A PEEK INSIDE RIAK

Steve Vinoski
Basho Technologies
Cambridge, MA USA
http://basho.com
@stevevinoski
vinoski@ieee.org
http://steve.vinoski.net/

Riak

- A distributed highly available eventually consistent highly scalable open source key-value database written primarily in Erlang.

https://github.com/basho/riak

Why Erlang?

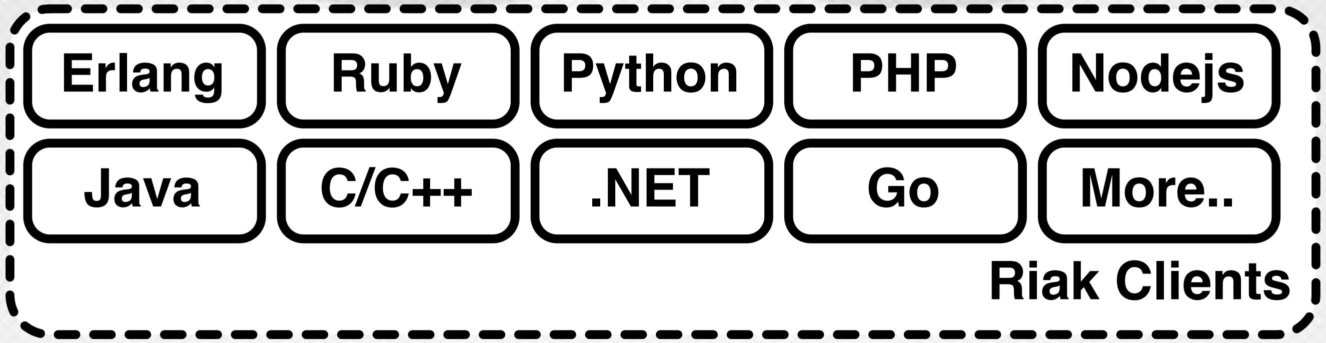
- See Basho CTO Justin Sheehy's recent blog post on why Basho uses Erlang:

  http://basho.com/erlang-at-basho-five-years-later/

Riak

- Modeled after Amazon Dynamo, see http://docs.basho.com/riak/latest/references/dynamo/
- Also provides MapReduce, secondary indexes, and full-text search
- Built for operational ease

Riak Architecture



Erlang Ruby Python PHP Nodejs

Java C/C++ .NET Go More..

