

# DISTRIBUTED PATTERNS IN ACTION

<http://git.io/MYrjpQ>

Eric Redmond  
**@coderoshi**







**MONEY**  
best superpower ever

A

B

C



# RESOURCE EXPANSION (SOLUTION: SHARDING)

# SHARDING INCREASES RISK



# FAULT-TOLERANCE (SOLUTION: REPLICATION)

REPLICATION IS THE  
ROOT OF ALL EVIL





# THE CAUSE OF MOST NETWORK PARTITIONS



# THE CAP THEOREM SUCKS

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




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

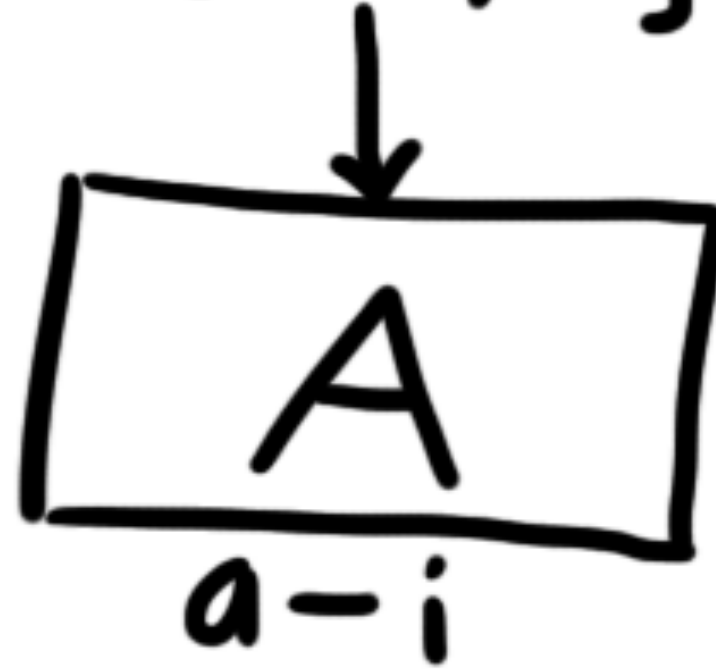


DON'T DISTRIBUTE DATASTORES,  
STORE DISTRIBUTED DATA

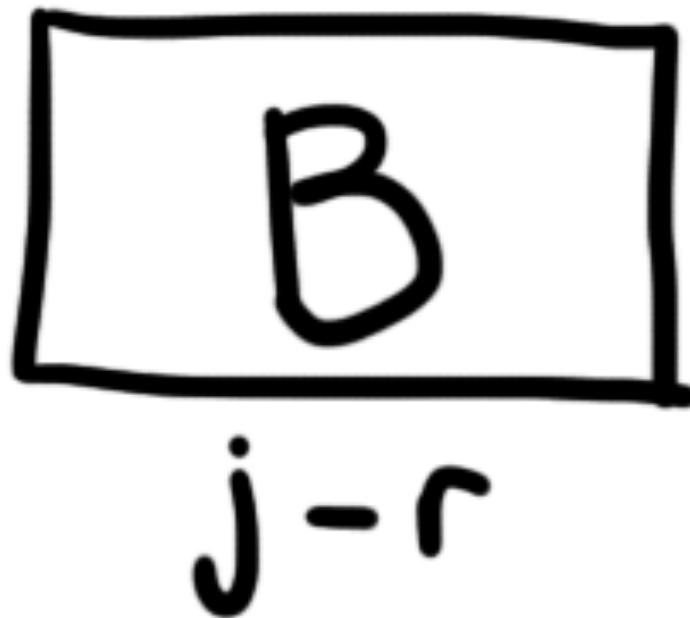
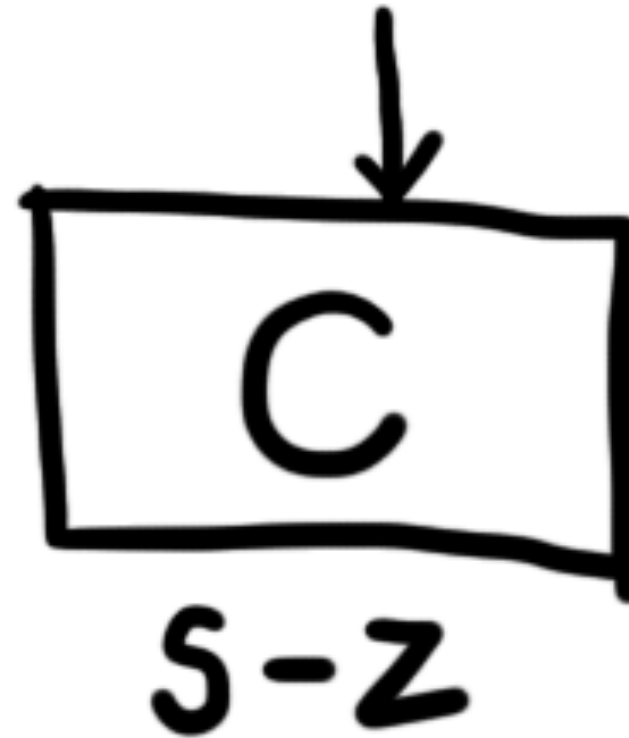


IF IT CAN HAPPEN,  
AT SCALE IT WILL HAPPEN

{apple, 



{zebra, 

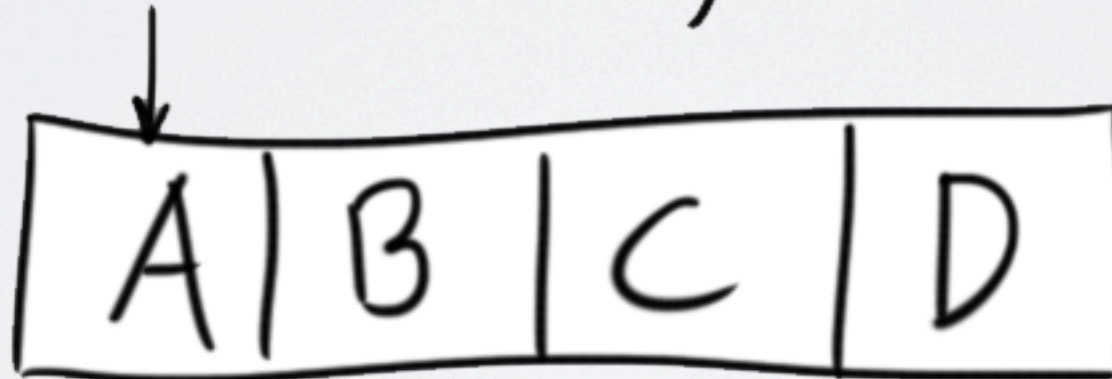




$$4 \ (4 \% 3 = 1)$$



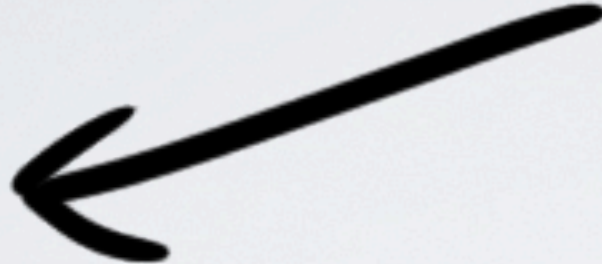
$$4 \ (4 \% 4 = 0)$$



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

```
100000.times do |i|
  tracknodes[i] = h.node(i)
end
```

```
h.add("K")
```

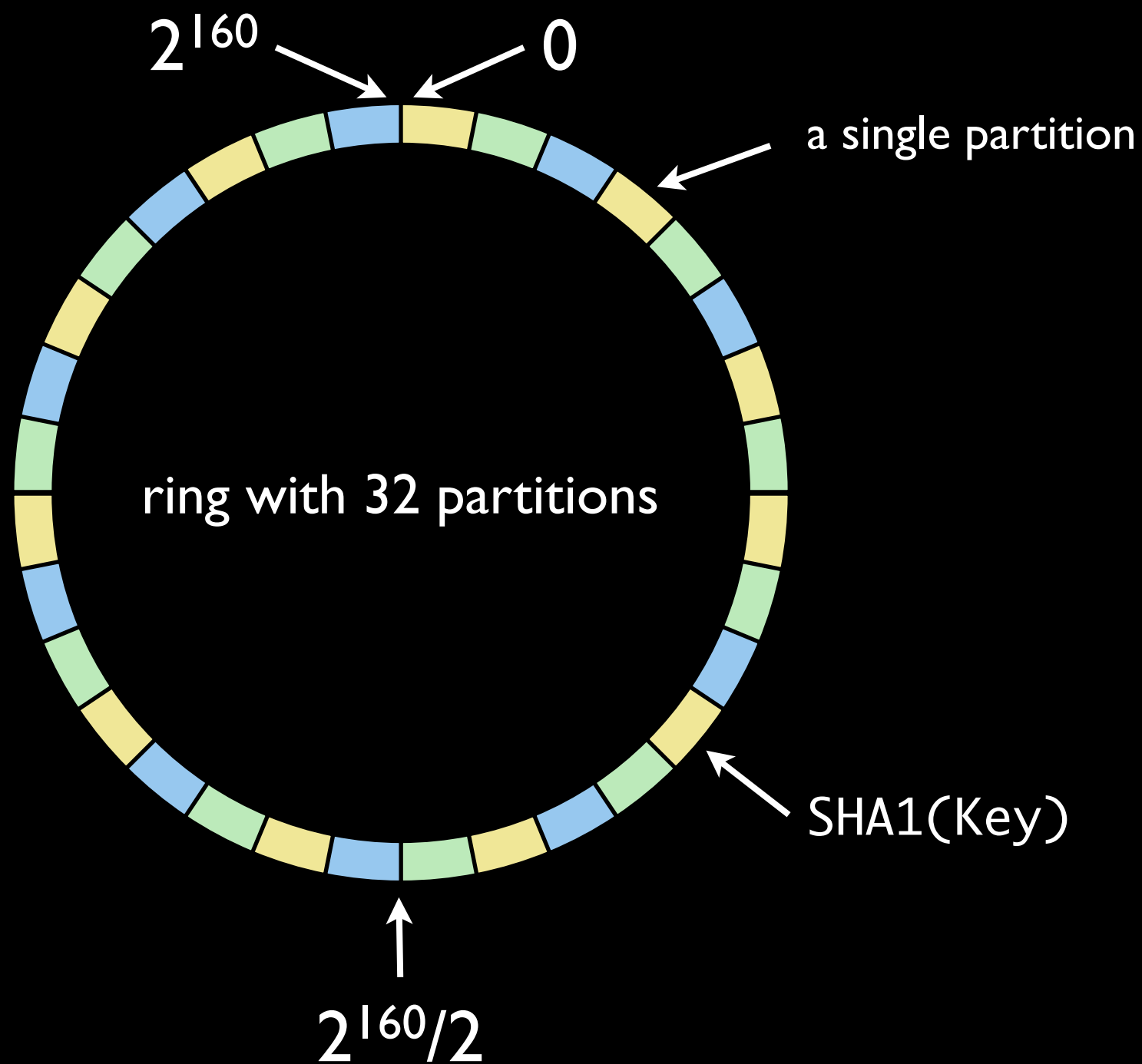


```
misses = 0
100000.times do |i|
  misses += 1 if tracknodes[i] != h.node(i)
end
```

```
puts "misses: #{(misses.to_f/100000) * 100}%"
```

***misses: 90.922%***

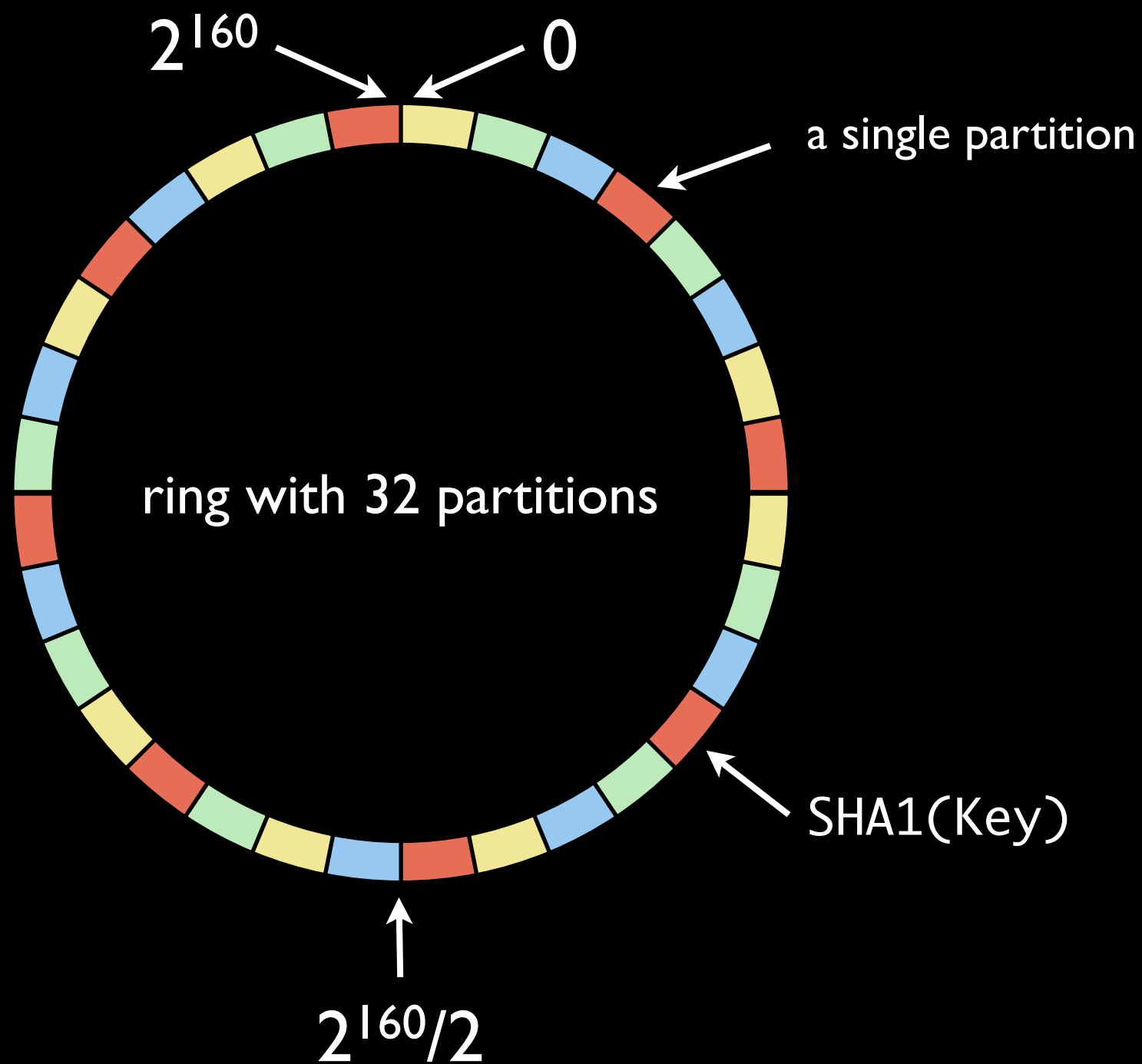




Node 0

Node 1

Node 2



Node 0

Node 1

Node 2

Node 3



```

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

```

```

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

```

*misses: 9.473%*



```
class Node
```

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

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

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

```
end
```



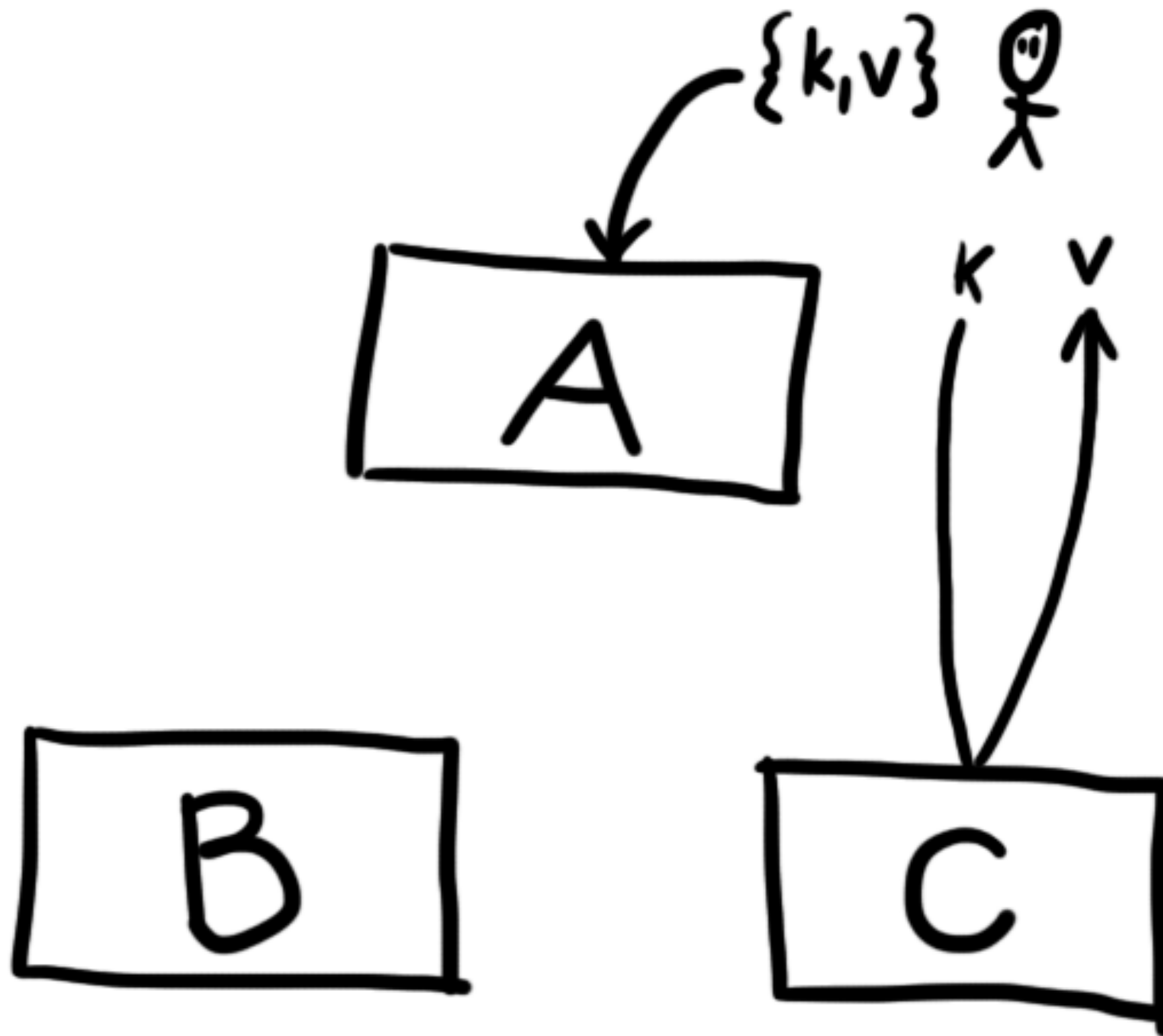
```
nodeA = Node.new( 'A', [ 'A', 'B', 'C' ] )  
nodeB = Node.new( 'B', [ 'A', 'B', 'C' ] )  
nodeC = Node.new( 'C', [ 'A', 'B', 'C' ] )
```

