Real Time Web

Push technologies overview

Vitaly Kushner
astrails.com
astrails.com/blog

Hi. My name is Vitaly Kushner from Astrails.

We are going to talk about real time web technologies today.

Slides & Transcript

- will be available at our blog at http://astrails.com/blog
- The link to Astrails company site is also available from the Devcon tracks page.
- There will also be a video of the talk at the Devcon site sometime soon.

During the talk I'm going to name things and give pointers but there is no need to write it down, the slides will be available from our blog later today, including the transcript of the talk.

The blog is at http://astrails.com/blog

There is also link to the Astrails web site from the Devcon site.

Some time later there will also be a video of the talk on the devcon site.

Astrails

2005

At Astrails we create web applications.

Web Applications

Complex

Web Applications

Mostly complex Web Applications. We usually use Ruby on Rails and lately also Node.js.

Requirements

Since 2005 when we started working on our first real web project we did tens of projects, so we observed how typical requirements changed over time.

Real Time

One of the growing trends in the recent years is more and more of a "real time" functionality that is required.

By real time I don't mean constant guaranteed response time or anything like that. We are not talking about embedded systems etc.

On the web "real time" means something different. It usually means that an application user gets new content in the application w/o refreshing the browser, and in the case this new content is coming from another user it is made available "close to real time", which usually means fast enough so that you can have a 'chat' for example. Typical delays will be under 1 sec.

Gmail

2004

GMail was one of the first widely known applications that popularized the technology back in about 2004-2005. At the time almost no web apps did this.

Today, it seems like every other one does.

Lets see how we could implement such a system.

Polling

No talk about the subject can escape the simplest technologies of all.

Polling

Simple

It’s main advantage is that it is simple.

Polling

- call server every x seconds
- server responds with new data if available
- or empty/negative response if not

You send request to the server every couple of seconds. You get new data back if its available or empty response otherwise.

Many times there is nothing wrong about polling. If you know "Campfire" a group chat application from 37signals was implemented with poling. Application was sending a request to check for new messages every 5 seconds or so.

Their "poller service" is written in C, so it can handle tons of connections using either threads or async networking programming style.

Don't discard polling right out of the start. It might just do the job, especially at the beginning when you are building your minimally viable product. You can always rewrite this part of the application later once you have lots of users. Remember that scaling problems are usually good problems. It means you are growing, so those are the good problems to have ;)

Polling

- resource intensive
- slow response

On the other hand polling is definitely not the best way to implement it in many respects.

It is resource intensive. it wastes bandwidths and CPU cycles on both client and server.

It also has quite a significant delay between the event and its delivery to the client.
It will be about half the polling interval on average and up to and above the polling interval in the worst case.

What other options do we have?

LongPoll

Long poll is a variation on the simple polling technique.

LongPoll

- keep the connection open until data available
- send data as soon as available and close the connection
- client comes back immediately

The difference is that in the case of no-data available the server will not immediately return negative response. Instead the server will keep the connection open and wait for new data to become available. Once the data comes it it is sent as a response to the long polling connection. Once client gets a response (or looses connection) it immediately issues another long polling request.

Note that in this case we have no polling interval, we call the server again immediately after we receive the data.

LongPoll

- uses less requests / bandwidth
- faster response

Long poll has a advantages over the regular poll.

- it uses much less bandwidth and it issues less requests.
- faster response. most of the time you have a pending connection
  from the client, so when new data comes in you usually able to push it to the
  client immediately.

LongPoll

More Complex

On the other hand it is more complex to get right.

LongPoll

- keeping connection open - more RAM
- detecting disconnects / reconnects

It uses more RAM to keep all the persistent connections.

You need to detect disconnects. Since you are not "talking" to the client you might not notice if the client disconnected. You will only get notified when you try to write into the socket to send the data to the client.

You might also get a new connection from a client while you still think you have the previous connection open.

LongPoll

- firewall / proxy issues

It also has issues with firewalls and proxies.

Long polling is what we used at Astrails at the beginning of 2006 when
we had to implement a chat server with some custom application functionality.
One of the problems that we had to deal with is that if you keep your
connection open with no data flowing through it, firewalls will usually discard
the connection state after some time. So the server still has to return
negative answer to the client if no new information becomes available after
some suitable timeout. At the time a safe timeout for this was around 25-30 seconds.

LongPoll

- Browser concurrent connections limit (RFC 2616 section 8.1.4)

Another problem with LongPoll is that browsers limit the number of connections that you can keep open to the same server.

RFC 2616 (HTTP/1.1) section 8.1.4 “Practical Considerations” states that “A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy.”

So if you keep one of them persistent for the purpose of LongPoll it means you have only one connection left for the other things like issuing ajax requests and downloading assets.

Note that at the present time most browsers have much this limit higher then 2, so this problem became less relevant.

Http Streaming

- never close the connection
- use HTTP chunked encoding to keep sending data
- send ‘noop’ pings every X seconds to keep the connection from being closed by proxies and firewalls

Another option is to use HTTP Streaming using chunked encoding to send dat to the client.