Riak Clients

image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

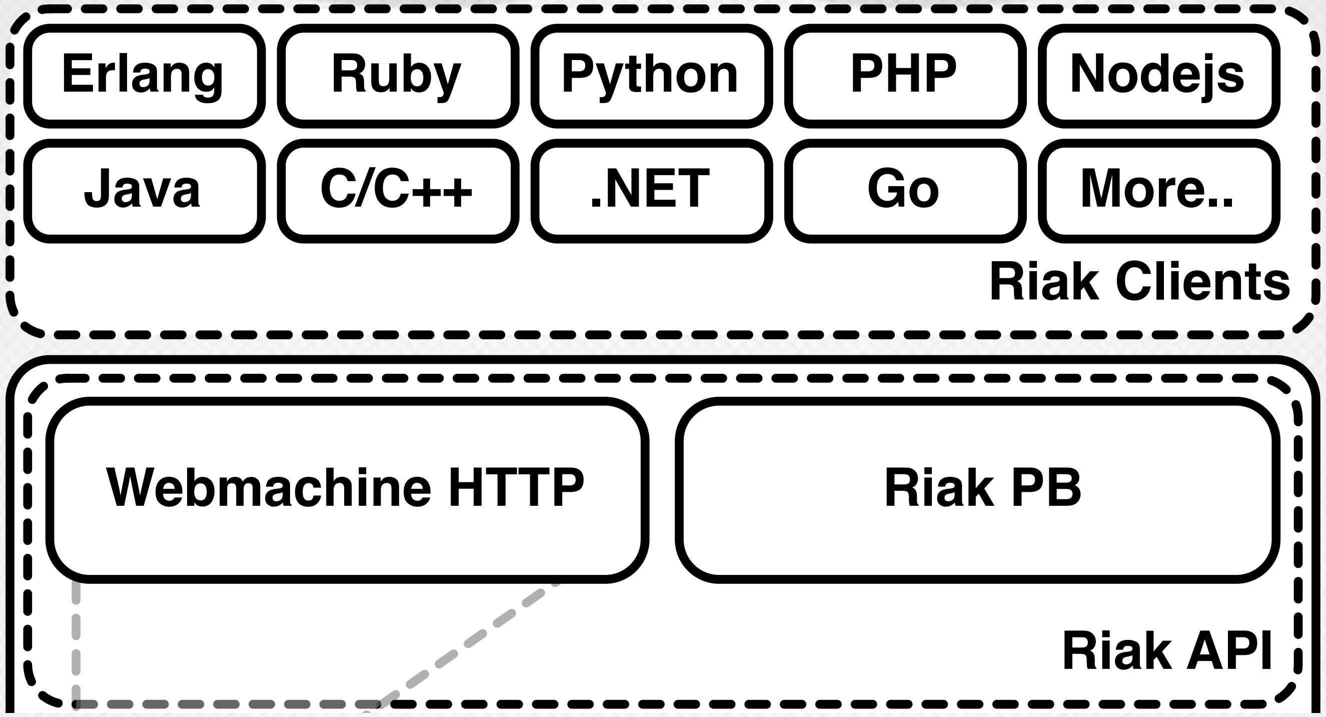


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

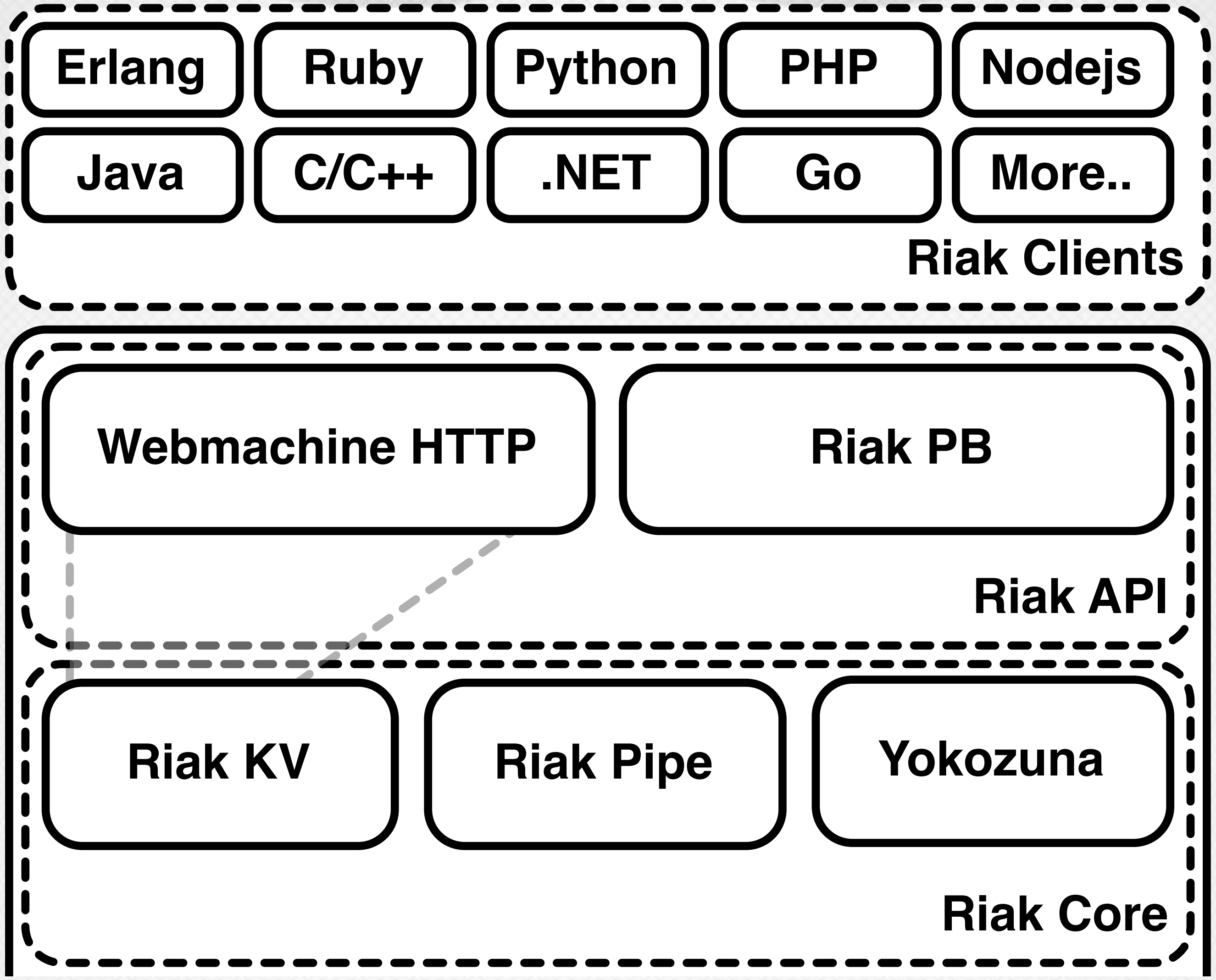


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

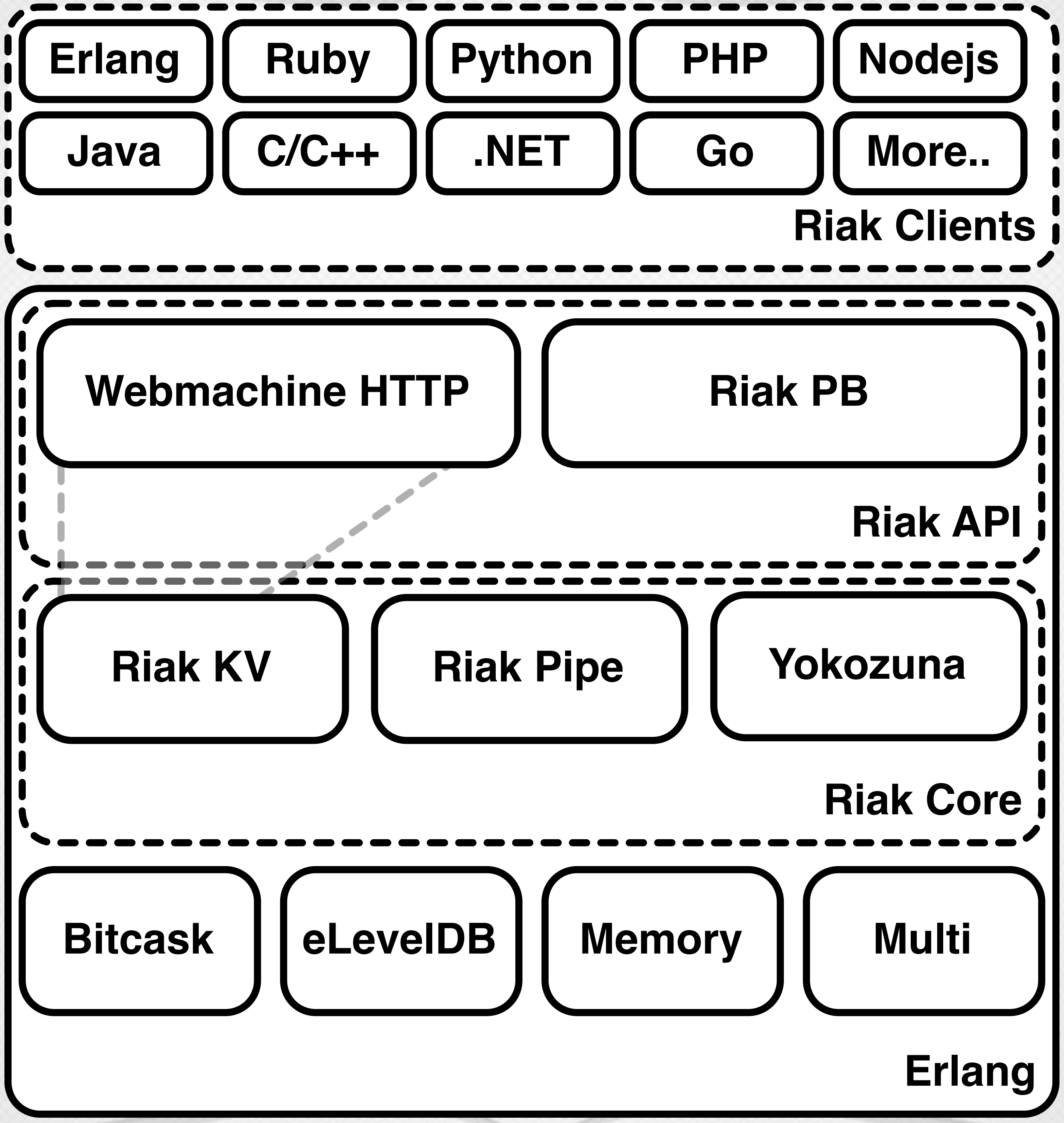
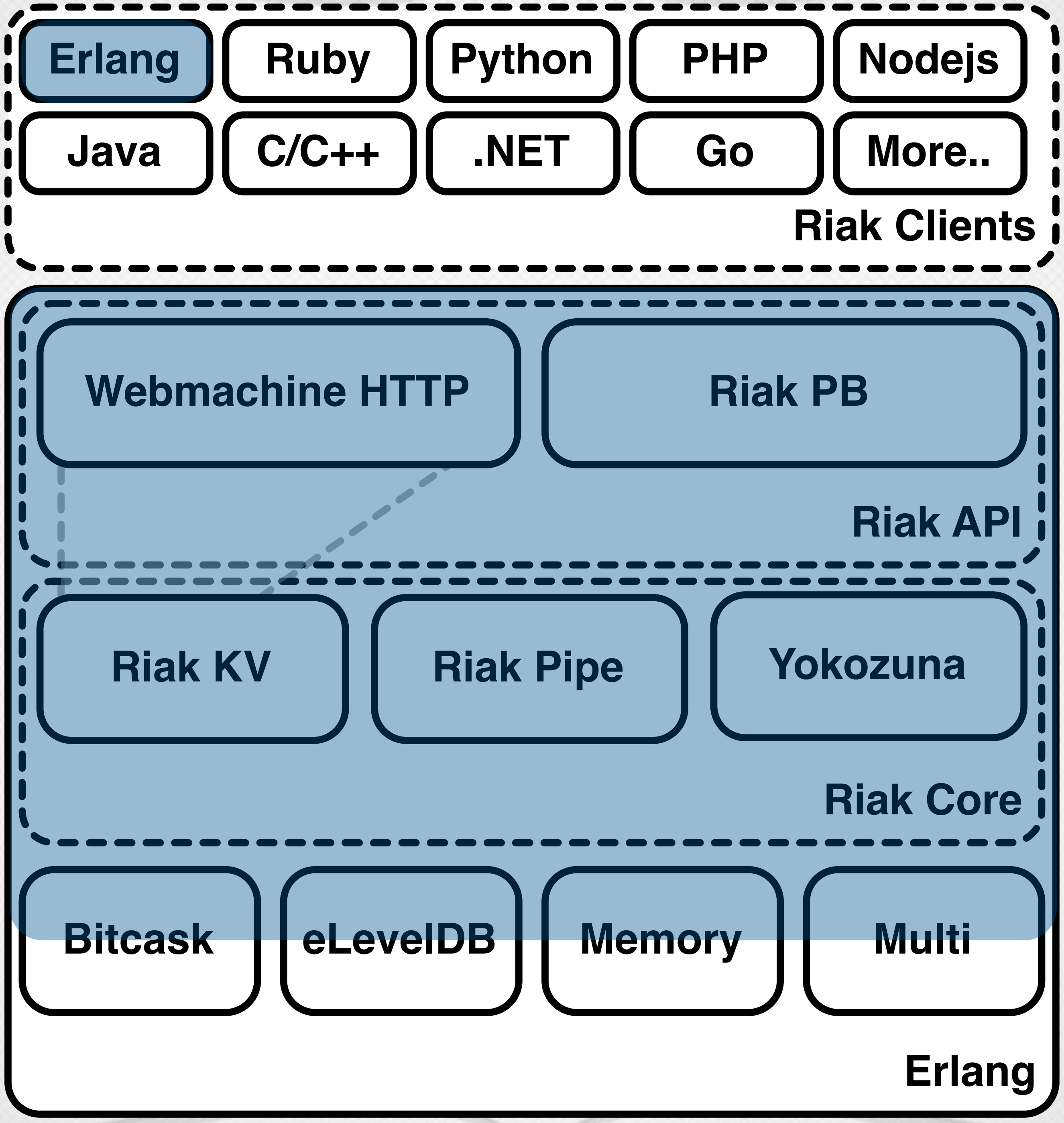


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

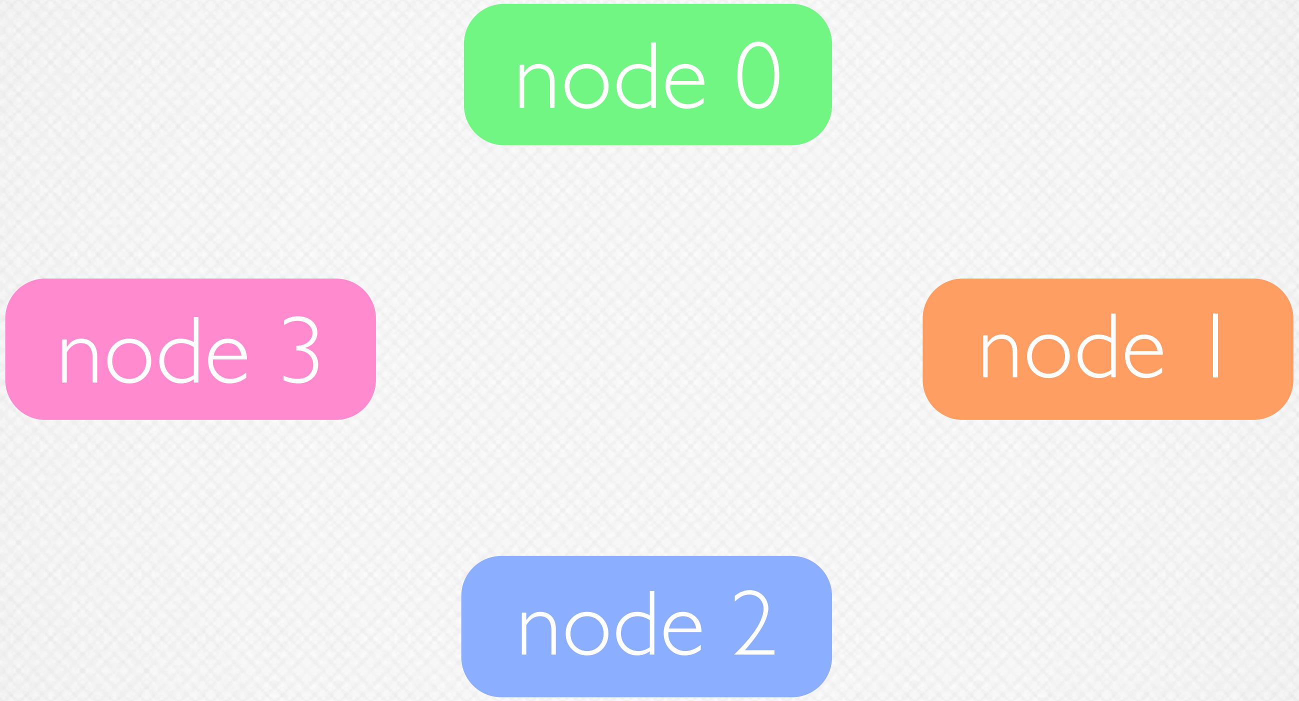
Riak Architecture



- Erlang parts

image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Cluster



node 0

node 3

node 1

node 2

Distributing Data

- Riak uses **consistent hashing** to spread data across the cluster
- Minimizes remapping of keys when number of nodes changes
- Spreads data evenly and minimizes hotspots

node 0

node 1

node 2

node 3

Consistent Hashing

- Riak uses SHA-1 as a hash function
- Treats its 160-bit value space as a ring
- Divides the ring into partitions called "virtual nodes" or vnodes (default 64)
- Each vnode claims a portion of the ring space
- Each physical node in the cluster hosts multiple vnodes

