Dynamo

Tom Anderson and Doug Woos

Dynamo motivation

Fast, available writes

- Shopping cart: always enable purchases

FLP: consistency and progress at odds

- Paxos: must communicate with a quorum

Performance: strict consistency = "single" copy

- Updates serialized to single copy
- Or, single copy moves

Why Fast Available Writes?

Amazon study: 100ms increase in response time

=> 5% reduction in revenue

Similar results at other ecommerce sites

99.99% availability

=> less than an hour outage/year (total)

Amazon revenue > $300K/minute

Dynamo motivation

Dynamo goals

- Expose "as much consistency as possible"
- Good latency, 99.9% of the time
- Easy scalability

Dynamo consistency

Eventual consistency

- Can have stale reads
- Can have multiple "latest" versions
- Reads can return multiple values

Not sequentially consistent

- Can't "defriend and dis"

External interface

get : *key* -> ([*value*], *context*)

- Exposes inconsistency: can return multiple values
- *context* is opaque to user (set of vector clocks)

put : (*key*, *value, context*) -> *void*

- Caller passes context from previous get

Example: add to cart

```
(carts, context) = get("cart-" + uid)
cart = merge(carts)
cart = add(cart, item)
put("cart-" + uid, cart, context)
```

Resolving conflicts in application

Applications can choose how to handle inconsistency:

- Shopping cart: take union of cart versions
- User sessions: take most recent session
- High score list: take maximum score

Default: highest timestamp wins

Context used to record causal relationships between gets and puts

- Once inconsistency resolved, should stay resolved
- Implemented using vector clocks

Dynamo's vector clocks

Each object associated with a vector clock

- e.g., [(node1, 0), (node2, 1)]

Each write has a coordinator, and is replicated to multiple other nodes

- In an eventually consistent manner

Nodes in vector clock are *coordinators*

Dynamo's vector clocks

Client sends clock with put (as context)

Coordinator increments its own index in clock, then replicates across nodes

Nodes keep objects with conflicting vector clocks

- These are then returned on subsequent gets

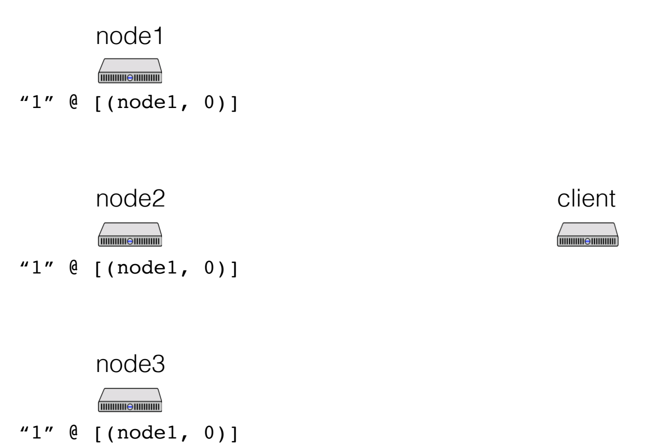
If clock(v1) < clock(v2), node deletes v1

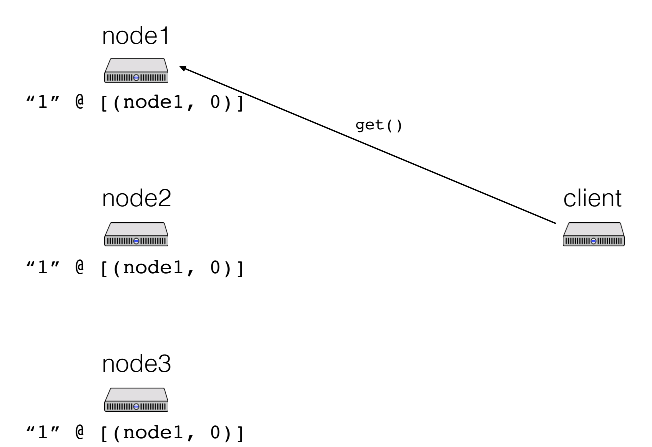
Dynamo Vector Clocks

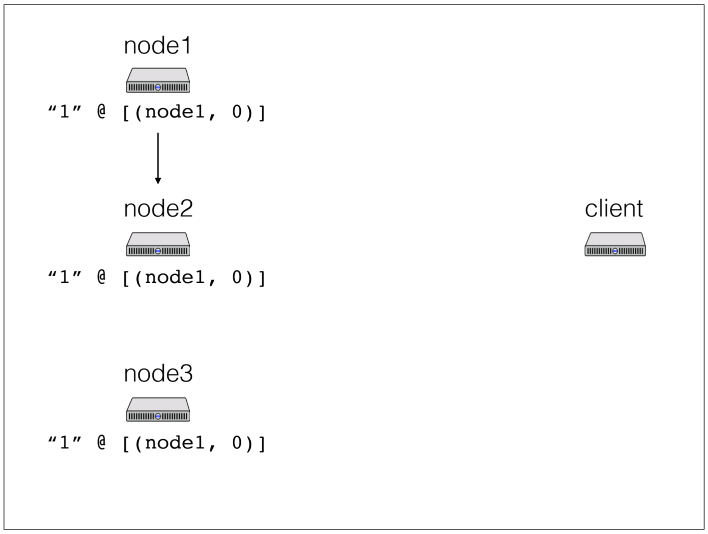
Vector clock returned as context with get

