XEP-0366: Entity Versioning

Sam Whited
mailto:sam@samwhited.com
xmpp:sam@samwhited.com
https://blog.samwhited.com/

2016-12-21
Version 0.1.2

Status	Type	Short Name
Deferred	Standards Track	EV

A method by which lists of items may be versioned so that servers will not need to send the entire list if it has not been modified, saving bandwidth and time with minimal state being stored by the server and client.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

This problem of "downloading the world" (downloading the entire roster every time a session is initialized, or receiving an entire disco items response every time a MUC list is queried, etc.) was partially addressed by Roster Versioning (XEP-0237) [1] which was later merged into RFC 6121 [2] §2.6. While this solved the problem for the roster, it did not account for other entities. Furthermore, roster versioning requires that the server maintain a great deal of state (roster items which should be pushed for each entity on reconnect, monotonically increasing counters, etc.) which can be difficult to store or synchronize in a large, distributed system. This XEP defines a method by which generic entity lists can be versioned and cached which is optimized for distributed systems with large entity lists, but which works equally well on small, single server deployments.

2 Requirements

- An extra round trip MUST NOT be required to initiate entity versioning.
- Clients that do not implement this protocol (but which use servers that do) MUST still be able to request and receive entities normally.
- Servers which implement this protocol MUST NOT be required to store multiple versions of an entity list or maintain other redundant state.
- Inconsistent state between servers in a cluster should not cause cache invalidation for the entire entity list.
- Large changes SHOULD NOT be required for existing servers / clients.

3 Glossary

Aggregate Token A hash which represents the state of a list of entities, and changes if any of the entities change.

Versioned Entity Any object which may be versioned (eg. rooms, users).

Version Token A short, case sensitive string which represents an entity and changes if that entity changes.

[1] XEP-0237: Roster Versioning <https://xmpp.org/extensions/xep-0237.html>.
[2] RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence <http://tools.ietf.org/html/rfc6121>.

4 Use Cases

4.1 Clients

- A client on a mobile device where bandwidth and throughput are limited has a very large roster which cause connecting to take an unacceptable amount of time. With entity versioning, subsequent connections after the first do not take as long, and use less bandwidth.
- A client often wants to view the list of multi-user chat rooms available on a servers MUC service. However, the list is very long and takes a long time to download. After enabling entity versioning the client can fetch the list, and then poll for changes at a later date without re-requesting the entire list.
- A client wishes to cache the features supported by servers of the contacts in their roster since their disco items is not likely to change often.

4.2 Servers

- A server is running in an environment where storing multiple versions of each users roster may put too much pressure on the storage backend. After enabling entity versioning, they only have to store a small token per user and can calculate the diffs to send to the client afterwards.
- A server maintains an out-of-band HTTP API for fetching information about MUC rooms to display on their web page. They wish to use a reverse proxy to cache API requests based on etags. Instead of attempting to check if the backend page has changed and generate etags, the room's entity version token is used as a weakly-validated ETag.

5 Entity Versioning Profiles

Because entity versioning is designed to be a generic system for syncing any sort of list in XMPP, and the format and requirements of various entity lists may vary greatly, no specific wire format is defined in this specification. Instead, the specifics for various lists will be left up to separate XEPs which will define entity versioning "profiles" which must be registered with the XMPP registrar. These profiles will define exactly how version tokens are represented in the specific list format for which they wish to use entity versioning. The rest of this document will provide details about entity versioning which will be common to all entity versioning profiles and do not need to be redefined in EV profile XEPs. It will also define an EV profile for fetching the roster.
The roster entity versioning profile which is used as an example throughout this document will use the namespace 'urn:xmpp:entityver:profile:roster:0' as described in the XMPP Registrar Considerations section of this document.

6 Discovering Support

If a server supports entity versioning, it MUST inform the connecting client when returning stream features during the stream negotiation process. This is done by including a <ver/> element, qualified by the 'urn:xmpp:entityver:0' namespace with child <profile> nodes for each supported entity versioning profile. At the latest, this SHOULD be done when informing a client that resource binding is required. For example if the server only supports versioning of rosters it might return:

Listing 1: Stream Features

```
<stream:features>
  <bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'>
    <required/>
  </bind>
  <ver xmlns='urn:xmpp:entityver:0'>
    <profile xmlns='urn:xmpp:entityver:profile:roster:0'/>
  </ver>
</stream:features>
```

The entity versioning stream feature is merely informative and therefore is never mandatory-to-negotiate.
Clients, servers, and other entities that support Service Discovery (XEP-0030) [3] and entity versioning must respond to service discovery requests with a feature of 'urn:xmpp:entityver:0' and with a feature for each EV profile supported by the responding entity as described in the relavant specifications. Eg. a response from a server that supports roster versioning for the requesting entity might look like the following:

Listing 2: Service discovery information response

```
<iq from='shakespeare.lit'
    id='ku6e51v3'
    to='kingclaudius@shakespeare.lit/castle'
    type='result'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <feature var='urn:xmpp:entityver:0'/>
    <feature var='urn:xmpp:entityver:profile:roster:0'/>
  </query>
</iq>
```

[3] XEP-0030: Service Discovery <https://xmpp.org/extensions/xep-0030.html>.