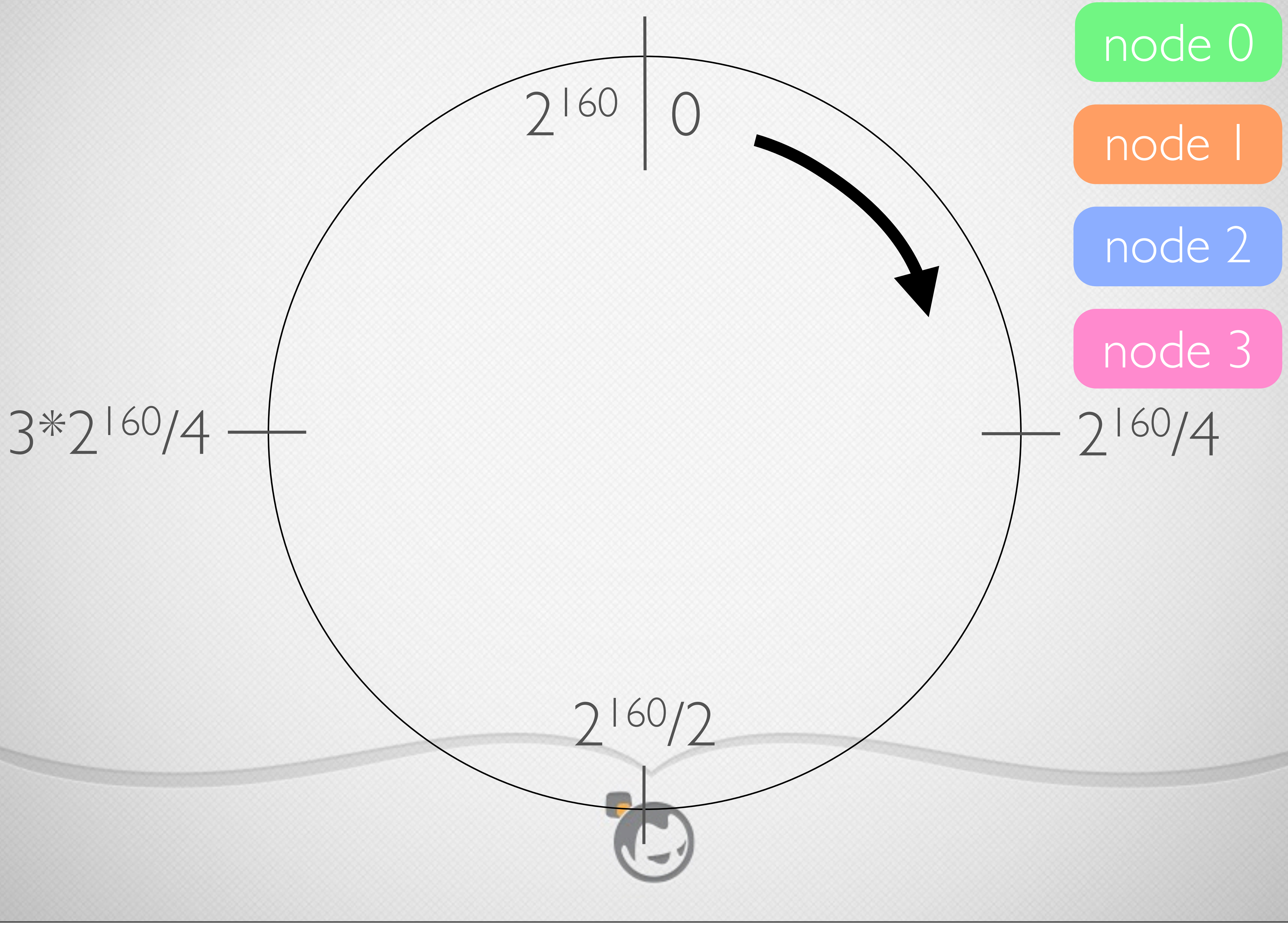
node 0

node 1

node 2

node 3

Hash Ring



2^{160} 0

$2^{160}/4$

$2^{160}/2$

$3*2^{160}/4$

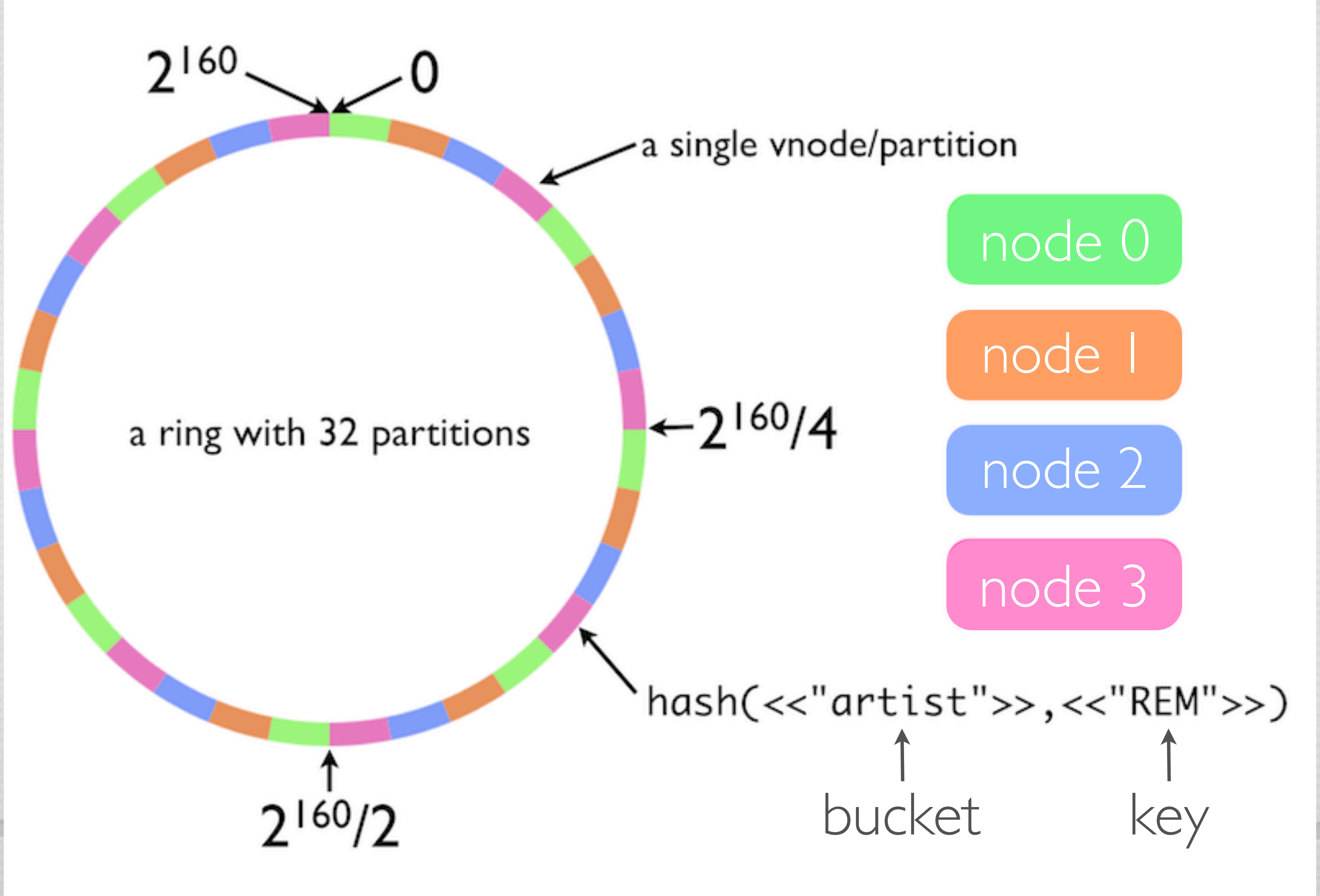
node 0

node 1

node 2

node 3

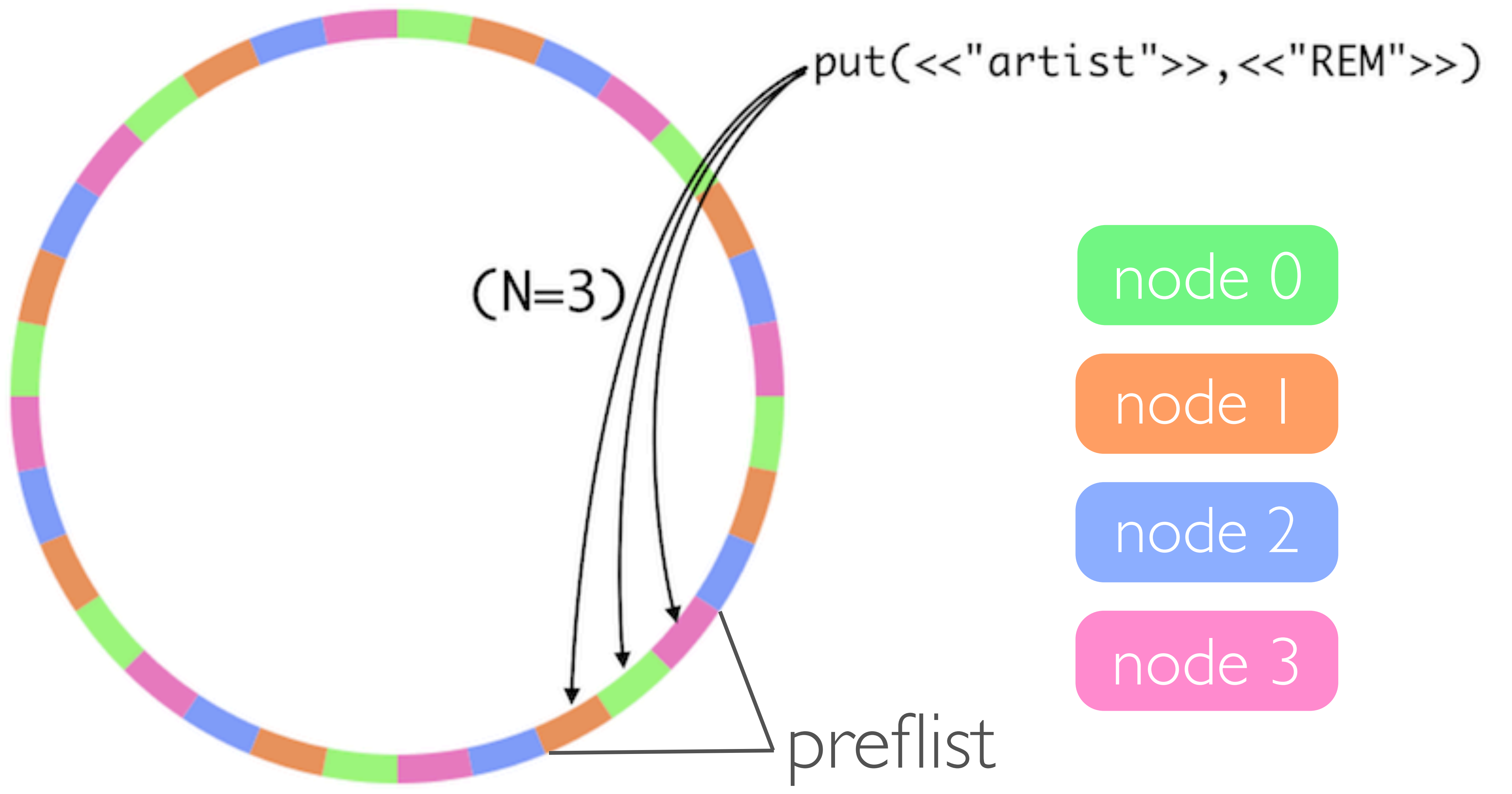
Hash Ring


2^{160}
0
a single vnode/partition
a ring with 32 partitions
$2^{160}/4$
$2^{160}/2$
node 0
node 1
node 2
node 3
hash(<<"artist">>,<<"REM">>)
bucket
key

N/R/W Values

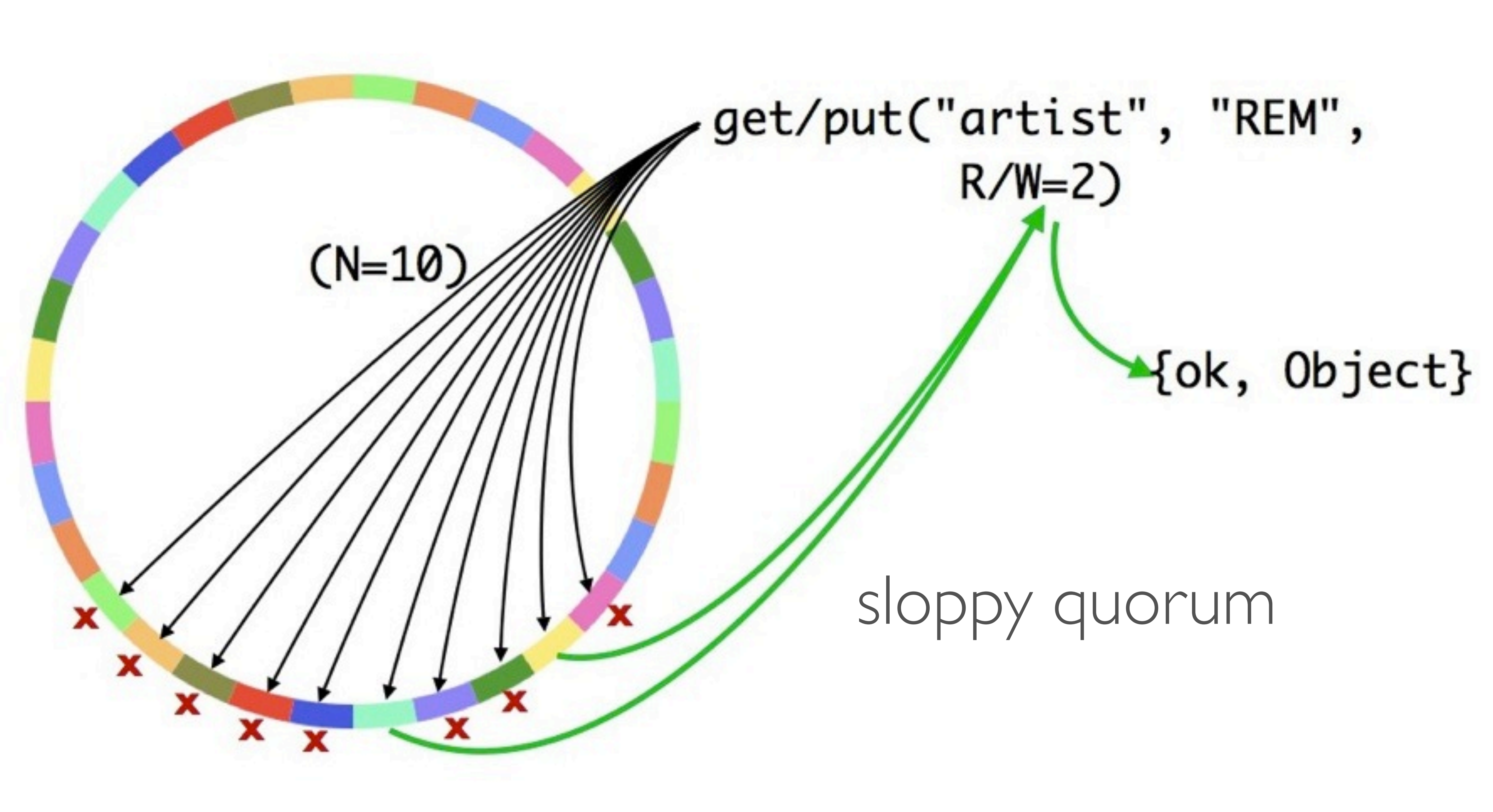
- N = number of replicas to store (default 3, can be set per bucket)
- R = read quorum = number of replica responses needed for a successful read (can be specified per-request)
- W = write quorum = number of replica responses needed for a successful write (can be specified per-request)

