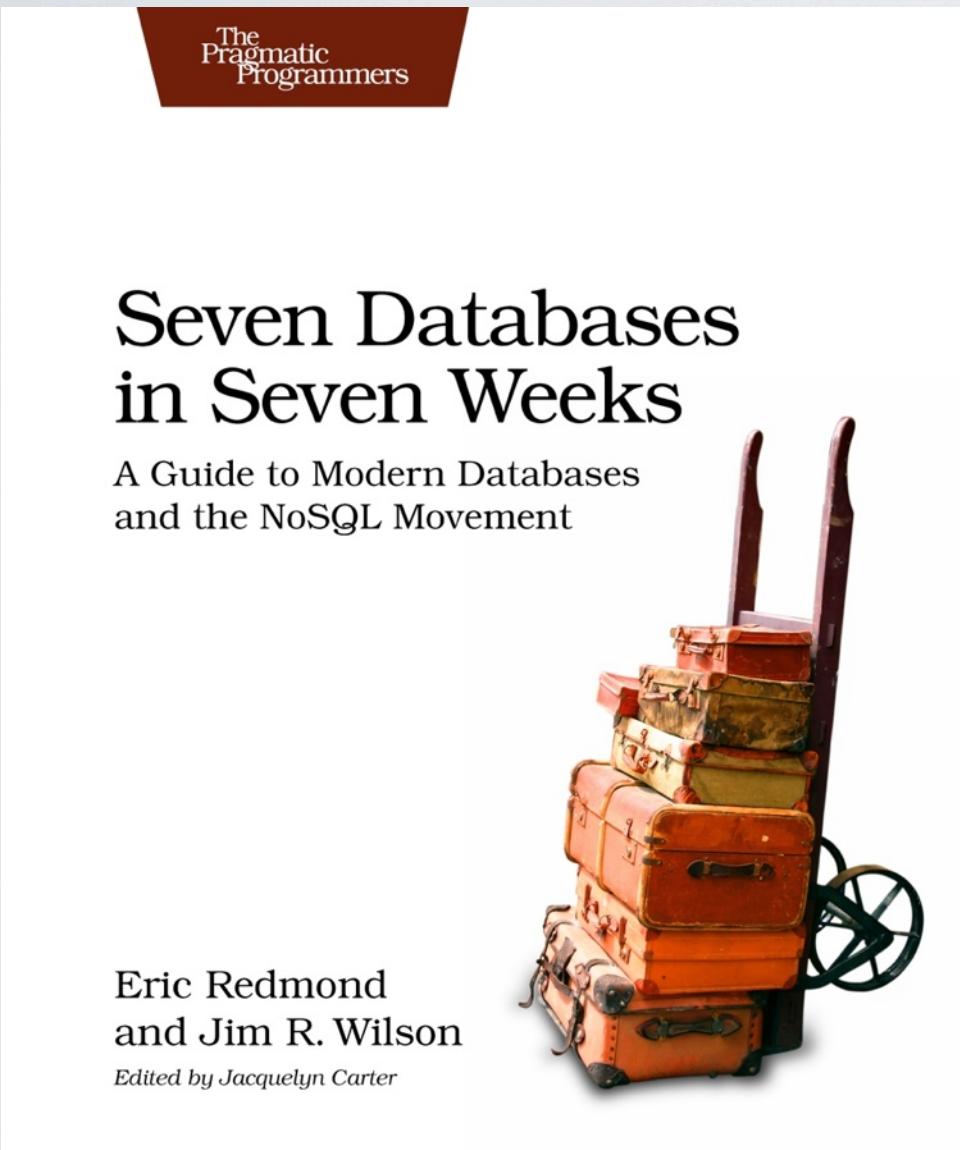
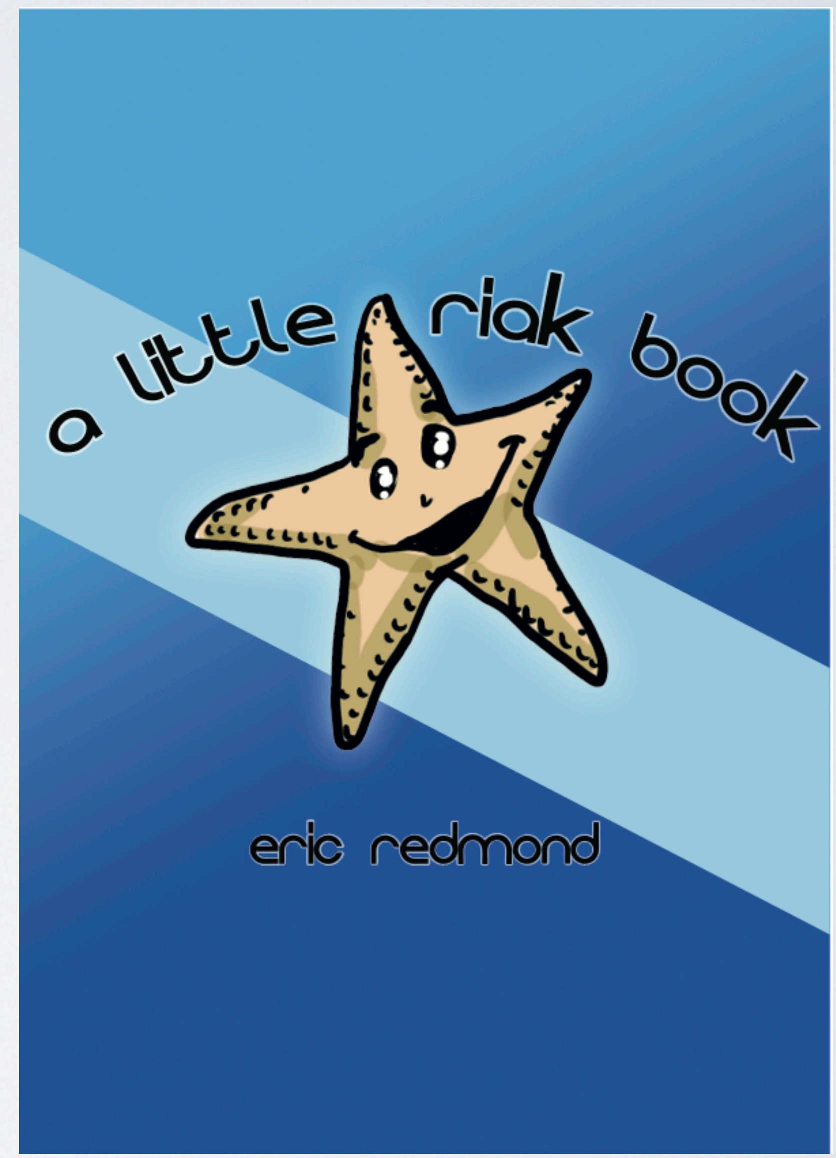
DISTRIBUTED SYSTEMS WITH RIAK

Eric Redmond
@coderoshi

basho



http://pragprog.com/book/rwdata



http://github.com/coderoshi/little_riak_book


WHY?

WHY RIAK?

Riak distributes data across **multiple nodes** to be

Scalable

Horizontal scaling allows for expansions of resources by adding/removing computers to/from the network.

Available

Composed of autonomous and interconnected computers that isolate successes and failures

DISTRIBUTED SYSTEM

A collection of autonomous **computers** running **processes** that exchange **messages** over a **network**, attempting to solve a **shared** problem.

CAP THEOREM

- **C**onsistent
- **A**vailable
- **P**artition-Tolerant

* http://codahale.com/you-cant-sacrifice-partition-tolerance

EVENTUALLY CONSISTENT

Perfect is the enemy of good

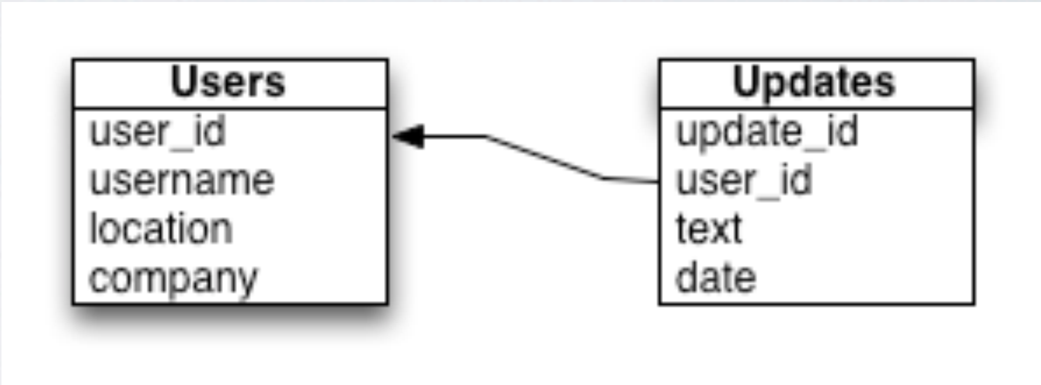
- How Eventual?
- How Consistent?

BY DATA STRUCTURE

- Relational
- Graph
- Document
- Column Family
- Key/Value

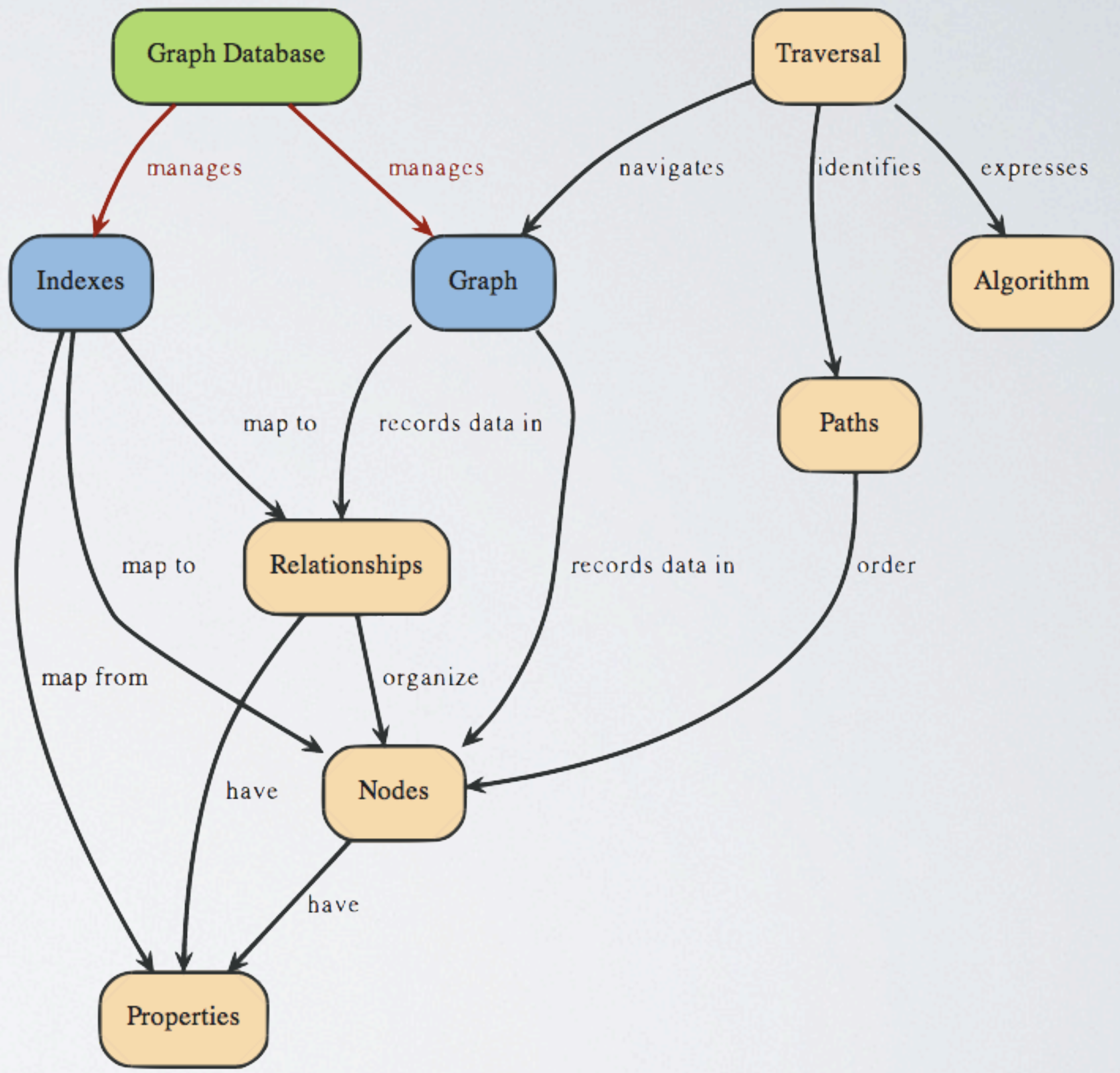
RELATIONAL

- We all know an love it
- 40+ years of research and experience
- Great for **query flexibility** (not so much **assignment flexibility**)
- Distribution is an after-market feature (usually)
- MySQL, PostgreSQL, Oracle