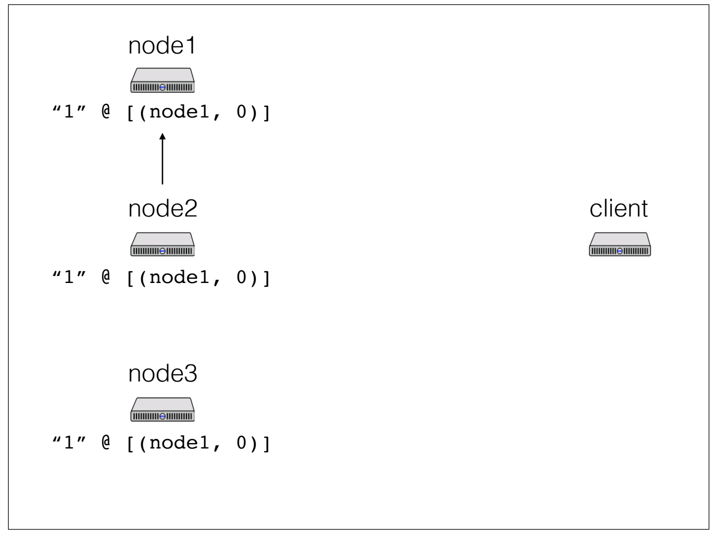
- Merge of all returned objects' clocks

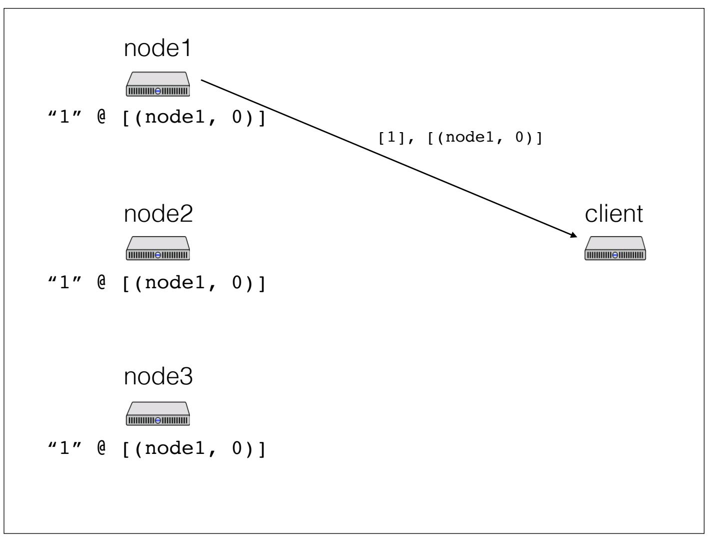
Used to detect inconsistencies on write

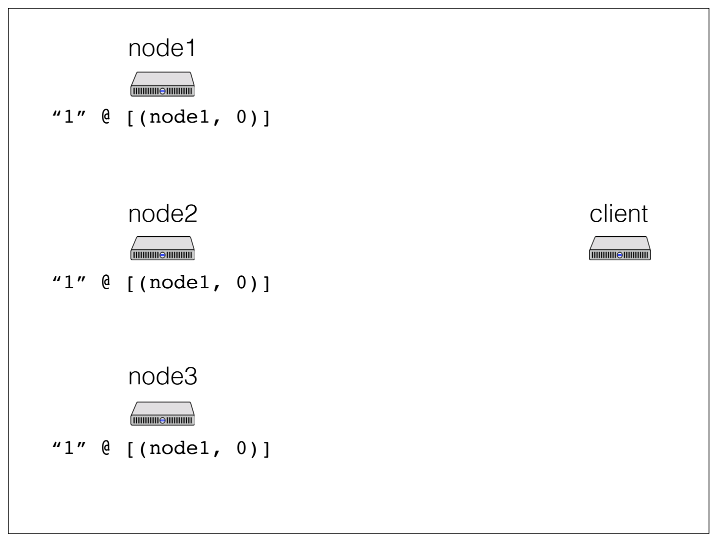


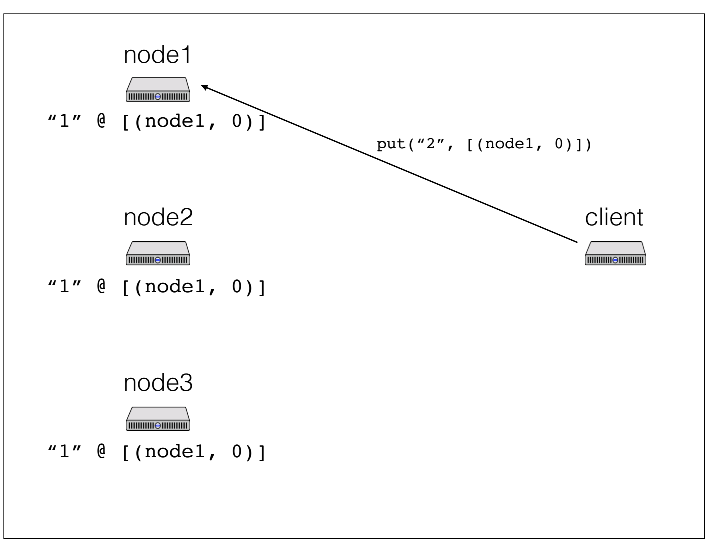


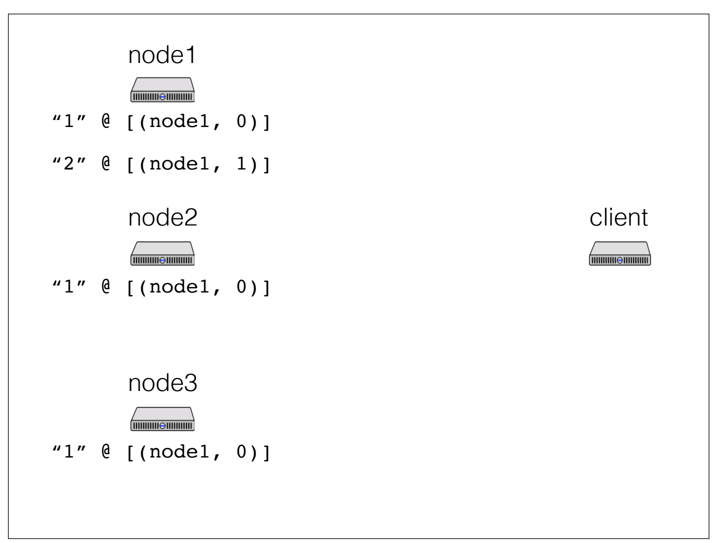

node1
"1" @ [(node1, 0)]
node2
"1" @ [(node1, 0)]
client
node3
"1" @ [(node1, 0)]

