Eventually Consistent HTTP with Statebox and Riak

Author: Bob Ippolito (@etrepum)
Date: November 2011
Venue: QCon San Francisco 2011

Introduction

This talk isn't really about web. It's about how we model data for the web.

HTTP itself is not the interesting part of our systems. Our systems are mostly JSON over HTTP at the network boundary, nothing too clever!

Mochi's Business

- We provide platforms for Flash game developers
- Ads, analytics, virtual currency, social, scores, etc.
- Terabytes of data to report on

Just a few years ago…

- Millions of tuples was big
- Scale up vertically
- Single master SQL databases
- (Still works great for most companies)

Why was this easy?

- ACID is cheap on a single node
- Efficient to establish a total ordering for events
- Single node systems do not have network partitions!
- Most businesses can probably still get away with this

Why does this break?

- Availability is important (we're global!)
- Too expensive to scale vertically
- Schema evolution is hard
- Sharding not always possible, and rarely fun

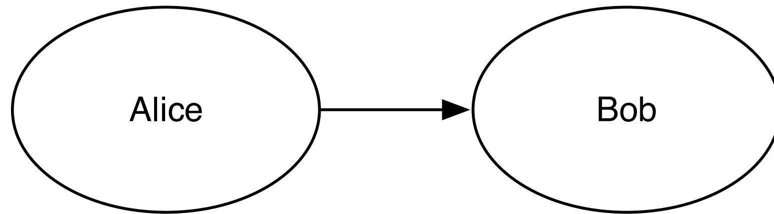
Case Study: Friendwad

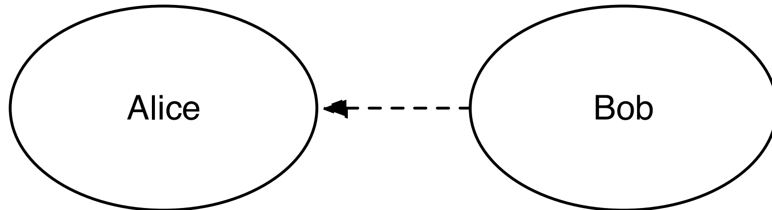
- A social graph aggregator
- MochiGames, Facebook, Twitter, Myspace (oops!)
- Original implementation built on Mnesia
- Mnesia causes us pain

Friendwad Data Model

- Twitter-like social digraph
- Each user has a unique id
- following: user ids that this user follows
- followers: user ids that follow this user

Friendwad Diagram





Mnesia Implementation

- Table in mnesia for user records (id, following, followers)
- Multi-row transaction for each graph change
- At least two rows in each transaction, possibly more (third-party import)

Why not Mnesia?

- Mnesia issues beyond the scope of this talk :)
- Anyway, we decided to migrate to Riak

