

# Dynamo, Five Years Later

Andy Gross

Chief Architect, Basho Technologies

QCon London 2013



# Dynamo

- ✦ Published October 2007 @ SOSP
- ✦ Describes a collection of distributed systems techniques applied to low-latency key-value storage
- ✦ Spawned (along with BigTable) many imitators, an industry (LinkedIn -> Voldemort, Facebook -> Cassandra)
- ✦ Authors nearly got fired from Amazon for publishing



# Riak - A Dynamo Clone

- ✦ First lines of first prototype written in Fall 2007 on a plane on the way to my Basho interview
- ✦ “Technical Debt” is another term we use at Basho for this code
- ✦ Mostly Erlang with some C/C++
- ✦ Apache2 Licensed
- ✦ First release in 2009, 1.3 released 2/21/13



# Basho





# Basho

- ✦ Founded late 2007 by ex-Akamai people
- ✦ Currently ~120 employees, distributed, with offices in Cambridge, San Francisco, London, and Tokyo
- ✦ We sponsor of Riak Open Source
- ✦ We sell Riak Enterprise (Riak + Multi-DC replication)
- ✦ We sell Riak CS (S3 clone backed by Riak Enterprise)



# Principles

- ✦ Always-writable
- ✦ Incrementally scalable
- ✦ Symmetrical
- ✦ Decentralized
- ✦ Heterogenous
- ✦ Focus on SLAs, tail latency



# Techniques

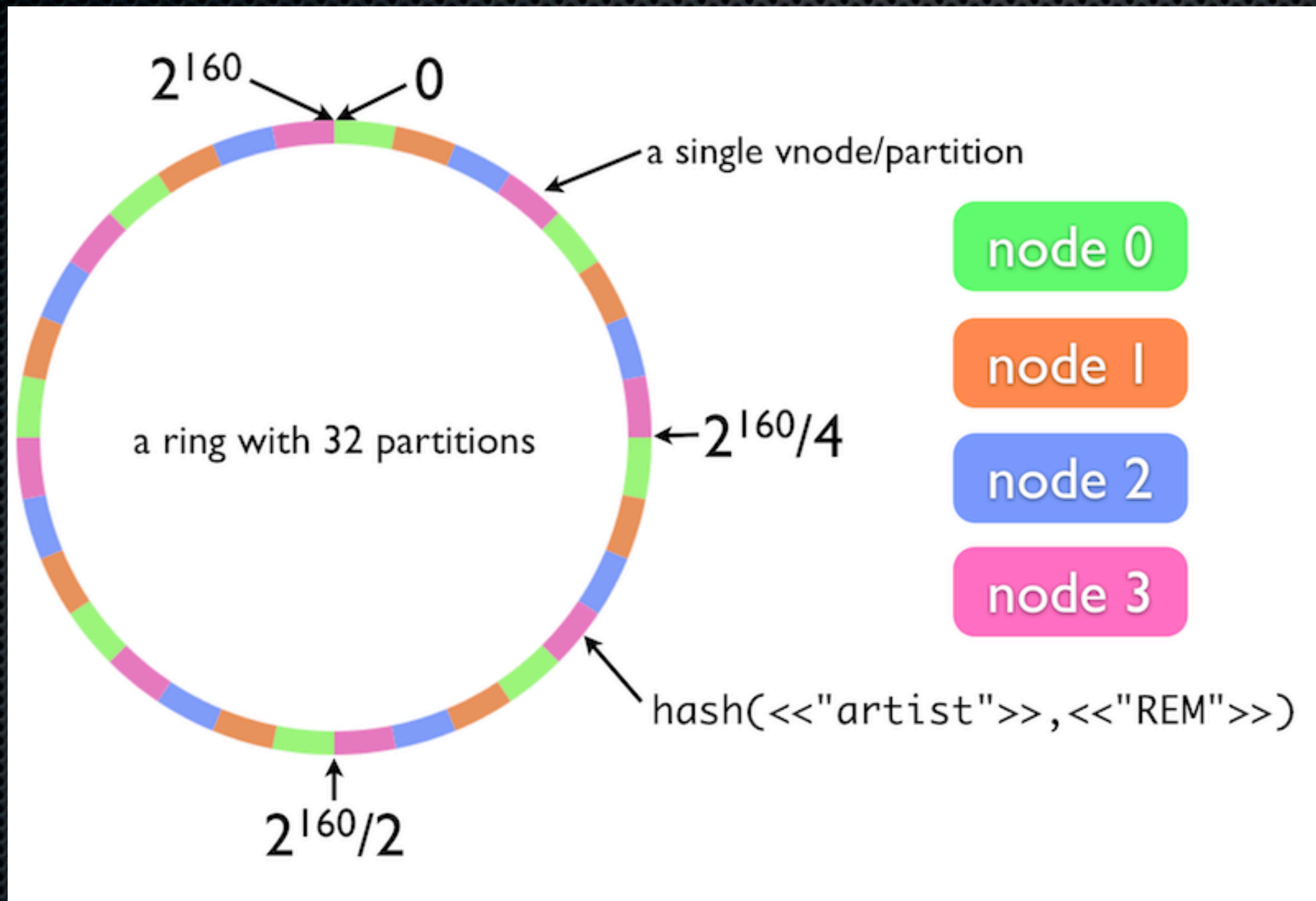
- ✦ Consistent Hashing
- ✦ Vector Clocks
- ✦ Read Repair
- ✦ Anti-Entropy
- ✦ Hinted Handoff
- ✦ Gossip Protocol



# Consistent Hashing

- ✦ Invented by Danny Lewin and others @ MIT/Akamai
- ✦ Minimizes remapping of keys when number of hash slots changes
- ✦ Originally applied to CDNs, used in Dynamo for replica placement
- ✦ Enables incremental scalability, even spread
- ✦ Minimizes hot spots







# Vector Clocks

- ✦ Introduced by Mattern et al, in 1988
- ✦ Extends Lamport's timestamps (1978)
- ✦ Each value in Dynamo tagged with vector clock
- ✦ Allows detection of stale values, logical siblings



# Read Repair

- ✦ Update stale versions opportunistically on reads (instead of writes)
- ✦ Pushes system toward consistency, after returning value to client
- ✦ Reflects focus on a cheap, always-available write path



# Hinted Handoff

- ✦ Any node can accept writes for other nodes if they're down
- ✦ All messages include a destination
- ✦ Data accepted by node other than destination is handed off when node recovers
- ✦ As long as a single node is alive the cluster can accept a write



# Anti-Entropy

- ✦ Replicas maintain a Merkle Tree of keys and their versions/hashes
- ✦ Trees periodically exchanged with peer vnodes
- ✦ Merkle tree enables cheap comparison
- ✦ Only values with different hashes are exchanged
- ✦ Pushes system toward consistency

