 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'/>
```

## 6.4. 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'/>
```

## 7. Roster Management

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@domain>). 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 resource.

Note: There are important interactions between rosters and subscriptions; these are defined under Integration of Roster Items and Presence Subscriptions (Section 8), and the reader must refer to that section for a complete understanding of roster management.

### 7.1. Syntax and Semantics

Rosters are managed using IQ stanzas, specifically by means of a `<query/>` child element qualified by the `'jabber:iq:roster'` namespace. The `<query/>` element MAY contain one or more `<item/>` children, each describing a unique roster item or "contact".

The "key" or unique identifier for each roster item is a JID, encapsulated in the `'jid'` attribute of the `<item/>` element (which is REQUIRED). The value of the `'jid'` attribute SHOULD be of the form `<user@domain>` if the item is associated with another (human) instant messaging user.

The state of the presence subscription in relation to a roster item is captured in the `'subscription'` attribute of the `<item/>` element. Allowable values for this attribute are:

- o "none" -- the user does not have a subscription to the contact's presence information, and the contact does not have a subscription to the user's presence information
- o "to" -- the user has a subscription to the contact's presence information, but the contact does not have a subscription to the user's presence information
- o "from" -- the contact has a subscription to the user's presence information, but the user does not have a subscription to the contact's presence information
- o "both" -- both the user and the contact have subscriptions to each other's presence information

Each `<item/>` element MAY contain a `'name'` attribute, which sets the "nickname" to be associated with the JID, as determined by the user (not the contact). The value of the `'name'` attribute is opaque.

Each `<item/>` element MAY contain one or more `<group/>` child elements, for use in collecting roster items into various categories. The XML character data of the `<group/>` element is opaque.

## 7.2. Business Rules

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" (incoming IQ of type "set" containing a roster item) to ensure that it is from a trusted source; specifically, the stanza MUST either have no 'from' attribute (i.e., implicitly from the server) or have a 'from' attribute whose value matches the user's bare JID (of the form <user@domain>) or full JID (of the form <user@domain/resource>); otherwise, the client SHOULD ignore the "roster push".

## 7.3. Retrieving One's Roster on Login

Upon connecting to the server and becoming an active resource, a client SHOULD request the roster before sending initial presence (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 OPTIONAL). If an available resource does not request the roster during a session, the server MUST NOT send it presence subscriptions and associated roster updates.

Example: Client requests current roster from server:

```
<iq from='juliet@example.com/balcony' type='get' id='roster_1'>
  <query xmlns='jabber:iq:roster' />
</iq>
```

Example: Client receives roster from server:

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

```
</iq>
```

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

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 stanza of type "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 that have requested the roster and (2) replies with an IQ result to the sending resource:

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

```
<iq to='juliet@example.com/chamber'
    type='set'
    id='a78b4q6ha464'>
  <query xmlns='jabber:iq:roster'>
    <item jid='nurse@example.com'
          name='Nurse'
          subscription='none'>
      <group>Servants</group>
    </item>
  </query>
</iq>
```

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

```
<iq to='juliet@example.com/balcony' type='result' id='roster_2' />
```

As required by the semantics of the IQ stanza kind as defined in [XMPP-CORE], each resource that received the roster push MUST reply with an IQ stanza of type "result" (or "error").

Example: Resources reply with an IQ result to the server:

```
<iq from='juliet@example.com/balcony'
  to='example.com'
  type='result'
  id='a78b4q6ha463' />
<iq from='juliet@example.com/chamber'
  to='example.com'
  type='result'
  id='a78b4q6ha464' />
```

### 7.5. 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 from='juliet@example.com/chamber' 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.

## 7.6. Deleting a Roster Item

At any time, a user MAY delete an item from his or her roster by sending an IQ set to the server 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 from='juliet@example.com/balcony' 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 Removing a Roster Item and Cancelling All Subscriptions (Section 8.6).

## 8. Integration of Roster Items and Presence Subscriptions

### 8.1. 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 instant messaging applications.

There are four primary subscription states:

- o None -- the user does not have a subscription to the contact's presence information, and the contact does not have a subscription to the user's presence information



- o To -- the user has a subscription to the contact's presence information, but the contact does not have a subscription to the user's presence information
- o From -- the contact has a subscription to the user's presence information, but the user does not have a subscription to the contact's presence information
- o Both -- both the user and the contact have subscriptions to each other's presence information (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.

Narrative explanations of how these subscription states interact with roster items in order to complete certain defined use cases are provided in the following sub-sections. Full details regarding server and client handling of all subscription states (including pending states between the primary states listed above) is provided in Subscription States (Section 9).

The server **MUST NOT** send presence subscription requests or roster pushes to unavailable resources, nor to available resources that have not requested the roster.

The 'from' and 'to' addresses 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 the IQ semantics defined in [XMPP-CORE]).

## 8.2. User Subscribes to Contact

The process by which a user subscribes to a contact, including the interaction between roster items and subscription states, is described 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 sending an IQ stanza of type='set' containing a <query/> element qualified by 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='set1'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='contact@example.org'
      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 to the sending resource with an IQ result indicating the success of the roster set:

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

```
<iq type='result' id='set1' />
```

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

```
<presence to='contact@example.org' 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@example.org'
      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 on behalf of the user, then 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 shown above.

5. The user's server MUST also stamp the presence stanza of type "subscribe" with the user's bare JID (i.e., <user@example.com>) as the 'from' address (if the user provided a 'from' address set to the user's full JID, the server SHOULD remove the resource identifier). 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@example.com'
  to='contact@example.org'
  type='subscribe' />
```

Note: If the user's server receives a presence stanza of type "error" from the contact's server, it MUST deliver the error stanza to the user, whose client MAY determine that the error is in response to the outgoing presence stanza of type "subscribe" it sent previously (e.g., by tracking an 'id' attribute) and then choose to resend the "subscribe" request or revert the roster to its previous state by sending a presence stanza of type "unsubscribe" to the contact.

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 available resource from which the contact 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; normally this is done by adding a roster item for the contact to the user's roster, with a state of "None + Pending In" as defined under Subscription States (Section 9), however a server SHOULD NOT push or deliver roster items in that state to the contact). No matter when the subscription request is delivered, the contact must decide whether or not to approve it (subject to the contact's configured preferences, the contact's client MAY approve or refuse the subscription request without presenting it to the contact). Here we assume the "happy path" that the contact approves the subscription request (the alternate flow of declining the subscription request is defined in Section 8.2.1). In this case, the contact's client (1) SHOULD perform a roster set specifying the desired nickname and group for the user (if any); and (2) MUST send a presence stanza of type "subscribed" to the user in order to approve the subscription request.

```
<iq type='set' id='set2'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='user@example.com'
      name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>
```

```
<presence to='user@example.com' 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' (the server MUST send this even if the contact did not perform a roster set); (2) MUST return an IQ result to the sending resource indicating the success of the roster set; (3) MUST route the presence stanza of type "subscribed" to the user, first stamping the 'from' address as the bare JID (<contact@example.org>) of the contact; and (4) MUST send available presence from all of the contact's available resources to the user:

```
<iq type='set' to='contact@example.org/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='user@example.com'
      subscription='from'
      name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>

<iq type='result' to='contact@example.org/resource' id='set2' />

<presence
  from='contact@example.org'
  to='user@example.com'
  type='subscribed' />

<presence
  from='contact@example.org/resource'
  to='user@example.com' />
```

Note: If the contact's server receives a presence stanza of type "error" from the user's server, it MUST deliver the error stanza to the contact, whose client MAY determine that the error is in response to the outgoing presence stanza of type "subscribed" it sent previously (e.g., by tracking an 'id' attribute) and then choose to resend the "subscribed" notification or revert the roster to its previous state by sending a presence stanza of type "unsubscribed" to the user.

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' and ask='subscribe' or (b) subscription='from' and 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@example.com'
  from='contact@example.org'
  type='subscribed' />

<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='contact@example.org'
      subscription='to'
      name='MyContact'>
      <group>MyBuddies</group>
    </item>
  </query>
</iq>

<presence
  from='contact@example.org/resource'
  to='user@example.com/resource' />
```

9. Upon receiving the presence stanza of type "subscribed", the user SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the user's server know that it MUST no longer send notification of the subscription state change to the user (see Section 9.4).

