XEP-0024: Publish/Subscribe

DJ Adams
mailto:dj.adams@pobox.com
xmpp:dj@gnu.mine.nu

Piers Harding
mailto:piers@ompa.net
xmpp:piers@gnu.mine.nu

2003-04-22
Version 0.2

Status	Type	Short Name
Retracted	Standards Track	None

A publish-subscribe protocol for Jabber.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Abstract

Pubsub (”publish/subscribe”) is a technique for coordinating the efficient delivery of information from publisher to consumer. This specification describes the use of pubsub within a Jabber context and is a result of two separate but related goals:

- to be able to exchange information _within_ a Jabber environment (for example continuously changing personal information between users)
- to be able to exchange information _using_ Jabber as a mechanism for
  - organising that exchange
  - providing transport for the information

The specification details the use of the Jabber protocol elements and introduces a new namespace, jabber:iq:pubsub. It also includes notes on actual implementation of such a mechanism in Jabber.

2 Introduction

It’s clear that as Jabber is deployed over a wider spectrum of platforms and circumstances, more and more information will be exchanged. Whether that information is specific to Jabber (JSM) users, or components, we need an mechanism to be able to manage the exchange of this information in an efficient way.
For example, it is currently the trend to embed information about a particular client’s circumstance inside presence packets, either in the <status/> tag or in an <x/> extension. One example that comes to mind is ”song currently playing on my MP3 player” (to which I have to admit some responsibility for the meme in the first place). While embedding information inside presence packets and having that information diffused to the users who are subscribed to that user’s presence has the desired effect, it has a couple of non-trivial drawbacks:

- the diffusion is inefficient, sending potentially huge amounts of data to recipients who aren’t interested
- the distribution is tied to closely to presence subscription; any entity that wants to receive information must be subscribed to the source’s presence, and there is no mechanism for specifying _what_ information they wish to receive. It is also arguably too closely tied to the JSM to be useful for _component_-based information exchange.

This is above and beyond the simple fact that this overloading of presence packets and the presence subscription and diffusion mechanism can only end in tears.
It would be far better to have a separate (sub-)protocol that enabled entities to take part in publish/subscribe relationships, and have a service that facilitated the efficient exchange of

information. Not only would it relax the undue pressure on the presence mechanism, but it would also allow people to use Jabber, which is, after all, about exchanging structured content between endpoints, as a publish/subscribe _mechanism_ in its own right.
This specification describes a publish/subscribe protocol in terms of IQ packets with payload data in a new namespace, jabber:iq:pubsub. The choice for this namespace is slightly arbitrary - it was the same namespace used in temas's original document, seems to fit well, and we need a namespace to focus on.[1]
The aim of the specification is to provide for a facility where Jabber entities can subscribe to (consume) and publish (emit) information in an efficient and organised way. These entities could be JSM users or components.
Push technology is back with a vengeance. Jabber can play a fundamental part.

3 The Specification

The pubsub services will be typically provided by a component. In what follows, there are generally three parties involved:

- the subscriber
- the pubsub service
- the publisher

Bear in mind that it is perfectly possible for a subscriber to be a publisher, and a publisher to be a subscriber, too.
The pubsub traffic will be carried in info/query (IQ) packets. All of the data in these packets will be qualified by the jabber:iq:pubsub namespace.
Pubsub scenarios can be seen in a subscribe (or unsubscribe) context or a publish context. In light of this, we will examine the IQ packets used in these contexts.