GRAPH

- Store values as nodes in a graph
- Fast for very complex relationships
- "If you can whiteboard it, you can graph it"
- Neo4j , HyperGraphDB, InfiniteGraph



* http://docs.neo4j.org/chunked/milestone/what-is-a-graphdb.html

DOCUMENT

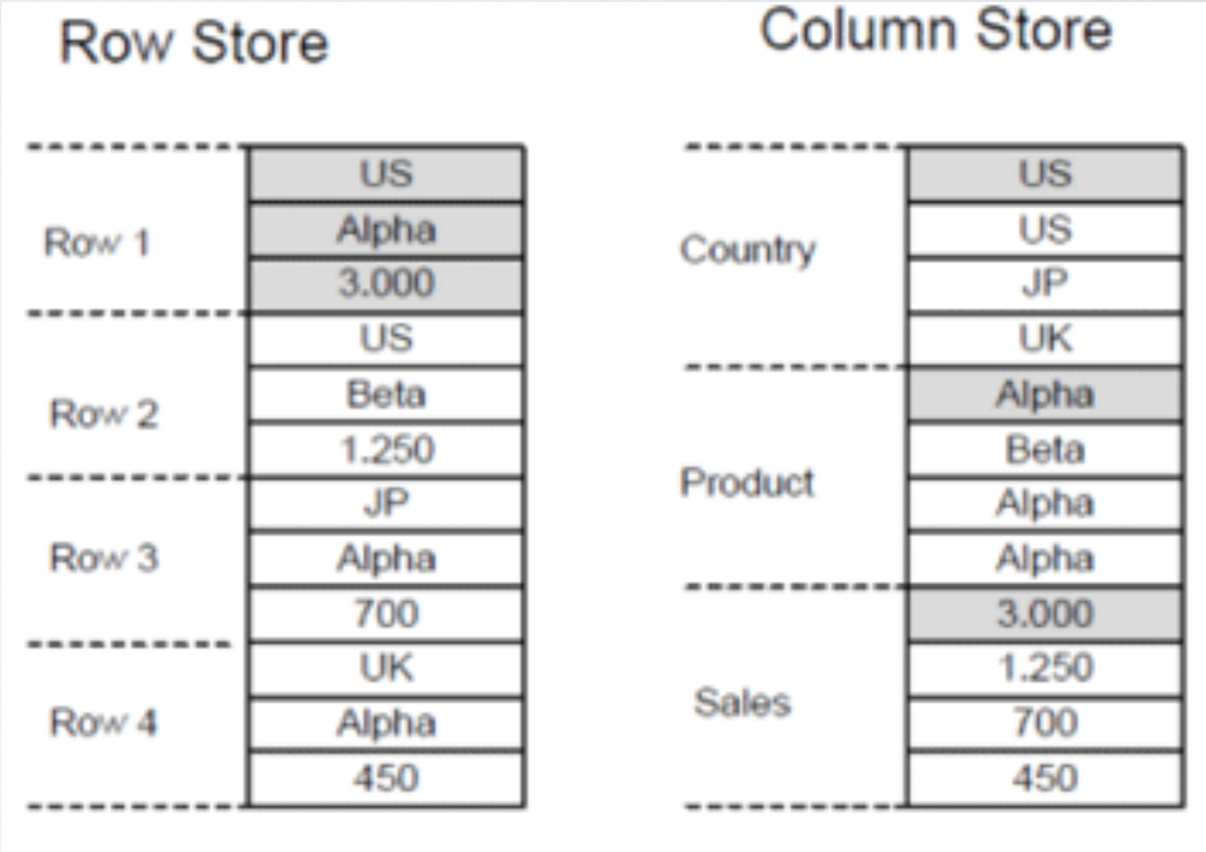
- Key based, but schemaless data
- MongoDB, CouchDB, Couchbase

```
{
    "_id" : "fred",
    "items" : [
        {
            "id" : "slingshot",
            "type" : "weapon",
            "damage" : 23,
            "ranged" : true
        },
        {
            "id" : "sword",
            "type" : "weapon",
            "damage" : 50,
            "ranged" : false
        }
    ]
}
```

* http://stackoverflow.com/questions/12536644/updating-incrementing-a-value-within-a-sub-document-in-mongodb

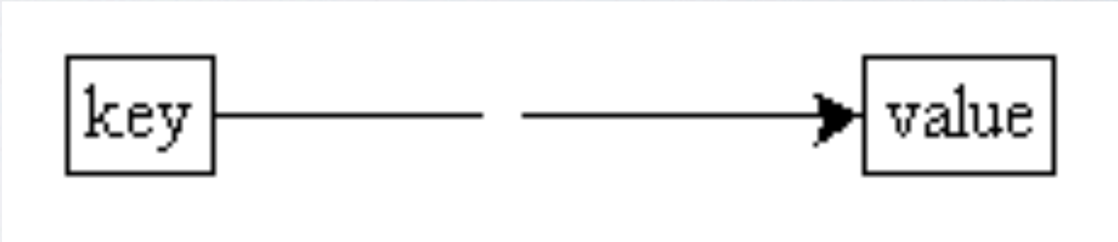
COLUMN FAMILY

- Column-based, made to scale out
- Easy to manage columns of data
- Great for time series data
- HBase, Cassandra, Hypertable



* http://www.cubrid.org/blog/dev-platform/platforms-for-big-data/

KEY/VALUE STORE

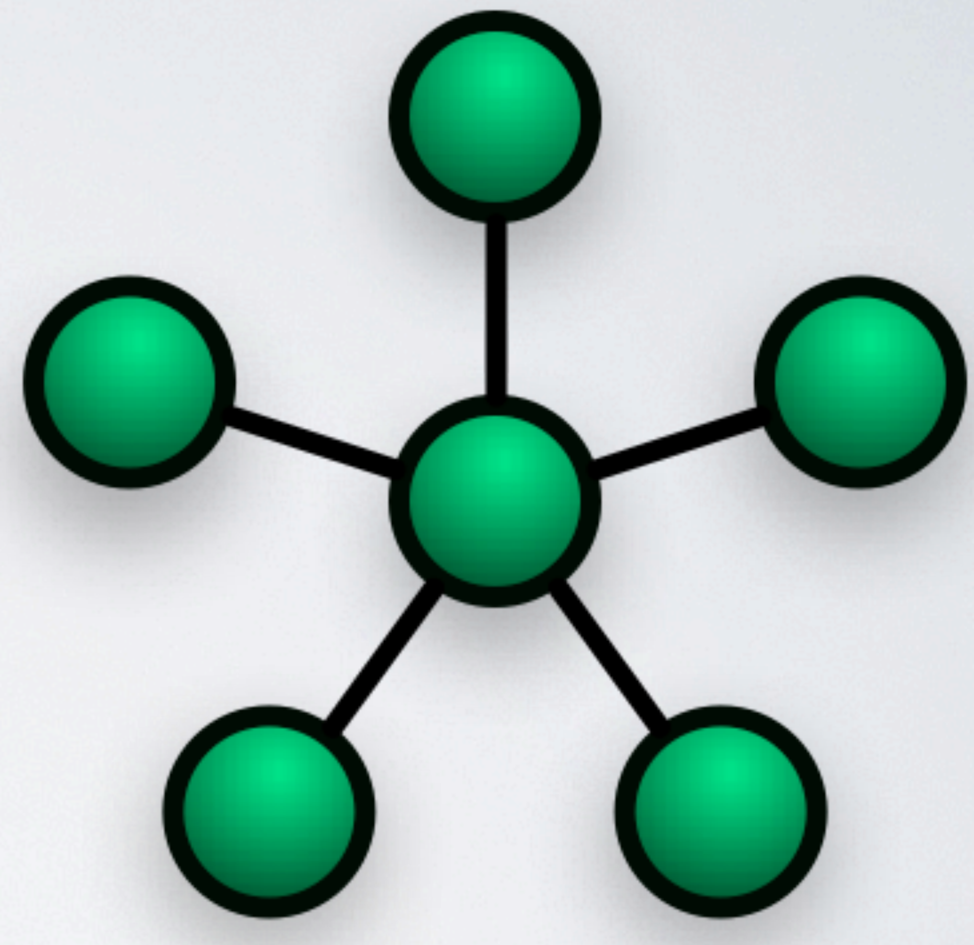


- Fast, flexible, easily scalable
- Can by annoying to query complex datastructures
- You can can always chain lookups
- Redis, Memcached, Riak

BY LOGICAL TOPOLOGY

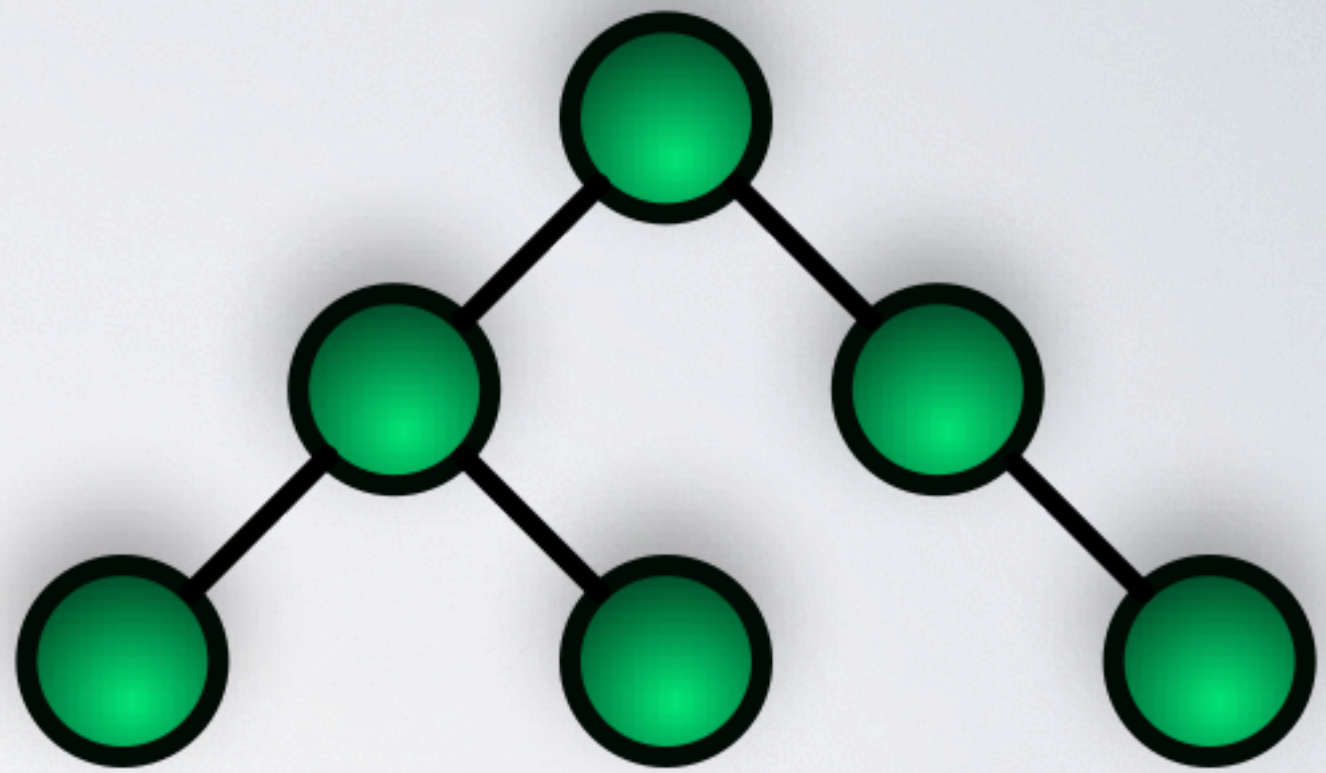
- Star: MySQL Cluster
- Tree: MongoDB
- Mesh: Riak, Cassandra

