Introduction to riak_ensemble

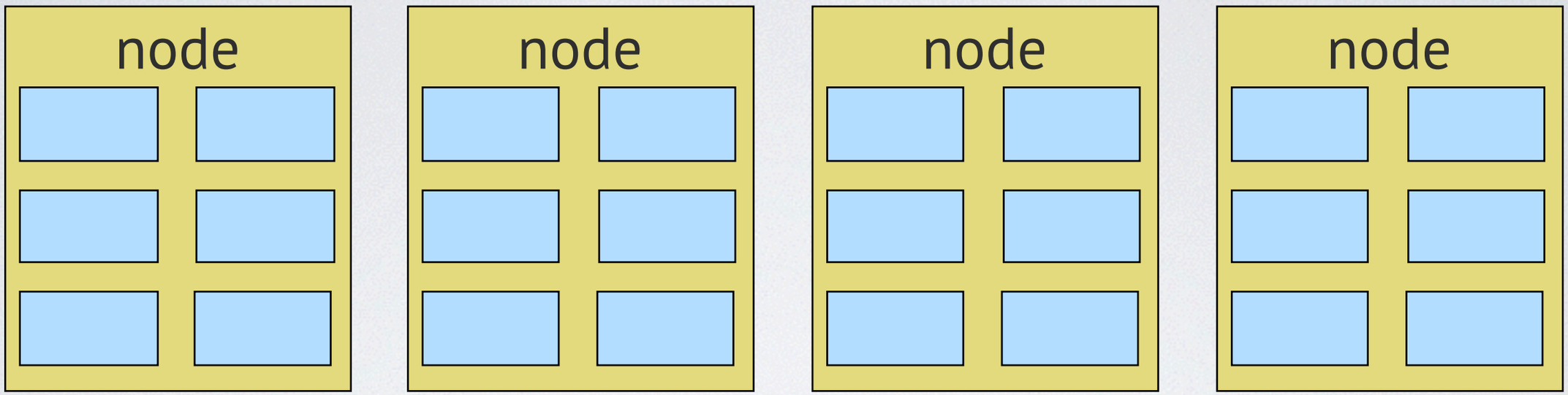
Joseph Blomstedt (@jtuple)
Basho Technologies

basho

riak

riak_ensemble

Paxos framework for scalable consistent system


node
node
node
node
node
node
node
node

What about state?

App

App

App

App

Database

App App App App

Riak Riak Riak Riak

Riak Riak Riak Riak

What if I'm

writing

a database?

What about

embedded state?

Mnesia!