From the perspective of the user, there now exists a subscription to the contact's presence information; from the perspective of the contact, there now exists a subscription from the user.

#### 8.2.1. Alternate Flow: Contact Declines Subscription Request

The above activity flow represents the "happy path" regarding the user's subscription request to the contact. The main alternate flow occurs if the contact refuses the user's subscription request, as described below.

1. If the contact wants to refuse 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 8.2):

```
<presence to='user@example.com' 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@example.org>) of the contact:

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

Note: If the contact's server previously added the user to the contact's roster for tracking purposes, it MUST remove the relevant item at this time.

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@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='contact@example.org'
      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 "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the user's server know that it **MUST** no longer send notification of the subscription state change to the user (see Section 9.4).

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.

### 8.3. Creating a Mutual Subscription

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

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

```
<presence to='user@example.com' 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@example.org>) of the contact:

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



```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='subscribe' />
```

Note: If the contact's server receives a presence stanza of type "error" from the user's server, it MUST deliver the error stanza to the contact, whose client MAY determine that the error is in response to the outgoing presence stanza of type "subscribe" it sent previously (e.g., by tracking an 'id' attribute) and then choose to resend the "subscribe" request or revert the roster to its previous state by sending a presence stanza of type "unsubscribe" to the user.

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 available resource for which the user 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 approve it (subject to the user's configured preferences, the user's client MAY approve or refuse the subscription request without presenting it to the user). Here we assume the "happy path" that the user approves the subscription request (the alternate flow of declining the subscription request is defined in Section 8.3.1). In this case, the user's client MUST send a presence stanza of type "subscribed" to the contact in order to approve the subscription request.

```
<presence to='contact@example.org' 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@example.com>) of the user; and (3) MUST send to the contact the full XML of the last presence stanza with no 'to' attribute received by the server from each of the user's available resources (subject to privacy lists in force for each session):

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

<presence
  from='user@example.com'
  to='contact@example.org'
  type='subscribed' />

<presence
  from='user@example.com/resource'
  to='contact@example.org' />
```

Note: If the user's server receives a presence stanza of type "error" from the contact's server, it MUST deliver the error stanza to the user, whose client MAY determine that the error is in response to the outgoing presence stanza of type "subscribed" it sent previously (e.g., by tracking an 'id' attribute) and then choose to resend the subscription request or revert the roster to its previous state by sending a presence stanza of type "unsubscribed" to the contact.

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' and ask='subscribe' or (b) subscription='from' and 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@example.com'
  to='contact@example.org'
  type='subscribed' />

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

<presence
  from='user@example.com/resource'
  to='contact@example.org/resource' />
```

6. Upon receiving the presence stanza of type "subscribed", the contact SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the contact's server know that it MUST no longer send notification of the subscription state change to the contact (see Section 9.4).

The user and the contact now have a mutual subscription to each other's presence -- i.e., the subscription is of type "both".

#### 8.3.1. Alternate Flow: User Declines Subscription Request

The above activity flow represents the "happy path" regarding the contact's subscription request to the user. The main alternate flow occurs if the user refuses the contact's subscription request, as described below.

1. If the user wants to refuse 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 8.3):

```
<presence to='contact@example.org' 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@example.com>) of the user:

```
<presence
  from='user@example.com'
  to='contact@example.org'
  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@example.com'
  to='contact@example.org'
  type='unsubscribed'/>
```

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='user@example.com'
      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 "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the contact's server know that it MUST no longer send notification of the subscription state change to the contact (see Section 9.4).

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".

## 8.4. Unsubscribing

At any time after subscribing to a contact's presence information, 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" is sent. Both possible scenarios are described below.

### 8.4.1. Case #1: Unsubscribing When Subscription is Not Mutual

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

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

```
<presence to='contact@example.org' 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@example.com>) of the user:

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

```
<presence
  from='user@example.com'
  to='contact@example.org'
  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 unavailable or has not requested the roster, 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@example.com'
      subscription='none'
      name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>
```

```
<presence
  from='user@example.com'
  to='contact@example.org'
  type='unsubscribe' />
```

4. Upon receiving the presence stanza of type "unsubscribe", the contact SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the contact's server know that it MUST no longer send notification of the subscription state change to the contact (see Section 9.4).
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 all of the contact's available resources to the user:

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  type='unavailable' />
```

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

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  type='unavailable' />
```

7. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the user's server know that it MUST no longer send notification of the subscription state change to the user (see Section 9.4).

#### 8.4.2. Case #2: Unsubscribing When Subscription is Mutual

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

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

```
<presence to='contact@example.org' 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@example.com>) of the user:

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

```
<presence
  from='user@example.com'
  to='contact@example.org'
  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 unavailable or has not requested the roster, 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@example.com'
      subscription='to'
      name='SomeUser'>
      <group>SomeGroup</group>
    </item>
  </query>
</iq>
```

```
<presence
  from='user@example.com'
  to='contact@example.org'
  type='unsubscribe' />
```

4. Upon receiving the presence stanza of type "unsubscribe", the contact SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the contact's server know that it MUST no longer send notification of the subscription state change to the contact (see Section 9.4).

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 all of the contact's available resources to the user:

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  type='unavailable' />
```

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

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  type='unavailable' />
```

7. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the user's server know that it MUST no longer send notification of the subscription state change to the user (see Section 9.4).

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 information. 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 under Removing a Roster Item and Cancelling All Subscriptions (Section 8.6).