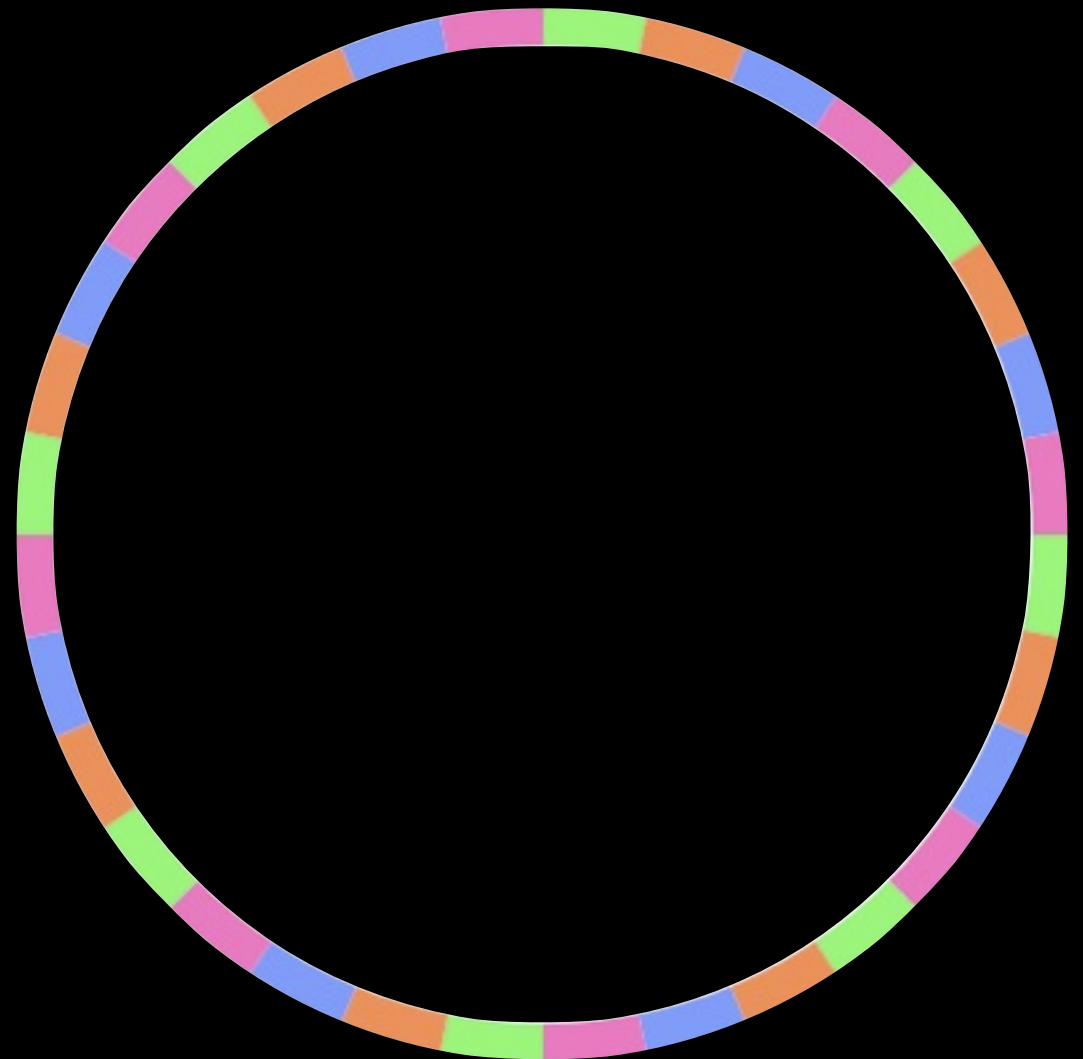


# Gossip Protocol

- ✦ Decentralized approach to managing global state
- ✦ Trades off atomicity of state changes for a decentralized approach
- ✦ Volume of gossip can overwhelm networks without care