N/R/W Values



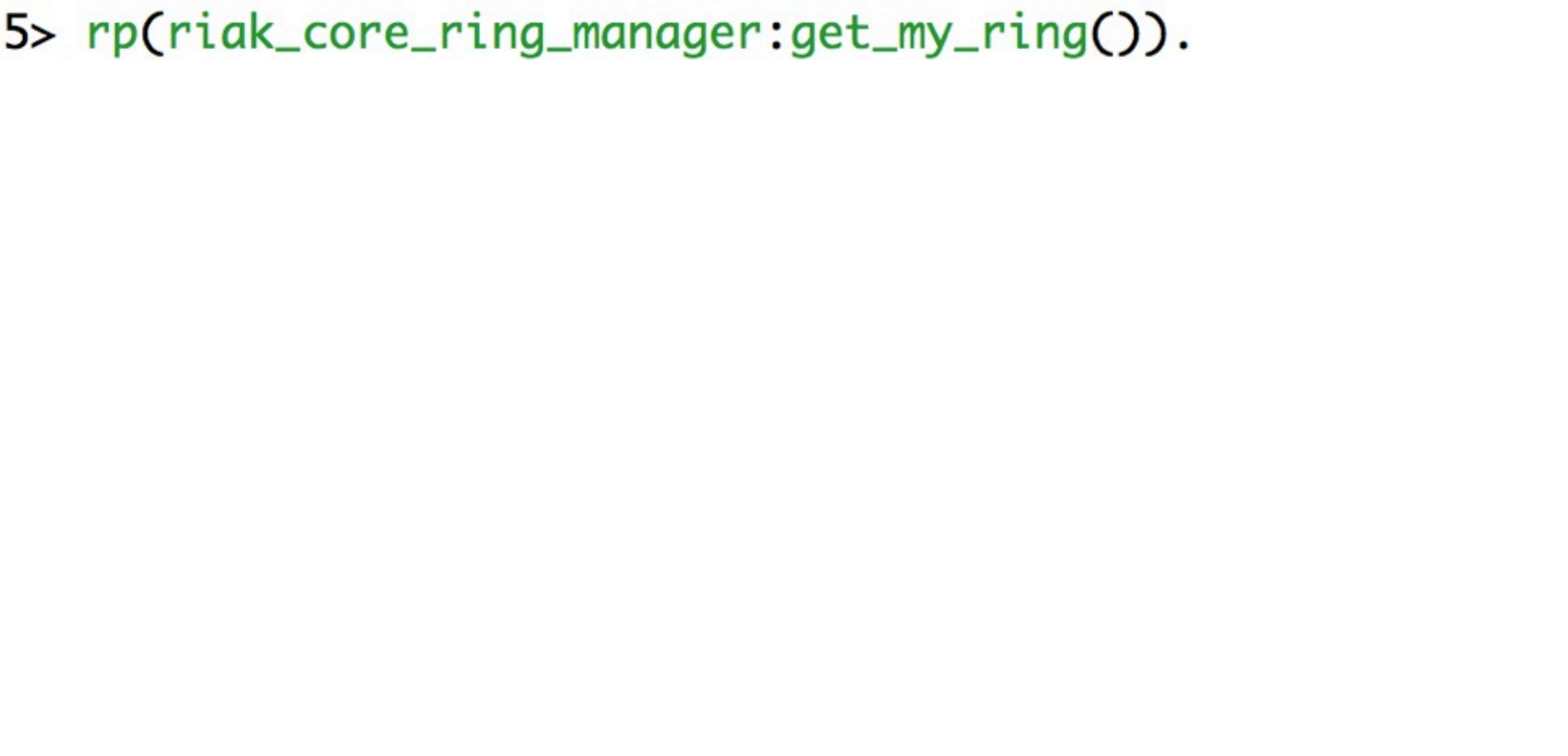
for details see http://docs.basho.com/riak/latest/dev/advanced/cap-controls/

