Databases for the New World

Parallel Functional Programming

John Hughes

New Demands on Databases

- Very high throughput
  - Requires large clusters to process all the operations
- Low and predictable latency
  - Good customer experience for (almost) *all* customers
  - Not average latency, but 99,9th percentile
- Always available
  - Think Amazon, Twitter, Facebook
  - Failed operations==lost business

”The Network is Reliable” (aphyr.com)

- ”During a planned network reconfiguration to improve reliability, Fog Creek suddenly lost access to their network. A network loop had formed…it resulted in two hours of total service unavailability.”

- ”Mystery RabbitMQ partitions…upping the partition detection timeout to 2 minutes reduced the frequency of partitions, but didn’t prevent them altogether.”

”The Network is Reliable” (aphyr.com)

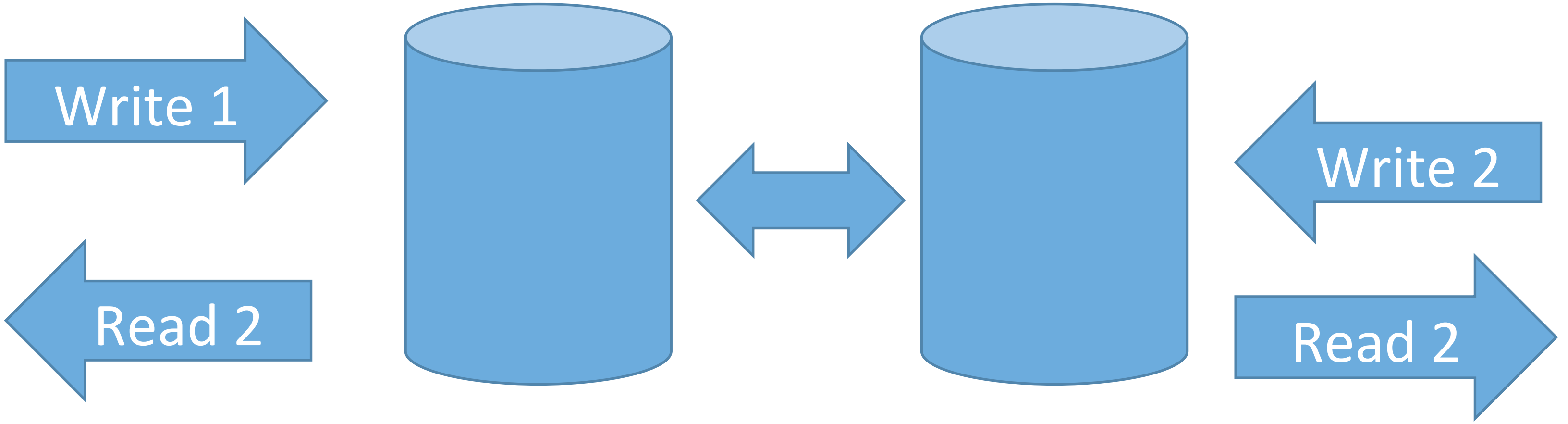
- ”DRBD split-brain…both nodes can remain online and accept writes…the only realistic option is to discard all writes not made to a selected component.”

- ”Github…a 90 second network partition caused file servers to send ”Shoot the other node in the head” messages to each other…when the network recovered, both nodes shot each other at the same time…recovering took five hours.”

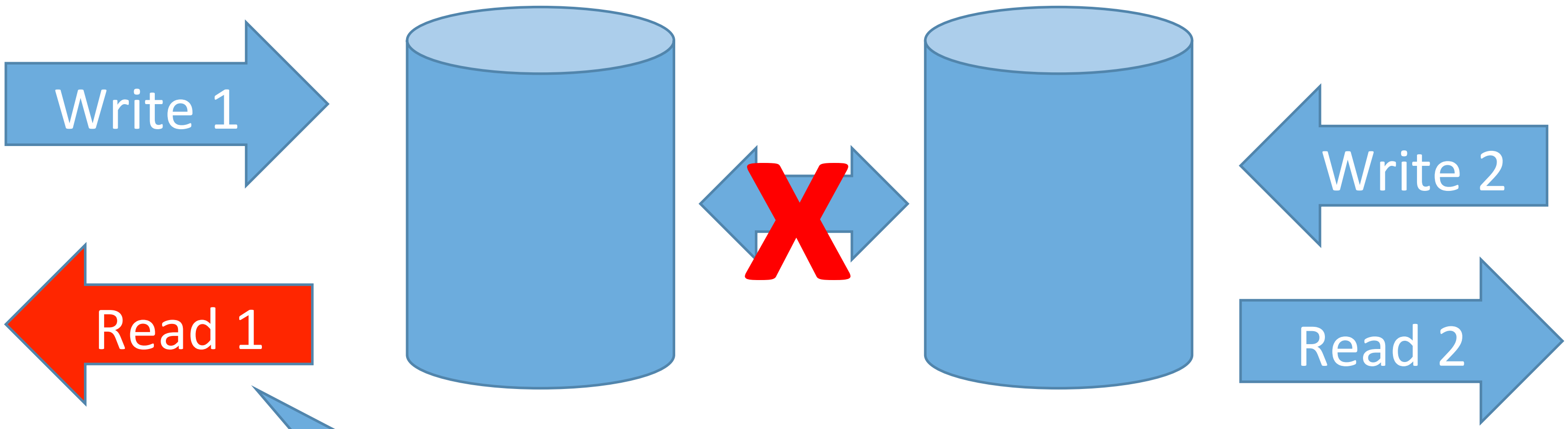
”The Network is Reliable” (aphyr.com)

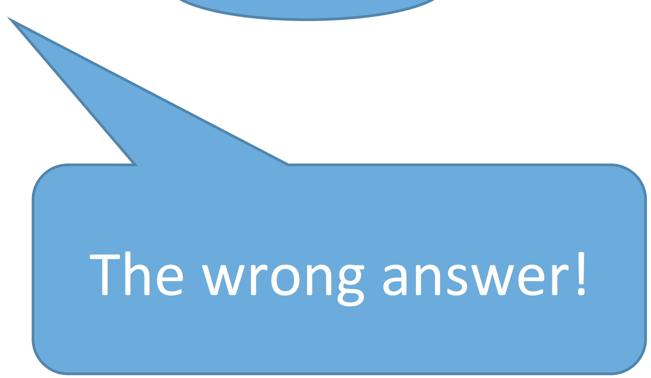
- ”MongoDB…partition caused *two hours* of write loss…network events causing failover on EC2 are common…simultaneous primaries accepting writes for *multiple days* are not unknown.”