{inconsistent_database,
running_partitioned_network}

CAP Theorem

basho

riak

Consistency

Availability

Partition-tolerance

basho

riak

Consistency

Availability

Partition-tolerance

basho

riak

CP

AP

Consistency

Availability

Partition-tolerance

basho

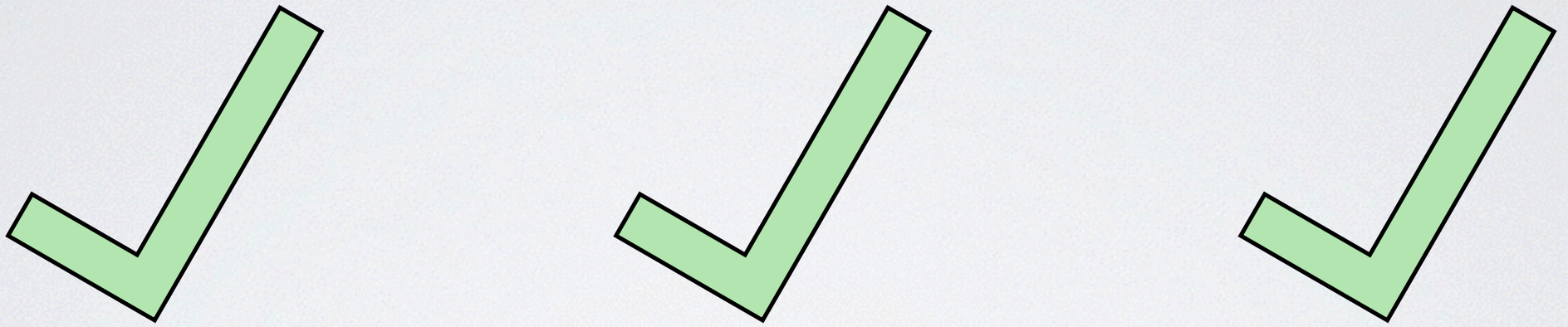
riak

Node 1 Node 2 Node 3 Node 4 Node 5

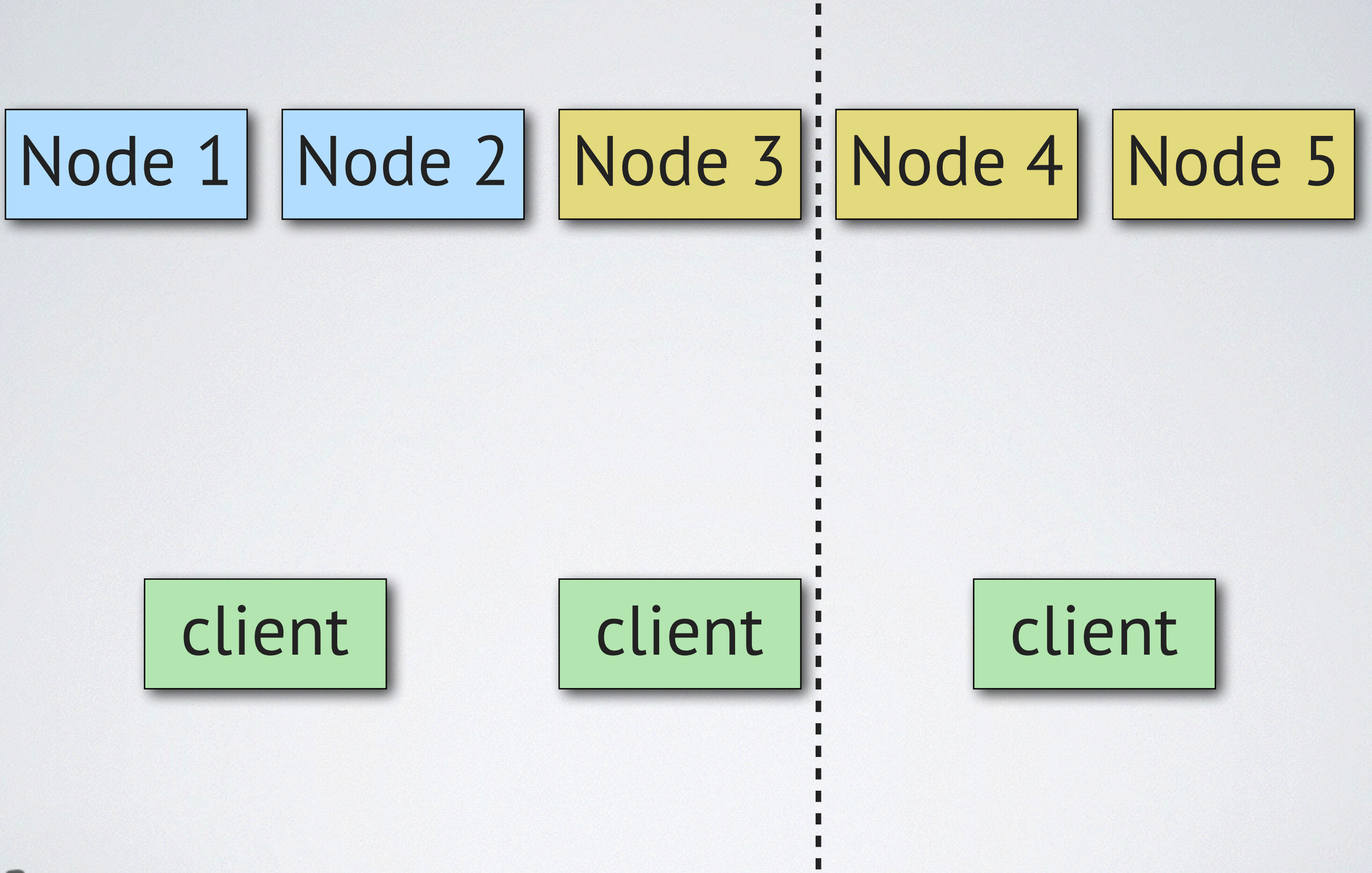
client client client

basho riak

Node 1 | Node 2 | Node 3 | Node 4 | Node 5



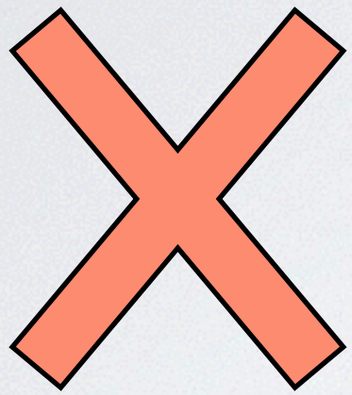
client | client | client

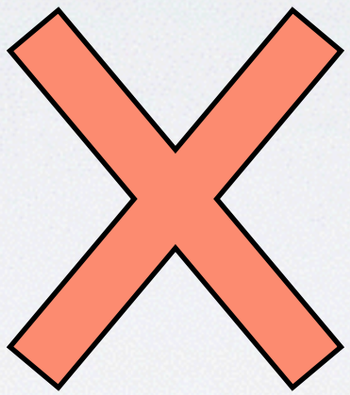


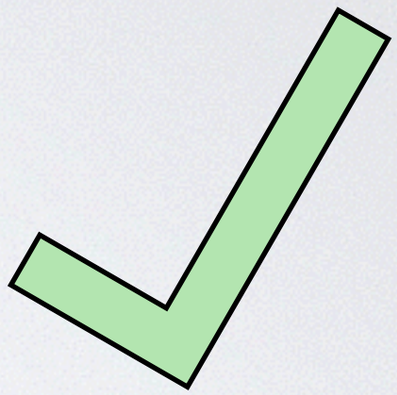
basho

riak

Node 1 | Node 2 | Node 3 | Node 4 | Node 5





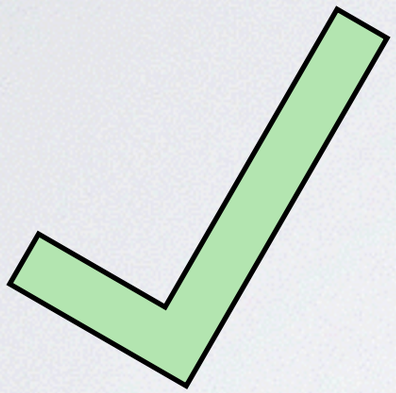


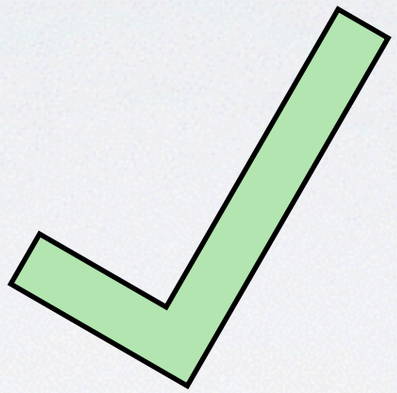
client | client | client

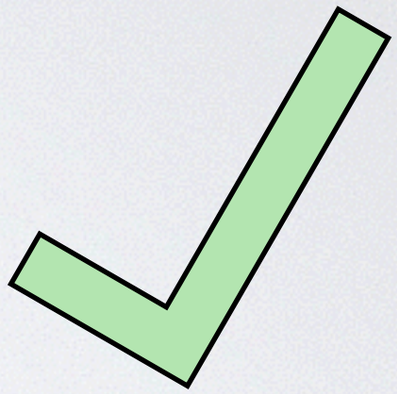
basho

riak

Node 1 | Node 2 | Node 3 | Node 4 | Node 5





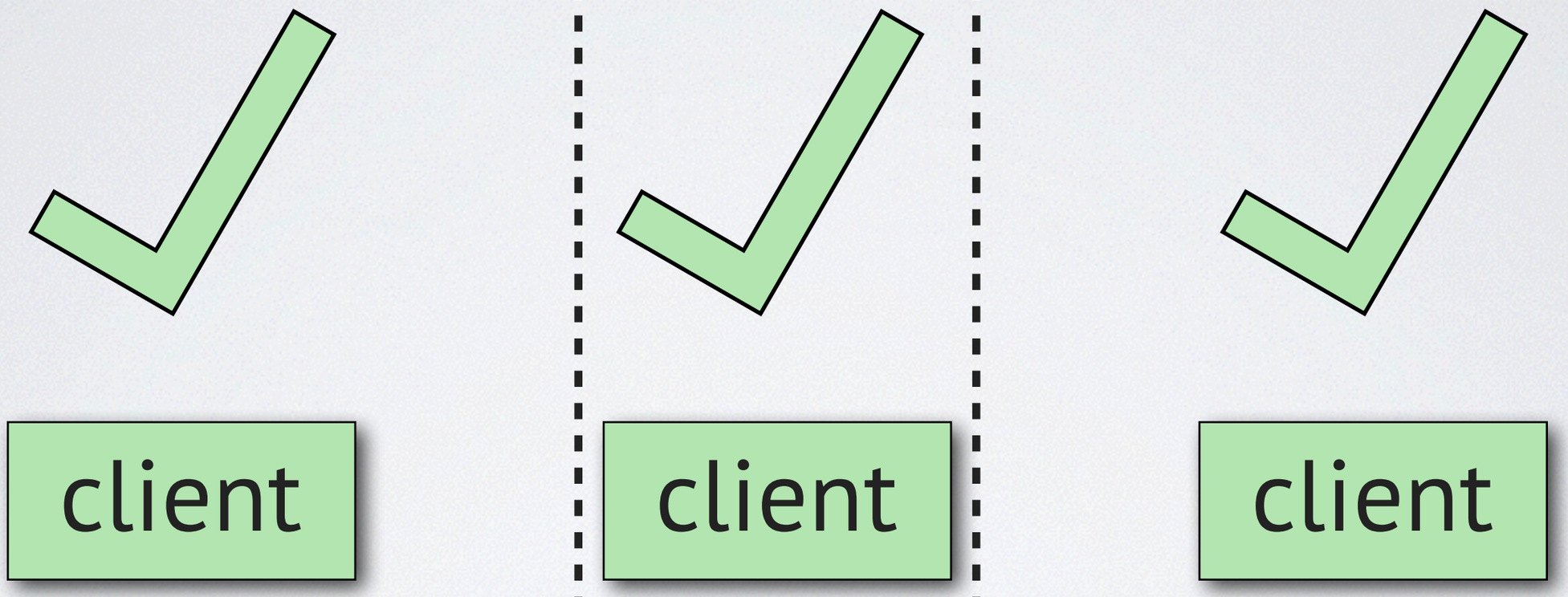