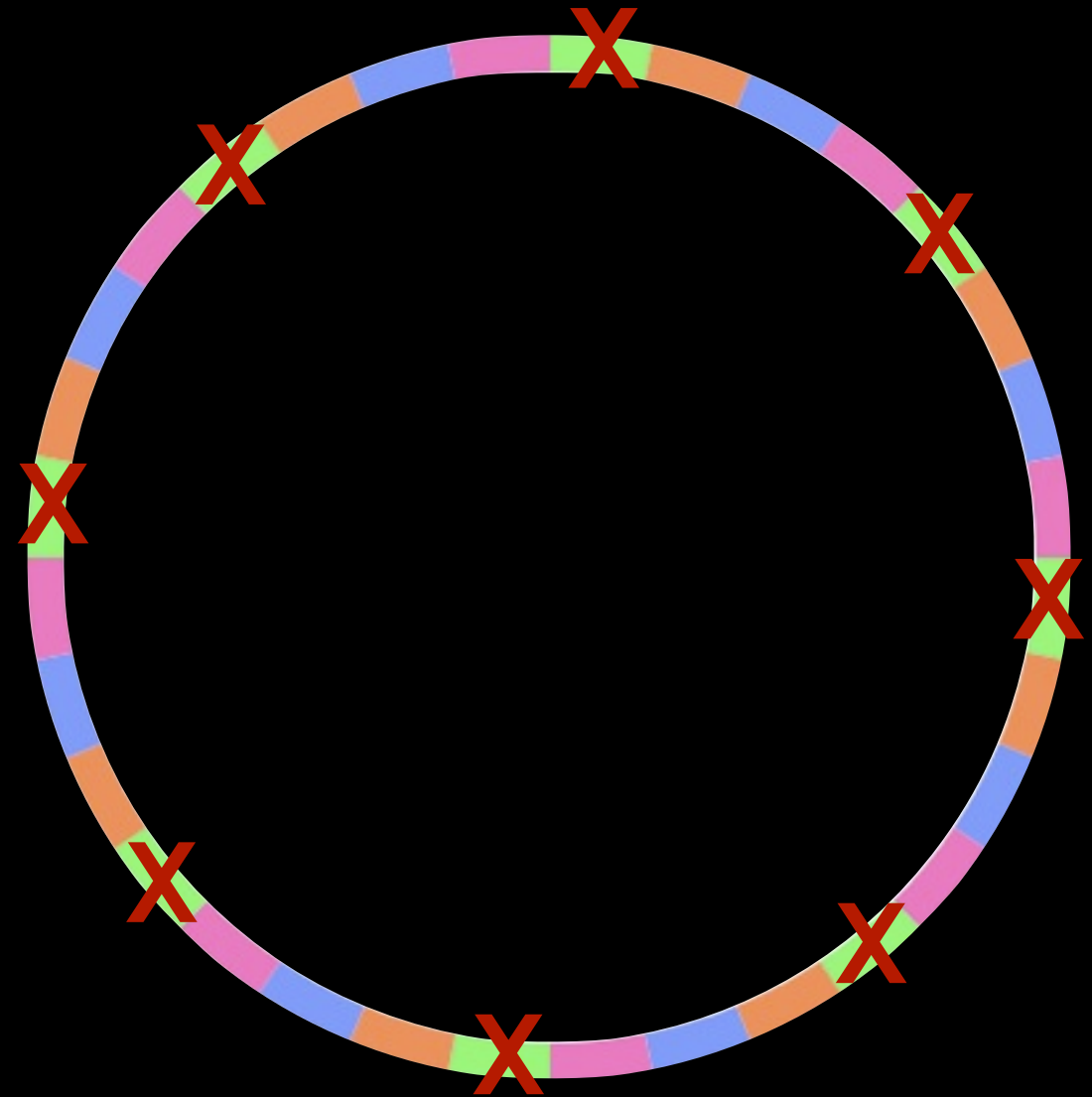
# Hinted Handoff





# Hinted Handoff

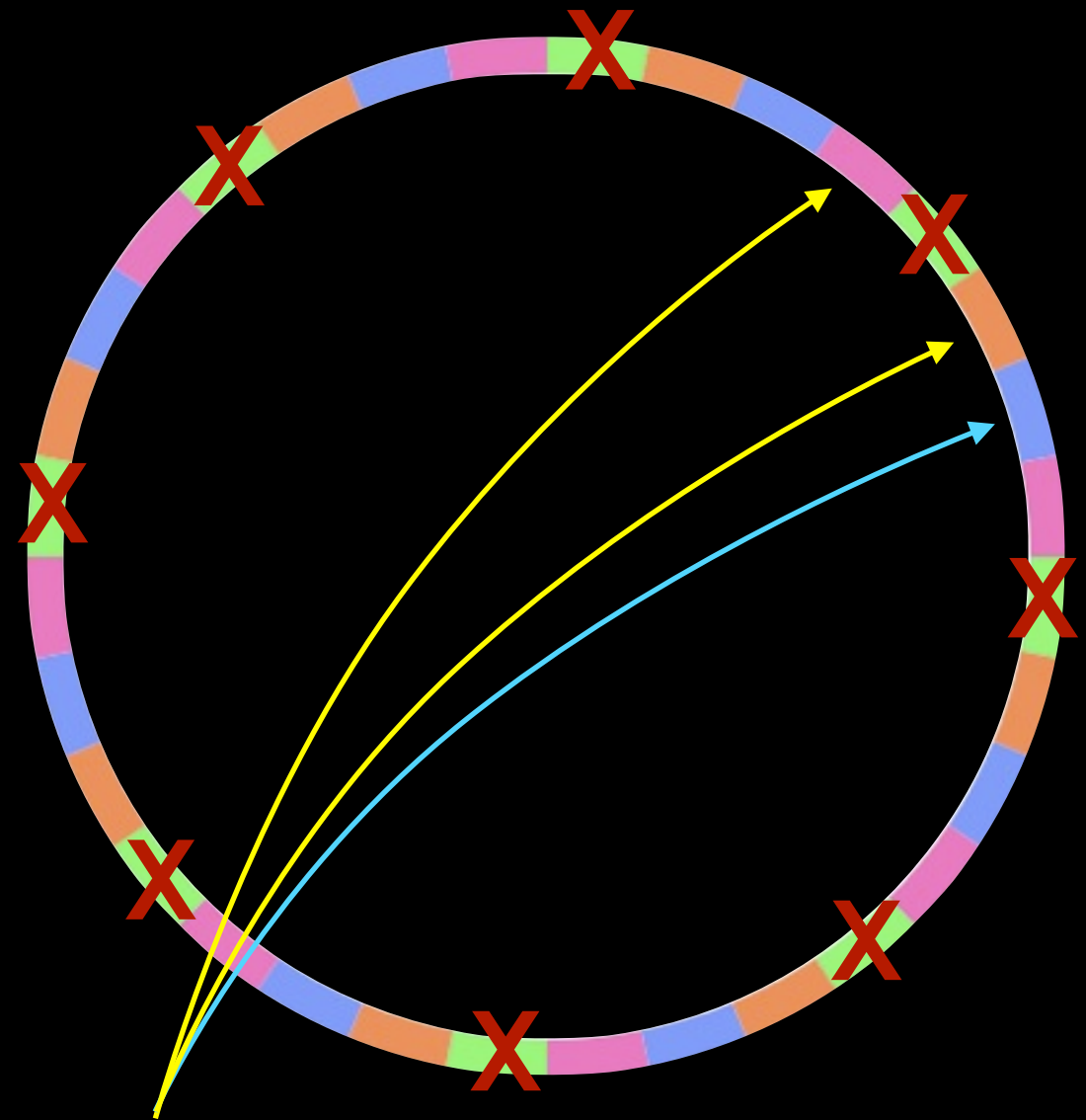
- Node fails





# Hinted Handoff

- Node fails
- Requests go to fallback

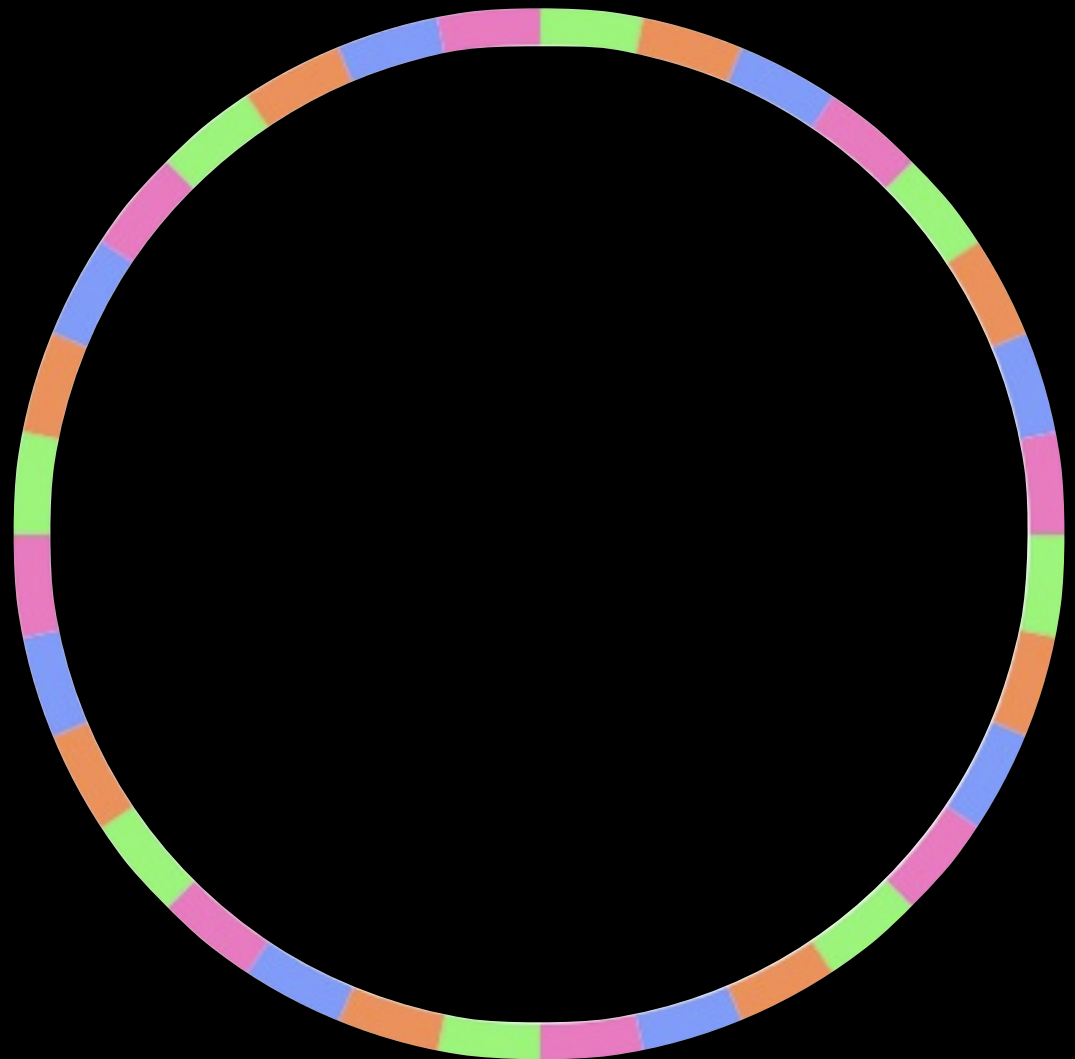


`hash("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")`