3.1 Subscribe/Unsubscribe Context

A potential consumer, or recipient, of published information, needs to request that he be sent that published information. Requesting to receive, or be pushed, information is known as subscribing.
A subscription request generally takes this form:

Listing 1: General form of a subscription

```
SEND: <iq type='set' from='subscriber' to='pubsub' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
```

[1] It may well be that we will move to a URI-based namespace in the form of a URL pointing to this specification.

```
          <subscribe to='publisher'>
            <ns>namespace:1</ns>
             <ns>namespace:2</ns>
             ...
             <ns>namespace:N</ns>
          </subscribe>
        </query>
      </iq>

RECV: <iq type='result' to='subscriber' from='pubsub' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher'>
            <ns>namespace:1</ns>
             <ns>namespace:2</ns>
             ...
             <ns>namespace:N</ns>
          </subscribe>
        </query>
      </iq>
```

3.1.1 Publisher-Specific Subscriptions and Unsubscriptions

Subscriptions can be specific to a publisher, in which case a to attribute is specified in the <subscribe/> tag:

Listing 2: Publisher-specific subscription

```
SEND: <iq type='set' to='pubsub.localhost'
             from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
             to='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
        </query>
      </iq>
```

In this case, the namespaces specified will be added to any existing list of namespaces already recorded for that subscriber:publisher relationship. In other words, it's a relative, not an absolute, subscription request.
It is also possible in a publisher-specific subscription to omit specific namespaces, if you want to be sent everything that particular publisher might publish:

Listing 3: Publisher-specific subscription without namespace specification

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher'/>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher'/>
        </query>
      </iq>
```

This type of subscription should have the effect of absolutely replacing any previous namespace-specific subscription to the publisher specified.
If a subscriber wishes to cancel a subscription from a particular publisher, he can send an unsubscribe like this:

Listing 4: Publisher-specific unsubscription

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe to='publisher'>
            <ns>namespace:1</ns>
          </unsubscribe>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe to='publisher'>
            <ns>namespace:1</ns>
          </unsubscribe>
        </query>
      </iq>
```

This should have the effect of removing the subscription from that publisher for the namespaces specified.

You can also send an unsubscribe without specifying any namespaces:

Listing 5: Publisher-specific general unsubscription

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe to='publisher'/>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe to='publisher'/>
        </query>
      </iq>
```

This should have the effect of removing any subscription relationship with the publisher specified. Note, however, that this won't stop the subscriber being pushed information from that publisher if he's specified a "publisher-generic" subscription (see next section).

3.1.2 Non-Publisher-Specific Subscriptions and Unsubscriptions

As well as being able to subscribe to specific publishers, it is also possible to subscribe to receive data, according to namespace, regardless of publisher:

Listing 6: General namespace specific subscription

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
        </query>
```

```
      </iq>
```

This means that the subscriber wishes to be pushed information in the namespaces specified, regardless of who publishes it. Like the publisher-specific subscribe that specifies namespaces, this request is relative, in the namespaces are added to any existing namespaces already recorded for this generic subscription.
Subscribing to everything from everyone is probably not a good idea and we should not allow this. (The format of the request is actually used in an IQ-get context - see later).

Listing 7: This is not allowed

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe/>
        </query>
      </iq>

RECV: <iq type='error' from='pubsub.localhost'
            to='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe/>
        </query>
        <error code='405'>Not Allowed</error>
      </iq>
```

Likewise, you can unsubscribe from certain namespaces in this non-publisher-specific context like this:

Listing 8: General unsubscription to specific namespaces

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </unsubscribe>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </unsubscribe>
```

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

If there are any subscriptions to specific publishers for the namespaces specified here, they should be removed (for those namespaces) in addition to the removal from the 'all publishers' list.
Finally, a subscriber can wipe the slate clean like this:

Listing 9: Wiping the slate

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe/>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <unsubscribe/>
        </query>
      </iq>
```

which should have the effect of removing all namespace subscriptions from everywhere.

3.1.3 Further Notes

All the examples so far have shown actions on the subscriber's part, and have consisted of IQ-sets. In an IQ-set, within the jabber:iq:pubsub namespace, multiple children can exist in the query payload, but those children must be of the same type. In other words, you can send multiple <subscribe/>s, or multiple <unsubscribe/>s, but not a combination of the two.
This is allowed:

Listing 10: Subscribing to more than one publisher at once

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisherA'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
          <subscribe to='publisherB'>
            <ns>namespace:3</ns>
          </subscribe>
        </query>
```

```
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisherA'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
          <subscribe to='publisherB'>
            <ns>namespace:3</ns>
          </subscribe>
        </query>
      </iq>
```