client | client | client

basho

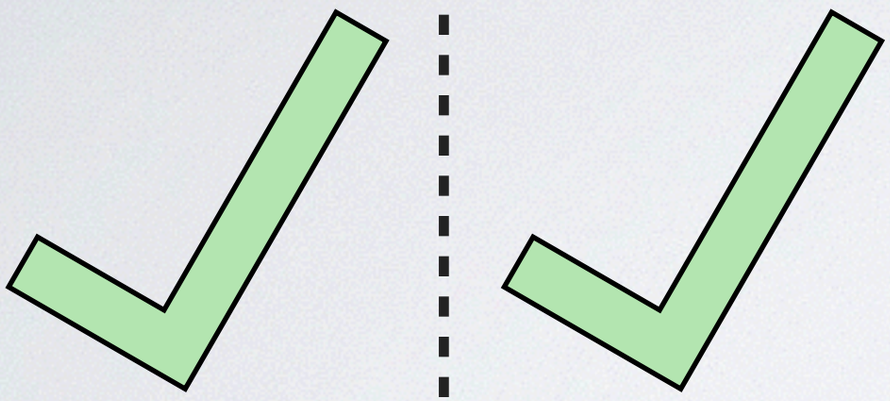
riak

Node 1 | Node 2 | Node 3 | Node 4 | Node 5



client | client | client

Node 1 | Node 2 | Node 3 | Node 4 | Node 5

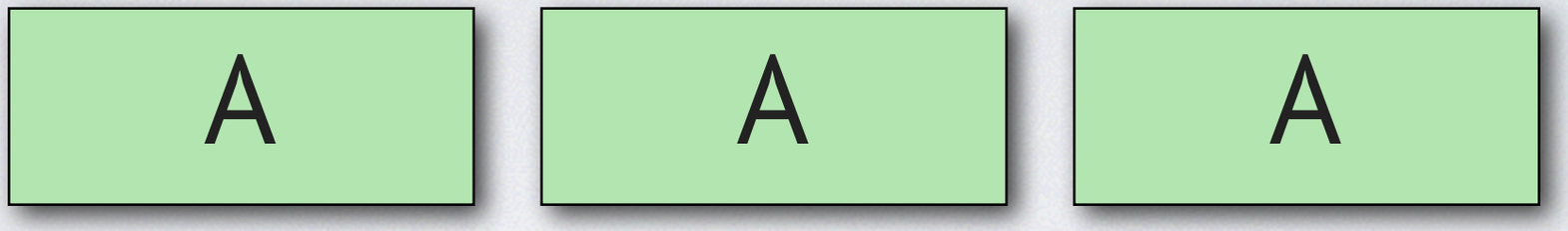


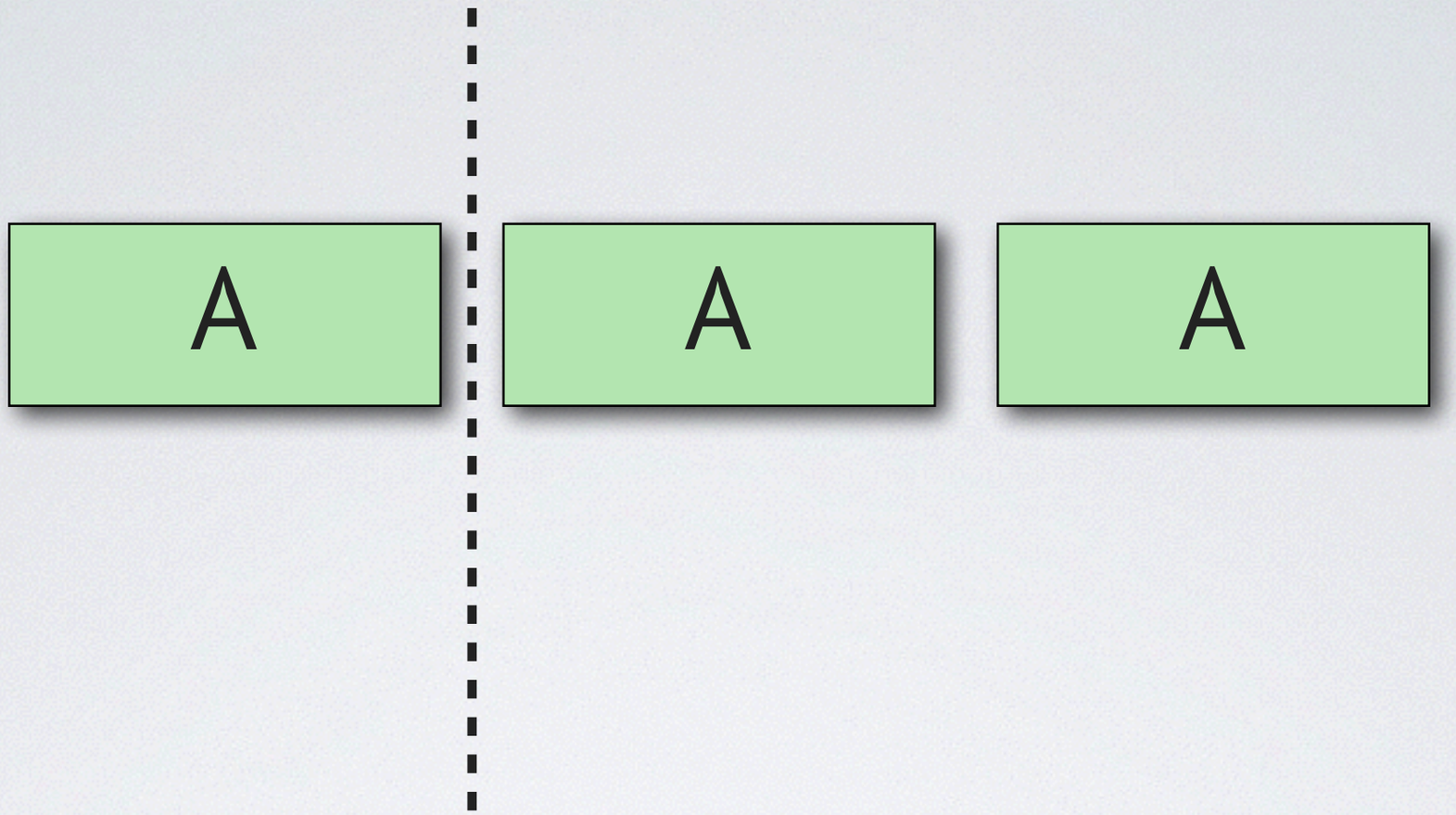
client | client | client | client | client

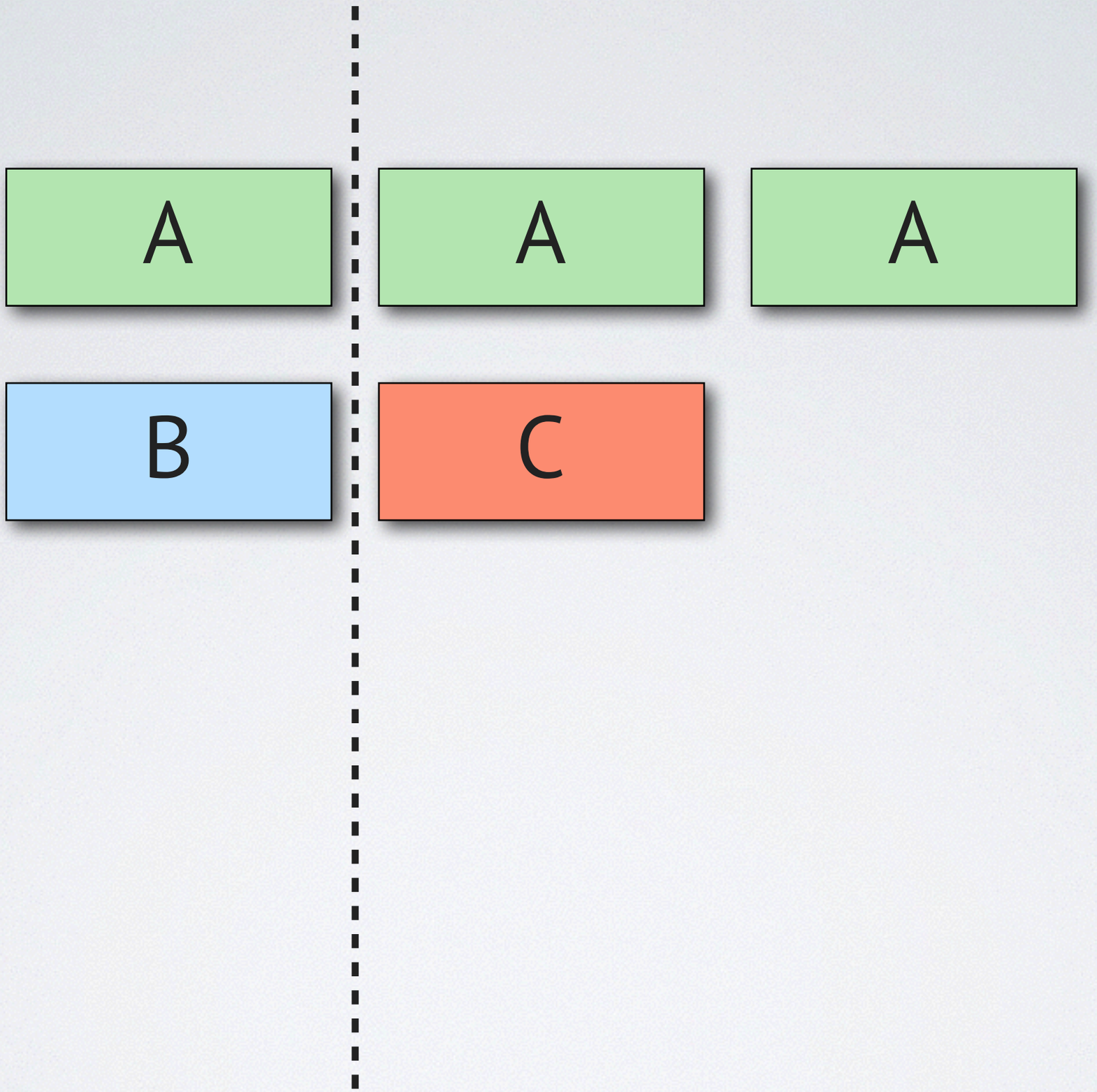
basho

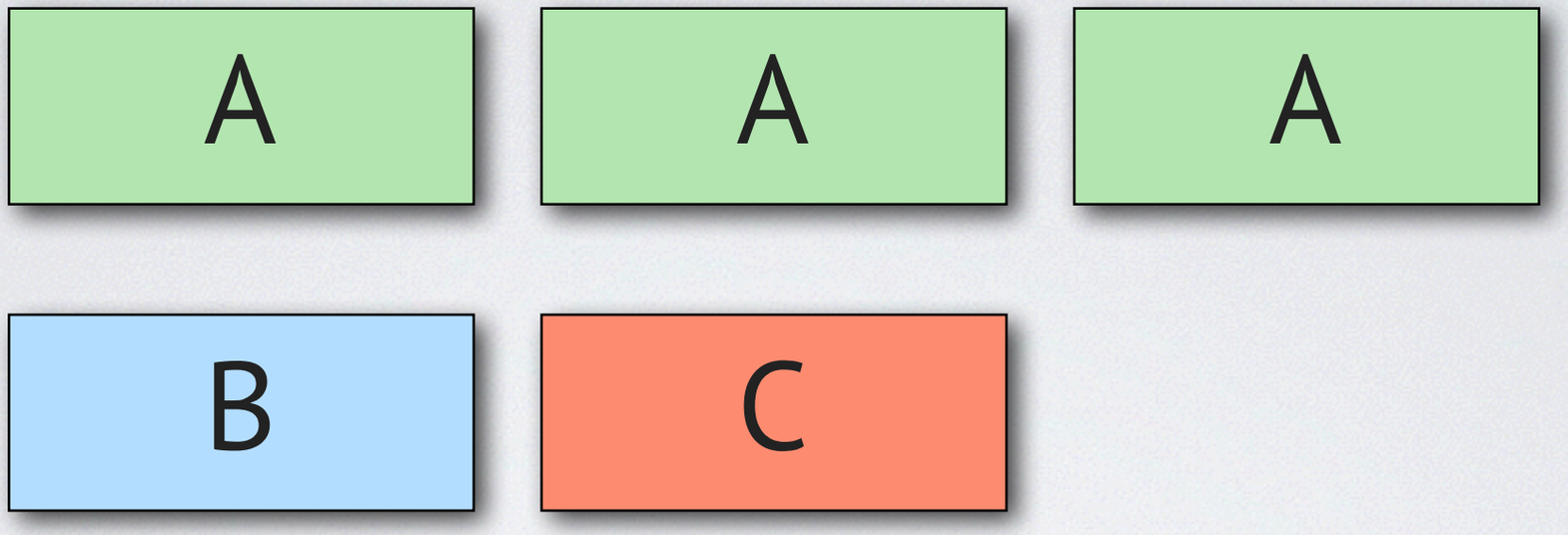
riak

Eventual Consistency


A
A
A


A
A
A


A
A
A
B
C


A
A
A
B
C

A A A

B C

{B,C} {B,C} {B,C}

Write Once
Immutable
Last Write Wins
Business Rules
Sets/Counters/Maps

Consensus

quorum consensus
chain replication
virtual synchrony

quorum consensus

chain replication

virtual synchrony

basho

riak

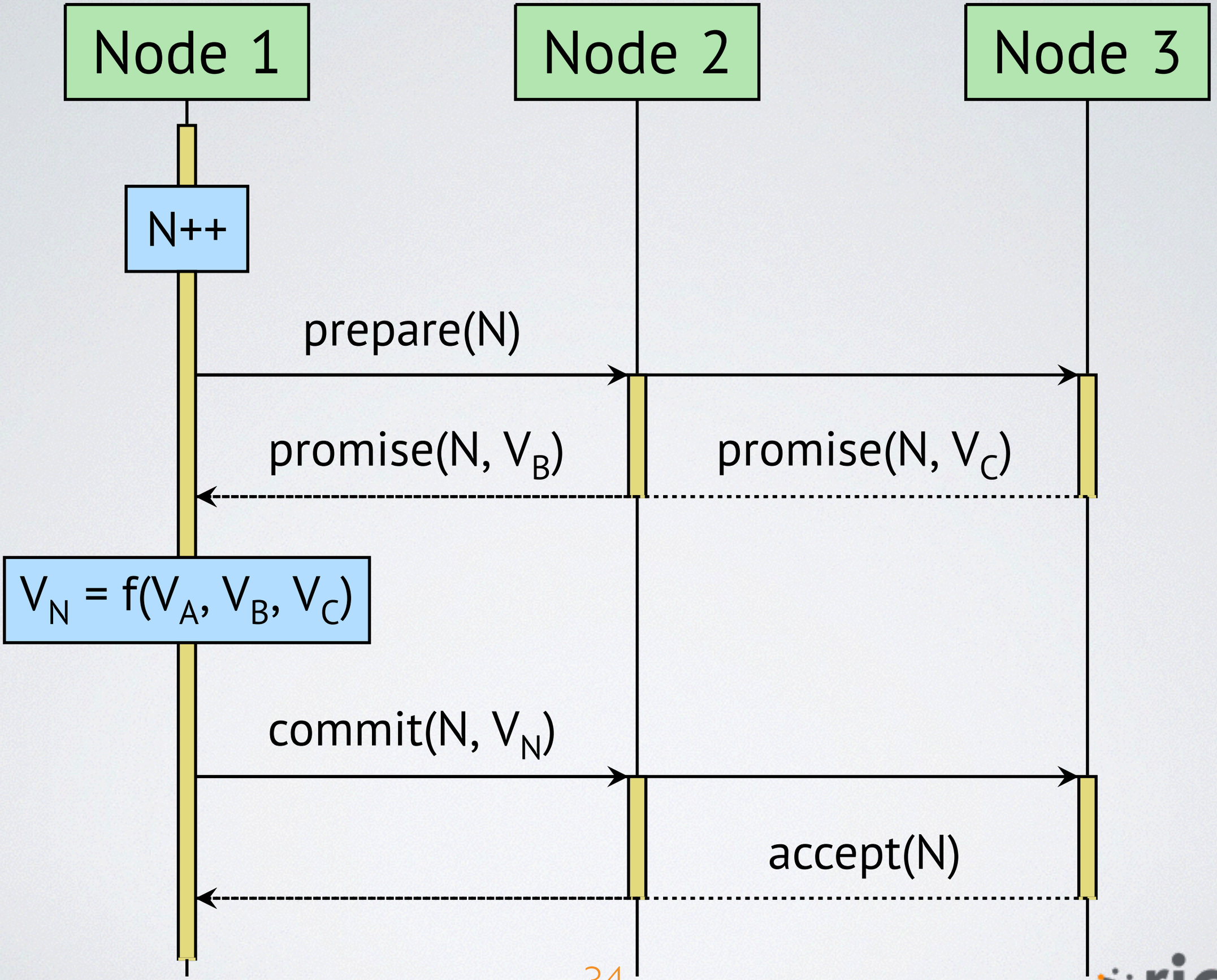
Quorum Consensus

Paxos

ZK Atomic Broadcast

Raft

Paxos


Node 1
Node 2
Node 3
N++
prepare(N)
promise(N, V_B)
promise(N, V_C)
$V_N = f(V_A, V_B, V_C)$
commit(N, V_N)
accept(N)

