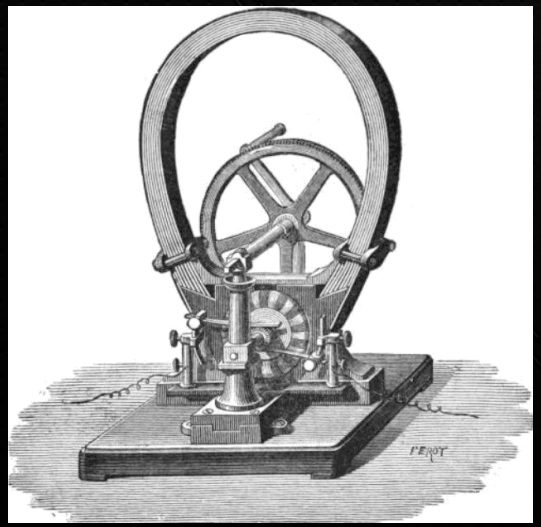
Dynamo: not just datastores

Susan Potter

strangeloop
·2011·

September 2011



Overview: In a nutshell

Figure: "a highly available key-value storage system"

Overview: Not for all apps

Overview: Agenda

- Distribution & consistency
- *t* - Fault tolerance
- N, R, W
- Dynamo techniques
- Riak abstractions
- Building an application
- Considerations
- Oh, the possibilities!

Distributed models & consistency

- Are you *on* ACID?
- Are *your apps* ACID?
- Few *apps require* strong consistency
- Embrace BASE for high availability

Distributed models & consistency

- Are you *on* ACID?
- Are *your apps* ACID?
- Few *apps require* strong consistency
- Embrace BASE for high availability

Distributed models & consistency

- Are you *on* ACID?
- Are *your apps* ACID?
- Few *apps require* strong consistency
- Embrace BASE for high availability

Distributed models & consistency

- Are you *on* ACID?
- Are *your apps* ACID?
- Few *apps require* strong consistency
- Embrace BASE for high availability

Distributed models & consistency

B A S E

Basically Available Soft-state Eventual consistency

Distributed models & consistency

B A S E

Basically Available Soft-state Eventual consistency

</RANT>

t-Fault tolerance: Two kinds of failure

FAILURE

Failures are divided in two classes:
Those who thought and never did, and those who did and never thought.

t-Fault tolerance: More on failure

- **Failstop**

  fail in well understood / deterministic ways

- **Byzantine**

  fail in arbitrary non-deterministic ways

t-Fault tolerance: More on failure

- **Failstop**

  fail in well understood / deterministic ways

  t-Fault tolerance means $t + 1$ nodes

- **Byzantine**

  fail in arbitrary non-deterministic ways

t-Fault tolerance: More on failure

- **Failstop**

  fail in well understood / deterministic ways

  t-Fault tolerance means $t + 1$ nodes

- **Byzantine**

  fail in arbitrary non-deterministic ways

t-Fault tolerance: More on failure

- **Failstop**

  fail in well understood / deterministic ways

  t-Fault tolerance means $t + 1$ nodes

- **Byzantine**

  fail in arbitrary non-deterministic ways

  t-Fault tolerance means $2t + 1$ nodes

CAP Controls: N, R, W

- N = number of replicas

  must be $1 \leq N \leq nodes$

- R = number of responding read nodes

  can be set per request

- W = number of responding write nodes

  can be set per request

- Q = quorum "majority"

  set in stone: $Q = N/2 + 1$

- Tunability

  when $R = W = N \implies strong$

  when $R + W > N \implies quorum$ where $W = 1, R = N$ or $W = N, R = 1$ or $W = R = Q$

CAP Controls: N, R, W

- N = number of replicas

  must be $1 \leq N \leq nodes$
- R = number of responding read nodes

  can be set per request
- W = number of responding write nodes

  can be set per request
- Q = quorum "majority rules"

  set in stone: $Q = N/2 + 1$
- Tunability

  when $R = W = N \implies strong$

  when $R + W > N \implies quorum$ where $W = 1, R = N$ or $W = N, R = 1$ or $W = R = Q$

CAP Controls: N, R, W

- N = number of replicas

  must be $1 \leq N \leq nodes$
- R = number of responding read nodes

  can be set per request
- W = number of responding write nodes

  can be set per request
- Q = quorum "majority rules"

  set in stone: $Q = N/2 + 1$
- Tunability

  when $R = W = N \implies strong$

  when $R + W > N \implies quorum$ where $W = 1, R = N$ or $W = N, R = 1$ or $W = R = Q$

