Realtime Web @HuffingtonPost

Websockets, SockJS and RabbitMQ

Adam Denenberg
VP Engineering
@denen
adam.denenberg@huffingtonpost.com

Huffington Post

- 500 MM PVs/week
- 12 MM UVs/week
- 200MM+ Comments, 2MM Comments per week on average
- Strong community

Huffpost Live

- 12 hour live streaming network
- Bring the community into the conversation
- Real time commenting across our 30+ live segments per day from NY and LA
- Real time segment transitions across our live stream
- Real time updates of content below the video player
- Browser refresh was not really an option, we needed to push
- Could not guarantee everyone was on HTML5 browsers
- DEMO

Tech Stack

- Ruby/Rails - CMS and APIs
- Backbone.js - Client UI Framework
- Erlang - Websockets and AMQP bridge
- MongoDB - Database
- Memcache - Caching
- Varnish - Edge caching
- Elastic Search - Searching

Realtime Messages

- Comments are being ingested from our central commenting platform
- Video transitions are being initiated by our production team via our internal CMS
- Resources below video player are being pushed and reordered in realtime by producers
- Various inputs to publish a realtime message, needed a generic solution that could accommodate all these needs without too much burden on the publishing app

Some options we looked at

- Node.js / Socket.io
- SockJS
- EM-Websocket
- CometD
- There are infinitely more not listed here

Results

- Node.js / Socket.io
  - Didnt want a flash fallback
  - Was not crazy about the maturity level of node or the concurrency story for multi-core (it didnt exist)
  - Required persistent backend to scale horizontally, i believe only Redis is supported
  - Focus was changing to engine.io
- EM-Websocket
  - Wasnt very confident that ruby could scale and handle the concurrency but we had a lof of Ruby experience
- CometD
  - Only really offered a long-polling option, we wanted to be able to take advantage of websockets for browsers that supported it and not require an upgrade later on
  - Websocket support buggy and not fully supported
- SockJS
  - No flash fallback
  - Auto fallback to xhr-polling, JSONP, etc if browsers dont support websockets
  - no change in code for different browsers
  - native websocket client support
  - Nice support for load balancers and no shared state
  - written in Erlang :)

Decision?

- SockJS :)
- Integrated the sockjs-client javascript API into our backbone application
- Tested (and using) native websocket client on iOS, Android and Adobe Flash (AIR)
- Worked with our Loadbalancers

What is SockJS?

SockJS is a browser JavaScript library that provides a WebSocket-like object. SockJS gives you a coherent, cross-browser, Javascript API which creates a low latency, full duplex, cross-domain communication channel between the browser and the web server.

Under the hood SockJS tries to use native WebSockets first. If that fails it can use a variety of browser-specific transport protocols and presents them through WebSocket-like abstractions.

Load Balancing in SockJS

- Session URL => *URL/prefix/server/session*
- From SockJS Protocol:

The session between the client and the server is always initialized by the client. The client chooses `server_id`*, which should be a three digit number: 000 to 999. It can be supplied by user or randomly generated. The main reason for this parameter is to make it easier to configure load balancer - and enable sticky sessions based on first part of the url.*

Second parameter `session_id` *must be a random string, unique for every session.*

- http://mydomain.com/myprefix/050/1y3d3roe/websocket

Comments Workflow

- Comments at the Huffington Post are all moderated, both by machine learning technology and humans using an internal service and set of APIs
- Comments are either auto-rejected, auto-approved, or placed into a manual moderation queue where they are manually approved
- Realtime comments was one of our primary use cases for websockets
- We bridged the workflow between the Websocket infrastructure and the comment infrastructure by building an AMQP bridge, which essentially consumed every approved comment and then became a message producer (similar to shovel but we needed to do some transformation)

CMS Workflow

- Producer in control room manages the realtime web portal. Decides when to transition videos to the next segment
- Producer in control room manages the resource well below video and reorders as needed
- The CMS becomes a producer of a new message to initiate global state change of application
  - Leveraged AMQP/EventMachine inside CMS apps