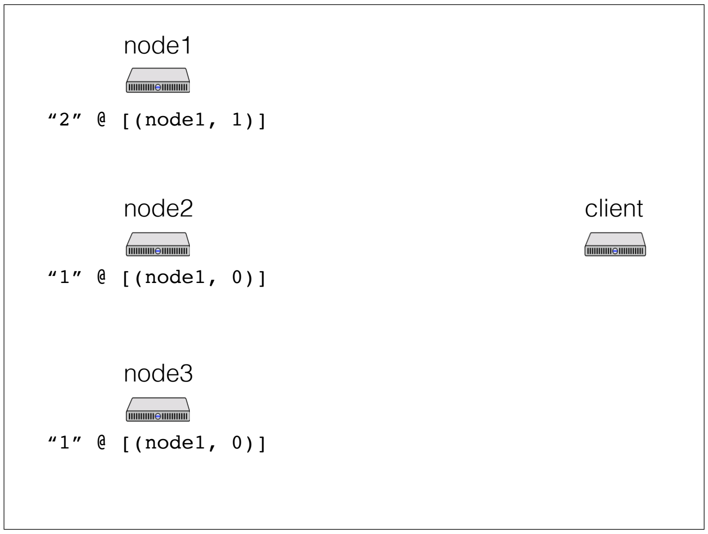

node1
"1" @ [(node1, 0)]
node2
"1" @ [(node1, 0)]
client
node3
"1" @ [(node1, 0)]

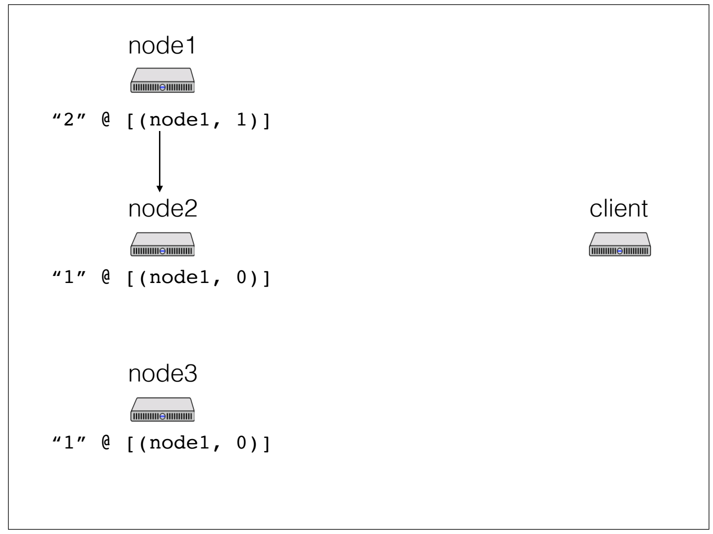

node1
"1" @ [(node1, 0)]
[1], [(node1, 0)]
node2
"1" @ [(node1, 0)]
client
node3
"1" @ [(node1, 0)]

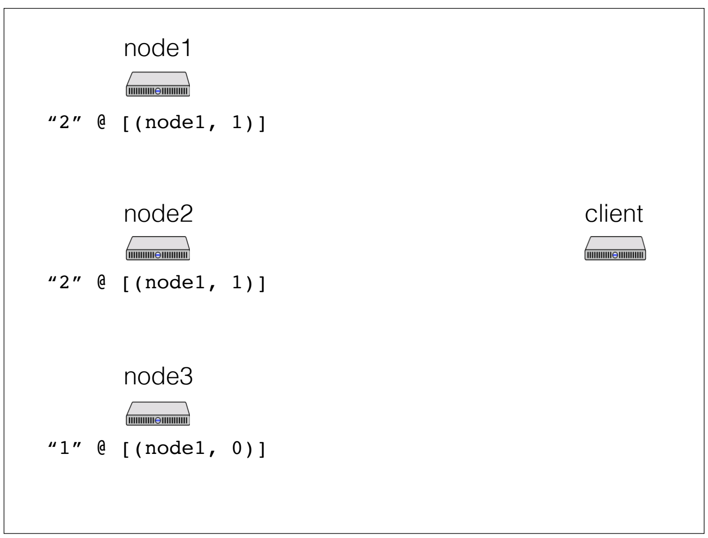

node1
"1" @ [(node1, 0)]
node2
"1" @ [(node1, 0)]
client
node3
"1" @ [(node1, 0)]

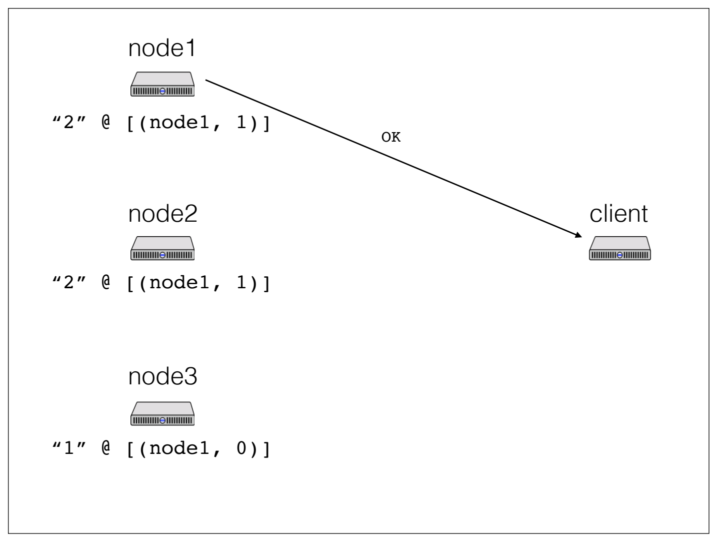

node1
"1" @ [(node1, 0)]
put("2", [(node1, 0)])
node2
"1" @ [(node1, 0)]
client
node3
"1" @ [(node1, 0)]