7 Entity Sync

7.1 Version Tokens

Version tokens are short case-sensitive strings which are generated by the server. Their format is not defined in this spec, but a recommendation may be found in the Implementation Notes. Version tokens are akin to a weakly-validated etag for the entity in question.
Servers that implement this protocol must assign such a version token to each entity that is controlled by the server. The server SHOULD then update this version every time any mutable property of the entity changes (eg. when the subscription status of a user changes). The server MAY choose to update this token at any time (to force the clients to invalidate their cached representation of the object). This version token MUST then be included with every object representation of that entity transmitted in the stream. This is done by including a sub-node called "version" qualified by the entity versioning XML namespace defined in this document. Similarly, clients MAY also add version nodes for each version token they possess to the request for a list (not specifying a version token will force the server to send information on that entity to the client). If a client sends up a list of version tokens, the server MUST then check to see if those tokens correspond to any entity which it knows about, and not send down any entities with matching version tokens in the response.
For example, a versioned roster request might look like this:

Listing 3: Roster Request

```
<!-{}- Client -{}->
<iq from='romeo@montague.lit/home' id='56' to='romeo@montague.lit'
    type='get'>
  <query xmlns='jabber:iq:roster'>
    <item jid='bill@shakespeare.lit'>
      <version xmlns='urn:xmpp:entityver:0'>25P2A7H8</version>
    </item>
    <item jid='anne@shakespeare.lit'>
      <version xmlns='urn:xmpp:entityver:0'>VIZSVF0D</version>
    </item>
  </query>
</iq>

<!-{}- Server -{}->
<iq from='romeo@montague.lit' id='56' to='romeo@montague.lit/home'
    type='result>
␣␣<query␣xmlns='jabber:iq:roster'>
␣␣␣␣<item␣subscription='both'␣jid='bill@shakespeare.lit'>
␣␣␣␣␣␣<version␣xmlns='urn:xmpp:entityver:0'>9ZFZXVP9</version>
␣␣␣␣</item>
␣␣</query>
</iq>
```

Note that in this case there may be three roster items total (and the client only knows about two of them), or there may be two total roster items and the server is informing the client about a change to "bill@shakespeare.lit". Version tokens MUST also be present in roster pushes:

Listing 4: Roster Push

```
<iq from='romeo@montague.lit' id='ah382g678jka7' to='romeo@montague.
    lit/home' type='set'>
  <query xmlns='jabber:iq:roster' ver='ver34'>
    <item jid='tybalt@shakespeare.lit' subscription='remove'>
      <version xmlns='urn:xmpp:entityver:0'>XWE4MUUP</version>
    </item>
  </query>
</iq>
```

7.2 Cache Invalidation

When a client syncs with the server and indicates that it has a version token in its cache that does not match any entity on the server (or when the server wants to remove an entity from the clients cache for any other reason), the server MUST reply with an empty <version/> node. When the client receives such an empty version node it SHOULD purge the entity from its cache. For example, the following would remove the roster item 'bill@shakespeare.lit' from the cache:

Listing 5: Cache invalidation

```
<iq from='romeo@montague.lit' id='56' to='romeo@montague.lit/home'
    type='result>
␣␣<query␣xmlns='jabber:iq:roster'>
␣␣␣␣<item␣subscription='both'␣jid='bill@shakespeare.lit'>
␣␣␣␣␣␣<version␣xmlns='urn:xmpp:entityver:0'/>
␣␣␣␣</item>
␣␣</query>
</iq>
```

Roster pushes that indicate a deleted item MUST also remove the version from the cache (and need not contain an empty <version/> element).