### 8.5. 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 described below.

#### 8.5.1. Case #1: Cancelling When Subscription is Not Mutual

In the first case, the user has a subscription to the contact's presence information but the contact does not have a subscription to the user's presence information (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@example.com' 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@example.org>`) of the contact; and (3) SHOULD send unavailable presence from all of the contact's available resources to the user:

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

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  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 unavailable or has not requested the roster, 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 all of the user's available resources; and (3) MUST deliver the unavailable presence to all of the user's available resources:

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

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  type='unavailable' />
```

4. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the user's server know that it MUST no longer send notification of the subscription state change to the user (see Section 9.4).

#### 8.5.2. Case #2: Cancelling When Subscription is Mutual

In the second case, the user has a subscription to the contact's presence information and the contact also has a subscription to the user's presence information (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@example.com' 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@example.org>) of the contact; and (3) SHOULD send unavailable presence from all of the contact's available resources to all of the user's available resources:

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

<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />

<presence
  from='contact@example.org/resource'
  to='user@example.com'
  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 unavailable or has not requested the roster, 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 all of the user's available resources; and (3) MUST deliver the unavailable presence to all of the user's available resources:

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

```
<presence
  from='contact@example.org'
  to='user@example.com'
  type='unsubscribed' />
```

```
<presence
  from='contact@example.org/resource'
  to='user@example.com'
  type='unavailable' />
```

4. Upon receiving the presence stanza of type "unsubscribed", the user SHOULD acknowledge receipt of that subscription state notification through either "affirming" 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 does not necessarily affect the subscription state (see Subscription States (Section 9) for details), but instead lets the user's server know that it MUST no longer send notification of the subscription state change to the user (see Section 9.4).

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 information. 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 under Removing a Roster Item and Cancelling All Subscriptions (Section 8.6).

#### 8.6. 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, the roster management protocol 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@example.org'
      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 that have requested the roster of the roster item removal; (3) MUST inform the resource that initiated the removal of success; and (4) SHOULD send unavailable presence from all of the user's available resources to the contact:

```
<presence
  from='user@example.com'
  to='contact@example.org'
  type='unsubscribe' />

<presence
  from='user@example.com'
  to='contact@example.org'
  type='unsubscribed' />
```

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
      jid='contact@example.org'
      subscription='remove' />
  </query>
</iq>

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

<presence
  from='user@example.com/resource'
  to='contact@example.org'
  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 unavailable or has not requested the roster, 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 all of the contact's available resources:

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

<presence
  from='user@example.com'
  to='contact@example.org'
  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 unavailable or has not requested the roster, 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 all of the contact's available resources:

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

<presence
  from='user@example.com'
  to='contact@example.org'
  type='unsubscribed' />
```

Upon receiving the presence stanza of type "unavailable" addressed to the contact, the contact's server MUST deliver the unavailable presence to all of the user's available resources:

```
<presence
  from='user@example.com/resource'
  to='contact@example.org'
  type='unavailable' />
```

Note: 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.

## 9. Subscription States

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

### 9.1. Defined States

There are nine possible subscription states, which are described here from the user's (not contact's) perspective:



1. "None" = contact and user are not subscribed to each other, and neither has requested a subscription from the other
2. "None + Pending Out" = contact and user are not subscribed to each other, and user has sent contact a subscription request but contact has not replied yet
3. "None + Pending In" = contact and user are not subscribed to each other, and contact has sent user a subscription request but user has not replied yet (note: contact's server SHOULD NOT push or deliver roster items in this state, but instead SHOULD wait until contact has approved subscription request from user)
4. "None + Pending Out/In" = contact and user are not subscribed to each other, contact has sent user a subscription request but user has not replied yet, and user has sent contact a subscription request but contact has not replied yet
5. "To" = user is subscribed to contact (one-way)
6. "To + Pending In" = user is subscribed to contact, and contact has sent user a subscription request but user has not replied yet
7. "From" = contact is subscribed to user (one-way)
8. "From + Pending Out" = contact is subscribed to user, and user has sent contact a subscription request but contact has not replied yet
9. "Both" = user and contact are subscribed to each other (two-way)

#### 9.2. Server Handling of Outbound Presence Subscription Stanzas

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

Because it is possible for the user's server and the contact's server to lose synchronization regarding subscription states, the user's server MUST without exception route all outbound presence stanzas of type "subscribe" or "unsubscribe" to the contact so that the user is able to resynchronize his or her subscription to the contact's presence information if needed.

The user's server SHOULD NOT route a presence stanza of type "subscribed" or "unsubscribed" to the contact if the stanza does not result in a subscription state change from the user's perspective, and MUST NOT make a state change. If the stanza results in a subscription state change, the user's server MUST route the stanza to the contact and MUST make the appropriate state change. These rules are summarized in the following tables.

Table 1: Recommended handling of outbound "subscribed" stanzas

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

Table 2: Recommended handling of outbound "unsubscribed" stanzas

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"

### 9.3. Server Handling of Inbound Presence Subscription Stanzas

Inbound presence subscription stanzas request a subscription-related action from the user (via the "subscribe" type), inform the user of subscription-related actions taken by the contact (via the "unsubscribe" type), or enable the contact to manage the user's access to the contact's presence information (via the "subscribed" and "unsubscribed" types).

When the user's server receives a subscription request for the user from the contact (i.e., a presence stanza of type "subscribe"), it MUST deliver that request to the user for approval if the user has not already granted the contact access to the user's presence information and if there is no pending inbound subscription request; however, the user's server SHOULD NOT deliver the new request if there is a pending inbound subscription request, since the previous subscription request will have been recorded. If the user has already granted the contact access to the user's presence information, the user's server SHOULD auto-reply to an inbound presence stanza of type "subscribe" from the contact by sending a presence stanza of type "subscribed" to the contact on behalf of the user; this rule enables the contact to resynchronize the subscription state if needed. These rules are summarized in the following table.

Table 3: Recommended handling of inbound "subscribe" stanzas

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

\* Server SHOULD auto-reply with "subscribed" stanza

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

Table 4: Recommended handling of inbound "unsubscribe" stanzas

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"

\* Server SHOULD auto-reply with "unsubscribed" stanza

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

Table 5: Recommended handling of inbound "subscribed" stanzas

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

When the user's server receives a presence stanza of type "unsubscribed" for the user from the contact, it MUST deliver the stanza to the user and MUST change the subscription state if there is a pending outbound request for access to the contact's presence information or if the user currently has access to the contact's presence information. Otherwise, the user's server SHOULD NOT deliver the stanza and MUST NOT change the subscription state. These rules are summarized in the following table.

Table 6: Recommended handling of inbound "unsubscribed" stanzas

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"

#### 9.4. Server Delivery and Client Acknowledgement of Subscription Requests and State Change Notifications

When a server receives an inbound presence stanza of type "subscribe" (i.e., a subscription request) or of type "subscribed", "unsubscribe", or "unsubscribed" (i.e., a subscription state change notification), in addition to sending the appropriate roster push (or updated roster when the roster is next requested by an available resource), it MUST deliver the request or notification to the intended recipient at least once. A server MAY require the recipient to acknowledge receipt of all state change notifications (and MUST require acknowledgement in the case of subscription requests, i.e., presence stanzas of type "subscribe"). In order to require acknowledgement, a server SHOULD send the request or notification to the recipient each time the recipient logs in, until the recipient acknowledges receipt of the notification by "affirming" or "denying" the notification, as shown in the following table:

Table 7: Acknowledgement of subscription state change notifications

STANZA TYPE	ACCEPT	DENY
subscribe	subscribed	unsubscribed
subscribed	subscribe	unsubscribe
unsubscribe	unsubscribed	subscribed
unsubscribed	unsubscribe	subscribe

Obviously, given the foregoing subscription state charts, some of the acknowledgement 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 a user'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 Removing a Roster Item and Cancelling All Subscriptions (Section 8.6)), the server MUST treat a roster remove request as equivalent to sending both of those presence stanzas for purposes of determining whether to continue sending subscription state change notifications of type "subscribe" or "subscribed" to the user.

## 10. 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 [IMP-REQS]). In XMPP this is done by managing one's privacy lists using the 'jabber:iq:privacy' namespace.

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

- o Retrieving one's privacy lists.
- o Adding, removing, and editing one's privacy lists.
- o Setting, changing, or declining active lists.
- o Setting, changing, or declining the default list (i.e., the list that is active by default).
- o Allowing or blocking messages based on JID, group, or subscription type (or globally).

