ejabberd state of the art to implement one-to-many chat services

process one

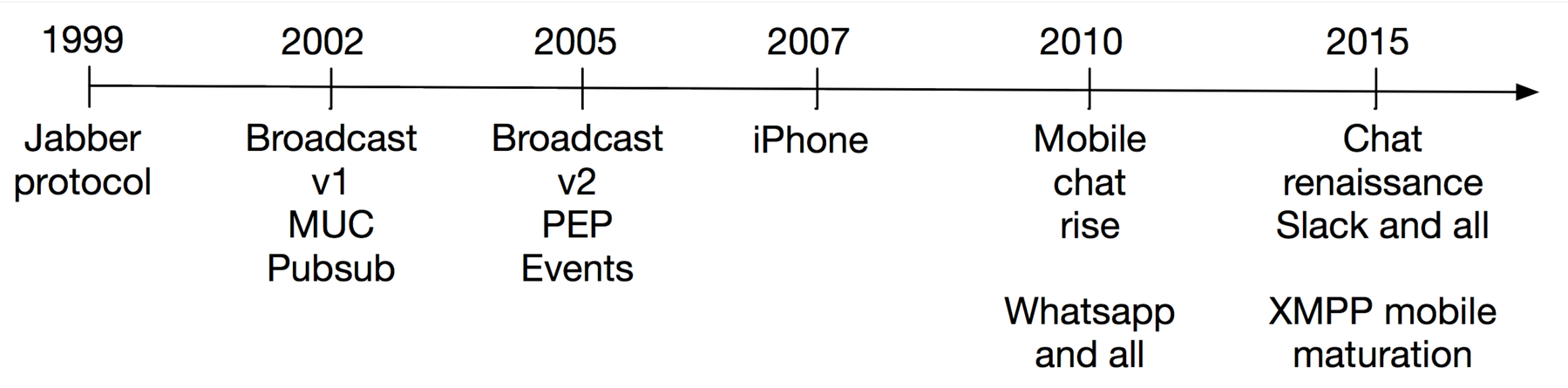
Leader in Messaging and Push Solutions

17th November 2015

Mickaël Rémond <mremond@process-one.net>

One-to-Many chat history in the XMPP context

XMPP Timeline



XMPP Timeline

XMPP was originally designed for one-to-one chat and presence in 1999.

Multiple send

Only way to send messages to multiple users was to send multiple times the same message.

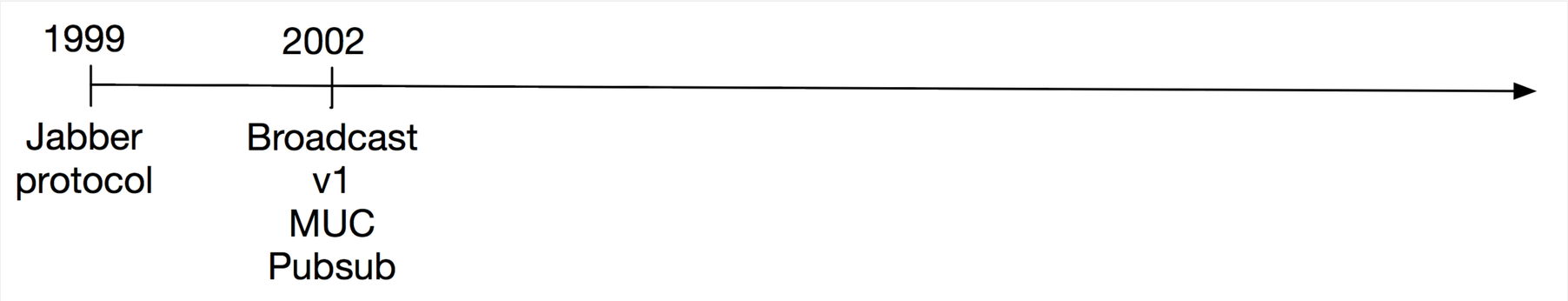
```
    <message to='hildjj@jabber.org/Work'>
      <body>Hello, world!</body>
    </message>
  ...
```

Sending announcements

For admin, there was broadcast command-line tools that were server implementation dependant.

Warn about server maintenance.

XMPP Timeline



2002 is critical year in XMPP. Multiple broadcast features added.

- XEP-0033: Extended Stanza Addressing - Early 2002
    - Specification is designed to send a single message to multiples users:

```
<message to='multicast.jabber.org'>
  <addresses xmlns='http://jabber.org/protocol/address'>
    <address type='to' jid='hildjj@jabber.org/Work' desc='Joe Hildebrand'/>
    <address type='cc' jid='jer@jabber.org/Home' desc='Jeremie Miller'/>
  </addresses>
  <body>Hello, world!</body>
</message>
```

...