N/R/W Values



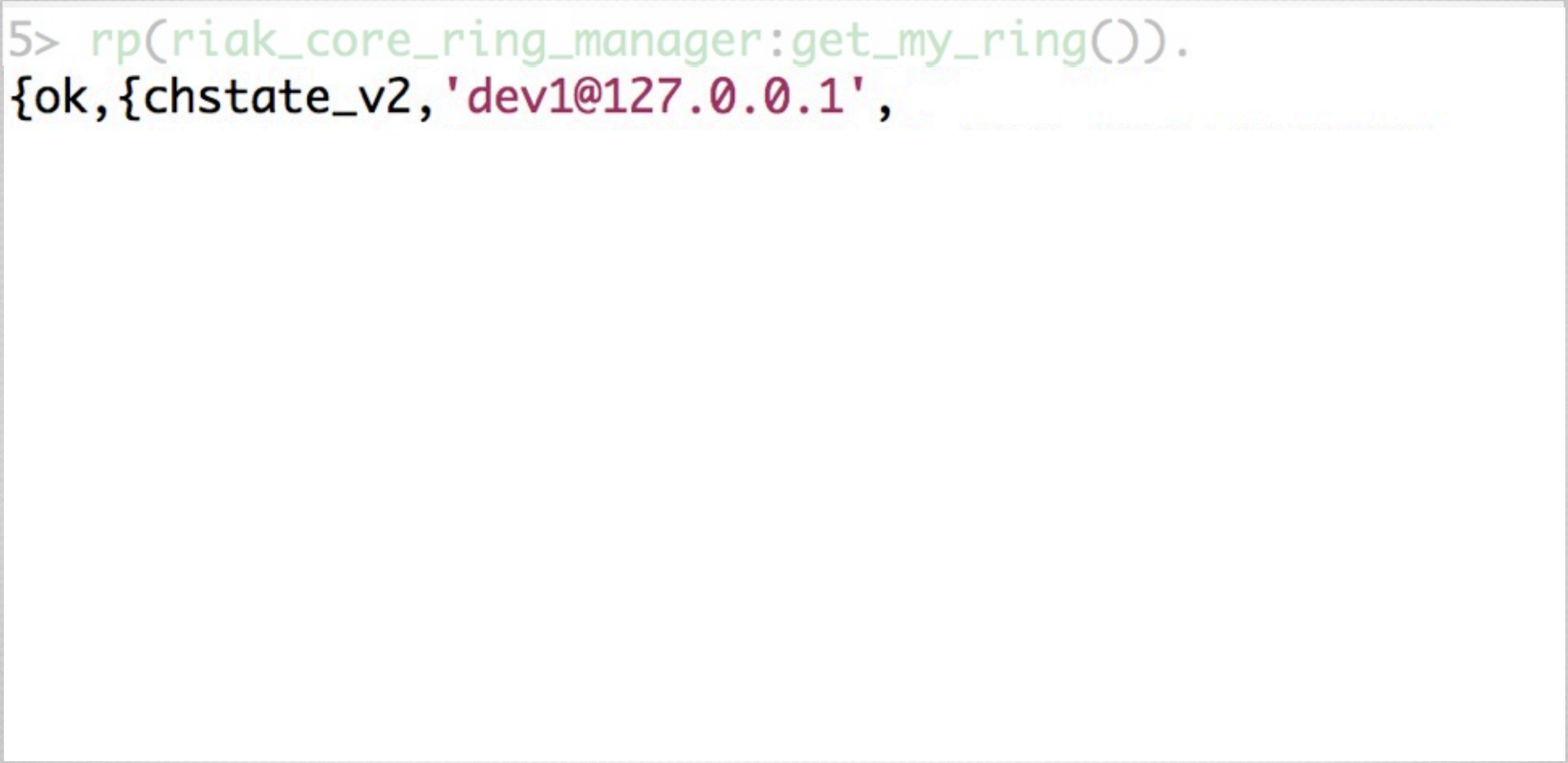
sloppy quorum

Riak's Ring



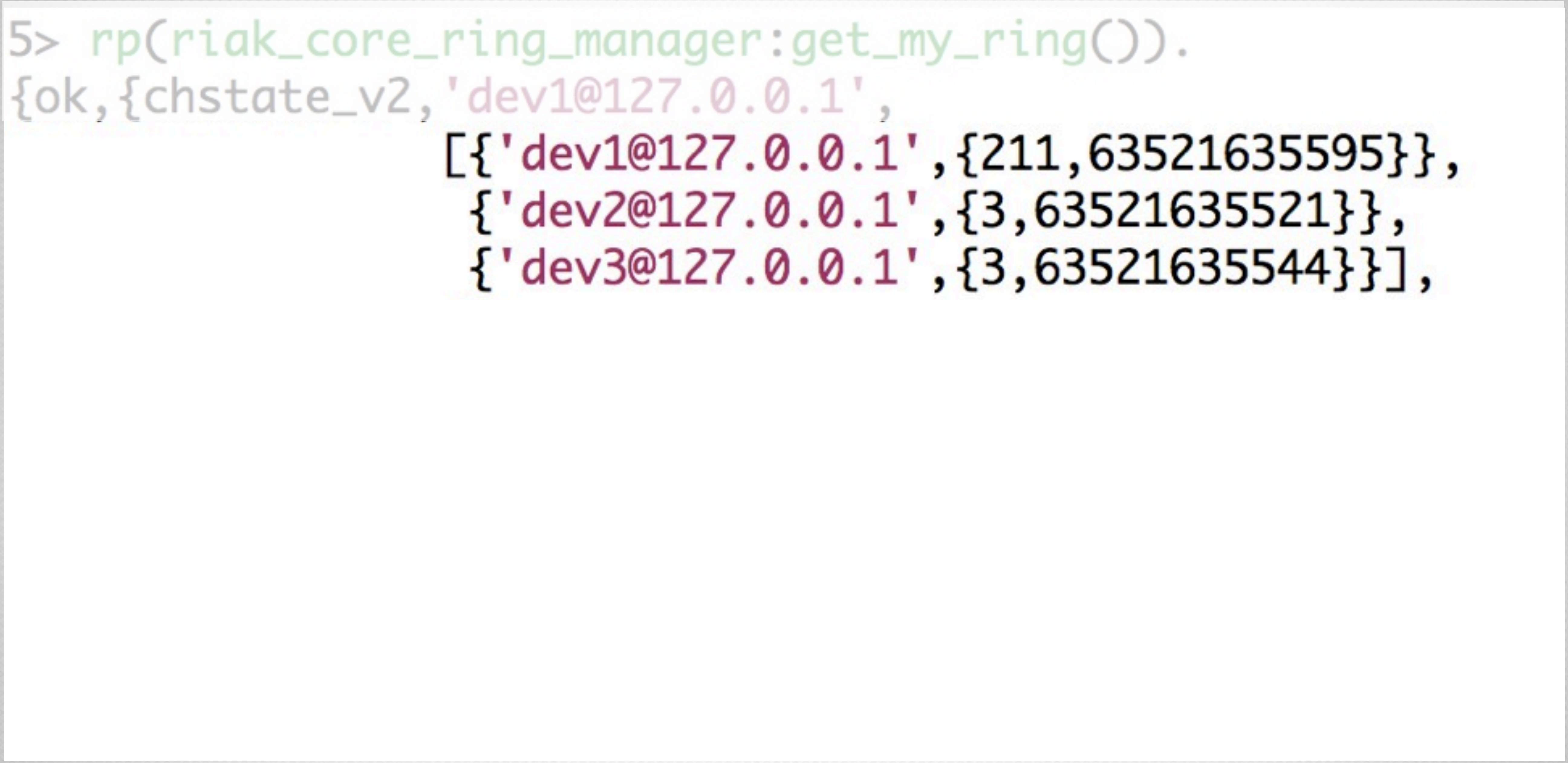
```
5> rp(riak_core_ring_manager:get_my_ring()).
```

Riak's Ring



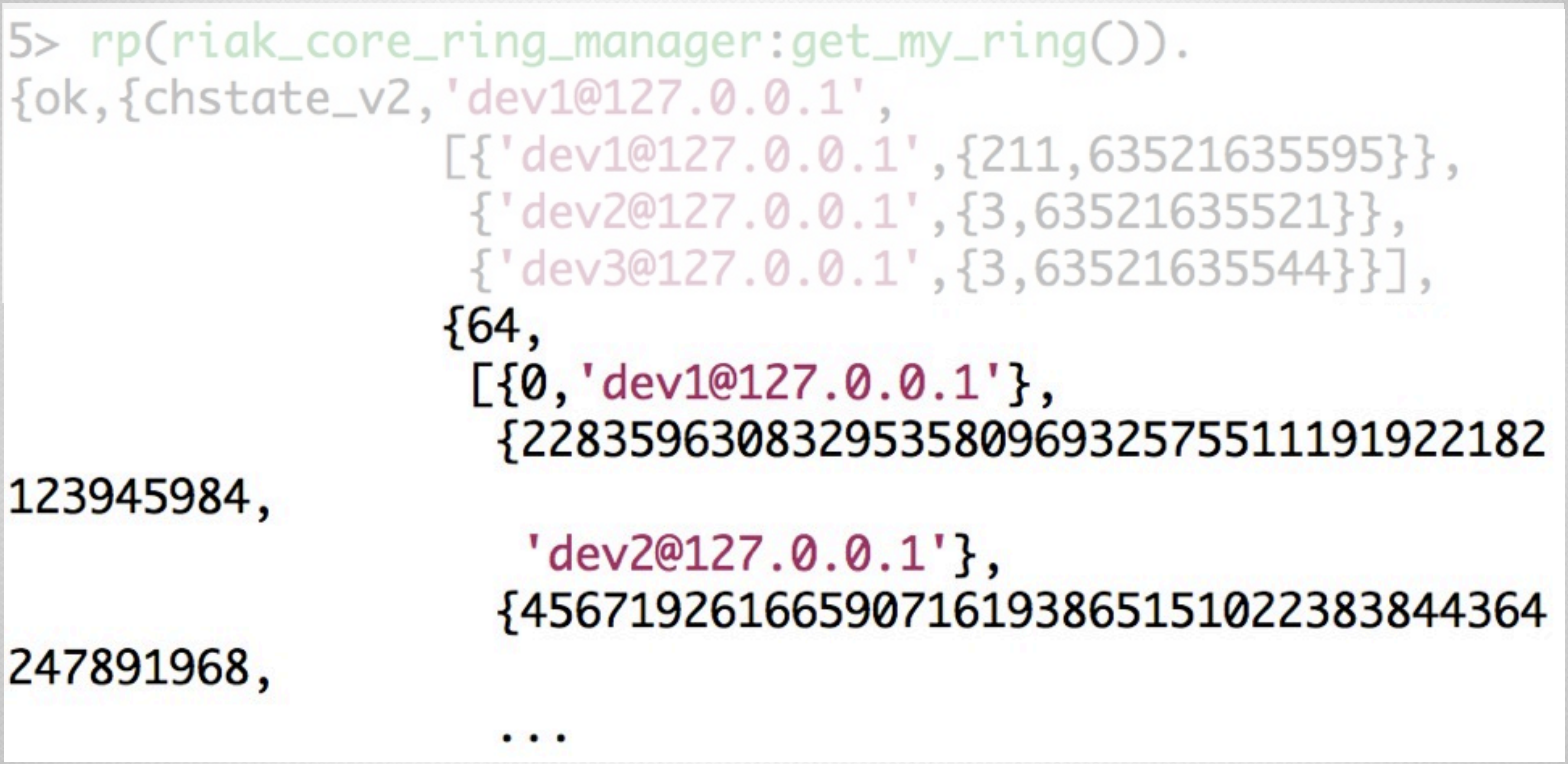
```
5> rp(riak_core_ring_manager:get_my_ring()).
{ok,{chstate_v2,'dev1@127.0.0.1',
```

Riak's Ring



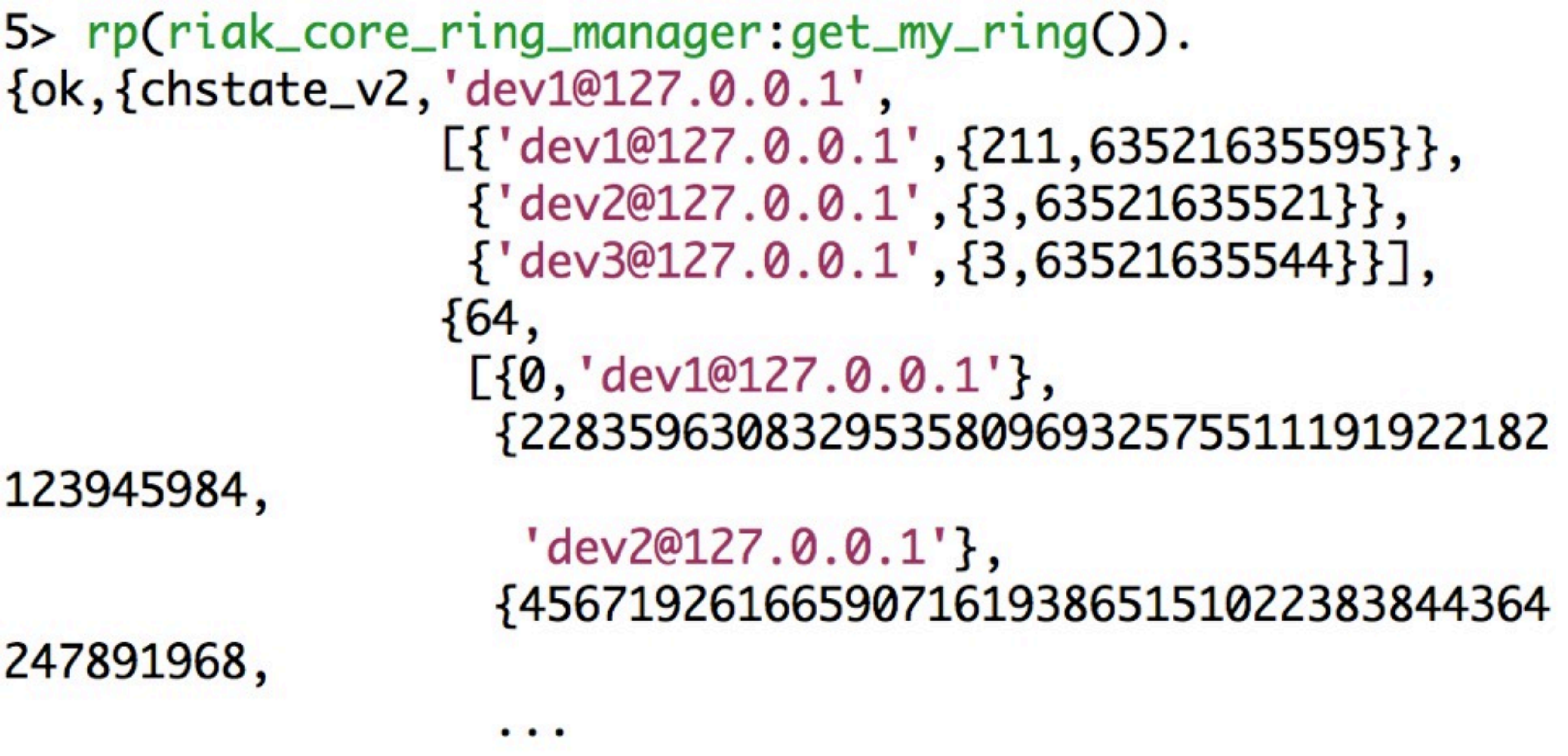
```
5> rp(riak_core_ring_manager:get_my_ring()).
{ok,{chstate_v2,'dev1@127.0.0.1',
                [{'dev1@127.0.0.1',{211,63521635595}},
                 {'dev2@127.0.0.1',{3,63521635521}},
                 {'dev3@127.0.0.1',{3,63521635544}}],
```

Riak's Ring



```
5> rp(riak_core_ring_manager:get_my_ring()).
{ok,{chstate_v2,'dev1@127.0.0.1',
                [{'dev1@127.0.0.1',{211,63521635595}},
                 {'dev2@127.0.0.1',{3,63521635521}},
                 {'dev3@127.0.0.1',{3,63521635544}}],
                {64,
                 [{0,'dev1@127.0.0.1'},
                  {22835963083295358096932575511191922182
123945984,
                   'dev2@127.0.0.1'},
                  {45671926166590716193865151022383844364
247891968,
                  ...
```

Riak's Ring



```
5> rp(riak_core_ring_manager:get_my_ring()).
{ok,{chstate_v2,'dev1@127.0.0.1',
                [{'dev1@127.0.0.1',{211,63521635595}},
                 {'dev2@127.0.0.1',{3,63521635521}},
                 {'dev3@127.0.0.1',{3,63521635544}}],
                {64,
                 [{0,'dev1@127.0.0.1'},
                  {22835963083295358096932575511191922182
123945984,
                   'dev2@127.0.0.1'},
                  {45671926166590716193865151022383844364
247891968,
                  ...
```