# Hinted Handoff

- Node fails
- Requests go to fallback
- Node comes back

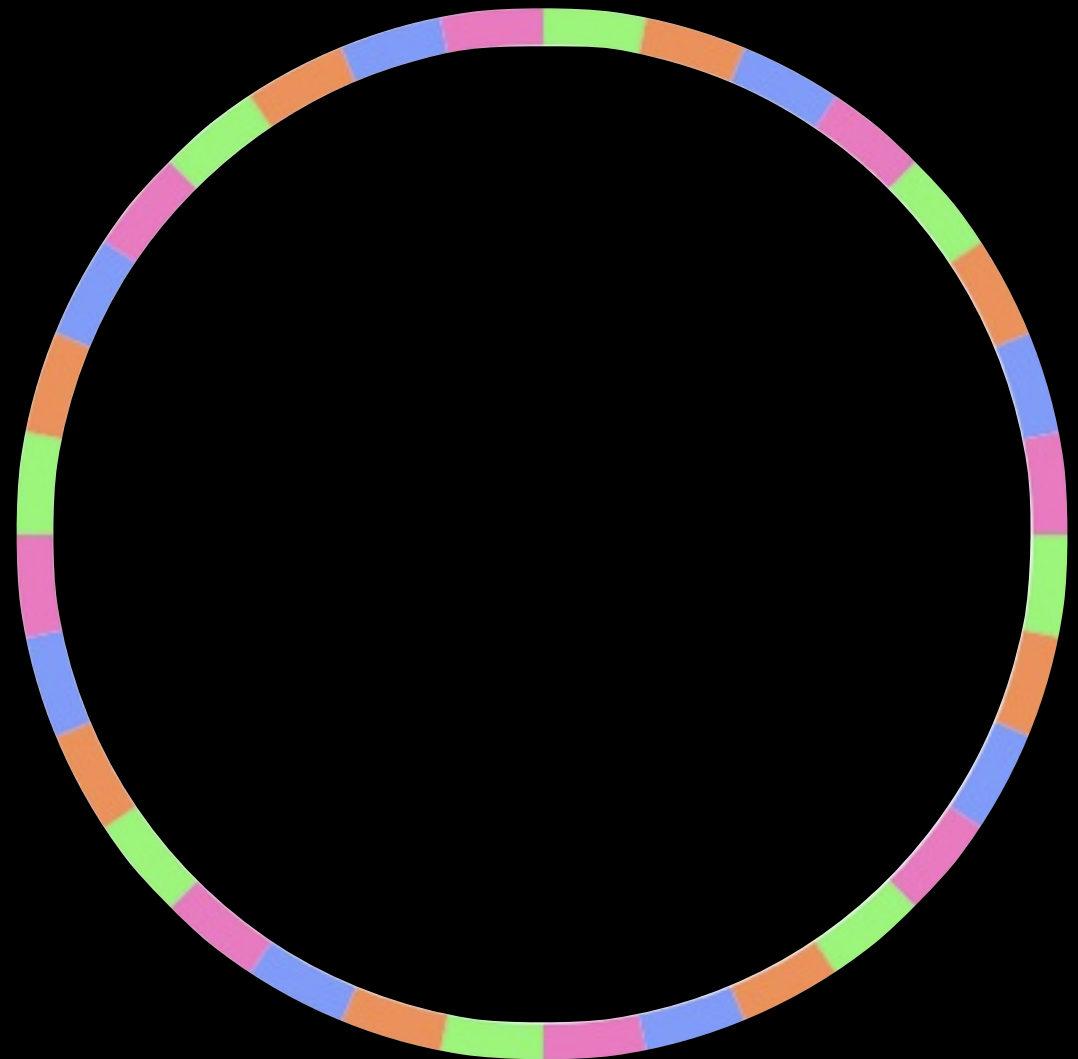


`hash("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")`



# Hinted Handoff

- Node fails
- Requests go to fallback
- Node comes back
- “Handoff” - data returns to recovered node