Network Partitions in a Database Cluster



Network Partitions in a Database Cluster





The only alternative is to block

➔ *not available*

C onsistency

A vailability

P artition-tolerance

theorem

Pick two!

Conjecture: Eric Brewer, 2000
Proof: Gilbert and Lynch, 2002

C onsistency

A vailability

P artition-tolerance

theorem

Pick two!

Conjecture: Eric Brewer, 2000
Proof: Gilbert and Lynch, 2002

Luckily…

- For many applications, consistency is not *essential*
  - E.g. Facebook posts
- ”Eventual” consistency is good enough
  - Eventually we get the right answer
  - Mechanisms to discover and tolerate inconsistencies
- Often, simple queries are all that is needed
  - Primary key only, no relational joins

Amazon Dynamo (2007)

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

More than 2,000 citations

Many, many systems follow this design

ABSTRACT

Reliability at massive scale is … face at Amazon.com, one of the … the world; even the slightest … consequences and impacts c… platform, which provides servi… is implemented on top of an inf… servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the r… software systems.

This paper presents the design … highly available key-value stor… core services use to provide … achieve this level of availabili… under certain failure scenarios. … versioning and application-assis… that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

…n has learned from operating …liability and scalability of a …application state is managed. …zed, loosely coupled, service … hundreds of services. In this …need for storage technologies …ple, customers should be able … shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for … that it can always write to and … its data needs to be available …

…cture comprised of millions of …f operation; there are always a …rver and network components … As such Amazon's software … a manner that treats failure …thout impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the

What has this to do with Erlang?

- Erlang excels at *scalable services* (e.g. WhatsApp), which often need a scalable database

- Erlang is good at *implementing* a scalable distributed database
    - CouchDB, Couchbase, Riak, Scalaris, Dynomite…

- Basho's Riak is one of the big noSQL databases
    - (Rovio, Danish medical card, UK National Health Service)

API

- Dynamo is a *Distributed Key-Value Store*

```
get :: Key -> Value
get :: Key -> {Value, Context}
put :: (Key, Value) -> ok
put :: (Key, Context, Value) -> ok
```

get :: Key -> RiakObject

put :: RiakObject -> ok

Riak splits a key into a *key* and a *bucket* (like a table name)