Ring State

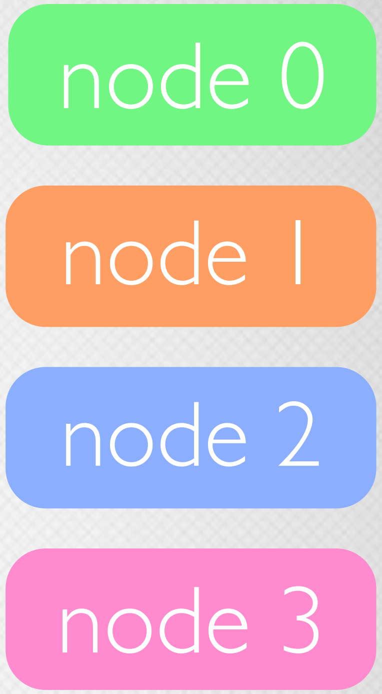
- All nodes in a Riak cluster are peers, no masters or slaves
- Nodes exchange their understanding of ring state via a gossip protocol

Distributed Erlang

- Erlang has distribution built in — it's required for supporting multiple nodes for reliability
- By default Erlang nodes form a mesh, every node knows about every other node
- Riak uses this for intra-cluster communication

Distributed Erlang

- Riak lets you simulate a multi-node installment on a single machine, nice for development
- "make devrel" or "make stagedevrel" in a riak repository clone (git://github.com/basho/riak.git)
- Let's assume we have nodes dev1, dev2, and dev3 running in a cluster, nothing on the 4th node yet
- Instead of starting riak, let's start the 4th node as just a plain distributed erlang node



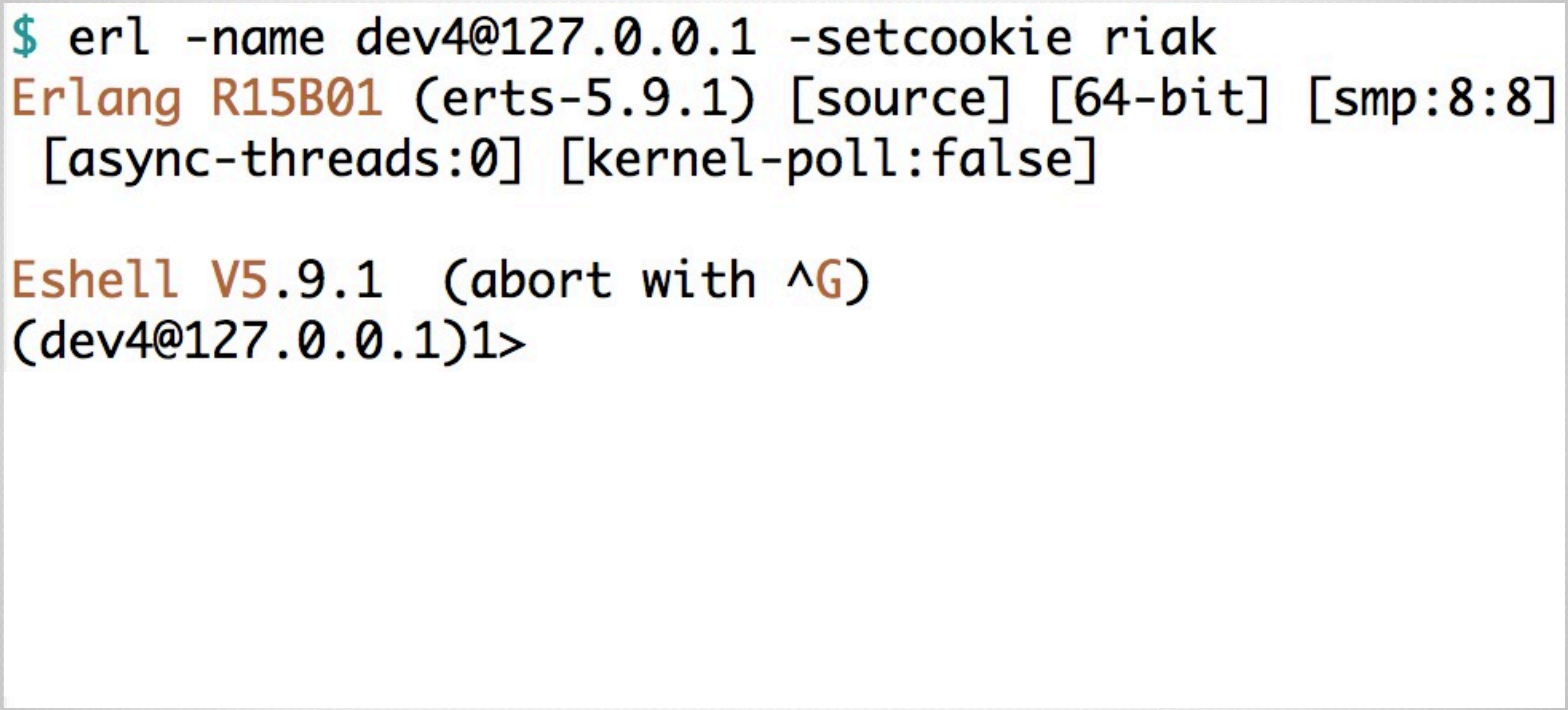
node 0

node 1

node 2

node 3

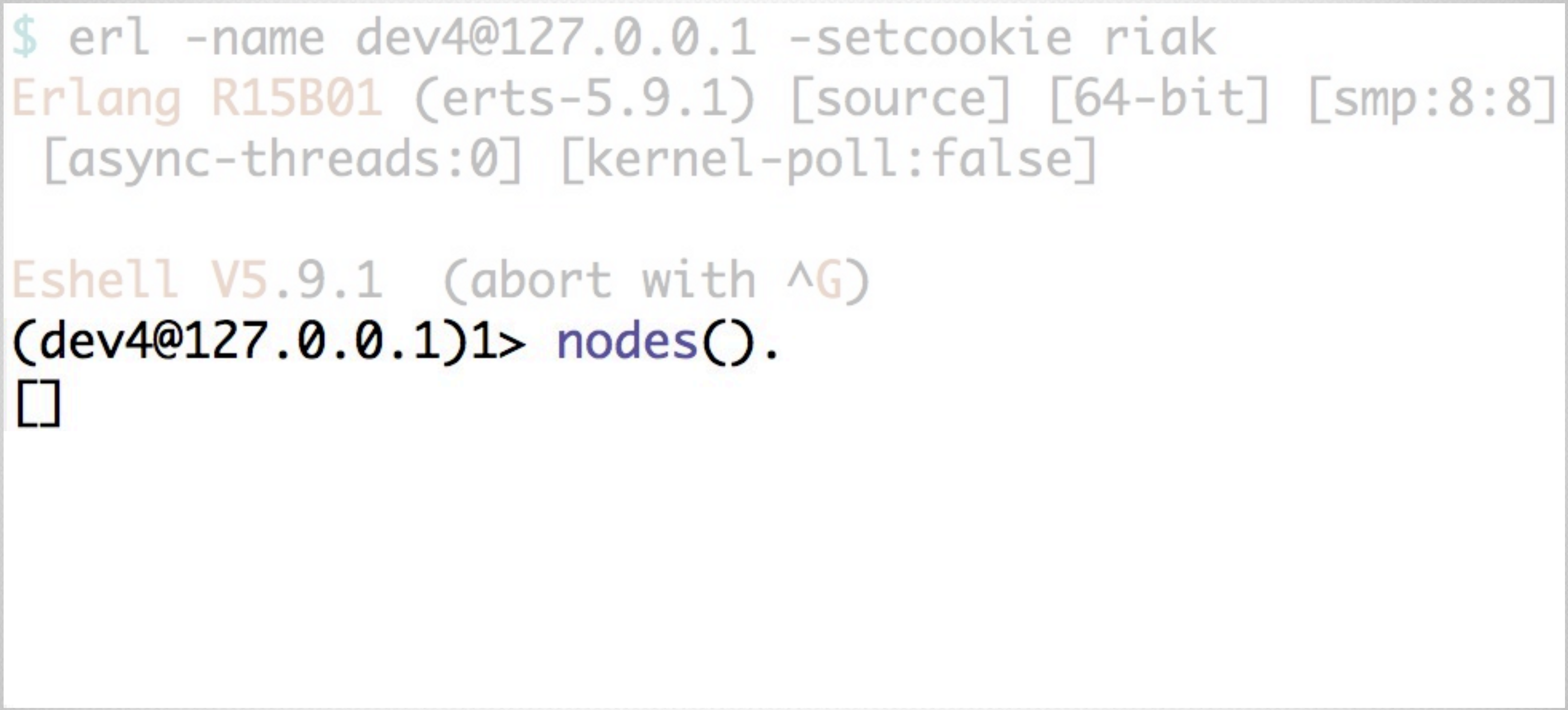
Distributed Erlang



```
$ erl -name dev4@127.0.0.1 -setcookie riak
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]
 [async-threads:0] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
(dev4@127.0.0.1)1>
```

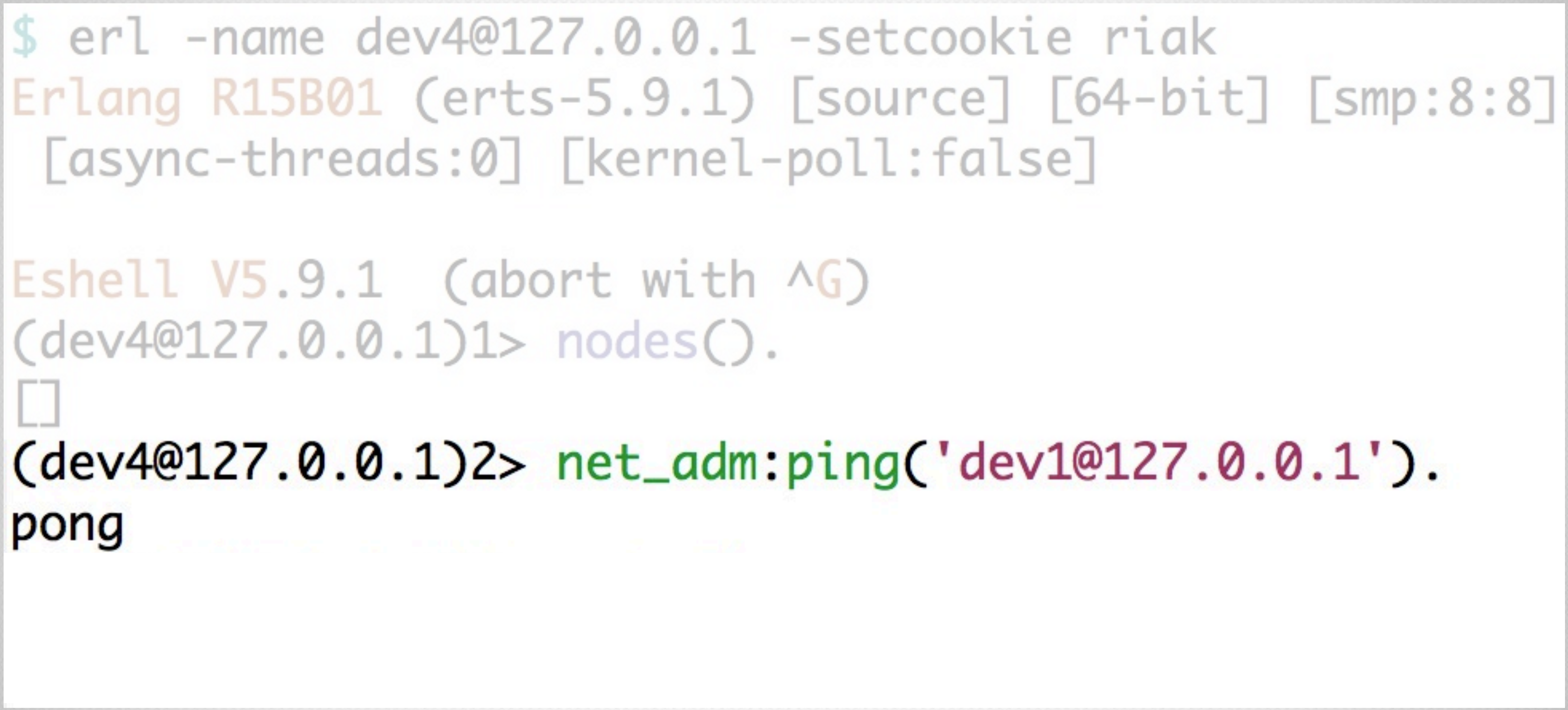
Distributed Erlang



```
$ erl -name dev4@127.0.0.1 -setcookie riak
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]
 [async-threads:0] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
(dev4@127.0.0.1)1> nodes().
[]
```

