DISTRIBUTED SYSTEMS
WITH RIAK

Eric Redmond
@coderoshi
Seven Databases in Seven Weeks
A Guide to Modern Databases and the NoSQL Movement

Eric Redmond
and Jim R. Wilson
Edited by Jacquelyn Carter

http://pragprog.com/book/rwdata

WHY?
<table>
<thead>
<tr>
<th>WHY RIAK?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Riak</strong> distributes data across <strong>multiple nodes</strong> to be</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scalable</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal scaling allows for expansions of resources by adding/removing computers to/from the network.</td>
<td>Composed of autonomous and interconnected computers that isolate successes and failures.</td>
</tr>
</tbody>
</table>
DISTRIBUTED SYSTEM

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

• **Consistent**
• **Available**
• **Partition-Tolerant**

EVENTUALLY CONSISTENT
Perfect is the enemy of good

• How Eventual?
• How Consistent?
BY DATA STRUCTURE

- Relational
- Graph
- Document
- Column Family
- Key/Value
• We all know and 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
Key based, but schemaless data

MongoDB, CouchDB, Couchbase

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 data structures

• You can always chain lookups

• Redis, Memcached, Riak
BY LOGICAL TOPOLOGY

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

• Master, Slaves

• mysql, 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
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
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]]
h = NaiveHash.new(["A", "B", "C"])  

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

# ...
# ...

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

puts h.node("foo")  # => A
h.add("D")
puts h.node("foo")  # => A
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%
A single partition SHA1(Key)

Ring with 32 partitions

Node 0
Node 1
Node 2
A single partition

Ring with 32 partitions

SHA1(Key)

2^{160} / 2

Node 0
Node 1
Node 2
Node 3
SHA1BITS = 160

```ruby
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")
```

```ruby
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"
```

```ruby
# misses: 9.473%
```

Thursday, February 21, 13
# 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

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
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
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
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)
OMQ

• Higher level than sockets, lower level than middlewares

• Transport agnostic (Mem, IPC, TCP, PGM, etc)

• Message-oriented, not stream or datagram
# REQUEST/REPLY

**Request**

<table>
<thead>
<tr>
<th>Direction</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send/receive pattern</td>
<td>send, receive, send, receive...</td>
</tr>
<tr>
<td>Incoming route strategy</td>
<td>Last peer</td>
</tr>
<tr>
<td>Outgoing route strategy</td>
<td>Round-robin</td>
</tr>
</tbody>
</table>

**Reply**

<table>
<thead>
<tr>
<th>Direction</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send/receive pattern</td>
<td>receive, send, receive, send...</td>
</tr>
<tr>
<td>Incoming route strategy</td>
<td>Fair-queued</td>
</tr>
<tr>
<td>Outgoing route strategy</td>
<td>Last peer</td>
</tr>
</tbody>
</table>
# 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
  end

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

Thursday, February 21, 13
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
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
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
end

def method_missing(method, *args, &block)
  socket, payload = args
  payload.send( "bad message" ) if payload
end
/* A.json */
{
  "name": "A",
  "port": 2200
}
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

  # ...
  
  Thursday, February 21, 13
# ...

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

# 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

<table>
<thead>
<tr>
<th>Publish</th>
<th>Subscribe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direction</strong></td>
<td><strong>Direction</strong></td>
</tr>
<tr>
<td>Unidirectional</td>
<td>Unidirectional</td>
</tr>
<tr>
<td><strong>Send/receive pattern</strong></td>
<td><strong>Send/receive pattern</strong></td>
</tr>
<tr>
<td>Send only</td>
<td>Receive only</td>
</tr>
<tr>
<td><strong>Incoming route strategy</strong></td>
<td><strong>Incoming route strategy</strong></td>
</tr>
<tr>
<td>N/A</td>
<td>Fair-queued</td>
</tr>
<tr>
<td><strong>Outgoing route strategy</strong></td>
<td><strong>Outgoing route strategy</strong></td>
</tr>
<tr>
<td>Fan out</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Publisher connects to Subscriber(s) unidirectionally, sending messages as a publisher and receiving messages as a subscriber.
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
        end
        @ring.cluster(nodes)
      end
      pub.send( "ring " + nodes.join(','))
      rep.send( "true" )
    end
  end
end
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
end
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
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?
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

```
Node A  Node B  Node C  Node D  Node E
```

C & D respond first

replicate to

eventually replicate to

**R = 2**

Read an Object

```
Node A  Node B  Node C  Node D  Node E
```

C & E respond first

request from
EVENTUAL CONSISTENCY

Perfect is the enemy of good

• How \textit{eventual} is eventual consistency?

• How \textit{consistent} is eventual consistency?

• Probabilistically Bounded Staleness: \url{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
$N=3, R=2, W=2$
def put(socket, payload)
    key, value = payload.split(' ', 2)
    socket.send( do_put(key, value).to_s )
end

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

    #...

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

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

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]
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
end


```ruby
# 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"}]

Thursday, February 21, 13
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
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
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
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
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
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
THANK YOU
@coderoshi
github.com/coderoshi/dds