XEP-0237: Roster Versioning

Peter Saint-Andre
mailto:peter@andyet.net
xmpp:stpeter@stpeter.im
https://stpeter.im/

Dave Cridland
mailto:dave.cridland@surevine.com
xmpp:dave.cridland@surevine.com

2012-02-08
Version 1.3

Status	Type	Short Name
Obsolete	Standards Track	N/A

This specification defines a proposed modification to the XMPP roster protocol that enables versioning of rosters such that the server will not send the roster to the client if the roster has not been modified, thus saving bandwidth during session establishment.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

Although XMPP rosters can become quite large, they tend to change infrequently. Therefore it can be inefficient for the server to send the roster to the client during session establishment if the roster has not been modified. This document defines a small modification to the XMPP roster protocol specified in XMPP IM [1] that enables "versioning" of roster information.
The basic model is that if the client specifies a version ID when it requests the roster, the server returns an empty IQ-result. If the roster has been modified, the server sends versioned roster pushes for each roster item that has been touched in any way since the version specified by the client. The client processes each roster push as it normally would, modifying its local version ID with each roster push it receives. This enables the client to receive only the items that have been modified, not the entire roster.
Note: The protocol described herein has been incorporated into RFC 6121 [2].

2 Protocol

2.1 Stream Feature

If a server supports roster versioning, it MUST inform the connecting entity when returning stream features during the stream negotiation process (at the latest, when informing a client that resource binding is required). This is done by including a <ver/> element qualified by the 'urn:xmpp:features:rosterver' namespace.

Listing 1: Stream features

```
<stream:features>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <required/>
  </bind>
  <ver xmlns='urn:xmpp:features:rosterver'/>
</stream:features>
```

The roster versioning stream feature is merely informative and therefore is never mandatory-to-negotiate.

2.2 Data Format

This document adds a new 'ver' attribute to the <query/> element qualified by the 'jabber:iq:roster' namespace, defined as follows.

[1] RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence <http://tools.ietf.org/html/rfc6121>.
[2] RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence <http://tools.ietf.org/html/rfc6121>.

Definition: The **'ver' attribute** is a string that identifies a particular version of the roster information. The value MUST be generated only by the server and MUST be treated by the client as opaque. The server can use any appropriate method for generating the version ID, such as a hash of the roster data or a strictly-increasing sequence number.

2.3 Client Request

If a client supports roster versioning and the server to which it has connected advertises support for roster versioning as described under Stream Feature, then the client MUST include the 'ver' element in its request for the roster. If the server does not advertise support for roster versioning, the client MUST NOT include the 'ver' attribute. If the client includes the 'ver' attribute in its roster get, it sets the attribute's value to the version ID associated with its last cache of the roster.

Listing 2: Roster get with version ID

```
C: <iq from='romeo@montague.lit/home' id='r1h3vzp7' to='romeo@montague
   .lit' type='get'>
     <query xmlns='jabber:iq:roster' ver='ver14'/>
   </iq>
```

If the client has not yet cached the roster or the cache is lost or corrupted, but the client wishes to bootstrap the use of roster versioning, it MUST set the 'ver' attribute to the empty string (i.e., **ver=""**).
Naturally, if the client does not support roster versioning or does not wish to bootstrap the use of roster versioning, it will behave like an RFC-3921-compliant client by not including the 'ver' attribute.

2.4 Server Response

Whether or not the roster has been modified since the version ID enumerated by the client, the server MUST either return the complete roster as described in RFC 3921 (including a 'ver' attribute that signals the latest version) or return an empty IQ-result (thus indicating that any roster modifications will be sent via roster pushes, as described below). In general, unless returning the complete roster would (1) use less bandwidth than sending individual roster pushes to the client (e.g., if the roster contains only a few items) or (2) the server cannot associate the version ID with any previous version it has on file, the server SHOULD send an empty IQ-result and then send the modifications (if any) via roster pushes.

Listing 3: Empty roster result

```
S: <iq from='romeo@montague.lit' id='r1h3vzp7' to='romeo@montague.lit/
   home' type='result'/>
```

Note: This empty IQ-result is different from an empty <query/>, thus disambiguating this usage from an empty roster.
If the roster has not been modified since the version ID enumerated by the client, the server will simply not send any roster pushes to the client (until and unless some relevant event triggers a roster push during the lifetime of the client's session).
If the roster has been modified since the version ID enumerated by the client, the server MUST then send one roster push to the client for each roster item that has been modified since the version ID enumerated by the client. (We call a roster push that is sent for purposes of roster version synchronization an "interim roster push".)
Definition: A **"roster modification"** is any modification to the roster data that would result in a roster push to a connected client. Therefore internal states related to roster processing within the server that would not result in a roster push to a connected client do not necessitate a change to the version.

