# XEP-0207: XMPP Eventing via Pubsub

Peter Saint-Andre
mailto:xsf@stpeter.im
xmpp:peter@jabber.org
http://stpeter.im/

2007-04-01
Version 1.0

| Status | Type | Short Name |
| --- | --- | --- |
| Active | Humorous | N/A |

This document specifies semantics for using the XMPP publish-subscribe protocol to handle generic XMPP events (including presence, one-to-one messaging, and groupchat).

# Legal

## Copyright



## Permissions



## Warranty



## Liability



## Conformance

This XMPP Extension Protocol has been contributed in full conformance with the XSF’s Intellectual Property Rights Policy (a copy of which can be found at <https://xmpp.org/about/xsf/ipr-policy> or obtained by writing to XMPP Standards Foundation, P.O. Box 787, Parker, CO 80134 USA).

## Contents


1 Introduction 1

2 Concepts and Approach 1

3 Presence 2

4 Rosters and Presence Subscriptions 4

5 Multi-User Chat 7

6 One-to-One Messaging 10

7 Conclusion 12

8 Security Considerations 12

9 IANA Considerations 13

10 XMPP Registrar Considerations 13

11 XML Schema 13

12 Acknowledgements 13

# 1 Introduction

Personal Eventing Protocol (XEP-0163) [1] (PEP) introduced the concept of "eventing" into the Extensible Messaging and Presence Protocol (see XMPP Core [2]). But PEP merely scratched the surface of eventing technologies based on the XMPP Publish-Subscribe (XEP-0060) [3] extension. This document extends the eventing concept to its ultimate conclusion: the ability to communicate all XMPP semantics via pubsub.

# 2 Concepts and Approach

Jabber technologies as invented by Jeremie Miller started out as a relatively lightweight XML messaging transport, but they have become unnecessarily -- even ridiculously -- bloated over the years. Formalization of the core Jabber protocols as the Extensible Messaging and Presence Protocol (XMPP) within the IETF seemed like a good idea at the time, but the extensible nature of the core protocols has tempted the developer community to extend XMPP six ways from Sunday. The result has been a plethora of different conceptual models for various extensions, such as Multi-User Chat (XEP-0045) [4] for multi-user communication and Ad-Hoc Commands (XEP-0050) [5] for structured entity-to-entity request/response semantics. These different models are inelegant and unnecessary. Indeed, even the inclusion of three different basic packet types (presence, message, and IQ) in the core protocol is overkill.
We can do better. In fact, we can reduce all the communication types and styles that are currently defined within the XMPP ecosystem to one model: publish-subscribe as specified in XEP-0060.
Consider:

- It has often been observed that presence (see XMPP IM [6]) is a form of publish-subscribe. [7]
- The primitive XMPP "roster" can be easily implemented using the pubsub permissions model.
- Multi-user chat too is a kind of publish-subscribe, since a single "publish" to the room results in multiple notifications to the room occupants, who are really subscribers to an information node.

[1] XEP-0163: Personal Eventing Protocol <https://xmpp.org/extensions/xep-0163.html>.
[2] RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core <http://tools.ietf.org/html/rfc6120>.
[3] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.
[4] XEP-0045: Multi-User Chat <https://xmpp.org/extensions/xep-0045.html>.
[5] XEP-0050: Ad-Hoc Commands <https://xmpp.org/extensions/xep-0050.html>.
[6] RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence <http://tools.ietf.org/html/rfc6121>.
[7] See, for instance, <http://mail.jabber.org/pipermail/xmppwg/2003-February/000636.html>.

- Even one-to-one messaging is just another example of publish-subscribe (in fact it is a special case of multi-user chat).

The remainder of this document will prove beyond a doubt that the older, multiple approaches are obsolete, and that there is indeed one model that serves all our needs: pubsub.

