Network Working Group
Internet-Draft
Expires: September 24, 2003

P. Saint-Andre
J. Miller
Jabber Software Foundation

March 26, 2003

XMPP Instant Messaging
draft-ietf-xmpp-im-06

Status of this Memo

Copyright Notice

Abstract

This document describes the specific extensions to and applications of the Extensible Messaging and Presence Protocol (XMPP) that are necessary to create a basic instant messaging and presence application such as that provided by the servers, clients, and other applications that comprise the XMPP-based Jabber network.

Table of Contents

1. Introduction

1.1 Overview

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

1.2 Requirements

For the purposes of this document, we stipulate that a basic instant messaging and presence application needs to enable a user to perform the following high-level use cases:

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

Detailed definitions of these functionality areas are contained in RFC 2779 [2], and the interested reader is directed to that document regarding the requirements addressed herein.

Note: although XMPP IM meets the requirements of RFC 2779, it was not designed explicitly with RFC 2779 in mind, since the base protocol evolved through an open development process within the Jabber open-source community, mainly in 1999. In addition, protocols addressing many other functionality areas have been defined and continue to be defined by the Jabber Software Foundation [3]. These include service discovery, multi-user chat, data gathering and forms submission, feature negotiation, message composing events, message expiration, delayed delivery, file transfer, XHTML message formatting, publish-subscribe, and transports for XML-RPC and SOAP. However, such

protocols are not described herein because they are not required by RFC 2779 [2].

1.3 Terminology

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

The capitalized key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [4].

1.4 Discussion Venue

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

1.5 Intellectual Property Notice

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

2. Authentication and Authorization

Because XMPP-based IM applications are usually implemented via a client-server architecture, a user must first acquire an account on a server. Although account provisioning is outside the scope of XMPP, methods for doing so include account creation by a server administrator as well as in-band account registration using the 'jabber:iq:register' namespace; the latter method is documented by the Jabber Software Foundation [3] at <http://www.jabber.org/protocol/>.

Once a user has acquired an account, the user MUST authenticate with the server hosting his or her account in order to gain access to the network. If a user's client is capable of authenticating by means of SASL, it MUST include a 'version' attribute (set to a value of "1.0") within the opening <stream> tag with which it initiates communications with the server. The protocol describing how a client authenticates with a server using SASL is defined in XMPP Core [1]. (Note: the Jabber precursor to XMPP contained a client-server authentication protocol that was enforced after the stream was negotiated; this protocol is not supported in XMPP but is documented by the Jabber Software Foundation [3] at <http://www.jabber.org/protocol/>.

Once a client has authenticated with a server, it MUST define a resource that the server can associate with the connection for purposes of authorization and addressing. This is necessary because stanzas sent to or received from the server within the context of an active IM session use a "full JID" (<user@domain/resource>) for addressing. Authorizing a resource is accomplished by means of the 'jabber:iq:auth' namespace as described below.

Step 1: Client queries server regarding information that is still required to begin a session:

```
<iq type='get' id='res_1'>
  <query xmlns='jabber:iq:auth'>
    <username>juliet</username>
  </query>
</iq>
```

Step 2: Server responds with the required fields (in this case, only the username and authorized resource):

```
<iq type='result' id='res_1'>
  <query xmlns='jabber:iq:auth'>
    <username>juliet</username>
    <resource/>
  </query>
</iq>
```

Step 3: Client sends name of authorized resource:

```
<iq type='set' id='res_2'>
  <query xmlns='jabber:iq:auth'>
    <username>juliet</username>
    <resource>balcony</resource>
  </query>
</iq>
```

Step 4: Server informs client of successful session initiation:

```
<iq type='result' id='res_2'/>
```

Step 4 (alt): Server informs client of XML formatting error:

```
<iq type='result' id='res_2'>
  <query xmlns='jabber:iq:auth'>
    <username>juliet</username>
  </query>
  <error class='app'>
    <auth-condition xmlns='jabber:iq:auth:error'>
      <no-resource-provided/>
    </auth-condition>
  </error>
</iq>
```

Step 4 (alt): Server informs client of resource formatting error:

```
<iq type='result' id='res_2'>
  <query xmlns='jabber:iq:auth'>
    <username>juliet</username>
    <resource> </username>
  </query>
  <error class='app'>
    <auth-condition xmlns='jabber:iq:auth:error'>
      <bad-resource-format/>
    </auth-condition>
  </error>
</iq>
```

Step 4 (alt): Server informs client of resource conflict (the desired resource name is already in use by another active connection):

```
<iq type='result' id='res_2'>
  <query xmlns='jabber:iq:auth'>
    <username>juliet</username>
    <resource>balcony</resource>
  </query>
  <error class='app'>
    <auth-condition xmlns='jabber:iq:auth:error'>
      <resource-conflict/>
    </auth-condition>
  </error>
</iq>
```

Note: existing implementations usually terminate the existing session when a client attempts to authorize a new resource with the same name as the existing session; however, is up to the implementation whether to terminate the existing session or to inform the attempted new session about the resource conflict and thus require the new session to authorize a different resource name.

3. Exchanging Messages

Exchanging messages is a basic use of XMPP and is effected when a user sends a message stanza to another user (or, more generally, another entity). As defined under Routing and Delivery Rules (Section 9), 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).