But this is not allowed:

Listing 11: Subscribes and unsubscribes in same IQ-set is not allowed

```
SEND: <iq type='set' to='pubsub.localhost'
            from='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisherA'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
          <unsubscribe to='publisherB'>
            <ns>namespace:3</ns>
          </unsubscribe>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisherA'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
          <unsubscribe to='publisherB'>
            <ns>namespace:3</ns>
          </unsubscribe>
        </query>
        <error code='400'>
          Bad Request: only subscribes or unsubscribes
        </error>
      </iq>
```

In the case where multiple <subscribe/>s or <unsubscribe/>s appear in an action, each element will be processed in turn, as they appear in the payload.

As well as actions, the subscriber can query his subscription using an IQ-get in the jabber:iq:pubsub namespace. This should return a list of the subscribers current subscriptions, like this:

Listing 12: Querying current subscription

```
SEND: <iq type='get' to='pubsub.localhost'
            from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe/>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
          <subscribe to='publisherA'>
            <ns>namespace:2</ns>
            <ns>namespace:4</ns>
          </subscribe>
          <subscribe to='publisherB'>
            <ns>namespace:5</ns>
          </subscribe>
        </query>
      </iq>
```

Note the two references to namespace:2 - one inside the non-publisher-specific subscription list and one inside the subscription list specific to publisherA. This example implies that the non-publisher-specific and publisher-specific subscription information should be kept separately. This is designed to make it easier on the subscriber to manage his specific subscriptions over time.

3.2 Publish Context

In contrast to the subscribe and unsubscribe context, the publishing context is a lot simpler to explain.
A publisher can publish information within a certain namespace, like this:

Listing 13: Publishing information

```
SEND: <iq type='set' to='pubsub.localhost'
            from='publisher@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
```

```
          <publish ns='foo'>
            <foo xmlns='foo'>bar</foo>
          </publish>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='publisher@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='foo'>
            <foo xmlns='foo'>bar</foo>
          </publish>
        </query>
      </iq>
```

It's also possible for a publisher to publish more than one item at once, like this:

Listing 14: Publishing information in different namespaces

```
SEND: <iq type='set' to='pubsub.localhost'
            from='publisher.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='foo'>
            <foo xmlns='foo'>bar</foo>
          </publish>
          <publish ns='jabber:x:oob'>
            <x xmlns='jabber:x:oob'>
              <url>http://www.pipetree.com/jabber/</url>
              <desc>Some stuff about Jabber</desc>
            </x>
          </publish>
        </query>
      </iq>

RECV: <iq type='result' from='pubsub.localhost'
            to='publisher.localhost' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='foo'>
            <foo xmlns='foo'>bar</foo>
          </publish>
          <publish ns='jabber:x:oob'>
            <x xmlns='jabber:x:oob'>
              <url>http://www.pipetree.com/jabber/</url>
              <desc>Some stuff about Jabber</desc>
            </x>
          </publish>
        </query>
      </iq>
```

Each published item is wrapped in a <publish/> tag. This tag must contain the namespace of the item being publishes, in an ns attribute, as shown. This is distinct from the xmlns attribute of the fragment of XML actually being published. It is theoretically none of the pubsub component's business to go poking around in the real published data, nor should it have to. It needs to know what namespace is qualifying the published information that has been received, so that the list of appropriate recipients can be determined.

3.3 Distributing Published Information

While it's the responsibility of the publishing entities to publish information, it's the responsibility of the pubsub component to push out that published data to the subscribers. The list of recipient subscribers must be determined by the information stored by the pubsub component as a result of receiving subscription requests (which are described earlier).
On receipt of an IQ-set containing published information, the pubsub entity must determine the list of subscribers to which that information should be pushed. If the IQ-set contains multiple <publish/> fragments, this process must be carried out for each one in turn. [2]
Taking the earlier example of the publishing of data in the 'foo' namespace, the following example shows what the pubsub component must send to push this foo data out to a subscriber.