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

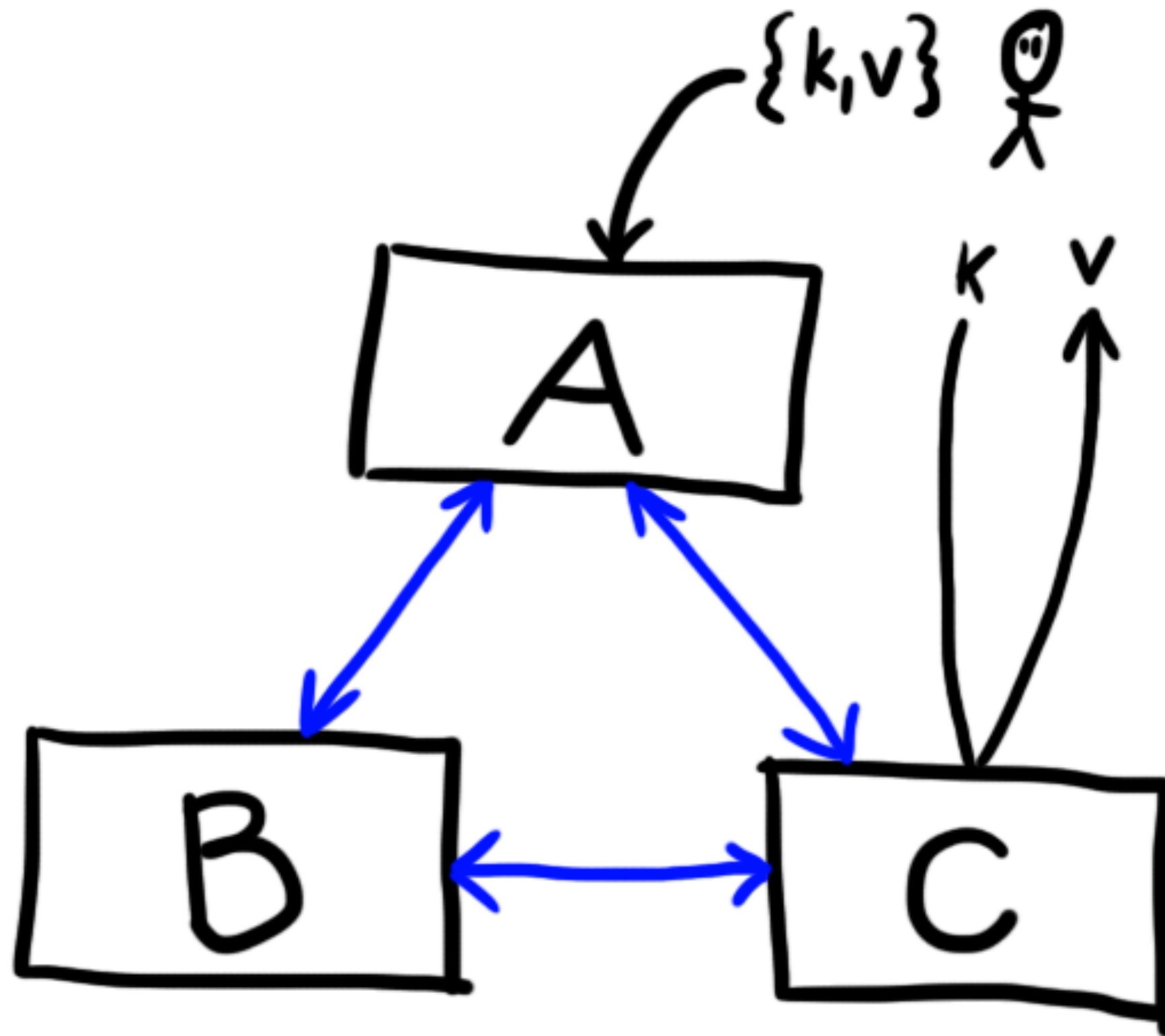
```
nodeB.put( "foo", "bar" )  
p nodeB.get( "foo" )    # "bar"
```

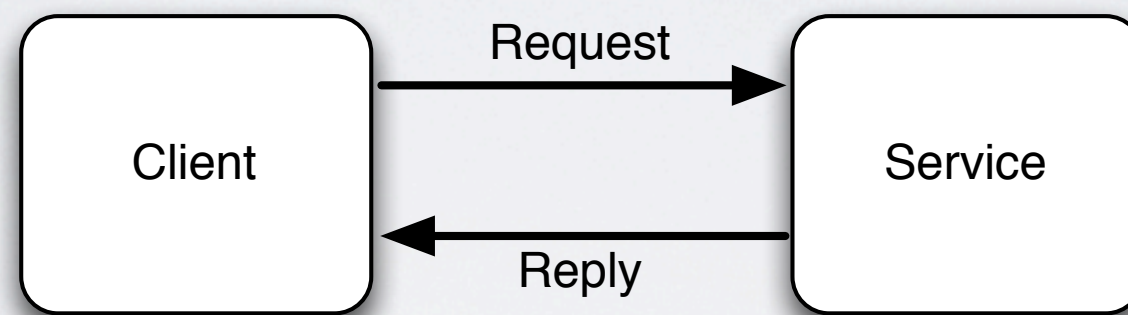


```
nodeC.put( "foo", "bar" )  
p nodeC.get( "foo" )    # nil
```











```
module Services
```

```
  def connect(port=2200, ip="127.0.0.1")
    ctx = ZMQ::Context.new
    sock = ctx.socket( ZMQ::REQ )
    sock.connect( "tcp://#{ip}:#{port}" )
    sock
  end
```



```
  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 )      # EVVVIILL!!!
      end
    end
  end
```



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

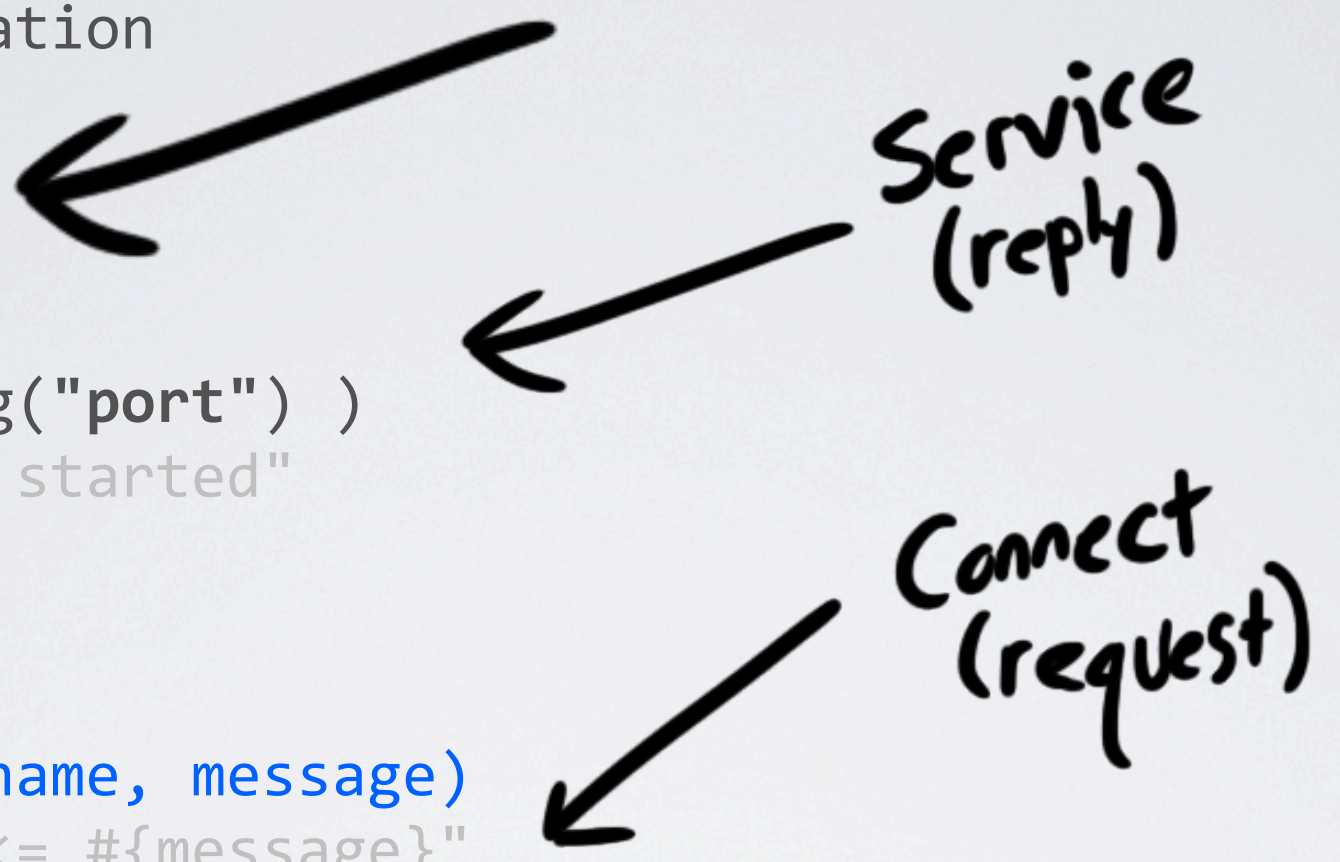


```
class Node
  include Configuration
  include Threads
  include Services

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

  def remote_call(name, message)
    puts "#{name} <= #{message}"
    req = connect(config("port", name), config("ip", name))
    resp = req.send(message) && req.recv
    req.close
    resp
  end

  # ...
```



Handwritten annotations with arrows pointing to code elements:

- An arrow points from the `include Services` line to the word **Service** in the handwritten text **Service (reply)**.
- An arrow points from the `service( config("port") )` line to the word **Service** in the handwritten text **Service (reply)**.
- An arrow points from the `remote_call` method definition to the word **Connect** in the handwritten text **Connect (request)**.



```
# ...
```

```
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)] = 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()

\$ ruby node.rb A

\$ ruby node.rb B

\$ ruby node.rb C

---

*# 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"

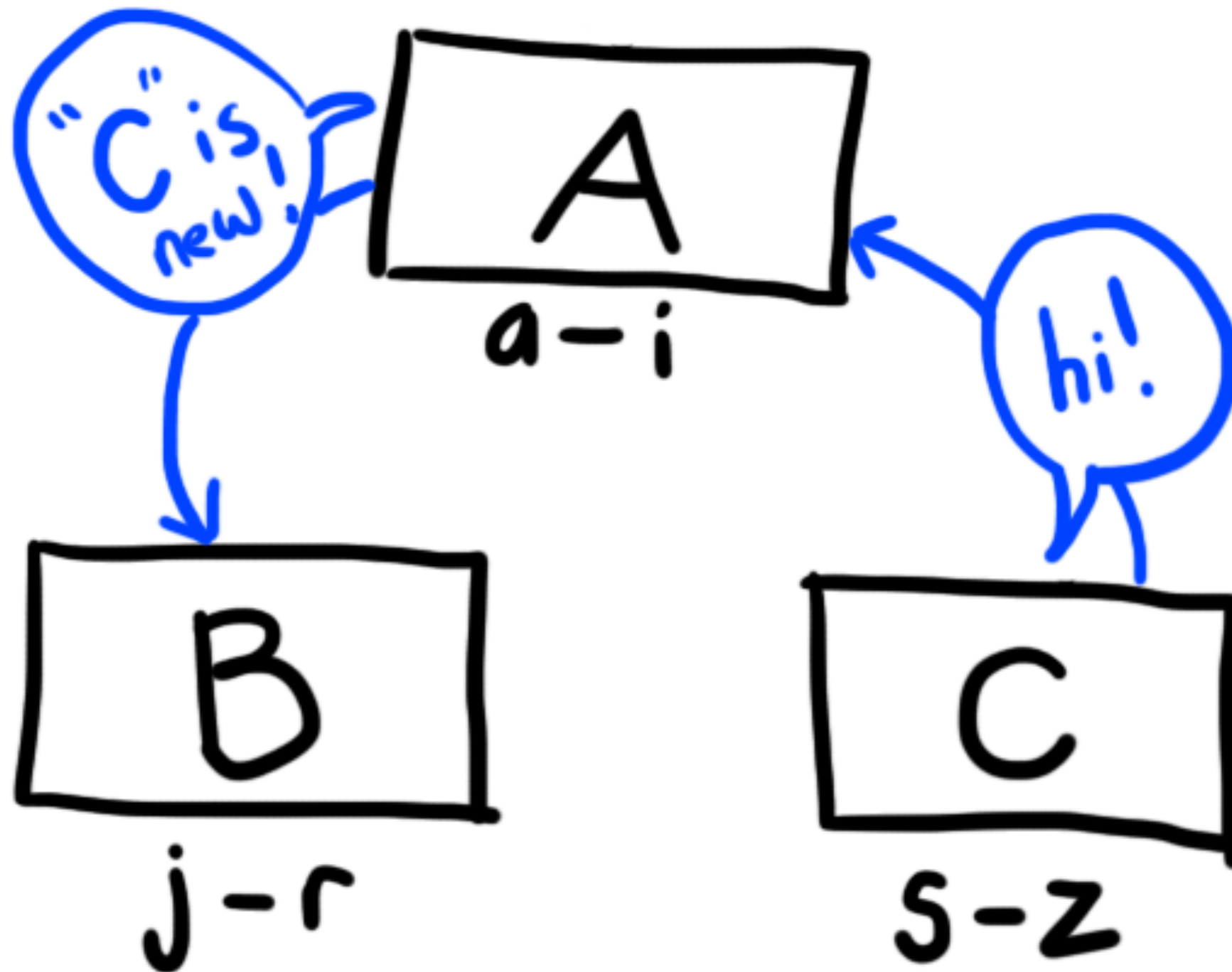
1000.times do |i|

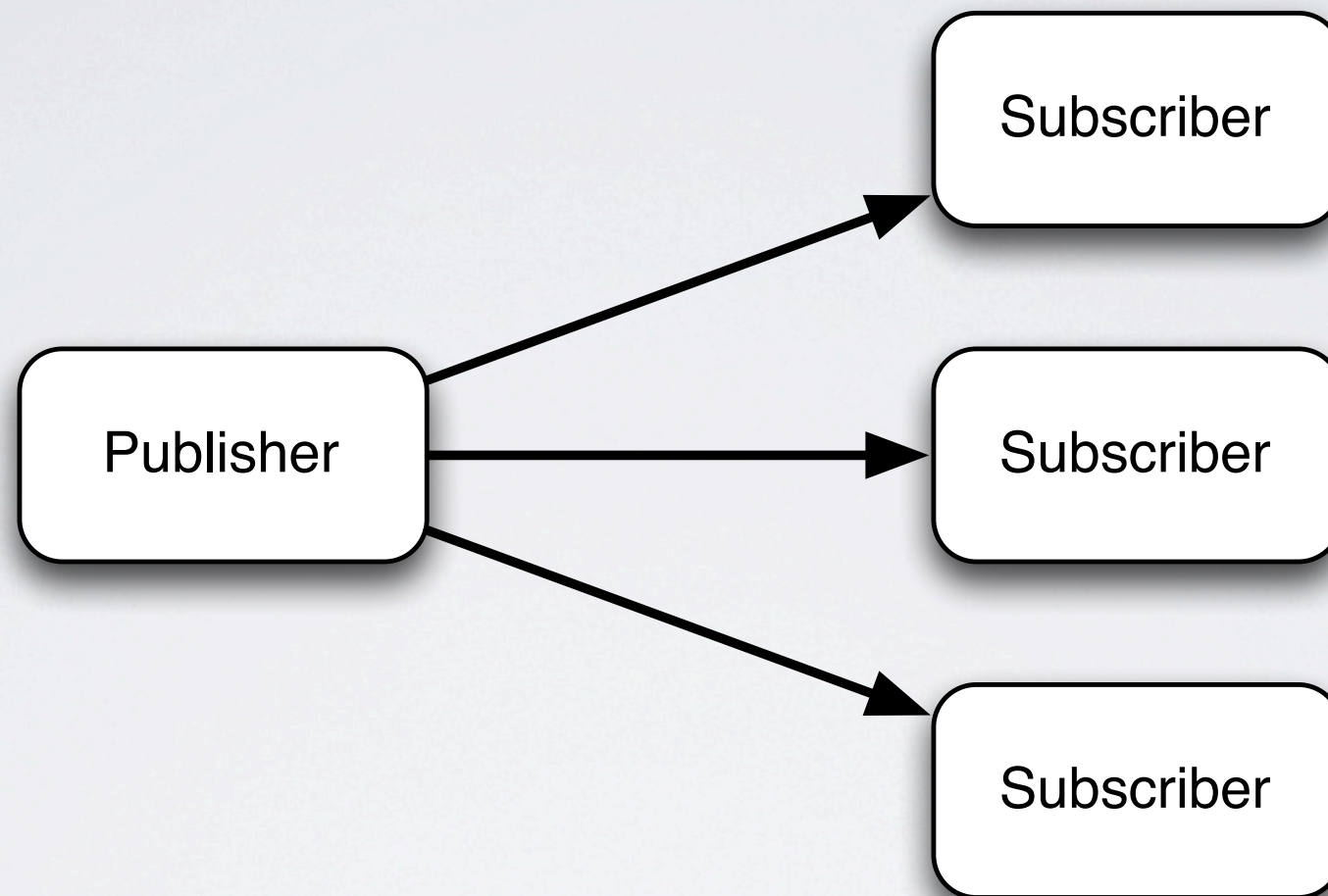
puts req.send( "get key#{i}" ) && req.recv

end

req.close









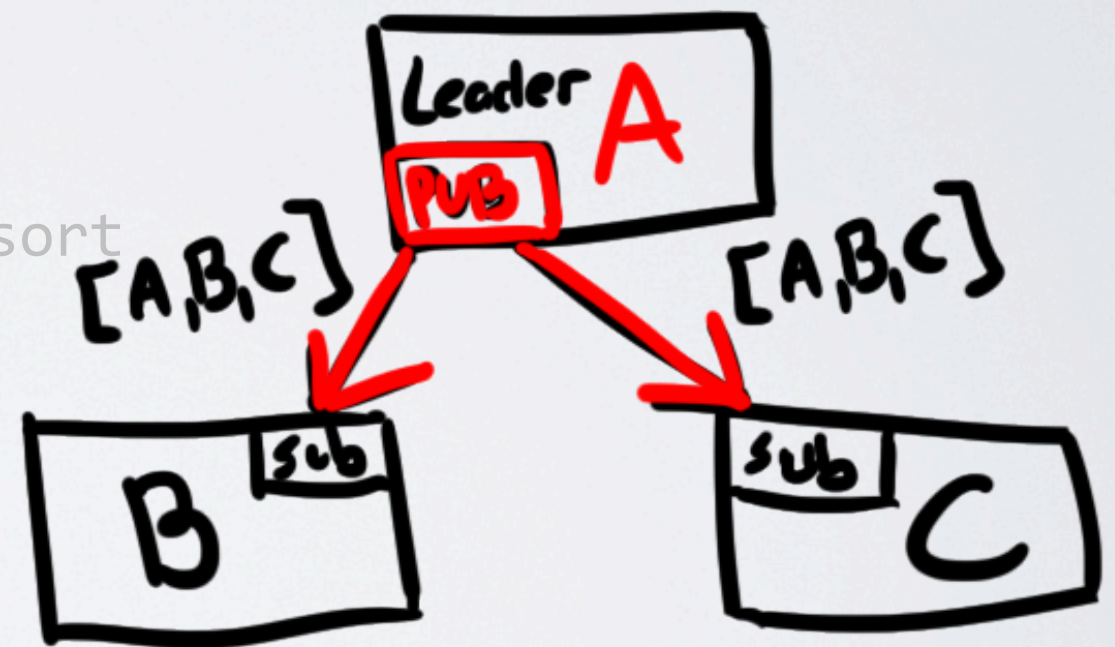
```