- o Allowing or blocking inbound presence notifications based on JID, group, or subscription type (or globally).
- o Allowing or blocking outbound presence notifications based on JID, group, or subscription type (or globally).
- o Allowing or blocking IQ stanzas based on JID, group, or subscription type (or globally).
- o Allowing or blocking 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.

### 10.1. Syntax and Semantics

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 rule 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='[allow|deny]'
        order='unsignedInt'>
        [<message/>]
        [<presence-in/>]
        [<presence-out/>]
        [<iq/>]
      </item>
    </list>
  </query>
</iq>
```

If the type is "jid", then the 'value' attribute MUST contain a valid Jabber ID. JIDs SHOULD be matched in the following order:

1. <user@domain/resource> (only that resource matches)
2. <user@domain> (any resource matches)
3. <domain/resource> (only that resource matches)
4. <domain> (the domain itself matches, as does any user@domain, domain/resource, or address containing a subdomain)

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 under Roster Syntax and Semantics (Section 7.1), where "none" includes entities that are totally unknown to the user and therefore not in the user's roster at all.

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 "allow" 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.)

The <item/> element MAY contain one or more child elements that enable an entity to specify more granular control over which kinds of stanzas are to be blocked (i.e., rather than blocking all stanzas). The allowable child elements are:

- o <message/> -- blocks incoming message stanzas
- o <iq/> -- blocks incoming IQ stanzas
- o <presence-in/> -- blocks incoming presence notifications
- o <presence-out/> -- blocks outgoing presence notifications



Within the 'jabber:iq:privacy' namespace, the <query/> child of an 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 sending entity violates this rule, the receiving entity MUST return 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 MUST NOT contain any <item/> child elements.

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

## 10.2. Business Rules

1. If there is an active list set for a session, it affects only the session(s) for which it is activated, and only for the duration of the session(s); the server MUST apply the active list only and MUST NOT apply the default list (i.e., there is no "layering" of lists).
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 in accordance with the Server Rules for Handling XML Stanzas (Section 11).
4. Privacy lists MUST be the first delivery rule applied by a server, superseding (1) the routing and delivery rules specified in Server Rules for Handling XML Stanzas (Section 11), and (2) the handling of subscription-related presence stanzas (and corresponding generation of roster pushes) specified in Integration of Roster Items and Presence Subscriptions (Section 8).
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 integer values of the 'order' attribute for each <item/>.

6. As soon as a stanza is matched against a privacy list rule, the server **MUST** appropriately handle the stanza in accordance with the rule and cease processing.
7. If no fall-through item is provided in a list, the fall-through action is assumed to be "allow".
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 or default list occurs during a user's session, subsequent processing based on that list **MUST** take into account the changed state or group (for all resources to which that list currently applies).
10. When the definition for a rule is modified, the server **MUST** send an IQ stanza of type "set" to all connected resources, containing a <query/> element with only one <list/> child element, where the 'name' attribute is set to the name of the modified privacy list. These "privacy list pushes" adhere to the same semantics as the "roster pushes" used in roster management, except that only the list name itself (not the full list definition or the "delta") is pushed to the connected resources. It is up to the receiving resource to determine whether to retrieve the modified list definition, although a connected resource **SHOULD** do so if the list currently applies to it.
11. When a resource attempts to remove a list or specify a new default list while that list applies to a connected resource other than the sending resource, the server **MUST** return a <conflict/> error to the sending resource and **MUST NOT** make the requested change.

### 10.3. Retrieving One's Privacy Lists

Example: Client requests names of privacy lists from server:

```
<iq from='romeo@example.net/orchard' 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 from='romeo@example.net/orchard' 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 from='romeo@example.net/orchard' 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 from='romeo@example.net/orchard' 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@example.org'
        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 to='romeo@example.net/orchard' 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 to='romeo@example.net/orchard' 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>
```

#### 10.4. 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 from='romeo@example.net/orchard' 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='romeo@example.net/orchard' />
```

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 to='romeo@example.net/orchard' 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, the connected resource MUST send an empty <active/> element with no 'name' attribute.

Example: Client declines the use of active lists:

```
<iq from='romeo@example.net/orchard' type='set' id='active3'>
  <query xmlns='jabber:iq:privacy'>
    <active />
  </query>
</iq>
```

Example: Server acknowledges success of declining any active list:

```
<iq type='result' id='active3' to='romeo@example.net/orchard' />
```

## 10.5. Managing the Default List

In order to change its default list (which applies to the user as a whole, not only the sending resource), 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: User requests change of default list:

```
<iq from='romeo@example.net/orchard' 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='romeo@example.net/orchard' />
```

If the user attempts to change which list is the default list but the default list is in use by at least one connected resource other than the sending resource, the server MUST return a <conflict/> stanza error to the sending resource:

Example: Client attempts to change the default list but that list is in use by another resource:

```
<iq to='romeo@example.net/orchard' type='error' id='default1'>
  <query xmlns='jabber:iq:privacy'>
    <default name='special' />
  </query>
  <error type='cancel'>
    <conflict
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

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 to='romeo@example.net/orchard' type='error' id='default1'>
  <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), the user MUST send an empty `<default/>` element with no 'name' attribute.

Example: Client declines the use of the default list:

```
<iq from='romeo@example.net/orchard' type='set' id='default2'>
  <query xmlns='jabber:iq:privacy'>
    <default />
  </query>
</iq>
```

Example: Server acknowledges success of declining any default list:

```
<iq type='result' id='default2' to='romeo@example.net/orchard' />
```

If one connected resource attempts to decline the use of a default list for the user as a whole but the default list currently applies to at least one other connected resource, the server MUST return a `<conflict/>` error to the sending resource:

Example: Client attempts to decline a default list but that list is in use by another resource:

```
<iq to='romeo@example.net/orchard' type='error' id='default3'>
  <query xmlns='jabber:iq:privacy'>
    <default />
  </query>
  <error type='cancel'>
    <conflict
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    </error>
</iq>
```



## 10.6. 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 from='romeo@example.net/orchard' 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>
```

Example: Server acknowledges success of list edit:

```
<iq type='result' id='edit1' to='romeo@example.net/orchard' />
```

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.

The server **MUST** now send a "privacy list push" to all connected resources:

Example: Privacy list push on list edit:

```
<iq to='romeo@example.net/orchard' type='set' id='push1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public' />
  </query>
</iq>
```

```
<iq to='romeo@example.net/home' type='set' id='push2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public' />
  </query>
</iq>
```

In accordance with the semantics of IQ stanzas defined in [XMPP-CORE], each connected resource MUST return an IQ result to the server as well:

Example: Acknowledging receipt of privacy list pushes:

```
<iq from='romeo@example.net/orchard'
  type='result'
  id='push1' />

<iq from='romeo@example.net/home'
  type='result'
  id='push2' />
```

#### 10.7. 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. As with list edits, the server MUST also send a "privacy list push" to all connected resources.

#### 10.8. 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 from='romeo@example.net/orchard' 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='romeo@example.net/orchard' />
```

If a user attempts to remove a list that is currently being applied to at least one resource other than the sending resource, the server MUST return a <conflict/> stanza error to the user; i.e., the user MUST first set another list to active or default before attempting to remove 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.

### 10.9. Blocking Messages

Server-side privacy lists enable a user to block incoming messages from other entities based on the entity's JID, roster group, or subscription status (or globally). The following examples illustrate the protocol. (Note: For the sake of brevity, IQ stanzas of type "result" are not shown in the following examples, nor are "privacy list pushes".)

Example: User blocks based on JID:

```
<iq from='romeo@example.net/orchard' 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 entity with the specified JID.