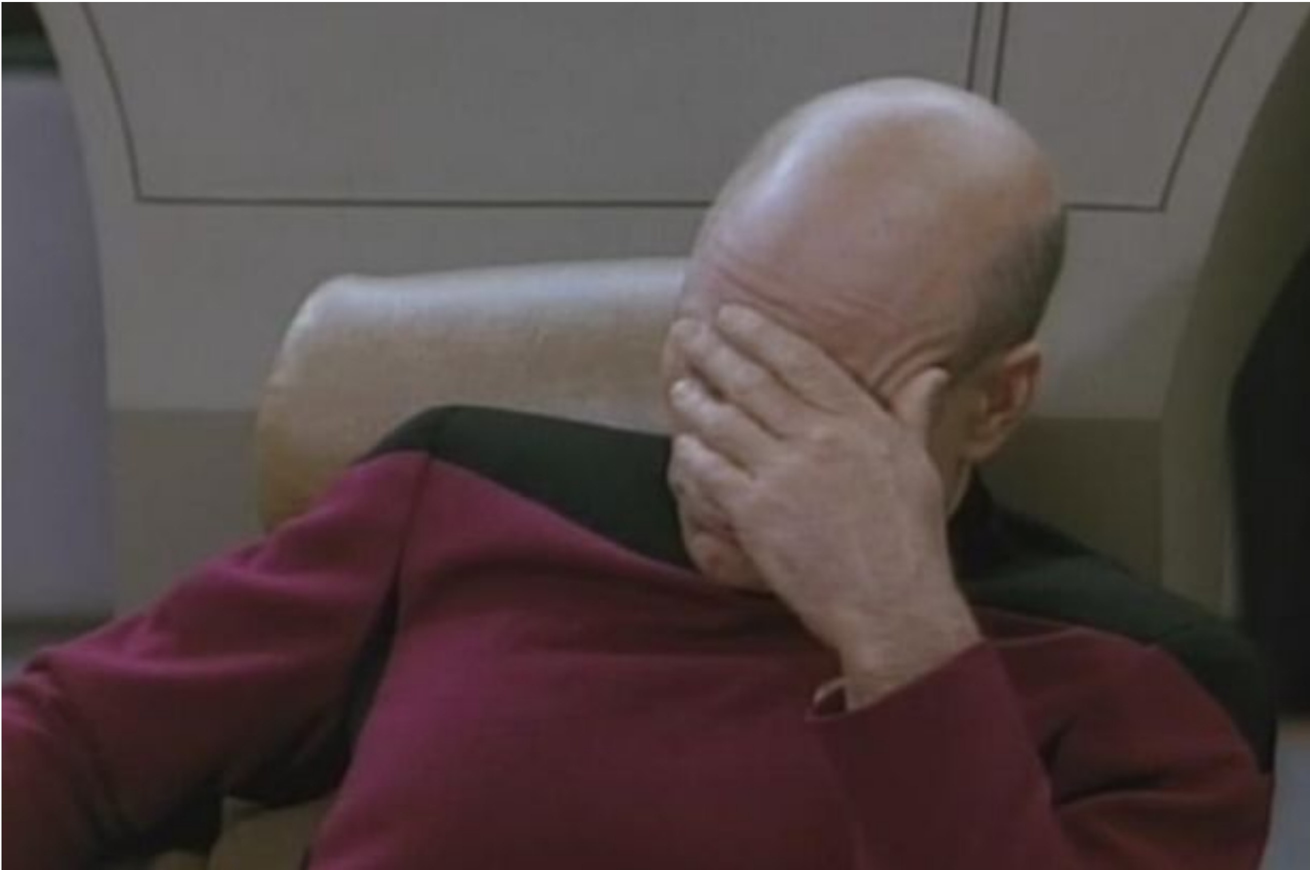
Riak

- Great solution for many of our data problems (thanks Basho!)
- Distributed eventually consistent key-value store
- But not a complete solution

Riak Migration

- The simplest thing that could possibly work (incorrectly)
- … appears correct with serialized glasses
- Riak not transactional even for changes to a single row

Riak Migration Continued



Eventual Inconsistency

- Popular user claimed they were missing entries in "followers"
- Verified that they were missing by looking at our analytics built from our transaction logs
- Especially non-transactional with `allow_mult=false`!
- My face probably still has a palm-shaped dent

Version Terminology

- Client `a` reads version `o` (original state)
- Transform from client `a` on `o` produces version `ao`
- Think function application `a(o())`

Adding a friend [1]

Original state `o` for `alice`, `bob` on read

```
id        | alice   | bob     |
followers | []      | []      |
following | []      | []      |
version   | o       | o       |
```

Adding a friend [2]

Write modified `bob` at version `ao`

```
id        | alice   | bob     |
followers | []      | []      |
following | []      | [alice] |
version   | o       | ao      |
```

Adding a friend [3]

Write modified `alice` at version `ao`

```
id        | alice   | bob     |
followers | [bob]   | []      |
following | []      | [alice] |
version   | ao      | ao      |
```

Interleaving for Fail

- To simulate failure we need multiple concurrent operations
- `a` is `alice → bob`
- `b` is `bob → carol`

Concurrency Pains [1]

`alice → bob` (a) initial state

```
id        | alice    | bob      | carol |
followers | []       | []       | []    |
following | []       | []       | []    |
version   | o        | o        | o     |
```

Concurrency Pains [2]

