Distributed Computing

A-12. Eventual Consistency

Remember ACID and CAP?

- ACID: Atomicity, Consistency, Isolation, Durability
- CAP: Consistency, Availability, Partition Tolerance
    - Can’t have all, choose two
- If you have a partition you can
    - Choose consistency (CP) and lose availability
    - Choose availability (AP) and lose consistency
    - Or some hybrid
- In the beginning of the course we’ve seen CP systems: if Paxos/Raft are partitioned, the minority will stop working
- Now, armed with what we’ve seen till now, we’ll delve in AP systems

Eventual Consistency

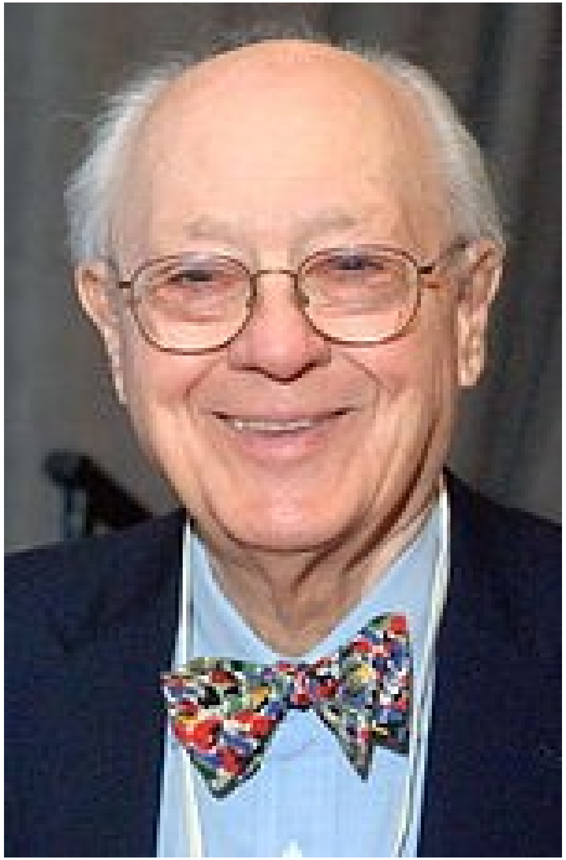
Reference

- Eric Brewer's talk NoSQL: Past, Present, Future, 2012.

NoSQL before SQL

- Charles Bachmann, 1973 Turing Award
- IDS (Integrated Data Store), navigational database

```
get department with name='Sales'
get first employee in set department-employees
until end-of-set do {
  get next employee in set department-employees
  process employee
}
```

1970s: Relational vs UNIX

- Relational: top-down approach
  - Easy-to-grasp abstraction, one API does it all (SQL)
  - Declarative language, user doesn't care about optimizing
  - Data outlasts implementations
  - Transactions
- UNIX: Bottom-up
  - Few, simple, efficient mechanisms
  - Compose tools

Two World Views

- Relational
  - Clean model, ACID transactions
  - Two kinds of developers: DB authors and SQL programmers
  - Do one important thing, do it well
- Systems
  - Bottom up, new modules to add functionality
  - One kind of programmer
  - Flexible systems that can grow to do new things

Brewer’s Story

- 1996-1998: build a search engine and a proxy cache
  - Didn’t use a DBMS: custom servers on top of file systems were faster
  - Because DBMS’s features cost performance
- 1997: ACID vs. BASE (Basically Available, Soft State, Eventual Consistency)
  - Not well received: people liked ACID
- 1999: CAP Theorem
- Mid-00‘s: Eventually consistent systems start to get used