basho

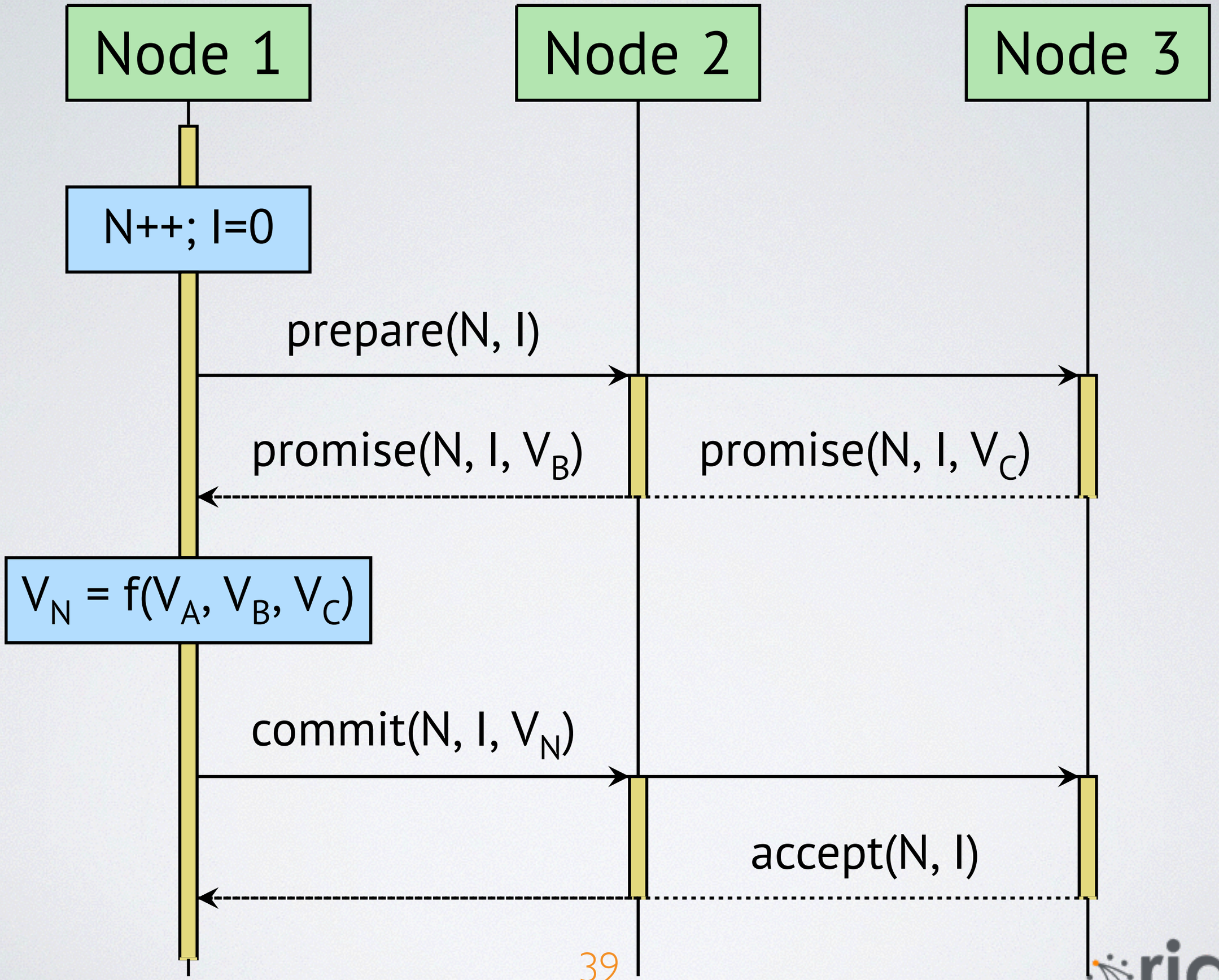
riak

Rinse/repeat for each request

2 round trips/request

Multi-Paxos

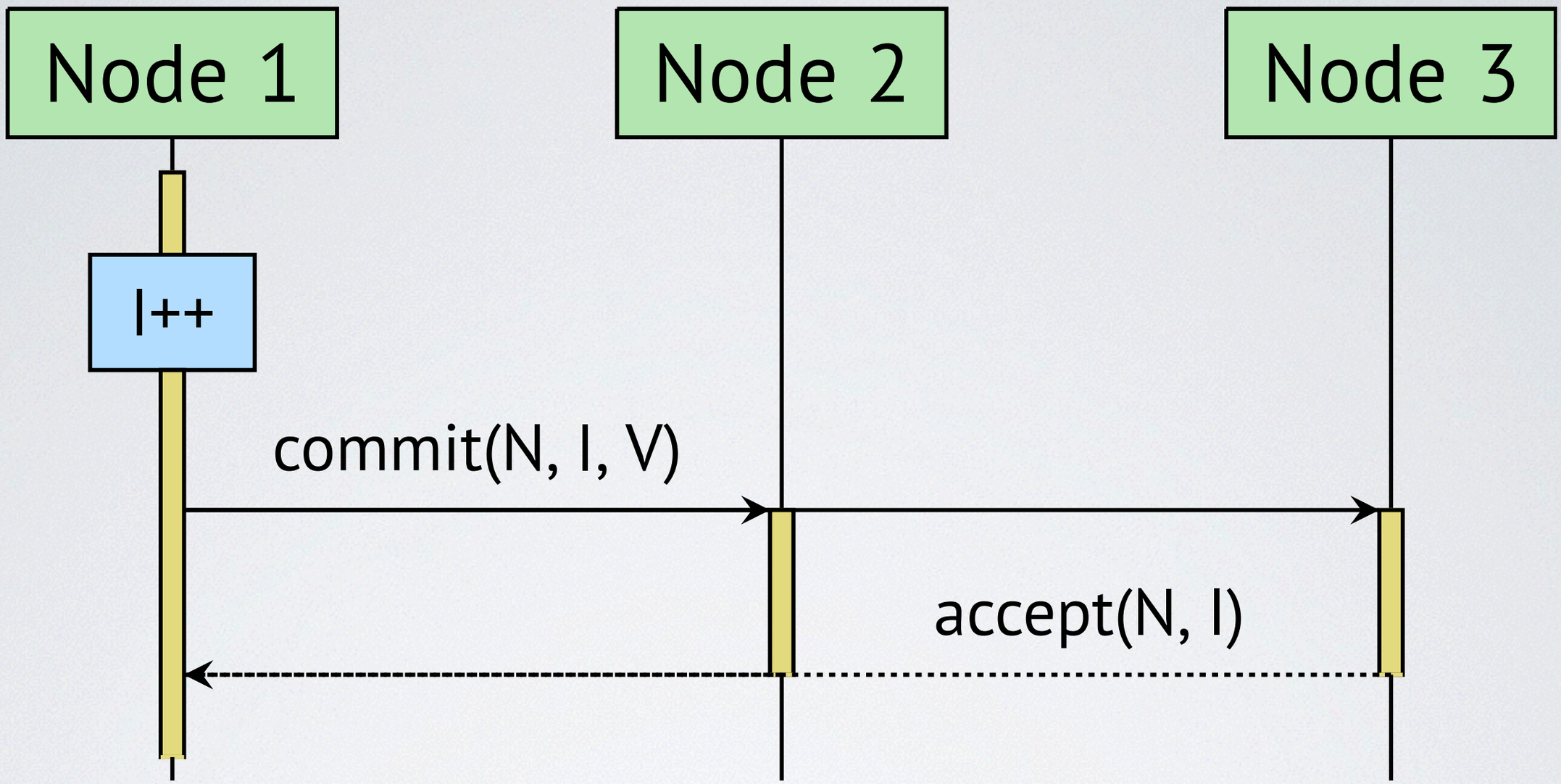
First Request


Node 1
Node 2
Node 3
N++; I=0
prepare(N, I)
promise(N, I, V_B)
promise(N, I, V_C)
$V_N = f(V_A, V_B, V_C)$
commit(N, I, V_N)
accept(N, I)

basho

riak

Each Additional Request


Node 1
Node 2
Node 3
I++
commit(N, I, V)
accept(N, I)

1 round trip/request (common case)

Problem

Shipping entire state each request is expensive

Solution

Paxos

+

Replicated Log

Problem

Now I have

N problems

Log recovery
Log trimming
Rollup
Snapshots
Fault Recovery

Paxos Made Live - An Engineering Perspective

Tushar Chandra
Robert Griesemer
Joshua Redstone

June 20, 2007

Abstract

We describe our experience in building a fault-tolerant data-base using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

Better Solution

Build log replication into protocol

Better Solution

ZK Atomic Broadcast

Raft

Zab

ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

Abstract

In this paper, we describe ZooKeeper, a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high performance

that implement mutually exclusive access to critical resources.

One approach to coordination is to develop services for each of the different coordination needs. For example, Amazon Simple Queue Service [3] focuses specifically on queuing. Other services have been developed specifically for leader election [25] and configuration [27]. Services that implement more powerful primitives can be used to implement less powerful ones. For example, Chubby [6] is a locking service with strong synchronization guarantees. Locks can then be used to implement leader election, group membership, etc.

When designing our coordination service, we moved away from implementing specific primitives on the server side, and instead we opted for exposing an API that enables application developers to implement their own primitives. Such a choice led to the implementation of a *coordination kernel* that enables new primitives without requiring changes to the service core. This ap-

A simple totally ordered broadcast protocol

Benjamin Reed
Yahoo! Research
Santa Clara, CA - USA
breed@yahoo-inc.com

Flavio P. Junqueira
Yahoo! Research
Barcelona, Catalunya - Spain
fpj@yahoo-inc.com

ABSTRACT

This is a short overview of a totally ordered broadcast protocol used by ZooKeeper, called Zab. It is conceptually easy to understand, is easy to implement, and gives high performance. In this paper we present the requirements ZooKeeper makes on Zab, we show how the protocol is used, and we give an overview of how the protocol works.

chines providing the service and always has a consistent view of the ZooKeeper state. The service tolerates up to f crash failures, and it requires at least $2f + 1$ servers.

Applications use ZooKeeper extensively and have tens to thousands of clients accessing it concurrently, so we require high throughput. We have designed ZooKeeper for workloads with ratios of read to write operations that are higher than 2:1; however, we have found that ZooKeeper's

YAHOO! LABS

TECHNICAL REPORT
YL-2010-0007

DISSECTING ZAB

Flavio Junqueira, Benjamin Reed, and Marco Serafini
Yahoo! Labs
701 First Ave
Sunnyvale, CA 94089
{fpj,breed,serafini@yahoo-inc.com}

Zab: High-performance broadcast for primary-backup systems

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini
Yahoo! Research
{fpj,breed,serafini}@yahoo-inc.com

***Abstract*—Zab is a crash-recovery atomic broadcast algorithm we designed for the ZooKeeper coordination service. ZooKeeper implements a primary-backup scheme in which a primary process executes clients operations and uses Zab to propagate the corresponding incremental state changes to backup processes[1]. Due the dependence of an incremental state change on the sequence of changes previously generated, Zab must guarantee that if it delivers a given state change, then all other changes it depends upon must be delivered first. Since primaries may crash, Zab must satisfy this requirement despite crashes of primaries.**

Applications using ZooKeeper demand high-performance from the service, and consequently, one important goal is the ability of having multiple outstanding client operations at a time. Zab enables multiple outstanding state changes by guaranteeing that at most one primary is able to broadcast state changes and have them incorporated into the state, and by using a synchronization phase while establishing a new primary. Before this synchronization phase completes, a new primary does not broadcast new state changes. Finally, Zab uses an identification scheme for state changes that enables a process to easily identify missing changes. This feature is key for efficient recovery.

Experiments and experience so far in production show that our design enables an implementation that meets the performance requirements of our applications. Our implementation of Zab can achieve tens of thousands of broadcasts per second, which is sufficient for demanding systems such as our Web-scale applications.

***Index Terms*—Fault tolerance, Distributed algorithms, Primary backup, Asynchronous consensus, Atomic broadcast**

scheme [5], [6], [7] to maintain the state of replica processes consistent. With ZooKeeper, a primary process receives all incoming client requests, executes them, and propagates the resulting non-commutative, incremental state changes in the form of *transactions* to the backup replicas using *Zab*, the ZooKeeper atomic broadcast protocol. Upon primary crashes, processes execute a recovery protocol both to agree upon a common consistent state before resuming regular operation and to establish a new primary to broadcast state changes. To exercise the primary role, a process must have the support of a quorum of processes. As processes can crash and recover, there can be over time multiple primaries and in fact the same process may exercise the primary role multiple times. To distinguish the different primaries over time, we associate an instance value with each established primary. A given instance value maps to at most one process. Note that our notion of instance shares some of the properties of views of group communication [8], but it presents some key differences. With group communication, all processes in a given view are able to broadcast, and configuration changes happen when any process joins or leaves. With Zab, processes change to a new view (or primary instance) only when a primary crashes or loses support from a quorum.

Critical to the design of Zab is the observation that each state change is *incremental with respect to the previous state*,

riak_zab

Raft

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University
(Draft of October 7, 2013)

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

terminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, students were able to answer questions about Raft 23% better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [28, 21]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to

raftconsensus.github.io

rafter

riak_ensemble

riak_ensemble

Paxos framework for scalable consistent system

basho

riak

Problem

Shipping entire state each request is expensive

Solution

Micro-states

Also solves

Scalability

Key/Value

Each key is independent state

Semantics

Conditional single key atomic operations

get/modify/put
fails if object changed
(eg. concurrent put)

Design

Simple multi-paxos

per key

basho

riak

1B keys

=

1B consensus groups?

basho

riak

No

Partition keys across N consensus groups

Partition keys across N ensembles

Ensembles emulate paxos per key

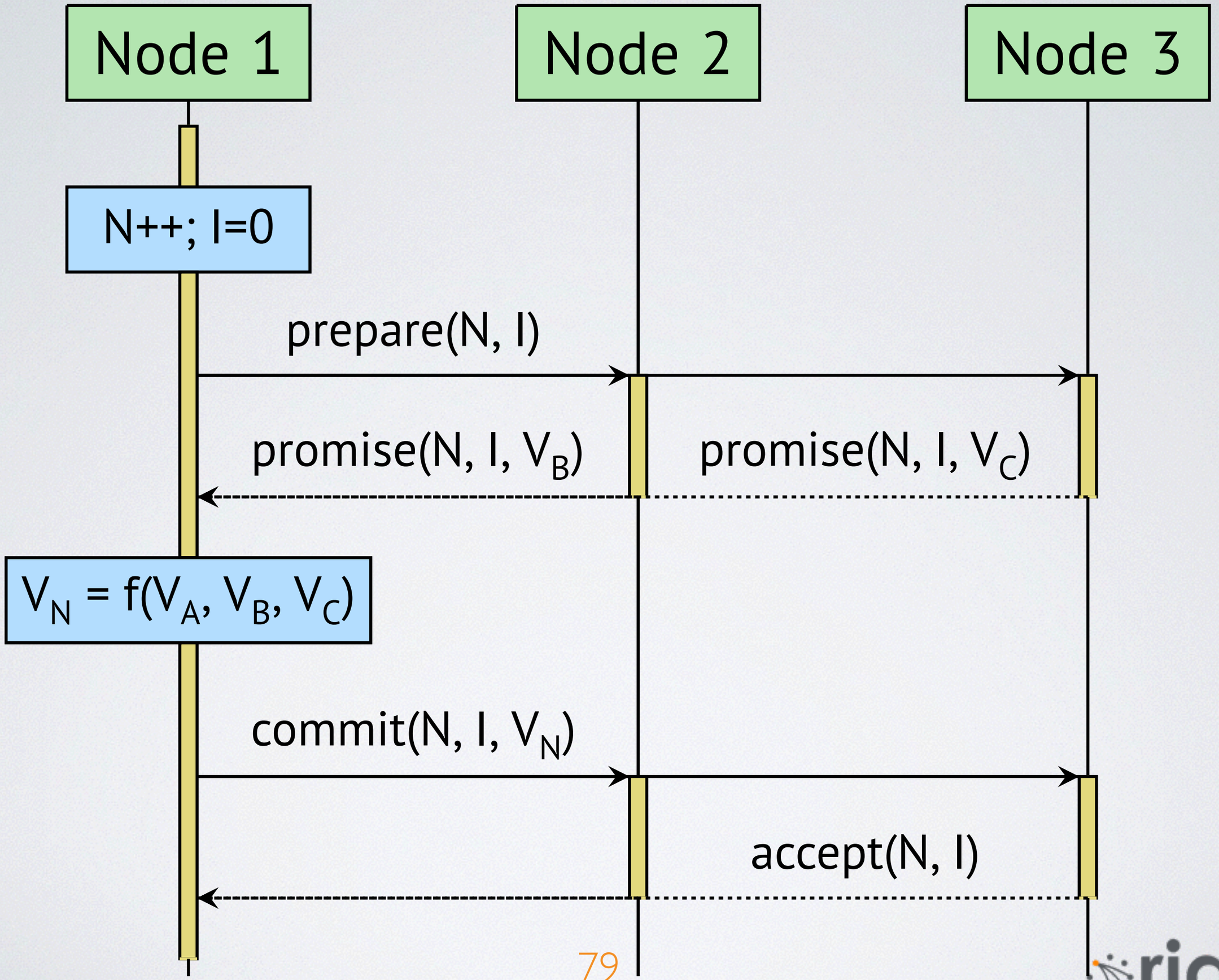
Each Ensemble

Elects leader

Establishes epoch

Supports get/put/modify

Establish a new epoch


Node 1
Node 2
Node 3
N++; I=0
prepare(N, I)
promise(N, I, V_B)
promise(N, I, V_C)
$V_N = f(V_A, V_B, V_C)$
commit(N, I, V_N)
accept(N, I)