Example: User blocks based on roster group:

```
<iq from='romeo@example.net/orchard' 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>
```

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

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

Example: User blocks based on subscription type:

```
<iq from='romeo@example.net/orchard' 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 entities with the specified subscription type.

Example: User blocks globally:

```
<iq from='romeo@example.net/orchard' 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.

#### 10.10. Blocking Inbound Presence Notifications

Server-side privacy lists enable a user to block incoming presence notifications from other entities based on the entity'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 is currently 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 from='romeo@example.net/orchard' 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 entity with the specified JID.

Example: User blocks based on roster group:

```
<iq from='romeo@example.net/orchard' 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 entities in the specified roster group.

Example: User blocks based on subscription type:

```
<iq from='romeo@example.net/orchard' 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 entities with the specified subscription type.

Example: User blocks globally:

```
<iq from='romeo@example.net/orchard' 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.

#### 10.11. Blocking Outbound Presence Notifications

Server-side privacy lists enable a user to block outgoing presence notifications to other entities based on the entity'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 are currently subscribed to the user's presence information. Thus this includes presence stanzas with no 'type' attribute or of type='unavailable' only.

Example: User blocks based on JID:

```
<iq from='romeo@example.net/orchard' 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 entity with the specified JID.

Example: User blocks based on roster group:

```
<iq from='romeo@example.net/orchard' 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 entities in the specified roster group.

Example: User blocks based on subscription type:

```
<iq from='romeo@example.net/orchard' 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>
```

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

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

Example: User blocks globally:

```
<iq from='romeo@example.net/orchard' 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.

#### 10.12. Blocking IQ Stanzas

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

Example: User blocks based on JID:

```
<iq from='romeo@example.net/orchard' 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 stanzas from the entity with the specified JID.



Example: User blocks based on roster group:

```
<iq from='romeo@example.net/orchard' 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 stanzas from any entities in the specified roster group.

Example: User blocks based on subscription type:

```
<iq from='romeo@example.net/orchard' 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 stanzas from any entities with the specified subscription type.

Example: User blocks globally:

```
<iq from='romeo@example.net/orchard' 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 stanzas from any other users.

### 10.13. Blocking All Communication

Server-side privacy lists enable a user to block all stanzas from and to other entities based on the entity's JID, roster group, or subscription status (or globally). Note that this includes subscription-related presence stanzas, which are excluded by Blocking Inbound Presence Notifications (Section 10.10). The following examples illustrate the protocol.

Example: User blocks based on JID:

```
<iq from='romeo@example.net/orchard' 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 entity with the specified JID.

Example: User blocks based on roster group:

```
<iq from='romeo@example.net/orchard' 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 entities in the specified roster group.

Example: User blocks based on subscription type:

```
<iq from='romeo@example.net/orchard' type='set' id='all13'>
  <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 entities with the specified subscription type.

Example: User blocks globally:

```
<iq from='romeo@example.net/orchard' type='set' id='all14'>
  <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.

#### 10.14. Blocked Entity Attempts to Communicate with User

If a blocked entity attempts to send message or presence stanzas 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 <service-unavailable/> 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:version' />
</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:version' />
  <error type='cancel'>
    <service-unavailable
      xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
</iq>
```

#### 10.15. 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:

- o allow communications from all JIDs in my roster (i.e., listing each JID as a separate list item), but block communications with everyone else
- o allow 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 block communications from everyone else
- o allow communications from any user with whom I have a subscription of 'both' or 'to' or 'from' (i.e., listing each subscription value separately), but block communications from everyone else

- o block 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>
```

## 11. Server Rules for Handling XML Stanzas

Basic routing and delivery rules for servers are defined in [XMPP-CORE]. This section defines additional rules for XMPP-compliant instant messaging and presence servers.

### 11.1. Inbound Stanzas

If the hostname of the domain identifier portion of the JID contained in the 'to' attribute of an inbound stanza matches a hostname of the server itself and the JID contained in the 'to' attribute is of the form <user@example.com> or <user@example.com/resource>, the server MUST first apply any privacy lists (Section 10) that are in force, then follow the rules defined below:

1. If the JID is of the form <user@domain/resource> and an available resource matches the full JID, the recipient's server MUST deliver the stanza to that resource.
2. Else if the JID is of the form <user@domain> or <user@domain/resource> and the associated user account does not exist, the recipient's server (a) SHOULD silently ignore the stanza (i.e., neither deliver it nor return an error) if it is a presence stanza, (b) MUST return a <service-unavailable/> stanza error to the sender if it is an IQ stanza, and (c) SHOULD return a <service-unavailable/> stanza error to the sender if it is a message stanza.
3. Else if the JID is of the form <user@domain/resource> and no available resource matches the full JID, the recipient's server (a) SHOULD silently ignore the stanza (i.e., neither deliver it

nor return an error) if it is a presence stanza, (b) MUST return a <service-unavailable/> stanza error to the sender if it is an IQ stanza, and (c) SHOULD treat the stanza as if it were addressed to <user@domain> if it is a message stanza.

4. Else if the JID is of the form <user@domain> 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 highest-priority available resource (if the resource did not provide a value for the <priority/> element, the server SHOULD consider it to have provided a value of zero). If two or more available resources have the same priority, the server MAY use some other rule (e.g., most recent connect time, most recent activity time, or highest availability as determined by some hierarchy of <show/> values) to choose between them or MAY deliver the message to all such resources. However, the server MUST NOT deliver the stanza to an available resource with a negative priority; if the only available resource has a negative priority, the server SHOULD handle the message as if there were no available resources (defined below). In addition, the server MUST NOT rewrite the 'to' attribute (i.e., it MUST leave it as <user@domain> rather than change it to <user@domain/resource>).
  2. For presence stanzas other than those of type "probe", the server MUST deliver the stanza to all available resources; for presence probes, the server SHOULD reply based on the rules defined in Presence Probes (Section 5.1.3). In addition, the server MUST NOT rewrite the 'to' attribute (i.e., it MUST leave it as <user@domain> rather than change it to <user@domain/resource>).
  3. For IQ stanzas, the server itself MUST reply on behalf of the user with either an IQ result or an IQ error, and MUST NOT deliver the IQ stanza to any of the available resources. 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; if not, the server MUST reply with a <service-unavailable/> stanza error.
5. Else if the JID is of the form <user@domain> and there are no available resources associated with the user, how the stanza is handled depends on the stanza type:

1. For presence stanzas of type "subscribe", "subscribed", "unsubscribe", and "unsubscribed", the server MUST maintain a record of the stanza and deliver the stanza at least once (i.e., when the user next creates an available resource); in addition, the server MUST continue to deliver presence stanzas of type "subscribe" until the user either approves or denies the subscription request (see also Presence Subscriptions (Section 5.1.6)).
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, or forward the message to the user via some other means (e.g., to the user's email account). However, if offline message storage or message forwarding is not enabled, the server MUST return to the sender a <service-unavailable/> stanza error. (Note: Offline message storage and message forwarding are not defined in XMPP, since they are strictly a matter of implementation and service provisioning.)
4. For IQ stanzas, the server itself MUST reply on behalf of the user with either an IQ result or an IQ error. Specifically, if the semantics of the qualifying namespace define a reply that the server can provide, the server MUST reply to the stanza on behalf of the user; if not, the server MUST reply with a <service-unavailable/> stanza error.