Partition Mode

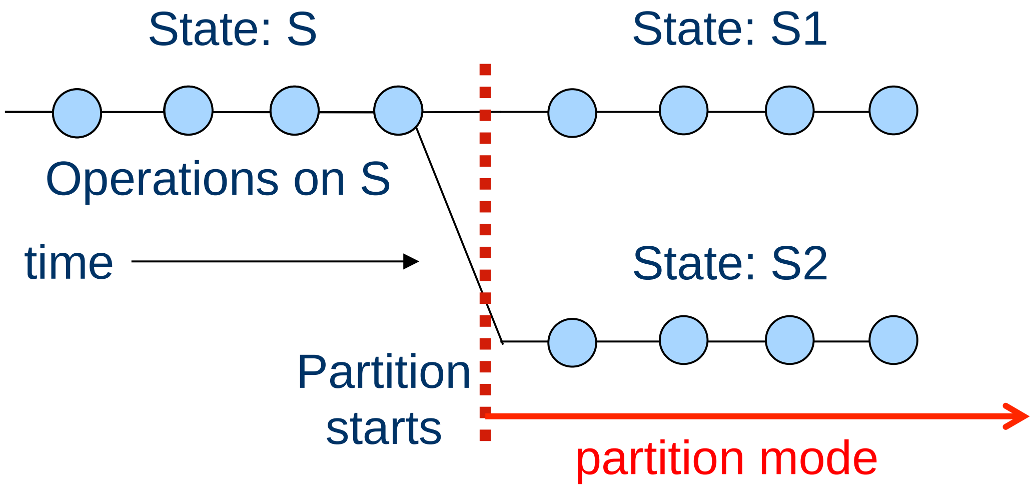


Image by E. Brewer

- Special mode the system can detect
  - Allow commits?
  - Give output?
  - Allow it to impact the outer world?

Partition Recovery (1)

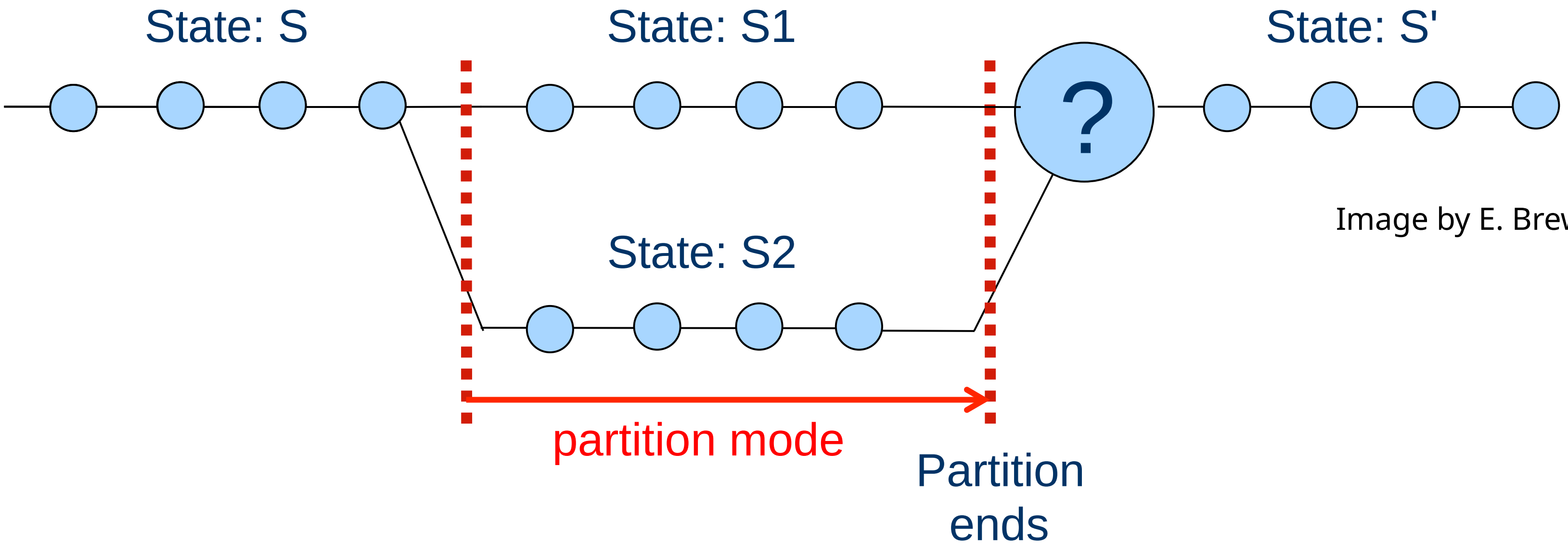


Image by E. Brewer

- How to get back to a consistent state afterwards?
  - Last writer wins?
  - Rules that depend on what has been done?