Detailed information regarding the syntax of message stanzas and their defined attributes and child elements may be found in XMPP Core [1].

3.1 Specifying an Intended Recipient

A client SHOULD specify an intended recipient for the message by providing an appropriate JID in the 'to' attribute of the <message/> element. Normally, the value of the 'to' attribute specifies an entity other than the sending user (see Section 9 for exceptions). The intended recipient MAY be any valid JID (a user on the same server, a user on a different server, the server itself, another server, etc.). If the JID is invalid or cannot be contacted, the entity discovering that fact (usually the sender's or recipient's server) MUST return an error to the sender.

3.2 Specifying a Message Type

As mentioned in XMPP Core [1], there are several defined types of messages (specified by means of a 'type' attribute within the <message/> element). In the context of an instant messaging application, a client MAY include a message type in order to capture the conversational context of the message, thus providing a hint regarding presentation (e.g., in a GUI). If no 'type' attribute is provided, the message SHOULD be assumed to be a standalone message to which the recipient MAY reply if desired. If the 'type' attribute is included, it SHOULD have one of the following values (any other value MAY be ignored):

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

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

history.

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

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

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').

3.3 Specifying a Message Subject

A message stanza MAY contain a child <subject/> element specifying the subject of the message. The subject MUST NOT contain mixed content. Multiple <subject/> elements MAY be included, as long as each contains an 'xml:lang' attribute with a distinct value.

A message with a subject:

```
<message to='romeo@montague.net' from="juliet@capulet.com/balcony'>
  <subject>Imploring</subject>
  <body>Wherefore art thou, Romeo?</body>
</message>
```

3.4 Specifying a Conversation Thread

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

A threaded conversation:

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

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

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

3.5 Specifying a Message Body

A message stanza MAY (and often will) contain a child <body/> element specifying the main content of the message as CDATA. The body MUST NOT contain mixed content. If it is necessary to provide the main message content in an alternate form (e.g., encrypted using the public key infrastructure or formatted using XHTML), the alternate form MUST be contained in an appropriately-namespaced child of the message stanza, as defined for any such extended namespace. Multiple <body/> elements MAY be included, as long as each contains an 'xml:lang' attribute with a distinct value.

3.6 Specifying Additional Information

A message stanza MAY house an element containing content that extends the meaning of the message (e.g., an encrypted form of the message body). In common usage this child element is often the <x/> element but MAY be any element, as long as the 'xmlns' namespace declaration is something other than the streams namespace or the default namespace; this extended namespace defines all elements contained within the child element.

3.7 Message-Related Errors

If a message sent by a sender cannot be delivered, the entity discovering that fact (usually either the sender's or recipient's server) MUST return that message to the sender in a message stanza of type "error" along with an appropriate error message (for details regarding stanza error syntax, see XMPP Core [1])

A message-related error:

```
<message
    to='romeo@montague.net'
    from='juliet@capulet.com'
    type='error'>
  <body>Sleep dwell upon thine eyes.</body>
  <error class='address'>
    <stanza-condition xmlns='urn:ietf:rfc:xmppcore-rfc-number:stanzas'>
      <jid-not-found/>
    </stanza-condition>
  </error>
</message>
```

4. Exchanging Presence Information

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

Detailed information regarding the syntax of presence stanzas and their defined attributes and child elements may be found in XMPP Core [1].

4.1 Client and Server Responsibilities

When a client connects to a server, it SHOULD send an initial presence stanza to the server to express default availability. This presence stanza MUST possess no 'to' address (signalling that it is meant to be handled by the server on behalf of the user) and SHOULD have no type.

Upon receiving initial presence from a client, the server MUST send presence probes from the full JID (user@domain/resource) of the user to any remote entities that are subscribed to the user's presence (as represented in the user's roster) in order to determine if they are available. (The remote server is responsible for responding to the presence probe only when (1) the probing entity has been allowed to access the probed entity's presence, e.g., by server rules or user subscriptions, and (2) the probed entity is available. The probing entity's server then informs the probing entity of the probed entity's last known available presence, for all of the probed entity's resources if applicable.)

Throughout the active session of a connected resource, the server is responsible for broadcasting any changes in the availability status of the connected resource to the subscribed entities that are available, so that such entities are kept apprised of availability changes.

Finally, when a connected resource becomes unavailable, the server MUST notify all of the subscribed entities that are available, as well as any entities to which the user sent directed presence during the active session for that connected resource.

4.2 Sending Initial Presence

Upon authenticating, a client SHOULD send initial presence to its server indicating that the connected resource is available for communications. This presence stanza MUST have no 'to' address and SHOULD have no type.

Initial presence sent from client to server:

```
<presence/>
```

4.3 Specifying Availability Status

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

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

Availability status:

```
<presence>
  <show>away</show>
</presence>
```

4.4 Specifying Detailed Status Information

In conjunction with the <show/> element, a client MAY provide detailed status information by using the <status/> element. The content of this element is a natural-language description of the client's current availability status.

Detailed status information:

```
<presence>
  <show>dnd</show>
  <status>Busy fighting the Romans</status>
</presence>
```

4.5 Probing for Presence

A server MAY probe for the current presence of another entity. A user or client SHOULD NOT send presence stanzas of type 'probe'.