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
end

```



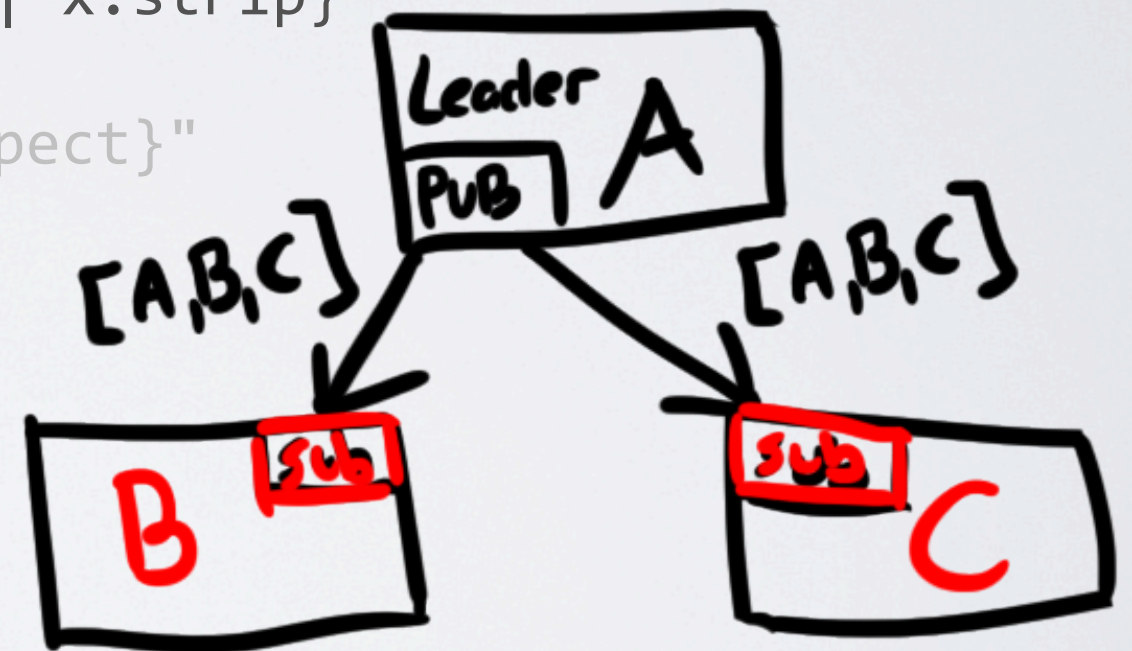


```

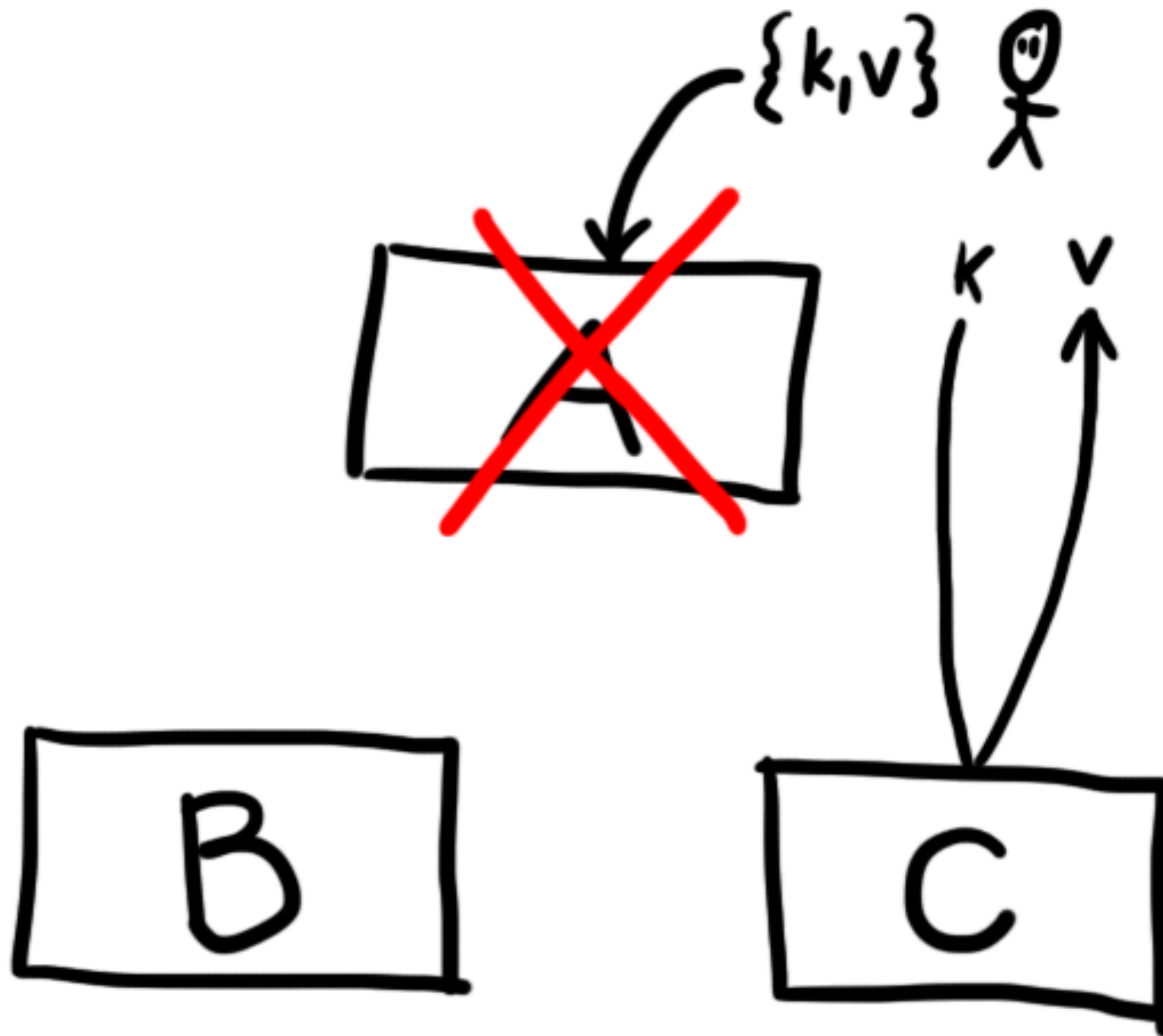
class Node
  # ...
  def track_cluster(sub_port)
    thread do
      ctx = ZMQ::Context.new
      sub = ctx.socket( ZMQ::SUB )
      sub.connect( "tcp://127.0.0.1:#{sub_port}" )
      sub.setsockopt( ZMQ::SUBSCRIBE, "ring" )

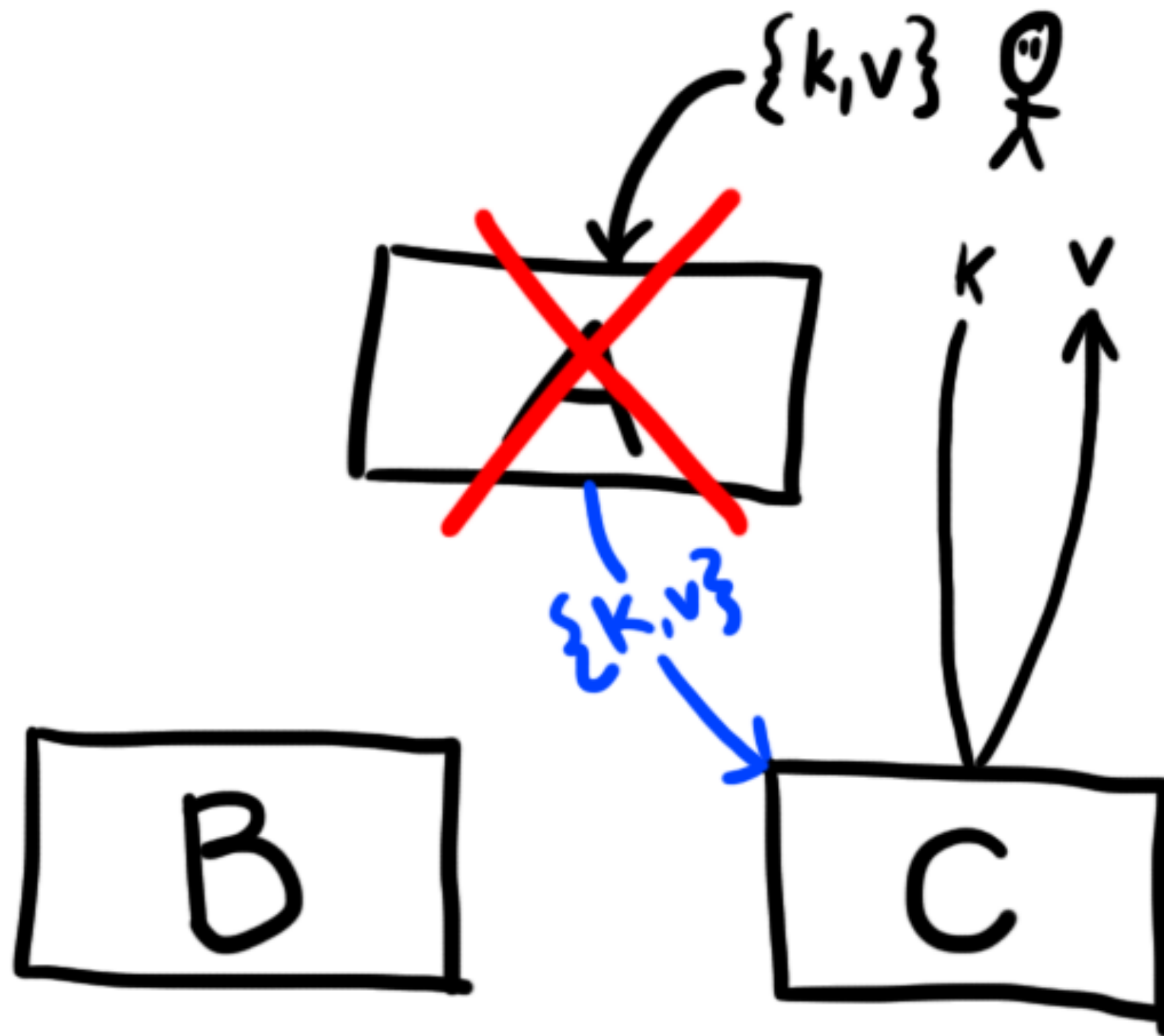
      while line = sub.recv
        _, nodes = line.split(' ', 2)
        nodes = nodes.split(',').map{|x| x.strip}
        @ring.cluster( nodes )
        puts "ring changed: #{nodes.inspect}"
      end
    end
  end
end

```



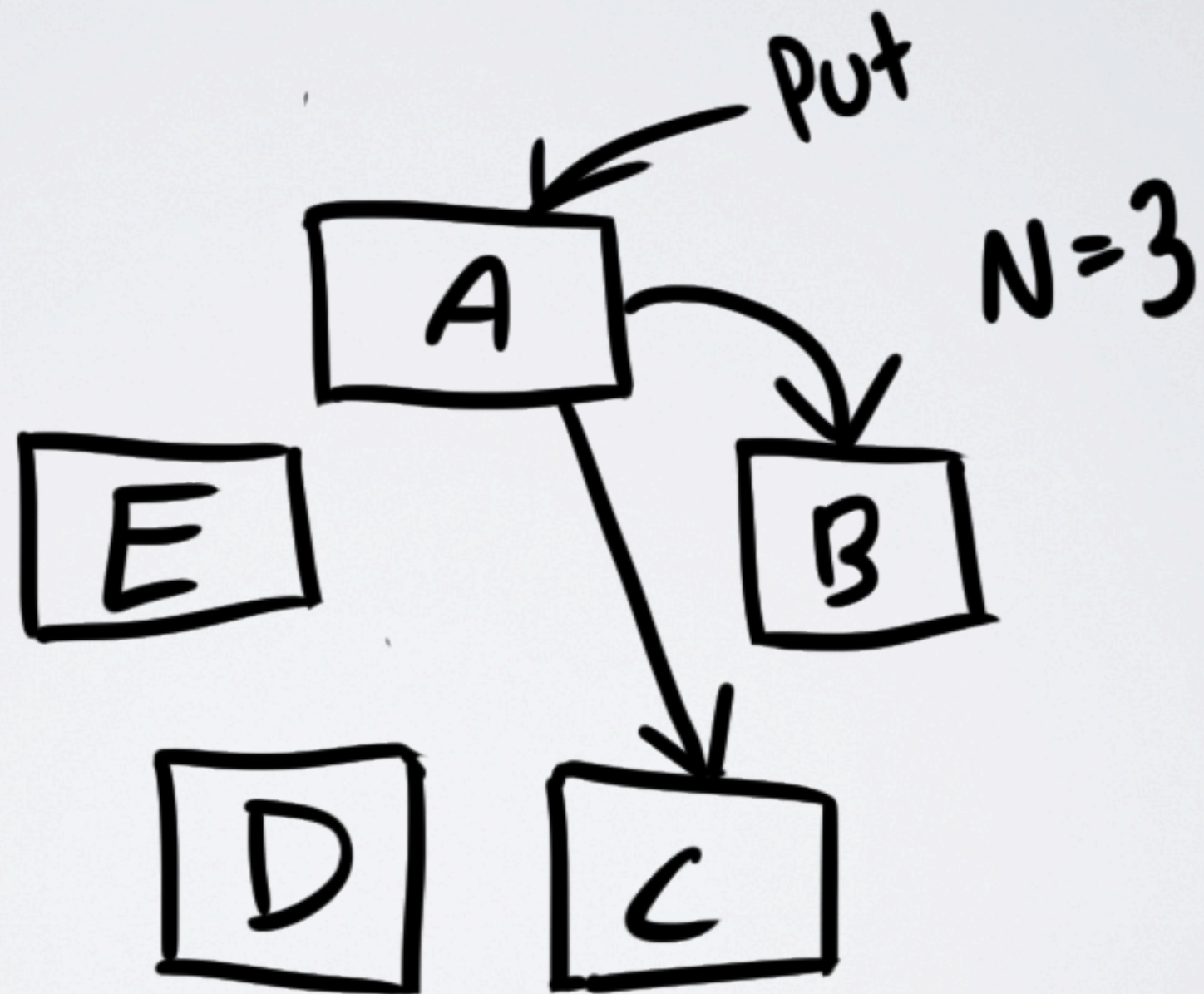


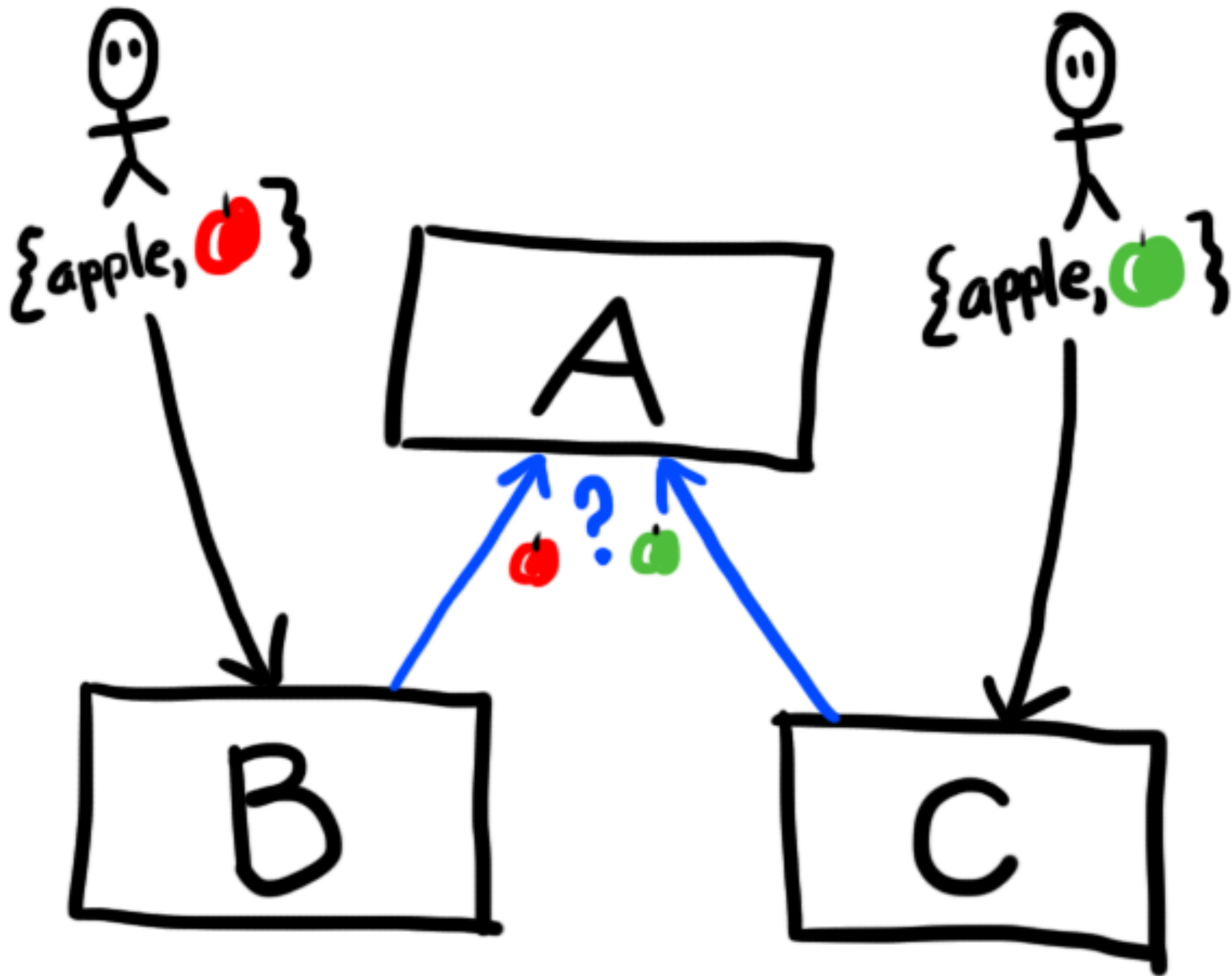




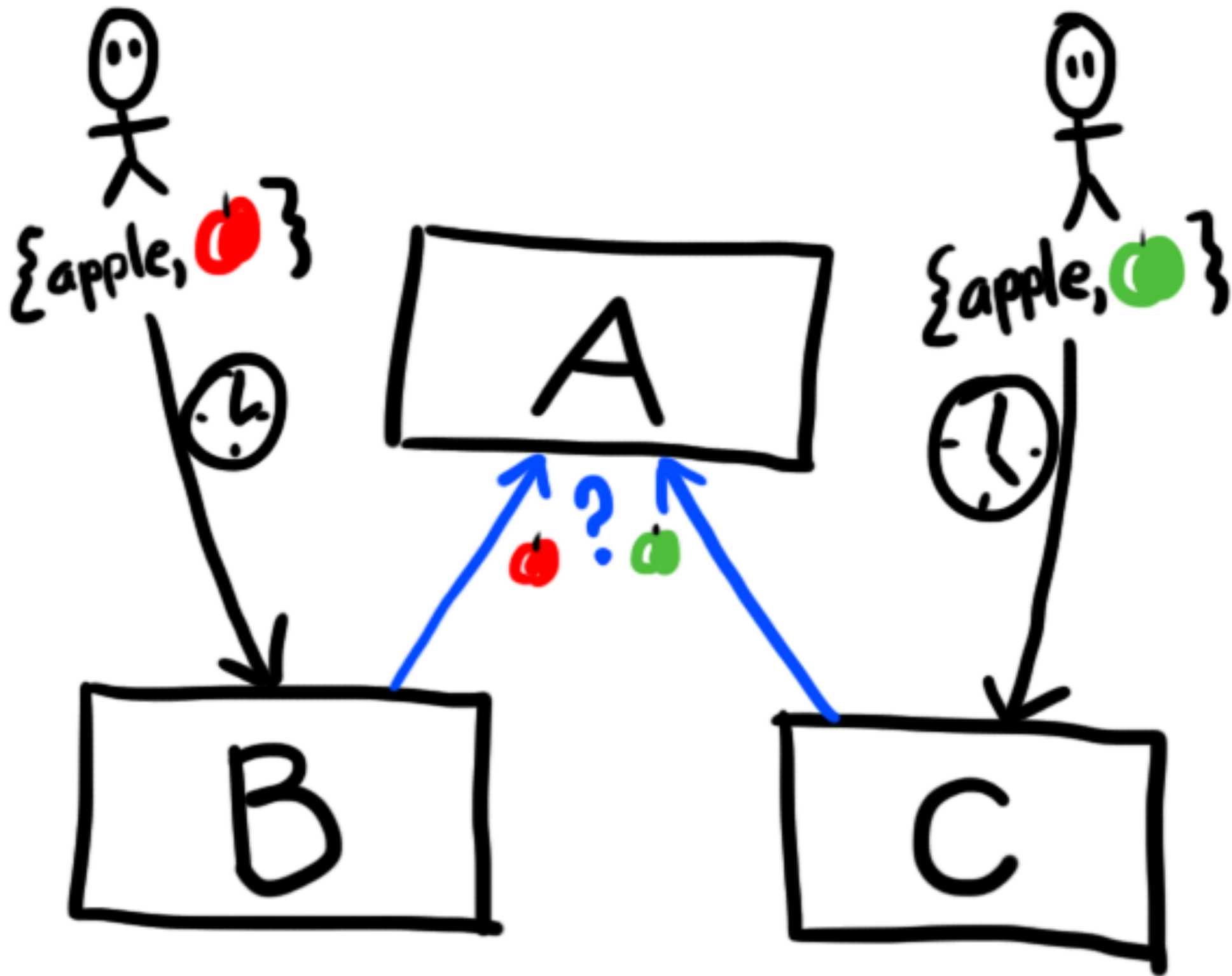