Listing 15: Pushing out published information to a subscriber

```
SEND: <iq type='set' to='subscriber@localhost/foosink'
            from='pubsub.localhost' id='push1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='foo' from='publisher@localhost'>
            <foo xmlns='foo'>bar</foo>
          </publish>
        </query>
      </iq>
```

The recipient is _not_ required to send an 'acknowledgement' in the form of an IQ-result; the idea that this _push_ of information is akin to how information is pushed in a live browsing context (see jabber:iq:browse documentation for more details).

3.4 Delivery Sensitivity

When a pubsub service receives a publish packet like the ones above, it needs to deliver (push) the information out according to the subscriptions that have been made.
However, we can introduce a modicum of sensitivity by using a presence subscription between the pubsub service and the subscriber(s). If the subscriber wishes only to receive

[2] Whether a pubsub component implementation should be allowed to batch up individual published information fragments for one recipient as a result of a large, multi-part incoming publishing IQ-set, is not specified here, the choice is down to the implementer. Receiving entities should be able to cope with being pushed an IQ-set with multiple fragments of published data.

information when he's online (this is a JSM-specific issue), then he needs to set up a presence subscription relationship with the pubsub service. The pubsub service should respond to presence subscriptions and unsubscriptions by

- accepting the (un)subscription request
- reciprocating the (un)subscription request

If the pubsub service deems that a published piece of information should be pushed to a subscriber, and there is a presence subscription relationship with that subscriber, the service should only push that information to the subscriber if he is available. If he is not available, the information is not to be sent.
Thus the subscriber can control the sensitivity by initiating (or not) a presence relationship with the service. If the subscriber wishes to receive information regardless of availability, he should not initiate a (or cancel any previous) presence relationship with the service.
This loose coupling of presence relationships for sensitivity allows this specification to be used in the wider context of component-to-component publish/subscribe where presence is not a given.

3.5 Use of Resources

When in receipt of a pubsub subscription request from an entity where a resource is specified in the JID, the pubsub component must honour the resource specified in the from attribute of the request. For example, here's a typical subscription request from a JSM user:

Listing 16: Incoming subscription request from a JSM user

```
RECV: <iq type='set' to='pubsub.localhost'
            from='subscriber@localhost/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher'>
            <ns>namespace:1</ns>
            <ns>namespace:2</ns>
          </subscribe>
        </query>
      </iq>
```

When storing the subscriber/publisher/namespace relationship matrix for eventual querying when a publisher publishes some information, the pubsub component must use the full JID, not just the username@host part.
Similarly, in this example:

Listing 17: Incoming subscription request from a component

```
RECV: <iq type='set' to='pubsub.localhost'
```

```
        from='news.server/politics-listener' id='s1'>
      <query xmlns='jabber:iq:pubsub'>
        <subscribe to='publisher'>
          <ns>news:politics:home</ns>
          <ns>news:politics:foreign:usa</ns>
        </subscribe>
      </query>
    </iq>
```

the full JID of the component subscriber - news.server/politics-listener, should be used to qualify the matrix.
This is because it allows the subscribing entities to arrange the receipt of pushed items by resource. In the case of a JSM user, it allows him to organise his clients, which may have different capabilities (some being able to handle the jabber:iq:pubsub data, others not) to receive the 'right' data. In the case of a component, it allows the component to associate component-specific data with incoming published namespace-qualified information.

4 Implementation Notes

While the specification describes the fundamental building blocks of the pubsub protocol, there are ideas that are not discussed above but nonetheless may be incorporated into an implementation. There are other considerations that have to be made in the wider context of publish and subscribe. Some of the main ones are discussed briefly here too.