CAP Controls: N, R, W

- N = number of replicas

  must be $1 \leq N \leq nodes$
- R = number of responding read nodes

  can be set per request
- W = number of responding write nodes

  can be set per request
- Q = quorum "majority rules"

  set in stone: $Q = N/2 + 1$
- Tunability

  when $R = W = N \implies strong$

  when $R + W > N \implies quorum$ where $W = 1, R = N$ or $W = N, R = 1$ or $W = R = Q$

CAP Controls: N, R, W

- N = number of replicas

  must be $1 \leq N \leq nodes$
- R = number of responding read nodes

  can be set per request
- W = number of responding write nodes

  can be set per request
- Q = quorum "majority rules"

  set in stone: $Q = N/2 + 1$
- Tunability

  when $R = W = N \implies strong$

  when $R + W > N \implies quorum$ where $W = 1, R = N$ or $W = N, R = 1$ or $W = R = Q$

Dynamo: Properties

- Decentralized

  no masters exist
- Homogeneous

  nodes have same capabilities
- No Global State

  no SPOFs for global state
- Deterministic Replica Placement

  each node can calculate where replicas should be placed
- Logical Time

  No reliance on physical time

Dynamo: Properties

- Decentralized

  no masters exist
- Homogeneous

  nodes have same capabilities
- No Global State

  no SPOFs for global state
- Deterministic Replica Placement

  each node can calculate where replicas should exist for key
- Logical Time

  No reliance on physical time

Dynamo: Properties

- Decentralized

  no masters exist
- Homogeneous

  nodes have same capabilities
- No Global State

  no SPOFs for global state
- Deterministic Replica Placement

  each node can calculate where replicas should exist for key
- Logical Time

  No reliance on physical time

Dynamo: Properties

- Decentralized

  no masters exist
- Homogeneous

  nodes have same capabilities
- No Global State

  no SPOFs for global state
- Deterministic Replica Placement

  each node can calculate where replicas should exist for key
- Logical Time

  No reliance on physical time

Dynamo: Properties

- Decentralized

  no masters exist
- Homogeneous

  nodes have same capabilities
- No Global State

  no SPOFs for global state
- Deterministic Replica Placement

  each node can calculate where replicas should exist for key
- Logical Time

  No reliance on physical time

Dynamo: Techniques

- Consistent Hashing
- Vector Clocks
- Gossip Protocol
- Hinted Handoff

Dynamo: Techniques

- Consistent Hashing
- Vector Clocks
- Gossip Protocol
- Hinted Handoff

Dynamo: Consistent hashing

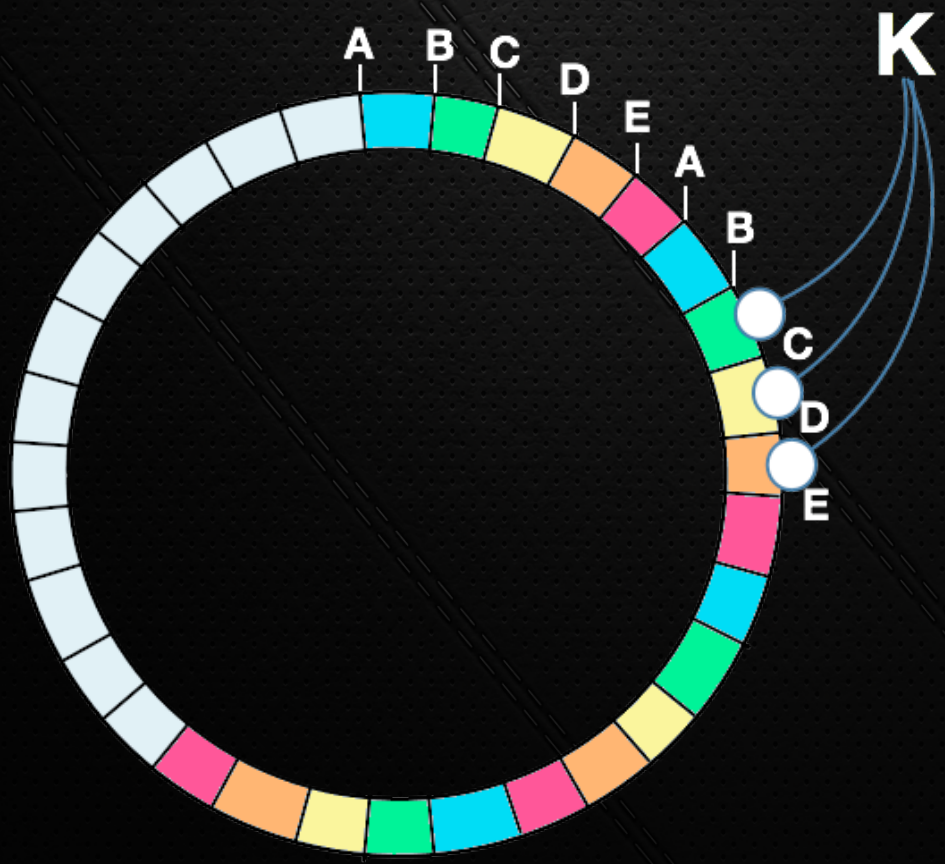


Figure: PrefsList for K is [C, D, E]

Dynamo: Vector clocks

```
-type vclock() ::
[{actor(), counter()}]
```

A occurred-before B if counters in A are
less than or equal to those in B for each actor

Dynamo: Gossip protocol

Figure: "You would never guess what I heard down the pub..."

Dynamo: Hinted handoff

Figure: "Here, take the baton and take over from me. KTHXBAI"