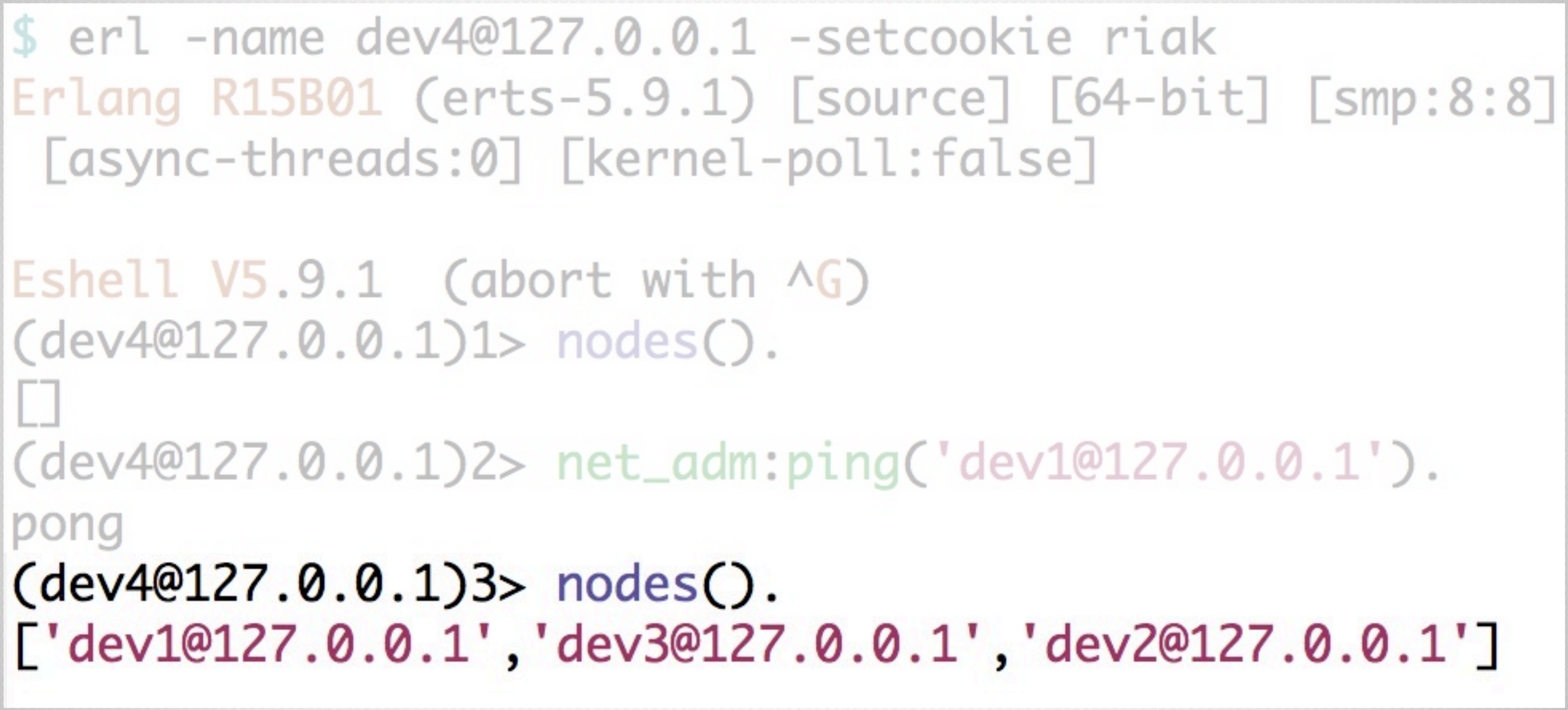
Distributed Erlang



```
$ erl -name dev4@127.0.0.1 -setcookie riak
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]
 [async-threads:0] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
(dev4@127.0.0.1)1> nodes().
[]
(dev4@127.0.0.1)2> net_adm:ping('dev1@127.0.0.1').
pong
```

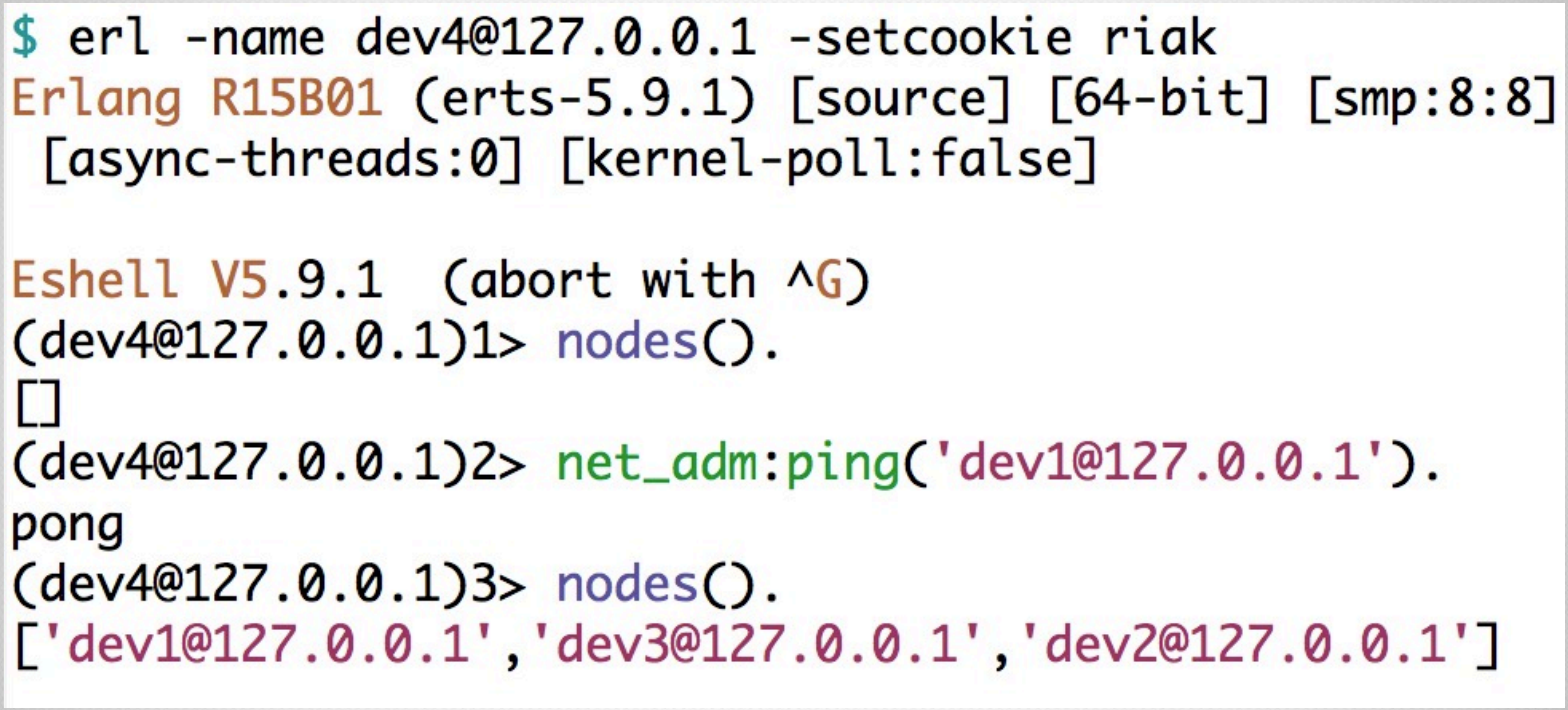
Distributed Erlang



```
$ erl -name dev4@127.0.0.1 -setcookie riak
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]
 [async-threads:0] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
(dev4@127.0.0.1)1> nodes().
[]
(dev4@127.0.0.1)2> net_adm:ping('dev1@127.0.0.1').
pong
(dev4@127.0.0.1)3> nodes().
['dev1@127.0.0.1','dev3@127.0.0.1','dev2@127.0.0.1']
```

Distributed Erlang

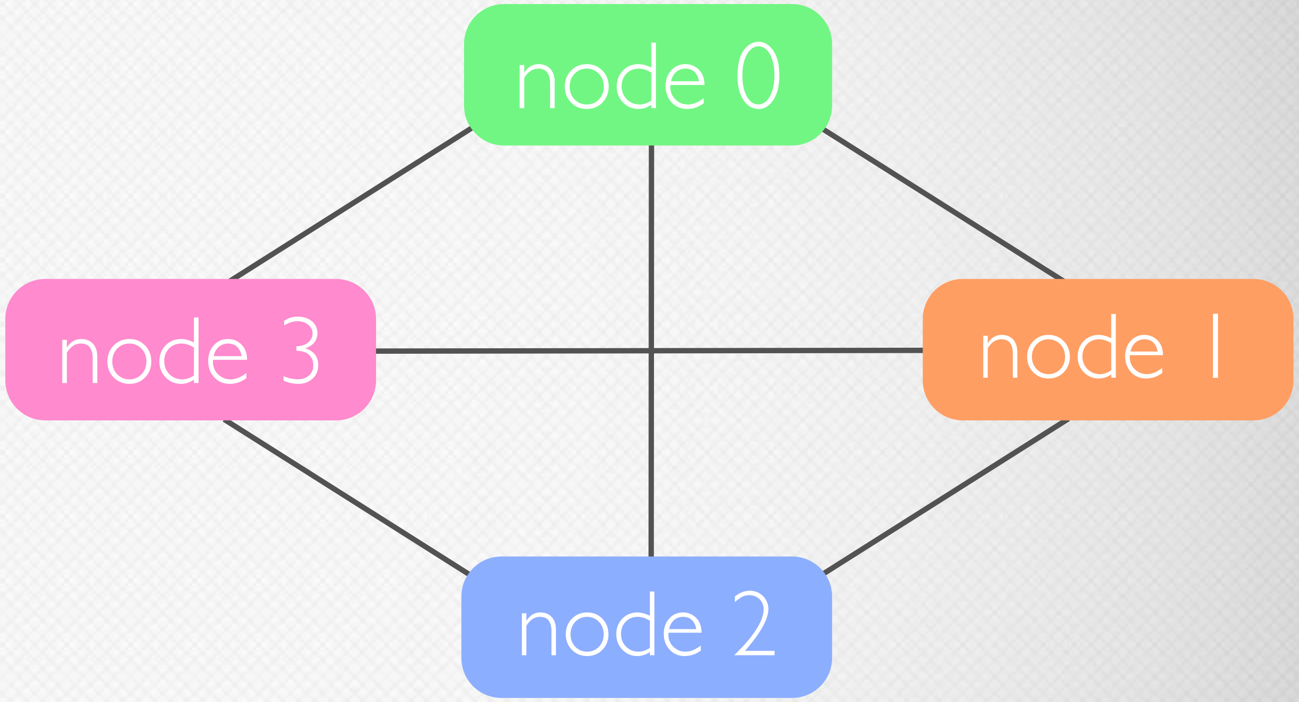


```
$ erl -name dev4@127.0.0.1 -setcookie riak
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]
 [async-threads:0] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
(dev4@127.0.0.1)1> nodes().
[]
(dev4@127.0.0.1)2> net_adm:ping('dev1@127.0.0.1').
pong
(dev4@127.0.0.1)3> nodes().
['dev1@127.0.0.1','dev3@127.0.0.1','dev2@127.0.0.1']
```