basho

riak

consensus state

epoch

sequence

membership

leader

basho

riak

K/V objects

epoch

sequence

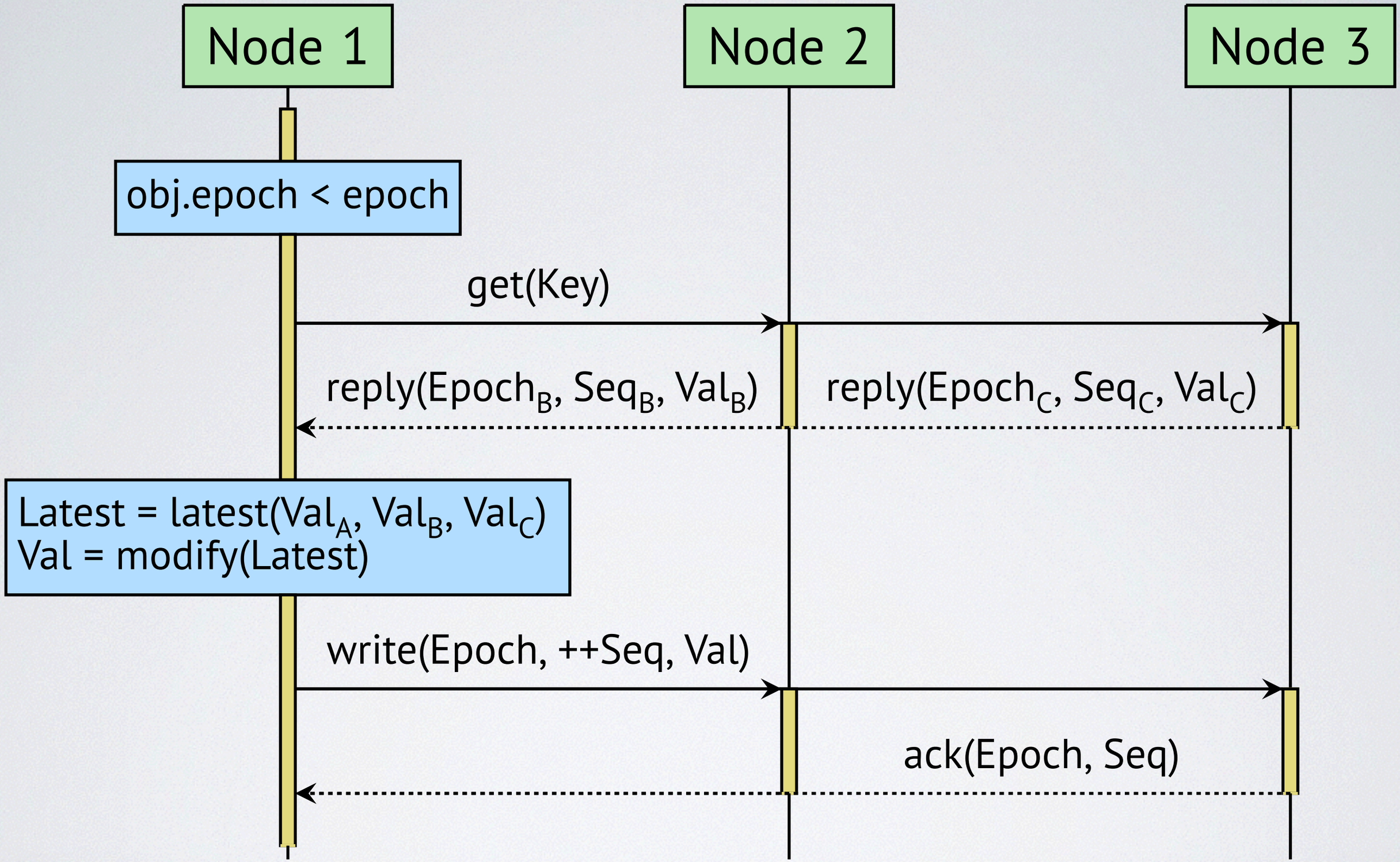
key

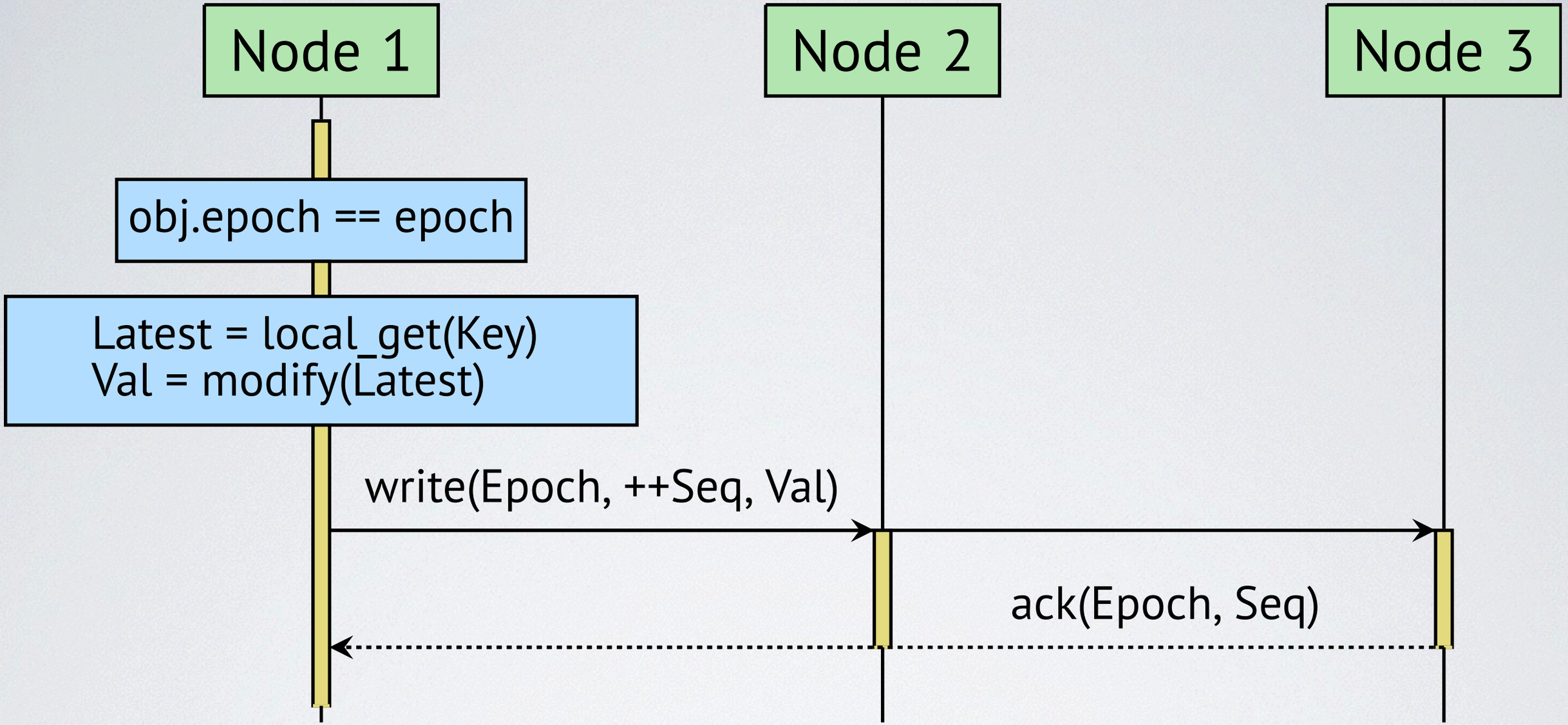
value

basho

riak

Put

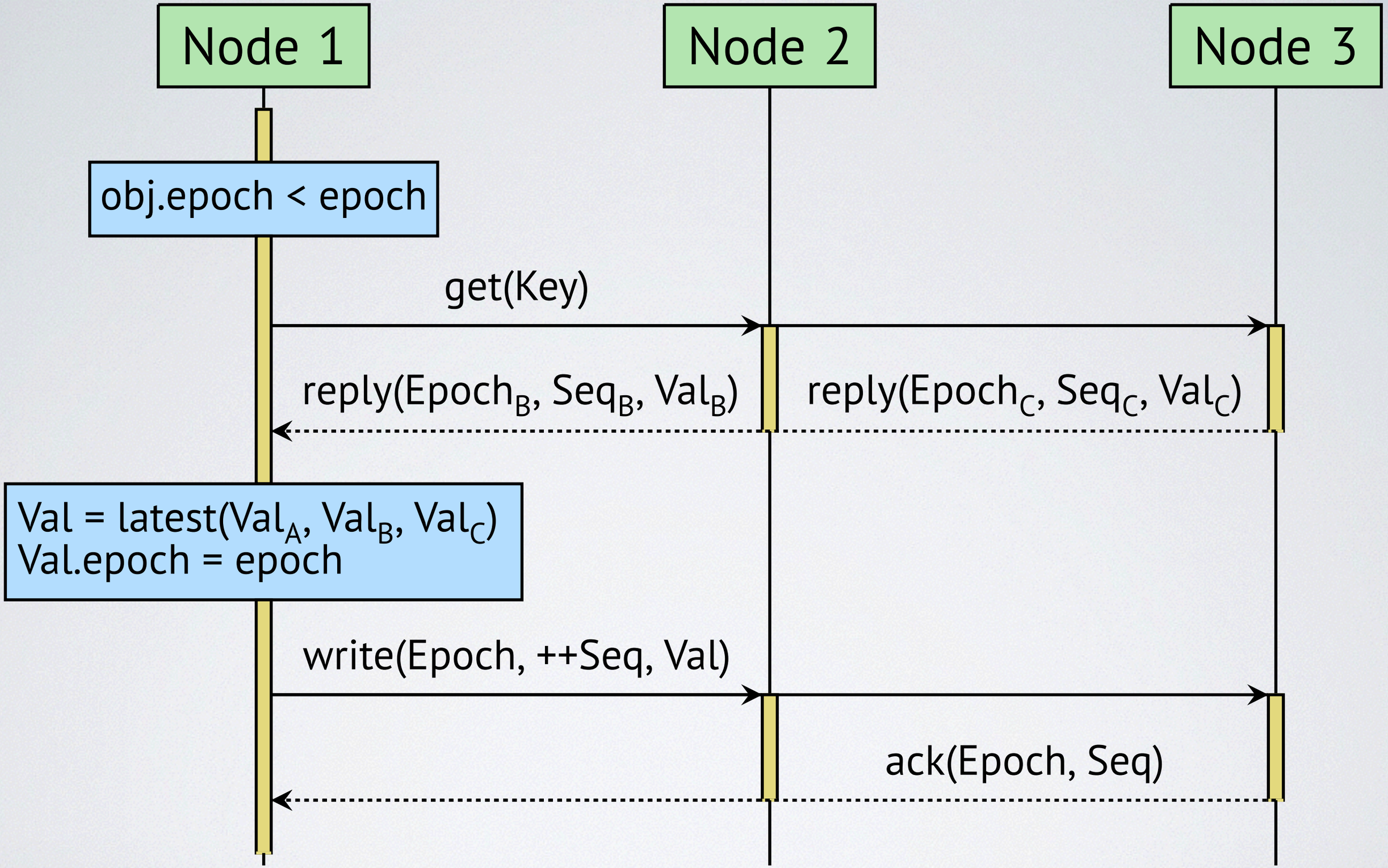

Node 1
Node 2
Node 3
obj.epoch < epoch
get(Key)
reply(Epoch$_B$, Seq$_B$, Val$_B$)
reply(Epoch$_C$, Seq$_C$, Val$_C$)
Latest = latest(Val$_A$, Val$_B$, Val$_C$)
Val = modify(Latest)
write(Epoch, ++Seq, Val)
ack(Epoch, Seq)

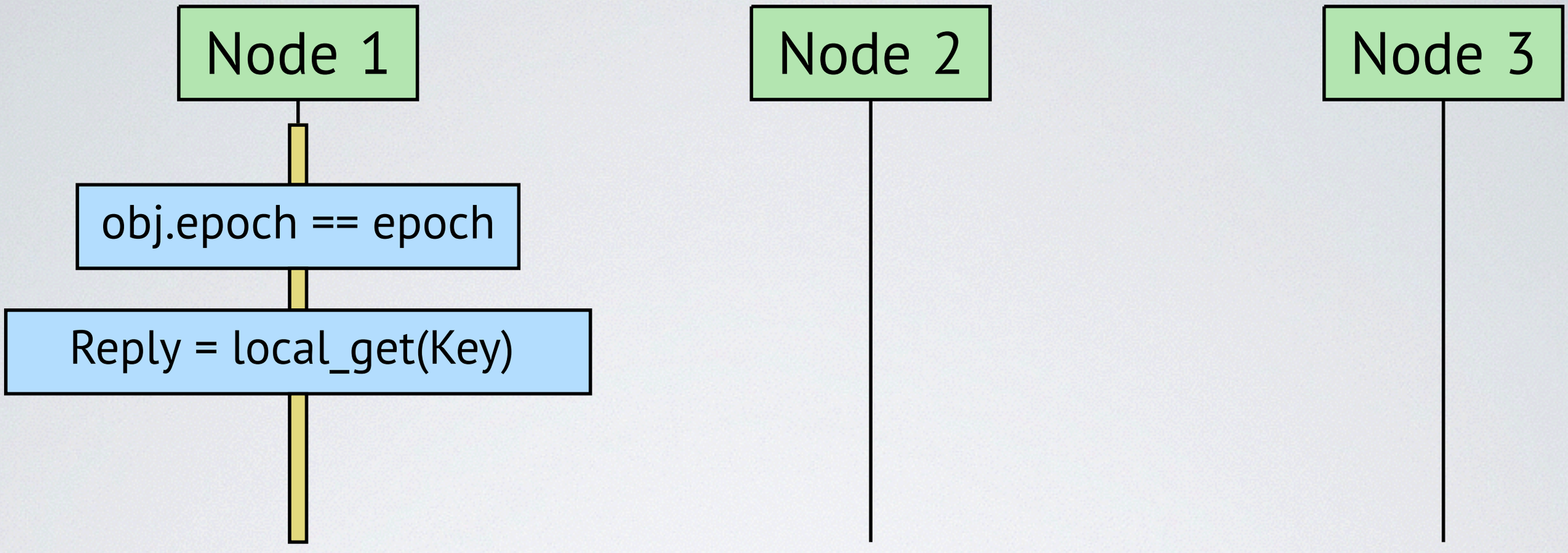

Node 1
Node 2
Node 3
obj.epoch == epoch
Latest = local_get(Key)
Val = modify(Latest)
write(Epoch, ++Seq, Val)
ack(Epoch, Seq)

2 roundtrips/put (worst)

1 roundtrip/put (best)

Get


Node 1
Node 2
Node 3
obj.epoch < epoch
get(Key)
reply(Epoch$_B$, Seq$_B$, Val$_B$)
reply(Epoch$_C$, Seq$_C$, Val$_C$)
Val = latest(Val$_A$, Val$_B$, Val$_C$)
Val.epoch = epoch
write(Epoch, ++Seq, Val)
ack(Epoch, Seq)


Node 1
Node 2
Node 3
obj.epoch == epoch
Reply = local_get(Key)

2 roundtrips/get (worst)

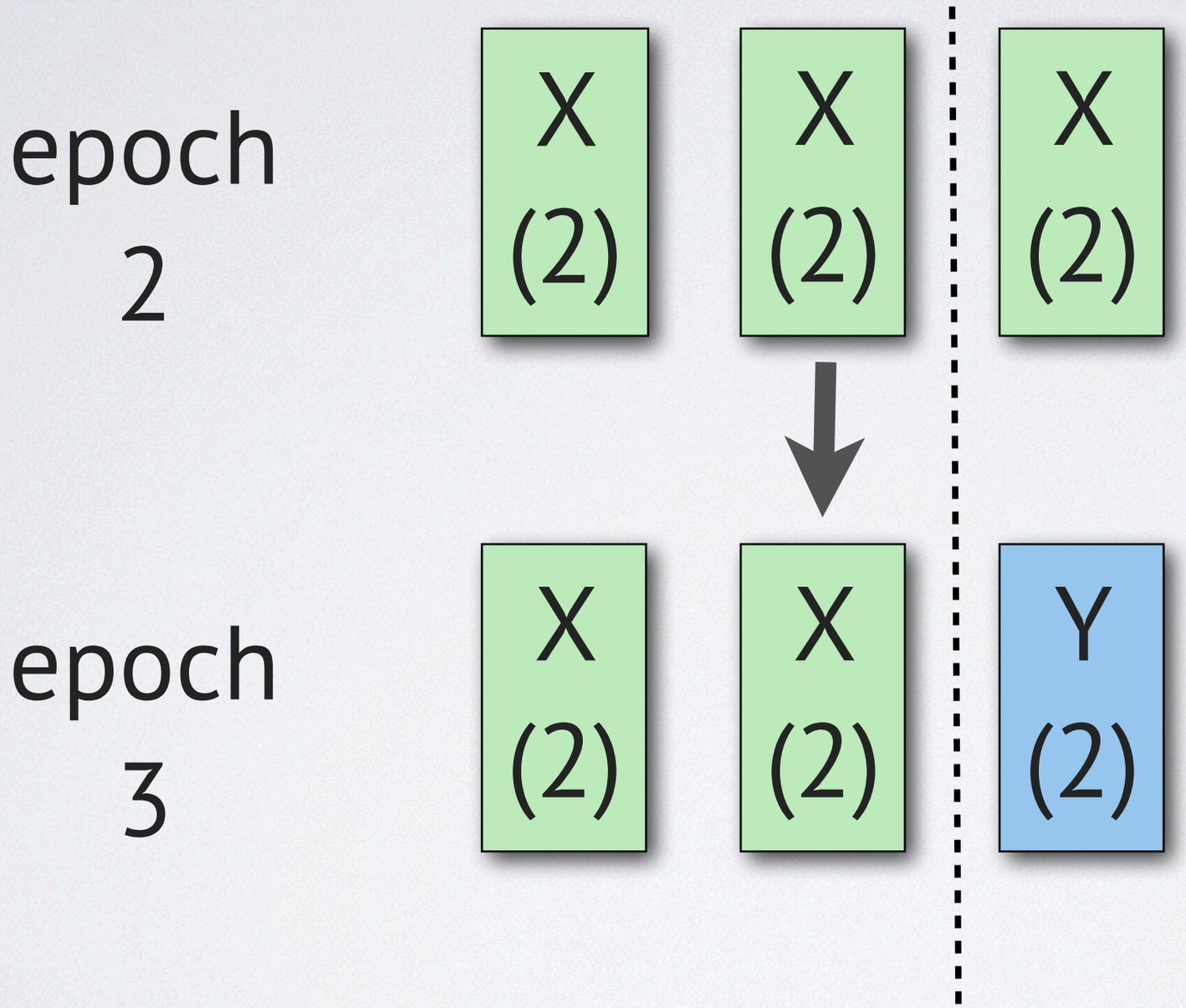
0 roundtrip/get (best)