## 3 Presence

Presence simply is pubsub, since it follows the classic "observer" pattern: multiple subscribers receive notifications whenever the publisher (typically an end user) generates an event related to network availability. Currently in XMPP this is done with the <presence/> stanza, which serves as a kind of pubsub primitive (though only for availability information). For example, Juliet may log into the capulet.lit server and send presence:

Listing 1: Presence Update

```
<presence from='juliet@capulet.lit/balcony'>
  <status>Moping</status>
</presence>
```

The capulet.lit server will then send notifications to all of the users who have subscribed to Juliet's presence:

Listing 2: Presence Notifications

```
<presence from='juliet@capulet.lit/balcony' to='romeo@montague.lit/
    mobile'>
  <status>Moping</status>
</presence>

<presence from='juliet@capulet.lit/balcony' to='nurse@capulet.lit/
    chamber'>
  <status>Moping</status>
</presence>

<presence from='juliet@capulet.lit/balcony' to='benvolio@montague.lit/
    pda'>
  <status>Moping</status>
</presence>

[etc.]
```

But the same functionality can be implemented more elegantly using pubsub:

Listing 3: Presence Publish

```
<iq from='juliet@capulet.lit/balcony' type='set' id='pres1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='presence'>
      <item>
        <presence from='juliet@capulet.lit/balcony' xmlns='
            jabber:client'>
          <status>Moping</status>
        </presence>
      </item>
    </publish>
  </pubsub>
</iq>

<iq to='juliet@capulet.lit/balcony' type='result' id='pres1'/>
```

The server (here implementing PEP) then sends notifications to the subscribers:

Listing 4: Presence Notifications via Pubsub

```
<message from='juliet@capulet.lit'
         to='romeo@montague.lit/mobile'
         id='presfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='presence'>
      <item>
        <presence from='juliet@capulet.lit/balcony' xmlns='
            jabber:client'>
          <status>Moping</status>
        </presence>
      </item>
    </items>
  </event>
</message>

<message from='juliet@capulet.lit'
         to='nurse@capulet.lit/chamber'
         id='presfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='presence'>
      <item>
        <presence from='juliet@capulet.lit/balcony' xmlns='
            jabber:client'>
          <status>Moping</status>
        </presence>
      </item>
    </items>
  </event>
</message>
```

```
<message from='juliet@capulet.lit'
         to='benvolio@montague.lit/pda'
         id='presfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='presence'>
      <item>
        <presence from='juliet@capulet.lit/balcony' xmlns='
            jabber:client'>
          <status>Moping</status>
        </presence>
      </item>
    </items>
  </event>
</message>
```

It is true that in this case the packets are significantly larger in the pubsub realization than in the old-fashioned presence realization. This is the price of elegance. Implementations SHOULD use native Transport Layer Security compression (see RFC 5246 [8]) or Stream Compression (XEP-0138) [9] at the application layer to conserve bandwidth.

## 4 Rosters and Presence Subscriptions

The original Jabber technologies included a kind of Buddy List ™ (called the "roster"). But the roster is simply a list of the entities that are subscribed to a user's presence. Occam's Razor would indicate that it is foolish to implement two concepts (presence subscription and roster) when one will solve the problem at hand. In XMPP Eventing via Pubsub, there is no need for a specialized "roster" since the same information is represented in the list of entities who are subscribed to the user's "presence" node in pubsub/PEP.
Using XMPP Eventing via Pubsub also cleans up the syntax for presence subscription management, which currently uses four specialized values of the <presence/> element's 'type' attribute: "subscribe", "subscribed", "unsubscribe", and "unsubscribed".
Thus for example a presence subscription request is currently made by sending the following presence stanza:

Listing 5: Presence Subscription Request

```
<presence from='bard@shakespeare.lit' to='juliet@capulet.lit' type='
    subscribe'/>
```

And that request is then delivered to the intended recipient:

Listing 6: Presence Subscription Request

[8] RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 <http://tools.ietf.org/html/rfc5246>.
[9] XEP-0138: Stream Compression <https://xmpp.org/extensions/xep-0138.html>.

```
<presence from='bard@shakespeare.lit' to='juliet@capulet.lit' type='
    subscribe'/>
```

In order to approve the subscription request, the user sends a presence stanza of type "subscribed":

Listing 7: Presence Subscription Approval

```
<presence from='juliet@capulet.lit' to='bard@shakespeare.lit' type='
    subscribed'/>
```

At this point the user's server also creates an entry on the user's roster for the relevant contact and pushes that entry to the user:

Listing 8: Roster Push

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

Observe how much more elegant it is to use XMPP Eventing via Pubsub:

Listing 9: Pubsub Subscription Request

```
<iq type='set'
    from='bard@shakespeare.lit/globe'
    to='juliet@capulet.lit'
    id='sub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe
        node='presence'
        jid='bard@shakespeare.lit'/>
  </pubsub>
</iq>
```

The pubsub service then sends an authorization request to the user:

Listing 10: Service sends authorization request to node owner

```
<message to='juliet@capulet.lit' from='capulet.lit' id='approve1'>
  <x xmlns='jabber:x:data' type='form'>
    <title>PubSub subscriber request</title>
    <instructions>
```

```
      To approve this entity&apos;s subscription request,
      click the OK button. To deny the request, click the
      cancel button.
    </instructions>
    <field var='FORM_TYPE' type='hidden'>
      <value>http://jabber.org/protocol/pubsub#subscribe_authorization
         </value>
    </field>
    <field var='pubsub#node' type='text-single' label='Node␣ID'>
      <value>presence</value>
    </field>
    <field var='pusub#subscriber_jid' type='jid-single' label='
       Subscriber␣Address'>
      <value>bard@shakespeare.lit</value>
    </field>
    <field var='pubsub#allow' type='boolean'
           label='Allow␣this␣JID␣to␣subscribe␣to␣this␣pubsub␣node?'>
      <value>false</value>
    </field>
  </x>
</message>
```

In order to approve the request, the owner shall submit the form and set the "pubsub#allow" field to a value of "1" or "true"; for tracking purposes the message MUST reflect the 'id' attribute originally provided.

Listing 11: User approves subscription request

```
<message from='juliet@capulet.lit/balcony' to='montague.lit' id='
   approve1'>
  <x xmlns='jabber:x:data' type='submit'>
    <field var='FORM_TYPE' type='hidden'>
      <value>http://jabber.org/protocol/pubsub#subscribe_authorization
         </value>
    </field>
    <field var='pubsub#node'>
      <value>presence</value>
    </field>
    <field var='pubsub#subscriber_jid'>
      <value>bard@shakespeare.lit</value>
    </field>
    <field var='pubsub#allow'>
       <value>true</value>
    </field>
  </x>
</message>
```

Simple. Elegant. And no need for a roster! The pubsub approach is bit more verbose, but then again clients and servers should implement and deploy stream compression if they are really

worred about bandwidth usage.

## 5 Multi-User Chat

The existing groupchat protocol for XMPP overloads the <presence/> stanza for temporary ”subscriptions” to a virtual room and uses the <message/> stanza (with a special type of ”groupchat”) to communicate information to multiple room occupants. Sound familiar? It’s just another form of pubsub!
In groupchat, a user joins a room by sending presence to ”room@service/nick”:

Listing 12: Groupchat Join

```
<presence from='juliet@capulet.lit/balcony' to='characters@chat.
    shakespeare.lit/JC'/>
```

But on the pubsub model that is merely a temporary subscription, which can be handled quite elegantly as so:

Listing 13: Groupchat Join via Pubsub

```
<iq type='set'
    from='juliet@capulet.lit/balcony'
    to='chat.shakespeare.lit'
    id='sub2'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe
        node='characters'
        jid='juliet@capulet.lit/balcony'/>
    <options node='characters' jid='juliet@capulet.lit/balcony'>
      <x xmlns='jabber:x:data' type='submit'>
        <field var='FORM_TYPE' type='hidden'>
          <value>http://jabber.org/protocol/pubsub#subscribe_options</
              value>
        </field>
        <field var='roomnick'><value>JC</value></field>
      </x>
    </options>
  </pubsub>
</iq>
```

In groupchat, room occupants can send messages to all other occupants via the <message/> stanza:

Listing 14: Groupchat Message

```
<message from='juliet@capulet.lit/balcony' to='characters@chat.
    shakespeare.lit' type='groupchat'>
  <body>hi</body>
</message>
```

The groupchat service then "reflects" that message to all the occupants:

Listing 15: Groupchat Message Delivery

```
<message from='characters@chat.shakespeare.lit/JC' to='
    bard@shakespeare.lit/globe' type='groupchat'>
  <body>hi</body>
</message>

<message from='characters@chat.shakespeare.lit/JC' to='romeo@montague.
    lit/mobile' type='groupchat'>
  <body>hi</body>
</message>

<message from='characters@chat.shakespeare.lit/JC' to='
    benvolio@montague.lit/pda' type='groupchat'>
  <body>hi</body>
</message>
```