4.6 Sending Final Presence

Upon ending its session with a server, a client SHOULD gracefully become unavailable by sending a final presence stanza that is explicitly of type unavailable.

Sending final presence to express unavailable state:

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

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

The server MUST NOT depend on receiving final presence from a connected resource, since the resource may become unavailable unexpectedly. If a server detects that a resource has become unavailable for any reason (either gracefully or ungracefully), it MUST send <presence type="unavailable"/> to all online entities that are subscribed to the associated user's presence, as well as to any entities to which the user sent directed presence during the active session for that connected resource.

4.7 Determining When a Contact Went Offline

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

Requesting the last active time of a user:

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

If the entity requesting the time of last activity is an authorized subscriber to the user's presence, the server SHOULD return an IQ stanza of type 'result' with the number of seconds since the user was last active (subject to service provisioning and privacy configuration at a particular deployment):

Returning the last active time of a user:

```
<iq from='user@domain' type='result' to='subscriber@domain/resource'>
  <query seconds='76490' xmlns='jabber:iq:last'/>
</iq>
```

If the entity requesting the time of last activity is not an authorized subscriber to the user's presence, the server MUST return an IQ stanza of type 'error' with an error condition of forbidden:

Requester is forbidden to view the last active time of a user:

```
<iq from='user@domain' type='result' to='subscriber@domain/resource'>
  <query xmlns='jabber:iq:last'/>
  <error class='access'>
    <stanza-condition xmlns='urn:ietf:rfc:xmppcore-rfc-number:stanzas'>
      <forbidden/>
    </stanza-condition>
  </error>
</iq>
```

5. Managing Subscriptions

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

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

5.1 Requesting a Subscription

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

Sending a subscription request:

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

5.2 Handling a Subscription Request

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

Accepting a subscription request:

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

Denying a presence subscription request:

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

A user's server MUST NOT automatically accept subscription requests on the user's behalf. All subscription requests MUST be directed to the user's client. If there is no connected resource associated with the user when the subscription request is received, the user's server MUST store the subscription request offline for delivery when the user next becomes available.

5.3 Cancelling a Subscription from Another Entity

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

Cancelling a previously granted subscription request:

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

5.4 Unsubscribing from Another Entity's Presence

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

Unsubscribing from an entity's presence:

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

6. Managing One's Roster

In XMPP, one's contact list is called a roster. A roster is stored by the server on the user's behalf so that the user may access roster information from any connected resource.

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

6.1 Retrieving One's Roster on Login

Upon connecting to the server, a client MAY request the roster (however, because receiving the roster may not be desirable for all resources, e.g., a connection with limited bandwidth, the client's request for the roster is OPTIONAL). If a connected resource does not request the roster during a session, it SHOULD never receive presence subscriptions and associated roster pushes.

Client requests current roster from server:

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

Client receives roster from the server:

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

6.2 Adding a Roster Item

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

Client adds a new item:

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

The server is responsible for updating the roster information in persistent storage, and also for pushing that change out to all connected resources associated with the user using an IQ stanza of type "set" (this is referred to as a "roster push"). This "roster

push" enables all connected resources to remain in sync with the server-based roster information.

Server replies with an IQ result to the sending resource and pushes the updated roster information to all connected resources:

```
<iq
    from='capulet.com'
    to='juliet@capulet.com/balcony'
    type='result'
    id='roster_2'/>
<iq
    from='capulet.com'
    to='juliet@capulet.com/balcony'
    type='set'/>
  <query xmlns='jabber:iq:roster'>
    <item
        name='Nurse'
        jid='nurse@capulet.com'
        subscription='none'>
      <group>Servants</group>
    </item>
  </query>
</iq>
<iq
    from='capulet.com'
    to='juliet@capulet.com/chamber'
    type='set'/>
  <query xmlns='jabber:iq:roster'>
    <item
        name='Nurse'
        jid='nurse@capulet.com'
        subscription='none'>
      <group>Servants</group>
    </item>
  </query>
</iq>
```

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.

6.3 Deleting a Roster Item

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

Client removes an item:

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

As with adding a roster item, when deleting a roster item the server is responsible for updating the roster information in persistent storage, and also for initiating a "roster push" to all connected resources associated with the user.

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

7. Integration of Roster Items and Presence Subscriptions

7.1 Overview

Some level of integration between roster items and presence subscriptions is normally expected by instant messaging users. This section describes the level of integration that must be supported within XMPP IM.

There are four primary subscription states:

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

Each of these states is reflected in the roster of both the user and the contact, thus resulting in durable subscription states. The details regarding how these subscription states interact with roster items is explained in the following sub-sections.

As noted above, if a connected resource does not request the roster during a session, it SHOULD never receive presence subscriptions and the associated roster pushes.