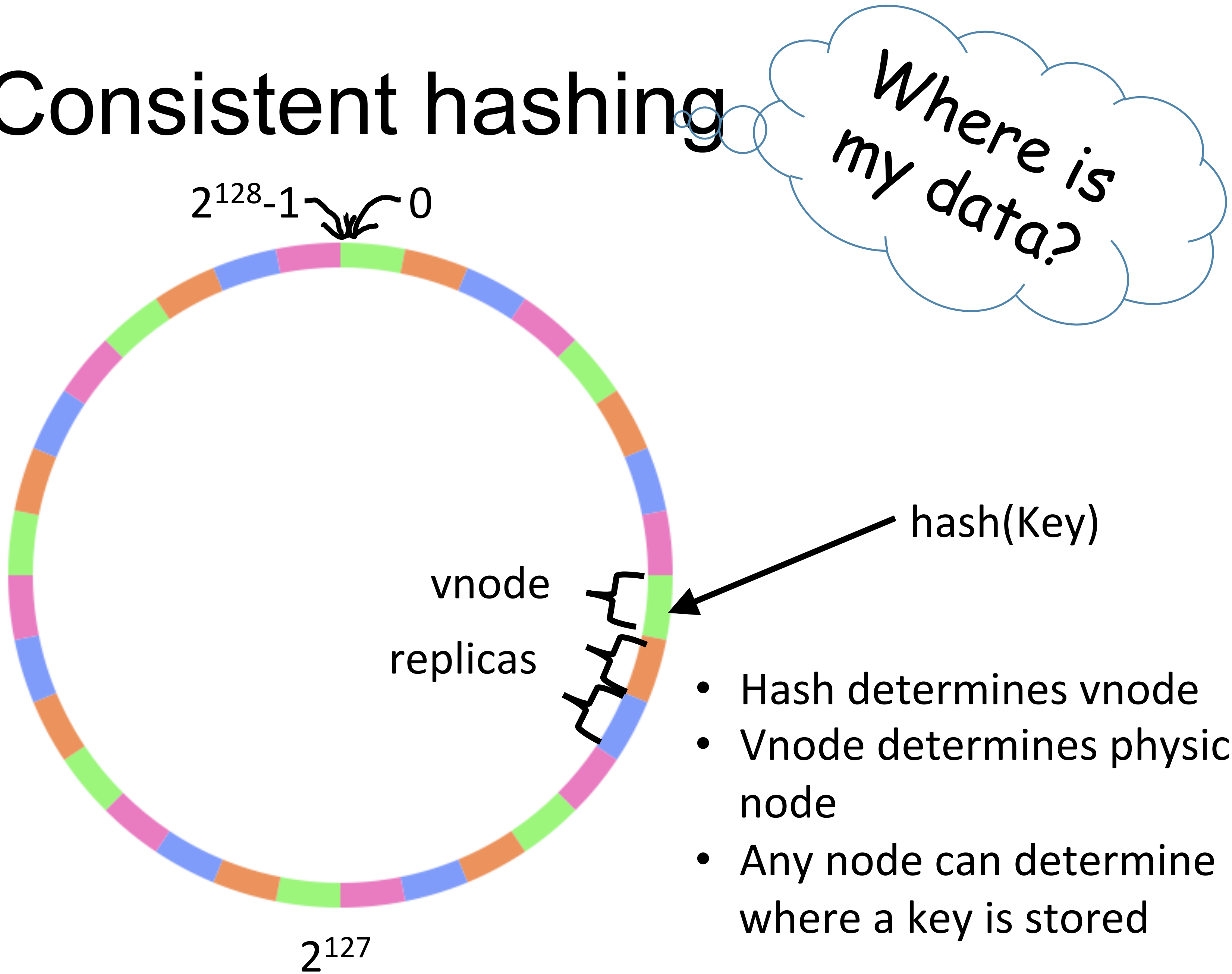
RiakObject

- Key
- Context
- Value

Cluster

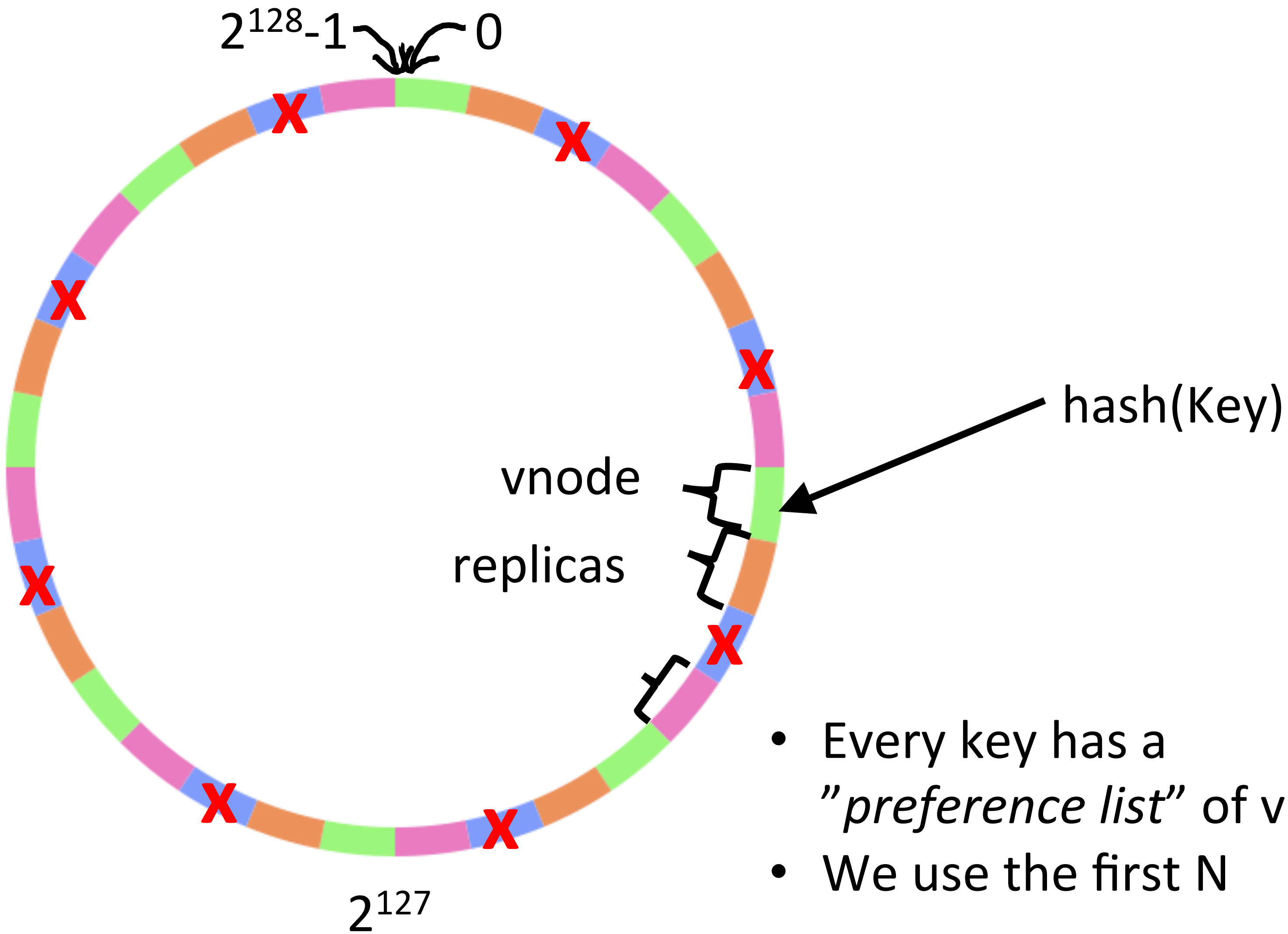
- Dynamo is designed for clusters of up to a few hundred machines
  - Largest Riak clusters are 60-80 machines
- Each machine handles a share of the load
  - Stores a part of the data (in a local back-end, such as Google LevelDB)
- Data is *replicated* N times for durability/availability
  - At Amazon, replicas are in *different data-centres*

Consistent hashing



- Hash determines vnode
- Vnode determines physical node
- Any node can determine where a key is stored

What if a node is unavailable?



- Every key has a *"preference list"* of vnodes
- We use the first N

Latency

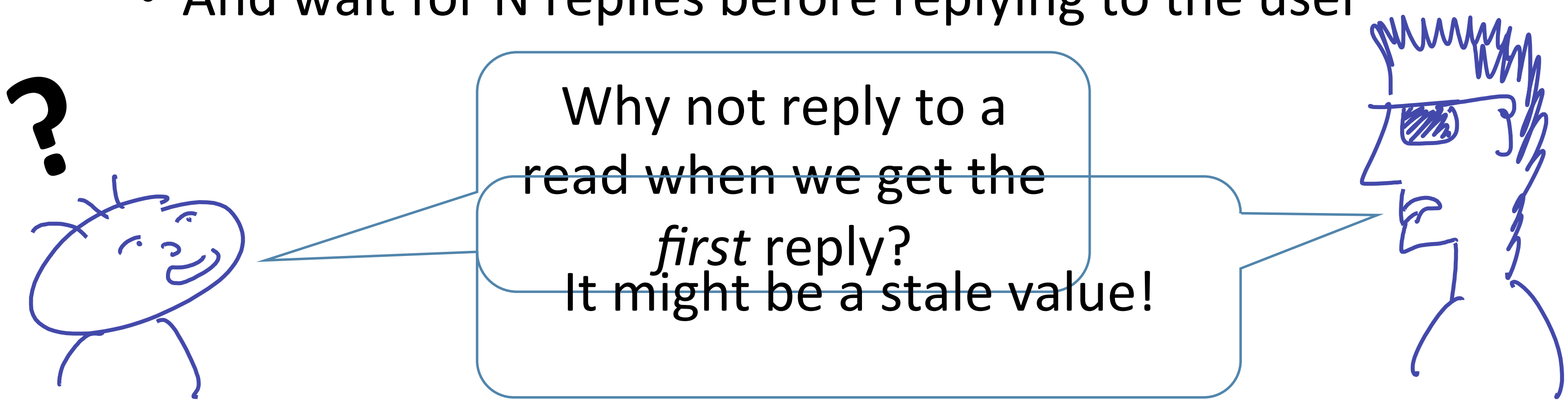
- Put: send writes to N nodes
  - And wait for N acknowledgements before acknowledging the user
- Read: send read requests to N nodes
  - And wait for N replies before replying to the user