riak_core: Abstractions

- Coordinator

  enforces consistency requirements, performing anti-entropy, `gen_fsm`, coordinates vnodes
- VNodes

  Erlang process, vnode to hashring partition, delegated work for its partition, unit of replication
- Watchers

  `gen_event`, listens for ring and service events, used to calculate fallback nodes
- Ring Manager

  stores local node copy of ring, and callbacks
- Ring Event Handlers

  notified about ring (and cluster?) changes and broadcasts, plus metadata changes

riak_core: Abstractions

- Coordinator

  enforces consistency requirements, performing anti-entropy, `gen_fsm`, coordinates vnodes

- VNodes

  Erlang process, vnode to hashring partition, delegated work for its partition, unit of replication

- Watchers

  `gen_event`, listens for ring and service events, used to calculate fallback nodes

- Ring Manager

  stores local node copy of ring (and cluster?) data

- Ring Event Handlers

  notified about ring (and cluster?) changes and broadcasts, plus metadata changes

riak_core: Abstractions

- Coordinator

  enforces consistency requirements, performing anti-entropy, `gen_fsm`, coordinates vnodes
- VNodes

  Erlang process, vnode to hashring partition, delegated work for its partition, unit of replication
- Watchers

  `gen_event`, listens for ring and service events, used to calculate fallback nodes
- Ring Manager

  stores local node copy of ring (and cluster?) data
- Ring Event Handlers

  notified about ring (and cluster?) changes and broadcasts plus metadata changes

riak_core: Abstractions

- Coordinator

  enforces consistency requirements, performing anti-entropy, `gen_fsm`, coordinates vnodes
- VNodes

  Erlang process, vnode to hashring partition, delegated work for its partition, unit of replication
- Watchers

  `gen_event`, listens for ring and service events, used to calculate fallback nodes
- Ring Manager

  stores local node copy of ring (and cluster?) data
- Ring Event Handlers

  notified about ring (and cluster?) changes and broadcasts plus metadata changes

riak_core: Abstractions

- Coordinator

  enforces consistency requirements, performing anti-entropy, `gen_fsm`, coordinates vnodes
- VNodes

  Erlang process, vnode to hashring partition, delegated work for its partition, unit of replication
- Watchers

  `gen_event`, listens for ring and service events, used to calculate fallback nodes
- Ring Manager

  stores local node copy of ring (and cluster?) data
- Ring Event Handlers

  notified about ring (and cluster?) changes and broadcasts plus metadata changes

riak_core: Coordinator

```
24  -module(riak_pipe_fitting).
25
26  -behaviour(gen_fsm).
27
28  %% API
29  -export([start_link/4]).
30  -export([eoi/1,
31           get_details/2,
32           worker_done/1,
33           workers/1]).
34  -export([validate_fitting/1,
35           format_name/1]).
```