STAR TOPOLOGY



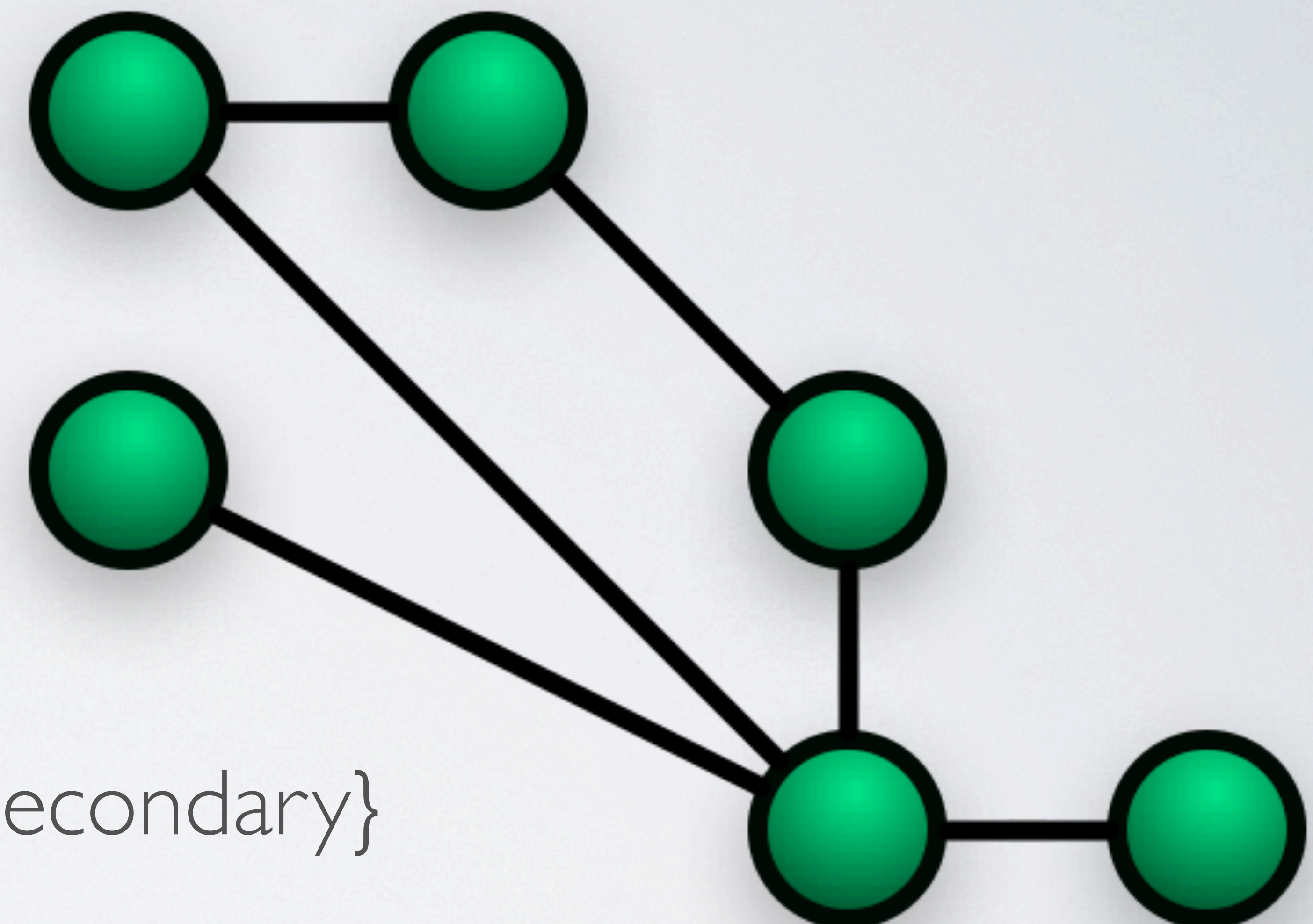
- Master, Slaves
- mysqld, ndb cluster

TREE TOPOLOGY



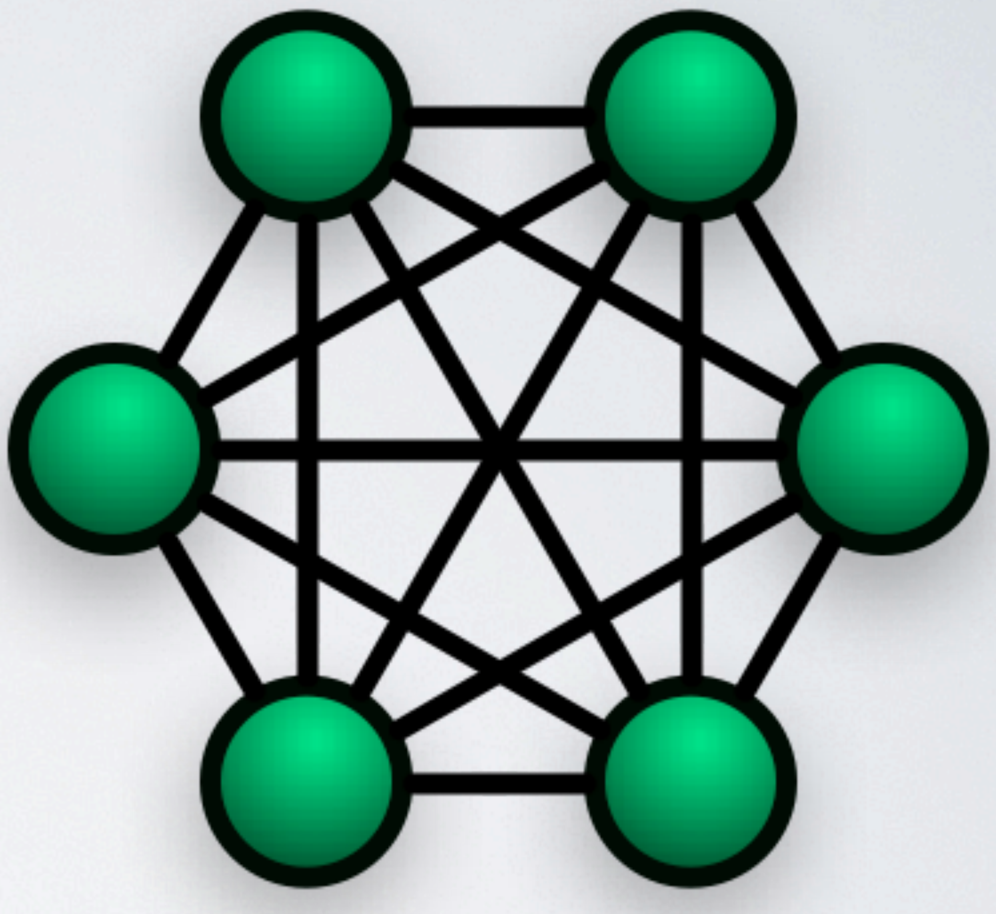
- router, master, slaves
- mongos, replset{primary, secondary}

(PARTIAL) MESH TOPOLOGY



- routers, master, slaves
- mongoses, replset{primary, secondary}

MESH TOPOLOGY



- nodes

PROBLEMS

- message delay
- time to execute
- clock drift

SOLUTIONS

- sloppy quorums
- supervisor process
- vector clock

DISTRIBUTED PATTERNS

https://github.com/coderoshi/dds

- DHT
- Message patterns
- Vector Clocks
- Merkle Tree
- Mapreduce

DISTRIBUTED HASH TABLE

- Consistent hash
- Distributes data evenly
- Minimal disruption when nodes are added/removed

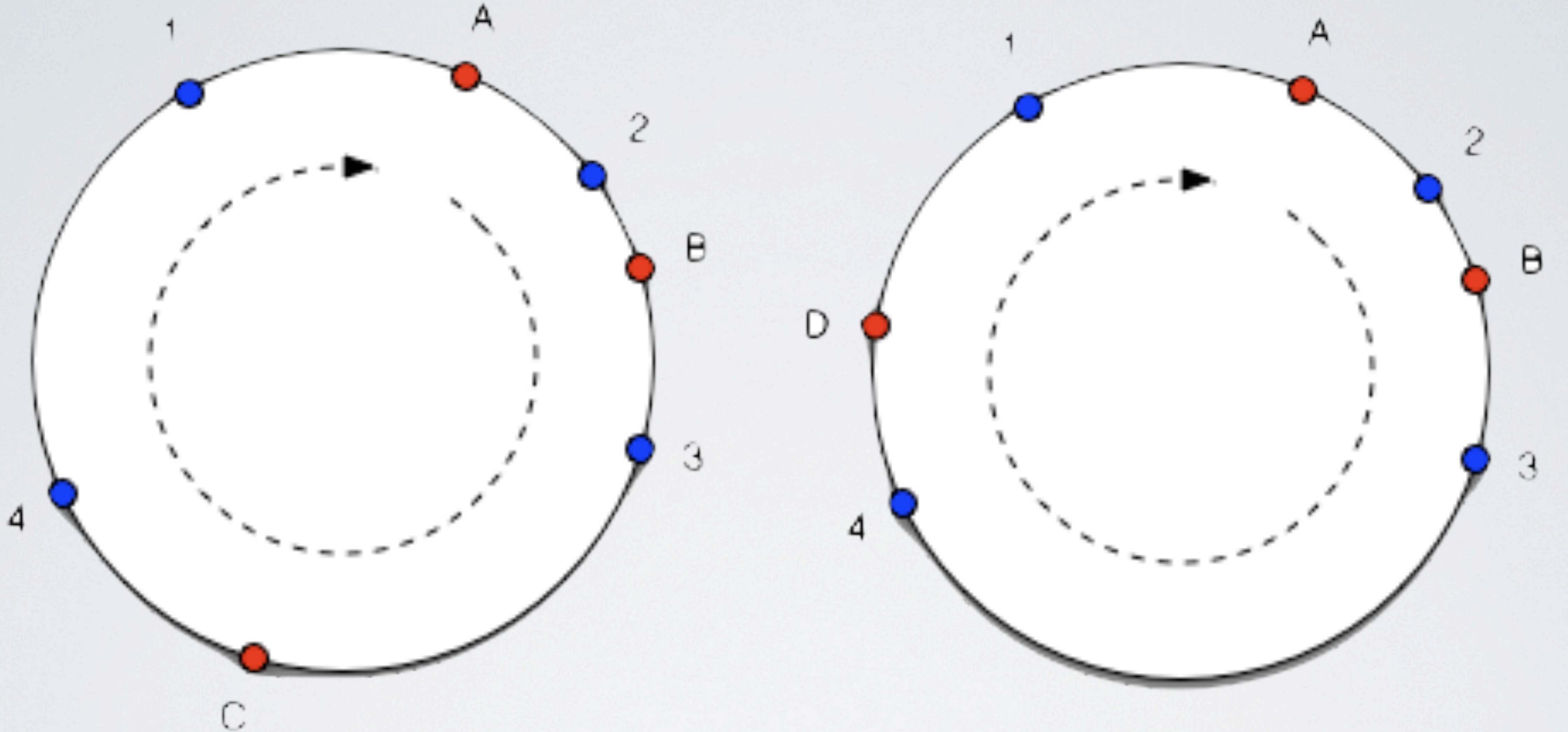
```ruby
class NaiveHash
  def initialize(nodes=[], spread=(1<<20))
    @nodes = nodes
    @spread = spread
    @array = Array.new(@nodes.length * @spread)
  end

  def hash(key)
    Digest::SHA1.hexdigest(key.to_s).hex
  end

  def add(node)
    @nodes << node
  end

  def node(key)
    length = @nodes.length * @spread
    @nodes[ (hash(key) % length) / @spread ]
  end
end
```

```ruby
class NaiveHash
  def initialize(nodes=[], spread=(1<<20))
    @nodes = nodes
    @spread = spread
    @array = Array.new(@nodes.length * @spread)
  end

  def hash(key)
    Digest::SHA1.hexdigest(key.to_s).hex
  end

  def add(node)
    @nodes << node
  end

  def node(key)
    length = @nodes.length * @spread
    @nodes[ (hash(key) % length) / @spread ]
  end
end
```