What if a node doesn't reply?

Mark it unavailable, use the next from the preference list.

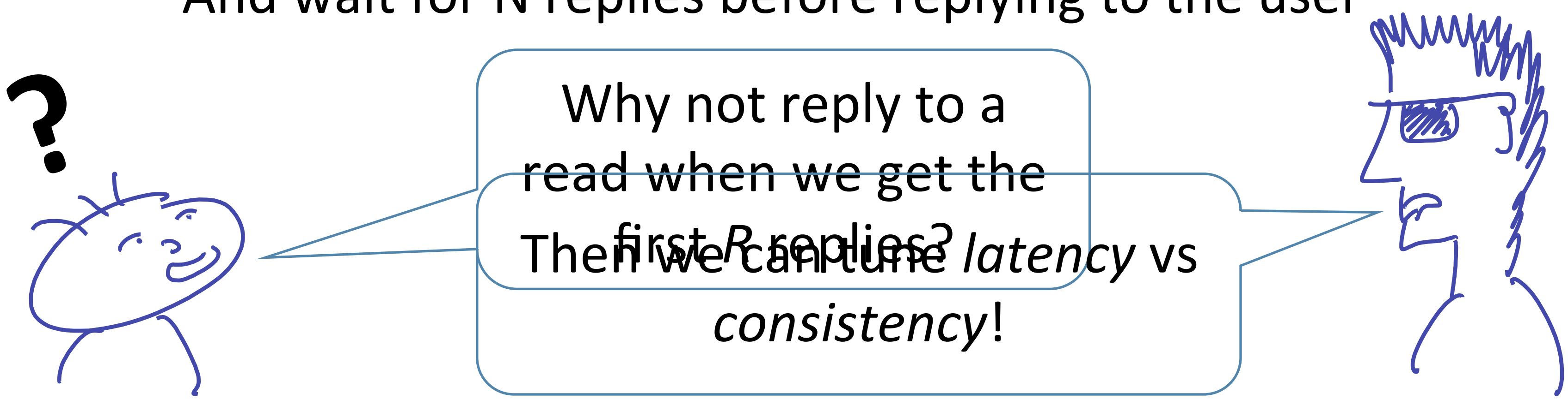
Latency

- Put: send writes to N nodes
  - And wait for N acknowledgements before acknowledging the user
- Read: send read requests to N nodes
  - And wait for N replies before replying to the user



Latency

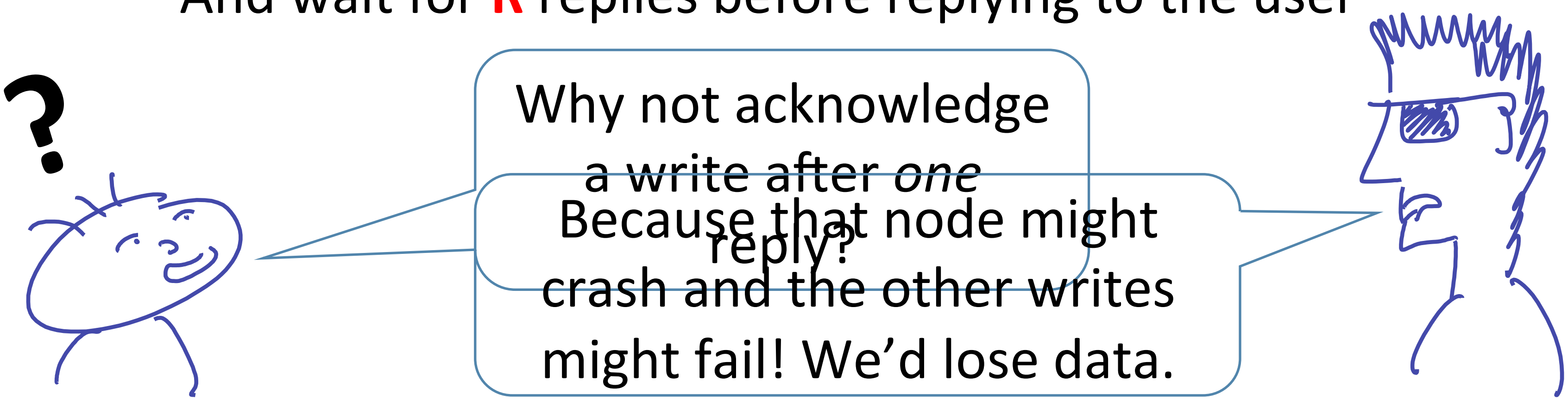
- Put: send writes to N nodes
  - And wait for N acknowledgements before acknowledging the user
- Read: send read requests to N nodes
  - And wait for N replies before replying to the user



Latency

- Put: send writes to N nodes
  - And wait for N acknowledgements before acknowledging the user