7.2 User Subscribes to Contact

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

1. In preparation for being able to render the contact in the user's client interface and for the server to keep track of the subscription, the user's client SHOULD send an IQ stanza of type='set' in the 'jabber:iq:roster' namespace for the new roster item; the <item/> element MUST possess a 'jid' attribute, MAY possess a 'name' attribute, may contain one or more <group/> child, and MUST NOT posses a 'subscription' attribute:

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

2. The server then MUST (1) reply with an IQ stanza of type='result' and (2) initiate a "roster push" for the new roster item to all connected resources associated with this user, setting the subscription state set to 'none':

```
<iq
    type='result'
    to='user@domain/resource'
    id='int1'/>

<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='none'/>
  </query>
</iq>
```

3. In order to initiate the subscription, the user's client MUST then send a presence stanza of type='subscribe' to the contact:

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

4. The server MUST then initiate a second "roster push" to all connected resources associated with the user, 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' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='none'
        ask='subscribe'/>
  </query>
</iq>
```

Note: if the user did not create a roster item before sending the subscription request, the server MUST create one and send the above roster push to all of the user's connected resources.

5. The server MUST also deliver the presence stanza to the contact or route it to the contact's server for delivery to the contact, first stamping the stanza with the user's bare JID as the 'from' address:

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

6. If the contact is online (i.e., there is a connected resource associated with the contact's account), the contact must now decide whether or not to accept the subscription request. (If the contact is offline, the contact's server MUST store the subscription request offline for delivery when the contact next becomes available.) Here we will assume the "happy path" that the contact accepts the subscription, in which case the contact's client MAY send a roster set to the server specifying the desired nickname and group for the user, and MUST send a presence stanza of type='subscribed' to the user.

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

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

7. The contact's server MUST now initiate a "roster push" to all connected resources associated with the contact, containing a roster item for the user with the subscription state set to

'from':

```
<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='from'/>
  </query>
</iq>
```

8. As a result of the fact that the contact has accepted the subscription request, the user's server MUST (1) deliver the presence stanza of type='subscribed' from the contact to the user, and (2) initiate a "roster push" to all connected resources associated with the user, containing an updated roster item for the contact with the subscription type set to a value of "to":

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

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

From the perspective of the user, there is now a subscription to the contact; from the perspective of the contact, there is now a subscription from the user. The contact's server MUST now send the contact's current presence information to the user. (Note: If at this point the user sends another subscription request to the contact, the user's server MUST "swallow" that request and not send it on the contact.)

7.2.1 Alternate Flow

The above activity flow represents the "happy path" related to the user's subscription request to the contact. The main alternate flow occurs if the contact denies the user's subscription request; in order to deny the request, the contact's client MUST send a presence stanza of type "unsubscribed" to the user:

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

The user's server MUST then (1) deliver that presence stanza to the user and (2) initiate a "roster push" to all connected resources associated with the user, with the subscription attribute set to a value of "none":

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

<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='none'/>
  </query>
</iq>
```

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

7.3 Creating a Mutual Subscription

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

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

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

2. The contact's server MUST then initiate a "roster push" to all connected resources associated with the contact, 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:

```
<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='from'
        ask='subscribe'/>
  </query>
</iq>
```

3. If the user is online (i.e., there is a connected resource associated with the user's account), the user must now decide whether or not to accept the subscription request. (If the user is offline, the user's server MUST store the subscription request offline for delivery when the user next becomes available.) Here we will assume the "happy path" that the user accepts the subscription, in which case the user's client MUST send a presence stanza of type='subscribed' to the contact.

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

4. The user's server MUST then initiate a "roster push" to all connected resources associated with the user, containing a roster item for the contact with the subscription attribute set to a value of "both":

```
<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='both'/>
  </query>
</iq>
```

5. As a result of the fact that the user has accepted the subscription request, the contact's server MUST (1) deliver the presence stanza of type='subscribed' from the user to the contact, and (2) initiate a "roster push" to all connected resources associated with the contact, containing an updated roster item for the user with the subscription type set to a value of "both":

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

<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
  <item
      jid='user@domain'
      name='SomeUser'
      subscription='both'/>
  </query>
</iq>
```

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

7.3.1 Alternate Flow

The above activity flow represents the "happy path" related to the contact's subscription request to the user. The main alternate flow occurs if the user denies the contact's subscription request; in order to deny the request, the user's client MUST send a presence stanza of type "unsubscribed" to the contact:

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

The contact's server MUST then (1) deliver that presence stanza to the contact and (2) initiate a "roster push" to all connected resources associated with the contact, with the subscription attribute set to a value of "from" and with no 'ask' attribute:

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

<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='from'/>
  </query>
</iq>
```

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

7.4 Unsubscribing

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

7.4.1 Case #1: Subscription Type 'to'

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

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

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

2. As a result, the user's server MUST send a "roster push" to each connected resource associated with the user, containing a roster item for the contact with the 'subscription' attribute set to a value of "from":

```
<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='from'/>
  </query>
</iq>
```