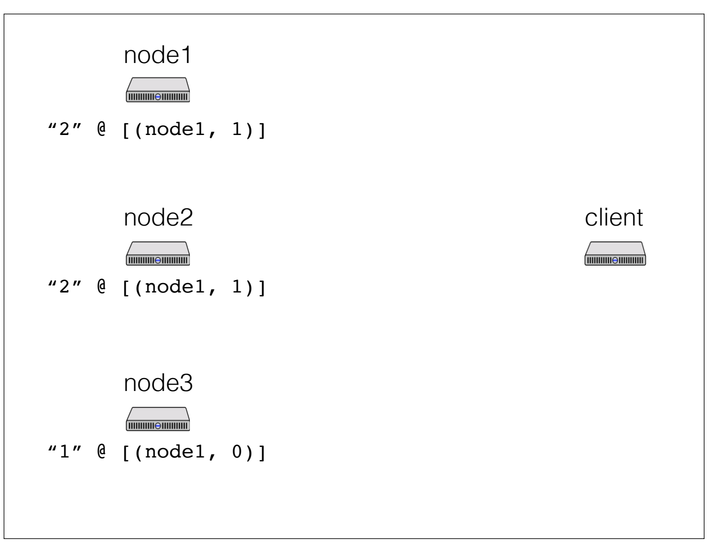

node1
"1" @ [(node1, 0)]
"2" @ [(node1, 1)]
node2
"1" @ [(node1, 0)]
client
node3
"1" @ [(node1, 0)]

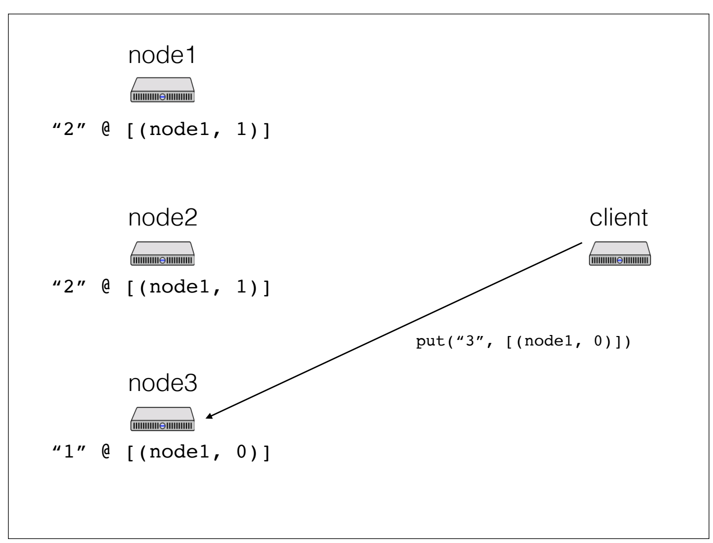

node1
“2” @ [(node1, 1)]
node2
“1” @ [(node1, 0)]
client
node3
“1” @ [(node1, 0)]

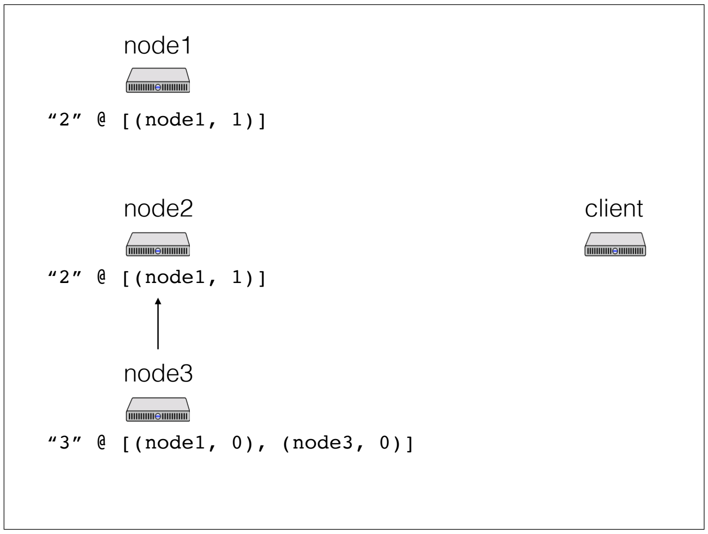

node1
“2” @ [(node1, 1)]
node2
“1” @ [(node1, 0)]
client
node3
“1” @ [(node1, 0)]

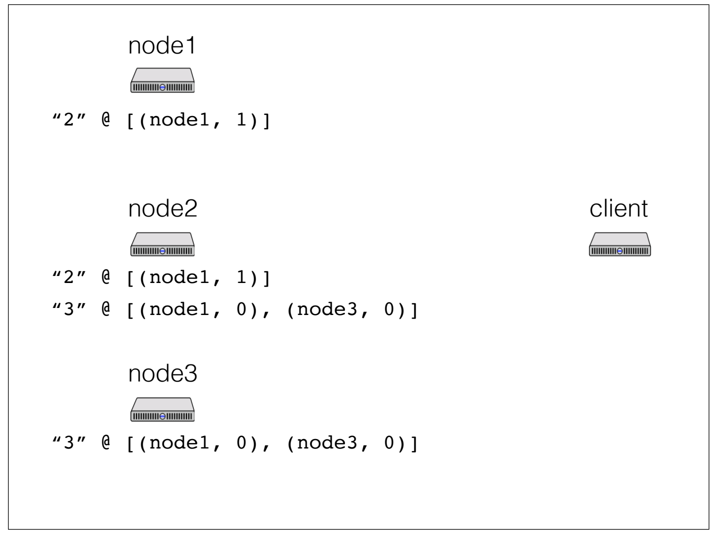

node1
“2” @ [(node1, 1)]
node2
“2” @ [(node1, 1)]
client
node3
“1” @ [(node1, 0)]