## 11.2. Outbound Stanzas

If the hostname of the domain identifier portion of the address contained in the 'to' attribute of an outbound stanza matches a hostname of the server itself, the server MUST deliver the stanza to a local entity according the rules for Inbound Stanzas (Section 11.1).

If the hostname of the domain identifier portion of the address contained in the 'to' attribute of an outbound stanza does not match a hostname of the server itself, the server MUST attempt to route the stanza to the foreign domain. The recommended order of actions is as follows:

1. First attempt to resolve the foreign hostname using an [SRV] Service of "xmpp-server" and Proto of "tcp", resulting in resource records such as "\_xmpp-server.\_tcp.example.com.", as specified in [XMPP-CORE].
2. If the "xmpp-server" address record resolution fails, attempt to resolve the "\_im" or "\_pres" [SRV] Service as specified in [IMP-SRV], using the "\_im" Service for <message/> stanzas and the "\_pres" Service for <presence/> stanzas (it is up to the implementation how to handle <iq/> stanzas). This will result in one or more resolutions of the form "\_im.<proto>.example.com." or "\_pres.<proto>.example.com.", where "<proto>" would be a label registered in the Instant Messaging SRV Protocol Label registry or the Presence SRV Protocol Label registry: either "\_xmpp" for an XMPP-aware domain or some other IANA-registered label (e.g., "\_simple") for a non-XMPP-aware domain.
3. If both SRV address record resolutions fail, attempt to perform a normal IPv4/IPv6 address record resolution to determine the IP address using the "xmpp-server" port of 5269 registered with the IANA, as specified in [XMPP-CORE].

Administrators of server deployments are strongly encouraged to keep the \_im.\_xmpp, \_pres.\_xmpp, and \_xmpp.\_tcp SRV records properly synchronized, since different implementations might perform the "\_im" and "\_pres" lookups before the "xmpp-server" lookup.

## 12. IM and Presence Compliance Requirements

This section summarizes the specific aspects of the Extensible Messaging and Presence Protocol that MUST be supported by instant messaging and presence servers and clients in order to be considered compliant implementations. All such applications MUST comply with the requirements specified in [XMPP-CORE]. The text in this section specifies additional compliance requirements for instant messaging and presence servers and clients; note well that the requirements described here supplement but do not supersede the core requirements. Note also that a server or client MAY support only presence or instant messaging, and is not required to support both if only a presence service or an instant messaging service is desired.

### 12.1. Servers

In addition to core server compliance requirements, an instant messaging and presence server MUST additionally support the following protocols:



- o All server-related instant messaging and presence syntax and semantics defined in this document, including presence broadcast on behalf of clients, presence subscriptions, roster storage and manipulation, privacy lists, and IM-specific routing and delivery rules

## 12.2. Clients

In addition to core client compliance requirements, an instant messaging and presence client **MUST** additionally support the following protocols:

- o Generation and handling of the IM-specific semantics of XML stanzas as defined by the XML schemas, including the 'type' attribute of message and presence stanzas as well as their child elements
- o All client-related instant messaging syntax and semantics defined in this document, including presence subscriptions, roster management, and privacy lists
- o End-to-end object encryption as defined in End-to-End Object Encryption in the Extensible Messaging and Presence Protocol (XMPP) [XMPP-E2E]

A client **MUST** also handle addresses that are encoded as "im:" URIs as specified in [CPIM], and **MAY** do so by removing the "im:" scheme and entrusting address resolution to the server as specified under Outbound Stanzas (Section 11.2).

## 13. Internationalization Considerations

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

## 14. Security Considerations

Core security considerations for XMPP are defined in the relevant section of [XMPP-CORE].

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

- o When a server processes an inbound stanza of any kind whose intended recipient is a user associated with one of the server's hostnames, the server MUST first apply any privacy lists (Section 10) that are in force (see Server Rules for Handling XML Stanzas (Section 11)).
- o When a server processes an inbound presence stanza of type "probe" whose intended recipient is a user associated with one of the server's hostnames, the server MUST NOT reveal the user's presence information if the sender is an entity that is not authorized to receive that information as determined by presence subscriptions (see Client and Server Presence Responsibilities (Section 5.1)).
- o When a server processes an outbound presence stanza with no type or of type "unavailable", it MUST follow the rules defined under Client and Server Presence Responsibilities (Section 5.1) in order to ensure that such presence information is not broadcasted to entities that are not authorized to know such information.
- o When a server generates an error stanza in response to receiving a stanza for a user who does not exist, the use of the `<service-unavailable/>` error condition helps protect against well-known dictionary attacks, since this is the same error condition that is returned if, for instance, the namespace of an IQ child element is not understood, or if offline message storage or message forwarding is not enabled for a domain.

## 15. IANA Considerations

For a number of related IANA considerations, refer to the relevant section of [XMPP-CORE].

### 15.1. 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. (This namespace name adheres to the format defined in The IETF XML Registry [XML-REG].)

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

Specification: RFC 3921

Description: This is the XML namespace name for session-related data in the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 3921.

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

## 15.2. Instant Messaging SRV Protocol Label Registration

Address Resolution for Instant Messaging and Presence [IMP-SRV] defines an Instant Messaging SRV Protocol Label registry for protocols that can provide services that conform to the "\_im" SRV Service label. Because XMPP is one such protocol, the IANA registers the "\_xmpp" protocol label in the appropriate registry, as follows:

Protocol label: \_xmpp

Specification: RFC 3921

Description: Instant messaging protocol label for the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 3921.

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

## 15.3. Presence SRV Protocol Label Registration

Address Resolution for Instant Messaging and Presence [IMP-SRV] defines a Presence SRV Protocol Label registry for protocols that can provide services that conform to the "\_pres" SRV Service label. Because XMPP is one such protocol, the IANA registers the "\_xmpp" protocol label in the appropriate registry, as follows:

Protocol label: \_xmpp

Specification: RFC 3921

Description: Presence protocol label for the Extensible Messaging and Presence Protocol (XMPP) as defined by RFC 3921.

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

## 16. References

### 16.1. Normative References

- [CPIM] Peterson, J., "Common Profile for Instant Messaging (CPIM)", RFC 3860, August 2004.
- [IMP-REQS] Day, M., Aggarwal, S., Mohr, G., and J. Vincent, "Instant Messaging/Presence Protocol Requirements", RFC 2779, February 2000.
- [IMP-SRV] Peterson, J., "Address Resolution for Instant Messaging and Presence", RFC 3861, August 2004.
- [SRV] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, February 2000.
- [TERMS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

- [XML] Bray, T., Paoli, J., Sperberg-McQueen, C., and E. Maler, "Extensible Markup Language (XML) 1.0 (2nd ed)", W3C REC-xml, October 2000, <<http://www.w3.org/TR/REC-xml>>.
- [XML-NAMES] Bray, T., Hollander, D., and A. Layman, "Namespaces in XML", W3C REC-xml-names, January 1999, <<http://www.w3.org/TR/REC-xml-names>>.
- [XMPP-CORE] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 3920, October 2004.
- [XMPP-E2E] Saint-Andre, P., "End-to-End Object Encryption in the Extensible Messaging and Presence Protocol (XMPP)", RFC 3923, October 2004.