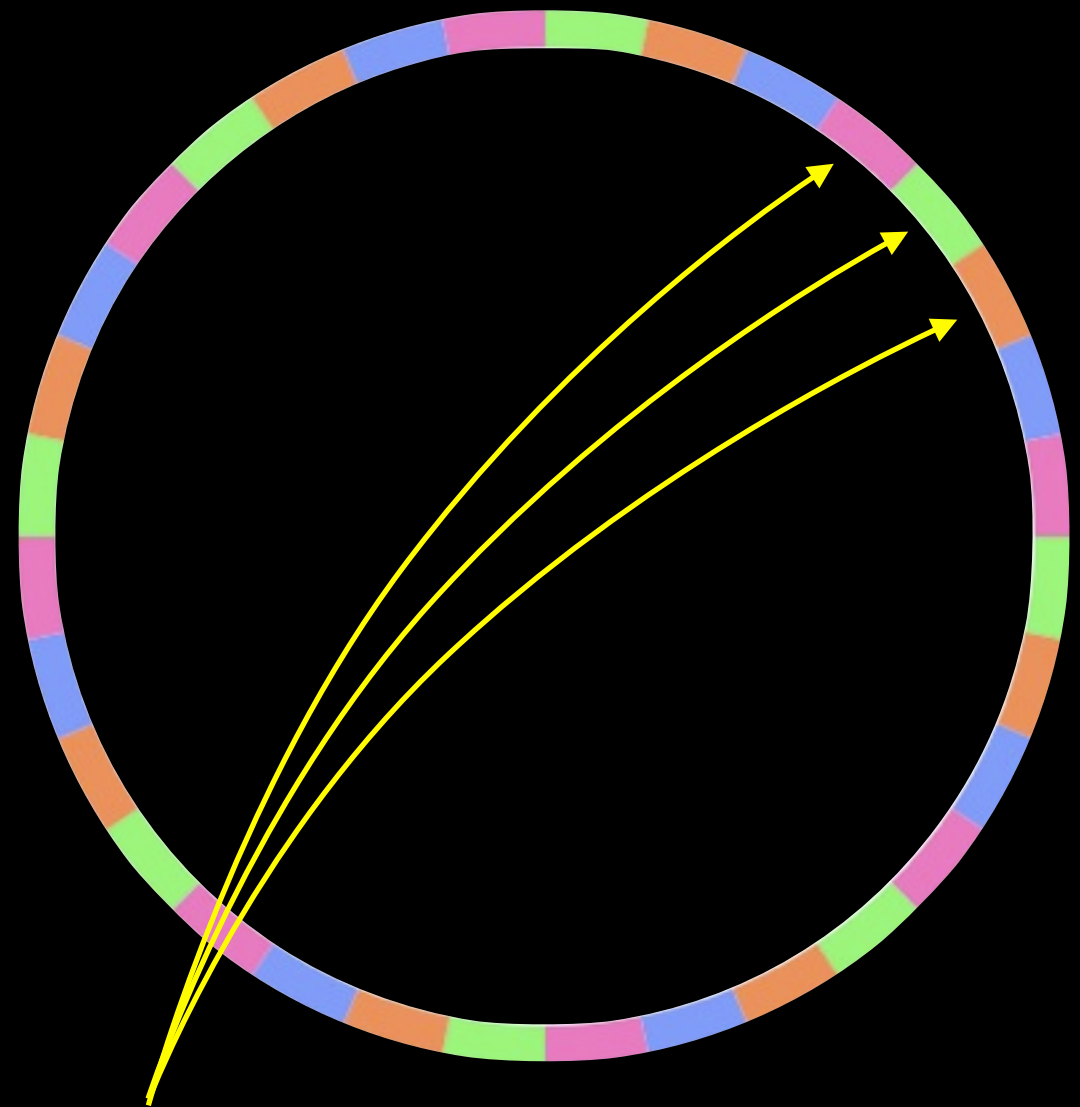


hash(“blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA”)



# Hinted Handoff

- Node fails
- Requests go to fallback
- Node comes back
- “Handoff” - data returns to recovered node
- Normal operations resume



hash(“blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA”)



# Anatomy of a Request

```
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")
```



# Anatomy of a Request

client  
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")  
↓  
Riak


---



# Anatomy of a Request

client  
Riak

`get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")`



Get Handler (FSM)



# Anatomy of a Request

client  
Riak

```
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")
```

---

Get Handler (FSM)

hash("blocks/  
6307C89A-710A-42CD-9FFB-2A6B39F983EA")  
== 10, 11, 12



# Anatomy of a Request

client  
Riak

```
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")
```

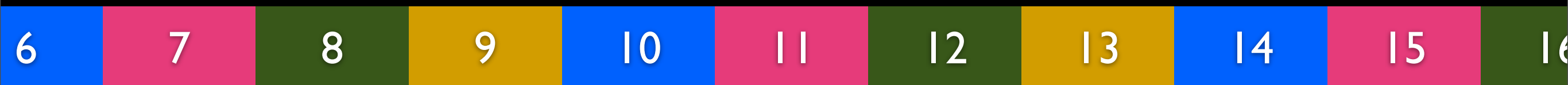
---

Get Handler (FSM)

hash("blocks/  
6307C89A-710A-42CD-9FFB-2A6B39F983EA")  
== 10, 11, 12

Coordinating node  
Cluster

---



The Ring



# Anatomy of a Request

client  
Riak

```
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")
```

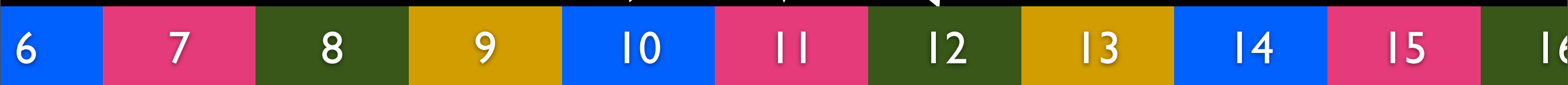
---

Get Handler (FSM)

Coordinating node  
Cluster

```
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")
```

---



The Ring



# Anatomy of a Request

get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")

client

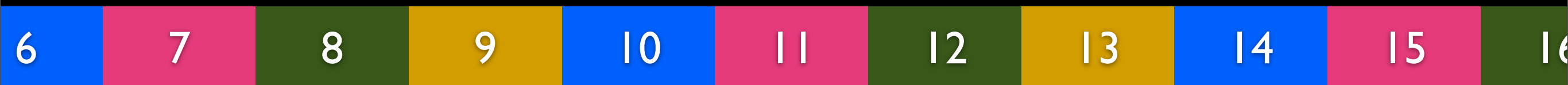
Riak

R=2

Get Handler (FSM)

Coordinating node

Cluster



The Ring

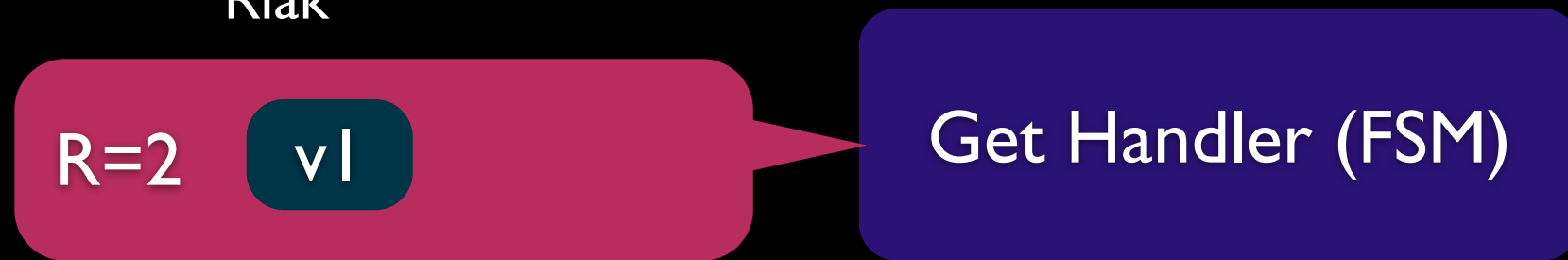


# Anatomy of a Request

get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")