3. The user's server MUST also route the unsubscribe "command" to the contact's server:

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

4. The contact's server MUST initiate a "roster push" to all connected resources associated with the contact, containing a roster item for the user with the 'subscription' attribute set to a value of "to" (if the contact is offline, the contact's server MUST modify the roster item and send that modified item the next time the contact requests the roster); the contact's server SHOULD also deliver the unsubscribe command to the contact:

```
<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='to'/>
  </query>
</iq>

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

5. The contact's server then MUST send unavailable presence from the contact to the user and MAY send a presence stanza of type "unsubscribed" to the user:

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

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

6. As a result, the user's server MUST deliver the unavailable presence from the user to the contact and (if received) the presence stanza of type "unsubscribed" from the contact to the user,

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

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

7.4.2 Case #2: Subscription Type 'both'

In the second case, the user has a subscription to the contact and the contact also has a subscription to the user.

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

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

2. As a result, the user's server MUST send a "roster push" to each connected resource associated with the user, containing a roster item for the contact with the 'ask' attribute set to unsubscribe to indicate that the unsubscribe is pending:

```
<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='both'
        ask='unsubscribe'/>
  </query>
</iq>
```

3. Since the unsubscribe command does not normally need to be approved by the contact, the contact's server then SHOULD auto-reply on behalf of the contact by sending a presence stanza of type "unsubscribed" to the user; additionally, it MUST send unavailable presence from the contact to the user:

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

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

4. The contact's server also MUST initiate a "roster push" to all connected resources associated with the contact, containing a roster item for the user with the 'subscription' attribute set to a value of "to" (if the contact is offline, the contact's server MUST modify the roster item and send that modified item the next time the contact requests the roster):

```
<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='to'/>
  </query>
</iq>
```

5. As a result, the user's server MUST (1) deliver the presence stanza of type='unsubscribed' from the contact to the user, (2) initiate a "roster push" to all connected resources associated with the user, containing an updated roster item for the contact with the subscription type set to a value of "from" and with no 'ask' attribute, and (3) deliver the unavailable presence from

the user to the contact:

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

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

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

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

7.5 Cancelling a Subscription

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

7.5.1 Case #1: Subscription Type 'from'

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

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

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

2. As a result, the contact's server MUST (1) send a "roster push" to each connected resource associated with the contact, containing a roster item for the user with the 'subscription' attribute set to a value of "none", and (2) send unavailable presence from the contact to the user:

```
<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='none'/>
  </query>
</iq>

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

3. The user's server MUST then initiate a "roster push" to all connected resources associated with the user, containing a roster item for the contact with the 'subscription' attribute set to a value of "none" (if the user is offline, the user's server MUST modify the roster item and send that modified item the next time the user requests the roster); additionally, it MUST also deliver the unavailable presence from the contact to the user:

```
<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='none'/>
  </query>
</iq>

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

7.5.2 Case #2: Subscription Type 'both'

In the second case, the user has a subscription to the contact and the contact also has a subscription to the user.

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

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

2. As a result, the contact's server MUST (1) send a "roster push" to each connected resource associated with the contact, containing a roster item for the user with the 'subscription' attribute set to a value of "to", and (2) send unavailable presence from the contact to the user:

```
<iq type='set' to='contact@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='user@domain'
        name='SomeUser'
        subscription='to'/>
  </query>
</iq>

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

3. The user's server MUST then initiate a "roster push" to all connected resources associated with the user, containing a roster item for the contact with the 'subscription' attribute set to a value of "from" (if the user is offline, the user's server MUST modify the roster item and send that modified item the next time the user requests the roster); additionally, it MUST also deliver the unavailable presence from the contact to the user:

```
<iq type='set' to='user@domain/resource'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        subscription='from'/>
  </query>
</iq>

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

Note: Obviously this does not result in removal of the roster item

from the contact's roster, and the contact still has a subscription to the user's presence. In order to more completely cancel a mutual subscription and fully remove the roster item from the contact's roster, the contact should update the roster item with subscription='remove' as defined in Section 7.6.

7.6 Removing a Roster Item and Cancelling All Subscriptions

Because there may be many steps involved in completely removing a roster item and reverting the subscription state to "none", XMPP IM includes a "shortcut" method for doing so. The process may be initiated by either a contact or a user no matter what the current subscription state is, by means of sending a roster set with the subscription attribute set to a value of "remove".

For example, a user may send the following XML:

```
<iq type='set'>
  <query xmlns='jabber:iq:roster'>
    <item
        jid='contact@domain'
        name='MyContact'
        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 MUST automatically cancel any existing presence subscription between the user and the contact by sending presence stanzas of type "unsubscribe" and "unsubscribed" from the user to the contact.

A contact may also send such a command, resulting in the same type of system behavior.

8. Blocking Communication

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

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

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

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

8.1 Syntax

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

processing order.

The syntax is as follows:

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

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

If the type is "group", then the 'value' attribute MUST contain the name of a group in the user's roster.

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

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

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

The 'order' attribute MUST be included and its value MUST be a non-negative integer that is unique among all items in the list. (If a client attempts to create or update a list with non-unique order values, the server MUST return to the client a <bad-request/> error of class "format" in the 'urn:ietf:rfc:xmppcore-rfc-number:stanzas' namespace.)

Within the 'jabber:iq:privacy' namespace, the <query/> child of a client-generated IQ stanza of type "set" MUST NOT include more than one child element (i.e., the stanza must contain only one <active/> element, one <default/> element, or one <list/> element); if a client violates this rule, the server MUST return to the client a <bad-

request/> error of class "format" in the 'urn:ietf:rfc:xmppcore-rfc-number:stanzas' namespace.)