## 16.2. Informative References

- [IMP-MODEL] Day, M., Rosenberg, J., and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000.
- [IRC] Oikarinen, J. and D. Reed, "Internet Relay Chat Protocol", RFC 1459, May 1993.
- [JEP-0054] Saint-Andre, P., "vcard-temp", JSF JEP 0054, March 2003.
- [JEP-0077] Saint-Andre, P., "In-Band Registration", JSF JEP 0077, August 2004.
- [JEP-0078] Saint-Andre, P., "Non-SASL Authentication", JSF JEP 0078, July 2004.
- [JSF] Jabber Software Foundation, "Jabber Software Foundation", <<http://www.jabber.org/>>.
- [VCARD] Dawson, F. and T. Howes, "vCard MIME Directory Profile", RFC 2426, September 1998.
- [XML-REG] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

## Appendix A. vCards

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

## Appendix B. XML Schemas

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

### B.1 jabber:client

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

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

  <xs:import namespace='urn:ietf:params:xml:ns:xmpp-stanzas'/>

  <xs:element name='message'>
    <xs:complexType>
      <xs:sequence>
        <xs:choice minOccurs='0' maxOccurs='unbounded'>
          <xs:element ref='subject'/>
          <xs:element ref='body'/>
          <xs:element ref='thread'/>
        </xs:choice>
        <xs:any namespace='##other'
          minOccurs='0'
          maxOccurs='unbounded'/>
        <xs:element ref='error'
          minOccurs='0'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
</xs:sequence>
<xs:attribute name='from'
               type='xs:string'
               use='optional' />
<xs:attribute name='id'
               type='xs:NMTOKEN'
               use='optional' />
<xs:attribute name='to'
               type='xs:string'
               use='optional' />
<xs:attribute name='type' use='optional' default='normal'>
  <xs:simpleType>
    <xs:restriction base='xs:NCName'>
      <xs:enumeration value='chat' />
      <xs:enumeration value='error' />
      <xs:enumeration value='groupchat' />
      <xs:enumeration value='headline' />
      <xs:enumeration value='normal' />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute ref='xml:lang' use='optional' />
</xs:complexType>
</xs:element>

<xs:element name='body'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute ref='xml:lang' use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name='subject'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute ref='xml:lang' use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name='thread' type='xs:NMTOKEN' />

<xs:element name='presence'>
```

```
<xs:complexType>
  <xs:sequence>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element ref='show' />
      <xs:element ref='status' />
      <xs:element ref='priority' />
    </xs:choice>
    <xs:any namespace='##other'
      minOccurs='0'
      maxOccurs='unbounded' />
    <xs:element ref='error'
      minOccurs='0' />
  </xs:sequence>
  <xs:attribute name='from'
    type='xs:string'
    use='optional' />
  <xs:attribute name='id'
    type='xs:NMTOKEN'
    use='optional' />
  <xs:attribute name='to'
    type='xs:string'
    use='optional' />
  <xs:attribute name='type' use='optional'>
    <xs:simpleType>
      <xs:restriction base='xs:NCName'>
        <xs:enumeration value='error' />
        <xs:enumeration value='probe' />
        <xs:enumeration value='subscribe' />
        <xs:enumeration value='subscribed' />
        <xs:enumeration value='unavailable' />
        <xs:enumeration value='unsubscribe' />
        <xs:enumeration value='unsubscribed' />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute ref='xml:lang' use='optional' />
</xs:complexType>
</xs:element>

<xs:element name='show'>
  <xs:simpleType>
    <xs:restriction base='xs:NCName'>
      <xs:enumeration value='away' />
      <xs:enumeration value='chat' />
      <xs:enumeration value='dnd' />
      <xs:enumeration value='xa' />
    </xs:restriction>
  </xs:simpleType>
```

```
</xs:element>

<xs:element name='status'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute ref='xml:lang' use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name='priority' type='xs:byte' />

<xs:element name='iq'>
  <xs:complexType>
    <xs:sequence>
      <xs:any namespace='##other'
        minOccurs='0' />
      <xs:element ref='error'
        minOccurs='0' />
    </xs:sequence>
    <xs:attribute name='from'
      type='xs:string'
      use='optional' />
    <xs:attribute name='id'
      type='xs:NMTOKEN'
      use='required' />
    <xs:attribute name='to'
      type='xs:string'
      use='optional' />
    <xs:attribute name='type' use='required'>
      <xs:simpleType>
        <xs:restriction base='xs:NCName'>
          <xs:enumeration value='error' />
          <xs:enumeration value='get' />
          <xs:enumeration value='result' />
          <xs:enumeration value='set' />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute ref='xml:lang' use='optional' />
  </xs:complexType>
</xs:element>

<xs:element name='error'>
  <xs:complexType>
    <xs:sequence xmlns:err='urn:ietf:params:xml:ns:xmpp-stanzas'>
```



```

    <xs:group    ref='err:stanzaErrorGroup' />
    <xs:element  ref='err:text'
                minOccurs='0' />
</xs:sequence>
<xs:attribute name='code' type='xs:byte' use='optional' />
<xs:attribute name='type' use='required'>
  <xs:simpleType>
    <xs:restriction base='xs:NCName'>
      <xs:enumeration value='auth' />
      <xs:enumeration value='cancel' />
      <xs:enumeration value='continue' />
      <xs:enumeration value='modify' />
      <xs:enumeration value='wait' />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>

</xs:schema>

```

## B.2 jabber:server

```

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

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

  <xs:import namespace='urn:ietf:params:xml:ns:xmpp-stanzas' />

  <xs:element name='message'>
    <xs:complexType>
      <xs:sequence>
        <xs:choice minOccurs='0' maxOccurs='unbounded'>
          <xs:element ref='subject' />
          <xs:element ref='body' />
          <xs:element ref='thread' />
        </xs:choice>
        <xs:any
          namespace='##other'
          minOccurs='0'
          maxOccurs='unbounded' />
        <xs:element ref='error'
          minOccurs='0' />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

```

```
<xs:attribute name='from'
              type='xs:string'
              use='required' />
<xs:attribute name='id'
              type='xs:NMTOKEN'
              use='optional' />
<xs:attribute name='to'
              type='xs:string'
              use='required' />
<xs:attribute name='type' use='optional' default='normal'>
  <xs:simpleType>
    <xs:restriction base='xs:NCName'>
      <xs:enumeration value='chat' />
      <xs:enumeration value='error' />
      <xs:enumeration value='groupchat' />
      <xs:enumeration value='headline' />
      <xs:enumeration value='normal' />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute ref='xml:lang' use='optional' />
</xs:complexType>
</xs:element>

<xs:element name='body'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute ref='xml:lang' use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name='subject'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute ref='xml:lang' use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name='thread' type='xs:NMTOKEN' />