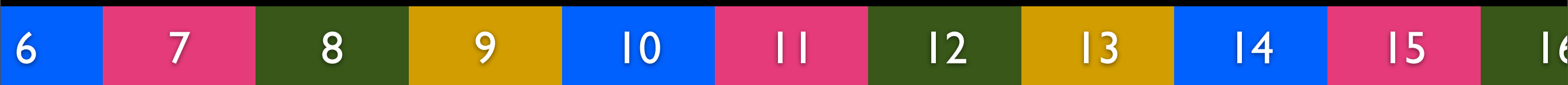
client

Riak



Coordinating node

Cluster

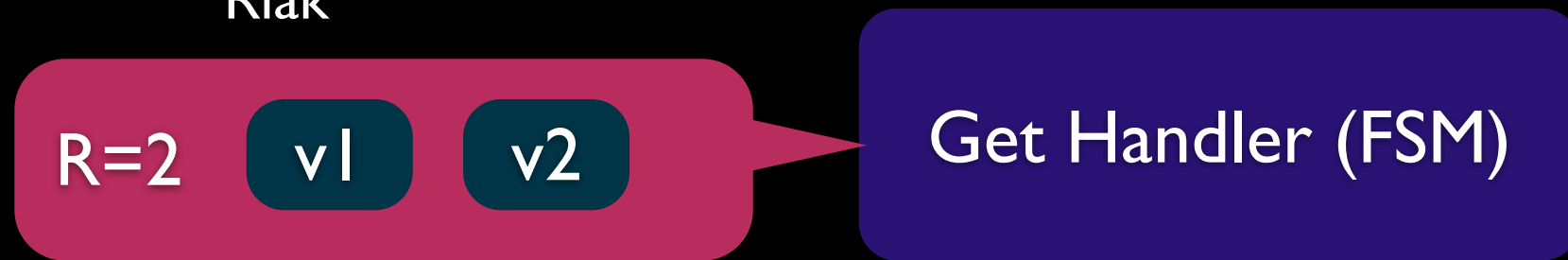


The Ring



# Anatomy of a Request

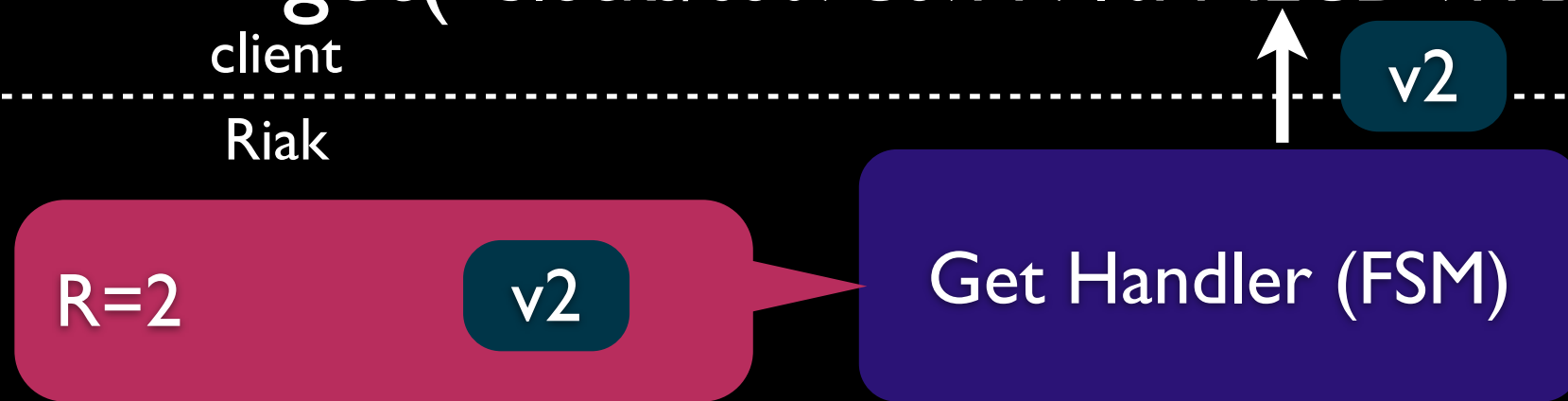
client  
get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")  
-----  
Riak





# Anatomy of a Request

get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")  
client  
Riak





# Anatomy of a Request

`get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")`

v2



# Read Repair

get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")

client

Riak

v2

R=2

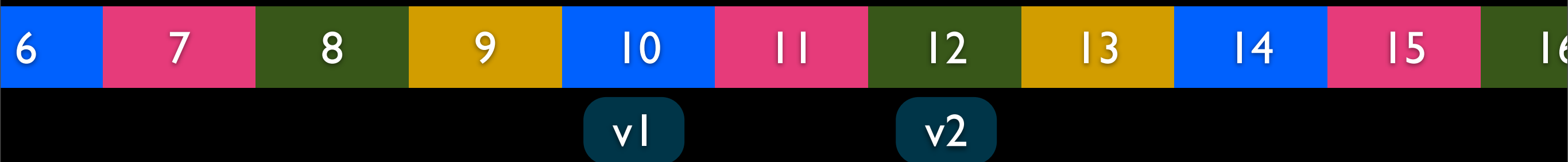
v1

v2

Get Handler (FSM)

Coordinating node

Cluster





# Read Repair

get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")

client

Riak

v2

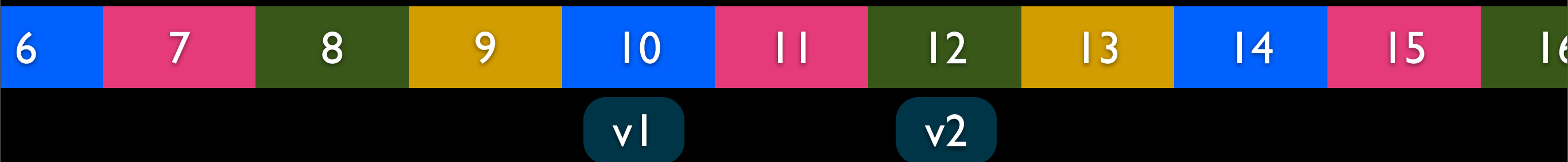
R=2

v2

Get Handler (FSM)

Coordinating node

Cluster





# Read Repair

get("blocks/6307C89A-710A-42CD-9FFB-2A6B39F983EA")

client

Riak

v2

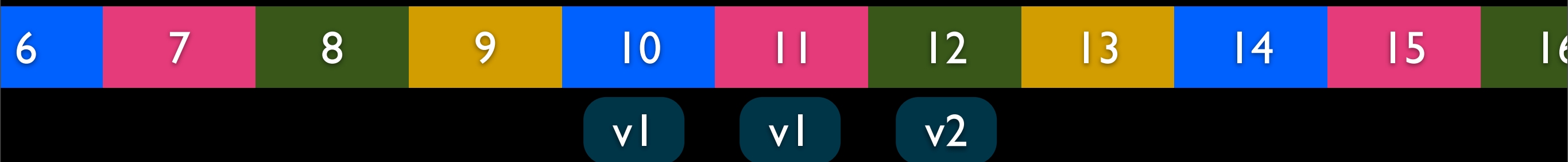
R=2

v2

Get Handler (FSM)