But on the pubsub model that is merely a publish resulting in multiple notifications, which can be handled quite elegantly as so:

Listing 16: Groupchat Publish

```
<iq from='juliet@capulet.lit/balcony' to='chat.shakespeare.lit' type='
    set' id='gc1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='characters'>
      <item>
        <message xmlns='jabber:client'>
          <body>hi</body>
        </message>
      </item>
    </publish>
  </pubsub>
</iq>

<iq from='chat.shakespeare.lit' to='juliet@capulet.lit/balcony' type='
    result' id='gc1'/>
```

The service then sends notifications to all the node subscribers:

Listing 17: Groupchat Notifications

```
<message from='chat.shakespeare.lit'
         to='bard@shakespeare.lit/globe'
         id='mucfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='characters'>
      <item>
        <message xmlns='jabber:client'>
          <body>hi</body>
        </message>
      </item>
    </items>
  </event>
  <addresses xmlns='http://jabber.org/protocol/address'>
    <address type='replyto' jid='juliet@capulet.lit/balcony'/>
  </addresses>
</message>

<message from='chat.shakespeare.lit'
         to='romeo@montague.lit/mobile'
         id='mucfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='characters'>
      <item>
        <message xmlns='jabber:client'>
          <body>hi</body>
        </message>
      </item>
    </items>
  </event>
  <addresses xmlns='http://jabber.org/protocol/address'>
    <address type='replyto' jid='juliet@capulet.lit/balcony'/>
  </addresses>
</message>

<message from='chat.shakespeare.lit'
         to='benvolio@montague.lit/pda'
         id='mucfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='characters'>
      <item>
        <message xmlns='jabber:client'>
          <body>hi</body>
        </message>
      </item>
    </items>
  </event>
  <addresses xmlns='http://jabber.org/protocol/address'>
    <address type='replyto' jid='juliet@capulet.lit/balcony'/>
  </addresses>
```

```
</message>
```

Here again the pubsub approach is slightly more verbose, but that's what stream compression is for.

## 6 One-to-One Messaging

It's really rather silly that XMPP includes two different models for messaging, one for groupchat and the other for one-to-one messages. Pubsub solves that problem by using one model for everything. In order to exchange messages, one of the parties simply creates a pubsub node with a whitelist model and adds the other person as a publisher (it may also be necessary to add the other party to the whitelist):

Listing 18: Creating a One-to-One Messaging Node

```
<iq type='set'
    from='juliet@capulet.lit/balcony'
    to='pubsub.shakespeare.lit'
    id='create1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <create node='me_and_romeo'/>
    <configure>
      <x xmlns='jabber:x:data' type='submit'>
        <field var='FORM_TYPE' type='hidden'>
          <value>http://jabber.org/protocol/pubsub#node_config</value>
        </field>
        <field var='pubsub#access_model'><value>whitelist</value></
            field>
        <field var='pubsub#publisher'><value>romeo@montague.lit</value
            ></field>
      </x>
    </configure>
  </pubsub>
</iq>

<iq type='result' from='pubsub.shakespeare.lit' to='juliet@capulet.lit
   /balcony' id='create1'/>

<iq type='set'
    from='juliet@capulet.lit/balcony'
    id='manage1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub#owner'>
    <subscriptions node='me_and_romeo'/>
      <subscription jid='romeo@capulet.lit' subscription='subscribed'/
          >
    </subscriptions>
```

```
  </pubsub>
</iq>

<iq type='result'
    to='juliet@capulet.lit/balcony'
    id='manage1'/>
```

Now Juliet can send a message to the node and it will be delivered to both parties (it's always nice to receive a bcc to one's sending address, the client can simply ignore it, or treat it as an ack):

Listing 19: Message Publish

```
<iq from='juliet@capulet.lit/balcony' to='pubsub.shakespeare.lit' type
    ='set' id='msg1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='me_and_romeo'>
      <item>
        <message xmlns='jabber:client'>
          <body>wherefore art thou?</body>
        </message>
      </item>
    </publish>
  </pubsub>
</iq>

<iq from='pubsub.shakespeare.lit' to='juliet@capulet.lit/balcony' type
    ='result' id='msg1'/>
```