- nodes = [“A”, “B”, “C”]
- spread = 4
- array = [[0,1,2,3],[4,5,6,7],[8,9,10,11]]

```ruby
h = NaiveHash.new(["A", "B", "C"])

puts h.node("foo")   # => C
h.add("D")
puts h.node("foo")   # => D
```

```ruby
h = NaiveHash.new(("A".."J").to_a)
elements = 100000
tracknodes = Array.new(elements)

elements.times do |i|
  tracknodes[i] = h.node(i)
end

h.add("K")

misses = 0
elements.times do |i|
  misses += 1 if tracknodes[i] != h.node(i)
end

puts "misses: #{(misses.to_f/elements) * 100}%"

# misses: 90.922%
```


1
A
2
B
3
4
C
1
A
2
B
D
3
4

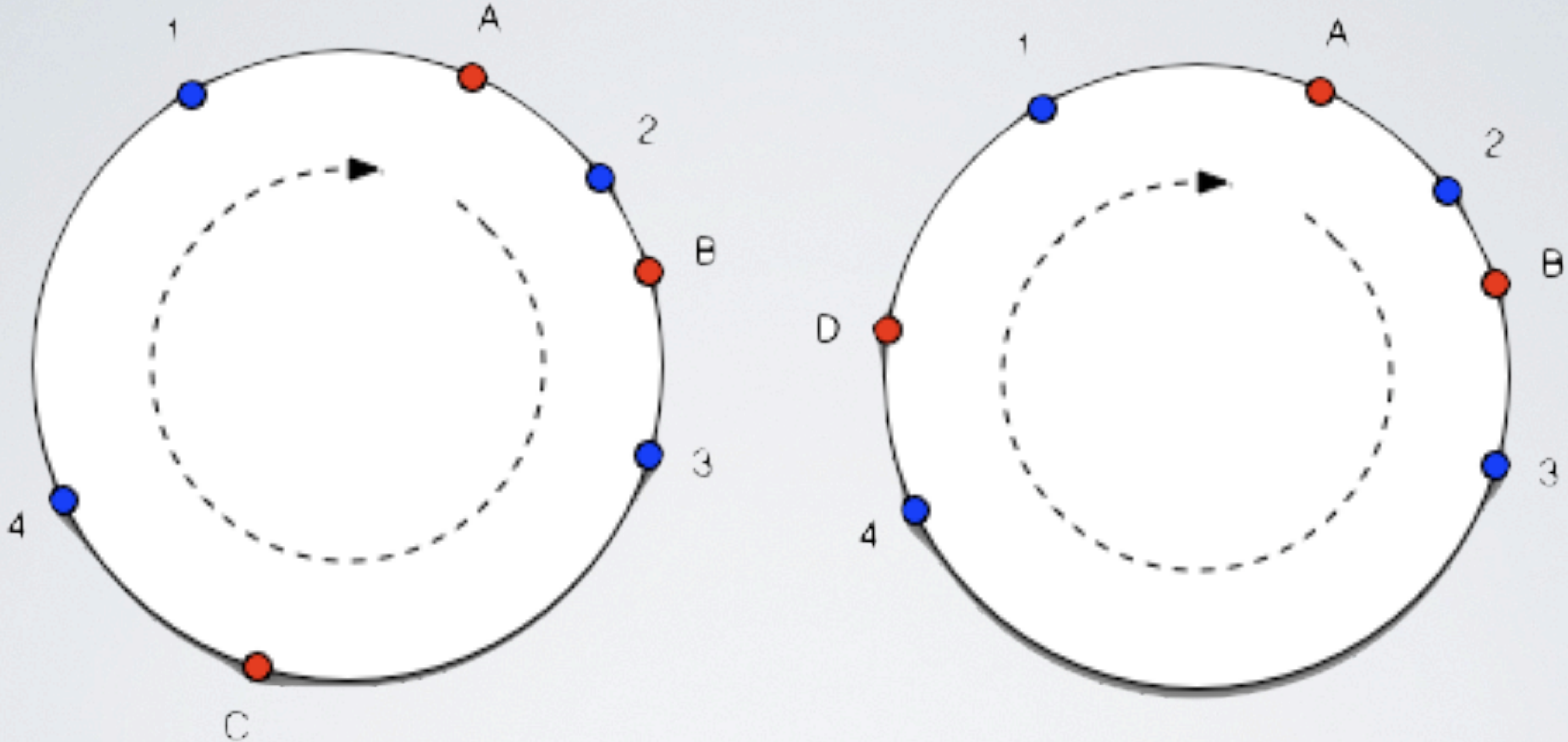
```ruby
class ConsistentHash
  def initialize(nodes=[])
    @ring = {}
    @nodesort = []
    for node in nodes
      add(node)
    end
  end

  def hash(key)
    Digest::SHA1.hexdigest(key.to_s).hex
  end

  # ...
```

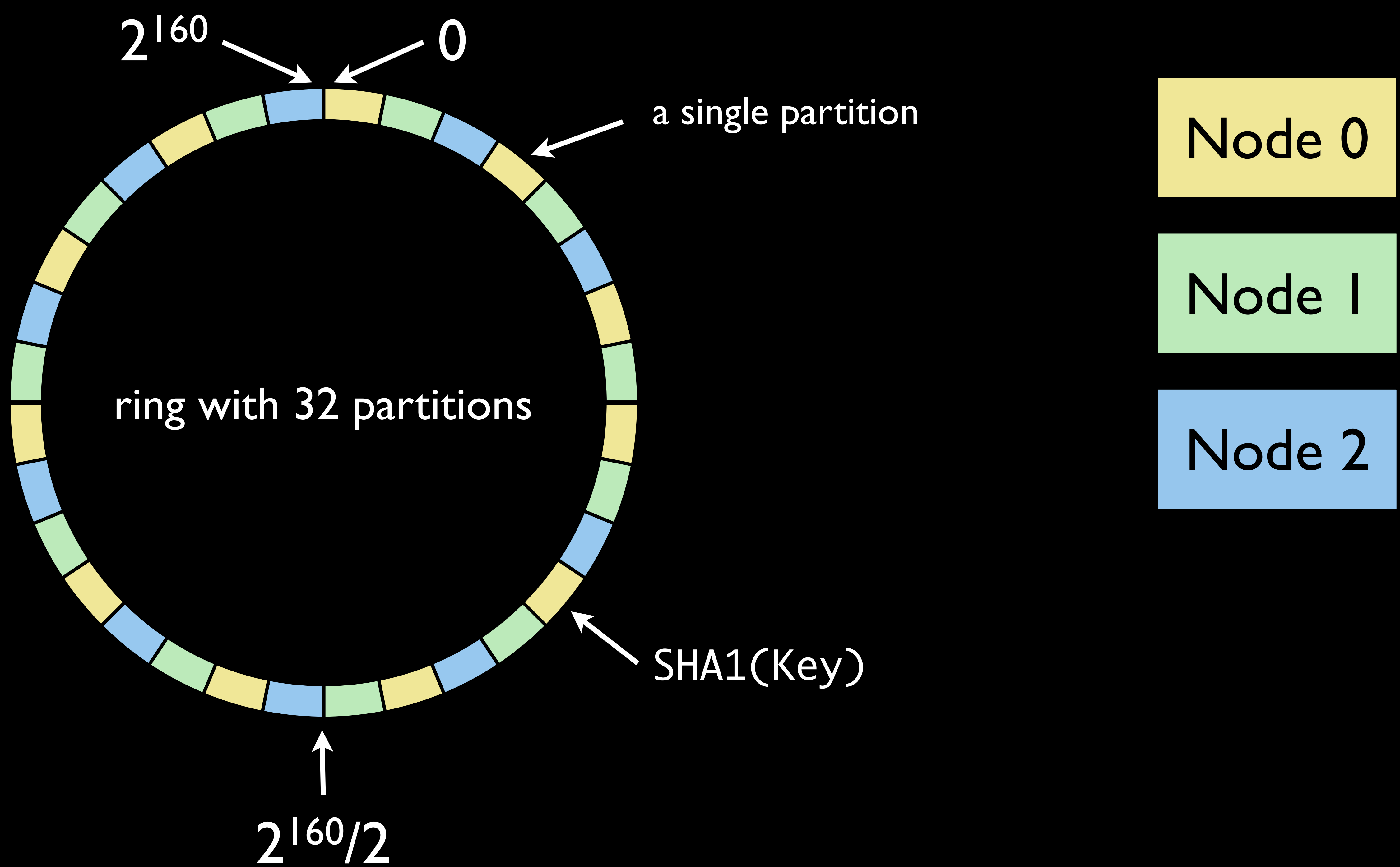
```ruby
  # ...

  def add(node)
    key = hash(node.to_s)
    @ring[key] = node
    @nodesort.push(key)
    @nodesort.sort!
  end

  def node(keystr)
    return nil if @ring.empty?
    key = hash(keystr)
    @nodesort.length.times do |i|
      node = @nodesort[i]
      return @ring[ node ] if key <= node
    end
    @ring[ @nodesort[0] ]
  end
end
```

```ruby
class ConsistentHash
  # ...

  def node(keystr)
    return nil if @ring.empty?
    key = hash(keystr)
    @nodesort.length.times do |i|
      node = @nodesort[i]
      return @ring[ node ] if key <= node
    end
    @ring[ @nodesort[0] ]
  end
end
```

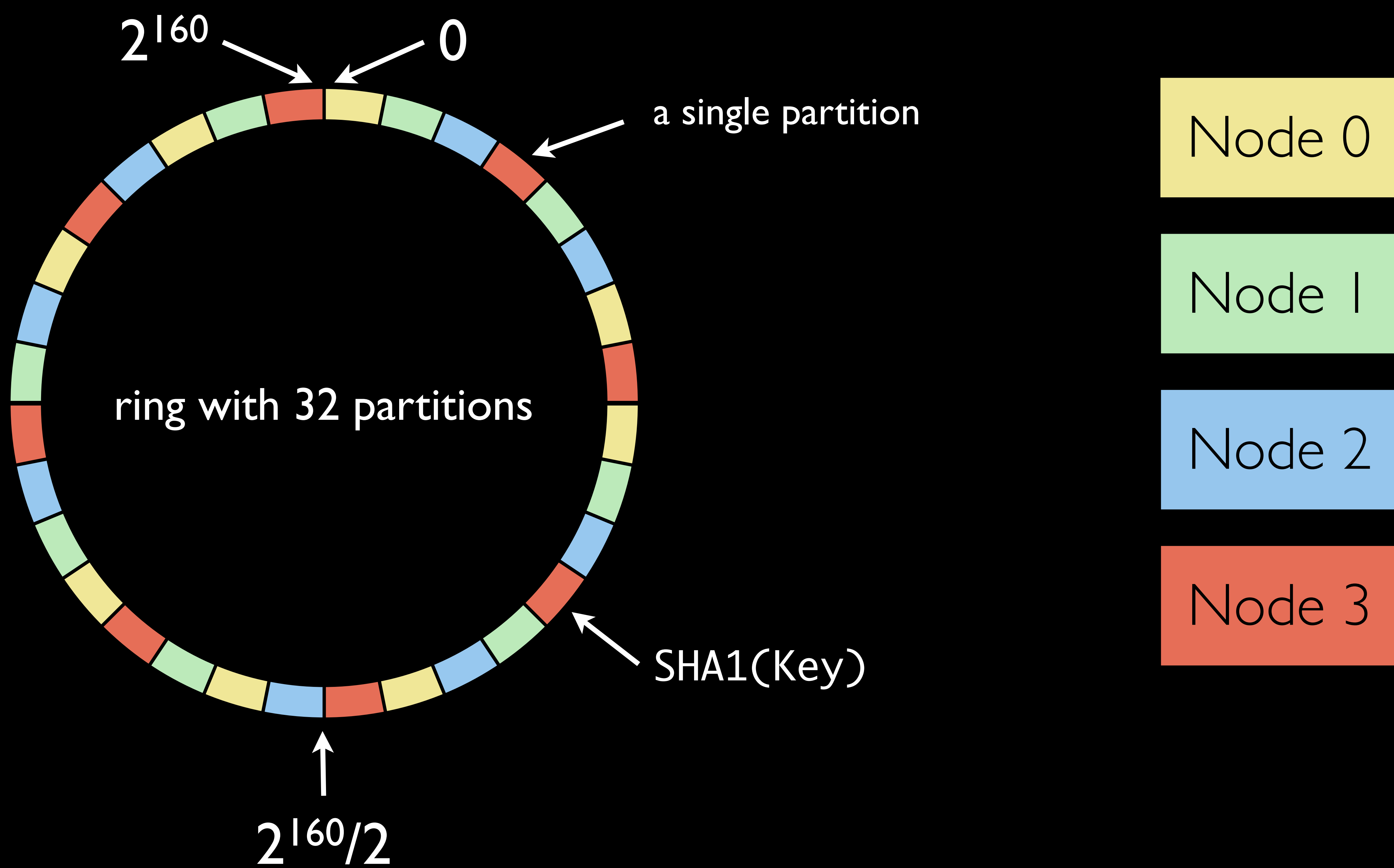
```ruby
h = ConsistentHash.new(["A", "B", "C"])

puts h.node("foo")   # => A
h.add("D")
puts h.node("foo")   # => A
```

```ruby
h = ConsistentHash.new(("A".."J").to_a)
elements = 100000
tracknodes = Array.new(elements)

elements.times do |i|
  tracknodes[i] = h.node(i)
end

h.add("K")

misses = 0
elements.times do |i|
  misses += 1 if tracknodes[i] != h.node(i)
end

puts "misses: #{(misses.to_f/elements) * 100}%"

# misses: 7.343%
```