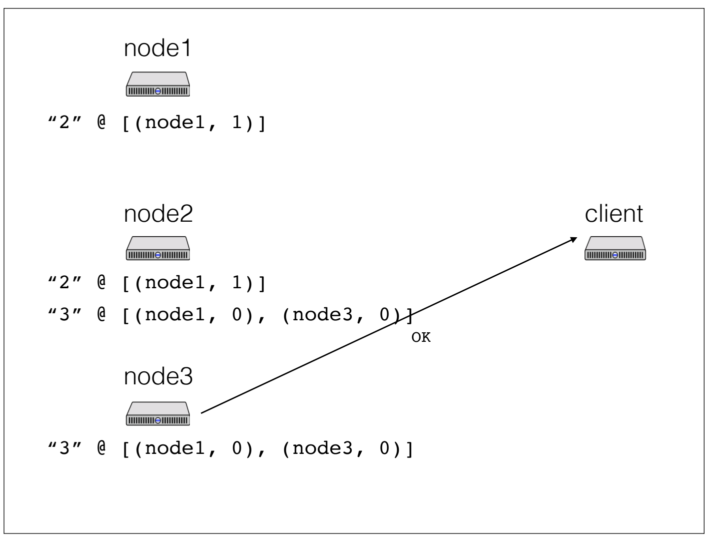

node1
“2” @ [(node1, 1)]
OK
node2
“2” @ [(node1, 1)]
client
node3
“1” @ [(node1, 0)]

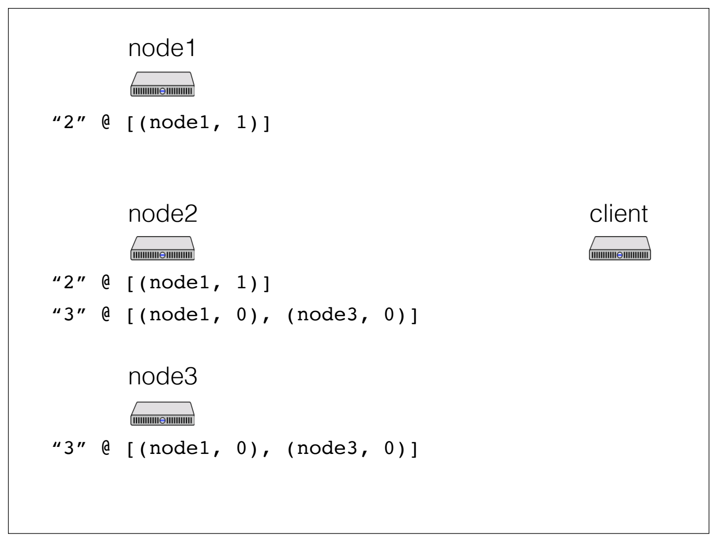

node1
“2” @ [(node1, 1)]
node2
“2” @ [(node1, 1)]
client
node3
“1” @ [(node1, 0)]

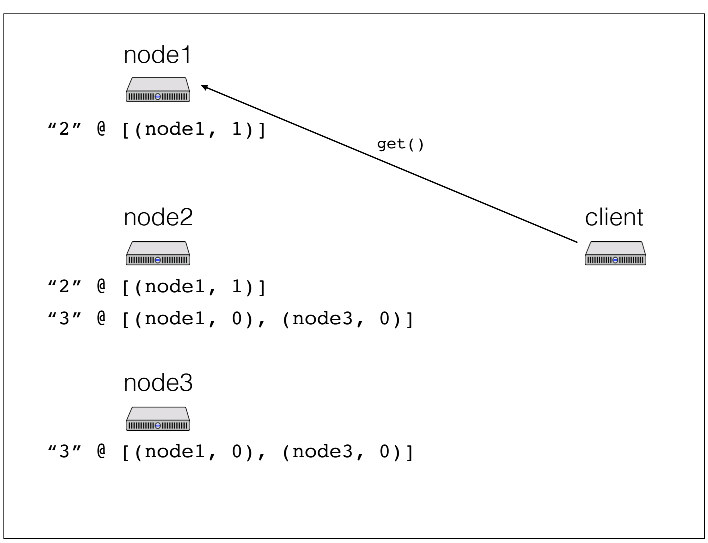

node1
“2” @ [(node1, 1)]
node2
“2” @ [(node1, 1)]
client
put(“3”, [(node1, 0)])
node3
“1” @ [(node1, 0)]

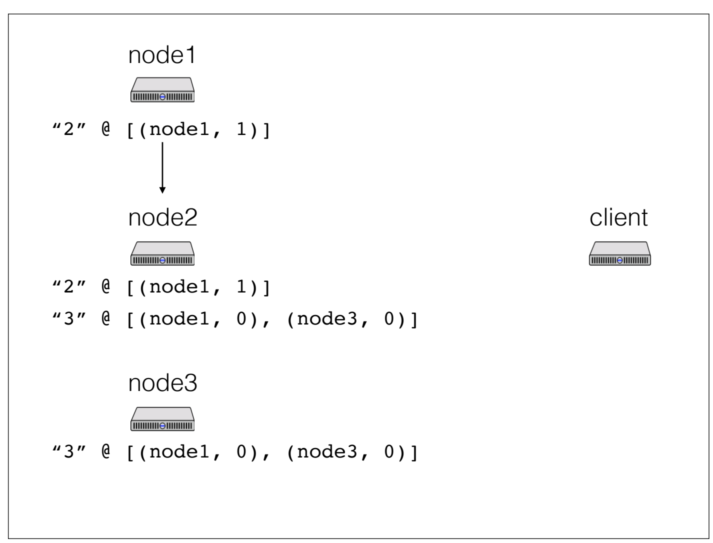

node1
"2" @ [(node1, 1)]
node2
client
"2" @ [(node1, 1)]
node3
"3" @ [(node1, 0), (node3, 0)]

