The Real Time Web

(Building It)
the real time web?
Social Objects

- The things we exchange
- Media: Writing, photos, audio, video, ...
- Metadata: Location, relationship data, personal data
Client

are we there yet?

bored

no.

are we there yet?

bored

no.

are we there yet?

bored

no.

are we there yet?

bored

no.

? no ? no ? no

distracted

distracted

distracted

distracted

Web

DB
Client

we're there

arrival

Server

let me know when we're there

we're there
problems and solutions
What are our goals?

- Real time
- Low cost
- Asynchronous
- Simple
HTTP?

• Works fantastically for web browsers.
• Hard to scale for frequent updates.
• Hard to scale for frequent polling.
• Asks the wrong question: “what happened in the past?”
HTTP Ping/Push?

- NAT traversal breaks desktop clients.
- HTTP time-outs
- Inconsistent APIs
- Authentication
SMTP?

• No verifiable authentication scheme
• No consistent approach to API design
• Servers not tuned for high volume of low-latency messages
Comet?

- GMail
- Successful for web-based clients
- One connection per user
- Requires polling
- Stretching the HTTP metaphor
Jabber

- Fulfills all the goals
- Open
- Simple (if you’re careful)
first steps
architecture

- Not p2p
- Client-to-server
- Clients maintain persistent connections
- Federation looks exactly like email
- Servers communicate directly
it’s all just xml

- a jabber session is simply two streaming XML documents
- the spec defines common elements
- but you can extend it at any time
- similar to html, but with a larger vocabulary
jabber addressing

• addresses look like email addresses:
  user@domain.com

• but you can omit the username (node):
  domain.com

• or you can include a resource:
  user@domain.com/mydevice
jabber federation

- i.e., why it’s not spammy
- s2s - server to server
- support for verified authentication of servers using SSL
- dialback authentication
- explicit whitelist by default
messages

• primary jabber payload. email.

• the simplest message is an addressed stanza with a body (the message)

• subject, alternate message types are available but client ui is poorly implemented

• we can send html and atom, too
presence

- online, offline, chat, away, xa
- the intellectual pivot between optimizing for store and forward (http, smtp) and streams of data
tracking presence

- we can subscribe and unsubscribe
- also allow, deny, or block requests
the roster

• your social connections
• maintains presence subscriptions
• maintains your whitelist
• synonomous with your buddy list on MSN / AIM / YahooIM
jabber basics
navigating jabber

- Nearly 200 specs defining all sorts of behaviour. Ignore them.
- Unless you need them.
- Core & IM: RFCs 3920 & 3921
stanzas

• XML Elements
• Shared attributes:
  • to, from, id, type
<message from="romeo@montague.net" to="juliet@capulet.com">
  <body>Hi!</body>
</message>
<message from="romeo@montague.net/orchard"
to="juliet@capulet.com"
id="msg:montague.net,1">
  <body>Hi!</body>
</message>
<presence from="romeo@montague.net"
to="juliet@capulet.com" />
presence

<presence from="romeo@montague.net"
to="juliet@capulet.com">
    <show>away</show>
    <status>swooning</status>
</presence>
<presence from="romeo@montague.net"
to="juliet@capulet.com"
type="subscribe" />
presence

<presence from="juliet@capulet.com"
to="romeo@montague.net"
type="subscribed" />
<presence from="romeo@montague.net" to="juliet@capulet.com" type="unsubscribe" />

<presence from="juliet@capulet.com" to="romeo@montague.net" type="unsubscribed" /></presence>
• Information Query
• Enables the roster, discovery, XEPs
• Should almost always be hidden behind libraries.
building applications
taking stock

- we can send/receive messages
- add / remove contacts
- track presence
- let's build something!
define bot behaviour

• what does your bot do?
• conversational
• informational
• recorder
define api behaviour

• what does your api look like?
• atom?
• custom xml with namespaces?
• we'll dig in a bit more later.
write the behaviour

• build a class or interface that handles messages
• test the class with mock xmpp stanzas
• mock out sending functions in your xmpp lib so you don't need an active connection
class MyHandler
  def on_message(message)
    puts "Got Message:"
    puts "from #{message.from}"
    puts "to #{message.to}"
    puts "body #{message.body}" 
    out = Jabber::Message.new(message.from, "got it!")
    yield out
  end
end
event handler

client = Jabber::Simple.new('user@ex.com', 'pwd')
handler = MyHandler.new

client.received_messages do |message|
  handler.on_message(message) do |out|
    client.send(out)
  end
end
event loop

client = Jabber::Simple.new('user@ex.com', 'pwd')
handler = MyHandler.new
loop do
  client.received_messages do |message|
    handler.on_message(message) do |out|
      client.send(out)
    end
  end
end
end
Handling presence

```ruby
client.status(:away, "eating")

client.presence_updates do |update|
  friend = update[0]
  presence = update[2]
  puts "#{friend.jid} is #{presence.status}"
end
```
handling presence