When a client adds or updates a privacy list, the <list/> element MUST contain at least one <item/> child element.

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

8.2 Business Rules

The active list affects only the session/resource for which it is activated, and only for the duration of the session. If a stanza is addressed to a specific resource, only the active list for that session is processed (i.e., the default list is ignored).

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.

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.

Privacy lists SHOULD be the first routing and delivery rule applied by a server, trumping the other rules specified in Section 9.

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

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

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

When a user updates the definition for a list or adds a new list (whether or not it is active), the server SHOULD NOT "push" that information out to all connected resources associated with the user's account, as is done for rosters. If a client or user wants to retrieve the current privacy list information, it SHOULD request the relevant list.

8.3 Retrieving One's Privacy Lists

Client requests names of privacy lists from server:

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

Server sends names of privacy lists to client, including default list and active list:

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

Client requests a privacy list from server:

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

Server sends a privacy list to client:

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

Client requests another privacy list from server:

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

Server sends another privacy list to client:

```
<iq type='result' id='getlist3' to='romeo@montague.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>
```

Client requests yet another privacy list from server:

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

Server sends yet another privacy list to client:

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

In this example, the user has three lists: (1) 'public', which allows communications from everyone except one specific entity; (2) 'private', which allows communications only from contacts who have a bi-directional subscription with the user; and (3) 'special', which allows communications only from three specific entities. The active list currently being applied by the server is the 'private' list.

If the user attempts to retrieve a list but a list by that name does

not exist, the server MUST return an application-specific "list not found" error to the user:

Client attempts to retrieve non-existent list:

```
<iq type='result' id='getlist5'>
  <query xmlns='jabber:iq:privacy'>
    <list name='The Empty Set'/>
  </query>
  <error class='app'>
    <privacy-condition xmlns='jabber:iq:privacy:error'>
      <list-not-found/>
    </privacy-condition>
  </error>
</iq>
```

The user may 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 an application-specific <too-many-lists/> error to the user.

Client attempts to retrieve more than one list:

```
<iq type='result' id='getlist6'>
  <query xmlns='jabber:iq:privacy'>
    <list name='public'/>
    <list name='private'/>
    <list name='special'/>
  </query>
  <error class='app'>
    <privacy-condition xmlns='jabber:iq:privacy:error'>
      <too-many-lists/>
    </privacy-condition>
  </error>
</iq>
```

8.4 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 scoped 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.

Client requests change of active list:

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

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

Server acknowledges success of active list change:

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

If the user attempts to set an active list but a list by that name does not exist, the server MUST return an application-specific "list not found" error to the user:

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

```
<iq type='result' id='active2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='The Empty Set'/>
  </query>
  <error class='app'>
    <privacy-condition xmlns='jabber:iq:privacy:error'>
      <list-not-found/>
    </privacy-condition>
  </error>
</iq>
```

In order to decline the use of any active list (i.e., to use the domain's stanza routing rules for the duration of the session), a user MUST send an empty <active/> element with no name.

Client declines the use of active lists:

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

8.5 Managing the Default List

In order to change the default list associated with an account, the user MUST send an IQ stanza of type 'set' with a <query/> element scoped 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.

Client requests change of default list:

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

Server acknowledges success of default list change:

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

If the user attempts to set a default list but a list by that name does not exist, the server MUST return an application-specific "list not found" error to the user:

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

```
<iq type='result' id='default2'>
  <query xmlns='jabber:iq:privacy'>
    <list name='The Empty Set'/>
  </query>
  <error class='app'>
    <privacy-condition xmlns='jabber:iq:privacy:error'>
      <list-not-found/>
    </privacy-condition>
  </error>
</iq>
```

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

Client declines the use of the default list:

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

8.6 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 scoped 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"); the same protocol is used to create a new list.

Client edits a privacy list:

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

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

Server acknowledges success of list edit:

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

In this example, the user has added one additional entity to the "blacklist" portion of this privacy list.

8.7 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 scoped by the 'jabber:iq:privacy' namespace that contains one empty <list/> child elements possessing a 'name' attribute whose value is set to the list name the user would like to remove.

Client removes a privacy list:

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

Server acknowledges success of list removal:

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

If a user attempts to remove an active list or the default list, the server MUST return an application-specific <cannot-be-removed/> error to the user.

If the user attempts to remove a list but a list by that name does not exist, the server MUST return an application-specific <list-not-found/> error to the user.

If the user attempts to remove more than one list in the same request, the server MUST return an application-specific <too-many-lists/> error to the user.

8.8 Blocking Messages

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

User blocks based on JID:

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

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

User blocks based on roster group:

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

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

User blocks based on subscription type:

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

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

User blocks globally:

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

As a result of creating and applying the foregoing list, the user

will not receive messages from any other users.