- XEP-0045: Multi-User Chat - Late 2002
    - IRC-like specification designing the concept of chat room.

```
<message
    from='hag66@shakespeare.lit/pda'
    id='hysf1v37'
    to='coven@chat.shakespeare.lit'
    type='groupchat'>
  <body>Harpier cries: 'tis time, 'tis time.</body>
</message>
```

...

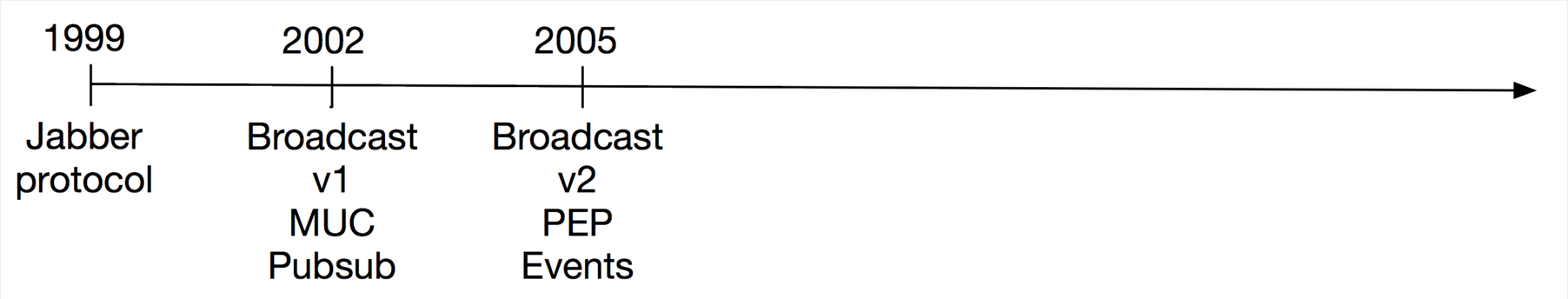
- XEP-0060: Publish-Subscribe - End 2002
    - Based on several iteration of a message broadcast mechanism.
    - Channel oriented broadcast mechanism
    - This is a general framework for broadcasting in XMPP, that rely on previous initiatives (Jabber events, etc.)

```
<iq type='set'
    from='hamlet@denmark.lit/blogbot'
    to='pubsub.shakespeare.lit'
    id='publish1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='princely_musings'>
      <item id='bnd81g37d61f49fgn581'>
        <entry xmlns='http://www.w3.org/2005/Atom'>
          <title>Soliloquy</title>
          <summary>
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them?
          </summary>
          <link rel='alternate' type='text/html'
                href='http://denmark.lit/2003/12/13/atom03'/>
          <id>tag:denmark.lit,2003:entry-32397</id>
          <published>2003-12-13T18:30:02Z</published>
          <updated>2003-12-13T18:30:02Z</updated>
        </entry>
      </item>
    </publish>
  </pubsub>
</iq>
```

...

- ejabberd XMPP server
    - It was also the year we started ejabberd development !

XMPP Timeline



From 2002 to 2005, Pubsub became a central feature for XMPP servers.

To spread clients adoption, new features were introduced to promote implicite subscriptions.

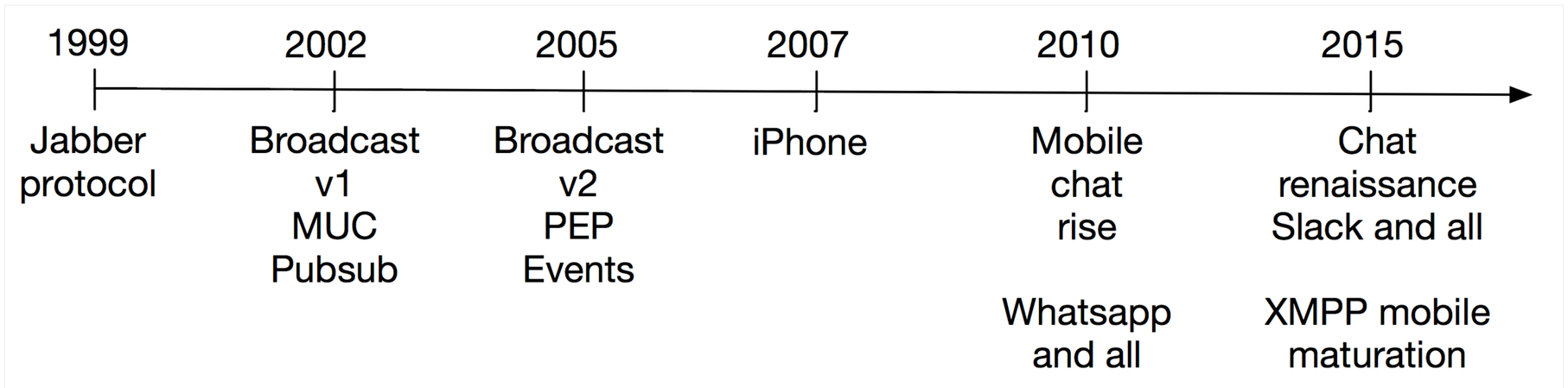
- XEP-0163: Personal Eventing Protocol
    - Introduces roster-based subscriptions.

```
<iq from='juliet@capulet.lit/balcony' type='set' id='pub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='http://jabber.org/protocol/tune'>
      <item>
        <tune xmlns='http://jabber.org/protocol/tune'>
          <artist>Gerald Finzi</artist>
          <length>255</length>
          <source>Music for "Love's Labors Lost" (Suite for small orchestra)</source>
          <title>Introduction (Allegro vigoroso)</title>
          <track>1</track>
        </tune>
      </item>
    </publish>
  </pubsub>
</iq>
```

...