Partition Recovery (2)

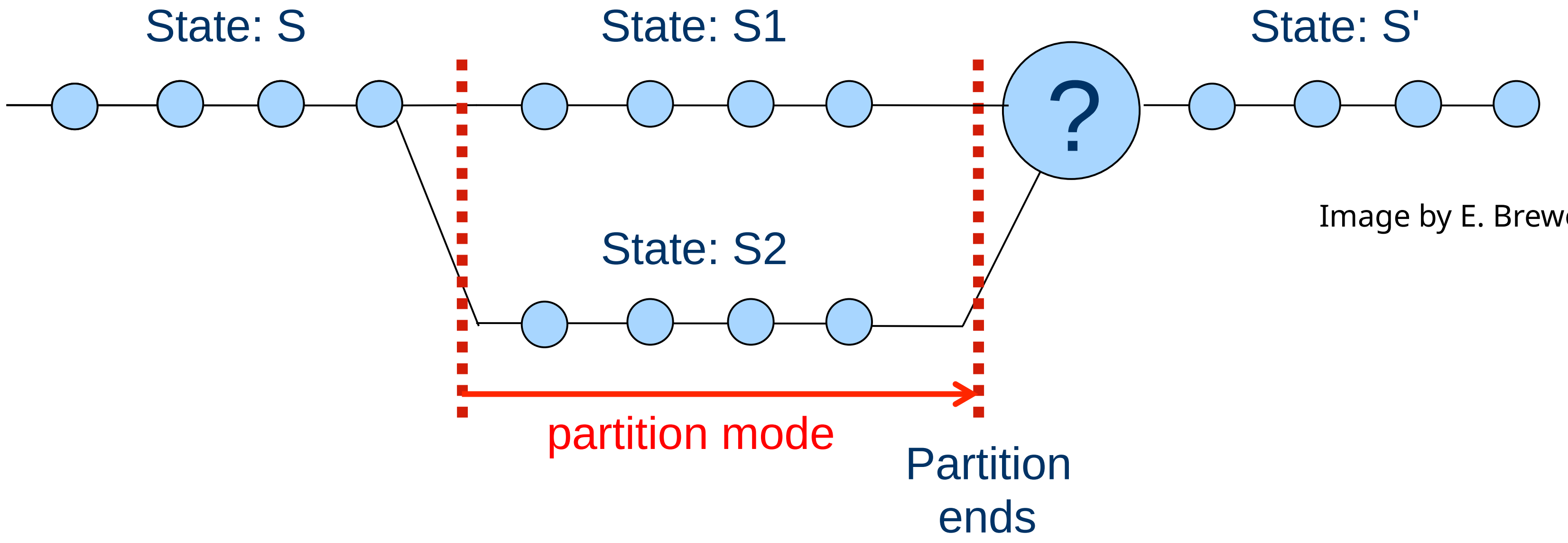


Image by E. Brewer

- Detect and repair mistakes
  - What have we done wrong?
  - How do we deal with it?

ATMs and “Stand In” Time

- ATMs do keep operating when isolated from the network
  - “Partition mode”, indeed
  - Operations are **commutative**: increment, decrement
- Limit damage
  - E.g., give out at most 200€ per person
- Partition recovery:
  - Detect errors (balance below zero)
  - Compensate (overdraft penalty)

Eventually Consistent Systems

- Three key issues to address:
  - Detect partitions
  - Define how “partition mode” works
  - Define how to do recovery
- The real world is eventually consistent!
  - There are “consistency rules” (laws, contracts, …)
  - You see problems (inconsistency detection)
  - You compensate for it (money, …)

Amazon Dynamo

References

- DeCandia et al. Dynamo: amazon's highly available key-value store. ACM SOSP 2007.
- Lakshman and Malik. Cassandra - A Decentralized Structured Storage System. ACM LADIS 2009.
- Cassandra documentation: Dynamo.

About Dynamo

- Origin: handling shopping carts at Amazon
- Availability affects income! As available as possible
  - Trade off with consistency
- Born as a key-value store
  - Later evolved as a more complete DB, as usual
- Uses techniques seen in all the course
- Cassandra (Facebook, now Apache) has a very similar architecture

Requirements

- “Always writeable”
  - You can always add an item to your shopping cart
  - Can write in partition mode
- *User-perceived* consistency
- Guaranteed latency measured at percentile **99.9**
- Parameters to tune cost, consistency, durability and latency
- Scale **out** to tens of thousands of servers
  - 2007: tens of millions requests, >3M checkouts in a single day