1
A
2
B
3
4
C
1
A
2
B
D
3
4



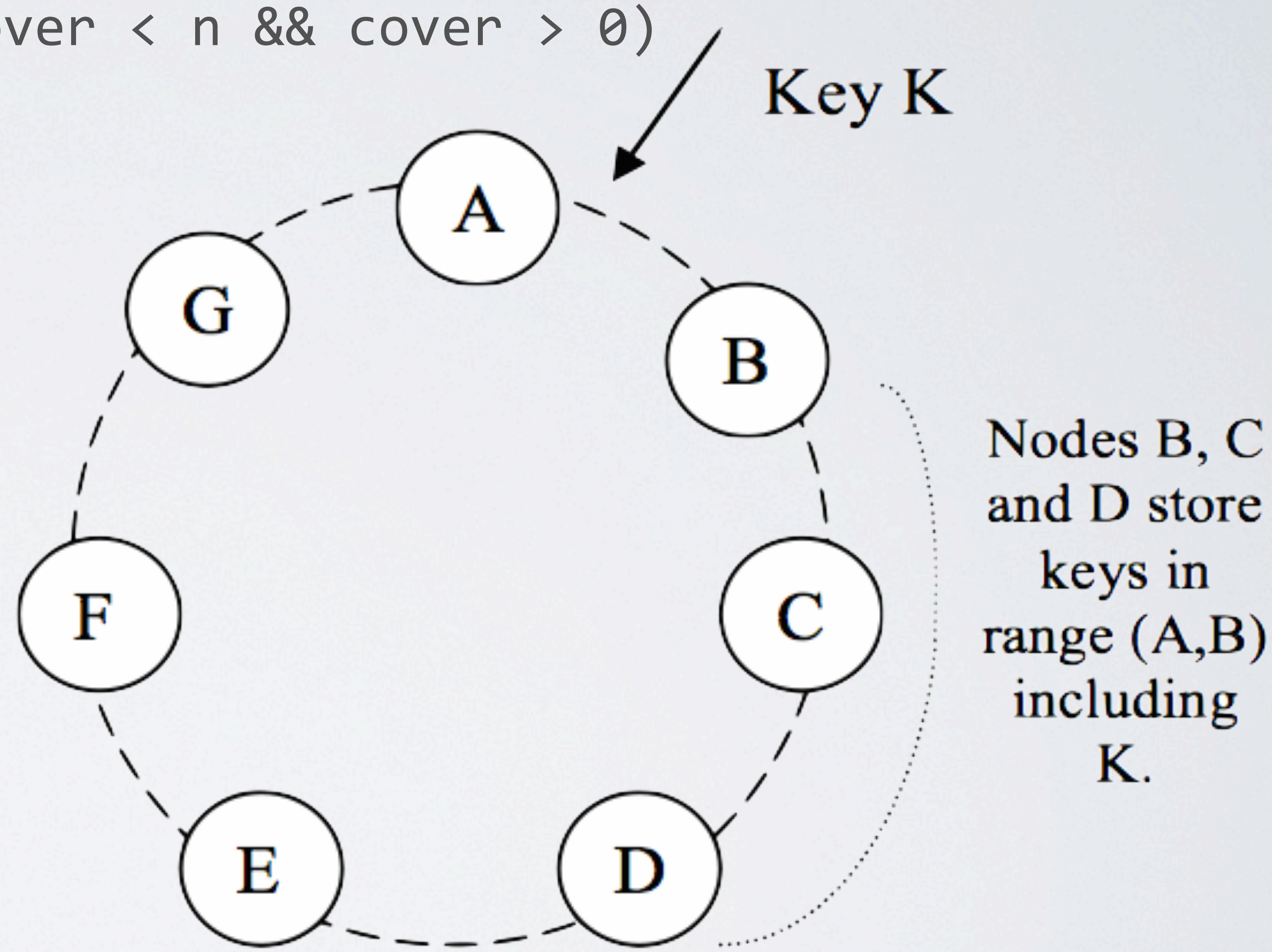
2^160
0
a single partition
ring with 32 partitions
SHA1(Key)
2^160/2
Node 0
Node 1
Node 2


2160
0
a single partition
ring with 32 partitions
SHA1(Key)
2160/2
Node 0
Node 1
Node 2
Node 3

```ruby
SHA1BITS = 160
class PartitionedConsistentHash
  def initialize(nodes=[], partitions=32)
    @partitions = partitions
    @nodes, @ring = nodes.clone.sort, {}
    @power = SHA1BITS - Math.log2(partitions).to_i
    @partitions.times do |i|
      @ring[range(i)] = @nodes[0]
      @nodes << @nodes.shift
    end
    @nodes.sort!
  end

  def range(partition)
    (partition*(2**@power)..(partition+1)*(2**@power)-1)
  end

  def hash(key)
    Digest::SHA1.hexdigest(key.to_s).hex
  end

  def add(node)
    @nodes << node
    partition_pow = Math.log2(@partitions)
    pow = SHA1BITS - partition_pow.to_i
    (0..@partitions).step(@nodes.length) do |i|
      @ring[range(i, pow)] = node
    end
  end

  def node(keystr)
    return nil if @ring.empty?
    key = hash(keystr)
    @ring.each do |range, node|
      return node if range.cover?(key)
    end
  end
end
```

```ruby
h = PartitionedConsistentHash.new(("A".."C").to_a)
puts h.node("foo")
h.add("D")
puts h.node("foo")


h = PartitionedConsistentHash.new(("A".."J").to_a)
elements = 100000
nodes = Array.new(elements)
elements.times do |i|
  nodes[i] = h.node(i)
end
puts "add K"
h.add("K")
misses = 0
elements.times do |i|
  misses += 1 if nodes[i] != h.node(i)
end
puts "misses: #{(misses.to_f/elements) * 100}%\n"

# misses: 9.473%
```

```ruby
# return a list of successive nodes
# that can also hold this value
def pref_list(keystr, n=3)
  list = []
  key = hash(keystr)
  cover = n
  @ring.each do |range, node|
    if range.cover?(key) || (cover < n && cover > 0)
      list << node
      cover -= 1
    end
  end
  return list
end
```




```ruby
puts h.node("foo")   # "B"

p h.pref_list("foo", 3)   # ["B", "C", "D"]
```

```
class NodeObject
  attr :value
  def initialize(value)
    @value = value
  end

  def to_s
    {:value=>value}.to_json
  end

  # takes a string and creates a NodeObject
  def self.deserialize(serialized)
    data = JSON.parse( serialized )
    NodeObject.new( data['value'] )
  end
end
```

```ruby
class Node

  def initialize(name, nodes=[], partitions=32)
    @name = name
    @data = {}
    @ring = PartitionedConsistentHash.new(nodes, partitions)
  end

  def put(key, value)
    if @name == @ring.node(key)
      puts "put #{key} #{value}"
      @data[ @ring.hash(key) ] = [NodeObject.new(value)]
    end
  end

  def get(key)
    if @name == @ring.node(key)
      puts "get #{key}"
      @data[@ring.hash(key)]
    end
  end

end
```

```ruby
class Node

  def initialize(name, nodes=[], partitions=32)
    @name = name
    @data = {}
    @ring = PartitionedConsistentHash.new(nodes, partitions)
  end

  def put(key, value)
    if @name == @ring.node(key)
      puts "put #{key} #{value}"
      @data[ @ring.hash(key) ] = [NodeObject.new(value)]
    end
  end

  def get(key)
    if @name == @ring.node(key)
      puts "get #{key}"
      @data[@ring.hash(key)]
    end
  end

end
```

```ruby
nodeA = Node.new( 'A', ['A', 'B', 'C'] )
nodeB = Node.new( 'B', ['A', 'B', 'C'] )
nodeC = Node.new( 'C', ['A', 'B', 'C'] )


nodeA.put( "foo", "bar" )
p nodeA.get( "foo" )   # nil


nodeB.put( "foo", "bar" )
p nodeB.get( "foo" )   # "bar"


nodeC.put( "foo", "bar" )
p nodeC.get( "foo" )   # nil
```

MESSAGING PATTERNS

- Request/Reply
- Publish/Subscribe
- Pipeline

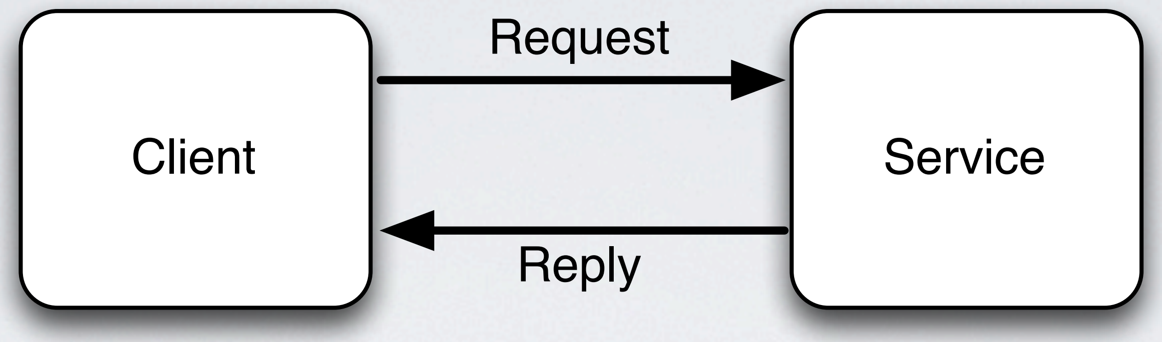
MESSAGING PATTERNS

- Request/Reply (Query nodes, forward requests)
- Publish/Subscribe (Keep hashes in sync across nodes)
- Pipeline (Load balance work across the nodes)

0MQ

- Higher level than sockets, lower level than middlewares
- Transport agnostic (Mem, IPC, TCP, PGM, etc)
- Message-oriented, not stream or datagram

REQUEST/REPLY



Request	
Direction	*Bidirectional*
Send/receive pattern	*send, receive, send, receive...*
Incoming route strategy	*Last peer*
Outgoing route strategy	*Round-robin*