The connection is kept open and is only re-opened if it disconnects.

Server can send empty/noop data “packets” to deal with firewall and proxy issues.

Adobe FLASH

- open TCP connection
- do anything you want ;)

Yet another option is to use Flash ;)

You can open TCP connection to your server and do almost anything you want.

Adobe FLASH

- not a standard
- is not universally available
- Flash is DEAD ;)

The problem of course is that Flash is not a web standard, its not universally available. For example it is not available on all iOS devices and not installed by default on many systems.

Also with all the recent mac/ios/android flash controversy it seems to be on its way out.

Web Sockets

- It IS going to be a W3C standard
- bi-directional full-duplex persistent TCP
- good browser support
- flash fallbacks available. example: https://github.com/gimite/web-socket-js

Which brings us to Web Sockets.

WebSockets is an emerging standard for bi-directional dull duplex TCP communication channel between a browser and a web server.

It is a new protocol which starts as a regular HTTP connection which is then ‘upgraded’ to a web socket connection.

Major browsers support WebSockets in their recent versions and there are fallback libraries available for older browsers that emulate websockets using flash.

Which one to use?

So, with all those options available, which one should you use?

ALL of them ;)

A somewhat surprising answer is that you should use almost all of them ;)

The trick is to use a library that will choose the best method based on the browser support and network conditions.

Socket.IO

http://socket.io

Socket.IO is one such library. Its great strength is that it will choose the most capable transport at runtime, without it affecting the API.

Available transports

- WebSocket
- Adobe® Flash® Socket
- AJAX long polling
- AJAX multipart streaming
- Forever Iframe
- JSONP Polling

Available transports:

- WebSocket
- Adobe® Flash® Socket
- AJAX long polling
- AJAX multipart streaming
- Forever Iframe
- JSONP Polling

Desktop browsers

- Internet Explorer 5.5+
- Safari 3+
- Google Chrome 4+
- Firefox 3+
- Opera 10.61+

Desktop Browsers:

- Internet Explorer 5.5+
- Safari 3+
- Google Chrome 4+
- Firefox 3+
- Opera 10.61+

Mobile Browsers

- iPhone Safari
- iPad Safari
- Android WebKit
- WebOs WebKit

Mobile Browsers:

- iPhone Safari
- iPad Safari
- Android WebKit
- WebOs WebKit

server side

```
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });

  socket.on('my other event', function (data) {
    console.log(data);
  });

});
```

This is an example of the server side code. As you can see we wait till we have connection with a new client, send them ‘news’ even with “hello world” payload. and define handler to a custom “my other event” event which just dumps the payload to the log.

you can define any number of events and payload can be any json object.

client side

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost');

  socket.on('news', function (data) {
    console.log(data);

    socket.emit('my other event', { my: 'data' });