```
def replicate(message, n)
  list = @ring.pref_list(n)
  results = []
  while replicate_node = list.shift
    results << remote_call(replicate_node, message)
  end
  results
end
```









# WHAT TO EAT FOR DINNER?

- Adam wants Pizza

**{value:"pizza", vclock:{adam:1}}**

- Barb wants Tacos

**{value:"tacos", vclock:{barb:1}}**

- Adam gets the value, the system *can't* resolve, so he gets *both*

**[{value:"pizza", vclock:{adam:1}},  
{value:"tacos", vclock:{barb:1}}]**

- Adam resolves the value however he wants

**{value:"taco pizza", vclock:{adam:2, barb:1}}**



*# artificially create a conflict with vclocks*

```
req.send('put 1 foo {"B":1} hello1') && req.recv
```

```
req.send('put 1 foo {"C":1} hello2') && req.recv
```

```
puts req.send("get 2 foo") && req.recv
```

```
sleep 5
```

*# resolve the conflict by decending from one of the vclocks*

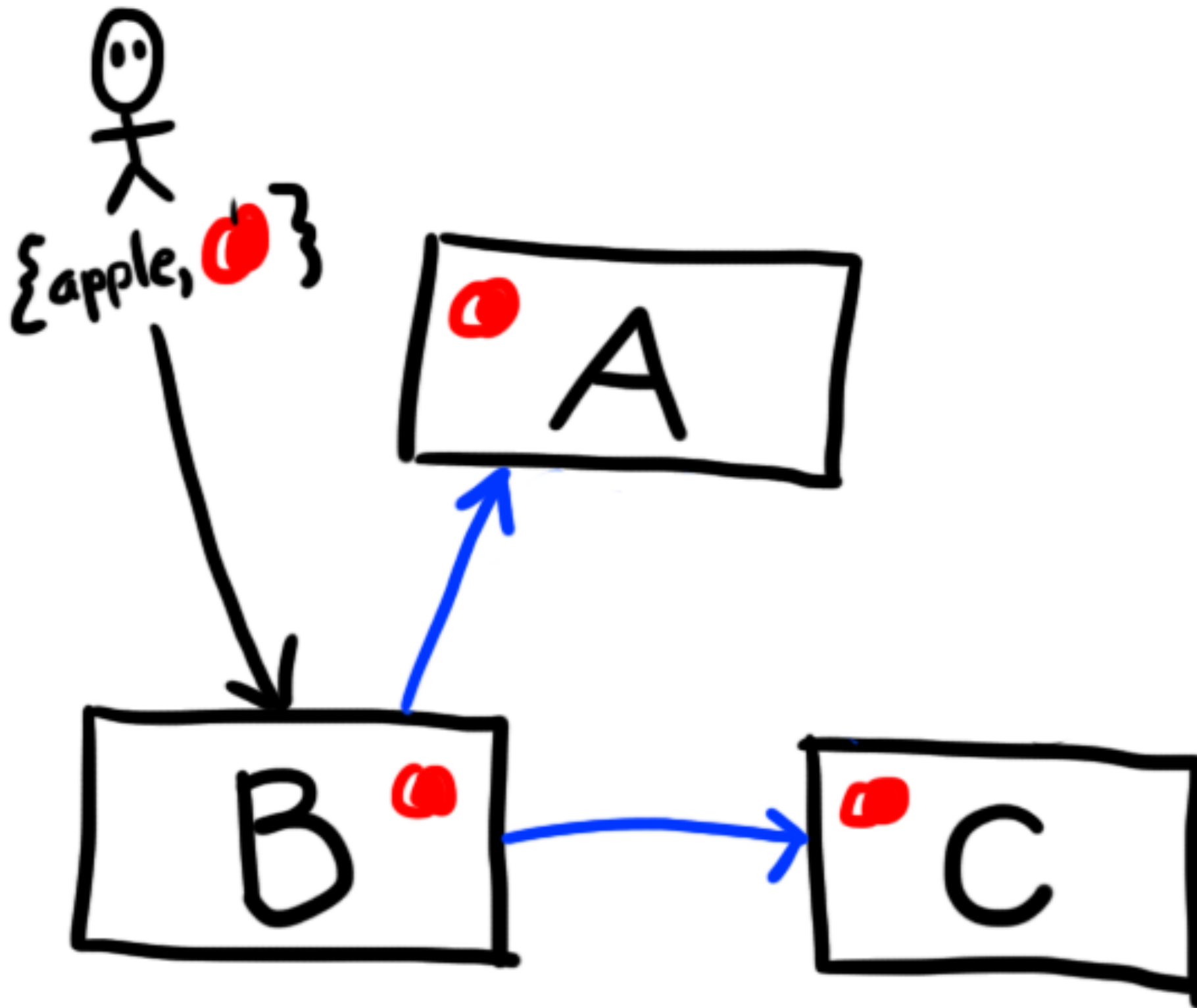
```
req.send('put 2 foo {"B":3} hello1') && req.recv
```

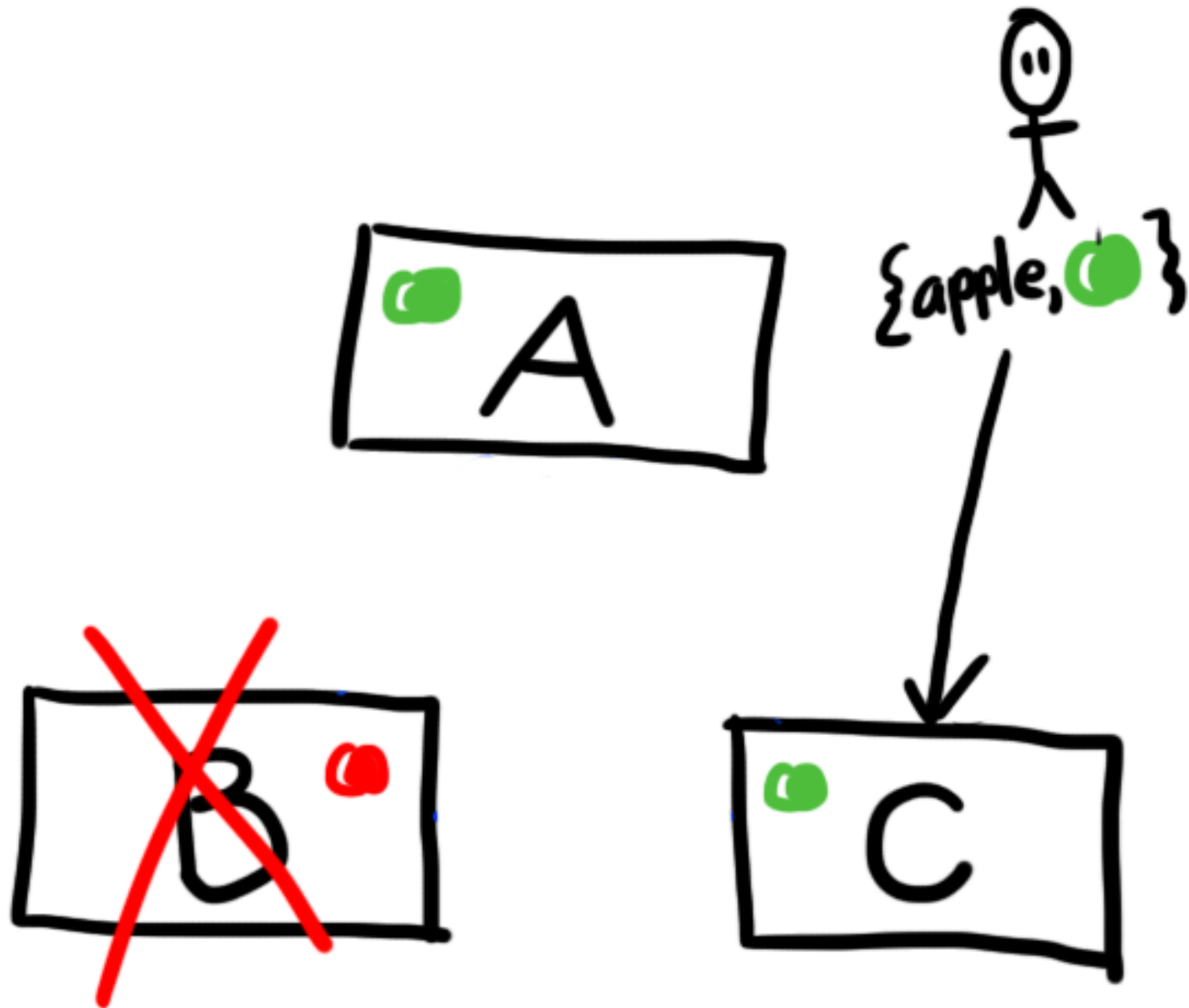
```
puts req.send("get 2 foo") && req.recv
```

# CONFLICT RESOLUTION

- choose a value at random