Figure:

riak_core: Commands

```
-type riak_cmd() ::
{verb(), key(), payload()}
```

Implement handle_command/3 clause for each command in your callback VNode module

riak_core: VNodes

```
70  handle_command({get, StatName}, _Sender, #state{stats=Stats}=State) ->
71      Reply =
72          case dict:find(StatName, Stats) of
73              error ->
74                  not_found;
75              Found ->
76                  Found
77          end,
78      {reply, Reply, State};
79
80  handle_command({set, StatName, Val}, _Sender, #state{stats=Stats0}=State) ->
81      Stats = dict:store(StatName, Val, Stats0),
82      {reply, ok, State#state{stats=Stats}};
83
84  handle_command({incr, StatName}, _Sender, #state{stats=Stats0}=State) ->
85      Stats = dict:update_counter(StatName, 1, Stats0),
86      {reply, ok, State#state{stats=Stats}};
87
88  handle_command({incrby, StatName, Val}, _Sender, #state{stats=Stats0}=State) ->
89      Stats = dict:update_counter(StatName, Val, Stats0),
90      {reply, ok, State#state{stats=Stats}};
91
92  handle_command({append, StatName, Val}, _Sender, #state{stats=Stats0}=State) ->
93      Stats = try dict:append(StatName, Val, Stats0)
94              catch _:_ -> dict:store(StatName, [Val], Stats0)
```

riak_core: VNodes

```
105  handle_handoff_command(?FOLD_REQ{foldfun=Fun, acc0=Acc0}, _Sender, State) ->
106      Acc = dict:fold(Fun, Acc0, State#state.stats),
107      {reply, Acc, State}.
108
109  handoff_starting(_TargetNode, _State) ->
110      {true, _State}.
111
112  handoff_cancelled(State) ->
113      {ok, State}.
114
115  handoff_finished(_TargetNode, State) ->
116      {ok, State}.
117
118  handle_handoff_data(Data, #state{stats=Stats0}=State) ->
119      {StatName, Val} = binary_to_term(Data),
120      Stats = dict:store(StatName, Val, Stats0),
121      {reply, ok, State#state{stats=Stats}}.
122
```

Figure: Sample handoff functions from RTS example app

riak_core: Project Structure

- rebar ;)

  also written by the Basho team, makes OTP building and deploying much less painless

- dependencies

  add you_app, riak_core, riak_kv, etc. as dependencies to shell project

- new in 1.0 stuff

  cluster vs ring membership, riak_pipe, etc.

riak_core: Project Structure

- rebar ;)

  also written by the Basho team, makes OTP building and deploying much less painless

- dependencies

  add you_app, riak_core, riak_kv, etc. as dependencies to shell project

- new in 1.0 stuff

  cluster vs ring membership, riak_pipe, etc.

riak_core: Project Structure

- rebar ;)

  also written by the Basho team, makes OTP building and deploying much less painless

- dependencies

  add you_app, riak_core, riak_kv, etc. as dependencies to shell project

- new in 1.0 stuff

  cluster vs ring membership, riak_pipe, etc.

Considerations

`Other:stuff()`

- Interface layer

  riak_core apps need to implement their own interface layer, e.g. HTTP, XMPP, AMQP, MsgPack, ProtoBuffers

- Securing application

  riak_core gossip does not address identity/authZ/authN between nodes; relies on Erlang cookies

- Distribution models

  e.g. pipelined, laned vs tiered

- Query/execution models

  e.g. map-reduce (M/R), ASsociated SET (ASSET)

Considerations

`Other:stuff()`

- Interface layer

  riak_core apps need to implement their own interface layer, e.g. HTTP, XMPP, AMQP, MsgPack, ProtoBuffers

- Securing application

  riak_core gossip does not address identity/authZ/authN between nodes; relies on Erlang cookies

- Distribution models

  e.g. pipelined, laned vs tiered

- Query/execution models

  e.g. map-reduce (M/R), ASsociated SET (ASSET)

Considerations

`Other:stuff()`

- Interface layer

  riak_core apps need to implement their own interface layer, e.g. HTTP, XMPP, AMQP, MsgPack, ProtoBuffers

- Securing application

  riak_core gossip does not address identity/authZ/authN between nodes; relies on Erlang cookies

- Distribution models

  e.g. pipelined, laned vs tiered

- Query/execution models

  e.g. map-reduce (M/R), ASsociated SET (ASSET)

Considerations

`Other:stuff()`

- Interface layer

  riak_core apps need to implement their own interface layer, e.g. HTTP, XMPP, AMQP, MsgPack, ProtoBuffers

- Securing application

  riak_core gossip does not address identity/authZ/authN between nodes; relies on Erlang cookies

- Distribution models

  e.g. pipelined, laned vs tiered

- Query/execution models

  e.g. map-reduce (M/R), ASsociated SET (ASSET)

Oh, the possibilities!