Reply	
Direction	*Bidirectional*
Send/receive pattern	*receive, send, receive, send...*
Incoming route strategy	*Fair-queued*
Outgoing route strategy	*Last peer*

```ruby
# Helper module to stage multiple
# threads then join them at once
module Threads
  def thread()
    @threads = [] unless defined?(@threads)
    @threads << Thread.new do
      begin
        yield   #execute code in the block
      rescue => e
        puts e.backtrace.join("\n")
      end
    end
  end

  def join_threads()
    @threads.each{|t| t.join }
  end
end
```

```ruby
require 'zmq'
require './threads'
include Threads

thread do   # server
  ctx = ZMQ::Context.new
  rep = ctx.socket( ZMQ::REP )
  rep.bind( "tcp://127.0.0.1:2200" )
  while line = rep.recv
    msg, payload = line.split(' ', 2)
    if msg == "put"
      rep.send( "Called 'PUT' with #{payload}" )
    end
  end
end

thread do   # client
  ctx = ZMQ::Context.new
  req = ctx.socket( ZMQ::REQ )
  req.connect( "tcp://127.0.0.1:2200" )
  puts req.send("put foo bar") && req.recv
  puts req.send( "put foo2 bar2" ) && req.recv
end

join_threads  # start server and client
```

```ruby
require 'zmq'
require './threads'
include Threads

thread do   # server
  ctx = ZMQ::Context.new
  rep = ctx.socket( ZMQ::REP )
  rep.bind( "tcp://127.0.0.1:2200" )
  while line = rep.recv
    msg, payload = line.split(' ', 2)
    if msg == "put"
      rep.send( "Called 'PUT' with #{payload}" )
    end
  end
end

thread do   # client
  ctx = ZMQ::Context.new
  req = ctx.socket( ZMQ::REQ )
  req.connect( "tcp://127.0.0.1:2200" )
  puts req.send("put foo bar") && req.recv
  puts req.send( "put foo2 bar2" ) && req.recv
end

join_threads  # start server and client
```

```ruby
module ReplyService
  # helper function to create a req/res service,
  # and relay message to corresponding methods
  def service(port)
    thread do
      ctx = ZMQ::Context.new
      rep = ctx.socket( ZMQ::REP )
      rep.bind( "tcp://127.0.0.1:#{port}" )
      while line = rep.recv
        msg, payload = line.split(' ', 2)
        send( msg.to_sym, rep, payload )
      end
    end
  end

  def method_missing(method, *args, &block)
    socket, payload = args
    payload.send( "bad message" ) if payload
  end
end
```

```
/* A.json */
{
  "name" : "A",
  "port" : 2200
}
```

```ruby
class Node
  include Threads
  include ReplyService

  def config(name)
    @configs[name] ||= JSON::load(File.read("#{name}.json"))
  end

  def start()
    service( config(@name)["port"] )
    puts "#{@name} started"
    join_threads()
  end

  def remote_call(remote_name, message)
    puts "#{remote_name} <= #{message}"
    remote_port = config(remote_name)["port"]

    ctx = ZMQ::Context.new
    req = ctx.socket( ZMQ::REQ )
    req.connect( "tcp://127.0.0.1:#{remote_port}" )
    resp = req.send(message) && req.recv
    req.close
    resp
  end

  # ...
```

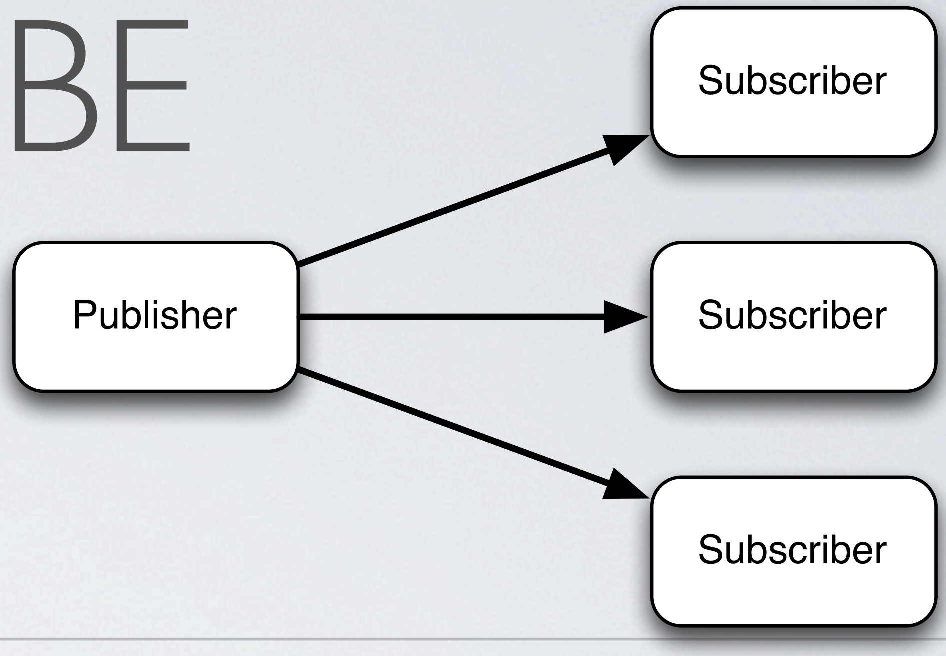
```ruby
# ...

def put(socket, payload)
  key, value = payload.split(' ', 2)
  socket.send( do_put(key, value).to_s )
end

def do_put(key, value)
  node = @ring.node(key)
  if node == @name
    puts "put #{key} #{value}"
    @data[@ring.hash(key)] = [NodeObject.new(value)]
  else
    remote_call(node, "put #{key} #{value}")
  end
end
```

```ruby
# start a Node as a Server
name = ARGV.first
node = Node.new(name, ['A','B','C'])
node.start()
```

```
$ ruby node.rb A
$ ruby node.rb B
$ ruby node.rb C
```

```ruby
# connect with a client
require 'zmq'

ctx = ZMQ::Context.new
req = ctx.socket(ZMQ::REQ)
req.connect( "tcp://127.0.0.1:2200" )

puts "Inserting Values"
1000.times do |i|
  req.send( "put key#{i} value#{i}" ) && req.recv
end

puts "Getting Values"
1000.times do |i|
  puts req.send( "get key#{i}" ) && req.recv
end

req.close
```

PUBLISH/SUBSCRIBE

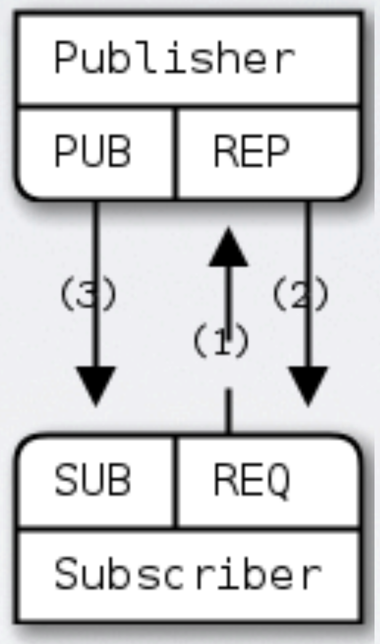


Publish	
Direction	*Unidirectional*
Send/receive pattern	*Send only*
Incoming route strategy	*N/A*
Outgoing route strategy	*Fan out*

Subscribe	
Direction	*Unidirectional*
Send/receive pattern	*Receive only*
Incoming route strategy	*Fair-queued*
Outgoing route strategy	*N/A*

```ruby
class Node
  # ...
  def coordinate_cluster(pub_port, rep_port)
    thread do
      ctx = ZMQ::Context.new
      pub = ctx.socket( ZMQ::PUB )
      pub.bind( "tcp://*:#{pub_port}" )
      rep = ctx.socket( ZMQ::REP )
      rep.bind( "tcp://*:#{rep_port}" )

      while line = rep.recv
        msg, node = line.split(' ', 2)
        nodes = @ring.nodes
        case msg
        when 'join'
          nodes = (nodes << node).uniq.sort
        when 'down'
          nodes -= [node]
        end
        @ring.cluster(nodes)

        pub.send( "ring " + nodes.join(','))
        rep.send( "true" )
      end
    end
  end
```

```ruby
class Node
  # ...
  def track_cluster(sub_port)
    thread do
      ctx = ZMQ::Context.new
      sub = ctx.socket( ZMQ::SUB )
      sub.connect( "tcp://127.0.0.1:#{sub_port}" )
      sub.setsockopt( ZMQ::SUBSCRIBE, "ring" )

      while line = sub.recv
        _, nodes = line.split(' ', 2)
        nodes = nodes.split(',').map{|x| x.strip}
        @ring.cluster( nodes )
        puts "ring changed: #{nodes.inspect}"
      end
    end
  end
```


Publisher
PUB
REP
(3)
(1)
(2)
SUB
REQ
Subscriber

```ruby
class Node
  # ...
  def coordinate_cluster(pub_port, rep_port)
    thread do
      ctx = ZMQ::Context.new
      pub = ctx.socket( ZMQ::PUB )
      pub.bind( "tcp://*:#{pub_port}" )
      rep = ctx.socket( ZMQ::REP )
      rep.bind( "tcp://*:#{rep_port}" )

      while line = rep.recv
        msg, node = line.split(' ', 2)
        nodes = @ring.nodes
        case msg
        when 'join'
          nodes = (nodes << node).uniq.sort
        when 'down'
          nodes -= [node]
        end
        @ring.cluster(nodes)

        pub.send( "ring " + nodes.join(','))
        rep.send( "true" )
      end
    end
  end
```

```ruby
class Node
  # ...
  def start(leader)
    coord_reqres = config(@name)["coord_req"]
    coord_pubsub = config(@name)["coord_pub"]

    track_cluster( coord_pubsub )
    coordinate_cluster( coord_pubsub, coord_reqres ) if leader
    inform_coordinator( "join", coord_reqres ) unless leader

    service( config(@name)["port"] )
    join_threads()
  end

  def close
    inform_coordinator( "down", config(@name)["coord_req"] )
    exit!
  end

  def inform_coordinator(action, req_port)
    ctx = ZMQ::Context.new
    req = ctx.socket(ZMQ::REQ)
    req.connect( "tcp://127.0.0.1:#{req_port}" )
    req.send( "#{action} #{@name}" ) && req.recv
    req.close
  end
end
```