Key Idea

- Chord in a datacenter (nodes are servers)
  - Consistent hashing: adding/removing one node at a time is cheap
- Completely decentralized
- Each item is replicated in the N nodes “after” a given key in the ring
  - Those nodes are called “preference list”
  - Replication guarantees durability

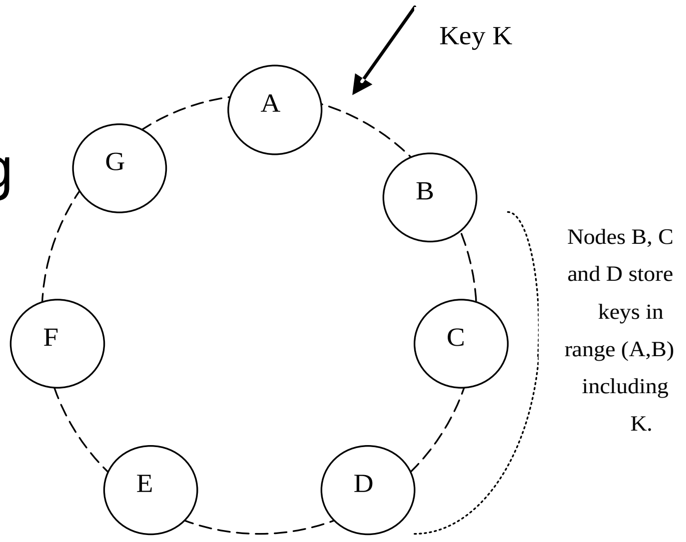


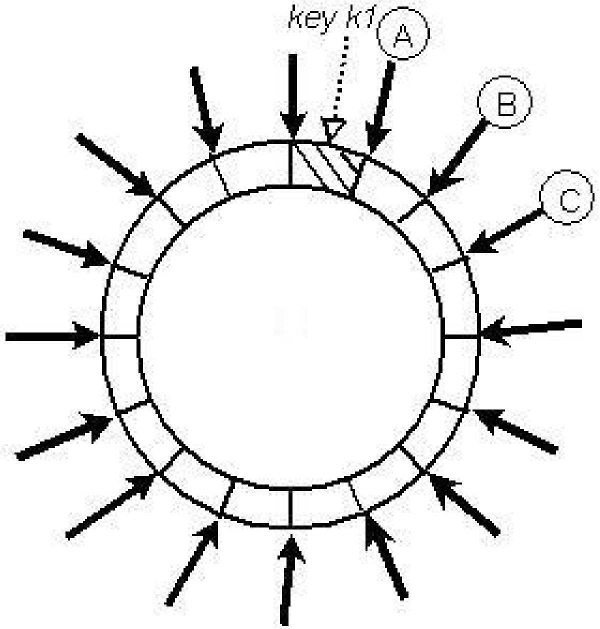
Image from DeCandia et al., SOSP ‘07

Lookup

- Unlike Chord, every node knows the **full partition table**
  - One hop to get to any piece of data
  - Routing table updated via a **gossiping** algorithm: every node periodically exchanges information with a random set of other nodes
  - Gossiping also handles **failure detection**
- Optionally, the client knows the routing table as well
  - Data can be directly asked to the node having it

Faster Partitioning

- Split the hashing space in Q partitions
- They get distributed equally between nodes
- When nodes join, they “steal” partitions from other nodes
- When they leave, they are redistributed to other nodes
- Transferring partitions **doesn’t require random disk accesses**



API

- get(key) → [value], version_info
- put(key, value, version_info)

- get() returns a **list** of values
  - May be more than one in case of inconsistencies
  - Will be handled by the client
- version_info is passed to the subsequent put to solve some inconsistencies
  - If something is created from scratch, version_info is null