Some challenges before we started

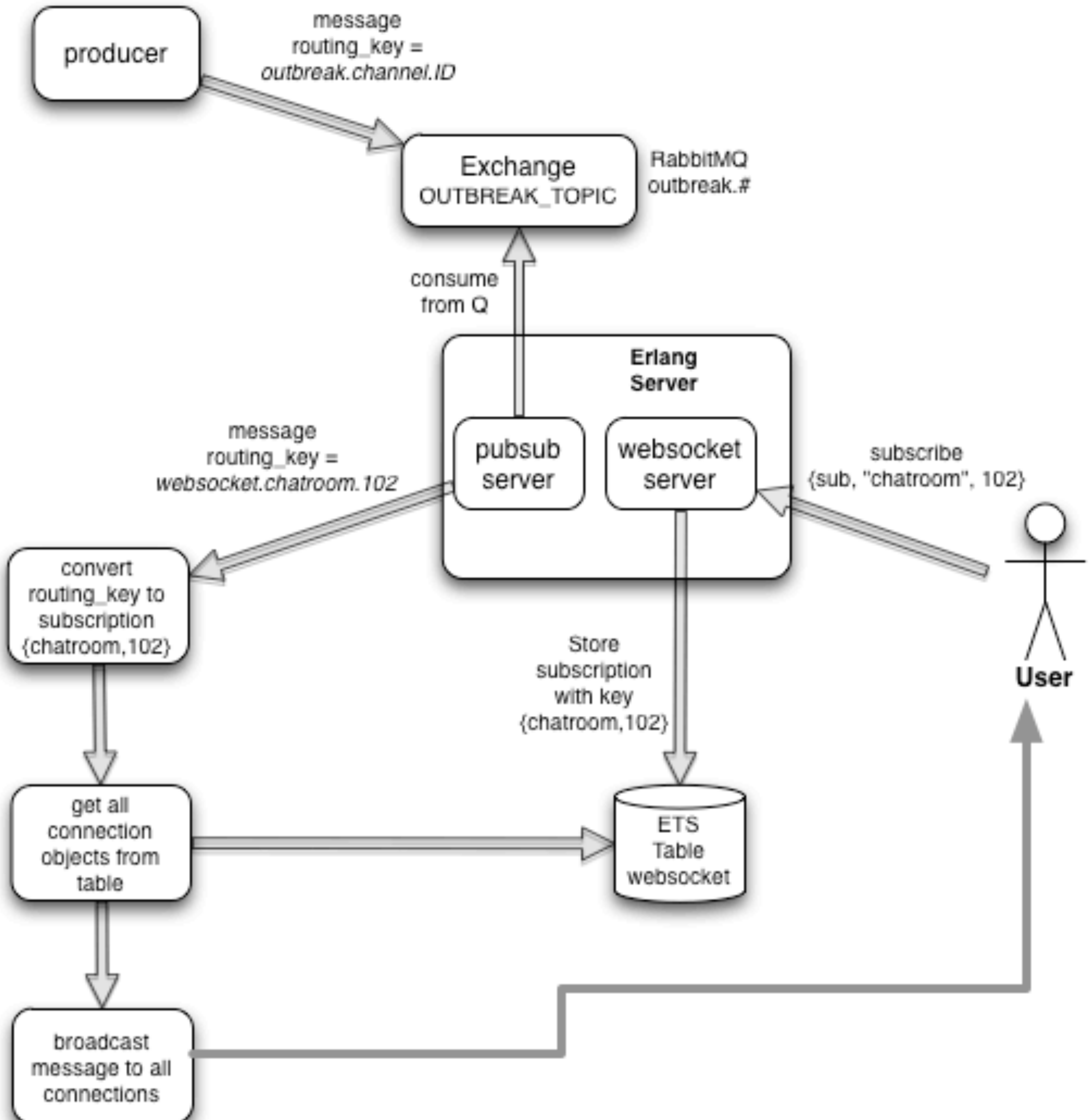
- Nobody knew erlang, and we didn't have a lot of time to build the platform
- Native support for websockets in the load balancers was very new and virtually beta code
- We were concerned about message latency. Our model is relatively low throughput low latency
- We didnt know if it would work :)

Outbreak

- We decided to name it Outbreak
- A set of infrastructure middleware components that allowed a generic mechanism to publish and subscribe
- Built with the mindset of being reused more broadly as time went on, didnt want it built too specific for our exact use case

Concept

- Outbreak is a very simple but generic concept
- Consumers wait for messages for the channels they are subscribed to
- Producers send messages to a predefined RabbitMQ topic
- Outbreak bridges the two so consumers and producers can know nothing about each other or care how messages are delivered


producer
message
routing_key =
outbreak.channel.ID
Exchange
OUTBREAK_TOPIC
RabbitMQ
outbreak.#
consume
from Q
Erlang
Server
pubsub
server
websocket
server
message
routing_key =
websocket.chatroom.102
subscribe
{sub, "chatroom", 102}
convert
routing_key to
subscription
{chatroom,102}
Store
subscription
with key
{chatroom,102}
User
get all
connection
objects from
table
ETS
Table
websocket
broadcast
message to all
connections