WHAT WE'VE DONE SO FAR

https://github.com/coderoshi/dds

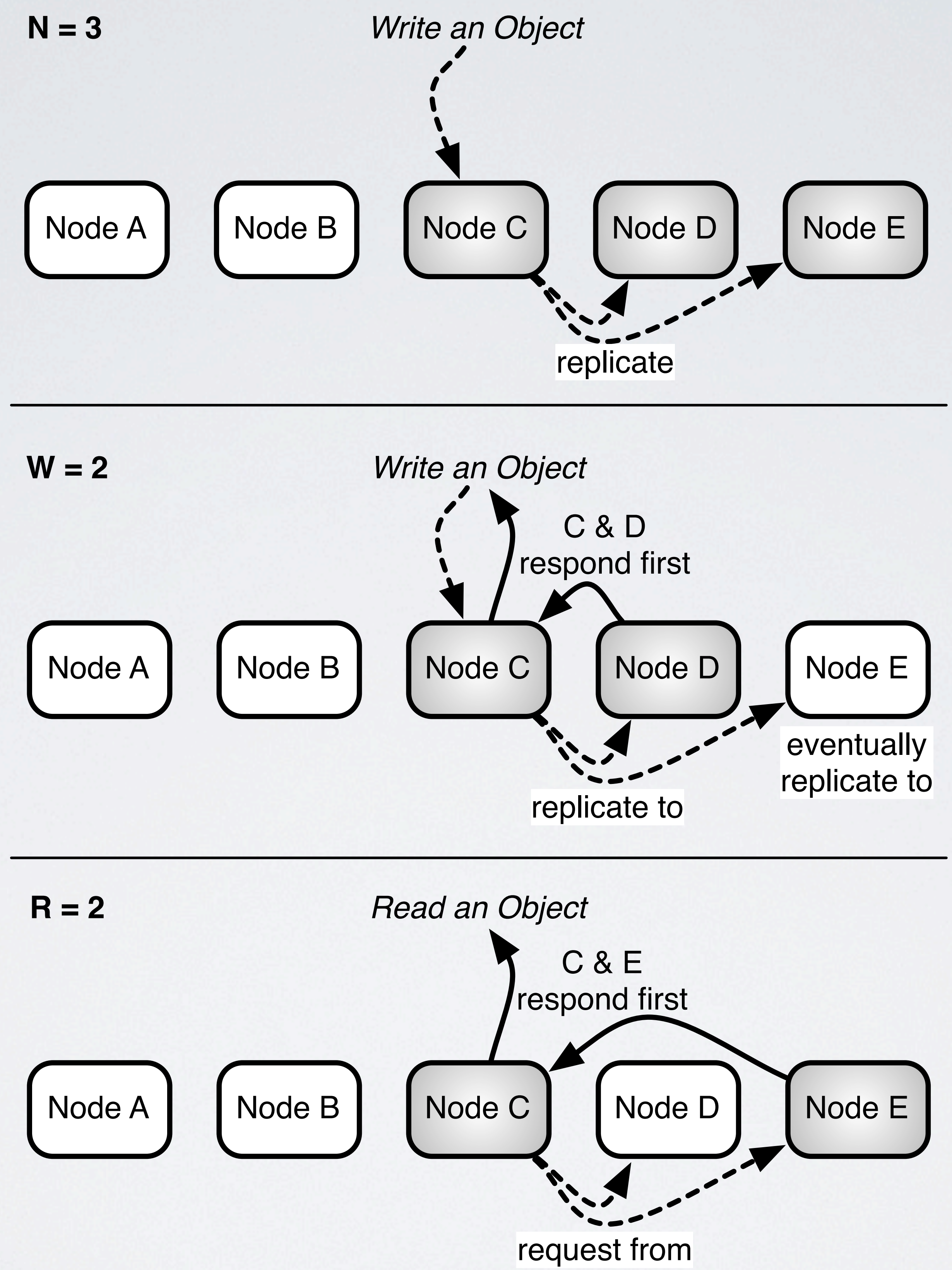
- Balanced Key Space
- Clients Connect to Nodes
- Distribute Objects across Nodes
- Request/Response from any Node
- Nodes Keep Themselves informed of Ring State

WHAT IF A NODE DIES?

REPLICATION

N/R/W

- **N** - # of **Nodes** to replicate a value to
- **R** - # of nodes to **Read** a value from
- **W** - # of nodes to **Write** a value to


N = 3
Write an Object
Node A
Node B
Node C
Node D
Node E
replicate
W = 2
Write an Object
C & D
respond first
Node A
Node B
Node C
Node D
Node E
eventually
replicate to
replicate to
R = 2
Read an Object
C & E
respond first
Node A
Node B
Node C
Node D
Node E
request from

EVENTUAL CONSISTENCY

Perfect is the enemy of good

- How **eventual** is eventual consistency?
- How **consistent** is eventual consistency?
- Probabilistically Bounded Staleness: http://pbs.cs.berkeley.edu/

N=3, R=1, W=1

You have at least a 75.32 percent chance of reading the last written version 0 ms after it commits.
You have at least a 91.4 percent chance of reading the last written version 10 ms after it commits.
You have at least a 99.96 percent chance of reading the last written version 100 ms after it commits.

Replica Configuration

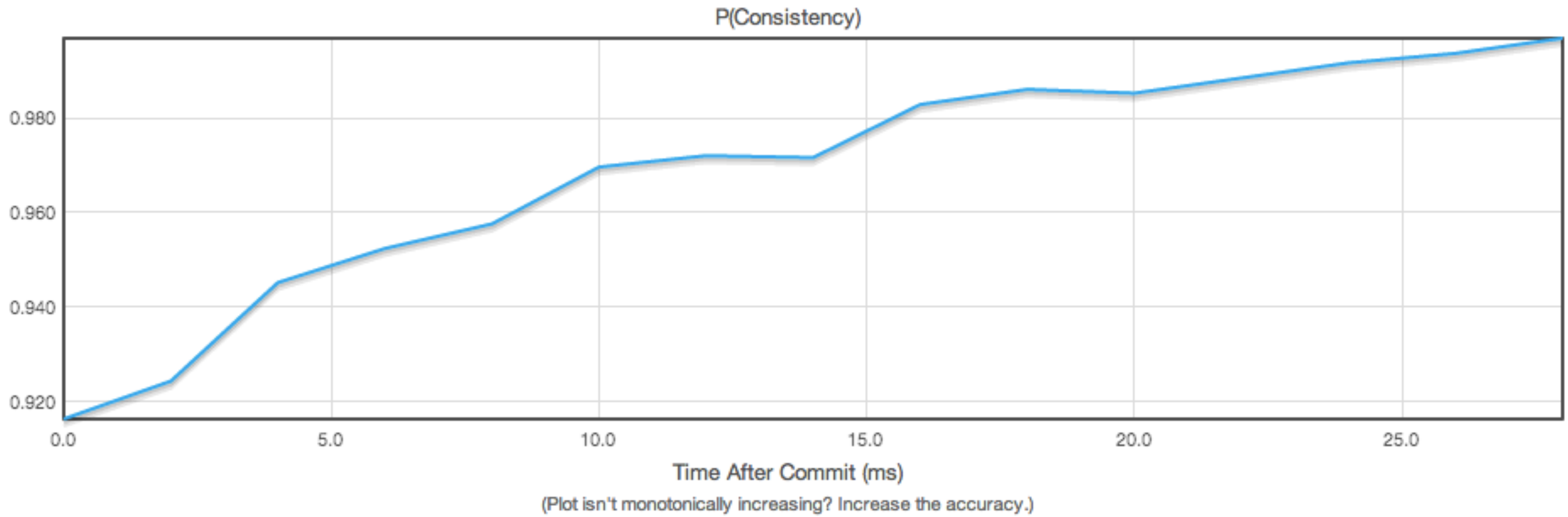
N: 3
R: 1
W: 1

Read Latency: Median 8.33 ms, 99.9th %ile 40.35 ms
Write Latency: Median 8.47 ms, 99.9th %ile 37.65 ms

Tolerable Staleness: 1 version

Accuracy: 2500 iterations/point

N=3, R=1, W=2



You have at least a 90.32 percent chance of reading the last written version 0 ms after it commits.
You have at least a 97.2 percent chance of reading the last written version 10 ms after it commits.
You have at least a 99.96 percent chance of reading the last written version 100 ms after it commits.

Replica Configuration

N: 3
R: 1
W: 2

Read Latency: Median 8.47 ms, 99.9th %ile 36.45 ms
Write Latency: Median 16.77 ms, 99.9th %ile 60.43 ms

Tolerable Staleness: 1 version

Accuracy: 2500 iterations/point

N=3,R=2,W=2


P(Consistency)
1.0100
1.0050
1.0000
0.9950
0.9900
0.00
0.50
1.00
1.50
2.00
2.50
Time After Commit (ms)
(Plot isn't monotonically increasing? Increase the accuracy.)
You are guaranteed to read the last written version 0 ms after it commits.
You are guaranteed to read the last written version 10 ms after it commits.
You are guaranteed to read the last written version 100 ms after it commits.
Replica Configuration
N: 3
R: 2
W: 2
Read Latency: Median 16.85 ms, 99.9th %ile 58.73 ms
Write Latency: Median 16.87 ms, 99.9th %ile 63.16 ms
Tolerable Staleness: 1 version
Accuracy: 2500 iterations/point

```ruby
def put(socket, payload)
  key, value = payload.split(' ', 2)
  socket.send( do_put(key, value).to_s )
end
```

```ruby
def put(socket, payload)
  n, key, value = payload.split(' ', 3)
  socket.send( do_put(key, value, n.to_i).to_s )
end
```

```ruby
def do_put(key, value, n=1)
  if n == 0   # 0 means insert locally
    puts "put 0 #{key} #{value}"
    @data[@ring.hash(key)] = [NodeObject.new(value)]

  elsif @ring.pref_list(key, n).include?(@name)
    puts "put #{n} #{key} #{value}"
    @data[@ring.hash(key)] = [NodeObject.new(value)]
    replicate( "put 0 #{key} #{value}", key, n )
    @data[@ring.hash(key)]

  else
    remote_call(node, "put #{n} #{key} #{value}")
  end
end
```

```ruby
def replicate(message, n)
  list = @ring.pref_list(n)
  results = []
  while replicate_node = list.shift
    results << remote_call(replicate_node, message)
  end
  results
end
```

MORE COPIES, MORE PROBLEMS

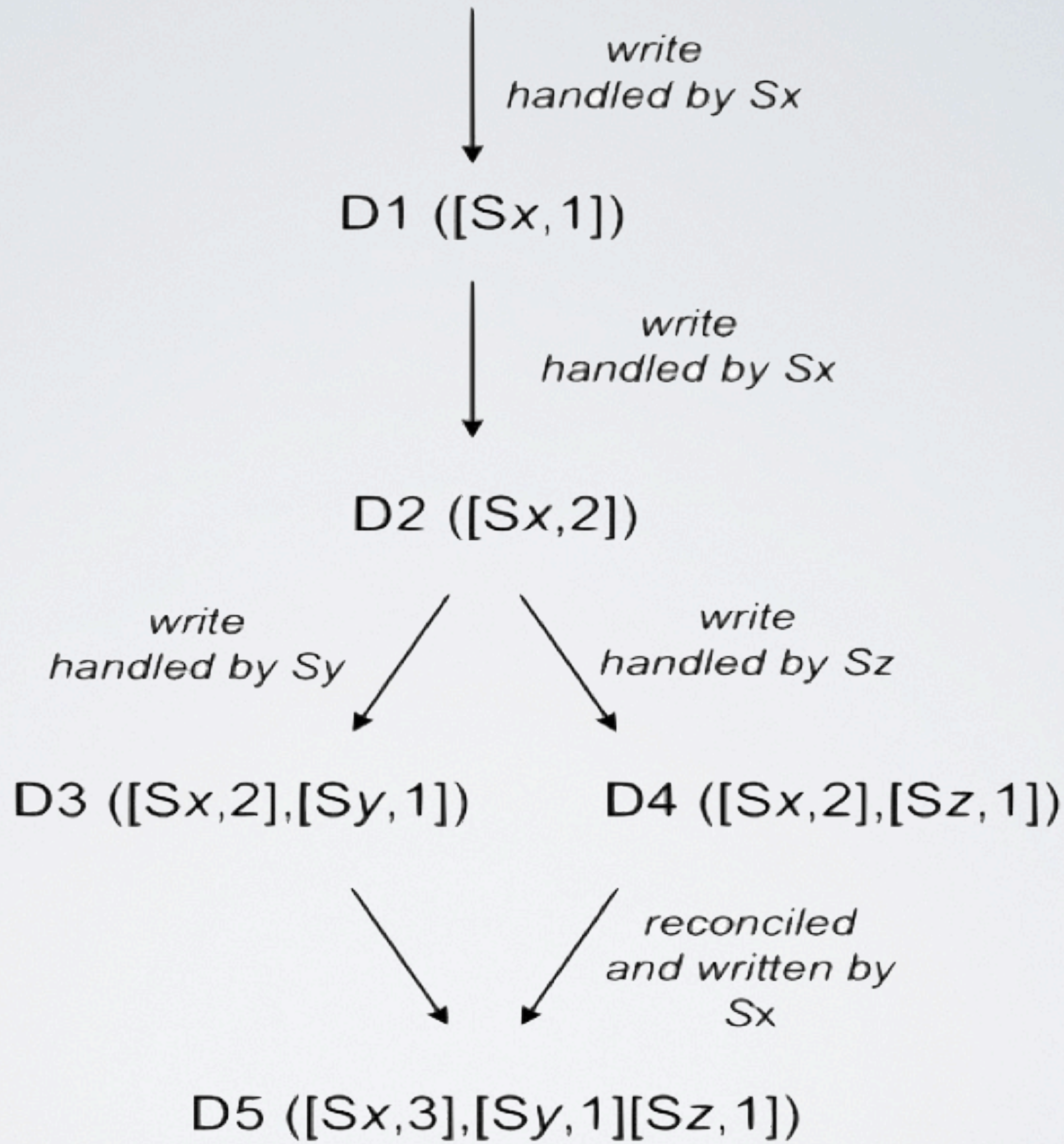
- Different versions on a node can conflict
- Which one is the most recent?
- Vector Clocks

VECTOR CLOCKS

- Cannot (generally) rely on system clocks to be synchronized
- We don't need a system clock, we only need a logical order of actions

WHAT TO EAT FOR DINNER?