- siblings (user resolution)
- defined resolution (eg. CRDT)

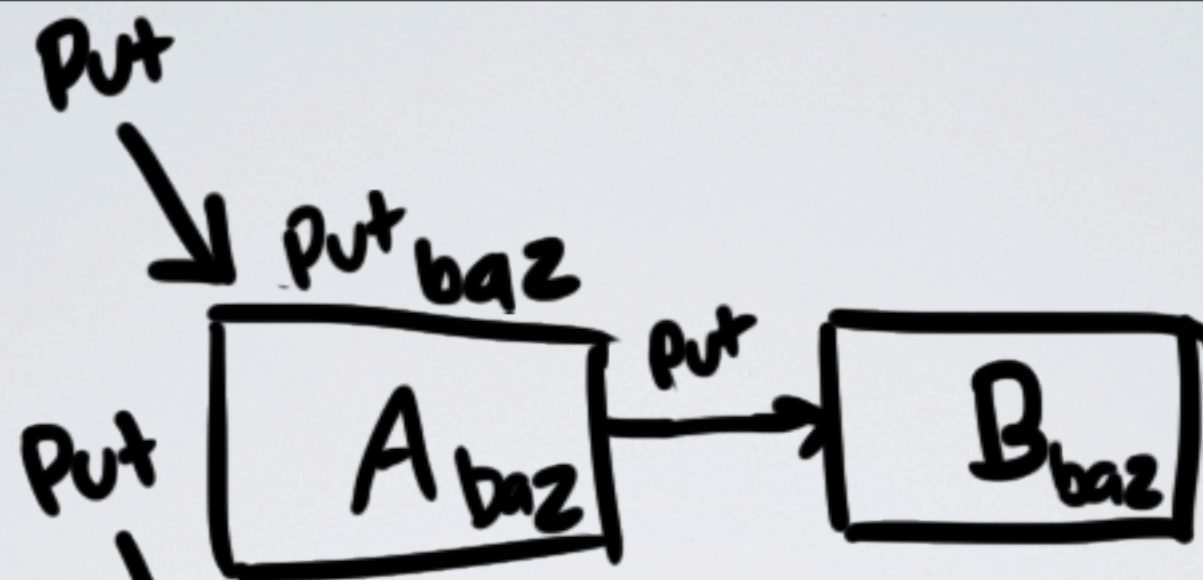




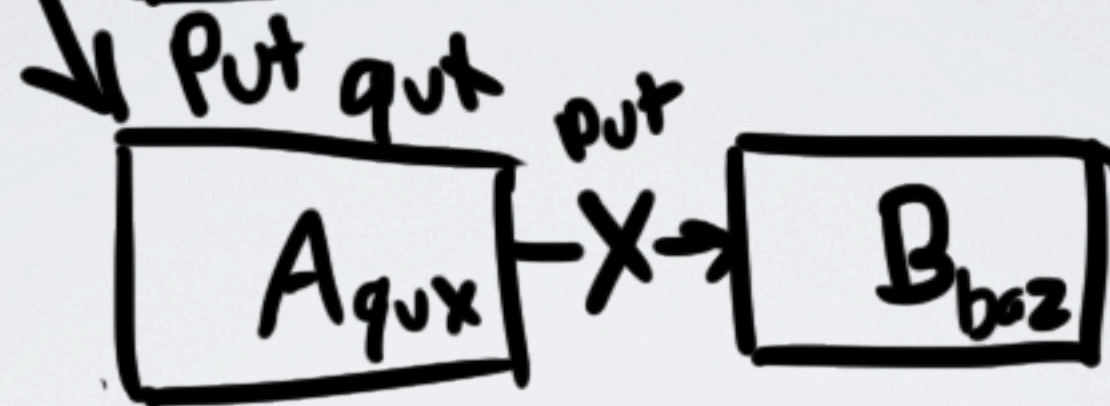




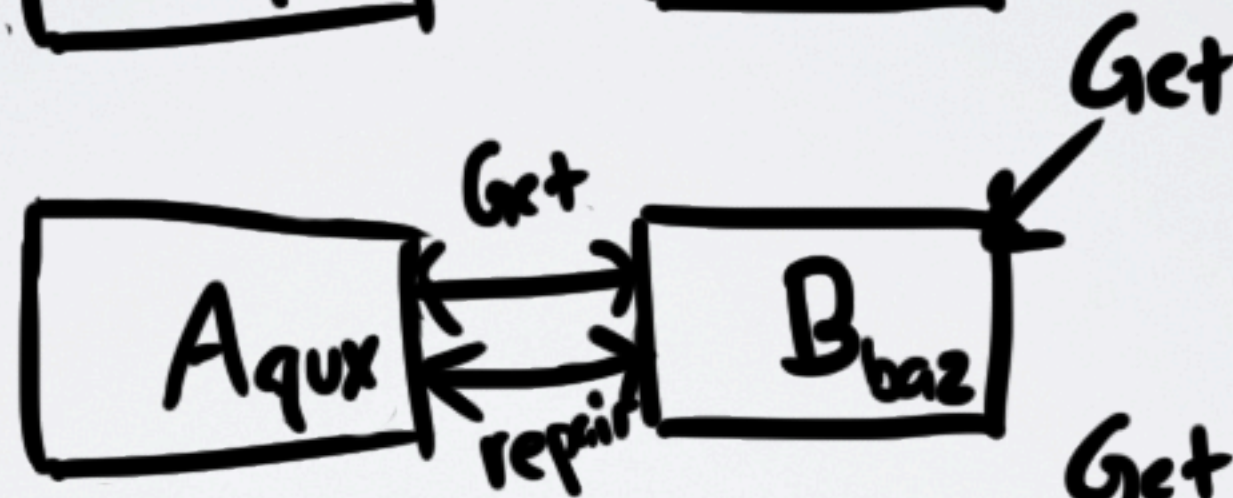
1.



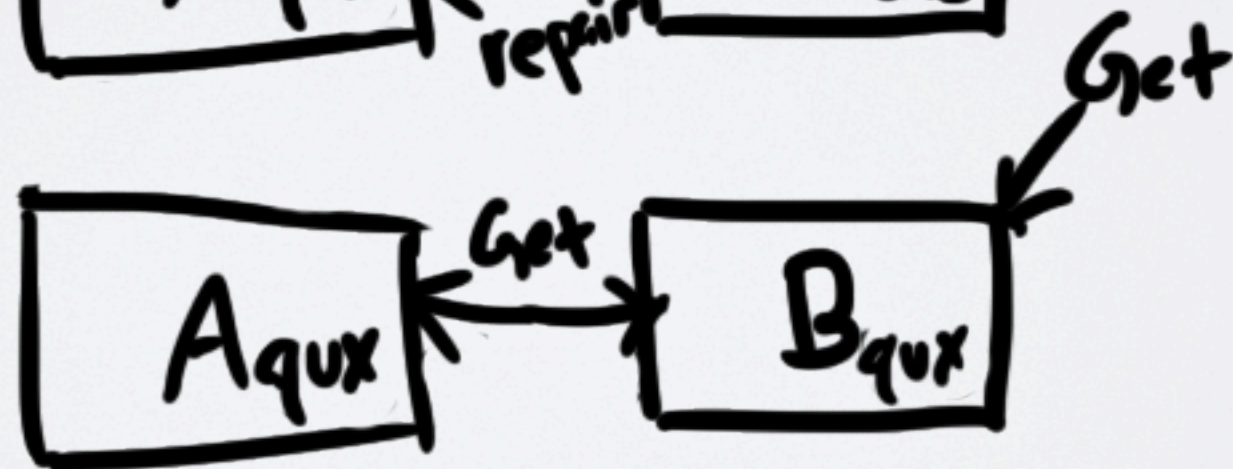
2.

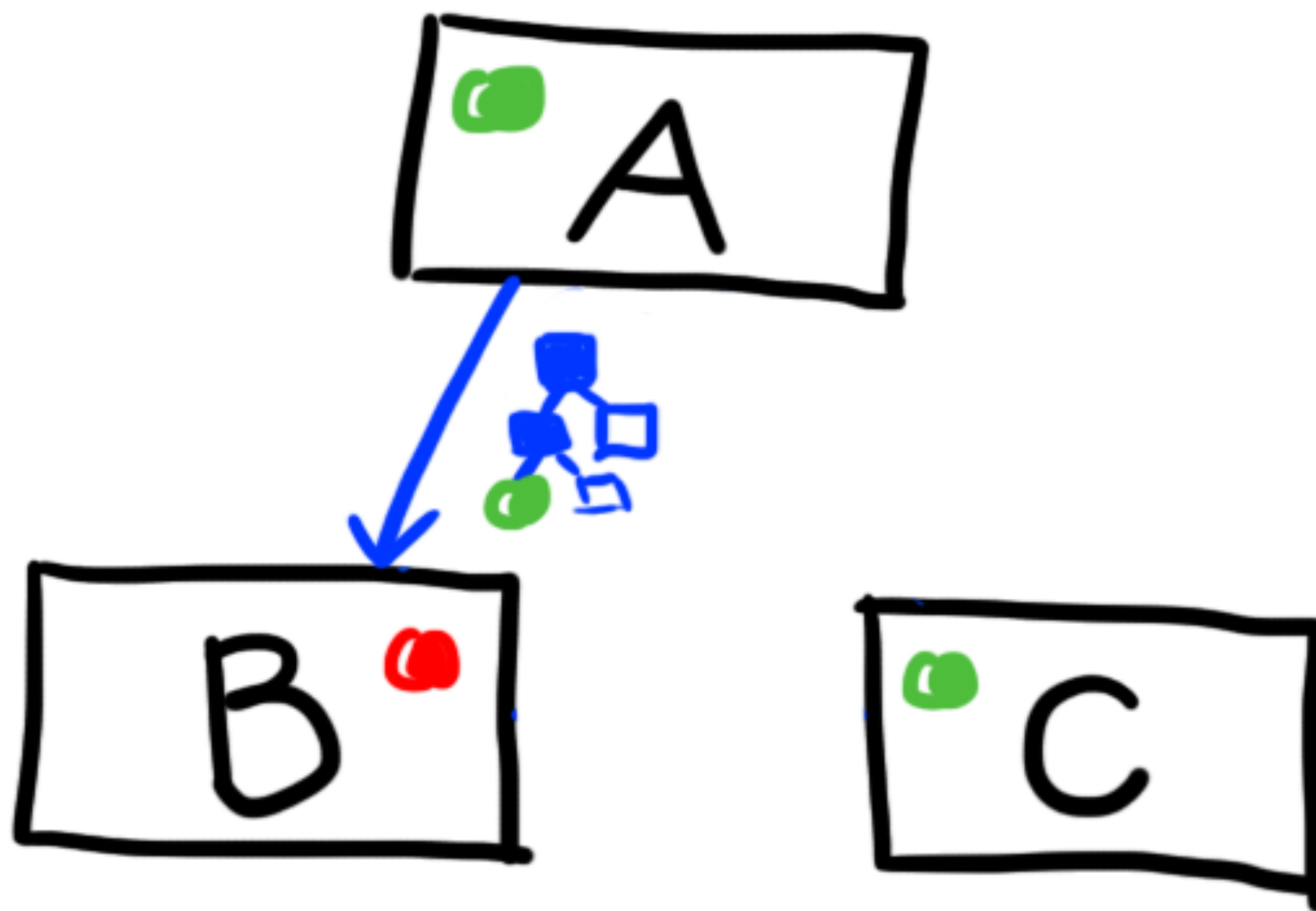


3.



4.



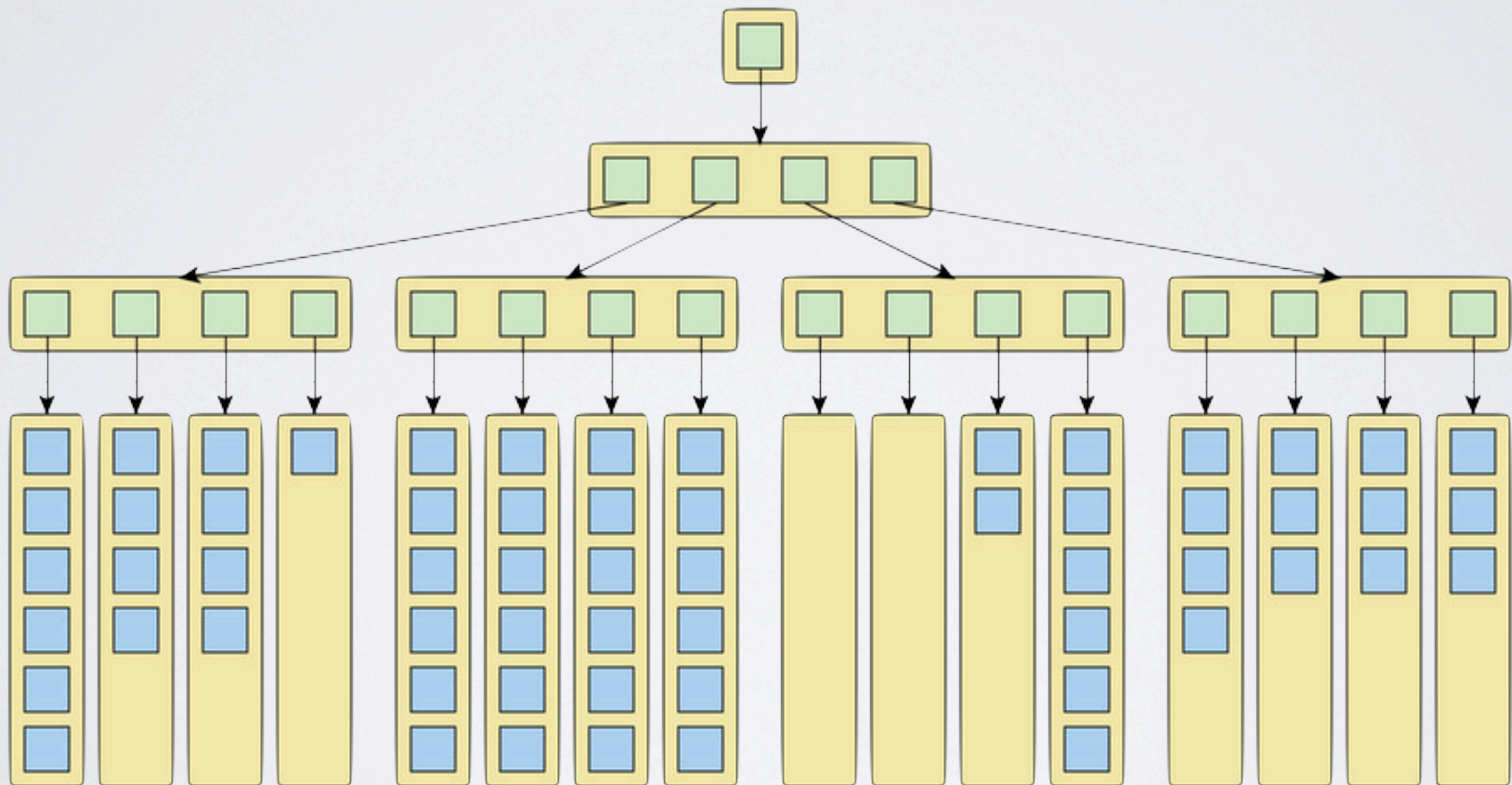






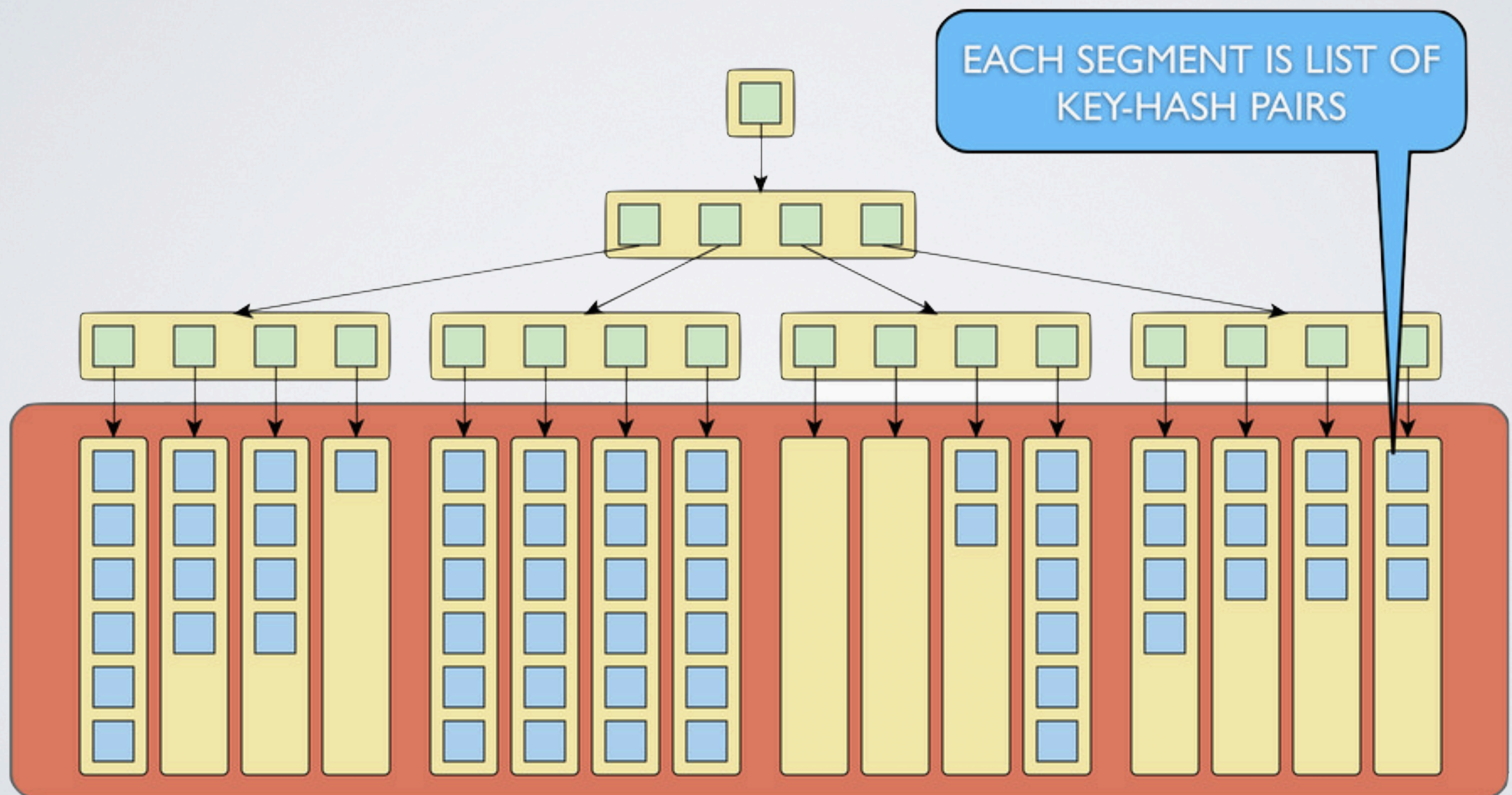
# MERKEL TREE

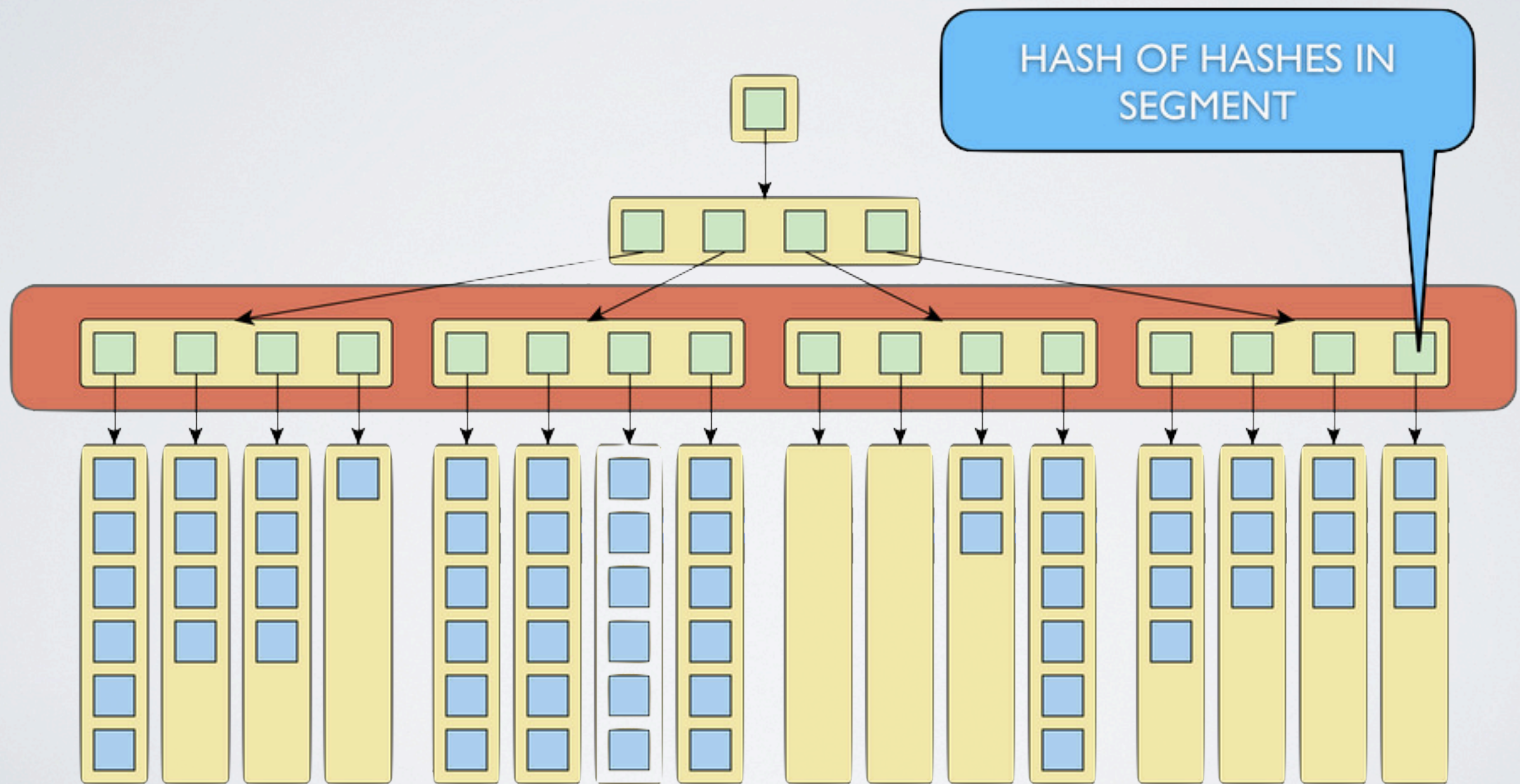




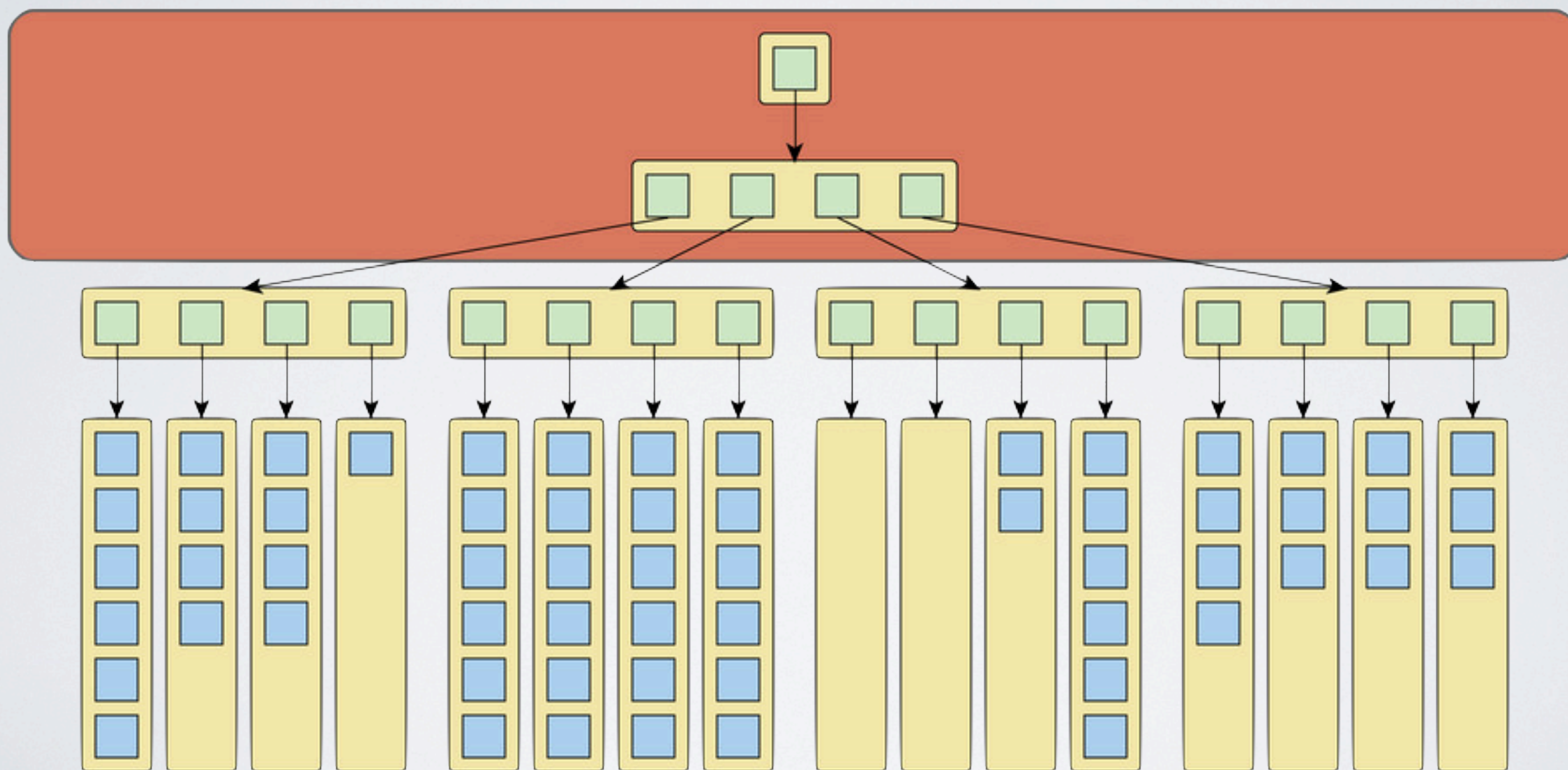
**\* Thanks Joe Blomstedt**

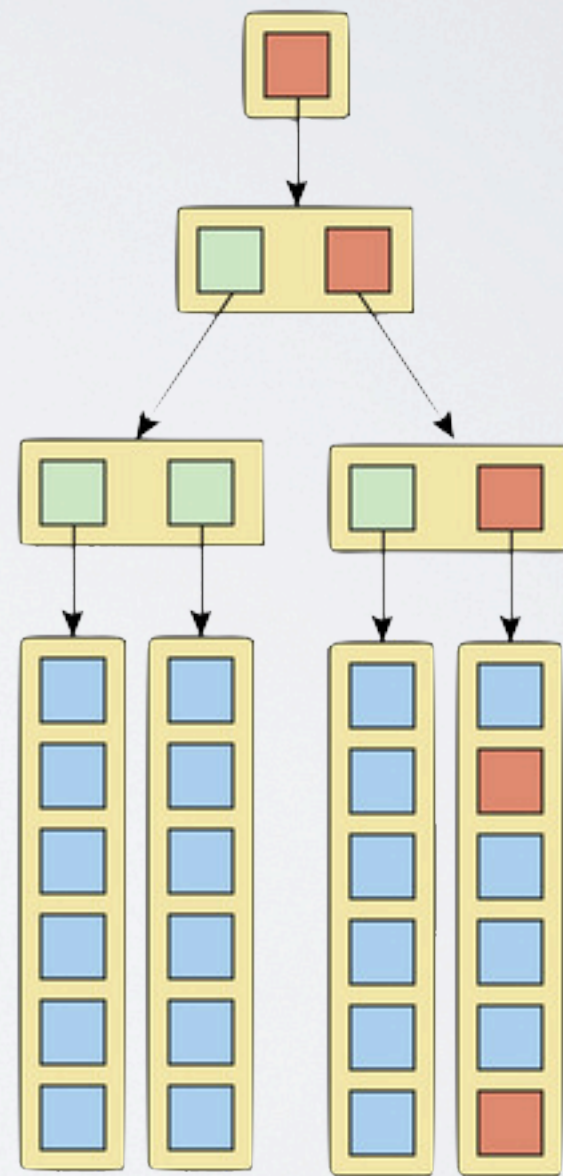