Leader abandons leadership if any quorum operation ever fails

Which forces new epoch to be established

Partial Writes

failed partial

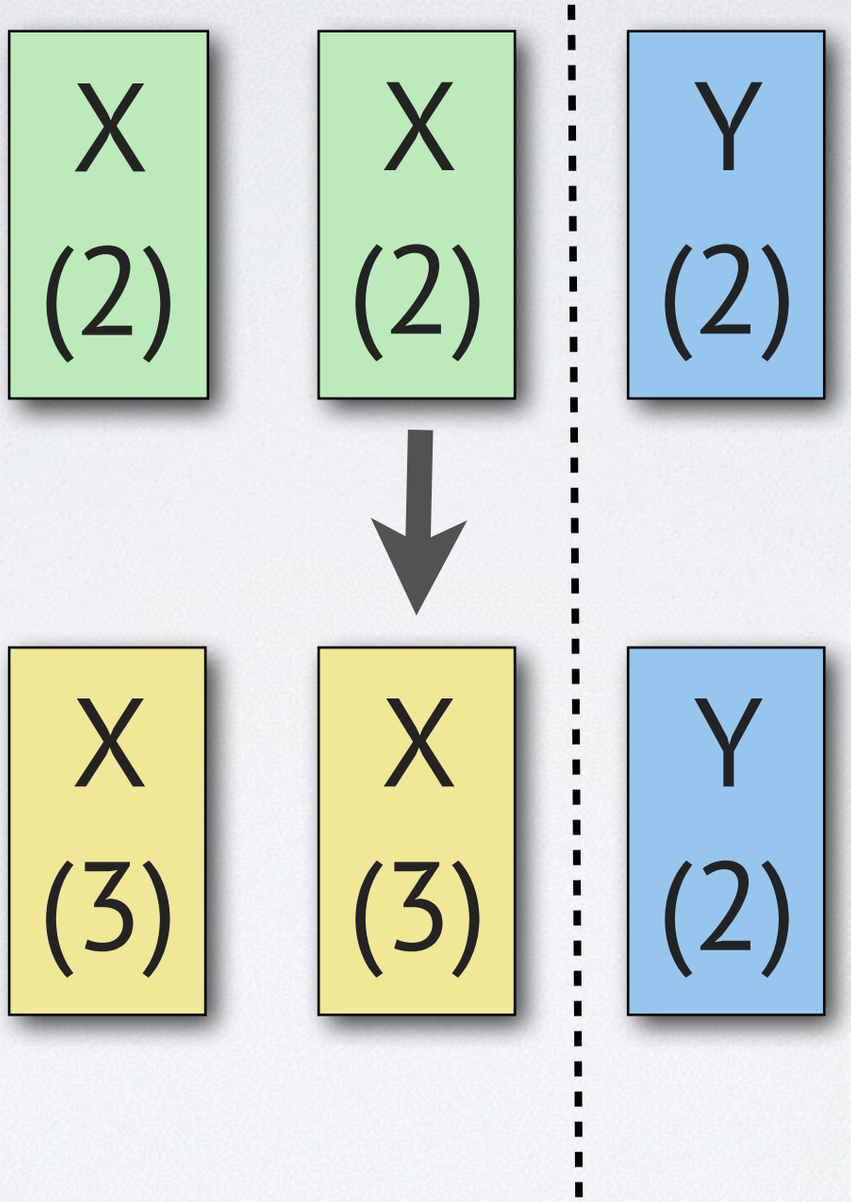


basho

riak

read / rewrite / reply X

epoch 3

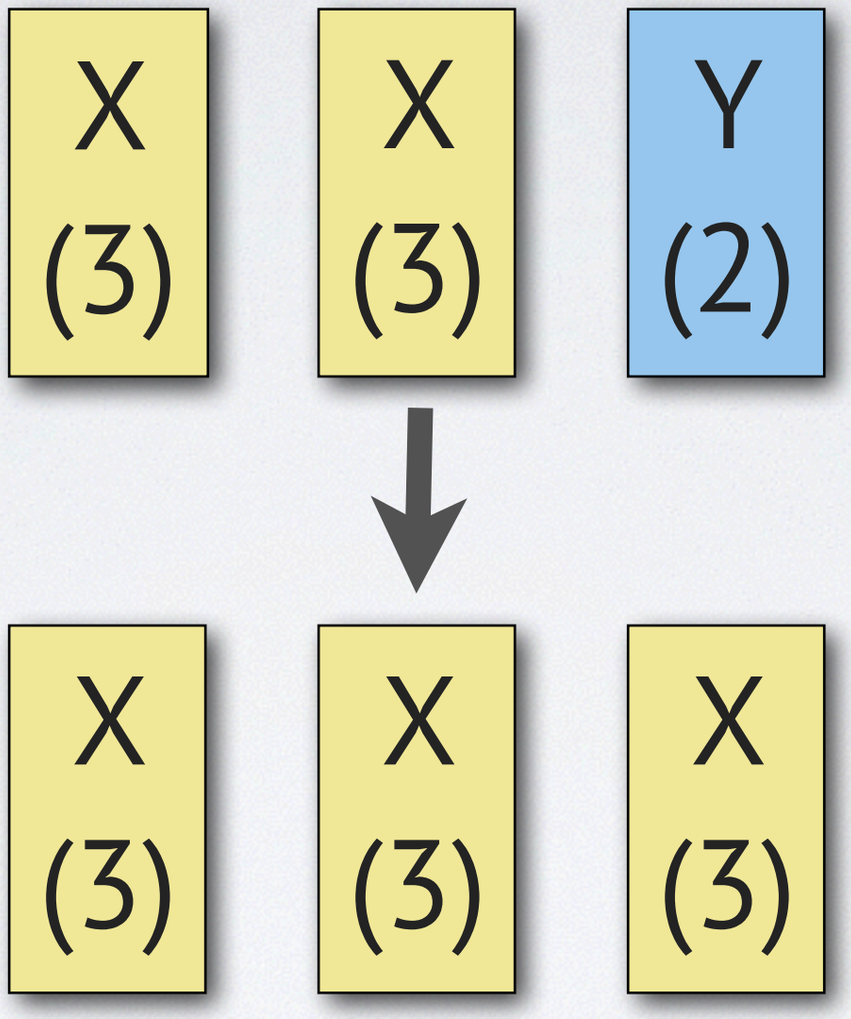


basho

riak

read / repair / reply X

epoch
3

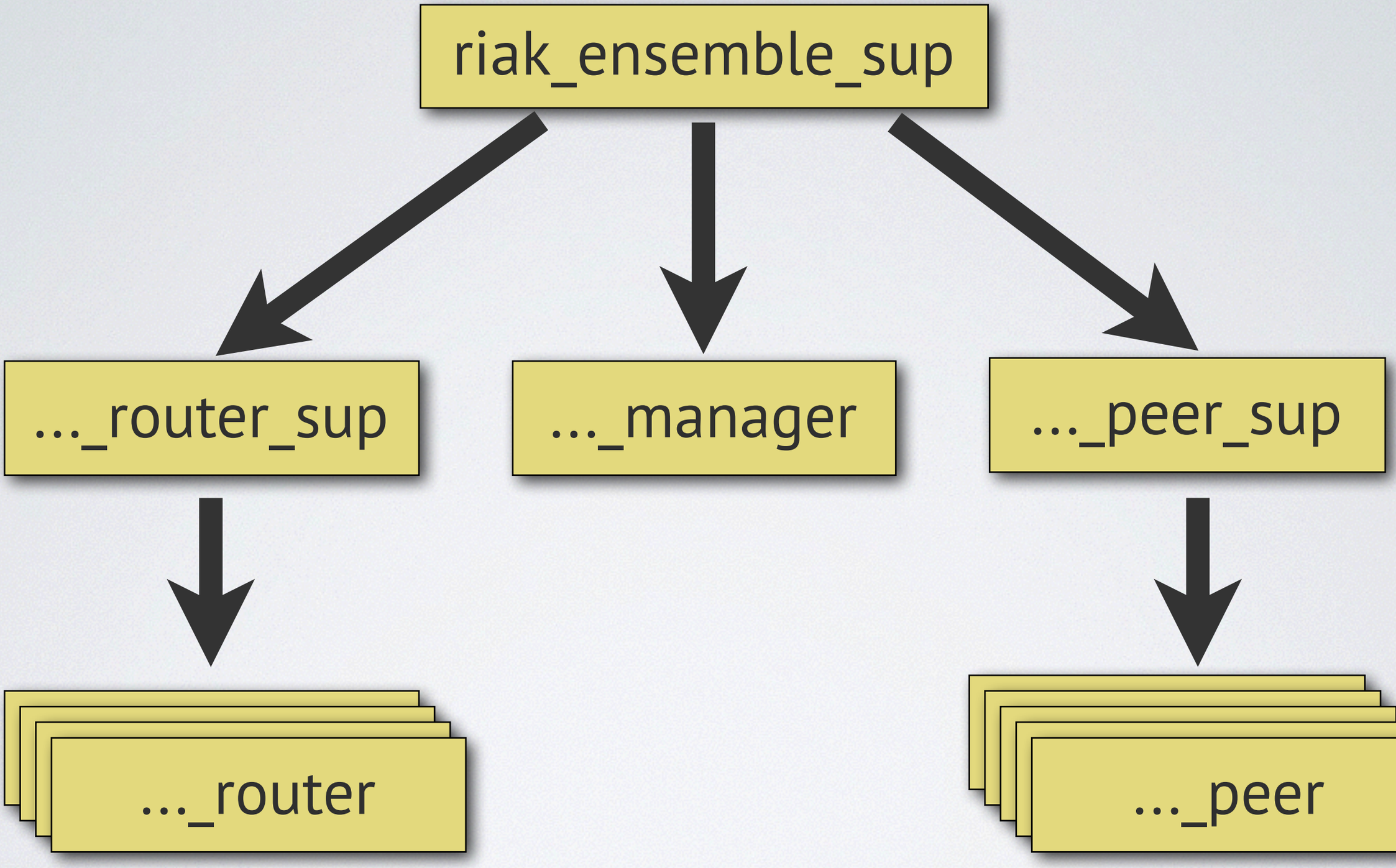


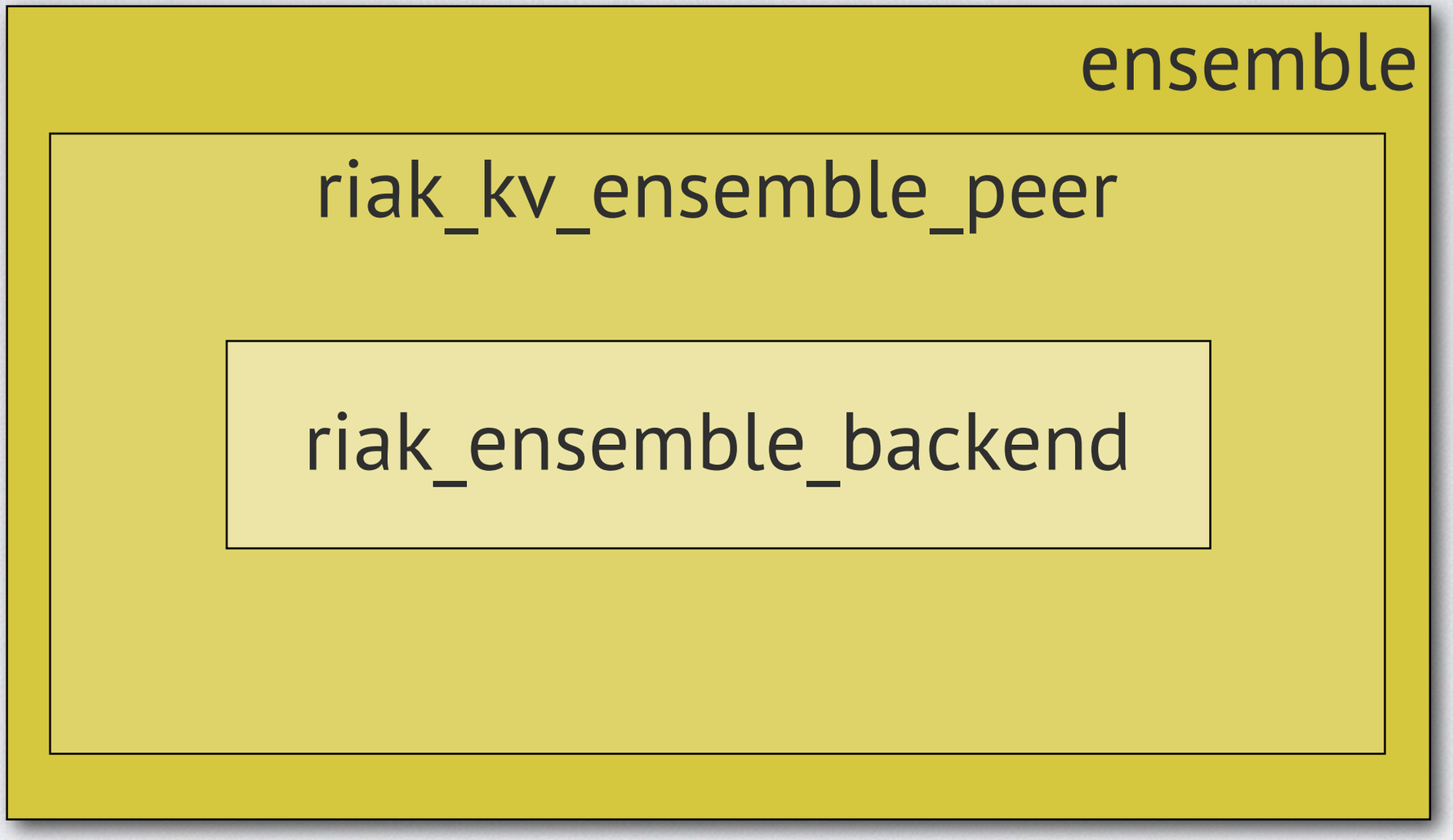
epoch
3

basho

riak

Architecture


riak_ensemble_sup
..._router_sup
..._manager
..._peer_sup
..._router
..._peer


ensemble
riak_kv_ensemble_peer
riak_ensemble_backend

```
%% Initialization callback that returns initial module state.
-callback init(ensemble_id(), peer_id(), [any()]) -> state().
```

```
%% Create a new opaque key/value object using whatever
%% representation the defining module desires.
-callback new_obj(epoch(), seq(), key(), value()) -> obj().

%% Accessors to retrieve epoch/seq/key/value from an opaque object.
-callback obj_epoch(obj()) -> epoch().
-callback obj_seq  (obj()) -> seq().
-callback obj_key  (obj()) -> term().
-callback obj_value(obj()) -> term().

%% Setters for epoch/seq/value for opaque objects.
-callback set_obj_epoch(epoch(), obj()) -> obj().
-callback set_obj_seq  (seq(),   obj()) -> obj().
-callback set_obj_value(term(),  obj()) -> obj().
```

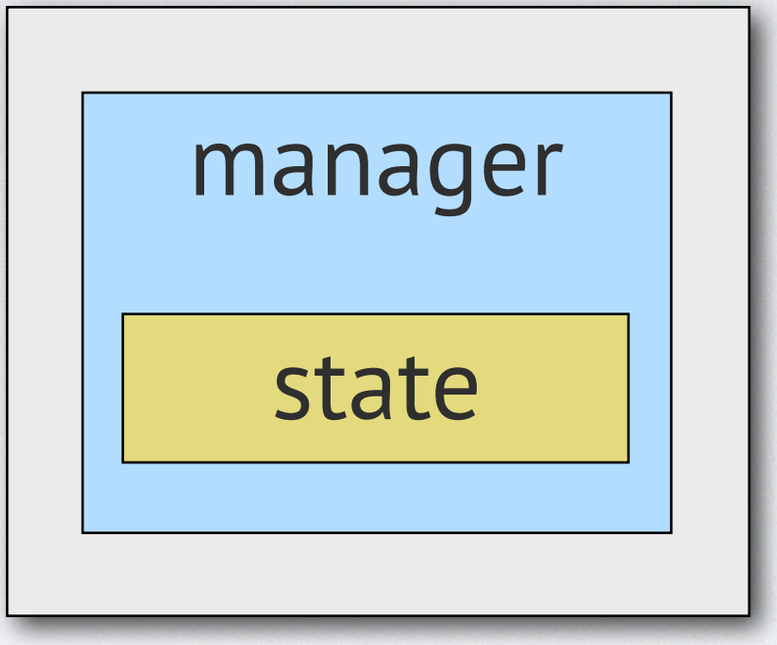
```
%% Callback for get operations. Responsible for sending a reply
%% to the waiting `from' process using {@link reply/2}.
-callback get(key(), from(), state()) -> state().

%% Callback for put operations. Responsible for sending a reply
%% to the waiting `from' process using {@link reply/2}.
-callback put(key(), obj(), from(), state()) -> state().
```

```
%% Callback for sync_request sent from a remote peer that wants to
%% sync with this peer. Responsible for sending a reply to the
%% waiting `from' peer using {@link reply/2}.
-callback sync_request(from(), state()) -> state().

%% Callback that should do whatever is necessary to bring this peer
%% up-to-date. Passed in a list of replies generated by `sync_request'
%% from a quorum of peers from each view. This callback can either
%% directly make the peer current and return `ok', or initiate some
%% longer lived background process and return `async', followed by
%% calling {@link sync_complete/1} or {@link sync_failed/1} when
%% finished/failed.
-callback sync([{peer_id(), any()}], state()) -> {ok, state()}       |
                                                 {async, state()}    |
                                                 {{error,_}, state()}.
```