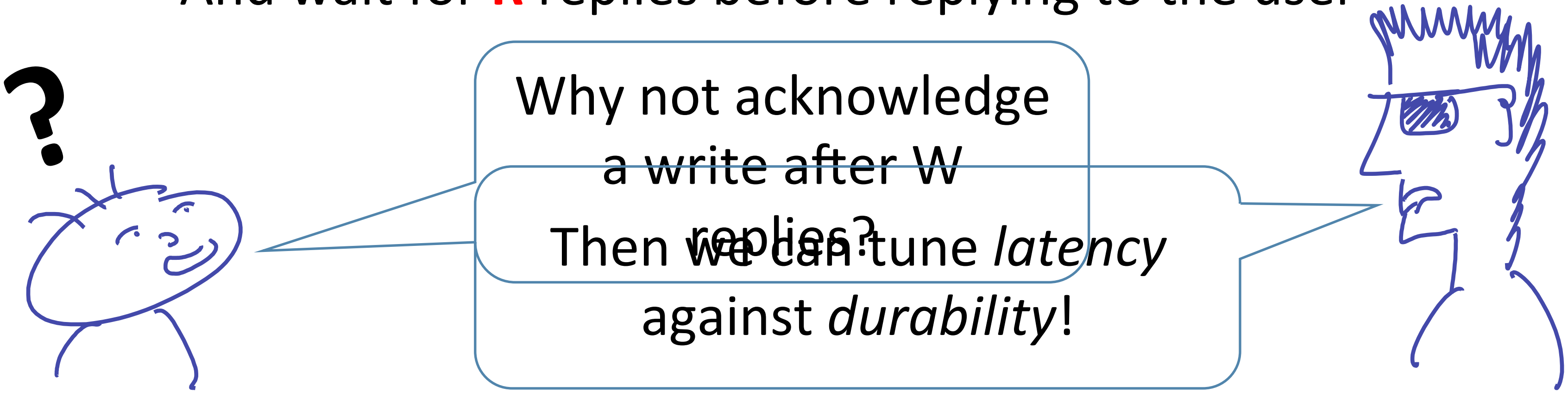
- Read: send read requests to N nodes
  - And wait for **R** replies before replying to the user



Latency

- Put: send writes to N nodes
  - And wait for N acknowledgements before acknowledging the user

- Read: send read requests to N nodes
  - And wait for **R** replies before replying to the user



Latency

- Put: send writes to N nodes
  - And wait for **W** acknowledg
    acknowledging the user

- Read: send read requests to N nodes
  - And wait for **R** replies before replying to user

In practice, a "sloppy quorum", because we use the first N *available* nodes

$R + W > N$

$2 + 2 > 3$

guarantees a *quorum*—each read sees the latest write

Handoff

- What happens if data is *written* to the wrong node, and then the correct node comes back up?
  - We know where the data *should* be stored—we record that on the replacement node.
  - When the correct node recovers, the replacement node ”hands off” the data to the correct one.
  - But before handoff is complete, stale data may be read.

Read repair

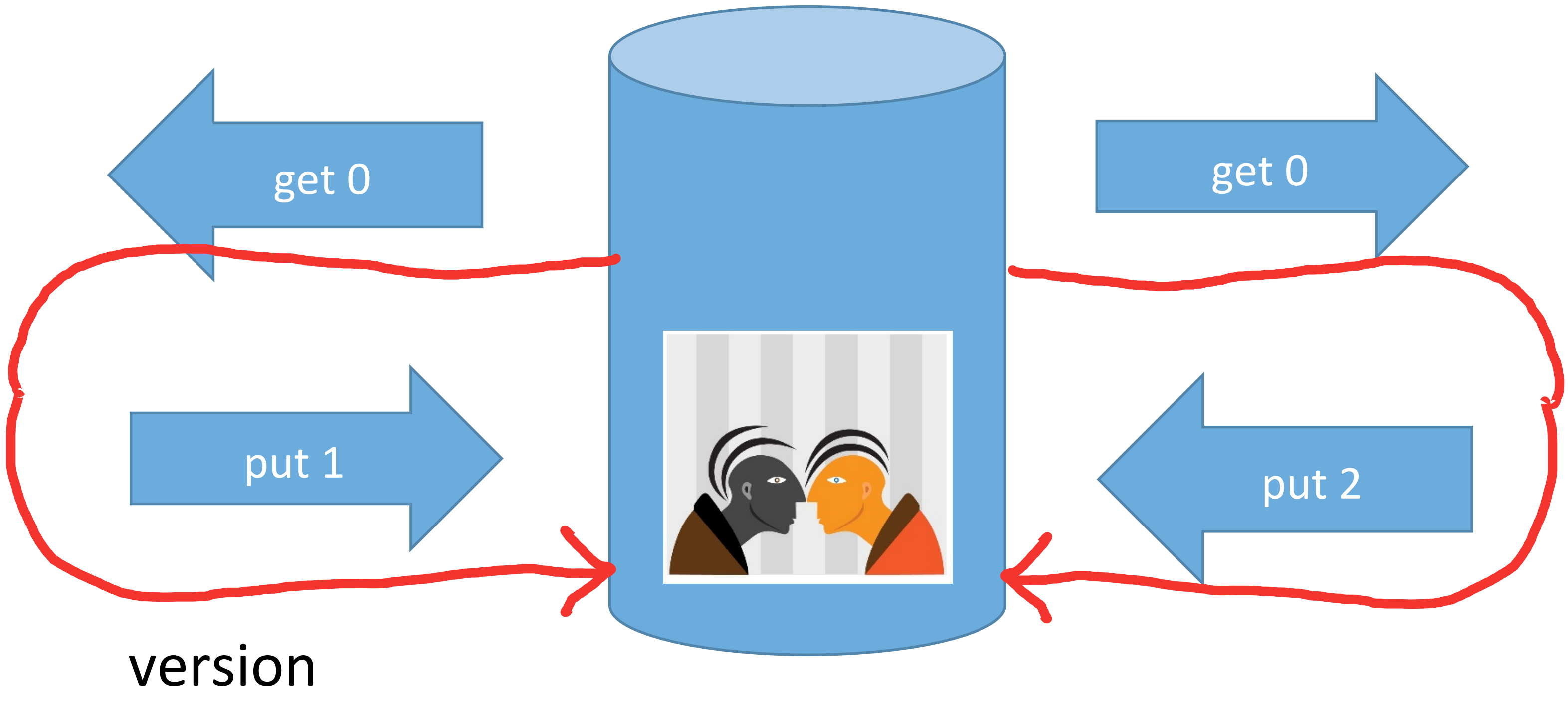
- What happens if we get a key, and one of the replicas returns stale data?
  - We can *update* that replica with fresh data from another replica… ”read repair”.
  - This is why the node co-ordinating a get request waits for all N replies, even if only R are needed to respond to the client.