7.3 Partial sync

For very large groups fetching an entire list may not be practical or necessary. For example, one might imagine a large corporation with a shared roster that is too large for its version tokens to be sent up to the server on every sync, or even to download fully the first time. To solve this, servers MAY choose to send down only a part of an entity list in response to

a query (unless the individual EV profile forbids partial list sync). How servers choose what items to return is an implementation detail that is out of the scope of this document. Some suggestions may be found in the Implementation Notes. On subsequent requests for the entity list, the server MAY choose to return more entities (eg. based on changes in its internal selection criteria), however it MUST NOT invalidate cached entities unless they have actually been removed from the list.
XEPs defining entity versioning profiles MUST include a section to indicate if partial sync is supported, and if so, how it will be indicated to the client (and how the client can request a full list). If no mechanism is specified, this is done by adding a boolean "full_list" attribute to the request, eg. a roster request for a partial list looks like:

Listing 6: Roster Request

```
<!-{}- Client -{}->
<iq from='romeo@montague.lit/home'
     id='56'
     to='romeo@montague.lit'
     type='get'>
  <query xmlns='jabber:iq:roster' full_list='false'>
    <item jid='bill@shakespeare.lit'>
      <version xmlns='urn:xmpp:entityver:0'>25P2A7H8</version>
    </item>
    <item jid='anne@shakespeare.lit'>
      <version xmlns='urn:xmpp:entityver:0'>VIZSVF0D</version>
    </item>
  </query>
</iq>

<!-{}- Server -{}->
<iq from='romeo@montague.lit' id='56' to='romeo@montague.lit/home'
    type='result'>
  <query xmlns='jabber:iq:roster' full_list='false'>
    <item subscription='both' jid='bill@shakespeare.lit'>
      <version xmlns='urn:xmpp:entityver:0'>9ZFZXVP9</version>
    </item>
  </query>
</iq>
```

When making a request for a partial list, clients do not need to send up every entity in their cache. Instead they MAY send up just those entities for which they wish to check for updates. The server MUST then respond with any updates for those entities, and MAY also add other entities to the list if desired. If the client requests a partial list but does not indicate that it has anything in its cache, what entities to return (if any) is left up to the server implementation.

7.4 List search

When a client has an incomplete versioned list, it may be beneficial to download more of the list without requesting the full list. To do this, servers which support entity versioning MUST supply a "search" IQ which can be used to discover list items matching a certain criteria. What data to match on (JID, metadata, associated vcards, etc.), and what type of search are left up to the server implementation and MAY be different between profiles.
Search queries are qualified by the 'urn:xmpp:entityver:0:search' namespace and MUST have a 'profile' attribute set to the namespace for which the search is being performed. For instance, searching the roster looks like the following:

Listing 7: Roster search

```
<!-{}- Client -{}->
<iq from='romeo@montague.lit/home'
     id='564'
     to='romeo@montague.lit'
     type='get'>
  <query xmlns="urn:xmpp:entityver:0:search" profile='
      urn:xmpp:entityver:profile:roster:0'>
    Search term
  </query>
</iq>

<!-{}- Server -{}->
<iq from='romeo@montague.lit' id='564' to='romeo@montague.lit/home'
    type='result'>
  <query xmlns="urn:xmpp:entityver:0:search" profile='
      urn:xmpp:entityver:profile:roster:0' type='result'>
    <item subscription='both' jid='matching_search@term.lit'>
      <version xmlns='urn:xmpp:entityver:0'>4YAZ7Y38</version>
    </item>
  </query>
</iq>
```

Search results SHOULD be added to the given list's cache. In this way, the full list does not need to be known.

7.5 Aggregate Tokens

While the version token approach to caching does not require a great deal of state to be stored on the client or the server, it does require a lot more information to be sent by the client when requesting a list of entities. For a very large list which is not likely to have changed, it may be useful know in advance if the roster has changed or not (so that we can avoid sending the large request entirely). To do this, we can request an aggregate version token from the server. This aggregate token is calculated by constructing a string of comma separated "ID:version"

pairs sorted in byte-wise order (because the ID:version pair is constructed before sorting, if two items in the list have the same ID they can still be sorted by the version token), and taking the MD5 hash of the constructed string. The ID in the pair is any ID or key that identifies the entity as defined in its profile (eg. a JID for roster items and most other entities). For example, if the server is calculating the aggregate version token for a roster, it might end up with the following string:

Listing 8: Aggregate token list

```
anne@shakespeare.lit:VIZSVF0D,bill@shakespeare.lit:25P2A7H8
```

Which results in the aggregate token:

Listing 9: Aggregate token

```
0514fc90e6c7981b06bbb2173bb8ef03
```

The actual request is an IQ sent to the server, or entity handling the versioned list which contains a query that specifies the namespace of the list we want to fetch. Eg. to fetch the aggregate token for the roster one would query the server with the query's XMLNS set to 'urn:xmpp:entityver:profile:roster:0':