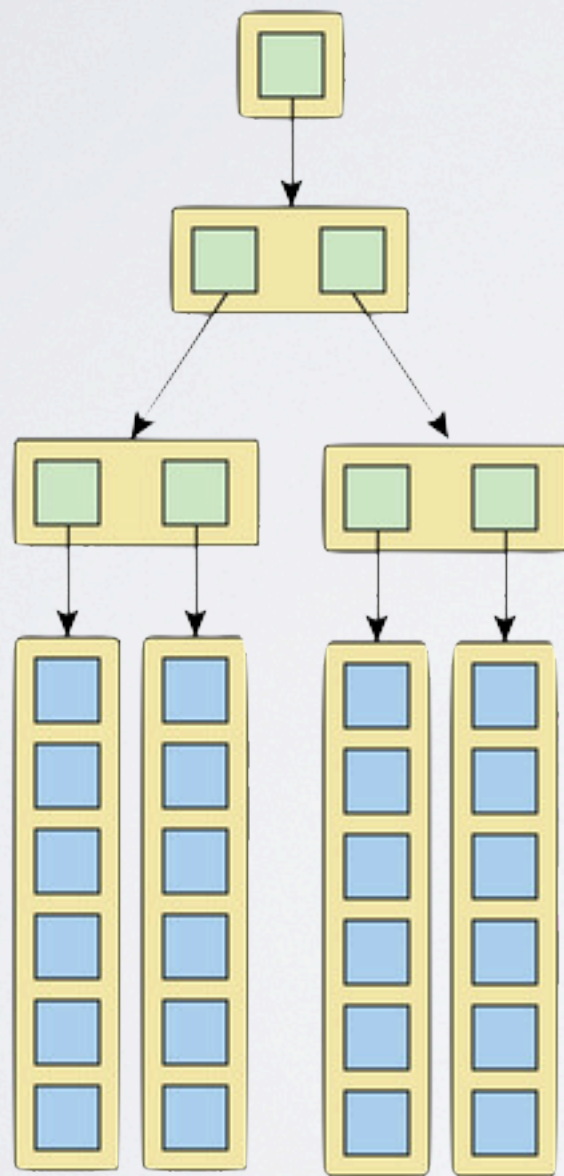




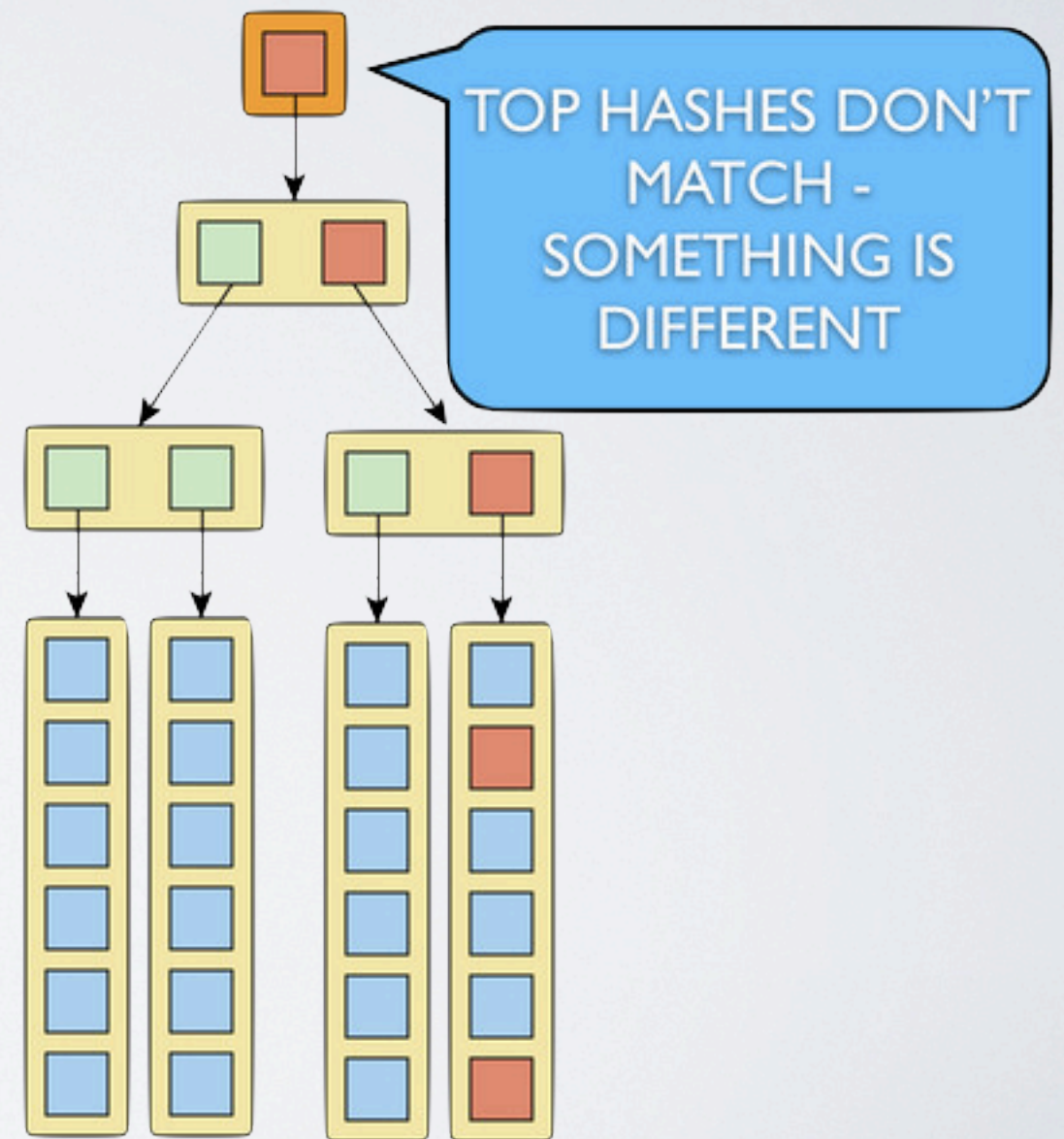
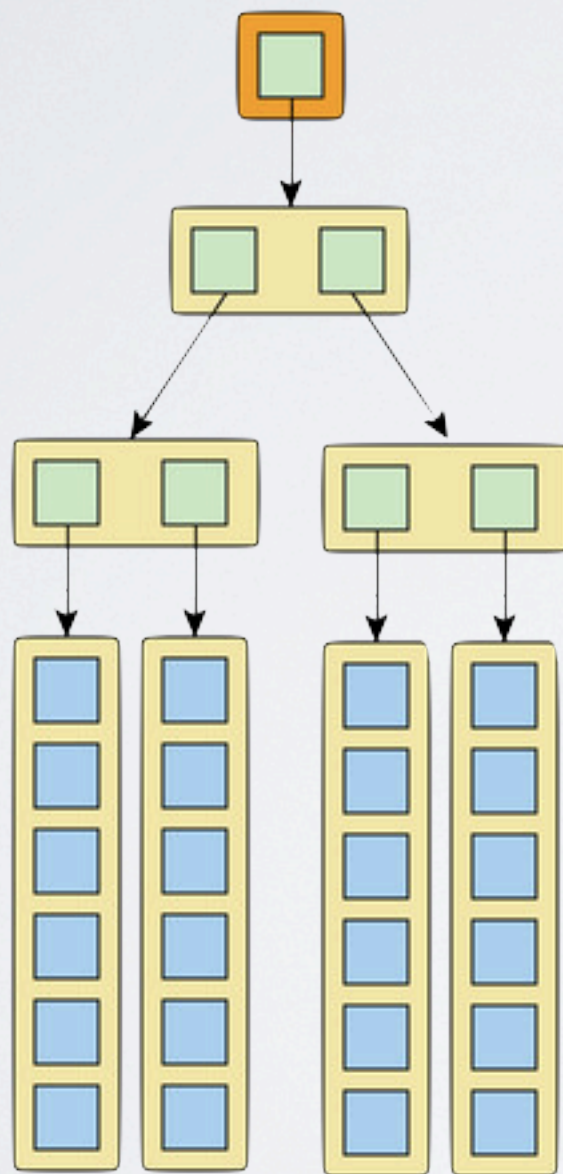




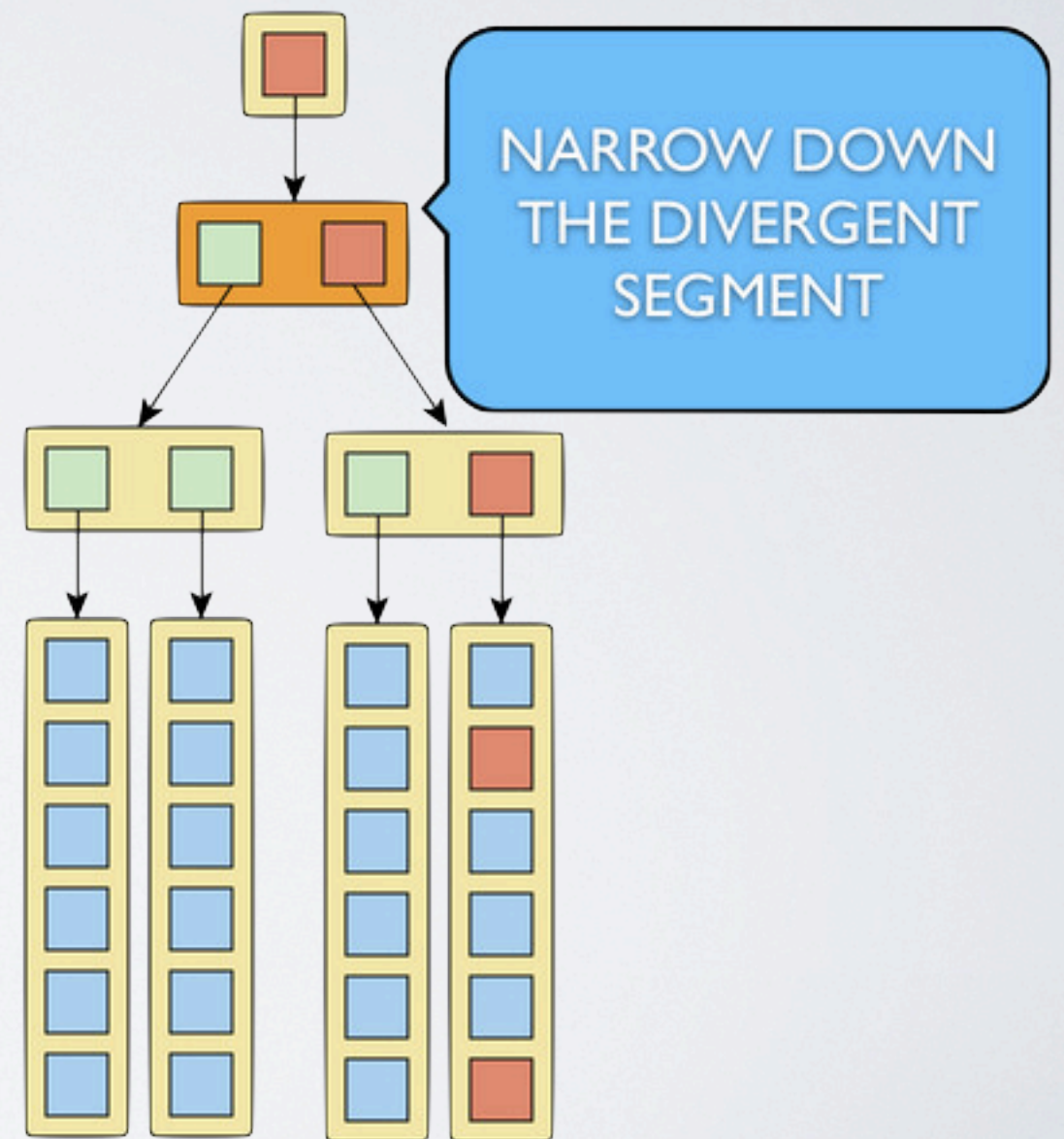
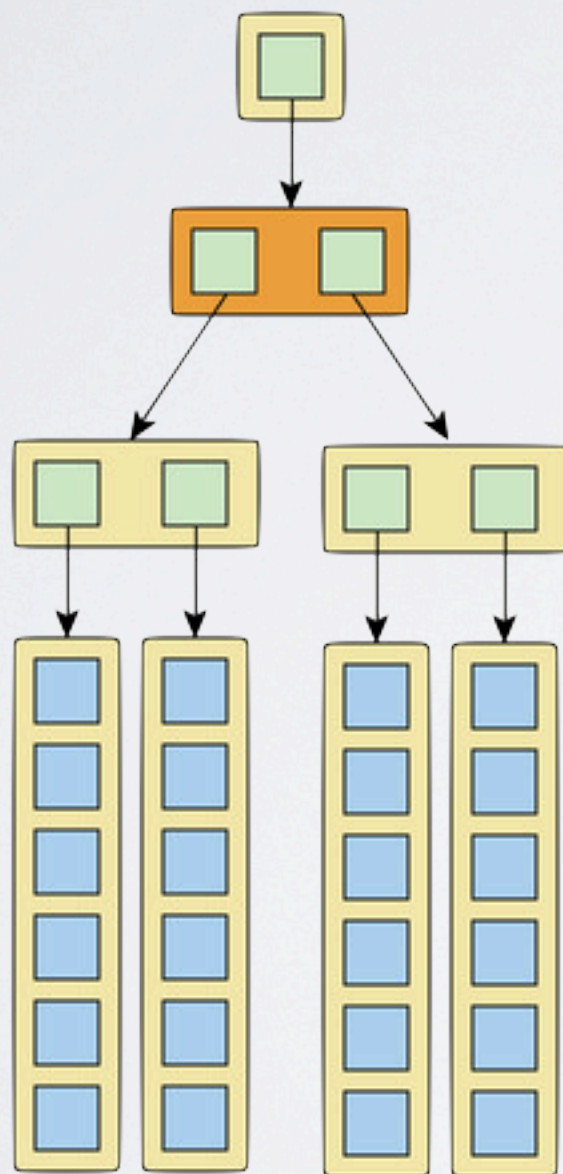




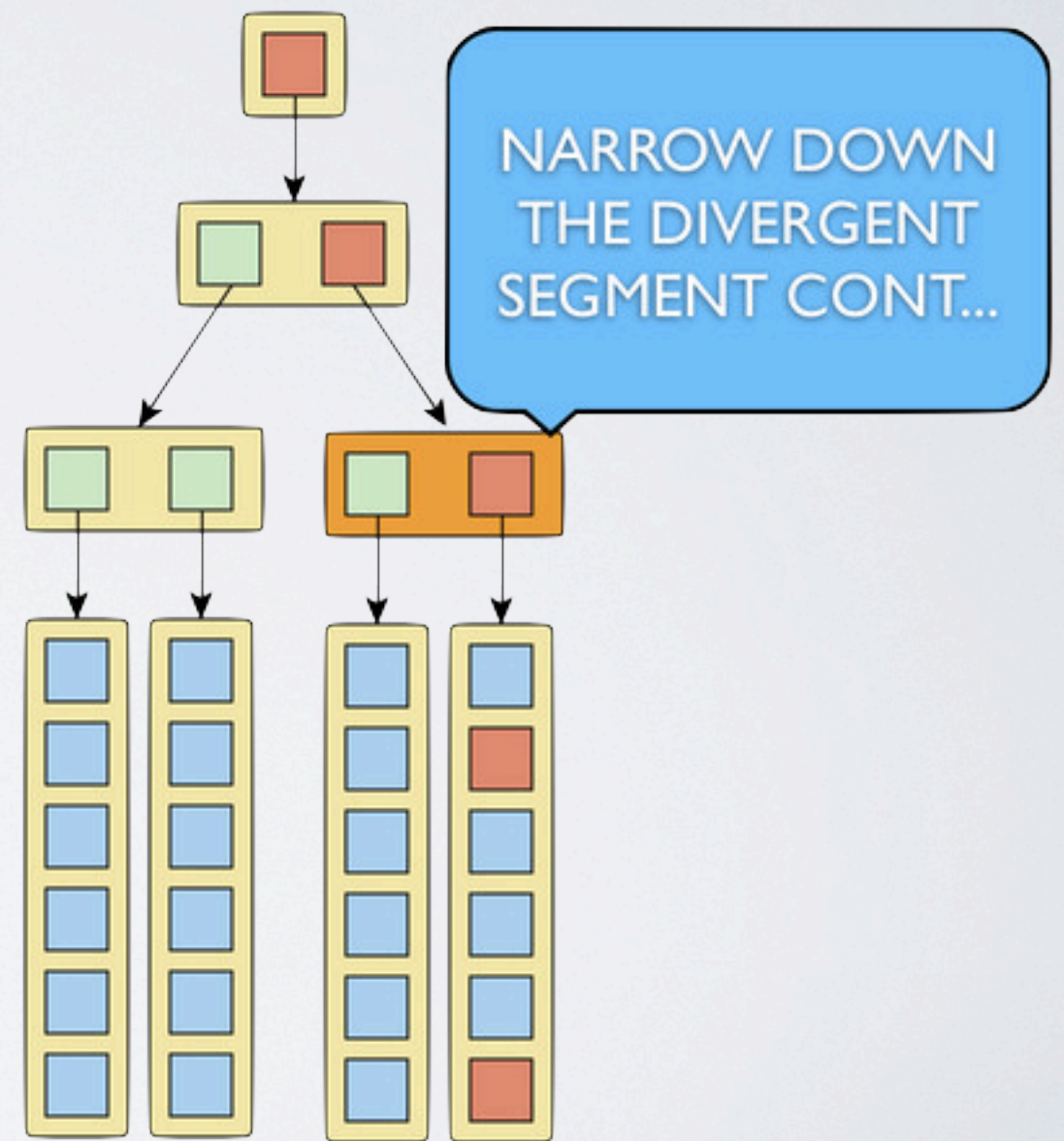
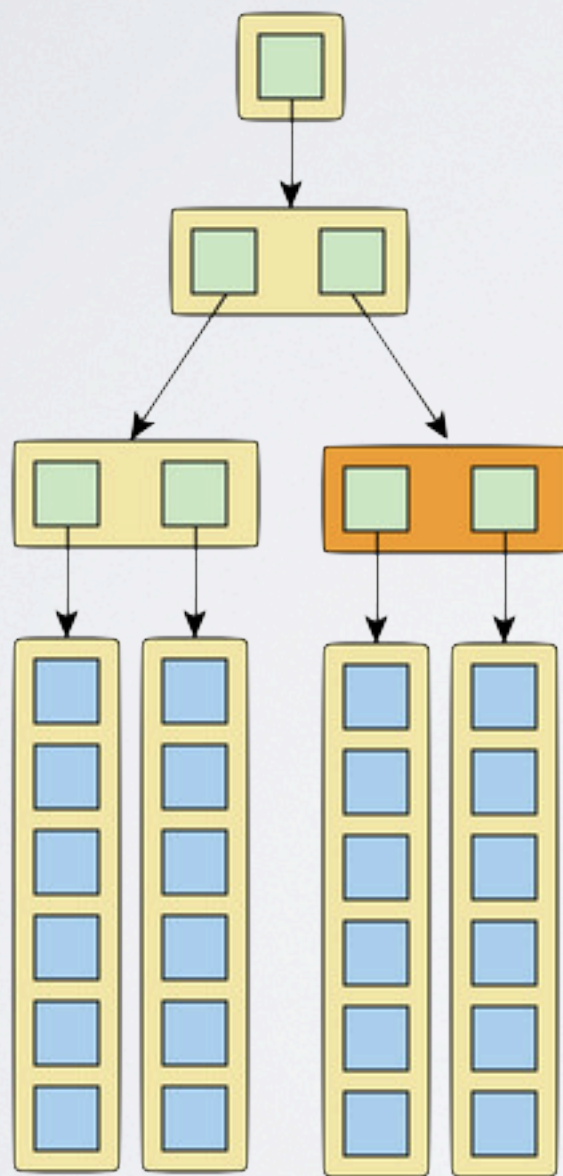




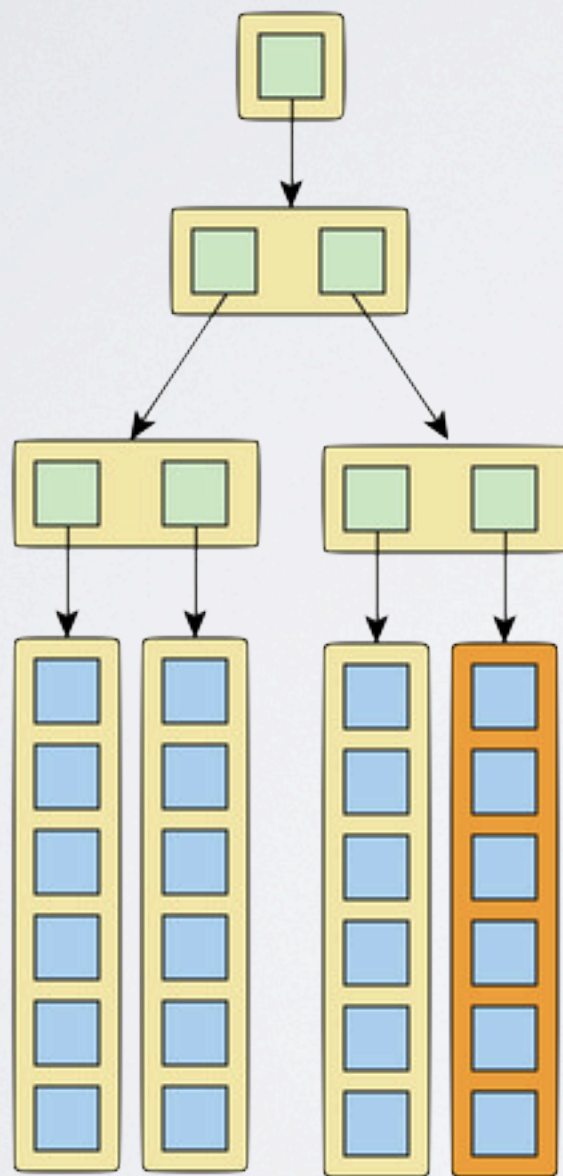




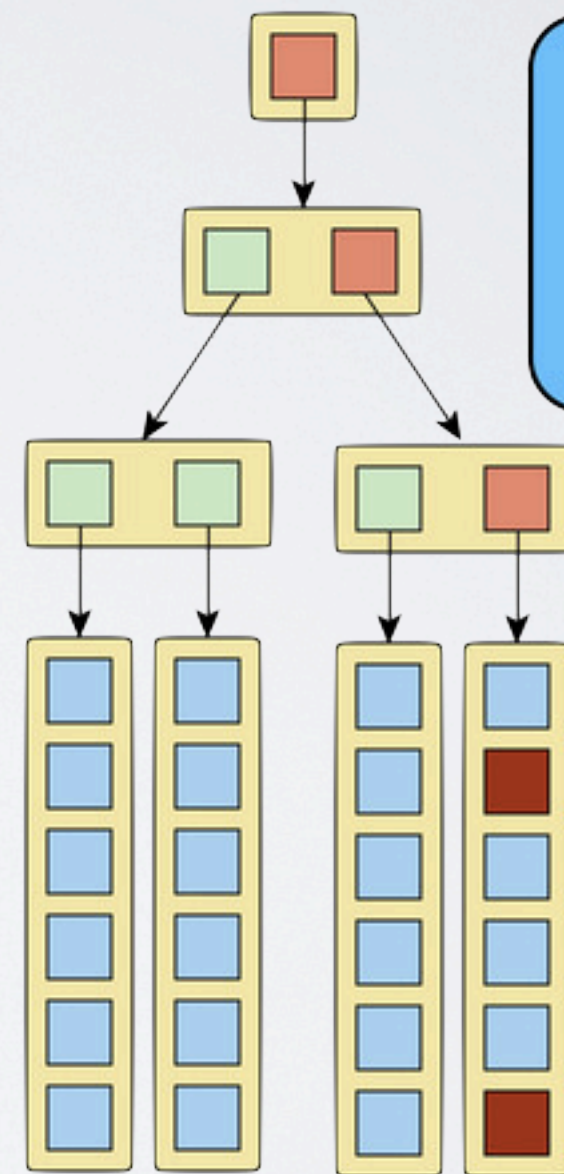
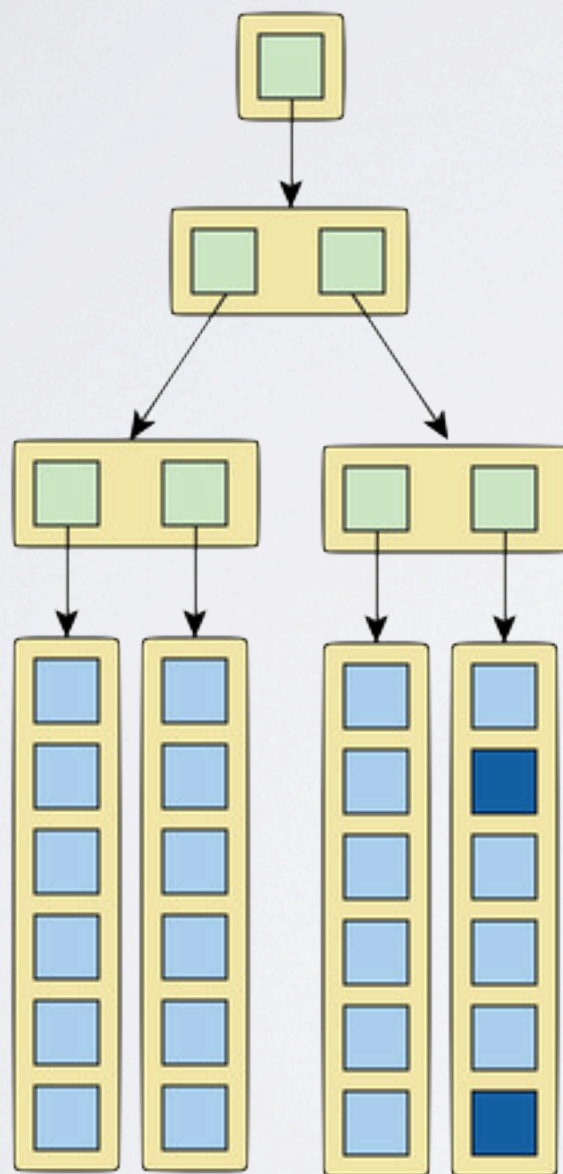




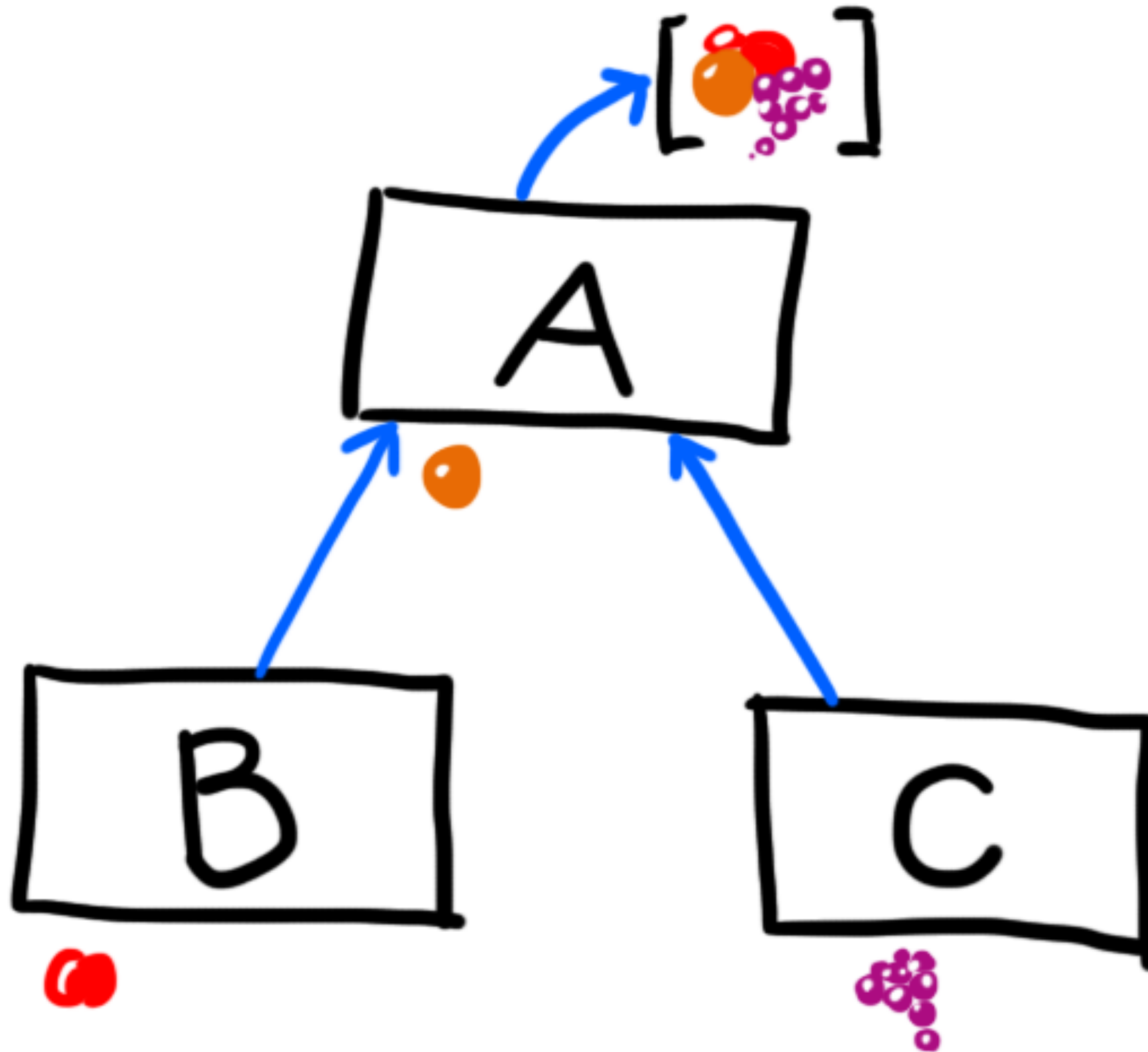








REPAIR (RE-INDEX)  
KEYS THAT ARE  
DIVERGENT (RED)



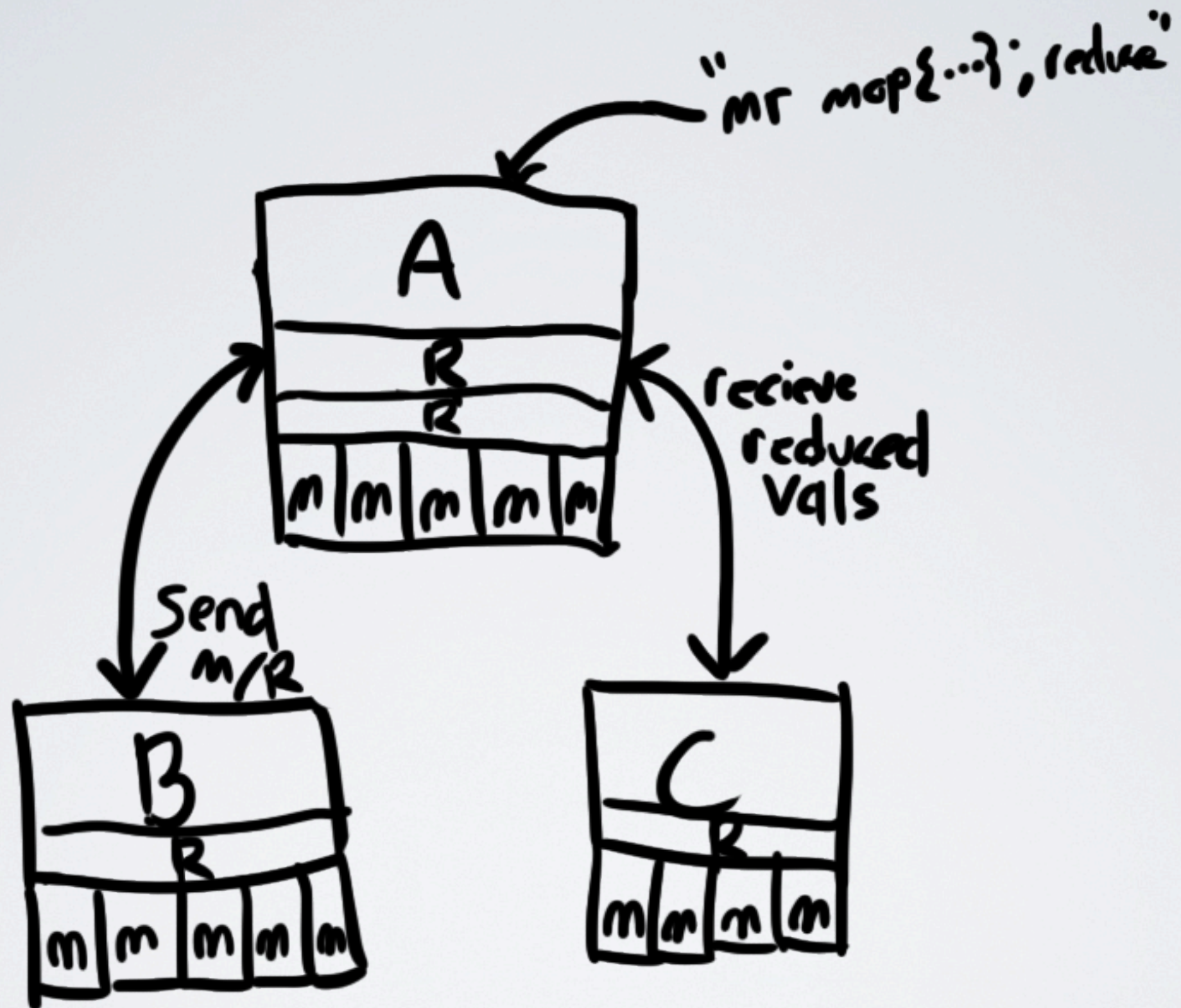