4.1 Publisher Discovery

There is no part of this pubsub specification that determines how a potential subscriber might discover publishers. After all, there are no rules governing which pubsub component a publisher could or should publish to. And since pubsub subscriptions are specific to a pubsub component, there is an information gap - "how do I find out what publishers there are, and through which pubsub components they're publishing information?"
This problem domain should be solved using other methods, not with the actual jabber:iq:pubsub specific namespace. A combination of jabber:iq:browse usage (the magic ointment that heals all things) and perhaps a DNS style (or at least root-node-based) knowledge hierarchy might be the right direction.
In the case where a server administrator wishes to facilitate pubsub flow between JSM users on a server, a pubsub component can be plugged into the jabberd backbone, and there is potentially no real issue with knowing which pubsub component to use, and where it is. But what about if the JSM users on one server wish to build pubsub relationships with JSM users on another server? (Note that this general question is not specific to JSM users, although that example will be used here). The next two sections look at how these things might pan out.

4.2 Cross-Server Relationships

When JSM users on server1 wish to subscribe to information published by JSM users on server2 (let's say it's the mp3 player info, or avatars) then there are some issues that come immediately to mind:

- Does a JSM user on server1 (userA@server1) send his IQ-set subscription to the pubsub component on server2 (pubsub.server2), or server1 (pubsub.server1)?
- If he sends it to pubsub.server2, can we expect pubsub.server2 to always accept that subscription request, i.e. to be willing to serve userA@server1 (if pubsub.server2 knows that pubsub.server1 exists)?
- Will there be performance (or at least server-to-server traffic) implications if many subscription relationships exist between subscribers on server1 and publishers on server2?

4.2.1 Proxy Subscriptions

To reduce the amount of server-to-server traffic, we can employ the concept of "proxy subscriptions". This is simply getting a pubsub component to act on behalf of a (server-local) subscriber. Benefit comes when a pubsub component acts on behalf of multiple (server-local) subscribers.
Here's how such proxy subscriptions can work, to reduce the amount of server-to-server traffic:
Step 1: Subscriber sends original subscription
JSM users on server1 wish to subscribe to information published by an entity on server2. Each of them sends a subscription request to the _local_ pubsub component:

```
SEND: <iq type='set' to='pubsub.server1'
            from='subscriber@server1/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher.server2'>
            <ns>namespace:1</ns>
          </subscribe>
        </query>
      </iq>
```

Step2: Pubsub component subscribes on subscriber's behalf
The pubsub component knows about the publisher, and where (to which pubsub component) that publisher publishes information. It formulates a subscription request and sends it to the remote pubsub component:

```
SEND: <iq type='set' to='pubsub.server2'
            from='pubsub.server1' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
```

```
          <subscribe to='publisher.server2'>
            <ns>namespace:1</ns>
          </subscribe>
        </query>
      </iq>
```

The remote pubsub component receives and acknowledges the subscription request, and the local pubsub component relays the response back to the original requester:

```
SEND: <iq type='result' from='pubsub.server1'
            to='subscriber@server1/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher.server2'>
            <ns>namespace:1</ns>
          </subscribe>
        </query>
      </iq>
```

If the remote pubsub server was unable or unwilling to accept the subscription request, this should be reflected in the response:

```
SEND: <iq type='error' from='pubsub.server1'
            to='subscriber@server1/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher.server2'>
            <ns>namespace:1</ns>
          </subscribe>
        </query>
        <error code='406'>Not Acceptable</error>
      </iq>
```

Step3: Publisher publishes information
The publisher, publisher.server2, publishes information in the namespace:1 namespace, to the remote pubsub component pubsub.server2:

```
SEND: <iq type='set' from='publisher.server2'
            to='pubsub.server2' id='p1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='namespace;1'>
            <stuff xmlns='namespace:1'>nonsense</stuff>
          </publish>
        </query>
      </iq>
```

Step4: Pubsub component receives published information
The pubsub component pushes the published information to pubsub.server1, who has been determined to be a valid recipient:

```
RECV: <iq type='set' from='pubsub.server2'
            to='pubsub.server1' id='p1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='namespace;1' from='publisher.server2'>
            <stuff xmlns='namespace:1'>nonsense</stuff>
          </publish>
        </query>
      </iq>
```