Sloppy Quorum

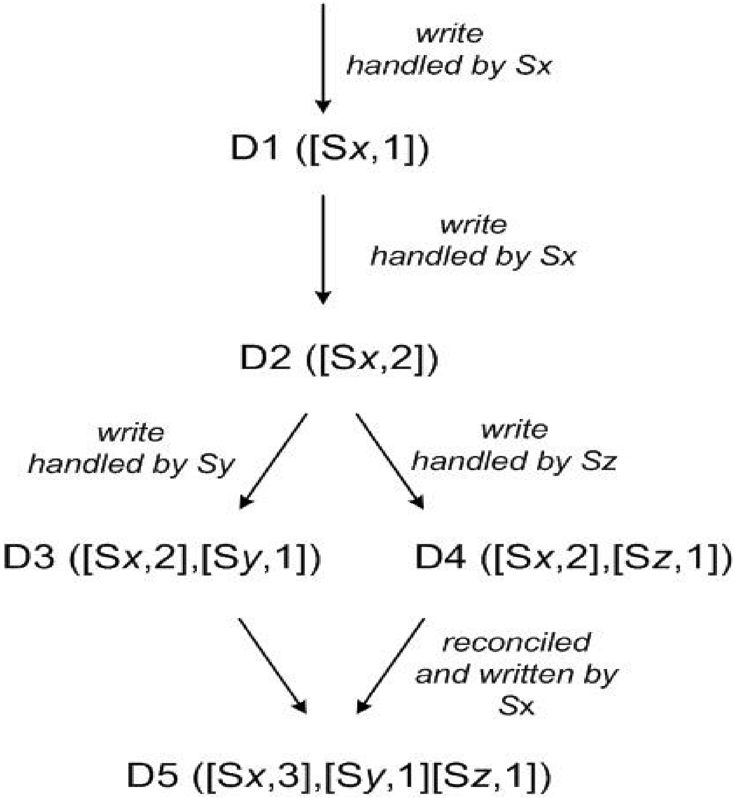
- Three configurable parameters:
  - N: number of copies for each piece of data (often, 3)
  - R: number of reads to get a successful read
    - Low R: fast read, high R: consistent
  - W: number of writes to get a successful write
    - Low W: fast write, high W: consistent
- If R+W > N, it is sort-of like a consensus algorithm, i.e., high consistency
  - Except failures

Example configurations:

- N=3, R=2, W=2 (default)
  - Consistent & durable
- N=x, R=1, W=x
  - Slow writes, fast reads (great for read-intensive workloads)
- N=R=W=1
  - Cache (e.g., web cache)

Solving (Some) Inconsistencies: Vector Clocks

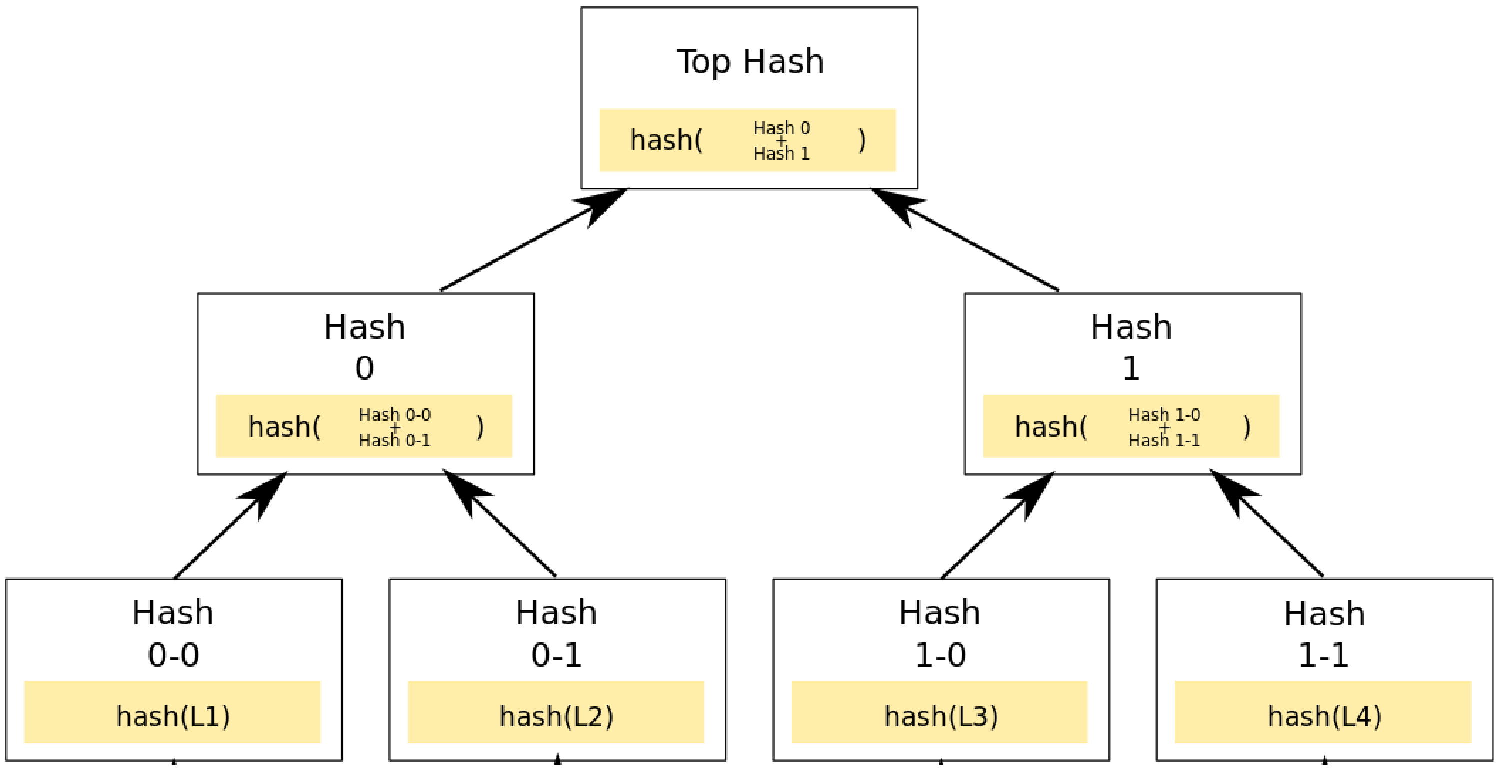
- version_info gets a counter value for each machine they have passed through
  - Idea from 1986 (Ladin and Liskov)
- One copy supersedes another if counters are not smaller for each machine
- Otherwise, they're independent and we ask the client what to do