`bob → carol` (b) initial state (all look same!)

```
id        | alice   | bob     | carol |
followers | []      | []      | []    |
following | []      | []      | []    |
version   | 0       | 0       | 0     |
```

Concurrency Pains [3]

`bob → carol` writes to `bob`

```
id        | alice   | bob     | carol |
followers | []      | []      | []    |
following | []      | [carol] | []    |
version   | o       | bo      | o     |
```

Concurrency Pains [4]

`alice` → `bob` writes to `alice`

```
id        | alice   | bob     | carol |
followers | []      | []      | []    |
following | [bob]   | [carol] | []    |
version   | ao      | bo      | o     |
```

Concurrency Pains [5]

`alice → bob` writes to `bob`

```
id        | alice   | bob     | carol |
followers | []      | [alice] | []    |
following | [bob]   | []      | []    |
version   | ao      | ao      | o     |
```

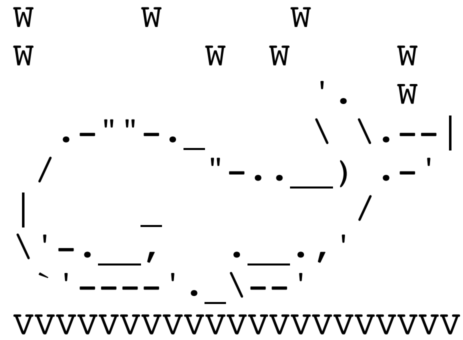
Concurrency Pains [6]

`bob → carol` writes to `carol`

```
id         | alice   | bob     | carol |
followers | []      | [alice] | [bob] |
following | [bob]   | []      | []    |
version   | ao      | ao      | bo    |
```

FAIL

Concurrency ruins everything.



Sibling Rivalry

- If `allow_mult` is on, the next read of `bob` will have two siblings (`[ao, bo]`) because they descend from the same vector clock.
- Default strategy is "last write wins", also known as pain

Simple fix?

Merging `ao` and `bo` is easy! Just union over `followers` and `following`.

Simple fix? NOPE!

But edges are not insert-only! That ruins everything.

It's better, but any inconsistency is just pain waiting to happen.

Fix all of the things

- Turn on `allow_mult=true`
- Implemented statebox in anger to solve the rest of the problem

Statebox Design Philosophy

- Adding code to Riak should be avoided (maintenance)
- The only option is to resolve conflicts on read
- Growth should be bounded and configurable
- Doesn't need to be language agnostic
- Minimize magic

What's Statebox?

- Opaque container
- Serializes current state
- With recent operations
- Provides merge operation
- Monad-like (not important)

Statebox Terminology

- `op()` :: N-ary function reference plus N-1 arguments
- `event()` :: `{timestamp(), op()}`

```
{fun ordsets:add_element/2, [kitten]}
```

Statebox Internals

- Designed to be used with Erlang's external term format (`term_to_binary`)
- Serializes function references, so is bound to exported code
- Prototyped in friendwad, but immediately extracted
- Open sourced because I couldn't find anything else like it

Statebox Theory

- Statebox algorithm can be used as-is with any eventually consistent KV store
- Similar to paper on CRDT (Convergent / Commutative Replicated Data Types)
- Stores current value plus a (configurably) bounded event queue
- Event queue is bound by length and can expire events by age

Declarative (ordsets)

```
Add = fun ordsets:add_element/2,
Empty = statebox:new(fun () -> [] end),
A = statebox:modify({Add, [a]}, Empty),
B = statebox:modify({Add, [b]}, Empty),
AB = statebox:merge([A, B]),
statebox:value(AB) =:= [a, b].
```

Composable

```
Empty = statebox_orddict:from_values([]),
Union = fun statebox_orddict:f_union/2,
A = statebox:modify([Union(following, [b]),
                     Union(followers, [c])],
                    Empty),
B = statebox:modify([Union(following, [b]),
                     Union(followers, [d])],
                    Empty),
AB = statebox:merge([A, B]),
statebox:value(AB) =:= [{followers, [c, d]},
                        {following, [b]}].
```