```
array = [{value:1},{value:3},{value:5}]
```

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

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







```
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
```



```
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
```



```
200.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
```

```
200.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
```



# ONE FINAL IMPROVEMENT

- **C!**
- **A!**
- **P!**



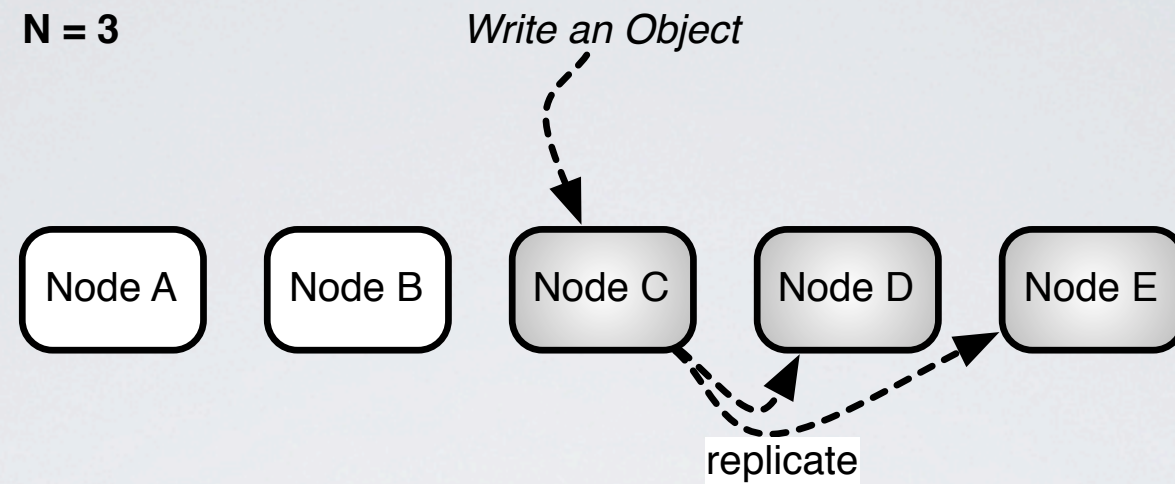


# N/R/W

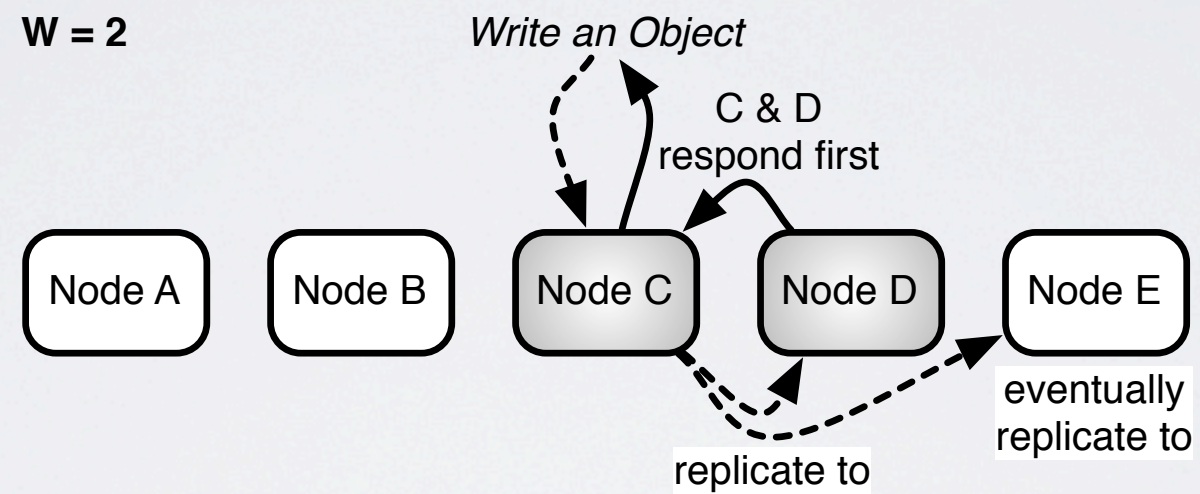
- **N!** # of **Nodes** to replicate a value to (in total)
- **R!** # of nodes to **Read** a value from (before success)
- **W!** # of nodes to **Write** a value to (before success)



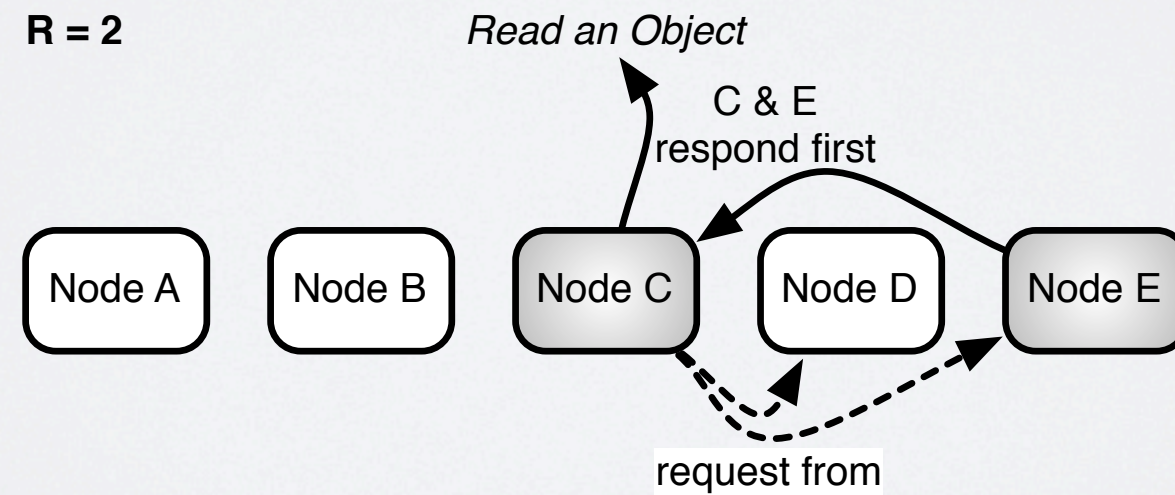
**N = 3**



**W = 2**



**R = 2**



# EVENTUALLY CONSISTENT

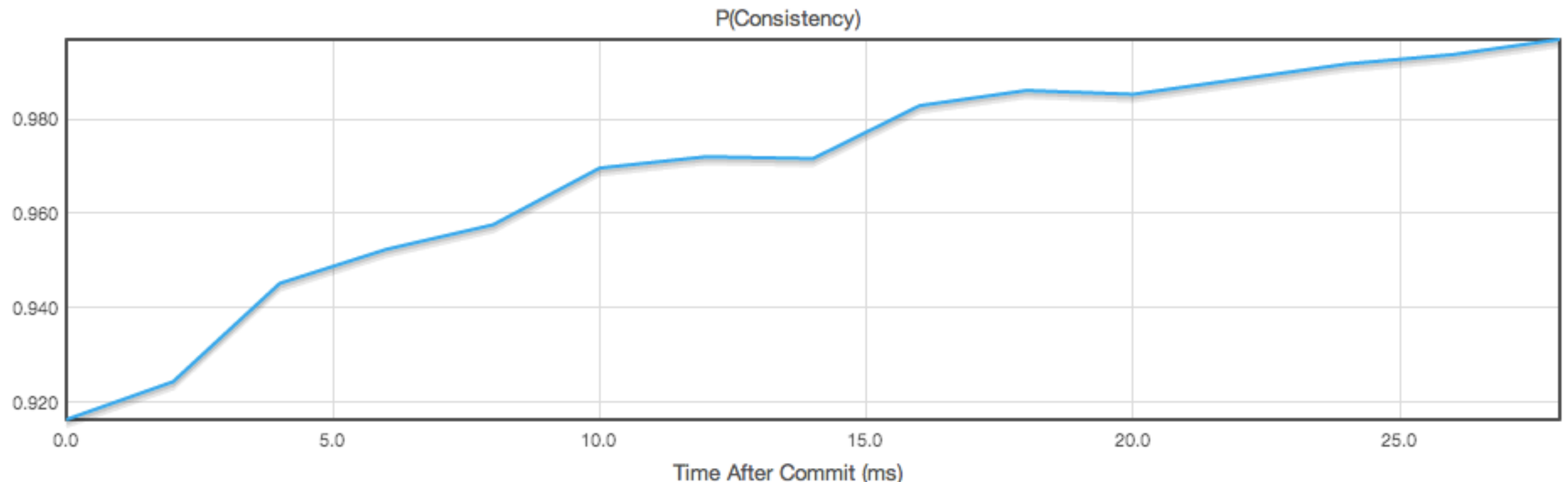
*Le mieux est l'ennemi du bien*

- How Eventual?
- How Consistent?



# Probabilistically Bounded Staleness

## $N=3, R=1, W=2$



(Plot isn't monotonically increasing? Increase the accuracy.)

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

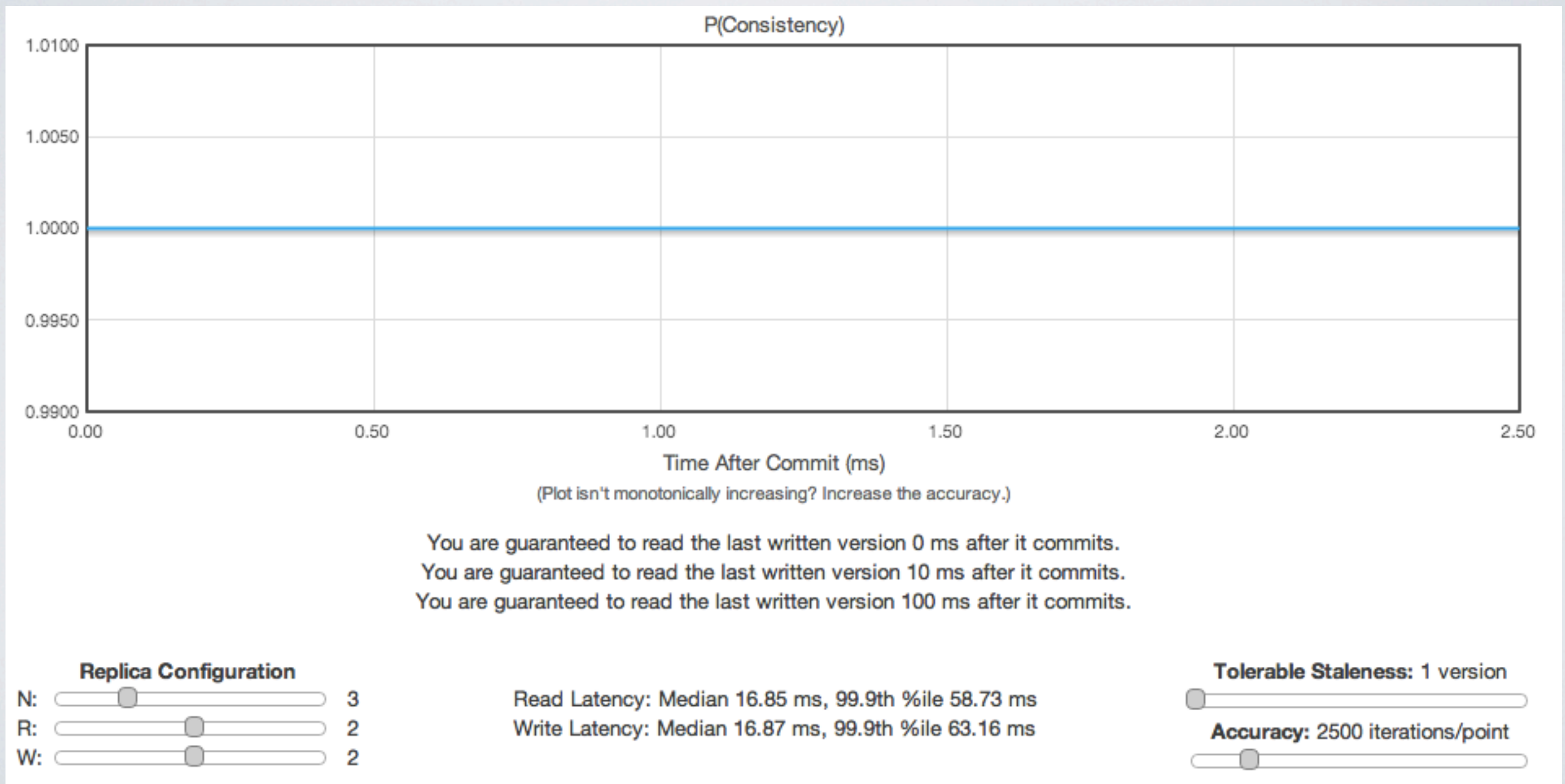
Read Latency: Median 8.47 ms, 99.9th %ile 36.45 ms  
Write Latency: Median 16.77 ms, 99.9th %ile 60.43 ms

Tolerable Staleness: 1 version

Accuracy: 2500 iterations/point

\* <http://pbs.cs.berkeley.edu>

$$N=3, R=2, W=2$$





A man with curly hair, wearing a flat cap, a light-colored shirt, a brown bow tie, and suspenders, is holding a piece of paper with a diagram. He has a surprised or concerned expression. The background is dark with a large, glowing circular light source in the upper left.

<http://git.io/MYrjpQ>

**YOU KNOW,  
FOR DATA**

\* thanks JD

Preference List

Key/Value

Vector Clocks

Distributed Hash Ring

Request/  
Response



Merkel Tree

Node Gossip

CRDT (counters, more coming)

Read Repair





basho

@coderoshi

The  
Pragmatic  
Programmers

## Seven Databases in Seven Weeks

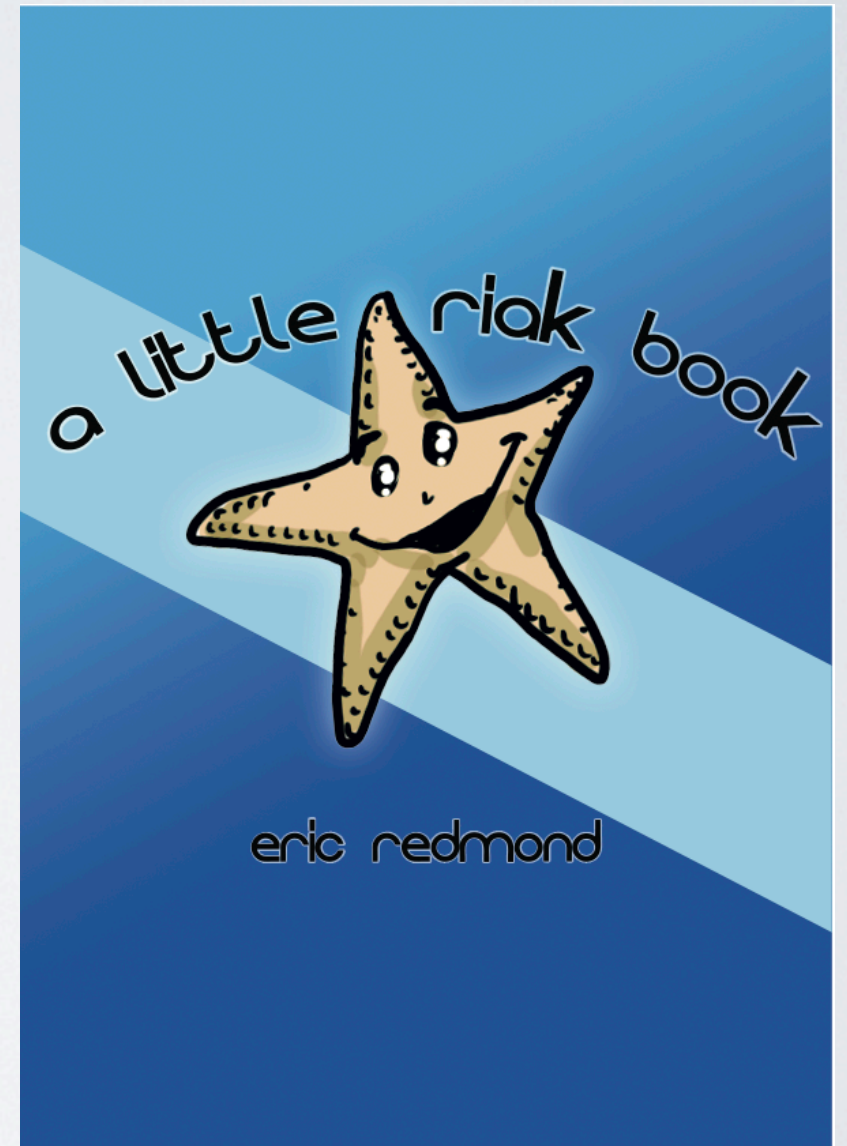
A Guide to Modern Databases  
and the NoSQL Movement

Eric Redmond  
and Jim R. Wilson

*Edited by Jacquelyn Carter*



<http://pragprog.com/book/rwdata>



<http://littleriakbook.com>