Failures

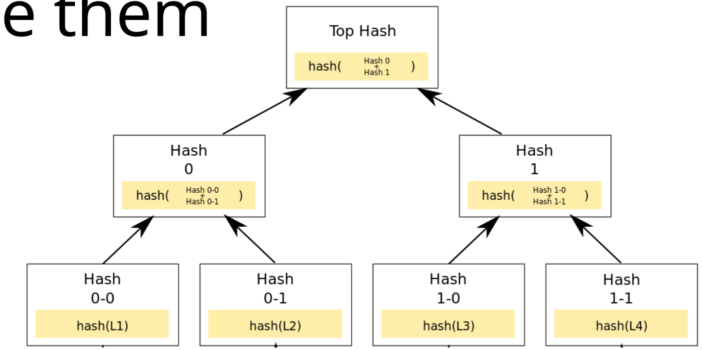
- When machines go offline, it’s considered transient
  - Permanent addition or removal is an administrator action
- Reads and writes spill over to the first machine in the ring after the N that should handle them by default
- When the machine comes back online, updates are reported to it
- Can create some rare inconsistencies even when R+W > N

Merkle Trees



Anti-Entropy

- Merkle trees are used to compare data between nodes that store replicas of a partition
- If the root is different, compare the children to find out which half is different, and so on recursively
- Fast way to spot differences & reconcile them



Client-Side Reconciliation

- If everything else fails, the client is presented with more than one return value
- What to do looks depends on the application
  - Amazon cart policy: in doubt, leave stuff it in the cart!
  - We’ve seen what an ATM would do
- Can be reminiscent of exception handling
- Rare: Amazon reports 0.06% cases of more than 1 value returned

Other Optimizations

- Buffered writes: wait for a few writes to be committed before writing to the disk
  - Performance/consistency tradeoff
- Throttling background operations
  - Slow down gossip/maintenance when many requests are around
- Let coordinate read/writes to nodes who are responding fastest
  - Additional load balancing

Some numbers from the paper

- Tens of thousands of machines
- Tens of millions of requests, >3M checkouts in a day
- Response time below 400ms at 99.9 percentile (avg below 40)
- In 99.94% cases, requests return exactly one version
- 99.9995% successful responses without timeouts
  - Equivalent to **2.5 minutes** of unavailability in a **year**

What About Cassandra?

- Project by Facebook, now handled by the Apache foundation
- Very similar architecture
- Zookeeper for routing table and seeds
- Rack-aware & datacenter-aware data placement
  - Again uses Zookeeper to elect a leader and coordinate it
- No vector clocks, just get a timestamp, and the latest wins
- Lightly-loaded nodes get “migrated” on the ring