<presence from="user@ex.com">
  <show>away</show>
  <status>eating</status>
</presence>
rosters

• should ideally be handled by libraries
• if not, at least aim for being able to fetch your roster using a library call
Roster roster = connection.getRoster();
Collection<RosterEntry> entries = roster.getEntries();
for (RosterEntry entry : entries) {
    System.out.println(entry);
}
process management

- Very difficult to run Jabber clients from non-persistent connections
- Run a persistent daemon that manages your Jabber connection
next steps
PubSub

- A mechanism for Publishing and Subscribing to feeds
- Like presence subscriptions, but for data
PubSub

- Over-specified
- Don't try to read the spec if you can avoid it
- Thankfully, the concept is simple and the core implementation is easy
PubSub Subscribe

<iq type='set' from='francisco@denmark.lit/barracks' to='example.com' id='sub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe
      node='http://example.com/updates'
      jid='francisco@denmark.lit/barracks'/>
  </pubsub>
</iq>
<iq type='result' from='example.com'
    to='francisco@denmark.lit/barracks' id='sub1'>
    <pubsub xmlns='http://jabber.org/protocol/pubsub'>
        <subscription
            node='http://example.com/updates'
            jid='francisco@denmark.lit/barracks'
            subscription='subscribed'/>
    </pubsub>
</iq>
PubSub Messages

<message from='example.com'

to='francisco@denmark.lit/barracks' id='foo'>
<body>blah</body>
<event xmlns='http://jabber.org/protocol/pubsub#event'>
<items node='http://twitter.com/xmpp'>
  <item id='http://twitter.com/blaine/statuses/324236243'>
    <entry>...</entry>
  </item>
</items>
</event>
</message>
PubSub Unsubscribe

<iq type='set' from='francisco@denmark.lit/barracks' to='example.com' id='unsub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <unsubscribe
      node='http://example.com/xmpp'
      jid='francisco@denmark.lit'/>
  </pubsub>
</iq>
PubSub Confirmation

<iq type='result'
   from='example.com'
   to='francisco@denmark.lit/barracks'
   id='unsub1'/>
• Personal Eventing via PubSub
• You can think of it as exactly the same as regular PubSub, except the node becomes relative to a user (full JID)
<iq type='set' from='francisco@denmark.lit/barracks' to='user@example.com' id='sub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe node='http://example.com/updates' jid='francisco@denmark.lit/barracks'/>
  </pubsub>
</iq>
Federation

• Social Network Federation
• Use PubSub to allow users on remote services to subscribe to each other
• Breaking down walled gardens
best practices
• keeping the api simple
• choosing where to use jabber
• atom over xmpp
scaling techniques
scalability?

- Jabber scales well out of the box for relatively small numbers of contacts.
- Stops working at around 35k contacts, due to roster presence behaviour.
- Come online, find out what everyone's presence is.
components

• In order to work around this, we use the component protocol, XEP-0114
• Horrendously bad documentation
• But thankfully it's simple
components

- A component allows you to handle everything for a JID, or a whole domain.
- You can turn off the roster!
- Without roster management, we now assume that our bot is *always* online.
components

- Components work just like client-to-server bots, but we need to handle presence ourselves.
- The easiest way is to do the following...
components

client = Jabber::Component.new('example.com')
client.connect("127.0.0.1")
client.auth("secret")
client.add_presence_callback do |presence|
  case presence.type.to_s
  when nil, 'unavailable': save_presence(presence)
  when 'probe': send_online(presence.from)
  when 'subscribe': send_subscribed(presence.from)
  end
end
end
horizontal scaling

- Many processes across machines
- Need a queuing mechanism
- We use Starling
- ActiveMQ, RabbitMQ, MySQL, local HTTP push are also viable options
horizontal scaling

client.add_message_callback do |message|
  incoming_message_queue.push message
end

loop do
  message = message_queue.pop
  client.send message
end
client connections

• If you plan to offer Jabber user accounts, you'll need to scale to many persistent connections.

• Thankfully, most Jabber servers do this part out of the box.
tools
Client Libraries

- Ruby: xmpp4r & xmpp4r-simple
- Java: Smack
- Python: twisted-words
- Perl: Net::Jabber
- Javascript: JSJaC
Jabber Servers

- ejabberd (recently 2.0)
- openfire
- Jabber XCP
Other Tools

- Debugging: Psi (cross-platform, fully featured)
- PubSub: Idavoll
Jabber-enabled

- livejournal
- twitter
- jaiku
- gtalk / gmail
- chesspark
- fire eagle (soon!)
questions?