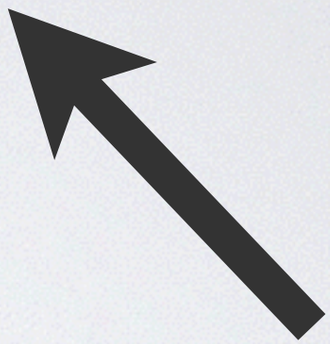
```
%% Callback for periodic leader tick. This function is called
%% periodically by an elected leader. Can be used to implement
%% custom housekeeping.
-callback tick(epoch(), seq(), peer_id(), views(), state()) -> state().

-callback ping(state()) -> {ok|async|failed, state()}.
```

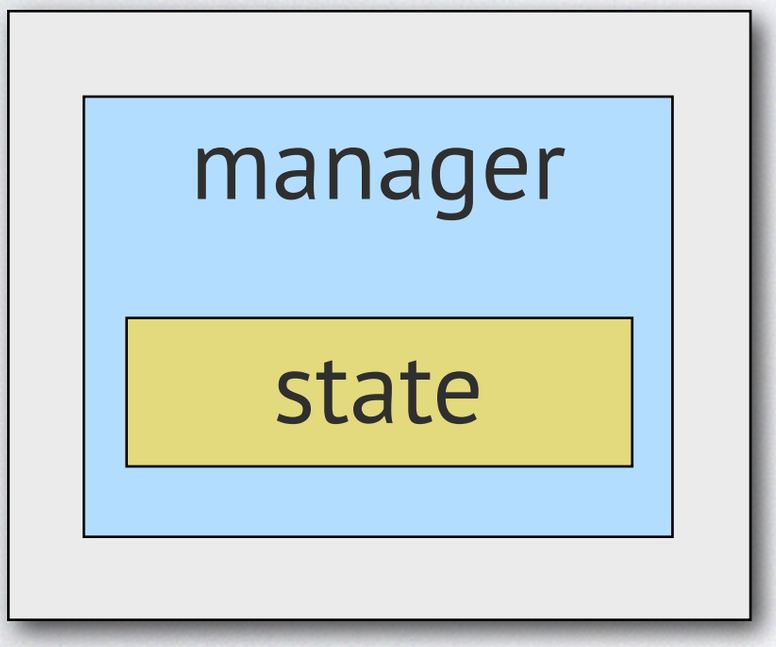
Clustering

gossip

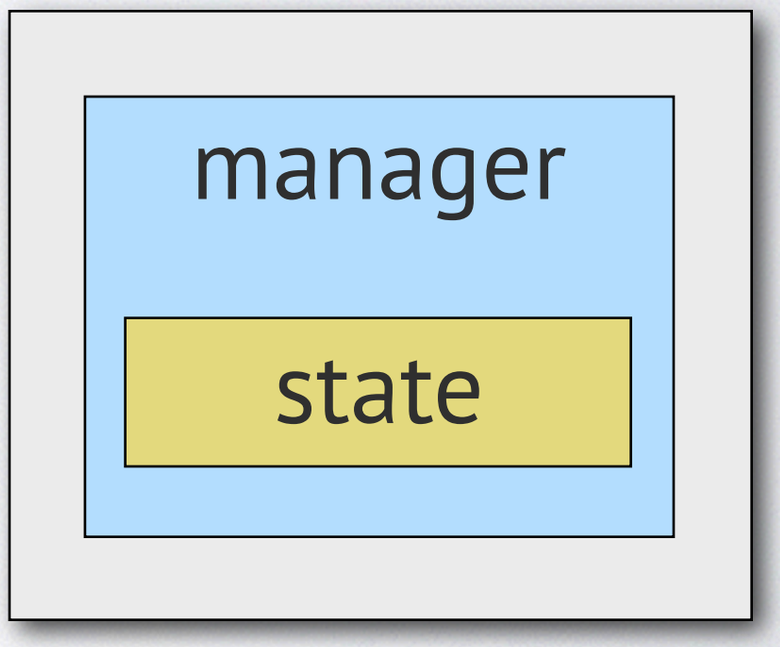


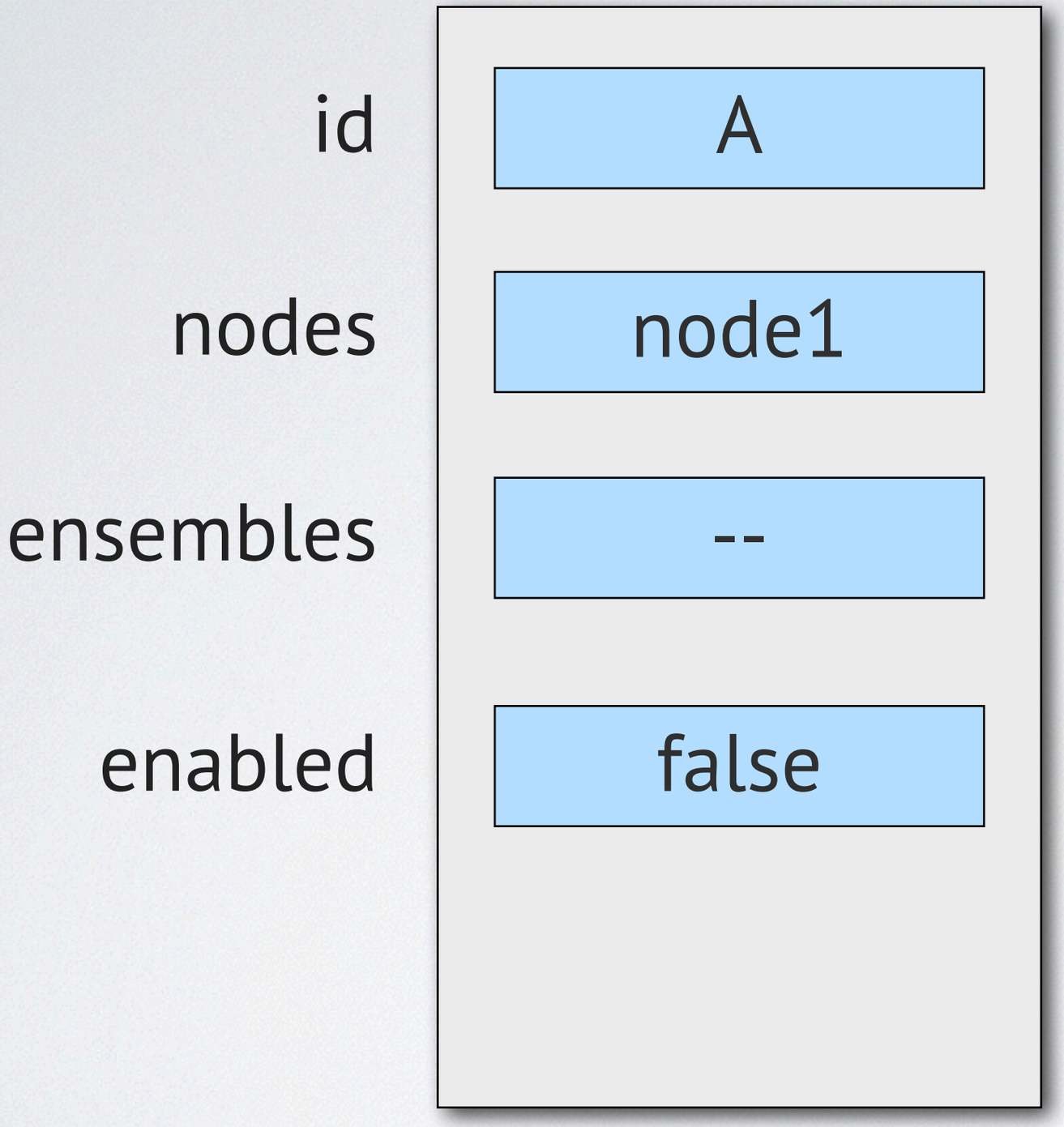
gossip



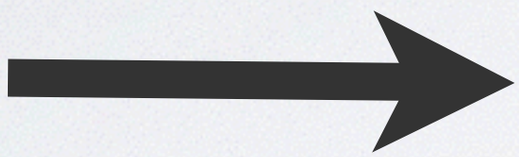


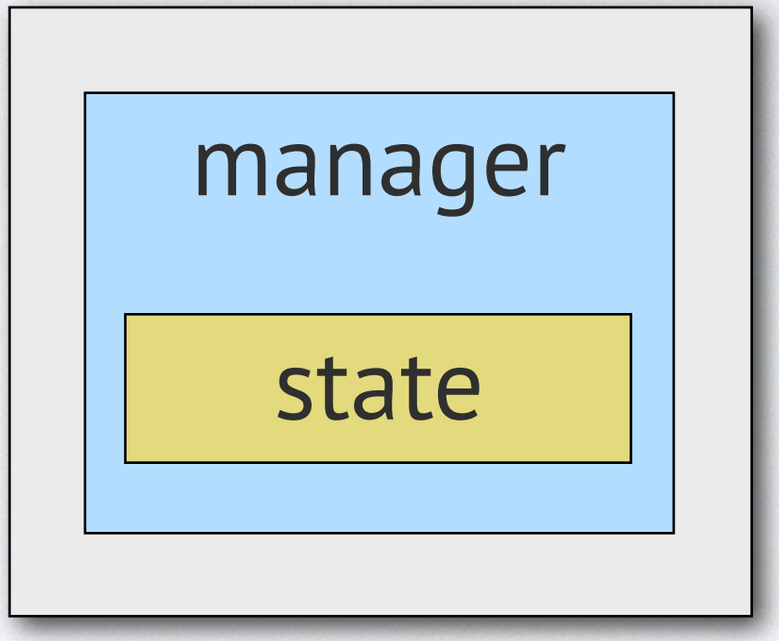
gossip

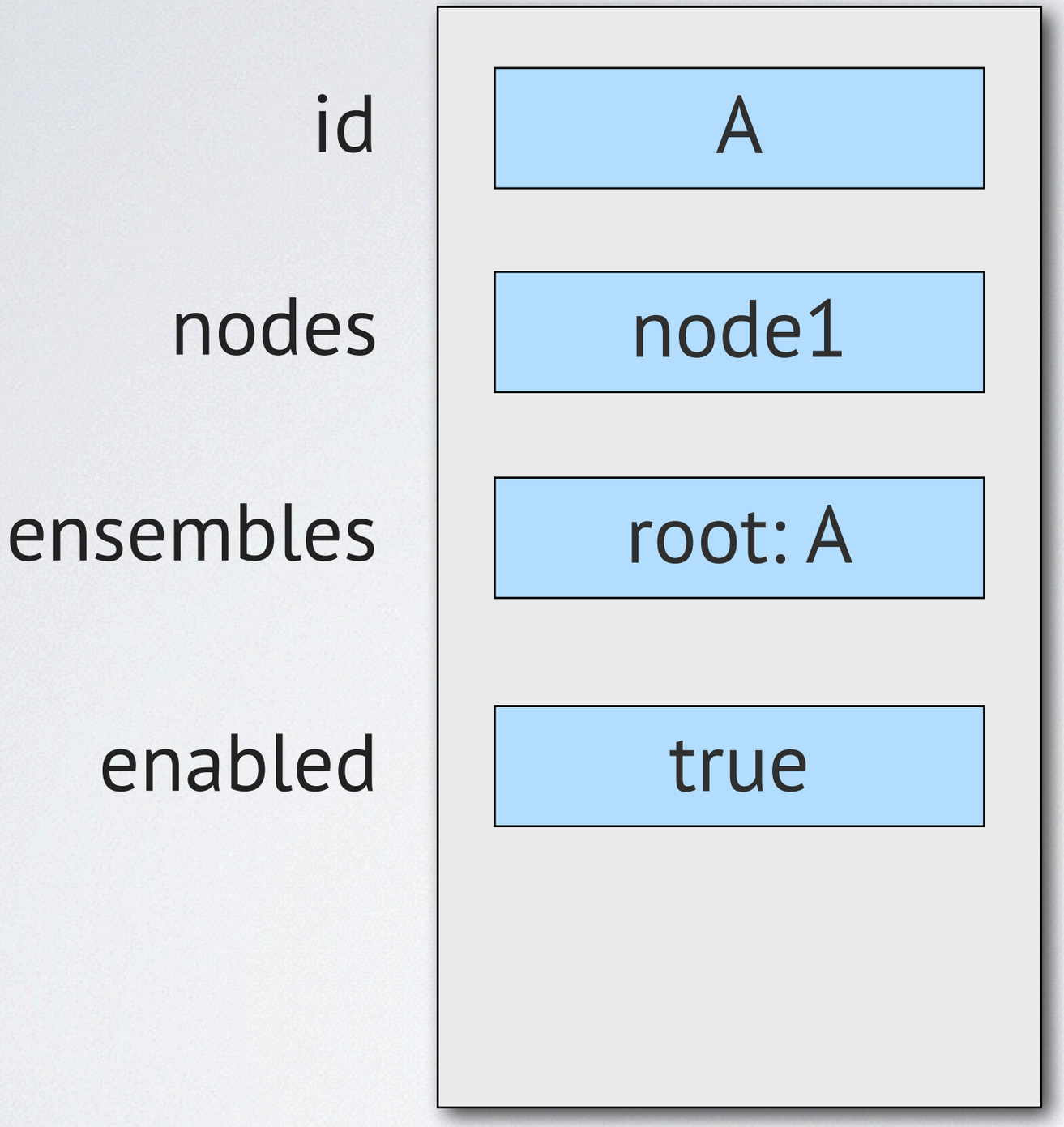


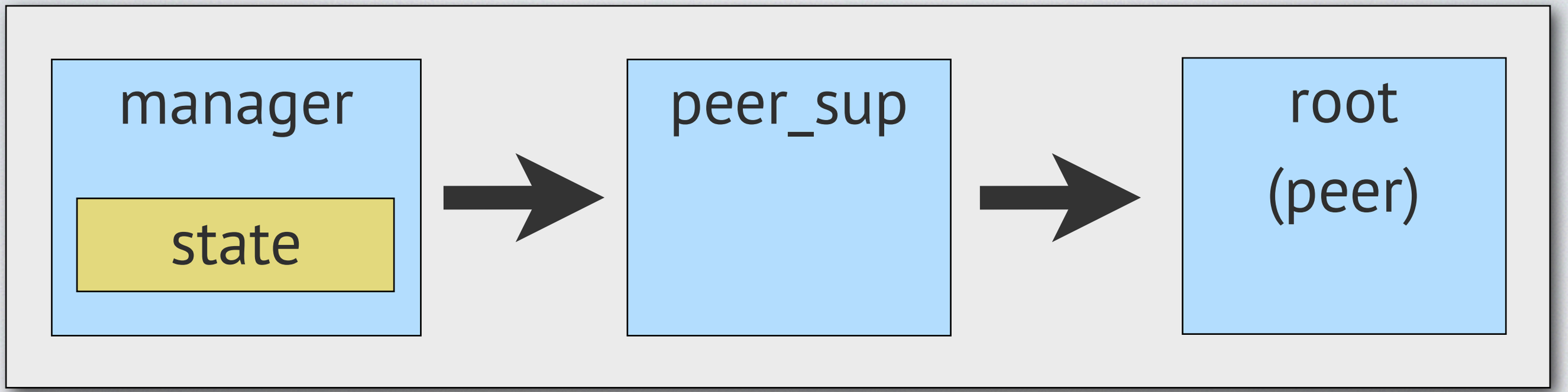

id
A
nodes
node1
ensembles
--
enabled
false

enable

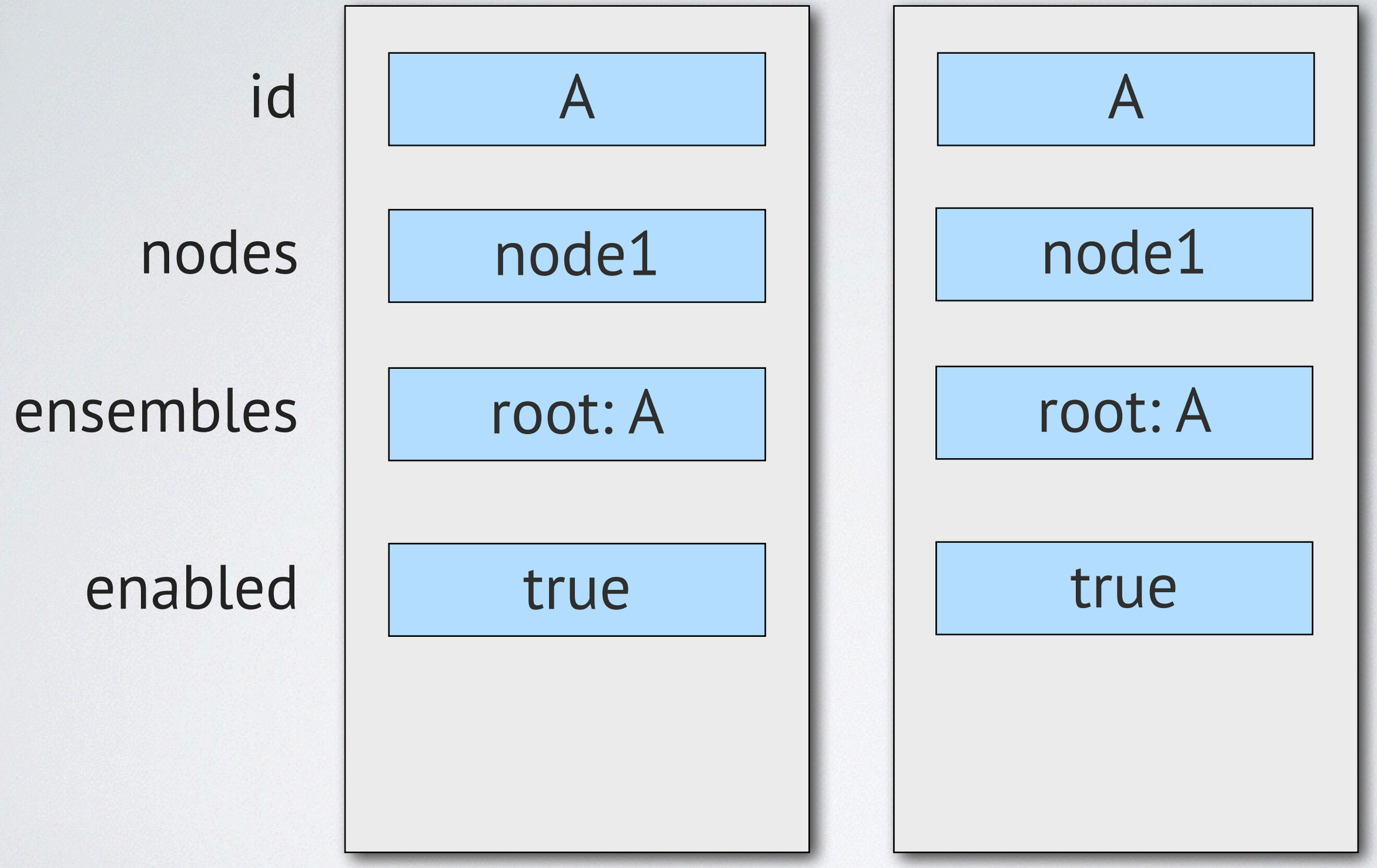





id
A
nodes
node1
ensembles
root: A
enabled
true

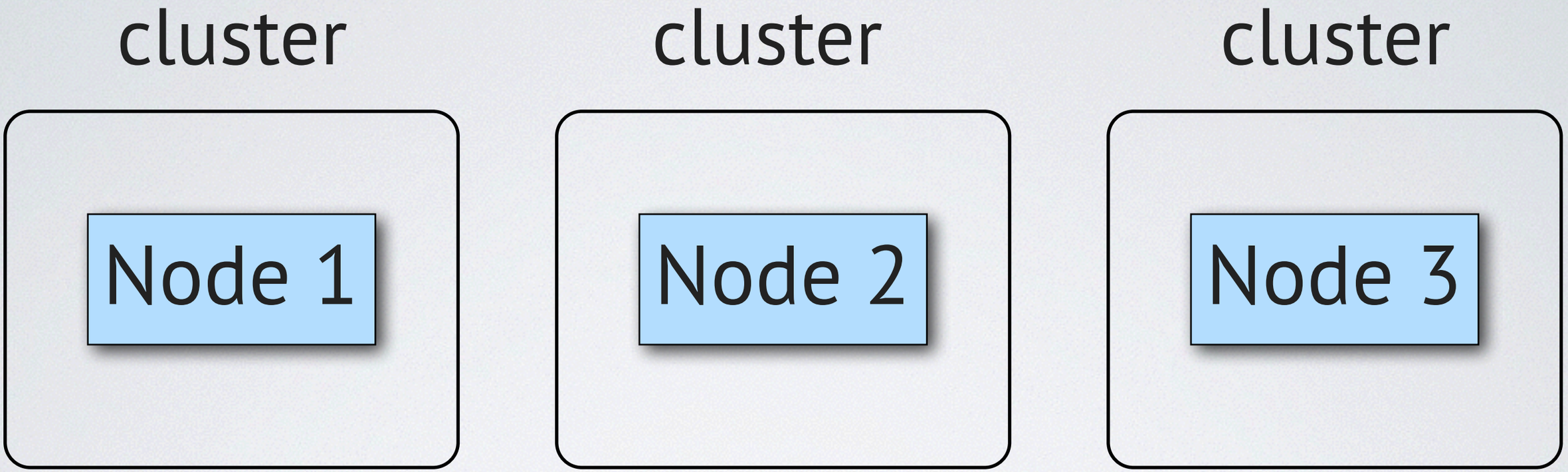

manager
state
peer_sup
root
(peer)