The service then sends notifications to both parties:

Listing 20: Groupchat Notifications

```
<message from='pubsub.shakespeare.lit'
         to='romeo@montague.lit/mobile'
         id='msgfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='me_and_romeo'>
      <item>
        <message xmlns='jabber:client'>
          <body>wherefore art thou?</body>
        </message>
      </item>
    </items>
  </event>
  <addresses xmlns='http://jabber.org/protocol/address'>
    <address type='replyto' jid='juliet@capulet.lit/balcony'/>
  </addresses>
</message>
```

```
<message from='pubsub.shakespeare.lit'
         to='juliet@capulet.lit/balcony'
         id='msgfoo'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='me_and_romeo'>
      <item>
        <message xmlns='jabber:client'>
          <body>wherefore art thou?</body>
        </message>
      </item>
    </items>
  </event>
  <addresses xmlns='http://jabber.org/protocol/address'>
    <address type='replyto' jid='juliet@capulet.lit/balcony'/>
  </addresses>
</message>
```

Beautiful, no?

## 7 Conclusion

Although this document shows how to do presence notifications, presence subscriptions, rosters, groupchat, and one-to-one messaging via pubsub, XMPP Eventing via Pubsub (XEP) is not limited to these functionality areas, which are provided only as examples. Indeed, XMPP Eventing via Pubsub (XEP) provides a generic mechanism for XMPP eventing that obviates the need for any future XMPP Extension Protocol (XEP) specifications other than payload formats for communication over the XMPP Eventing via Pubsub (XEP) transport. Truly, pubsub is the "one ring to bind them all" and the XEP XEP is the mother of all future XEPs. We have a clear path forward to a powerful, robust, payload-agnostic technology for the full range of eventing needs. Let us grasp the opportunity to rebuild XMPP the way it should have been built from the beginning: on top of a solid foundation of publish-subscribe. [10]

## 8 Security Considerations

In XMPP Eventing via Pubsub (XEP), access control is handled through a single permissions model, that of subscriptions to pubsub nodes. XEP nodes MUST have a default access model of "authorize" to prevent open data retrieval from potentially private data sources; this will result in a great deal of authorization requests and thus annoy typical end users to no end, but users will at least have the illusion of security, which is all they really want anyway.
End-to-end encryption is made more difficult in XMPP Eventing via Pubsub (XEP) since all

[10] But did we mention that developers really need to implement stream compression?

information passes through the pubsub service, which is typically associated with or hosted by the user's server. The solution is to run your own XMPP server in a high-security fashion. In particular, universal deployment of personal XMPP servers, domain certificates (X.509 / PKI) for channel encryption (TLS) and server-to-server trust (SASL), IPv6, DNSSEC, and IPsec will solve all security problems.
If that is not enough, XMPP can utilize onion routing schemes such as Tor for added security. Typically this results in high latency. But the word "instant" in "instant messaging" has always made XMPP seem quite frivolous (especially back when we called it "Jabber", what a silly word that is!), whereas "secure messaging" sounds like a serious technology. Who cares if delivery takes forever? (Oh and while we're at it, we should add per-hop acknowledgements for every stanza and perhaps full transactional abilities; however, those initiatives are beyond the scope of this document.)

## 9 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [11].

## 10 XMPP Registrar Considerations

This document requires no interaction with the XMPP Registrar [12].

## 11 XML Schema

Because XMPP Eventing via Pubsub simply reuses the protocol specified in XEP-0060, a separate schema is not needed.

## 12 Acknowledgements

Thanks to Maciek Niedzielski for inspiration.

[11] The Internet Assigned Numbers Authority (IANA) is the central coordinator for the assignment of unique parameter values for Internet protocols, such as port numbers and URI schemes. For further information, see <http://www.iana.org/>.

[12] The XMPP Registrar maintains a list of reserved protocol namespaces as well as registries of parameters used in the context of XMPP extension protocols approved by the XMPP Standards Foundation. For further information, see <https://xmpp.org/registrar/>.