Recognising stale data

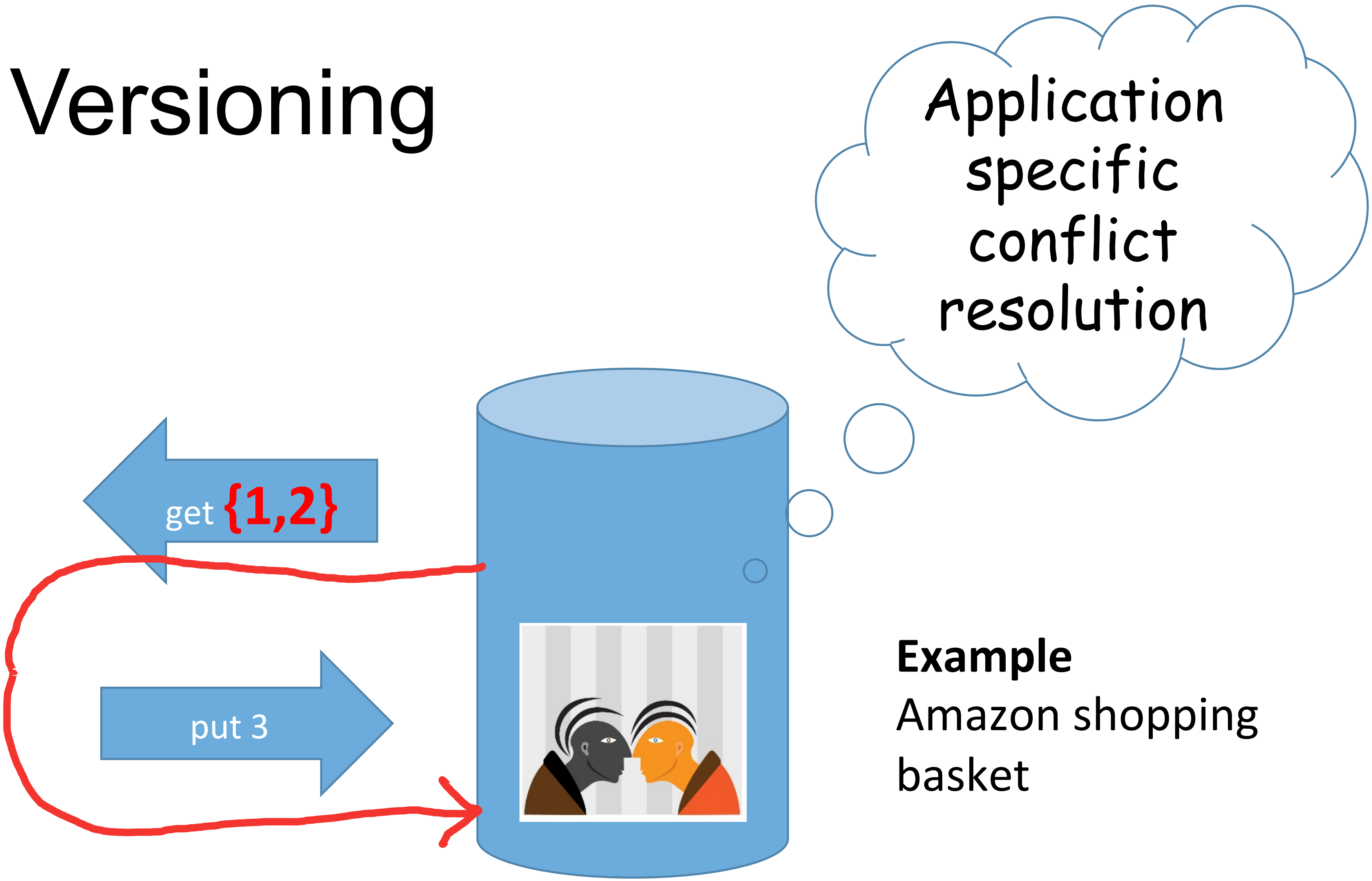
- How do we know when data is out-of-date?
  - Time stamps are *not* reliable!

- Remember that ”context” information…?
  - It’s *version* information… a *vector clock.*