`What:next()`

- Concurrency models

  e.g. actor, disruptor, evented/reactor, threading

- Consistency models

  e.g. vector-field, causal, FIFO

- Computation optimizations

  e.g. General Purpose GPU programming, native interfacing (NIFs, JInterface, Scalang?)

- Other optimizations

  e.g. Client discovery, Remote Direct Memory Access (RDMA)

Oh, the possibilities!

`What:next()`

- Concurrency models

  e.g. actor, disruptor, evented/reactor, threading

- Consistency models

  e.g. vector-field, causal, FIFO

- Computation optimizations

  e.g. General Purpose GPU programming, native interfacing (NIFs, JInterface, Scalang?)

- Other optimizations

  e.g. Client discovery, Remote Direct Memory Access (RDMA)

Oh, the possibilities!

`What:next()`

- Concurrency models

  e.g. actor, disruptor, evented/reactor, threading

- Consistency models

  e.g. vector-field, causal, FIFO

- Computation optimizations

  e.g. General Purpose GPU programming, native interfacing (NIFs, JInterface, Scalang?)

- Other optimizations

  e.g. Client discovery, Remote Direct Memory Access (RDMA)

Oh, the possibilities!

`What:next()`

- Concurrency models

  e.g. actor, disruptor, evented/reactor, threading

- Consistency models

  e.g. vector-field, causal, FIFO

- Computation optimizations

  e.g. General Purpose GPU programming, native interfacing (NIFs, JInterface, Scalang?)

- Other optimizations

  e.g. Client discovery, Remote Direct Memory Access (RDMA)

```
# finger $(whoami)


Login: susan                    Name: Susan Potter
Directory: /home/susan          Shell: /bin/zsh
On since Mon 29 Sep 1997 21:18 (GMT) on tty1 from :0
Too much unread mail on me@susanpotter.net
Now working at Assistly! Looking for smart developers!;)
Plan:
  github:  mbbx6spp
  twitter: @SusanPotter
```

Slides & Material

https://github.com/mbbx6spp/riak_core-templates

Figure: http://susanpotter.net/talks/strange-loop/2011/dynamo-not-just-for-datastores/

Examples

- Rebar Templates

  https://github.com/mbbx6spp/riak_core-templates

- Riak Pipes

  https://github.com/basho/riak_pipe

- Riak Zab

  https://github.com/jtuple/riak_zab

Questions?

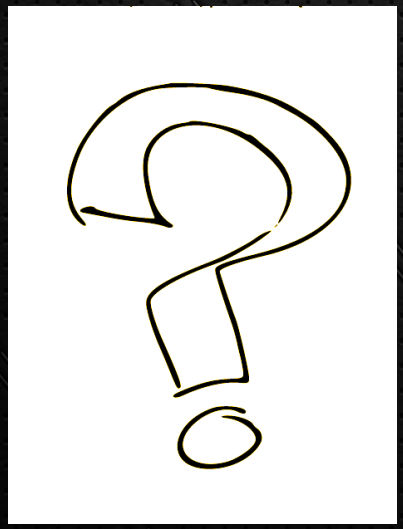


Figure: http://www.flickr.com/photos/42682395@N04/

@SusanPotter

Questions?

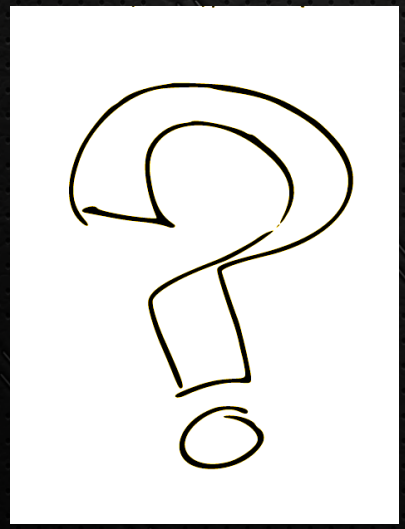


Figure: http://www.flickr.com/photos/42682395@N04/

@SusanPotter