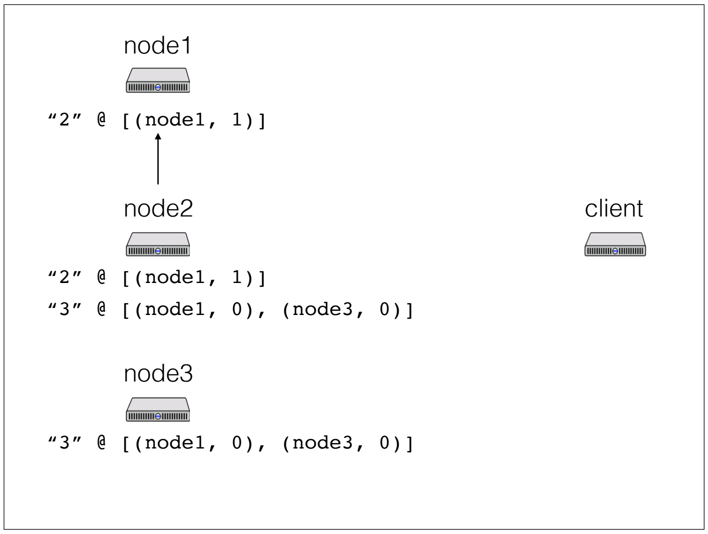

node1
"2" @ [(node1, 1)]
node2
client
"2" @ [(node1, 1)]
"3" @ [(node1, 0), (node3, 0)]
node3
"3" @ [(node1, 0), (node3, 0)]

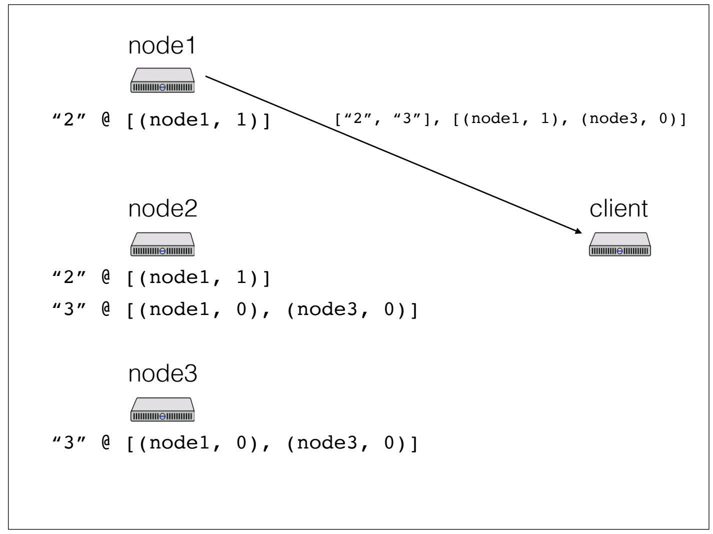

node1
"2" @ [(node1, 1)]
node2
client
"2" @ [(node1, 1)]
"3" @ [(node1, 0), (node3, 0)]
OK
node3
"3" @ [(node1, 0), (node3, 0)]

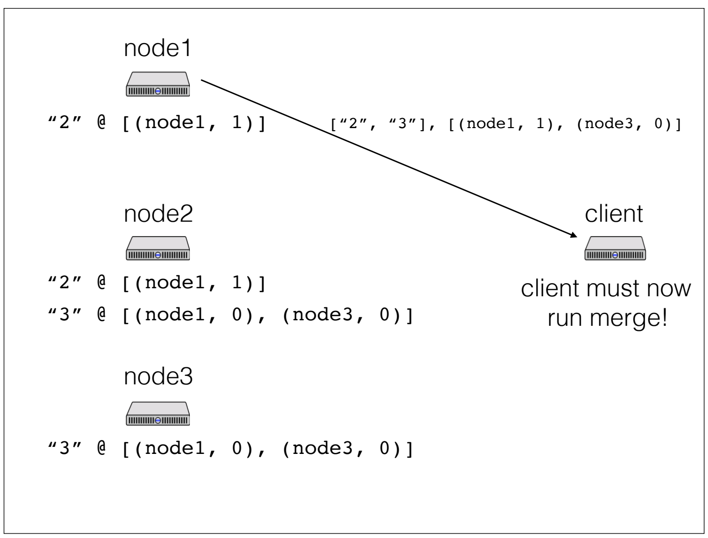

node1
"2" @ [(node1, 1)]
node2
client
"2" @ [(node1, 1)]
"3" @ [(node1, 0), (node3, 0)]
node3
"3" @ [(node1, 0), (node3, 0)]

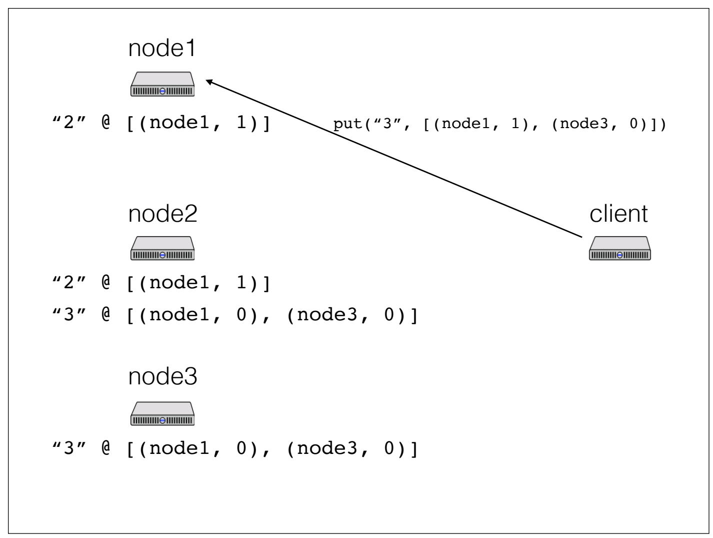

node1
"2" @ [(node1, 1)]
get()
node2
client
"2" @ [(node1, 1)]
"3" @ [(node1, 0), (node3, 0)]
node3
"3" @ [(node1, 0), (node3, 0)]