- There has been many domains specific extensions to broadcast dedicated message information type:
    - XEP-0080: User Location
    - XEP-0084: User Avatar
    - XEP-0107: User Mood
    - XEP-0172: User Nickname
    - ...

Pubsub tends to be the central mechanism to broadcast messages in XMPP, acting as indirection layer between Publishers and Subscribers.



When XMPP was designed, there was no mobile at that time and Wifi was only just spreading, most of the chat happened on a physical wire.

Since smartphones became ubiquitous, most chat sessions happen on mobile devices.

Custom protocols and XEP extensions focus on that use case:

- XEP-0184: Message Delivery Receipt
- XEP-0198: Stream Management
- XEP-0280: Message Carbons
- XEP-0357: Push Notifications
- **Custom ProcessOne extensions**, some of them coming before the ideas were added to an official XEP.

What's next in XMPP history ?

Integration with other tools, platform, environments:

- API
- OAuth 2.0

Sending messaging to many users – Use Cases

- **`mod_announce`:** Broadcasting admin messages to multiple users.
- **ejabberd API:** Sending notification messages to users.
- **Extended Stanza Addressing:** Multiple recipients to single messages.
- **Multi-User Chat:** several use cases. This is typically many to many. Everyone is usually both publisher and subscribers.
- **Pubsub:** Broadcasting to interested parties: generally this is One or Few (publishers) to many (subscribers) .

`mod_announce`
, send messages to your user base

Feature reserved to a set of admin users, configurable through ejabberd ACL.

It can be used through XMPP thanks to Ad-Hoc Commands, as defined in XEP-0050.

The broadcast can target either:

- The whole user base. For users that are currently offine, message will be store in offline storage for delivery on reconnection.
- Only online users.

This feature is typically used for broadcasting service messages, e.g. to warn users about a maintenance process.

Using `mod_announce` through Ad-Hoc Commands

Feature discovery:

- Discover server support for commands:

```
<iq type='get' to='localhost'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

...

- IQ result will includes reference to 'commands' server feature:

```
<iq from="localhost" type="result">
  <query xmlns="http://jabber.org/protocol/disco#info">
 …
      <feature var="http://jabber.org/protocol/commands"/>
 …
</query>
</iq>
```

…

Commands list:

- You can retrieve the list of commands

```
<iq type='get' to='localhost'>
  <query xmlns='http://jabber.org/protocol/disco#items'
         node='http://jabber.org/protocol/commands'/>
</iq>
```

...

- IQ result shows, among other commands, that 'announce' commands are enabled (`mod_announce`):

```
<iq from="localhost" type="result">
  <query xmlns="http://jabber.org/protocol/disco#items"
node="http://jabber.org/protocol/commands">
    ...
    <item node="http://jabber.org/protocol/admin#announce" name="Send announcement to all
online users" jid="localhost"/>
    <item node="http://jabber.org/protocol/admin#announce-all" name="Send announcement to
all users" jid="localhost"/>
    <item node="http://jabber.org/protocol/admin#announce-allhosts" name="Send
announcement to all online users on all hosts" jid="localhost"/>
    <item node="http://jabber.org/protocol/admin#announce-all-allhosts" name="Send
announcement to all users on all hosts" jid="localhost"/>
    ...
  </query>
</iq>
```

...

Command info:

- You can retrieve command informations:

```
<iq type='get' to='localhost'>
    <query xmlns='http://jabber.org/protocol/disco#info'
           node='http://jabber.org/protocol/admin#announce'/>
</iq>
```

...

- And receive more details:

```
<iq from="localhost" type="result">
 <query xmlns="http://jabber.org/protocol/disco#info"
        node="http://jabber.org/protocol/admin#announce">
  <identity category="automation" type="command-node"
            name="Send announcement to all online users"/>
  <feature var="http://jabber.org/protocol/commands"/>
 </query>
</iq>
```

...

Execute announce command:

- You can start execution of command with:

```
<iq type='set' to='localhost' id='exec1'>
  <command xmlns='http://jabber.org/protocol/commands'
                    node='http://jabber.org/protocol/admin#announce'
                    action='execute'/>
</iq>
```

...

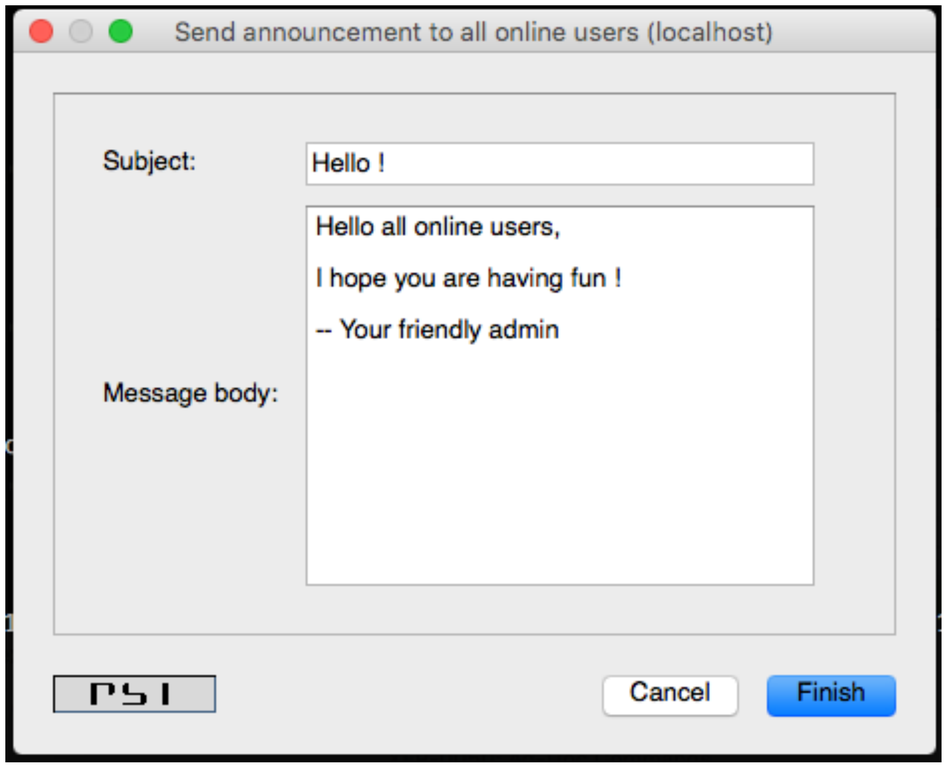
- And you receive form fields to fill out in IQ result:

```
<iq from="localhost" type="result” id="exec1">
<command xmlns="http://jabber.org/protocol/commands" status="executing"
node="http://jabber.org/protocol/admin#announce" sessionid="2015-11-13T14:43:12.831577Z">
  <x xmlns="jabber:x:data" type="form">
  <field type="hidden" var="FORM_TYPE">
    <value>http://jabber.org/protocol/admin</value>
  </field>
  <title>Send announcement to all online users</title>
  <field type="text-single" label="Subject" var="subject"/>
  <field type="text-multi" label="Message body" var="body"/>
  </x>
 </command>
</iq>
```

...

XMPP Client displays form:

- At this stage, client can render the form if it supports ad-hoc commands.
- Here is the resulting form rendered by Psi client:



Submit 'announce' form:

- Here is how the submitted form translates into XMPP:

```
<iq type="set" to="localhost" id="aaf7a">
 <command xmlns="http://jabber.org/protocol/commands"
node="http://jabber.org/protocol/admin#announce" sessionid="2015-11-13T14:43:12.831577Z">
  <x xmlns="jabber:x:data" type="submit">
   <field type="hidden" var="FORM_TYPE">
    <value>http://jabber.org/protocol/admin</value>
   </field>
   <field type="text-single" var="subject">
    <value>Hello !</value>
   </field>
   <field type="text-multi" var="body">
    <value>Hello all online users,</value>
    <value></value>
    <value>I hope you are having fun !</value>
    <value></value>
    <value>-- Your friendly admin</value>
   </field>
  </x>
 </command>
</iq>
```

...

- Result IQ confirms the processing of the broadcast:

```
<iq from="localhost" type="result" id="aaf7a">
  <command xmlns="http://jabber.org/protocol/commands"
           status="completed"
  node="http://jabber.org/protocol/admin#announce"
  sessionid="2015-11-13T14:48:09.302076Z"/>
</iq>
```

...

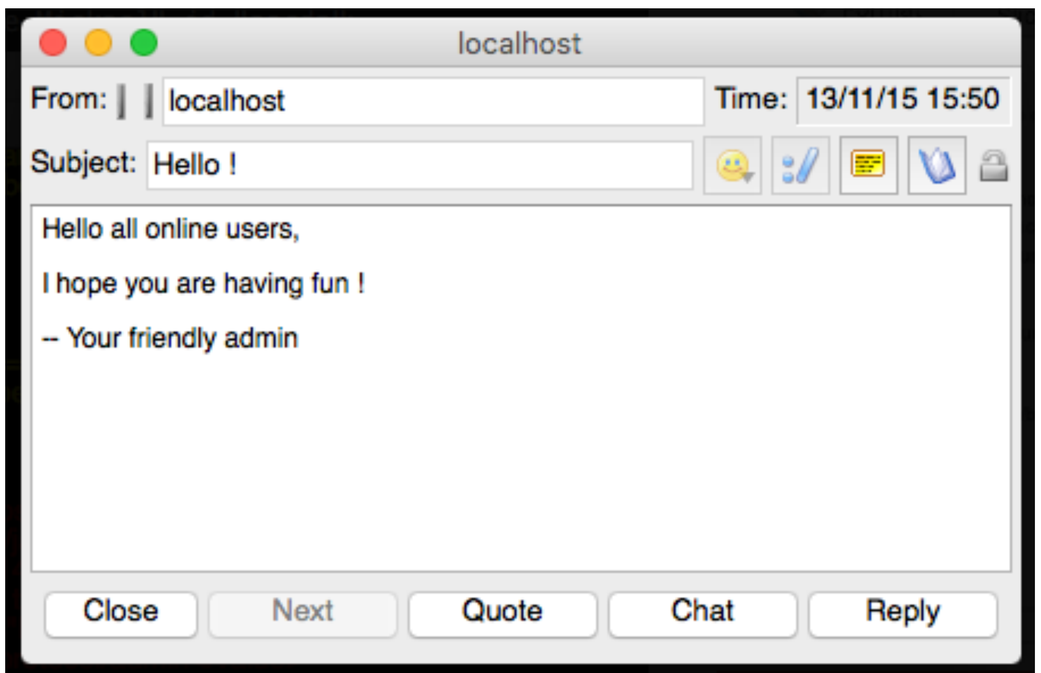
Server executes broadcast:

- As a result a message of type 'headline' is send to all users:

```
<message from="localhost" type="headline">
 <subject>Hello !</subject>
  <body>Hello all online users,


I hope you are having fun !


-- Your friendly admin

</body>
</message>
```

...

- In Psi client, it is displayed in a popup:



Sending message broadcast through ejabberd API

mod_announce to online users work consistently across all our platforms, no matter how they are configured.

For ejabberd SaaS and ejabberd Business Edition, we offer a direct API allowing to broadcast messages to list of users.

This is typically use to send broadcast message to large parts of the user base from our customer backend.

Two types of ReST API for broadcast:

- `send_message` to all connected users. Parameters are:
    - Message Payload.
    - Rate: Number of messages to send per minute.
- `send_message` to list of JID. Parameters are:
    - Message Payload .
    - Rate.
    - List of JIDs.

Extended Stanza Addressing

How does it work ?

- When server supports it, users can use Extended Stanza Addressing to send messages to list of recipients.
- Messages should be send to a "multicast" service performing consistency checks (authorization, etc.) and sending the message to all the recipients.
- Service acts like a message broadcast proxy.
- It is mimicking SMTP headers:
    - to
    - cc
    - bcc
    - replyto
    - noreply

Example:

- Client sends message to local server with address header:

```
<message to='header1.org' from='a@header1.org/work'>
   <addresses xmlns='http://jabber.org/protocol/address'>
     <address type='to'  jid='to@header1.org'/>
     <address type='cc'  jid='cc@header1.org'/>
     <address type='bcc' jid='bcc@header1.org'/>
     <address type='to'  jid='to@header2.org'/>
     <address type='cc'  jid='cc@header2.org'/>
     <address type='bcc' jid='bcc@header2.org'/>
     <address type='to'  jid='to@noheader.org'/>
     <address type='cc'  jid='cc@noheader.org'/>
     <address type='bcc' jid='bcc@noheader.org'/>
   </addresses>
   <body>Hello, World!</body>
</message>
```

...

- Server delivers to local addresses. Here is on example:

```
<message to='to@header1.org' from='a@header1.org/work'>
   <addresses xmlns='http://jabber.org/protocol/address'>
      <address type='to' jid='to@header1.org' delivered='true'/>
      <address type='cc' jid='cc@header1.org' delivered='true'/>
      <address type='to' jid='to@header2.org' delivered='true'/>
      <address type='cc' jid='cc@header2.org' delivered='true'/>
      <address type='to' jid='to@noheader.org' delivered='true'/>
      <address type='cc' jid='cc@noheader.org' delivered='true'/>
   </addresses>
   <body>Hello, World!</body>
</message>
```

...

Multi-User Chat Use Cases

As of today Multi-User chat is used in several contexts, for several types of use cases. :

- IRC-Like community chat: Open interest groups.
- Team collaboration: Private chat room.
- Online events: Moderated rooms.
- Mobile chat placeholders for multi users conversations.

MUC Rooms as IRC-like community chat

This is the original design of the MUC protocol.

The MUC room protocol is presence-based:

- It is made so that you can follow conversation only when you have join a given room.
- It means that, conveniently you can connect on clients without having to join public open MUC room. They can be high traffic.
- Like on IRC, you were not supposed to catch up on messages while your were offline, but could by referring to daily HTML log files.

MUC protocol is presence-based

- You join a MUC room by sending a presence to the MUC room:

```
<presence to="erlang@conference.localhost/Mickaël"/>
```

...

- Your presence is broadcasted to all other occupants (included yourself):

```
<presence from="erlang@conference.localhost/Mickaël" xml:lang="en"
to="test@localhost/MacBook-Pro-de-Mickael">
 <x xmlns="http://jabber.org/protocol/muc#user">
  <item affiliation="owner" role="moderator" jid="test@localhost/MacBook-Pro-de-
Mickael"/>
  <status code="110"/>
  <status code="201"/>
 </x>
</presence>
```

...

- You can leave the room at any time by sending unavailble presence to room:

```
<presence
    to="erlang@conference.localhost/Mickaël"
    type='unavailable'/>
```

...

- You presence unavailable is broadcasted to all occupants:

```
<presence from="erlang@conference.localhost/Mickaël"
      type="unavailable" xml:lang="en" to="test@localhost/MacBook-Pro-de-Mickael">
 <x xmlns="http://jabber.org/protocol/muc#user">
  <item affiliation="owner" role="none"/>
  <status code="110"/>
 </x>
</presence>
```

...

Note that when you disconnect, your presence unavailable is broadcasted automatically to the room by ejabberd.
That's why you are not an occupant of the room when you are offline.

Occupants messages are broadcasted

- After you have joined the room, you can send a message to the room:

```
<message type="groupchat" to="erlang@conference.localhost" id="ab78a">
 <body>Hello</body>
</message>
```

...

- Messages are broadcasted to all other occupants:

```
<message from="erlang@conference.localhost/Mickaël" type="groupchat" xml:lang="en"
to="test@localhost/MacBook-Pro-de-Mickael" id="ab78a">
 <body>hello</body>
</message>
```

...

- MUC messages are extensible, like any XMPP messages. You can add custom content:

```
<message type="groupchat" to="erlang@conference.localhost" id="ab78a">
 <body>Hello</body>
 <x xmlns="p1:custom">
  <datablob>MyData</datablob>
 </x>
</message>
```

...

- That extra content is kept in the broadcasted element and can be leverage by custom clients:

```
<message from="erlang@conference.localhost/Mickaël" type="groupchat" xml:lang="en"
to="test@localhost/MacBook-Pro-de-Mickael" id="ab78a">
 <body>hello</body>
 <x xmlns="p1:custom">
  <datablob>MyData</datablob>
 </x>
</message>
```

...

MUC Rooms as Community Chat are discoverable

- You can browse the MUC room list but discovering item on MUC service:

```
<iq type="get" to="conference.localhost" id="ab35a">
 <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

...

- Service return paginated list of MUC rooms:

```
<iq from="conference.localhost" type="result" to="admin@localhost/MacBook-Pro-de-Mickael"
id="ab35a">
 <query xmlns="http://jabber.org/protocol/disco#items">
  <item name="erlang (1)" jid="erlang@conference.localhost"/>
 </query>
</iq>
```

...

This allows you to find MUC rooms matching your interests.

- Open rooms allows you to discover more information on its state without entering it (disco#info):

```
<iq type="get" to="erlang@conference.localhost" id="ab36a">
 <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>
```

...

- They return the state of the MUC:

```
<iq from="erlang@conference.localhost" type="result" to="admin@localhost/MacBook-Pro-de-
Mickael" id="ab36a">
 <query xmlns="http://jabber.org/protocol/disco#info">
  <identity category="conference" type="text" name="erlang"/>
  <feature var="vcard-temp"/>
  <feature var="http://jabber.org/protocol/muc"/>
  <feature var="muc_public"/>
  <feature var="muc_temporary"/>
  <feature var="muc_open"/>
  <feature var="muc_semianonymous"/>
  <feature var="muc_moderated"/>
  <feature var="muc_unsecured"/>
  <x xmlns="jabber:x:data" type="result">
   <field type="hidden" var="FORM_TYPE">
    <value>http://jabber.org/protocol/muc#roominfo</value>
   </field>
   <field label="Room description" var="muc#roominfo_description">
    <value/>
   </field>
   <field label="Number of occupants" var="muc#roominfo_occupants">
    <value>1</value>
   </field>
  </x>
 </query>
</iq>
```

...