Step5: Pubsub component forwards published information to original subscriber
The local pubsub component then diffuses the information received to the original subscriber:

```
SEND: <iq type='set' from='pubsub.server1'
            to='subscriber@server1/resource' id='p1'>
        <query xmlns='jabber:iq:pubsub'>
          <publish ns='namespace;1' from='publisher.server2'>
            <stuff xmlns='namespace:1'>nonsense</stuff>
          </publish>
        </query>
      </iq>
```

This way, only a single published element must travel between servers to satisfy a multiplex of subscribed entities at the delivery end.
Of course, this mechanism will rely upon knowledge about pubsub components and where they're available; furthermore, it will require knowledge about where publisher entities publish their information. This knowledge, and the mechanisms to discover this sort of information, is not to be covered in this spec, which purely deals with the subscription and publishing of information. As SOAP is to UDDI (to use a slightly controversial pair of technologies), so is jabber:iq:pubsub to this discovery mechanism as yet undefined. To include the definition of such a discovery mechanism in this specification is wrong on two counts:

- Discovery mechanisms by nature should not be tied to specific areas
- Trying to load too much onto jabber:iq:pubsub will only produce a complex and hard-to-implement specification

After all, the jabber:iq:pubsub spec as defined here is usable out of the box for the simple scenarios, and scenarios where discovery is not necessary or the information can be exchanged in other ways.

4.2.2 Willingness to Serve

There are some situations where it might be appropriate for a pubsub component to refuse particular subscription requests. Here are two examples:

- Where a pubsub component that's been designed, implemented, or configured to handle local-only pubsub traffic, and a subscription request is received, specifying a publisher that the local pubsub component knows to be one that publishes to a remote pubsub component [3]. In this case, the local pubsub component would be unwilling to provoke a server-to-server connection and therefore unwilling to honour the request.
- Where a pubsub component receives a subscription request from a remote subscriber, and that pubsub component knows that there's a pubsub component local to the subscriber. In this case, the (administrator of the) remote pubsub component might want to encourage proxy subscriptions.

A refusal could take one of a number of guises:

Listing 18: A flat refusal

```
SEND: <iq type='error' from='pubsub.server2'
            to='subscriber@server1/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher.server2'>
            <ns>namespace:1</ns>
          </subscribe>
        </query>
        <error code='406'>Local pubsub only</error>
      </iq>
```

Listing 19: A refusal with redirection

```
SEND: <iq type='error' from='pubsub.server2'
            to='subscriber@server1/resource' id='s1'>
        <query xmlns='jabber:iq:pubsub'>
          <subscribe to='publisher.server2'>
            <ns>namespace:1</ns>
          </subscribe>
        </query>
        <error code='302' jid='pubsub.server1'/>
      </iq>
```

Note: This 302 redirect is not covered in the general protocol specification, but it's an interesting concept :-)

4.3 Subscriber Anonymity and Acceptance?

The jabber:iq:pubsub specification makes no provision for publishers to query a pubsub component to ask for a list of those entities that are subscribed to (namespaces) it (publishes). This is deliberate. Do we wish to add to the specification to allow the publisher to discover

[3] under other circumstances, this would trigger a 'Proxy Subscription', as described earlier, if supported

this information? If so, it must be as an optional 'opt-in' (or 'opt-out') tag for the subscriber, to determine whether his JID will show up on the list. [4]
Associated with this is the semi-reciprocal issue of acceptance? The specification deliberately makes no provision for a subscription acceptance mechanism (where the publisher must first accept a subscriber's request, via the pubsub component). If we're to prevent the publishers knowing who is subscribing, ought we to give them the power of veto, to 'balance things out'? Note that if we do, the acceptance issue is not necessarily one for the pubsub specification to resolve; there are other ways of introducing access control, at least in a component environment; use of a mechanism that the Jabber::Component::Proxy Perl module represents is one example: wedge a proxy component in front of a real (pubsub) component and have the ability to use ACLs (access control lists) to control who gets to connect to the real component.

[4] Even if there is no provision for querying the subscribers, perhaps we should make a provision for the publisher to ask the pubsub component for a list of namespaces that have been subscribed to (for that publisher).