Listing 4: Roster pushes

```
S: <iq from='romeo@montague.lit' id='ah382g67' to='romeo@montague.lit/
   home' type='set'>
     <query xmlns='jabber:iq:roster' ver='ver34'>
       <item jid='tybalt@shakespeare.lit' subscription='remove'/>
     </query>
   </iq>

S: <iq from='romeo@montague.lit' id='b2gs90j5' to='romeo@montague.lit/
   home' type='set'>
     <query xmlns='jabber:iq:roster' ver='ver42'>
       <item jid='bill@shakespeare.lit' subscription='both'/>
     </query>
   </iq>

S: <iq from='romeo@montague.lit' id='c73gs419' to='romeo@montague.lit/
   home' type='set'>
     <query xmlns='jabber:iq:roster' ver='ver72'>
       <item jid='nurse@shakespeare.lit' name='Nurse' subscription='to
          '>
         <group>Servants</group>
       </item>
     </query>
   </iq>

S: <iq from='romeo@montague.lit' id='dh361f35' to='romeo@montague.lit/
   home' type='set'>
     <query xmlns='jabber:iq:roster' ver='ver96'>
       <item jid='juliet@shakespeare.lit' name='Juliet' subscription='
          both'>
         <group>VIPs</group>
       </item>
     </query>
```

```
    </iq>
```

These ”interim roster pushes” can be understood as follows:

1. Imagine that the client had an active presence session for the entire time between its cached roster version (say, ”ver14”) and the new roster version (say, ”ver96”).
2. During that time, the client might have received roster pushes related to various roster versions. However, some of those roster pushes might have contained intermediate updates to the same roster item (e.g., modifications to the subscription state for bill@shakespeare.lit from ”none” to ”to” and from ”to” to ”both”).
3. The interim roster pushes would not include all of the intermediate steps, only the final result of all modifications applied to each item while the client was in fact offline (say, ”ver34”, ”ver42”, ”ver72”, and ”ver96”).

The client MUST handle an ”interim roster push” in the same way it handles any roster push (indeed, from the client’s perspective it cannot tell the difference between an ”interim” roster push and a ”live” roster push). If the client’s session ends before it receives all of the interim roster pushes, when requesting the roster after reconnection it SHOULD request the version associated with the last roster *push* it received during the session that was disconnected, not the version associated with the roster *result* it received at the start of the session that was disconnected.
When roster versioning is enabled, the server MUST include the updated roster version with each roster push. Roster pushes MUST occur in order of modification and the version contained in a roster push MUST be unique.

3 Examples

This section provides a detailed scenario that illustrates the use of roster versioning. In this example the client gets disconnected before the server has had a chance to send all of its roster pushes, but this is immaterial to the synchronization process.

Listing 5: The roster synchronization process

```
C: <iq from='romeo@montague.lit/home' id='r1h3vzp7' to='romeo@montague
   .lit' type='get'>
     <query xmlns='jabber:iq:roster' ver='ver14'/>
   </iq>

S: <iq from='romeo@montague.lit' id='r1h3vzp7' to='romeo@montague.lit/
   home' type='result'/>

S: <iq from='romeo@montague.lit' id='ah382g67' to='romeo@montague.lit/
   home' type='set'>
```

```
    <query xmlns='jabber:iq:roster' ver='ver34'>
      <item jid='tybalt@shakespeare.lit' subscription='remove'/>
    </query>
  </iq>

S: <iq from='romeo@montague.lit' id='b2gs90j5' to='romeo@montague.lit/
  home' type='set'>
    <query xmlns='jabber:iq:roster' ver='ver42'>
      <item jid='bill@shakespeare.lit' subscription='both'/>
    </query>
  </iq>

S: </stream:stream>

[ reconnection ]

C: <iq from='romeo@montague.lit/home' id='r2xa7gf9' to='romeo@montague
  .lit' type='get'>
    <query xmlns='jabber:iq:roster' ver='ver42'/>
  </iq>

S: <iq from='romeo@montague.lit' id='r2xa7gf9' to='romeo@montague.lit/
  home' type='result'/>

S: <iq from='romeo@montague.lit' id='c73gs419' to='romeo@montague.lit/
  home' type='set'>
    <query xmlns='jabber:iq:roster' ver='ver72'>
      <item jid='nurse@shakespeare.lit' name='Nurse' subscription='to
          '>
        <group>Servants</group>
      </item>
    </query>
  </iq>

S: <iq from='romeo@montague.lit' id='dh361f35' to='romeo@montague.lit/
  home' type='set'>
    <query xmlns='jabber:iq:roster' ver='ver96'>
      <item jid='juliet@shakespeare.lit' name='Juliet' subscription='
          both'>
        <group>VIPs</group>
      </item>
    </query>
  </iq>
```

4 Implementation Guidelines

This specification is specifically designed to allow for a wide range of implementation choices. These range from highly simplistic but inefficient, to very efficient but quite complex.