- {alice:1} => "pizza"
- {alice:1,bob:1} => "tacos"
- {alice:2,bob:1} => "taco pizza"


write
handled by Sx
D1 ([Sx,1])
write
handled by Sx
D2 ([Sx,2])
write
handled by Sy
write
handled by Sz
D3 ([Sx,2],[Sy,1])
D4 ([Sx,2],[Sz,1])
reconciled
and written by
Sx
D5 ([Sx,3],[Sy,1][Sz,1])

```ruby
class VectorClock
  attr_reader :vector
  def initialize(vector={})
    @vector = vector
  end

  def increment(clientId)
    count = @vector[clientId] || 0
    @vector[clientId] = count + 1
  end

  def descends_from?(vclock2)
    (self <=> vclock2) >= 0 rescue false
  end

  def conflicts_with?(vclock2)
    (self <=> vclock2) rescue return true ensure false
  end

  #...
```

```ruby
def <=>(vclock2)
  equal, descendant, ancestor = true, true, true
  @vector.each do |cid, count|
    if count2 = vclock2.vector[cid]
      equal, descendant = false, false if count < count2
      equal, ancestor = false, false if count > count2
    elsif count != 0
      equal, ancestor = false, false
    end
  end
  vclock2.vector.each do |cid2, count2|
    if !@vector.include?(cid2) && count2 != 0
      equal, descendant = false, false
    end
  end
  if equal then return 0
  elsif descendant && !ancestor then return 1
  elsif ancestor && !descendant then return -1
  end
  raise "Conflict"
end
```

```ruby
vc = VectorClock.new
vc.increment( "adam" )
vc.increment( "barb" )

vc2 = VectorClock.deserialize(vc.to_s)
puts vc <=> vc2                     # => 0

vc2.increment( "adam" )
puts vc2.descends_from?(vc)    # => true

vc.increment( "barb" )
puts vc2.conflicts_with?(vc)   # => true
```

PROBLEMS

- Vector clocks grow forever
- Conflicts require resolution:
  - choose at random
  - siblings (user resolution)
  - pre-defined resolution (eg. CRDT)

CRDT

http://hal.archives-ouvertes.fr/inria-00555588/

- Conflict-free Replicated Data Types

CRDT

http://hal.archives-ouvertes.fr/inria-00555588/

- Conflict-free Replicated Data Types
- Convergent Replicated Data Types

CRDT

http://hal.archives-ouvertes.fr/inria-00555588/

- Conflict-free Replicated Data Types
- Convergent Replicated Data Types
- Commutative Replicated Data Types

THE PROBLEM

- Client A
  - GET counter = 1
  - Increment counter
  - PUT counter 2
- Client B
  - GET counter = 1
  - Increment counter
  - PUT counter 2

Siblings! counter = [2, 2]

counter should be 3, not 2 or 4

THE SOLUTION

- Client A
  - PUT counter +1
  - GET counter => [+1,+1]
- Client B
  - PUT counter +1
  - GET counter => [+1,+1]

Siblings! counter = [+1, +1, +1]

If siblings occur, just aggregate the results

Resolve conflict as = [+3]

```ruby
class Node
  # ...
  def get_counter(socket, payload)
    n, key = payload.split(' ', 2)
    node_objects = do_get( key, n.to_i, :counter )
    # roll up any siblings
    value = node_objects.reduce(0) do |sum,v|
      sum + v.value.to_i
    end
    socket.send( value.to_s )
  end


  def do_put(key, vc, value, n=1, crdt=nil)
    #...
      node_objects = (current_objs || node_objects) if crdt
      # increment counter if this is a counter CRDT
      if crdt == :counter && !node_objects.last.nil?
        last_object = node_objects.last
        last_object.value = last_object.value.to_i + value.to_i
      else
        node_objects += [NodeObject.new(value, vclock)]
      end
    #...
  end
```

```
# Use counters
req.send( "put_counter 1 foo +1" ) && req.recv
req.send( "put_counter 1 foo +2" ) && req.recv
req.send( "put_counter 2 foo +1" ) && req.recv
puts req.send( "get_counter 2 foo" ) && req.recv

# 4
```

COMMON TYPES

- Counters
- Sets
- Graphs

SET PROBLEM

['GWTW']

- Client A
  - PUT cart {add:"GWTDT"}
- Client B
  - PUT cart [
    {add:"BNW"},
    {sub:"GWTW"}]

WHAT WE'VE DONE SO FAR

https://github.com/coderoshi/dds

- Nodes Replicate Writes and Reads
- Version Writes via Vector Clocks
- Simplify Conflict Resolution with CRDTs

HOW DOES REPAIR HAPPEN?

ENTROPY

- *Anti-Entropy* (AE) through Read Repair
- *Active Anti-Entropy* (AAE) with a Merkel Tree

ENTROPY

Increased disorder over time

- Nodes **A** and **B** contain value "**baz**" *(for some key "foo")*
- Node **A** is updated with the value "**qux**"
- Node **B** still contains "**baz**"

READ REPAIR

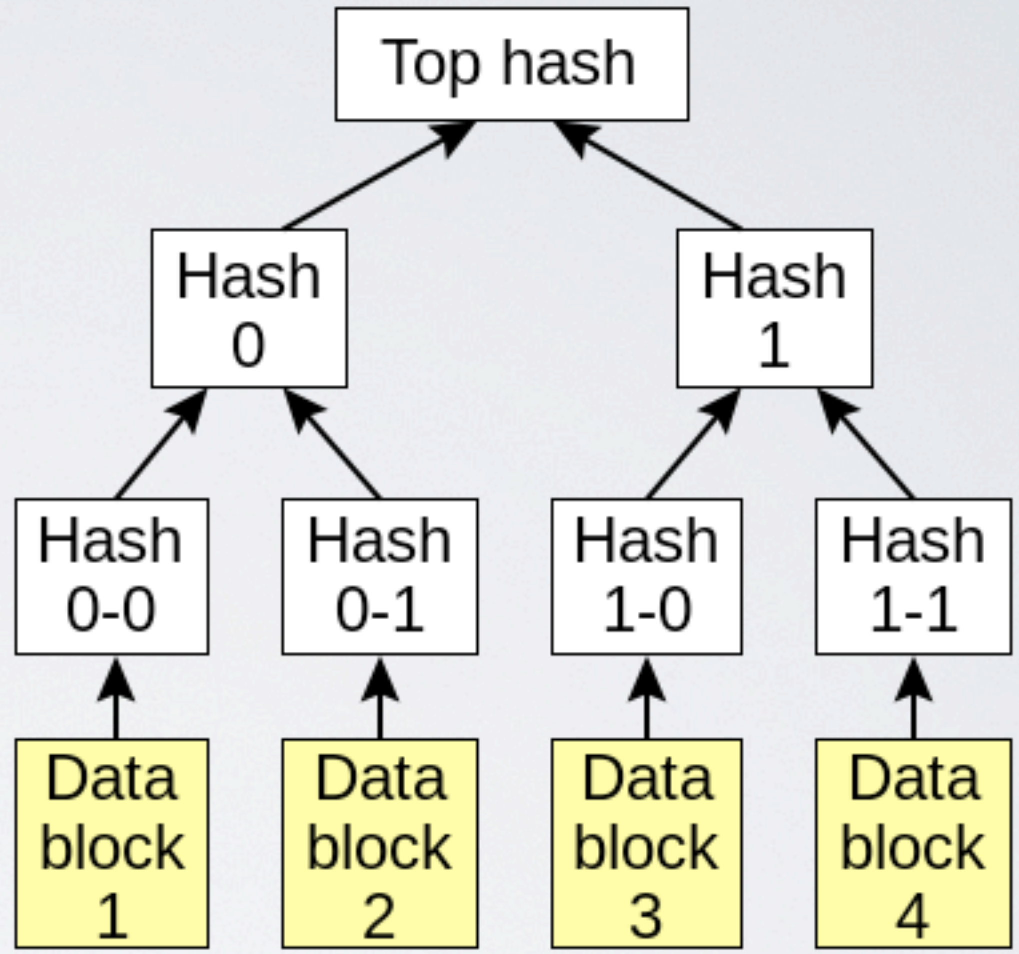
```ruby
def do_get(key, n=1, crdt=nil)
  #...
  repair(key, n)
  return results
end

def repair(key, n)
  list = @ring.pref_list(key, n) - [@name]
  puts "Repairing #{key}"
  list.map do |replicate_node|
    Thread.new do
      results = remote_call( replicate_node, "get 0 #{key}" )
      if (remote_objs = NodeObject.deserialize(results)) != 'null'
        # if local is nil or descends, update local
        local = @data[ @ring.hash(key) ]
        vclock = local && local.first.vclock
        descends = remote_objs.find{|o| o.vclock.descends_from?(vclock)}
        if vclock == nil || descends
          @data[ @ring.hash(key) ] = nos
        end
      end
    end
  end
end
```

```ruby
ctx = ZMQ::Context.new

req1 = ctx.socket( ZMQ::REQ )
req1.connect( "tcp://127.0.0.1:2200" )
req1.send( "put 0 foo {\"B\":1} baz" ) && req1.recv
req1.close

req2 = ctx.socket( ZMQ::REQ )
req2.connect( "tcp://127.0.0.1:2201" )
req2.send( "put 0 foo {} qux" ) && req2.recv

# trigger read repair
puts req2.send( "get 2 foo" ) && req2.recv
sleep 1
# read repair should be complete
puts req2.send( "get 2 foo" ) && req2.recv

req2.close



# [{"value":"qux","vclock":"{\"B\":1}"}]
# [{"value":"baz","vclock":"{\"B\":1,\"A\":1}"}]
```

WHY WAIT?

MERKEL TREE

- A tree of hashes
- Periodically passed between nodes



COMPLEX QUERIES?

MAP/REDUCE

- Popularized by Google then Hadoop
- Transform each object
- Aggregate those transformed Objects

```
array  = [{value:1},{value:3},{value:5}]

mapped = array.map{|obj| obj[:value]}
# [1, 3, 5]

mapped.reduce(0){|sum,value| sum + value}
# 9
```

```ruby
1000.times do |i|
  req.send( "put 2 key#{i} {} #{i}" ) && req.recv
end

req.send( "mr map{|k,v| [1]}; reduce{|vs| vs.length}" )
puts req.recv
```

```
1000.times do |i|
  req.send( "put 2 key#{i} {} #{i}" ) && req.recv
end

req.send( "mr map{|k,v| [1]}; reduce{|vs| vs.length}" )
puts req.recv
```

```
1000.times do |i|
  req.send( "put 2 key#{i} {} #{i}" ) && req.recv
end

req.send( "mr map{|k,v| [1]}; reduce{|vs| vs.length}" )
puts req.recv
```

```ruby
class Map
  def initialize(func_str, data)
    @data = data
    @func = func_str
  end

  def call
    eval(@func, binding)
  end

  # calls given map block for every value
  def map
    @data.map{|k,v| yield(k,v) }.flatten
  end
end
```

```ruby
module Mapreduce

  def mr(socket, payload)
    map_func, reduce_func = payload.split(/\;\s+reduce/, 2)
    reduce_func = "reduce#{reduce_func}"
    socket.send( Reduce.new(reduce_func, call_maps(map_func)).call.to_s )
  end

  def map(socket, payload)
    socket.send( Map.new(payload, @data).call.to_s )
  end

  # run in parallel, then join results
  def call_maps(map_func)
    results = []
    nodes = @ring.nodes - [@name]
    nodes.map {|node|
      Thread.new do
        res = remote_call(node, "map #{map_func}")
        results += eval(res)
      end
    }.each{|w| w.join}
    results += Map.new(map_func, @data).call
  end

end
```

Key/Value

Preference List

Vector Clocks

Distributed Hash Ring

Request/
Response

riak

Merkle Tree

Node Gossip

CRDT (coming)

Read Repair

THANK YOU

@coderoshi

github.com/coderoshi/dds