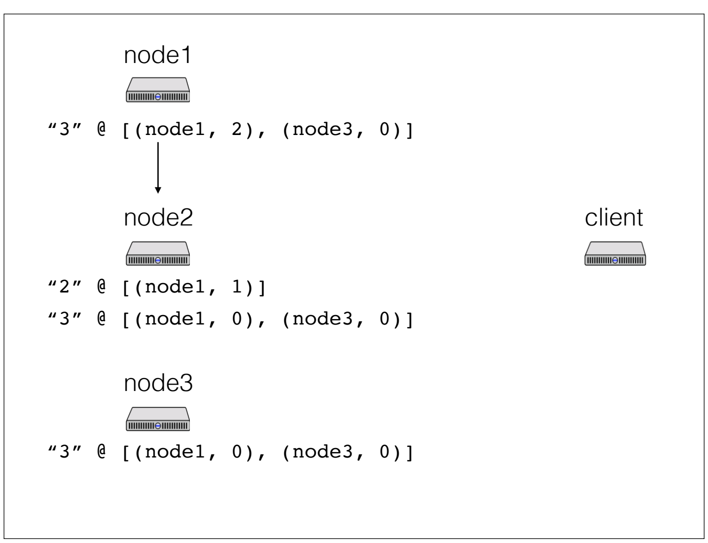

node1
"2" @ [(node1, 1)]
node2
client
"2" @ [(node1, 1)]
"3" @ [(node1, 0), (node3, 0)]
node3
"3" @ [(node1, 0), (node3, 0)]

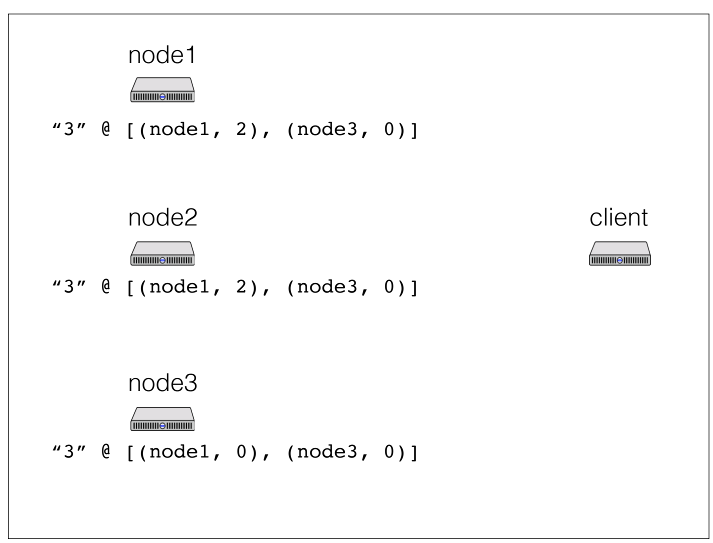

node1
“2” @ [(node1, 1)]
node2
client
“2” @ [(node1, 1)]
“3” @ [(node1, 0), (node3, 0)]
node3
“3” @ [(node1, 0), (node3, 0)]

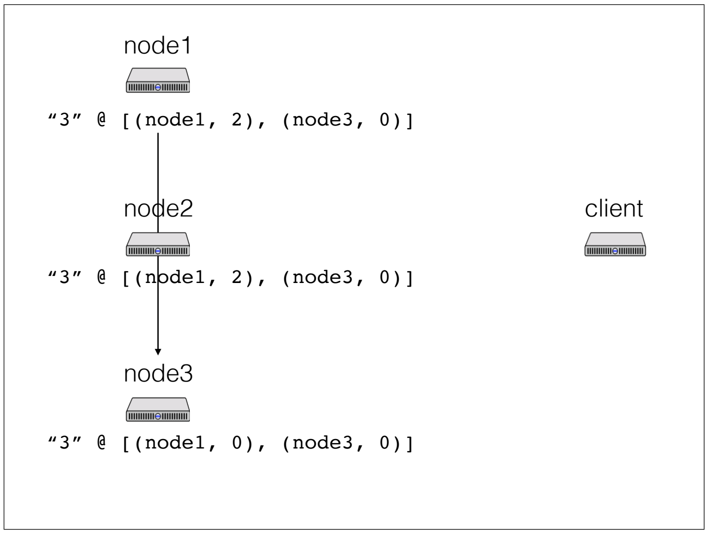

node1
“2” @ [(node1, 1)]
[“2”, “3”], [(node1, 1), (node3, 0)]
node2
client
“2” @ [(node1, 1)]
“3” @ [(node1, 0), (node3, 0)]
node3
“3” @ [(node1, 0), (node3, 0)]