Subscribing

- We built a very simplistic json structure that allowed the clients to communicate with the backend
- We allow 3 actions, ‘sub’, ‘unsub’, ‘query’
- format of the payload is { “action” : “sub”, “channel” : “chatroom”, “id” : 333 }
- Sub subscribes to the given channel and id
- Unsub unsubscribes the user from given channel and ID
- Query simply returns all of your active subscriptions

Subscribing

- When a subscription or unsubscription is received we store it in an ordered ETS table
- We store the SockJS connection object along with the channel and ID requested (we anchor the Tuple with Channel and ID since this is faster with ets:select())

```
subscribe(C,I,Conn) ->
  Rec = {{outbreak_util:tostring(C),outbreak_util:tostring(I),Conn},Conn},
  ets:insert(?WS_ETS_TABLE,Rec)
------------------------------------------------
unsubscribe(C,I,Conn) ->
  Key = {outbreak_util:tostring(C),outbreak_util:tostring(I),Conn},
  ets:delete(?WS_ETS_TABLE,Key),
```

Subscribing

- The Conn object from SockJS is special because it allows us to simply extract from the ETS table and call Conn:send() on it
- Users Conn object only lives on one node, no shared state
- We will see in the publishing slides how we use this to simply loop through all matching connections for a given Channel / ID combo

Publishing

- Currently we leverage RabbitMQ as our publishing queue
- We rely very heavily on the concept of Routing Keys and Topics
- We dont require any SockJS node to be aware of any other node.

Publishing

- Topics are leveraged so that all nodes receive a copy of the message, this prevents having to share state
- When a message is published it is published to a single Topic used by Outbreak with a routing key in the format of *prefix*.channel.id
- All outbreak nodes subscribe to a single topic named *prefix*.# where prefix is arbitrary and just a namespace
- In RabbitMQ '#' means any level of routing key
- The routing key is critical when publishing and determines which subscribers get the message

Quick Example Subscribe

- 2 users want to listen to a chatroom , UserA and UserB, each get sent to a different sockjs node
- They both send the payload to the server in the format { “action” : “sub”, “channel” : “chatroom”, “id” : 103 }
- Our server inserts 1 record to the ETS table on each node with the SockJS session object and the subscription {chatroom, 103}

Quick Example Publish

- Moderator in the backend decides to publish a message to chatroom 103
- He publishes a message to RabbitMQ Topic using the routing key outbreak.chatroom.103
- The consumer on both SockJS nodes receives a message on the Topic with a routing key outbreak.chatroom.103
- Our server converts that to Channel=chatroom and ID=103
- Each server queries ETS for sessions matching {chatroom, 103}
- We call Conn:send(msg) on the object in the ETS table

Some Challenges

- This model suits us but we are bound by the performance of a single rabbit server
- Monitoring RabbitMQ from our code took a lot of testing but now it works great and is quite robust. We can shut down rabbit nodes and the server recovers gracefully (thank you monitor())
- Native mobile clients needed to use native websockets which meant implementing our own heartbeats.
- I love Erlang, I do not love making a release :) That was a long battle but now works great.

Performance

- We got SockJS to 100,000 connections pretty easily with sub second latency. This required a fair bit of tuning
- +P, sysctl, etc.
- SockJS has a major performance flaw right now in that it JSON encodes every message, needs to be refactored to encode once publish many, will improve perf greatly
- Refactoring some message passing overhead with JSON issues can probably bring SockJS way higher

Tune your kernel

```
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
kernel.sem = 250 32000 100 128
net.core.rmem_default = 262144
net.core.rmem_max = 8388608
net.core.wmem_default = 262144
net.core.wmem_max = 8388608
net.core.netdev_max_backlog = 8192
net.core.somaxconn = 8192
net.ipv4.ip_local_port_range = 1024 65000
net.ipv4.tcp_tw_reuse = 1
```

Max Ports

in vm.args :

Increase number of processes

+P 512000

Increase number of concurrent ports/ sockets

-env ERL_MAX_PORTS 512000

Whats next?

- Team working on open sourcing outbreak
- Would like to build in such a way that the message bus was a configurable "adapter" so you can use ActiveMQ, RabbitMQ, ZeroMQ, etc. Allow developers to build adapters and just have an API
- Expose publishing as an HTTP interface
- Team will work on fixing some SockJS performance issues
- We are hiring :)

Questions ?

Adam Denenberg

adam.denenberg@huffingtonpost.com

@denen