This section provides suggestions, rather than instructions, on some lightweight approaches to conforming with the specification.

4.1 Syntactic Conformance

A server can conform to this specification by accepting and ignoring the 'ver' attribute in requests, and providing an empty 'ver' attribute in each roster push.
This provides no efficiency savings for clients.

4.2 Exact-Match Conformance

Using some digest (hash) of the roster, a server can identify unchanged rosters, and handle the case where the client sends a ver corresponding to the current roster state.
This will account for the majority of cases, and represents a substantial saving. Server implementors are advised to canonicalize the form and ordering of roster items prior to applying the hash function. This hash function need not be cryptographically secure, merely resistent to collisions, and it is advisable to pick one that is fast to compute.
No additional data need be stored, although storing the current hash will yield some performance advantage. This strategy is thought to be relatively safe in the face of data loss on the server.

4.3 Add-Only Conformance

Using a strictly increasing sequence for the 'ver' attribute, a server can "stamp" each roster item with its last change, and the roster as a whole with its last deletion. The server returns either the entire roster -- if a deletion has occured since the client's ver value -- or those changed items.
Deletions are thought to be rare compared to additions and modifications, and as such this approach captures almost all changes. The additional storage cost is also low.
Implementors could combine this strategy with the previous one, detecting a sequence of modifications yielding the same roster as the client has cached already, by constructing a ver attribute containing both a hash and sequence value. This might provide some resilience in the case of data loss.
Implementors are advised that a pure timestamp is not suitable for this approach, since under some circumstances system clocks can go backwards (e.g., because of an adjustment based on an update triggered by use of the Network Time Protocol as described in RFC 958 [3]).

[3] RFC 958: Network Time Protocol (NTP) <http://tools.ietf.org/html/rfc0958>.

4.4 Sending Pushes

There are two primary approaches to server-side generation of the 'ver' attribute: complete roster hashes and strictly increasing sequence numbers. Whether the server will send roster pushes varies depending on the approach taken. For instance, if a series of roster modifications result in a roster item that does not differ from the version cached by the client (e.g., a modification to the item's 'name' attribute and then a modification back to the original value), then a server that implements the "complete roster hashes" approach would not consider the item to have been modified for purposes of roster versioning and therefore would not push the item to the client in an interim roster push; however, a server that implements the "strictly increasing sequence numbers" approach would send a roster push in this situtation.

4.5 Client Implementation Guidelines

Client implementors are reminded that the value of the 'ver' attribute is entirely opaque, and they should behave identically with each strategy described above by simply conforming to the specification. The only storage requirement for this specification is the last seen 'ver' attribute.

5 Security Considerations

It is possible that client-side caching of roster information across sessions (rather than holding them in memory only for the life of a session) could introduce new vulnerabilities, such as misuse by malware. Implementations are advised to appropriately protect cached roster data.

6 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA) [4].

7 XMPP Registrar Considerations

7.1 Protocol Namespaces

This specification defines the following XML namespace:

[4] 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/>.

- urn:xmpp:features:rosterver

Upon advancement of this specification from a status of Experimental to a status of Draft, the XMPP Registrar [5] shall add the foregoing namespace to the registry located at <https://xmpp.org/registrar/stream-features.html>, as described in Section 4 of XMPP Registrar Function (XEP-0053) [6].

8 XML Schemas

8.1 jabber:iq:roster

This specification proposes addition of the 'ver' attribute to the schema for the 'jabber:iq:roster' namespace.

8.2 Stream Feature

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

<xs:schema
    xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='urn:xmpp:features:rosterver'
    xmlns='urn:xmpp:features:rosterver'
    elementFormDefault='qualified'>

  <xs:annotation>
    <xs:documentation>
      The protocol documented by this schema is defined in
      RFC 6121: http://tools.ietf.org/html/rfc6121
    </xs:documentation>
  </xs:annotation>

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

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

</xs:schema>
```

[5] 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/>.

[6] XEP-0053: XMPP Registrar Function <https://xmpp.org/extensions/xep-0053.html>.

9 Acknowledgements

Thanks to Dave Cridland, Richard Dobson, Leonid Evdokimov, Fabio Forno, Alexander Gnauck, Juha Hartikainen, Joe Hildebrand, Waqas Hussain, Justin Karneges, Sachin Khandelwal, Curtis King, Jonas Lindberg, Pedro Melo, Matthew Wild, Jiří Zárevúcký, and Florian Zeitz for their comments.