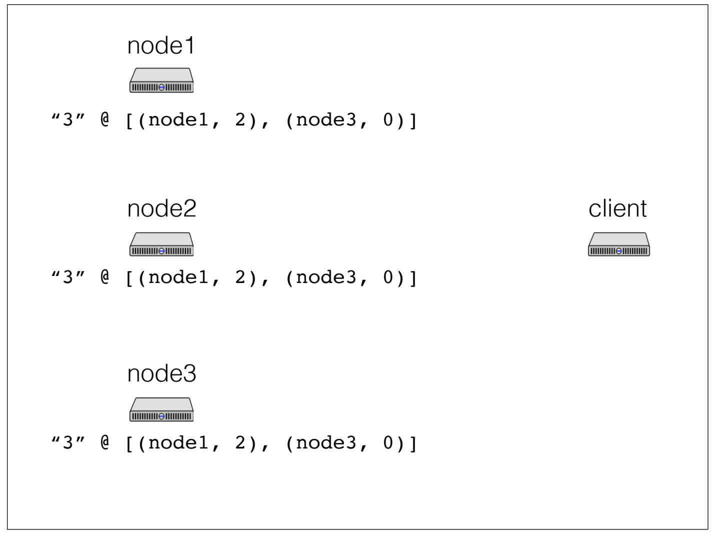

node1
“2” @ [(node1, 1)]
[“2”, “3”], [(node1, 1), (node3, 0)]
node2
client
“2” @ [(node1, 1)]
“3” @ [(node1, 0), (node3, 0)]
client must now run merge!
node3
“3” @ [(node1, 0), (node3, 0)]


node1
“2” @ [(node1, 1)]
put(“3”, [(node1, 1), (node3, 0)])
node2
client
“2” @ [(node1, 1)]
“3” @ [(node1, 0), (node3, 0)]
node3
“3” @ [(node1, 0), (node3, 0)]


node1
“3” @ [(node1, 2), (node3, 0)]
node2
client
“2” @ [(node1, 1)]
“3” @ [(node1, 0), (node3, 0)]
node3
“3” @ [(node1, 0), (node3, 0)]


node1
“3” @ [(node1, 2), (node3, 0)]
node2
client
“3” @ [(node1, 2), (node3, 0)]
node3
“3” @ [(node1, 0), (node3, 0)]





Where does each key live?

Goals:

- Balance load, even as servers join and leave
- Replicate across data centers
- Encourage put/get to see each other
- Avoid conflicting versions

Solution: consistent hashing

Recap: consistent hashing

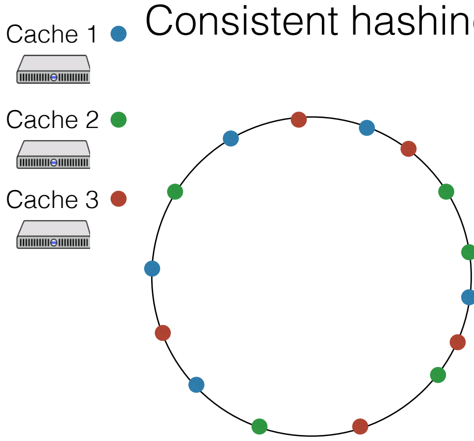
Node ids hashed to many pseudorandom points on a circle

Keys hashed onto circle, assigned to “next” node

Idea used widely:

- Developed for Akamai CDN
- Used in Chord distributed hash table
- Used in Dynamo distributed DB

Consistent hashing



Consistent hashing in Dynamo

Each key has a “preference list”—next nodes around the circle

- Skip duplicate virtual nodes
- Ensure list spans data centers

Slightly more complex:

- Dynamo ensures keys evenly distributed
- Nodes choose “tokens” (positions in ring) when joining the system
- Tokens used to route requests
- Each token = equal fraction of the keyspace

Replication in Dynamo

Three parameters: N, R, W

- N: number of nodes each key replicated on
- R: number of nodes participating in each read
- W: number of nodes participating in each write

Data replicated onto first N live nodes in pref list

- But respond to the client after contacting W

Reads see values from R nodes

Common config: (3, 2, 2)

Sloppy quorum

Never block waiting for unreachable nodes

- Try next node in list!

Want get to see most recent put (as often as possible)

Quorum: R + W > N

- Don't wait for all N
- R and W will (usually) overlap

Nodes ping each other

- Each has independent opinion of up/down

"Sloppy" quorum—nodes can disagree about which nodes are running

Replication in Dynamo

Coordinator (or client) sends each request (put or get) to first N reachable nodes in pref list

- Wait for R replies (for read) or W replies (for write)

Normal operation: gets see all recent versions

Failures/delays:

- Writes still complete quickly
- Reads eventually see

Ensuring eventual consistency

What if puts end up far away from first N?

- Could happen if some nodes temporarily unreachable
- Server remembers "hint" about proper location
- Once reachability restored, forwards data

Nodes periodically sync whole DB

- Fast comparisons using Merkle trees

Dynamo deployments

~100 nodes each

One for each service (parameters global)

How to extend to multiple apps?

Different apps use different (N, R, W)

- Pretty fast, pretty durable: (3, 2, 2)
- Many reads, few writes: (3, 1, 3) or (N, 1, N)
- (3, 3, 3)?
- (3, 1, 1)?

Dynamo results

Average *much* faster than 99.9%

- But, 99.9% acceptable

Inconsistencies rare in practice

- Allow inconsistency, but minimize it

Dynamo Revisited

Implemented as a library, not as a service

- Each service (eg shopping cart) instantiated a Dynamo instance

When an inconsistency happens:

- Is it a problem in Dynamo?
- Is it an intended side effect of Dynamo's design?

Every service runs its own ops => every service needs to be an expert at sloppy quorum

Dynamo DB

Replaced Dynamo the library with DynamoDB the service

DynamoDB: strictly consistent key value store

- validated with TLA and model checking
- eventually consistent as an option
- (afaik) no multikey transactions?

Dynamo is eventually consistent

Amazon is eventually strictly consistent!

Discussion

Why is symmetry valuable? Do seeds break it?

Dynamo and SOA

- What about malicious/buggy clients?

Issues with hot keys?

Transactions and strict consistency

- Why were transactions implemented at Google and not at Amazon?
- Do Amazon's programmers not want strict consistency?