8.9 Blocking Inbound Presence Notifications

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

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

User blocks based on JID:

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

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

User blocks based on roster group:

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

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

User blocks based on subscription type:

```
<iq type='set' id='presin3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presin-sub-example'>
      <item type='subscription' value='none' action='deny' order='9'>
        <presence-in/>
      </item>
    </list>
  </query>
</iq>
```

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

User blocks globally:

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

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

8.10 Blocking Outbound Presence Notifications

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

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

User blocks based on JID:

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

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

User blocks based on roster group:

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

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

User blocks based on subscription type:

```
<iq type='set' id='presout3'>
  <query xmlns='jabber:iq:privacy'>
    <list name='presout-sub-example'>
      <item type='subscription' value='none' action='deny' order='17'>
        <presence-out/>
      </item>
    </list>
  </query>
</iq>
```

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

User blocks globally:

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

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

8.11 Blocking IQs

Server-side privacy lists enable a user to block incoming IQ requests of type "get" or "set" from other users based on the other user's JID, roster group, or subscription status, or globally. The following examples illustrate the required protocol.

User blocks based on JID:

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

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

User blocks based on roster group:

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

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

User blocks based on subscription type:

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

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

User blocks globally:

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

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

8.12 Blocking All Communication

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

User blocks based on JID:

```
<iq type='set' id='all1'>
  <query xmlns='jabber:iq:privacy'>
    <list name='all-jid-example'>
      <item type='jid' value='tybalt@capulet.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 presence to, the user with the specified JID.

User blocks based on roster group:

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

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

User blocks based on subscription type:

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

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

User blocks globally:

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

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

8.13 Blocked Entity Attempts to Communicate with User

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

If a blocked entity attempts to send an IQ stanza of type "get" or "set" to the user, the user's server MUST return to the sending entity a <feature-not-implemented/> error of class "recipient" in the 'urn:ietf:rfc:xmppcore-rfc-number:stanzas' namespace, 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.

Blocked entity attempts to send IQ get:

```
<iq
    type='get'
    to='romeo@montague.net'
    from='tybalt@capulet.com/pda'
    id='probing1'>
  <query xmlns='jabber:iq:version'/>
</iq>
```

Server returns error to blocked entity:

```
<iq
    type='error'
    from='romeo@montague.net'
    to='tybalt@capulet.com/pda'
    id='probing1'>
  <query xmlns='jabber:iq:version'/>
  <error class='recipient'>
    <stanza-condition xmlns='urn:ietf:rfc:xmppcore-rfc-number:stanzas'>
      <feature-not-implemented/>
    </stanza-condition>
  </error>
</iq>
```

8.14 Higher-Level Heuristics

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

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

- accept communications from all JIDs in my roster (i.e., listing each JID as a separate list item), but deny communications with everyone else
- accept communications from any user who is in one of the groups that make up my roster (i.e., listing each group as a separate list item), but deny communications from everyone else
- accept communications from any user with whom I have a subscription of 'both' or 'to' or 'from' (i.e., listing each subscription value separately), but deny communications from everyone else
- deny communications from anyone whose subscription state is 'none'

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

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

9. Routing and Delivery Rules

9.1 Client Generation of To Addresses

Many XMPP stanzas possess a 'to' address that specifies the intended recipient of the stanza. When a server receives a stanza possessing a 'to' attribute from a connected client, it is responsible for either directly delivering the stanza to the intended recipient (if the recipient is served by the same server) or for routing the stanza to another server (if the recipient is not served by the same server). (This does not necessary imply that the recipient is on a different domain from the sender, since one server could host multiple domains.) If delivery to the intended recipient is unsuccessful or the recipient's server cannot be contacted, the sender's server is responsible for returning an error to the sender; if the recipient's server can be contacted but delivery by the recipient's server to the recipient is unsuccessful, the recipient's server is responsible for returning an error to the sender by way of the sender's server.

If a client does not specify a 'to' address on a stanza, it is implied that the stanza is meant to be handled by the sender's server on behalf of the sender. Although this functionality is normally not used in the case of message stanzas, it is quite common with regard to both presence and IQ stanzas. In the case of presence stanzas, the user's presence information is normally sent from the client to the user's server without a 'to' attribute, and subsequently broadcasted to all entities that are subscribed to the sender's presence information using a classic publish-subscribe model. In the case of IQ stanzas, requests in many extended namespaces (e.g., jabber:iq:roster) are normally sent from the client to the server without a 'to' attribute, and handled by the server on behalf of the user (e.g., to manage roster information stored by the server).

If a user's client would like to request information about the user's server itself (e.g., for the purpose of service discovery), it MUST include the server's JID in the 'to' address of the IQ request.

9.2 Server Handling of XML Stanzas

Any appropriate privacy rules (Section 8) SHOULD be applied by the server first. Following the application of any privacy rules, XML stanzas that are not handled directly by a server (e.g., for the purpose of data storage or rebroadcasting) MUST be routed or delivered to the intended recipient of the stanza as represented by a JID in the 'to' attribute. The following rules apply:

o If the JID contains a resource identifier (to="user@domain/

resource"), the stanza is delivered first to the resource that exactly matches the resource identifier.

- If the JID contains a resource identifier and there are no matching resources, but there are other connected resources associated with the user, then message stanzas are further processed as if no resource is specified (see next item). For all other stanzas, the server SHOULD return to the sender a <recipient-unavailable/> error of class "recipient" in the 'urn:ietf:rfc:xmppcore-rfc-number:stanzas' namespace.

- If the JID contains only a user@domain and there is at least one connected resource available for the user, the server SHOULD deliver the stanza to an appropriate resource based on the availability state, priority, and connect time of the connected resource(s). (For instance, the server MAY deliver the stanza to the resource with the highest value for the <priority/> element, and decide between resources of equal priority based on most recent connect time or most recent activity time; however, all such rules are implementation-specific.)

- If the JID contains only a user@domain and there are no connected resources available for the user (e.g., an IM user is offline), the server MAY choose to store the stanza on behalf of the user and deliver the stanza when a resource becomes available for that user. If offline storage is not enabled, the server MUST return to the sender a <service-unavailable/> error of class "server" in the 'urn:ietf:rfc:xmppcore-rfc-number:stanzas' namespace. Note: offline storage is not defined in XMPP since it is a matter of implementation.

10. IANA Considerations

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

11. Security Considerations

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

References

[1] Saint-Andre, P. and J. Miller, "XMPP Core (draft-ietf-xmpp-core-06, work in progress)", March 2003.

[2] Day, M., Aggarwal, S., Mohr, G. and J. Vincent, "A Model for Presence and Instant Messaging", RFC 2779, February 2000, <http://www.ietf.org/rfc/rfc2779.txt>.

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

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

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

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

Authors' Addresses

Peter Saint-Andre
Jabber Software Foundation

EMail: stpeter@jabber.org
URI: http://www.jabber.org/people/stpeter.php

Jeremie Miller
Jabber Software Foundation

EMail: jeremie@jabber.org
URI: http://www.jabber.org/people/jer.php

Appendix A. vCards

Sections 3.1.3 and 4.1.4 of RFC 2779 [2] require that it be possible to retrieve non-IM contact information for other users (e.g., telephone number or email address). An XML representation of the vCard specification defined in RFC 2426 [6] is in common use within the Jabber community to provide such information. Documentation of this protocol is maintained by the Jabber Software Foundation [3] at <http://www.jabber.org/protocol/>.

Appendix B. XML Schemas

B.1 jabber:iq:auth

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

  <xs:element name='query'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='username' maxOccurs='1'/>
        <xs:element ref='resource' minOccurs='0' maxOccurs='1'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name='username' type='xs:string'/>
  <xs:element name='resource' type='xs:string'/>

</xs:schema>
```