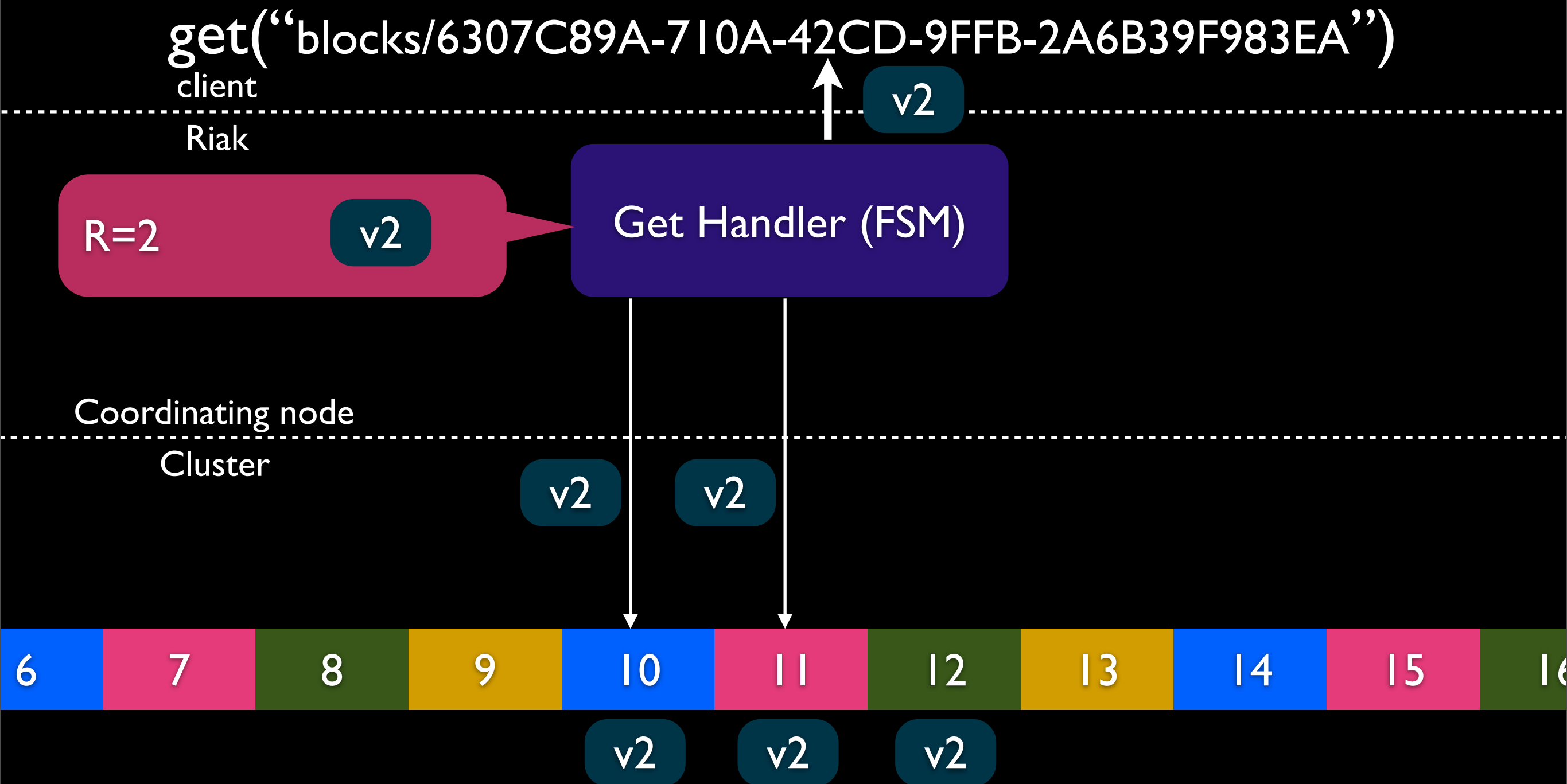
Coordinating node

Cluster





# Read Repair





# Riak Architecture

Erlang/OTP Runtime



# Riak Architecture

Erlang/OTP Runtime



Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

Riak Core

Riak KV

# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing

Riak Core

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing  
membership

Riak Core

Riak KV

# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing  
membership

handoff

Riak Core

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff  
membership   node-liveness

Riak Core

Riak KV

# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff   gossip  
membership   node-liveness

Riak Core

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff   gossip  
membership   node-liveness   buckets

Riak Core

Riak KV

# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff   gossip  
membership   node-liveness   buckets

Riak Core

vnode master

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff   gossip  
membership   node-liveness   buckets

Riak Core

vnode master

vnodes

Riak KV

# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff   gossip  
membership   node-liveness   buckets

Riak Core

vnode master

vnodes

storage backend

Riak KV



# Riak Architecture

Erlang/OTP Runtime

Client APIs

HTTP

Protocol Buffers

Erlang local client

Request Coordination

get

put

delete

map-reduce

consistent hashing   handoff   gossip  
membership   node-liveness   buckets

Riak Core

vnode master

vnodes

storage backend

JS Runtime

Riak KV



# Problems with Dynamo

- ✧ Eventual Consistency is harsh mistress
  - ✧ Pushes conflict resolution to clients
- ✧ Key/value data types limited in use
- ✧ Random replica placement destroys locality
- ✧ Gossip protocol can limit cluster size
- ✧  $R+W > N$  is **NOT** more consistent
- ✧ TCP Incast



# Key-Value Conflict Resolution

- ✦ Forcing clients to resolve consistency issues on read is a pain for developers
- ✦ Most end up choosing the server-enforced last-write-wins policy
- ✦ With many language clients, logic must be implemented many times
- ✦ One solution: <https://github.com/bumptech/montage>
- ✦ Another: Make everything immutable
- ✦ Another: CRDTs



# Optimize for Immutability

- ✦ “Accountants don’t use erasers” - Pat Helland
- ✦ Eventual consistency is \*great\* for immutable data
- ✦ Conflicts become a non-issue if data never changes
  - ✦ don’t need full quorums, vector clocks
  - ✦ backend optimizations are possible
- ✦ Problem space shifts to distributed GC
  - ✦ ... which is very hard, but not the user’s problem anymore



# CRDTs

- ✦ Conflict-free|Commutative Replicated Data Types
- ✦ A server side structure and conflict-resolution policy for richer datatypes like counters and sets amenable to eventual consistency
- ✦ Letia et al. (2009). *CRDTs: Consistency without concurrency control*: <http://hal.inria.fr/inria-00397981/en>
- ✦ Prototype here: [http://github.com/basho/riak\\_dt](http://github.com/basho/riak_dt)



# Random Placement and Locality

- ✦ By default, keys are randomly placed on different replicas
- ✦ But we have buckets!
- ✦ Containers imply cheap iteration/enumeration, but with random placement it becomes an expensive full-scan
- ✦ Partial Solution: hash function defined per-bucket can increase locality
- ✦ Lots of work done to minimize impact of bucket listings



# $(R+W > N) \neq \text{Consistency}$

- ✦  $R+W$  described in Dynamo paper as “consistency knobs”
  - ✦ Some Basho/Riak docs still say this too! :(
- ✦ Even if  $R=W=N$ , sloppy quorums and partial writes make reading old values possible
- ✦ “Read your own writes if your writes succeed but otherwise you have no idea what you’re going to read consistency (RYOWIWSBOYHNIWYGTRC)” - Joe Blomstedt
- ✦ Solution: actual “strong” consistency