<xs:element name='presence'>
  <xs:complexType>
```

```

<xs:sequence>
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element ref='show' />
    <xs:element ref='status' />
    <xs:element ref='priority' />
  </xs:choice>
  <xs:any namespace='##other'
    minOccurs='0'
    maxOccurs='unbounded' />
  <xs:element ref='error'
    minOccurs='0' />
</xs:sequence>
<xs:attribute name='from'
  type='xs:string'
  use='required' />
<xs:attribute name='id'
  type='xs:NMTOKEN'
  use='optional' />
<xs:attribute name='to'
  type='xs:string'
  use='required' />
<xs:attribute name='type' use='optional'>
  <xs:simpleType>
    <xs:restriction base='xs:NCName'>
      <xs:enumeration value='error' />
      <xs:enumeration value='probe' />
      <xs:enumeration value='subscribe' />
      <xs:enumeration value='subscribed' />
      <xs:enumeration value='unavailable' />
      <xs:enumeration value='unsubscribe' />
      <xs:enumeration value='unsubscribed' />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute ref='xml:lang' use='optional' />
</xs:complexType>
</xs:element>

<xs:element name='show'>
  <xs:simpleType>
    <xs:restriction base='xs:NCName'>
      <xs:enumeration value='away' />
      <xs:enumeration value='chat' />
      <xs:enumeration value='dnd' />
      <xs:enumeration value='xa' />
    </xs:restriction>
  </xs:simpleType>
</xs:element>

```

```
<xs:element name='status'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute ref='xml:lang' use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name='priority' type='xs:byte' />

<xs:element name='iq'>
  <xs:complexType>
    <xs:sequence>
      <xs:any namespace='##other'
        minOccurs='0' />
      <xs:element ref='error'
        minOccurs='0' />
    </xs:sequence>
    <xs:attribute name='from'
      type='xs:string'
      use='required' />
    <xs:attribute name='id'
      type='xs:NMTOKEN'
      use='required' />
    <xs:attribute name='to'
      type='xs:string'
      use='required' />
    <xs:attribute name='type' use='required'>
      <xs:simpleType>
        <xs:restriction base='xs:NCName'>
          <xs:enumeration value='error' />
          <xs:enumeration value='get' />
          <xs:enumeration value='result' />
          <xs:enumeration value='set' />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute ref='xml:lang' use='optional' />
  </xs:complexType>
</xs:element>

<xs:element name='error'>
  <xs:complexType>
    <xs:sequence xmlns:err='urn:ietf:params:xml:ns:xmpp-stanzas'>
      <xs:group ref='err:stanzaErrorGroup' />
      <xs:element ref='err:text' />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

        minOccurs='0' />
    </xs:sequence>
    <xs:attribute name='code' type='xs:byte' use='optional' />
    <xs:attribute name='type' use='required'>
        <xs:simpleType>
            <xs:restriction base='xs:NCName'>
                <xs:enumeration value='auth' />
                <xs:enumeration value='cancel' />
                <xs:enumeration value='continue' />
                <xs:enumeration value='modify' />
                <xs:enumeration value='wait' />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
</xs:element>

</xs:schema>

```

### B.3 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.4 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' />
      <xs:element ref='default'
        minOccurs='0' />
      <xs:element ref='list'
        minOccurs='0'
        maxOccurs='unbounded' />
    </xs:sequence>
  </xs:complexType>
</xs:element>

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

<xs:element name='default'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:NMTOKEN'>
        <xs:attribute name='name'
          type='xs:string'
          use='optional' />
      </xs:extension>
    </xs:simpleContent>
  </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 name='iq'
                  minOccurs='0'
                  type='empty' />
      <xs:element name='message'
                  minOccurs='0'
                  type='empty' />
      <xs:element name='presence-in'
                  minOccurs='0'
                  type='empty' />
      <xs:element name='presence-out'
                  minOccurs='0'
                  type='empty' />
    </xs:sequence>
    <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='order'
                  type='xs:unsignedInt'
                  use='required' />
    <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:attribute name='value'
                  type='xs:string'
                  use='optional' />
  </xs:complexType>
</xs:element>
```

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

</xs:schema>
```

## B.5 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='ask' use='optional'>
        <xs:simpleType>
          <xs:restriction base='xs:NCName'>
            <xs:enumeration value='subscribe' />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name='jid' type='xs:string' use='required' />
      <xs:attribute name='name' type='xs:string' use='optional' />
      <xs:attribute name='subscription' use='optional'>
        <xs:simpleType>
          <xs:restriction base='xs:NCName'>
```



```
        <xs:enumeration value='both' />
        <xs:enumeration value='from' />
        <xs:enumeration value='none' />
        <xs:enumeration value='remove' />
        <xs:enumeration value='to' />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
</xs:element>

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

</xs:schema>
```

## Appendix C. Differences Between Jabber IM/Presence Protocols and XMPP

This section is non-normative.

XMPP has been adapted from the protocols originally developed in the Jabber open-source community, which can be thought of as "XMPP 0.9". Because there exists a large installed base of Jabber implementations and deployments, it may be helpful to specify the key differences between the relevant Jabber protocols and XMPP in order to expedite and encourage upgrades of those implementations and deployments to XMPP. This section summarizes the differences that relate specifically to instant messaging and presence applications, while the corresponding section of [XMPP-CORE] summarizes the differences that relate to all XMPP applications.

### C.1 Session Establishment

The client-to-server authentication protocol developed in the Jabber community assumed that every client is an IM client and therefore initiated an IM session upon successful authentication and resource binding, which are performed simultaneously (documentation of this protocol is contained in [JEP-0078], published by the Jabber Software Foundation [JSF]). XMPP maintains a stricter separation between core functionality and IM functionality; therefore, an IM session is not created until the client specifically requests one using the protocol defined under Session Establishment (Section 3).

## C.2 Privacy Lists

The Jabber community began to define a protocol for communications blocking (privacy lists) in late 2001, but that effort was deprecated once the XMPP Working Group was formed. Therefore the protocol defined under Blocking Communication (Section 10) is the only such protocol defined for use in the Jabber community.

### Contributors

Most of the core aspects of the Extensible Messaging and Presence Protocol were developed originally within the Jabber open-source community in 1999. This community was founded by Jeremie Miller, who released source code for the initial version of the jabberd server in January 1999. Major early contributors to the base protocol also included Ryan Eatmon, Peter Millard, Thomas Muldowney, and Dave Smith. Work specific to instant messaging and presence by the XMPP Working Group has concentrated especially on IM session establishment and communication blocking (privacy lists); the session establishment protocol was mainly developed by Rob Norris and Joe Hildebrand, and the privacy lists protocol was originally contributed by Peter Millard.

### Acknowledgements

Thanks are due to a number of individuals in addition to the contributors listed. Although it is difficult to provide a complete list, the following individuals were particularly helpful in defining the protocols or in commenting on the specifications in this memo: Thomas Charron, Richard Dobson, Schuyler Heath, Jonathan Hogg, Craig Kaes, Jacek Konieczny, Lisa Dusseault, Alexey Melnikov, Keith Minkler, Julian Missig, Pete Resnick, Marshall Rose, Jean-Louis Seguneau, Alexey Shchepin, Iain Shigeoka, and David Waite. Thanks also to members of the XMPP Working Group and the IETF community for comments and feedback provided throughout the life of this memo.

### Author's Address

Peter Saint-Andre (editor)  
Jabber Software Foundation

EMail: [stpeter@jabber.org](mailto:stpeter@jabber.org)

## Full Copyright Statement

Copyright (C) The Internet Society (2004).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/S HE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM 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.

## Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the IETF's procedures with respect to rights in IETF Documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at [ietf-ipr@ietf.org](mailto:ietf-ipr@ietf.org).

## Acknowledgement

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