- If you discover the room for items:

```
<iq type="get" to="erlang@conference.localhost" id="ab37a">
 <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

...

- then, open MUC room will list occupants:

```
<iq from="erlang@conference.localhost" type="result" to="admin@localhost/MacBook-Pro-de-
Mickael" id="ab37a">
 <query xmlns="http://jabber.org/protocol/disco#items">
  <item name="Mickaël" jid="erlang@conference.localhost/Mickaël"/>
 </query>
</iq>
```

...

MUC Rooms for team collaboration

For collaboration between team member, rooms need to be a bit more private and transparent about participants.

It can be done simply by configuring your MUC room accordingly. Here are relevant room options:

- You can configure room to prevent them to appear in the public room list: `muc#roomconfig_publicroom`
- You can protect room by password: and `muc#roomconfig_passwordprotectedroom` `muc#roomconfig_roomsecret`
- Make room member only (and manage a member list): `muc#roomconfig_membersonly`
- You can also expose JID of occupants by making room non-anonymous: `muc#roomconfig_whois`

This allows other occupants to see your JID. Client can thus use the real JID to query your VCard and get real name and avatar for example.

Example post for room configuration form as submitted by owner:

```
<iq from='crone1@shakespeare.lit/desktop'
    id='create2'
    to='coven@chat.shakespeare.lit'
    type='set'>
  <query xmlns='http://jabber.org/protocol/muc#owner'>
    <x xmlns='jabber:x:data' type='submit'>
      <field var='FORM_TYPE'>
        <value>http://jabber.org/protocol/muc#roomconfig</value>
      </field>
      <field var='muc#roomconfig_roomname'>
        <value>A Dark Cave</value>
      </field>
      <field var='muc#roomconfig_roomdesc'>
        <value>The place for all good witches!</value>
      </field>
      <field var='muc#roomconfig_enablelogging'>
        <value>0</value>
      </field>
      <field var='muc#roomconfig_changesubject'>
        <value>1</value>
      </field>
      <field var='muc#roomconfig_allowinvites'>
        <value>0</value>
      </field>
      <field var='muc#roomconfig_allowpm'>
        <value>anyone</value>
      </field>
      <field var='muc#roomconfig_maxusers'>
        <value>10</value>
      </field>
      <field var='muc#roomconfig_publicroom'>
        <value>0</value>
      </field>
      <field var='muc#roomconfig_persistentroom'>
        <value>0</value>
      </field>
      <field var='muc#roomconfig_moderatedroom'>
        <value>0</value>
      </field>
      <field var='muc#roomconfig_membersonly'>
        <value>0</value>
      </field>
      <field var='muc#roomconfig_passwordprotectedroom'>
        <value>1</value>
      </field>
      <field var='muc#roomconfig_roomsecret'>
        <value>cauldronburn</value>
      </field>
      <field var='muc#roomconfig_whois'>
        <value>moderators</value>
      </field>
      <field var='muc#maxhistoryfetch'>
        <value>50</value>
      </field>
      <field var='muc#roomconfig_roomadmins'>
        <value>wiccarocks@shakespeare.lit</value>
        <value>hecate@shakespeare.lit</value>
      </field>
    </x>
  </query>
</iq>
```

...

Room history archiving:

The case of the MUC room history is now solved by a new specification.

With XEP-0313: Message Archive Management, MUC room service can be an archive provider.

This is supported by ejabberd since ejabberd 15.09.

MUC rooms for Online Events: Scale and “moderation”

MUC rooms being presence based, a lot of traffic get generated just to broadcast presence packets.

For large rooms, presence broadcast becomes a large bottleneck. Each join / leave / presence change means that it could be broadcasted to thousands of users.

To mitigate that issue, moderated rooms have been created:

- Moderation is enabled with a MUC room option: `muc#roomconfig_moderatedroom`
- They support an option to avoid visitors broadcasting their own presence: `muc#roomconfig_presencebroadcast`
- They support ability to request voice and grant and remove voice to occupants to limit the number of chat messages posted to the room and make the conversation readable.

Note: Moderated rooms are about moderating who can post messages, not about approving or rejecting individual messages. You still need custom extension for that.

MUC rooms for mobile group conversations

With the rise of mobile chat, the most prominent feature has been the use of group conversations.

This is kind of persistent continuous groupchat conversation that you share with friends. Think Whatsapp and all.

There were many difficulties to implement them over groupchat despite similar behaviour, that are now solved:

- Access to archives: In original MUC rooms, message not received while offline where not possible to retrieve.

=> This is now solved with MUC service support Message Archive Management

- Access to PEP status is possible in non-anonymous rooms through Personal Event Protocol Status

Remaining problem: Rooms are still presence-based. It means that it is currently not very compliant with typical mobile behaviour and push notification support.

Solution: Pushing MUC further with membership options

We propose an extension to MUC that does remove the constrain related to MUC being presence based.

Presence is still the core of the protocol and used to show status / availability of participants.

We **leverage membership** feature for long term involvement in a MUC room:

- Currently membership is mainly used to register room nickname.
- We extend it with membership options.

A MUC member can define his membership options. They include for now:

- **A forward JID:** The forward JID is used to send messages to members that are not currently occupant of the room
- **A forward format:** Which can be ‘forward’ to wrap stanza in a forward envelope or ‘none’ to disable the push notifications.

We are considering adding other membership options in the future, like extra tag information e.g. to define a custom sound for notification coming from a conversation.

Making MUC plugins with new MUC related hooks

Message and presence hooks:

```
muc_filter_message(Stanza, MUCState, RoomJID, FromJID, FromNick) -> Stanza | drop

muc_filter_presence(Stanza, MUCState, RoomJID, FromJID, FromNick) -> Stanza | drop
%%...
```

Example:

```
%% Declare module in ejabberd.yml
%% modules:
%%   mod_muc_filter: {}
-module(mod_muc_filter).