B.2 jabber:iq:auth:error

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

  <xs:element name='auth-condition'>
    <xs:complexType>
      <xs:choice maxOccurs='1'>
        <xs:element ref='no-resource-provided'/>
        <xs:element ref='bad-resource-format'/>
        <xs:element ref='resource-conflict'/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name='no-resource-provided' type='xs:string'/>
  <xs:element name='bad-resource-format' type='xs:string'/>
  <xs:element name='resource-conflict' type='xs:string'/>

</xs:schema>
```

B.3 jabber:iq:last

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

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

</xs:schema>
```

B.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' maxOccurs='1'/>
          <xs:element ref='default' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='list' minOccurs='0' maxOccurs='unbounded'/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

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

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

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

    <xs:element name='item'>
      <xs:complexType>
        <xs:sequence>
          <xs:element ref='iq' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='message' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='presence-in' minOccurs='0' maxOccurs='1'/>
          <xs:element ref='presence-out' minOccurs='0' maxOccurs='1'/>
        </xs:sequence>
        <xs:attribute name='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:nonNegativeInteger' 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:element name='iq'/>
    <xs:element name='message'/>
    <xs:element name='presence-in'/>
    <xs:element name='presence-out'/>

  </xs:schema>
```

B.5 jabber:iq:privacy:error

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

  <xs:element name='privacy-condition'>
    <xs:complexType>
      <xs:choice maxOccurs='1'>
        <xs:element ref='list-not-found'/>
        <xs:element ref='too-many-lists'/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name='list-not-found' type='xs:string'/>
  <xs:element name='too-many-lists' type='xs:string'/>

</xs:schema>
```

B.6 jabber:iq:roster

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

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

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

```
        <xs:attribute name='subscription' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:string'>
              <xs:enumeration value='to'/>
              <xs:enumeration value='from'/>
              <xs:enumeration value='both'/>
              <xs:enumeration value='none'/>
              <xs:enumeration value='remove'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name='ask' use='optional'>
          <xs:simpleType>
            <xs:restriction base='xs:string'>
              <xs:enumeration value='subscribe'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>

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

  </xs:schema>
```

Appendix C. Provisional Namespace Names

Note to RFC editor: prior to publication, the string 'xmppcore-rfc-number' must be replaced in all instances by the RFC number assigned to draft-ietf-xmpp-core. (In addition, please remove this appendix prior to publication.)

Appendix D. Revision History

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

D.1 Changes from draft-ietf-xmpp-im-05

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

D.2 Changes from draft-ietf-xmpp-im-04

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

D.3 Changes from draft-ietf-xmpp-im-03

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

D.4 Changes from draft-ietf-xmpp-im-02

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

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

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

D.5 Changes from draft-ietf-xmpp-im-01

- Made numerous small editorial changes.

D.6 Changes from draft-ietf-xmpp-im-00

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

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

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

- Specified the algorithm for digest passwords.

D.7 Changes from draft-miller-xmpp-im-02

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

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

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

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

- Noted that the only watchers are subscribers.

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

Full Copyright Statement

Acknowledgement

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