Versioning

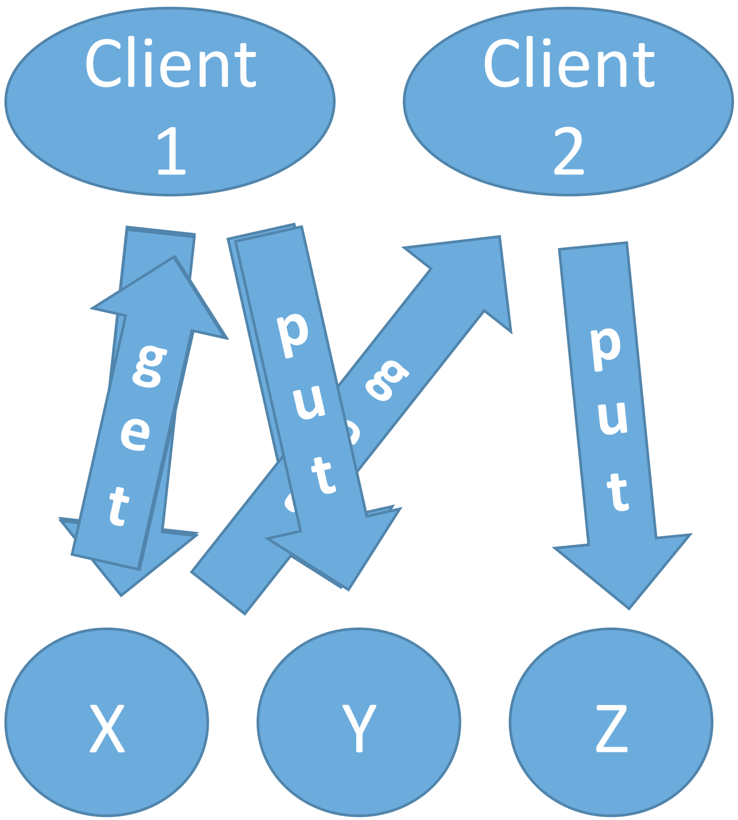


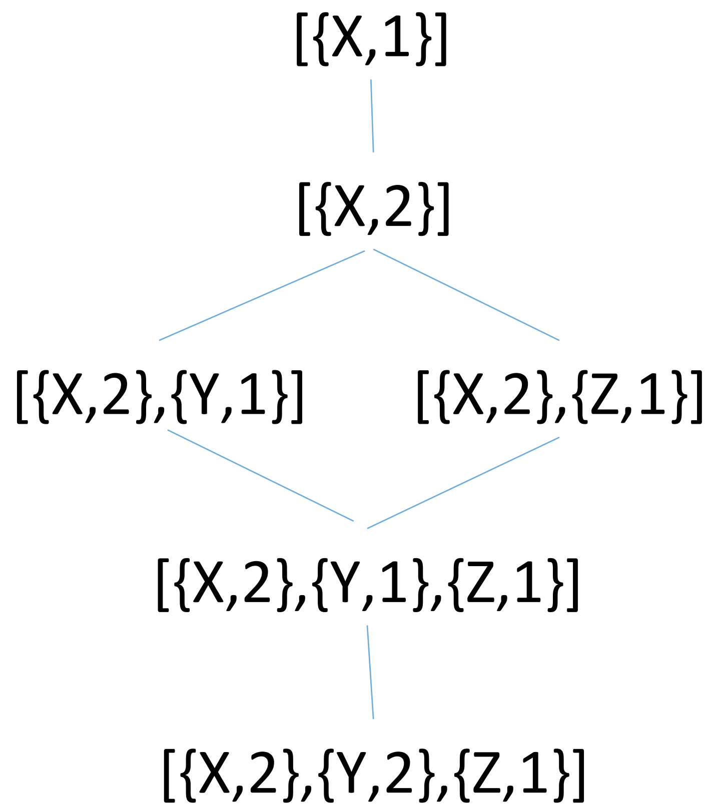
Versioning



Example
Amazon shopping basket

Vector Clocks





Vector Clocks

- Consist of a list of *node ids* and *counts*

- c1 *descends from* c2 if:
  $\forall$ n $\in$ Nodes. count(n,c1) >= count(n,c2)

- A value v1 supercedes v2 if its vector clock descends from the clock of v2.

How often do conflicts arise?

Number of versions	%ge of read requests
1	99,94
2	0,00057
3	0,00047
4	0,00009

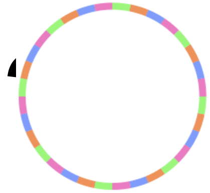
Conflict resolution is only needed occasionally
Source: Amazon

Deletion

- How do we *delete* a key?
- We don't! Write a "tombstone" over it…

  - This means dead keys can come back to life as a result of conflicts…

How do nodes know the ring?