id	A	B
nodes	node1	node2
ensembles	root: A	--
enabled	true	false


id
nodes
ensembles
enabled
A
node1
root: A
true
A
node1
root: A
true

basho

riak


cluster
cluster
cluster
Node 1
Node 2
Node 3

join

cluster

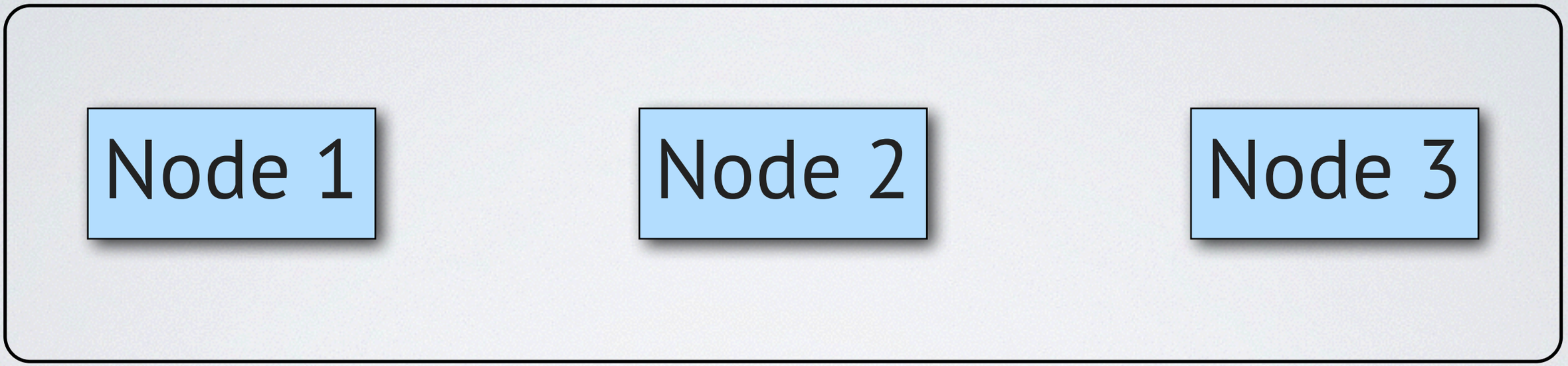
Node 1

cluster

Node 2

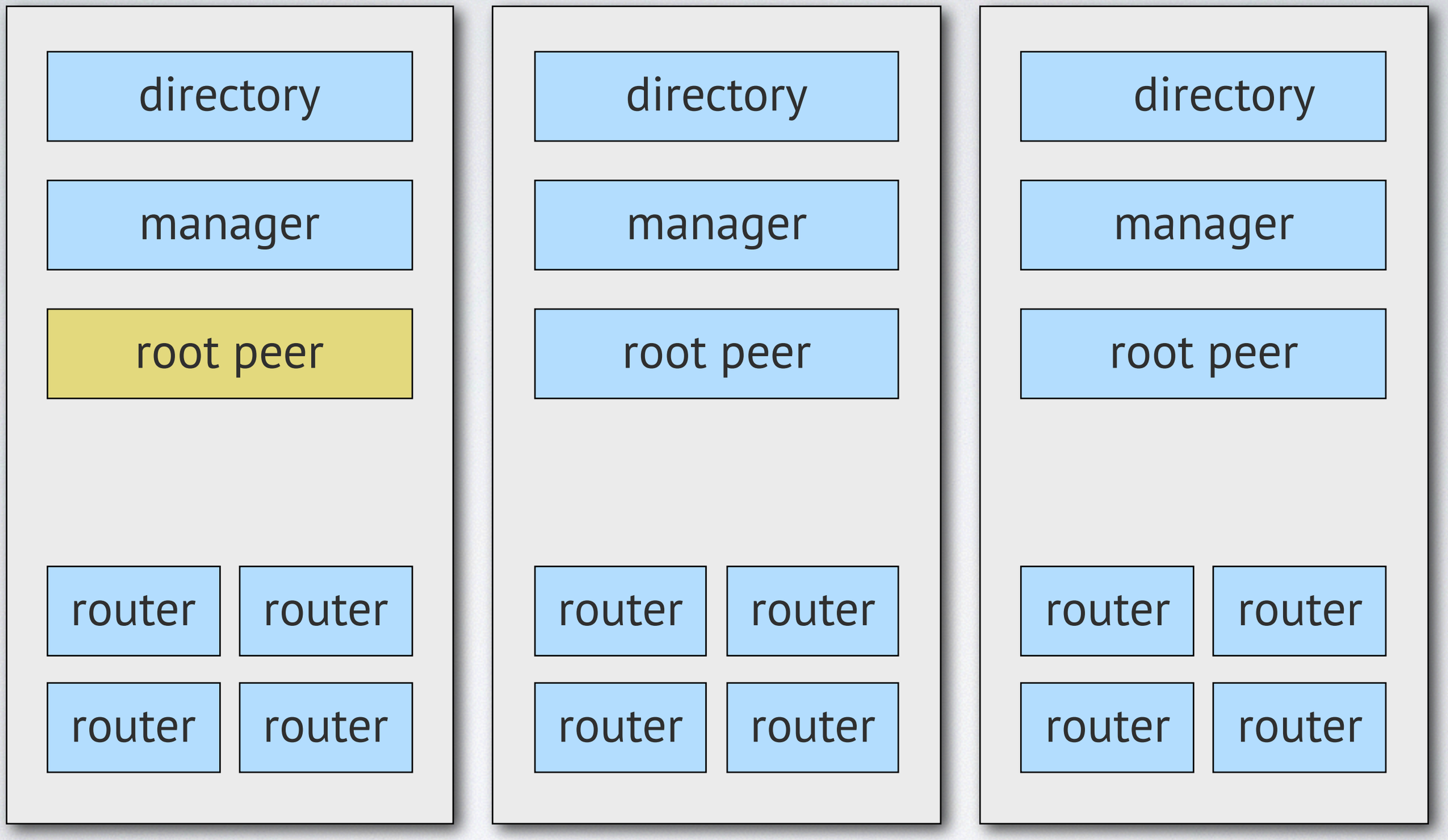
cluster

Node 3


cluster
Node 1
Node 2
Node 3

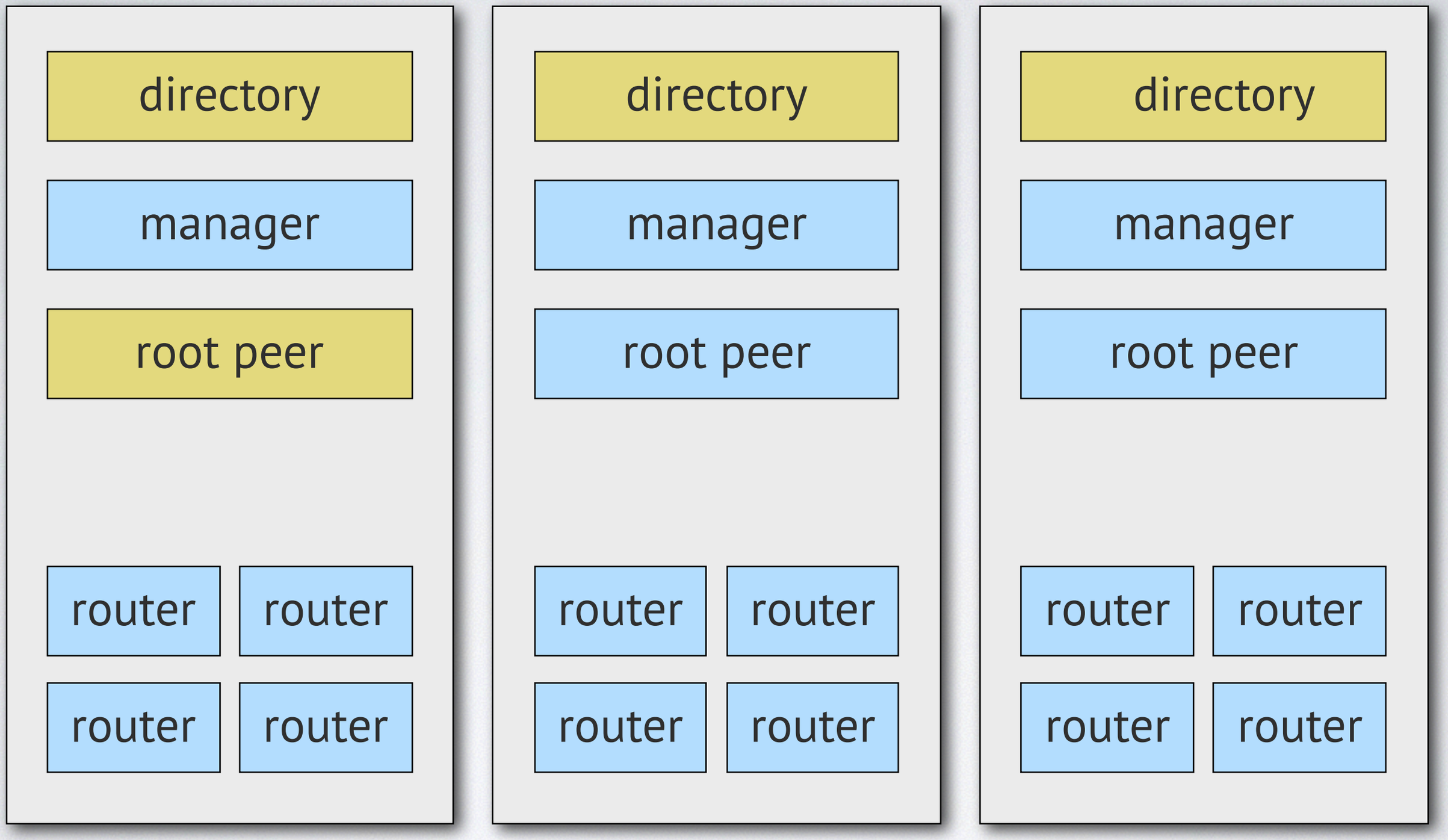
Creating Ensemble

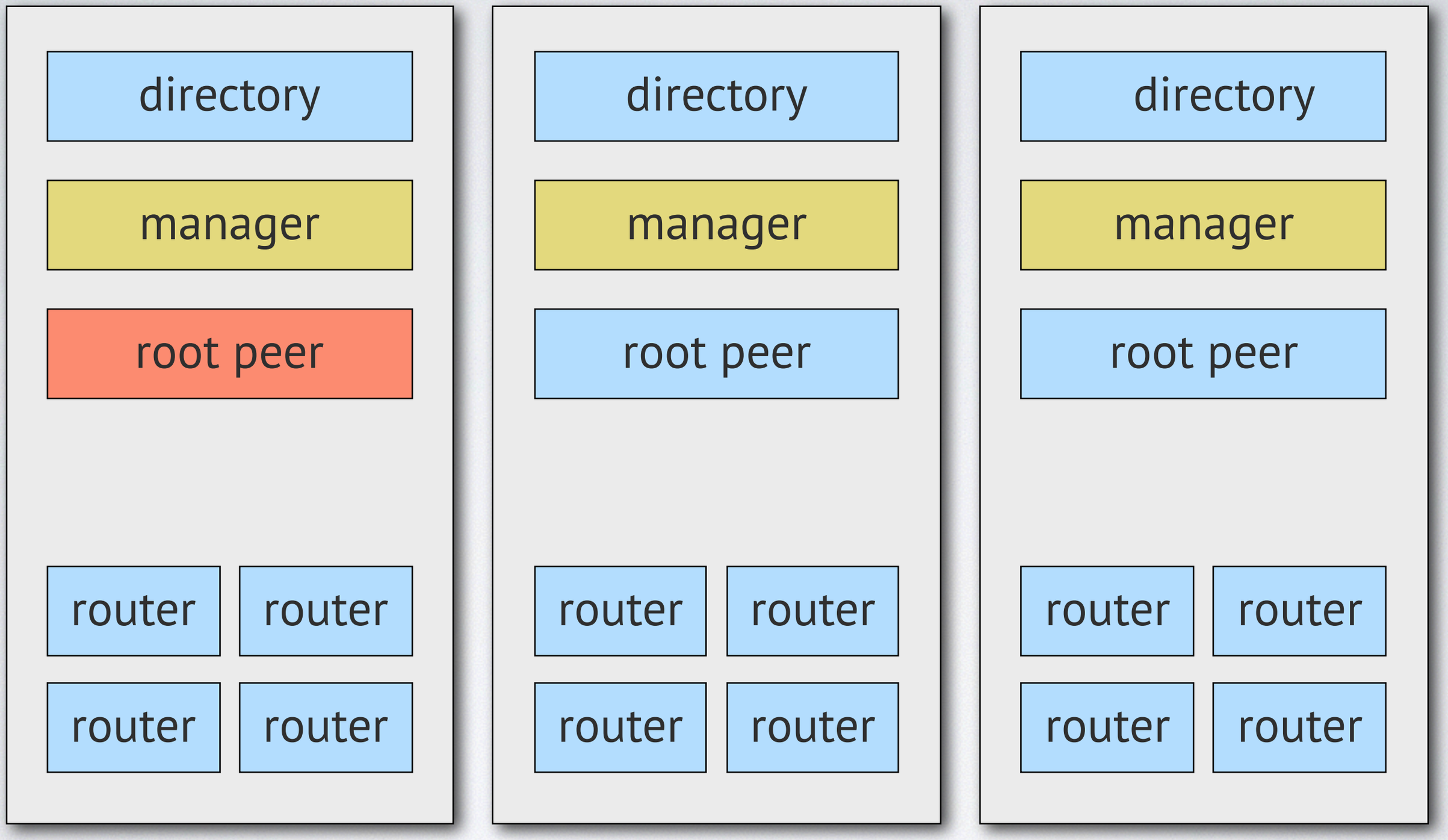
create ensemble

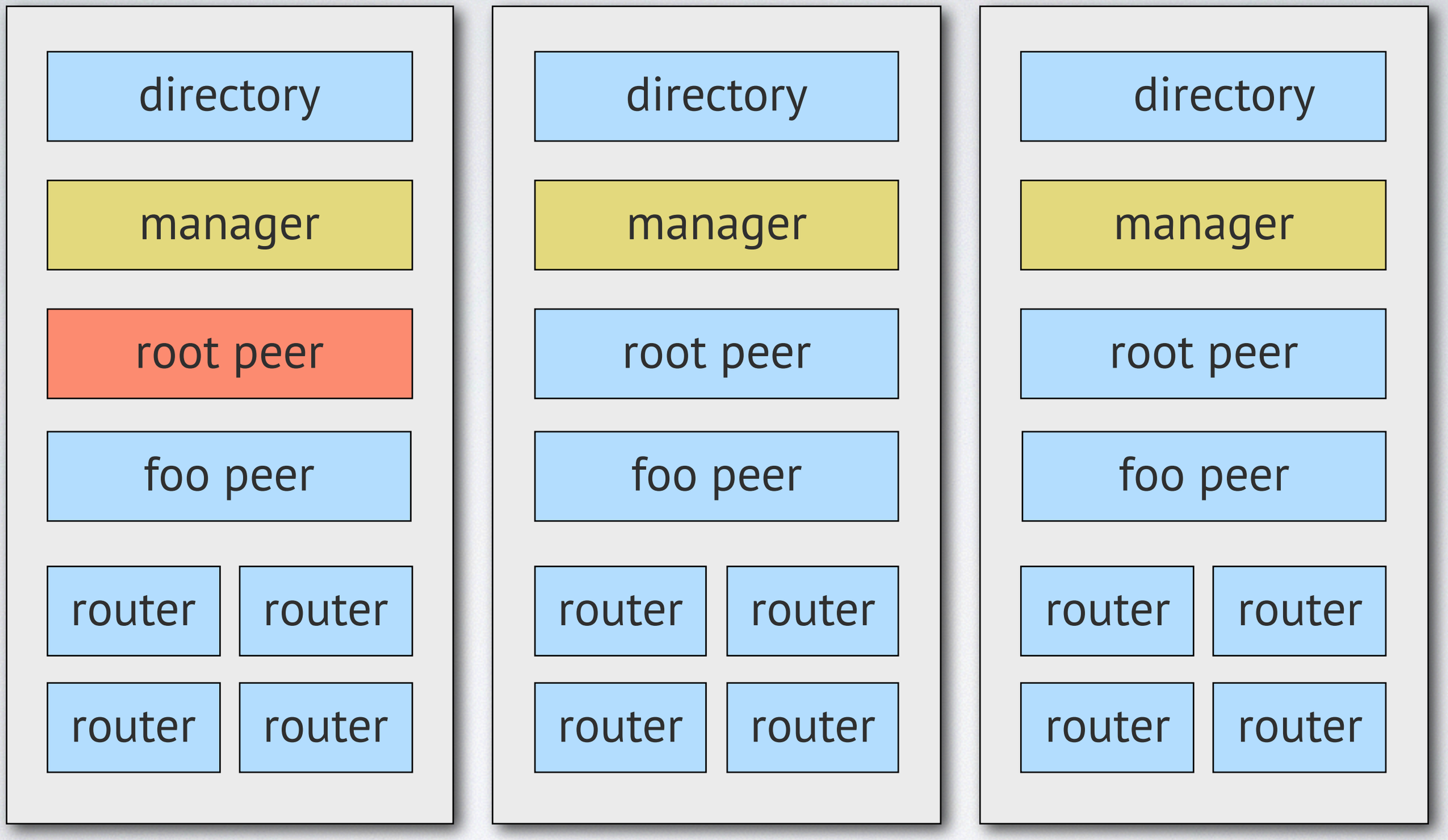


basho

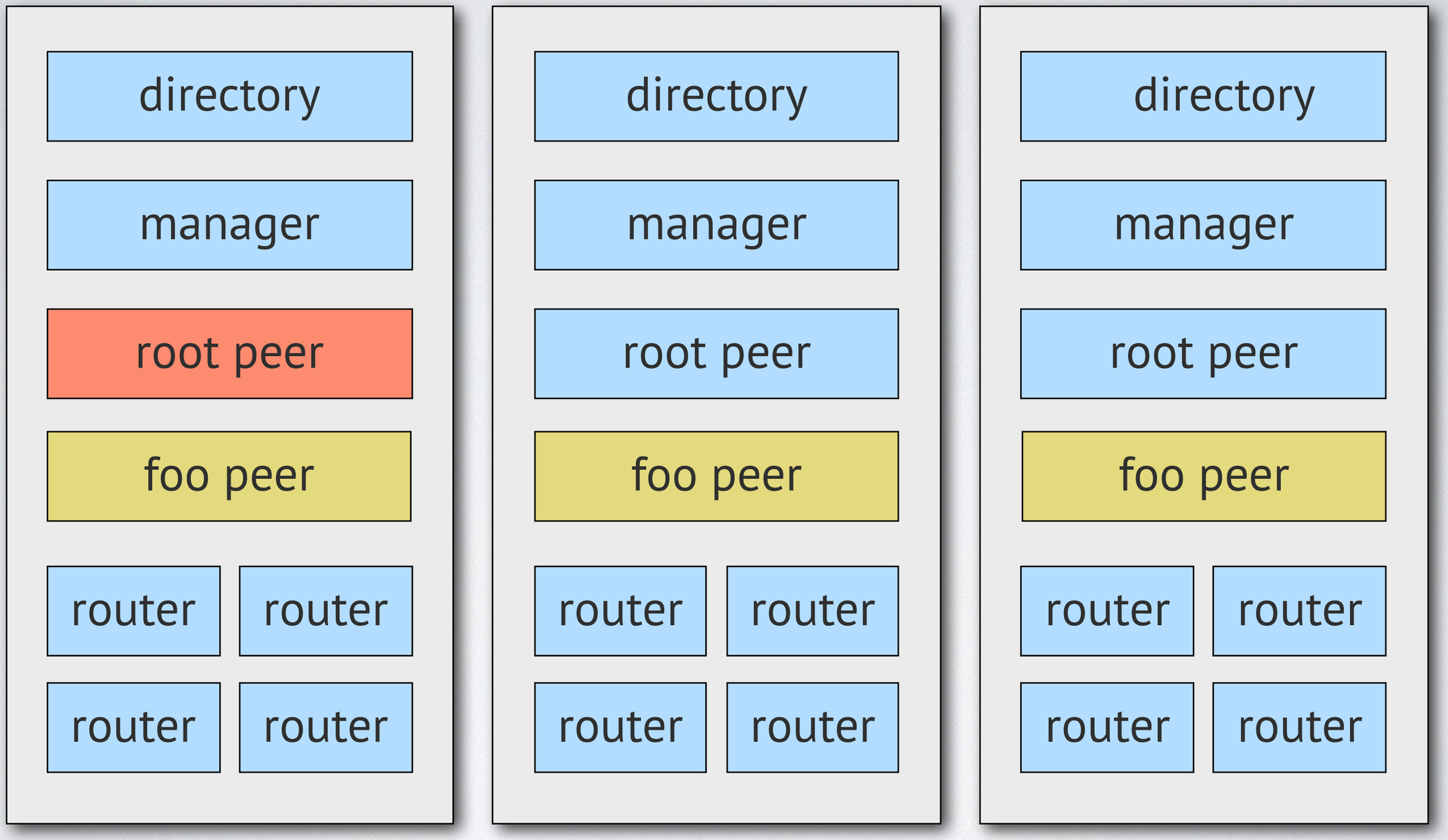
riak


directory
manager
root peer
directory
manager
root peer
directory
manager
root peer
router
router
router
router
router
router
router
router
router
router
router
router


directory
manager
root peer
router
router
router
router
directory
manager
root peer
router
router
router
router
directory
manager
root peer
router
router
router
router

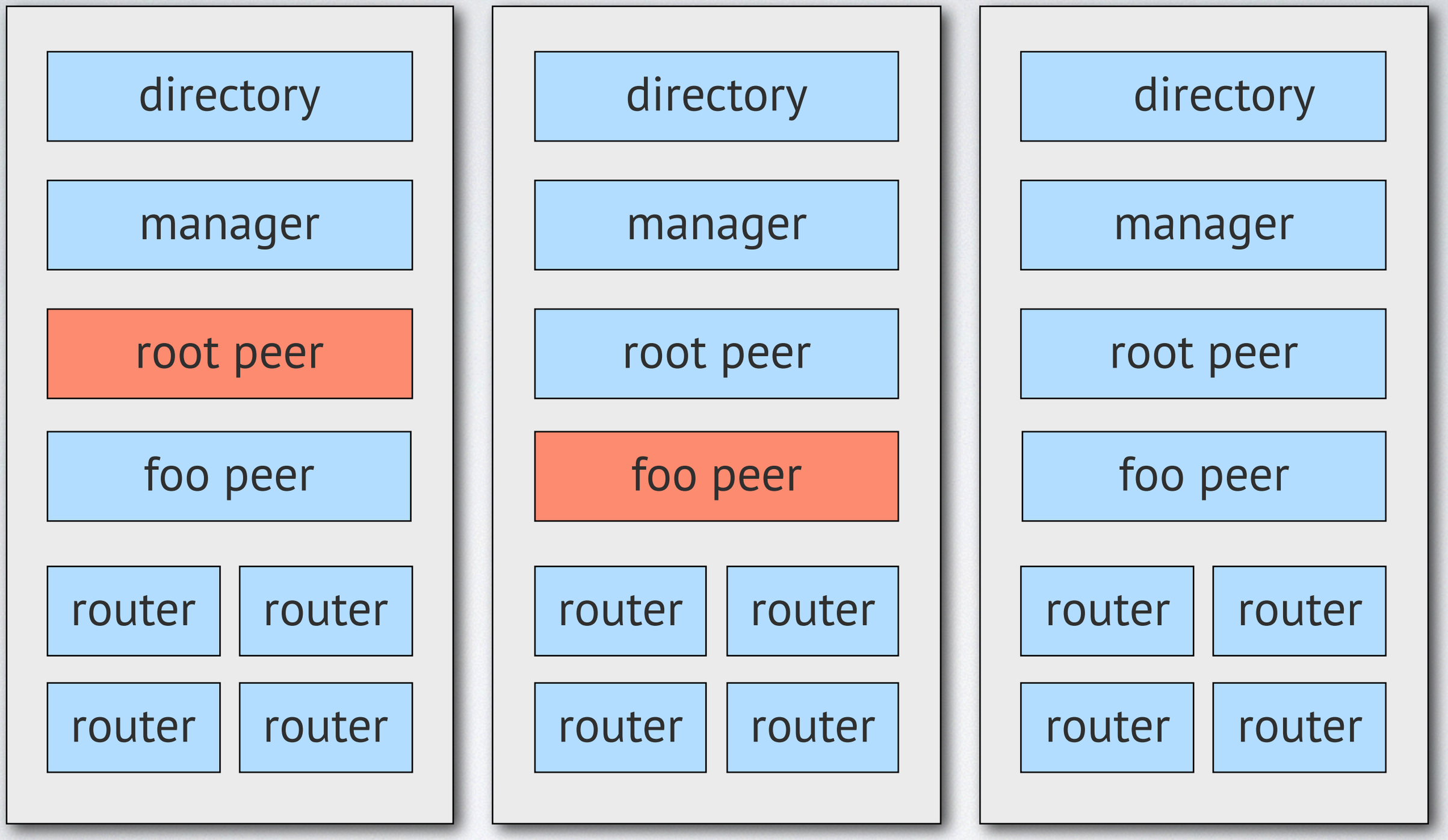

directory
manager
root peer
foo peer
router
router
router
router
directory
manager
root peer
foo peer
router
router
router
router
directory
manager
root peer
foo peer
router
router
router
router

election

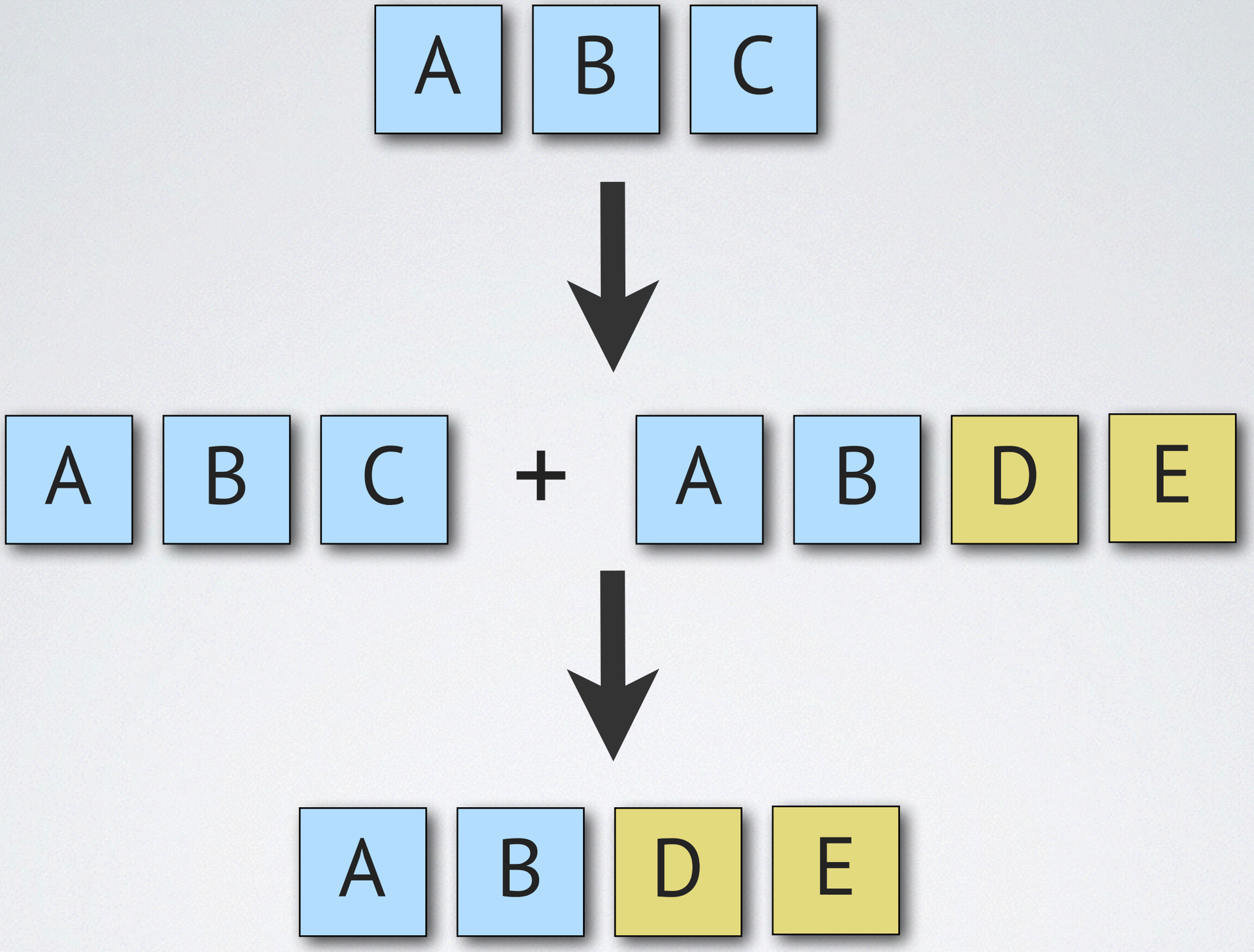


basho

riak


directory
manager
root peer
foo peer
router
router
router
router
directory
manager
root peer
foo peer
router
router
router
router
directory
manager
root peer
foo peer
router
router
router
router

Membership



basho

riak

riak_ensemble

Paxos framework for scalable consistent system

basho

riak

Questions?