  });
</script>
```

This is an example of the client side that talks to the server from the previous slide.

We establish connection and wait for the ‘news’ event, then we send the ‘my other event’ back.

As you can see the programming with Socket.io seems to be very simple. The hard part is that all this is completely asynchronous and such code tends to be complex to write and maintain, on the other hand there are no better alternatives to async for the server push support on the client.

What about the server side?

OK, but how do you implement the server side of things?

There are basically 2 kinds of options.

Smart Server

- custom app server
- maintains app state
- application business logic
- persistence

First option is what I call a “Smart server”

This is what we implemented back in 2006 at Astrails when we needed a chat server that will also handle payment metering and some other application level things.

Such a server will usually perform authentication, and application level business logic. Might also implement persistence and other application related things.

Smart Server

- javascripts: Node.js
  http://nodejs.org/
- ruby: event machine
  http://rubyeventmachine.com/
- python: Tornado
  http://www.tornadoweb.org/
- python: Twisted
  http://twistedmatrix.com/trac/

You can implement the Smart Server in almost any kind of server side language.

An example wold be a javascript server using node.js, or ruby server using event machine, or python server using Twisted or Tornado.

Architecture A

- Single server
- regular ‘pull’ requests
- push functionality

You can implement your whole application inside your smart push server.

The problem is that such servers usually require you to use async programming style which makes it harder then usual to do simple things.

So, for example, in case of Ruby it is harder to implement your whole application inside Event Machine then to implement a regular Ruby on Rails application.

The notable exception is javascript with Node.js. If you write your application using Node.js, it is fully async to begin with and so it natively supports push.

So this architecture is more frequently chosen when the runtime environment is async, like Node.js or Erlang.

In case of more traditional runtime environments like Ruby or Python another architecture is commonly used.

Architecture B

- regular app server (Rails, Django etc)
- separate push server (event machine, Node.js, twisted)

In this architecture we divide our applicaiton into ‘regular’ and ‘push’ parts.

The regular part is implemented using the regular web application framework like Ruby on Rails.

The ‘push’ part of the application is then implemented by an additional Push Server using one of the technologies we talked about.

In this case client can talk directly to the Push Server, or to the regular App Server. App Server can contact the Push Server to pass data to the clients.

Dumb Server

- generic code
- no application business logic
- usually no persistence
- communication through the app server
- direct relay possible

Another alternative architecture is to use what I call a ‘Dumb Server’.

In this case instead of implementing smart Push Server we can use an existing generic push server that is driven by our application server. So this is a variation on the Architecture B, but w/o any kind of application specific code on the Push Server side.

In this architecture client usually talks to the application server which validates and if needed passes the information to the other clients through the push server.

direct relay though the push server is also possible, but in this case a much stronger validation is required on the client side.

CometD

- Dojo Foundation
- http://cometd.org/
- https://github.com/cometd/cometd
- implemented in Java
- client bindings for many languages
- Bayeux protocol

There are many options available for the “dumb” push server.

One is CometD from Dojo Foundation.

It is a comet server implemented in Java with client bindings for many languages like ruby, python, javascript etc.

There is an extensive documentation and tutorials.

The protocol on the wire is an implementation of Bayeux

Bayeux protocol

- http://svn.cometd.com/trunk/bayeux/bayeux.html
- standard client to comet server protocol

Bayeux is an attempt to standardize the browser interaction with a comet server with the intent of re-use of the client side code with different comet server implementations and tries to cover all the relevant use cases, which means it is quite bloated and complex.

It is based around channels that you can subscribe to and send / receive messages but it is much more to it then just that.

Nginx HTTP Push

- http://pushmodule.slact.net
- Basic HTTP Push Relay Protocol
  http://pushmodule.slact.net/protocol.html

NginX is a super fast HTTP Server from Russia. It took internet by storm from being unknown to a present server market share of more then 10% within just a couple of years.

It has a module that implements HTTP Push.

Instead of Bayeux it implements a much simpler Basic HTTP Push Relay Protocol.

Hosted Solutions

- http://pusher.com
- http://beaconpush.com
- http://www.pubnub.com
- https://www.hydna.com

The common problem of the solutions like CometD and Nginx HTTP Push is that you need to setup/install, configure and maintain them. My personal preference is either custom Node.js based server if we really need a ‘smart server’, or an outsourced hosted solution for a ‘dumb server’.

Hosted Solutions

- http://parse.com
- http://stackmob.com
- http://www.spire.io
- http://www.applicasa.com
- many more...

Another kind of 3rd party solutions provide much more then just push services. They are mostly targeting mobile application developers and they enable creating a server-less mobile application.

They can provide stuff like persistence, user profiles, registrations and also some kind of server push technology, p2p messaging etc.

Pusher

server side (ruby)

```
Pusher['my-channel'].trigger('my-event',
                             {'message' => 'hello world'})
```

client side (javascript)

```
var channel = pusher.subscribe('my-channel');
channel.bind('my-event', function(data) {
  alert('Received my-event with message: ' + data.message);
});
```

Both Pusher and Pubnub APIs are centered around channels that you can subscribe to and send/receive events.

Pubnub

publish

```
PUBNUB.publish({
    channel : "hello_world",
    message : "data"
})
```

subscribe

```
// listen to events
PUBNUB.subscribe({
    channel : "hello_world",
    callback : alert
})
```

Server doesn't have to be the one publishing. In this example for the pubnub we can both publish and subscribe to the channel on the client side.

Authentication

- Authorization endpoints
- Random long channel names

One of the differences between pusher and pubnub is how they deal with security.

With pusher you have to implement an endpoint in your application that will be called by the Pusher service when a new client connects to a “private” channel

With PubNub’s way of just giving your channels long unpredictable channel names, then every client that knows the name is authorized by definition.

Client publishing

One thing you will need to decide is whether to enable clients to publish directly into the channels.

You can choose to enable it, in which case you will need to do much stricter validation on the client, and if you want to have ‘special’ server messages you will have to either use separate channels for that or sign your messages on the server so that clients can verify the origin.

another option is to disable client publishing altogether. In this case clients only talk to the app server which will validate and pass the messages to the other clients through the push server.

Recommendations

My personal choice is to go with a hosted "dumb server" solution like Pusher or Pubnub. I usually prefer hosted to roll-your-own solutions as it saves me lots of time that would otherwase be wasted on IT.

For those reasons exactly we chose Heroku as our deployment platform of choice at Astrails. Since moving to Heroku we saved literally tens of hours that were spent on IT across all the projects that we have to run or maintain.

Both Pusher and Pubnub are available as Heroku addons, so adding them to your Ruby on Rails applicaiton is literally just a click away.

Conclusions

As you can see the technology advanced and matured enough for it to be really easy to implement real time server push in your applications. To the point that it is usually a no brainer whether to include push in your web app or not.

Many web applications we use every day can benefit from real time features sparing us repetitive browser refreshing.

Q&A

Vitaly Kushner
astrails.com
astrails.com/blog