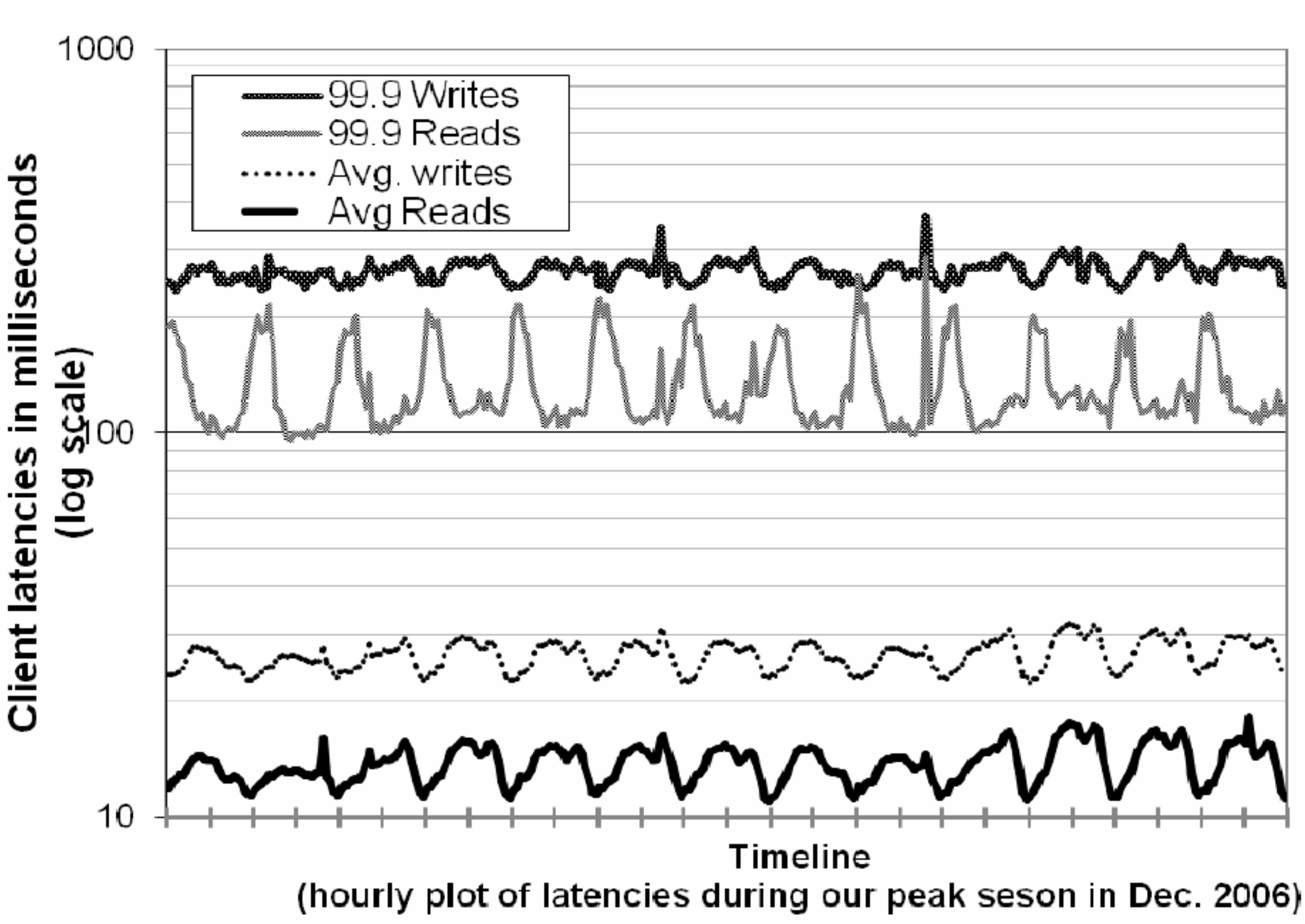


- Joining and leaving the ring is done explicitly
- Nodes ”*gossip*” the ring to each other
    - Periodically send the ring to random other nodes
    - All nodes quickly become aware of changes
- Basho is working on optimizations to this

How do nodes join or leave?

- A new node takes over its share of vnodes from other nodes
- For balanced load, it should take roughly the same number of vnodes from each other node
- Requires many more vnodes than nodes!
- Adding a node can take a day…

How well does it work?



Riak

- Riak Core
  - Distribution framework: vnodes, consistent hashing, the ring, vector clocks, etc
- Riak KV
  - The key-value store
- Riak Map-Reduce
  - For aggregating data
- Riak Pipe
  - A generalisation of map-reduce
- …

Borrowing from Sean Cribbs…

To review, Riak is...

To review, Riak is...

- Magical Horizontally-Scaling Unicorns

To review, Riak is...

- Magical Horizontally-Scaling Unicorns
- Application-Friendly Rainbows in the Cloud

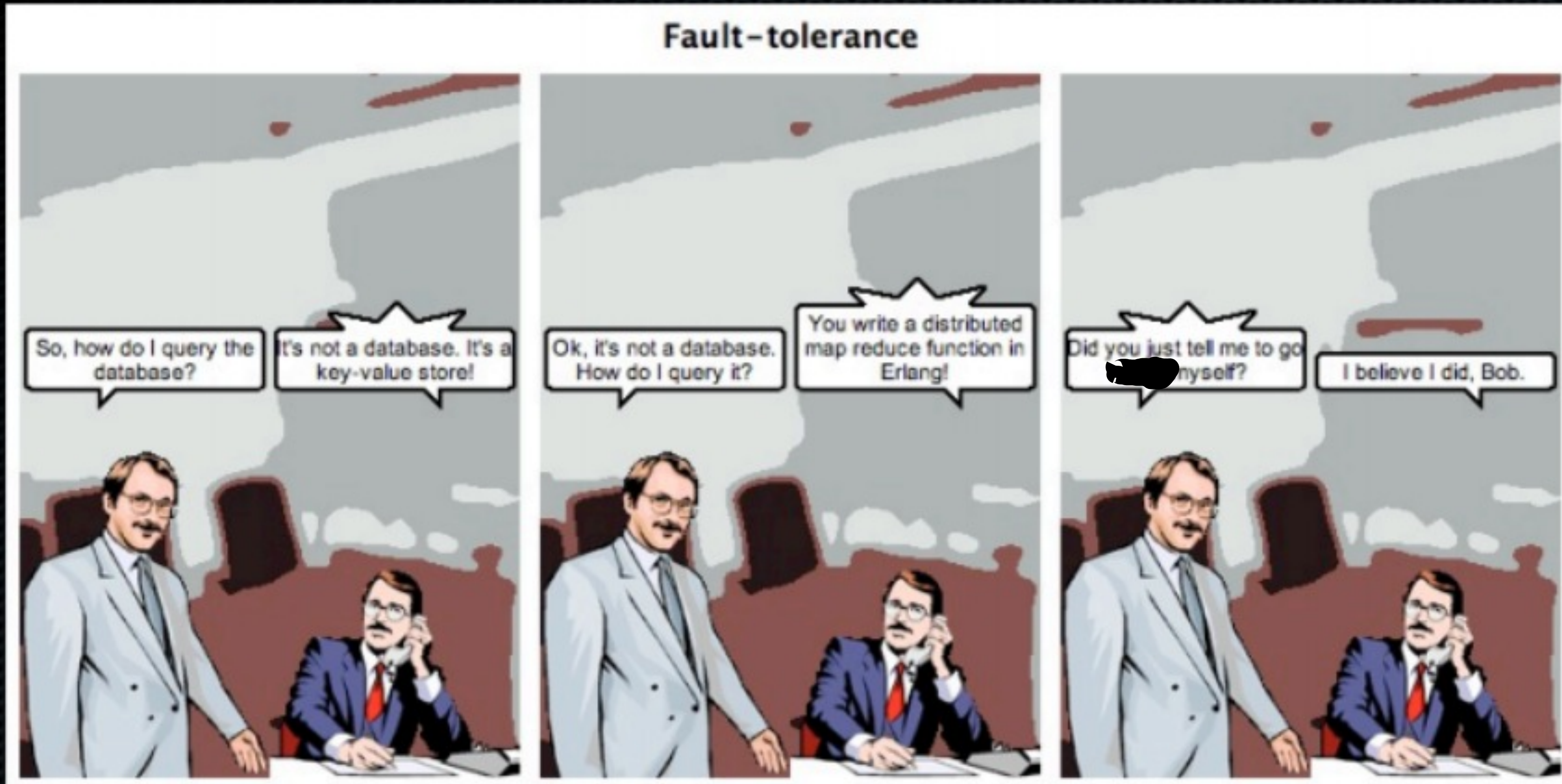
To review, Riak is...

- Magical Horizontally-Scaling Unicorns
- Application-Friendly Rainbows in the Cloud
- Data-Shredding MapReduce Ninjas

This Thursday

- **Guest lecture:** Russell Brown from Basho in the UK

- **CRDTs:** Convergent Replicated Datatypes
(a recent Riak extension for higher performance)
  - What are they?
  - Equivalents in Haskell
  - Turning research into a product
  - QuickCheck's rôle in all of this



Riak Map-Reduce

I believe I did, Bob.