# Strong Consistency in Riak

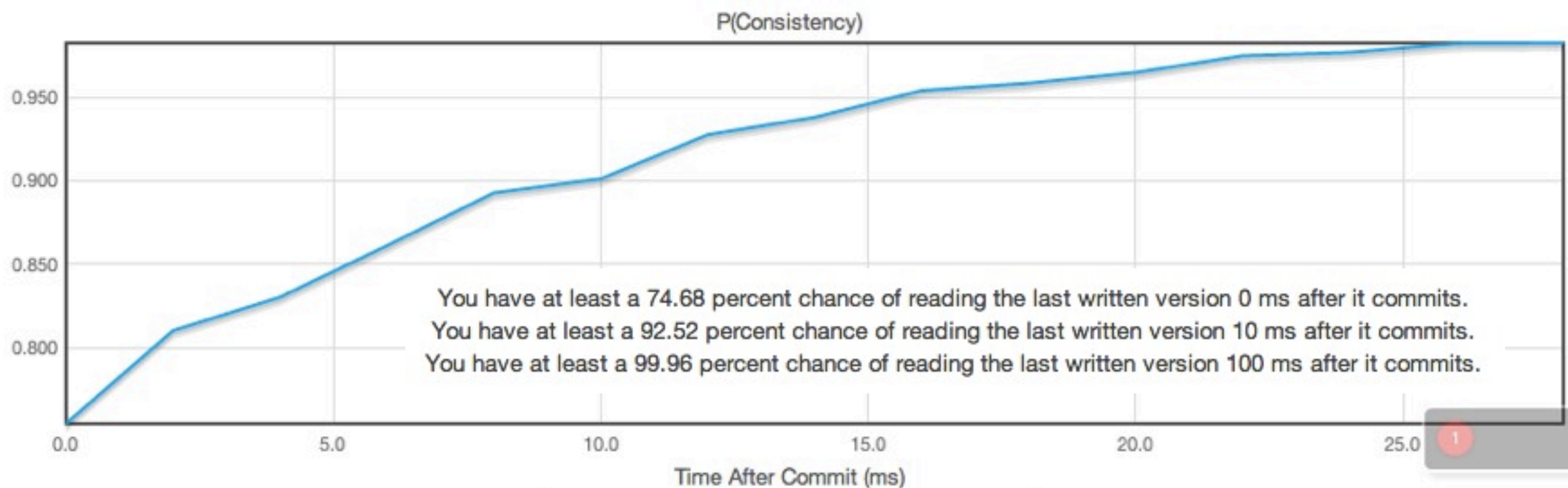
- ✦ CAP says you must choose C vs. A, but only during failures
- ✦ There's no reason we can't implement both models, with different tradeoffs
- ✦ Enable strong consistency on a per-bucket basis
- ✦ See Joe Blomstedt's talk at RICON 2012: <http://ricon2012.com>, earlier work at: [http://github.com/jtuple/riak\\_zab](http://github.com/jtuple/riak_zab)



# An Aside: Probabalistically Bounded Staleness

$R=W=1$ , .1ms latency at all hops

## How Eventual is Eventual Consistency? PBS in action under Dynamo-style quorums



Bailis et al. : <http://pbs.cs.berkeley.edu>



# TCP Incast

- ✦ “You can’t pour two buckets of manure into one bucket” - Scott Fritchie’s Grandfather
- ✦ “microbursts” of traffic sent to one cluster member
  - ✦ Coordinator sends request to three replicas
  - ✦ All respond with large-ish result at roughly the same time
  - ✦ Switch has to either buffer or drop packets
- ✦ Cassandra tries to mitigate: 1 replica sends data, others send hashes. We should do this in Riak.



# What Riak Did Differently (or wrong)

- ✦ Screwed up vector clock implementation
  - ✦ Actor IDs in vector clocks were *client* ids, therefore potentially unbounded
  - ✦ Size explosion resulted in huge objects, caused OOM crashes
  - ✦ Vector clock pruning resulted in false siblings
  - ✦ Fixed by forwarding to node in preflight circa 1.0



# What Riak Did Differently

- ✦ No active anti-entropy until v1.3
  - ✦ Early versions had slow, unstable AAE
  - ✦ Node loss required reading all objects and repopulating replicas via read repair
    - ✦ Ok for objects that are read often
    - ✦ Rarely-read objects N value decreases over time



# What Riak Did Differently

- ✦ Initial versions had an unavailability window during topology changes
  - ✦ Nodes would claim partitions immediately, before data had been handed off
  - ✦ New versions don't change request preflight until all data has been handed off
  - ✦ Implemented as 2PC-ish commit over gossip



# Riak, Beyond Dynamo

- ✦ MapReduce
- ✦ Search
- ✦ Secondary Indexes
- ✦ Pre/post-commit hooks
- ✦ Multi-DC replication
- ✦ Riak Pipe distributed computation
- ✦ Riak CS



# Riak CS

- ✦ Amazon S3 clone implemented as a proxy in front of Riak
- ✦ Handles eventual consistency issues, object chunking, multitenancy, and API for a much narrower use case
- ✦ Forced us to eat our own dogfood and get serious about fixing long-standing warts
- ✦ Drives feature development



# Riak the Product vs. Dynamo the Service

- Dynamo had luxury of being a service while Riak is a product
  - Screwing things up with Riak can not be fixed with an emergency deploy
  - Multiple platforms, packaging are challenges
  - Testing distributed systems is another talk entirely (QuickCheck FTW)
    - <http://www.erlang-factory.com/upload/presentations/514/TestFirstConstructionDistributedSystems.pdf>



# Riak Core

- ✧ Some of our best work!
- ✧ Dynamo abstracted
- ✧ Implements all Dynamo techniques without prescribing a use case
- ✧ Examples of Riak Core apps:
  - ✧ Riak KV!
  - ✧ Riak Search
  - ✧ Riak Pipe



# Riak Core

- ✦ Production deployments
  - ✦ OpenX: several 100+-node clusters of custom Riak Core systems
  - ✦ StackMob: proxy for mobile services implemented with Riak Core
- ✦ Needs to be *much* easier to use and better documented



# Multi-Datacenter Replication

- ✦ Intra-cluster replication in Riak is optimized for consistently low latency, high throughput
- ✦ WAN replication needs to deal with lossy links, long fat networks, TCP oddities
- ✦ MDC replication has 2 phases: full-sync (per-partition merkle comparisons), real-time (asynchronous, driven by post-commit hook)
- ✦ Separate policies settable per-bucket



# Erlang

- ✦ Still the best language for this stuff, but
  - ✦ We mix data and control messages over Erlang message passing. Switch to TCP (or uTP/UDT) for data
  - ✦ NIFs are problematic
  - ✦ VM tuning can be a dark art
- ✦ ~90 public repos of mostly-Erlang, mostly-awesome open source: <https://github.com/basho>



# Other Future Directions

- ✦ Security was not a factor in Dynamo's or Riak's design
  - ✦ Isolating Riak increases operational complexity, cost
- ✦ Statically sized ring is a pain
- ✦ Explore possibilities with smarter clients
- ✦ Support larger clusters
- ✦ Multitenancy, tenant isolation
- ✦ More vertical products like Riak CS



# Questions?

@argv0

<http://www.basho.com>

<http://github.com/basho>

<http://docs.basho.com>