-behaviour(gen_mod).

-include("../deps/p1_xml/include/xml.hrl").
-include("logger.hrl").

-export([start/2, stop/1,
         on_muc_message/5
        ]).

start(Host, _Opts) ->
    ?INFO_MSG("MUC filter module started.", []),
    ejabberd_hooks:add(muc_filter_message, Host, ?MODULE, on_muc_message, 50),
    ok.

stop(Host) ->
    ?INFO_MSG("MUC filter module stopped.", []),
    ejabberd_hooks:delete(muc_filter_message, Host, ?MODULE, on_muc_message, 50),
    ok.

on_muc_message(#xmlel{name = <<"message">>, children = Children} = Stanza, _MUCState,
_RoomJID, _FromJID, _FromNick) ->
    ?INFO_MSG("MUC Message: ~p", [Stanza]),
    FilteredChildren =
        lists:map(fun(#xmlel{name = <<"body">>, children = [{xmlcdata, Body}]} = Child) -
>
                          Child#xmlel{children = [{xmlcdata, rewrite(Body)}]};
                      (Child) ->
                          Child
                  end, Children),
    Stanza#xmlel{children = FilteredChildren}.

rewrite(Body) ->
    re:replace(Body,"foo","bar",[global,{return,binary}]).
%%...
```

Pubsub: broadcasting various type of events

Typical pubsub use cases are:

- Broadcast news to subscribers
- Monitoring
- Internet of Things (IoT)

Typical pubsub workflow is as follow:

- Owner create a pubsub node:

```
<iq type='set'
    to='pubsub.localhost'
    id='create1'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
      <create node='mynode'/>
    </pubsub>
</iq>
```

...

- and receive confirmation:

```
<iq from="pubsub.localhost" type="result" to="test@localhost/MacBook-Pro-de-Mickael"
id="create1">
 <pubsub xmlns="http://jabber.org/protocol/pubsub">
  <create node="mynode"/>
 </pubsub>
</iq>
```

...

- Subscriber subscribes to node:

```
<iq type='set'
    to='pubsub.localhost'
    id='sub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe
        node='mynode'
        jid='admin@localhost'/>
  </pubsub>
</iq>
```

...

- and receive confirmation:

```
<iq from="pubsub.localhost" type="result" to="admin@localhost/MacBook-Pro-de-Mickael"
id="sub1">
 <pubsub xmlns="http://jabber.org/protocol/pubsub">
  <subscription node="mynode" subscription="subscribed" subid="5A7AA540C5289"
jid="admin@localhost"/>
 </pubsub>
</iq>
```

...

- Publisher publish to node:

```
<iq type='set'
    to='pubsub.localhost'
    id='publish1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='mynode'>
      <item id='bnd81g37d61f49fgn581'>
        <body>Les sanglots longs des violons</body>
      </item>
    </publish>
  </pubsub>
</iq>
```

...

- Publisher receive publish acknowledgement:

```
<iq from="pubsub.localhost" type="result" to="test@localhost/MacBook-Pro-de-Mickael"
id="publish1">
 <pubsub xmlns="http://jabber.org/protocol/pubsub">
  <publish node="mynode">
   <item id="bnd81g37d61f49fgn581"/>
  </publish>
 </pubsub>
</iq>
```

...

- and subscribers receive pubsub message:

```
<message from="pubsub.localhost" type="headline" to="admin@localhost">
 <event xmlns="http://jabber.org/protocol/pubsub#event">
  <items node="mynode">
   <item id="bnd81g37d61f49fgn581">
    <body>Les sanglots longs des violons</body>
   </item>
  </items>
 </event>
</message>
```

...

Pubsub API / plugins

Pubsub is a large protocol. Hard to optimize for performance for all use cases.

ejabberd provides nodetree and node API to:

- allow full control on the pubsub behaviour, while being compliant from client perspective.
- enable performance optimisations.

Pubsub will be described with more details during dedicated talk.

Conclusion

ejabberd covers the whole scope of the message broadcasting spectre.

It is adequate to address the challenges of modern messaging platform:

- Adequately support the scope of features needed for mobile messaging
- Can be tuned / optimized for scalability

Online services are relying on proven tool to build robust and scalable messaging service.