Listing 10: Roster aggregate token request

```
<!-{}- Client -{}->
<iq to='bill@shakespeare.lit' type='get' id='bill1'>
  <query xmlns='urn:xmpp:entityver:profile:roster:0'/>
</iq>

<!-{}- Server -{}->
<iq to='bill@shakespeare.lit/home' type='result' id='bill1'>
  <query xmlns='urn:xmpp:entityver:profile:roster:0'>
    0514fc90e6c7981b06bbb2173bb8ef03
  </query>
</iq>
```

Because aggregate tokens are OPTIONAL to implement, clients MUST fall back to making a normal list request if any error is returned in response to an aggregate token IQ.
If an aggregate token is requested for a list that may contain more than one type of entity (eg. MUC rooms and pubsub nodes that live on the same component), then the server MUST return the aggregate token constructed with the entire list (rooms and pubsub nodes).
Because aggregate tokens are calculated for the entire list as seen by the client or server, they will never match if partial lists have been downloaded by the client.
Clients are also NOT REQUIRED to check aggregate tokens. However, clients MAY wish to check aggregate tokens before making a roster or MUC request when the cached roster or MUC list is very large. When to check aggregate tokens (if at all) is left up to the implementation.

8 Implementation Notes

Version tokens may not provide enough collision resistance across versioned entities (hereafter simply called ”entities”), and may vary from server to server, and therefore they MUST NOT be used as an entity identifier.
Version tokens SHOULD always be considered opaque to the client (eg. even if the version token is a derivable and consistent hash on the server side, clients should not need to know how the server is calculating the token).
The author RECOMMENDS using 8 character (32-bit) random alphanumeric ASCII strings (eg. AABd7z9T) for version tokens.
If a server which supports this XEP provides an HTTP API which can be used to fetch information about entities (eg. for listing information about MUC rooms that a server provides on the providers web page), the entities version token MAY be used as a weakly validated ETag for any API requests for that entity.
Servers following this specification may choose to send down partial entity lists in response to queries. For the case of rosters one or more of the following may be returned to the requesting entity during the initial roster sync:

- Users that are grouped with the requester in some way. Eg. for a company with a large shared roster which places the requesting client in the ”Marketing Department” group, the server may wish to return roster items that also share that group.
- Users whom the requester has contacted recently or frequently.
- Users that should always be returned as part of server policy.

9 Security Considerations

Client-side caching of entity information across sessions (rather than holding them in memory only for the life of a session) could pose a privacy risk, especially on shared systems. Implementations SHOULD protect cached entity data with strong encryption or other appropriate means.

10 IANA Considerations

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

[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/>.

11 XMPP Registrar Considerations

11.1 Protocol Namespaces

This specification defines the following XML namespace:

- urn:xmpp:entityver:0
- urn:xmpp:entityver:0:search

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].

11.2 Namespace Versioning

If the protocol defined in this specification undergoes a revision that is not fully backwards-compatible with an older version, the XMPP Registrar shall increment the protocol version number found at the end of the XML namespaces defined herein, as described in Section 4 of XEP-0053.

11.3 Entity Versioning Profiles Registry

The XMPP Registrar shall maintain a registry of entity versioning profiles. All EV profile registrations shall be defined in separate specifications (not in this document). Application types defined within the XEP series MUST be registered with the XMPP Registrar, resulting in protocol URNs of the form "urn:xmpp:entityver:profile:name:X" (where "name" is the registered name of the profile and "X" is a non-negative integer).
In order to submit new values to this registry, the registrant shall define an XML fragment of the following form and either include it in the relevant XMPP Extension Protocol or send it to the email address <registrar@xmpp.org>:

```
<profile>
  <name>The name of the entity versioning profile.</name>
  <desc>A natural-language summary of the profile.</desc>
  <listdef>
    The document in which the original list definition is specified.
  </listdef>
```

[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>.

```
  <doc>
    The document in which the EV profile for the list is specified (
        may be the
    same as <listdev/>).
  </doc>
</profile>
```

11.4 Entity Versioning Profiles

This specification defines the following entity versioning profile:

- urn:xmpp:entityver:profile:roster:0

Upon advancement of this specification from a status of Experimental to a status of Draft, the XMPP Registrar [7] shall add the following definition to the entity versioning profiles registry, as described in this document:

```
<profile>
  <name>Roster entity versioning</name>
  <desc>Allows versioning of entities in an XMPP roster.</desc>
  <listdef>RFC 6121</listdef>
  <doc>TODO: Insert this document once it is assigned a number</doc>
</profile>
```

12 XML Schema

TODO

13 Acknowledgements

The original entity versioning proposal was engineered and written by HipChat's Doug Keen.

[7] 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/>.