Statebox Example [1]

```
A     :: [kitten]
[{1, Union([kitten])}]
```

Statebox Example [2]

```
A     :: [kitten]
[{1, Union([kitten])}]

B     :: [puppy]
[{2, Union([puppy])}]
```

Statebox Example [3]

```
A     :: [kitten]
[{1, Union([kitten])}]

B     :: [puppy]
[{2, Union([puppy])}]

[A,B] :: [kitten, puppy]
[{1, Union([kitten])},
 {2, Union([puppy])}]
```

Statebox Merge

- B is newer, so use its value as the basis
- Merge sort event queues
- Apply ops in order from the beginning

Statebox Merge [1]

Use B's value (arbitrarily newest)

```
[puppy]
```

```
Value = [puppy]
```

Statebox Merge [2]

Apply ops oldest to newest (T=1)

```
union([kitten], [puppy])
```

```
Value = [kitten, puppy]
```

Statebox Merge [3]

Apply ops oldest to newest (T=2)

```
union([puppy], union([kitten], [puppy]))
```

```
Value = [kitten, puppy]
```

statebox_riak wrapper

```
%% bob → alice, bob → carol
S = statebox_riak:new([{riakc_pb_socket, P},
                       {expire_ms, 5000},
                       {max_queue, 50}]),
Union = fun statebox_orddict:f_union/2,
statebox_riak:apply_bucket_ops(
    <<"users">>,
    [{[<<"alice">>, <<"carol">>],
      Union(followers, [bob])},
     {[<<"bob">>],
      Union(following, [alice, carol])}],
    S).
```

Restrictions

- Operations must be repeatable (idempotent unary operation)
- Repeatable if and only if `F(V) = F(F(V))`
- Old operations in the queue are replayed in-order on merge, but seeded with newer data

Repeatable Operations

- Most set operations
- Most dictionary operations
- NOT most list operations (ordered lists may be ok!)
- NOT most integer operations

Non-repeatable ops?

- Many can be transformed to repeatable operations
- `statebox_counter` is one example

statebox_counter

- Represent a counter as an ordered list of events
- `[{{Timestamp, Ident}, Delta}]`
- `Ident` is just a unique-ish identifier (node counter, random number, etc.)
- Well tested proof of concept, but not in production use

counter optimizations

- Prevent unbounded growth by coalescing old events into a single event with a fixed `Ident`
- Events older than this are ignored

Other statebox usage

- achievements
- scorewad (via recordset)

achievements

- Manages achievements earned in games
- orddict of `{Achievement, Timestamp}`
- Stores to two keys: `User`, `User_Game`

achievements orddict

Store oldest entry for achievement.

```
f_store_min(Key, New) ->
    {fun ?MODULE:op_store_min/3, [Key, New]}.

op_store_min(Key, New, D) ->
    orddict:update(
        Key,
        fun (Old) -> min(Old, New) end,
        New,
        D).
```

scorewad

- Manages high score boards for > 15,000 games
- Keeps top 50 scores per game for day, week, month, all time
- Also stores scores per user for social leaderboards
- Built recordset to migrate some of this to riak + statebox

recordset

An optionally fixed-size ordered set of complex terms.

- User defined identity
- User defined sorting
- Optional and efficient fixed-sizedness

recordset example (trivial)

```
Empty = recordset:new(fun erlang:'=:='/2,
                      fun erlang:'<'/2,
                      [{max_size, 2}]),
Full = lists:foldl(fun recordset:add/2,
                   Empty,
                   lists:seq(300, 400)),
[399, 400] =:= recordset:to_list(Full).
```

What's next?

- statebox already does what we want it to
- More helper modules or projects will be added over time

Better than Statebox?

- We'd all be better off if this kind of data structure was built-in to the database
- Higher level APIs! KV is fine but I want more from my database
- Redis-like features, but concurrent and multi-node

Why Riak could do it better

- Simple clients: DB can reconcile state before return
- Efficiency: Can store less data (ring state, forced serialization, vclocks)

Questions?

- Twitter: @etrepum
- Mochi Media: www.mochimedia.com
- Slides: etrepum.github.com/statebox_qconsf_2011
- git.io/statebox
- git.io/statebox_riak
- git.io/recordset