Distributed Erlang Mesh

- Nodes talk to each other occasionally to check liveness
- Mesh approach makes it easy to set up a cluster
- Currently scales up to about 150 nodes, work underway to make it scale larger

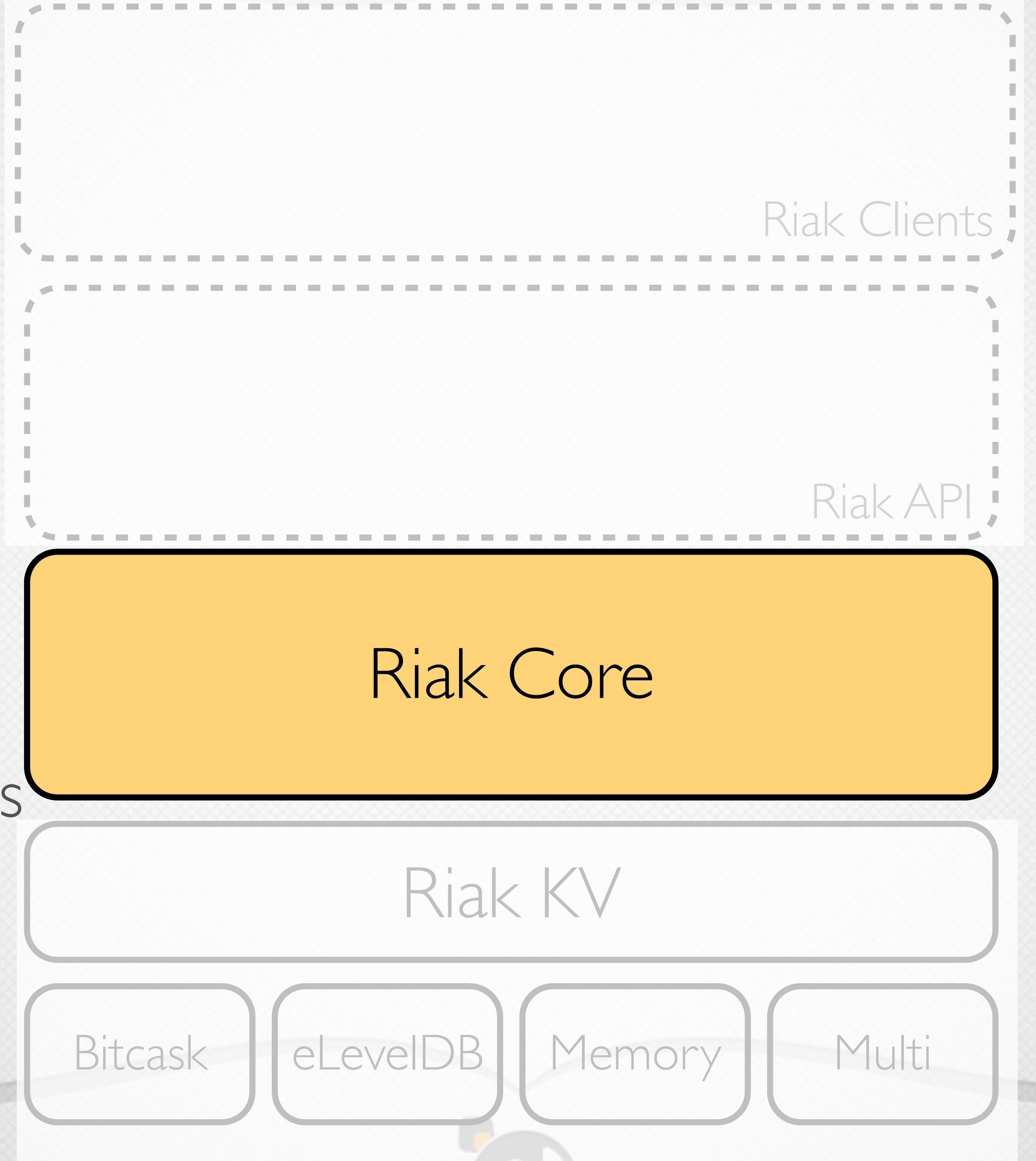


Gossip

- Riak nodes are peers, there's no master
- But the ring has state, such as what vnodes each node has claimed
- Nodes periodically send their understanding of the ring state to other randomly chosen nodes
- Riak gossip module also provides an API for sending ring state to specific nodes

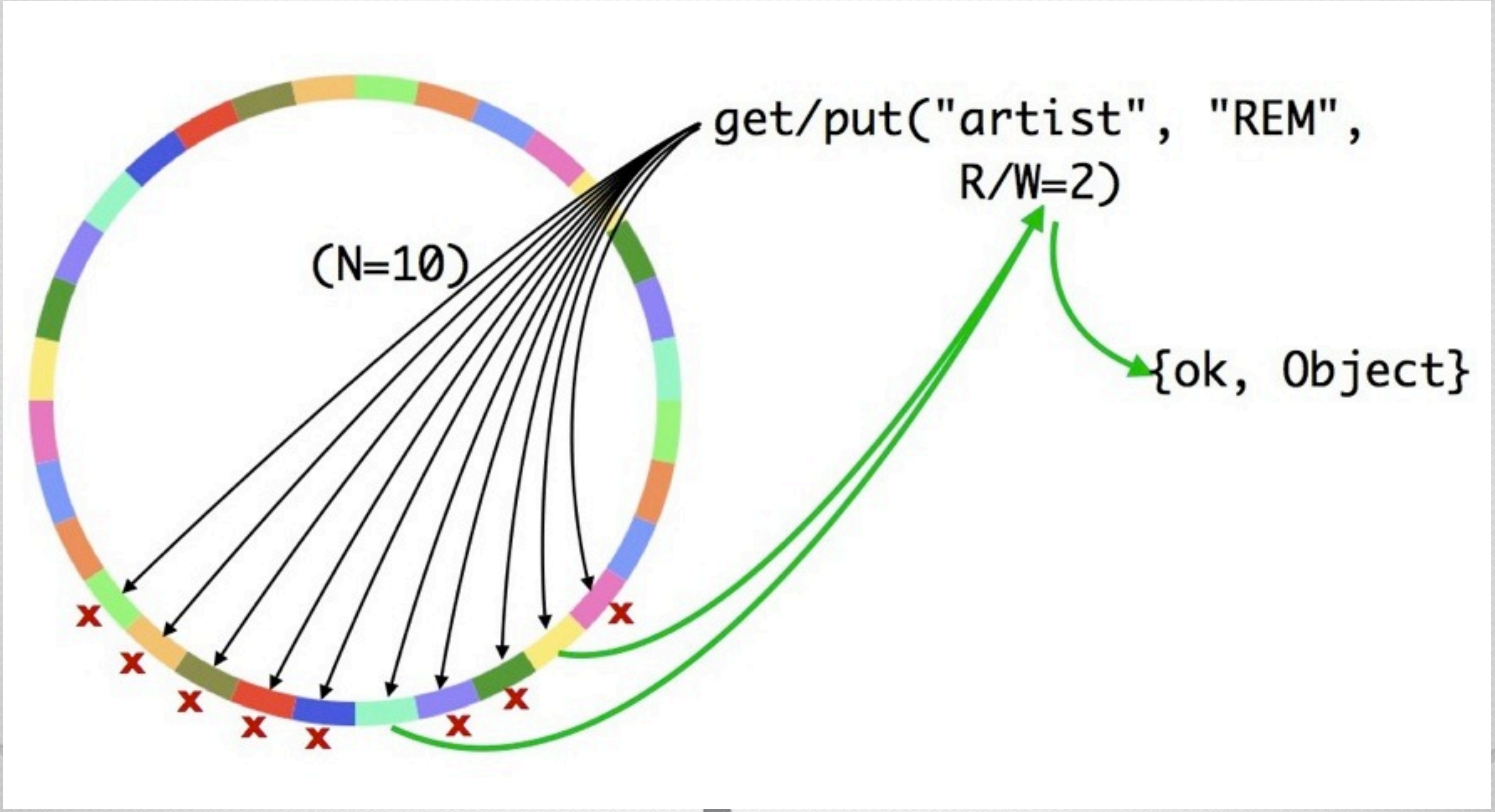
Riak Core

- consistent hashing
- vector clocks
- sloppy quorums



- gossip protocols
- virtual nodes (vnodes)
- hinted handoff

N/R/W Values



Hinted Handoff

- Fallback vnode holds data for unavailable primary vnode
- Fallback vnode keeps checking for availability of primary vnode
- Once primary vnode becomes available, fallback hands off data to it
- Fallback vnodes are started as needed, thanks to Erlang lightweight processes

Read Repair

- If a read detects a vnode with stale data, it is repaired via asynchronous update
- Helps implement eventual consistency
- Riak supports active anti-entropy (AAE) to actively repair stale values

Core Protocols

- Gossip, handoff, read repair, etc. all require intra-cluster protocols
- Erlang distribution and other features help significantly with protocol implementations
- Erlang monitors allow processes and nodes to watch each other while interacting
  - A monitoring process/node is notified if a monitored process/node dies, great for aborting failed interactions

Protocols With Erlang/OTP

- Erlang's Open Telecom Platform (OTP) provides libraries of standard modules
- And also **behaviors**: implementations of common patterns for concurrent, distributed, fault-tolerant Erlang apps

OTP Behavior Modules

- An OTP behavior is similar to an abstract base class in OO terms, providing:
  - a message handling tail-call optimized loop
  - integration with underlying OTP system for code upgrade, tracing, process management, etc.

OTP Behaviors

- application: plugs into Erlang application controller
- supervisor: manages and monitors worker processes
- gen_server: server process framework
- gen_fsm: finite state machine framework
- gen_event: event handling framework

Gen_server

- Generic server behavior for handling messages
- Supports server-like components, distributed or not
- “Business logic” lives in app-specific callback module
- Maintains state in a tail-call optimized receive loop

Gen_fsm

- Behavior supporting finite state machines (FSMs)
- Tail-call loop for maintaining state, like gen_server
- States and events handled by app-specific callback module
- Allows events to be sent into an FSM either sync or async

Riak And Gen_*

- Riak makes heavy use of these behaviors, e.g.:
    - FSMs for get and put operations
    - Vnode FSM
    - Gossip module is a gen_server

Riak Behaviors

- riak_kv_backend: behavior for storage backends
  - all storage backends have to provide the callback functions the riak_kv_backend behavior expects
  - checked at compile time
- riak_core_coverage_fsm: behavior to create and execute a plan to cover a set of vnodes, for example for secondary index queries or listing buckets
- riak_pipe_qcover_fsm: enqueue work on a covering set of vnodes

INTEGRATION

Riak Architecture

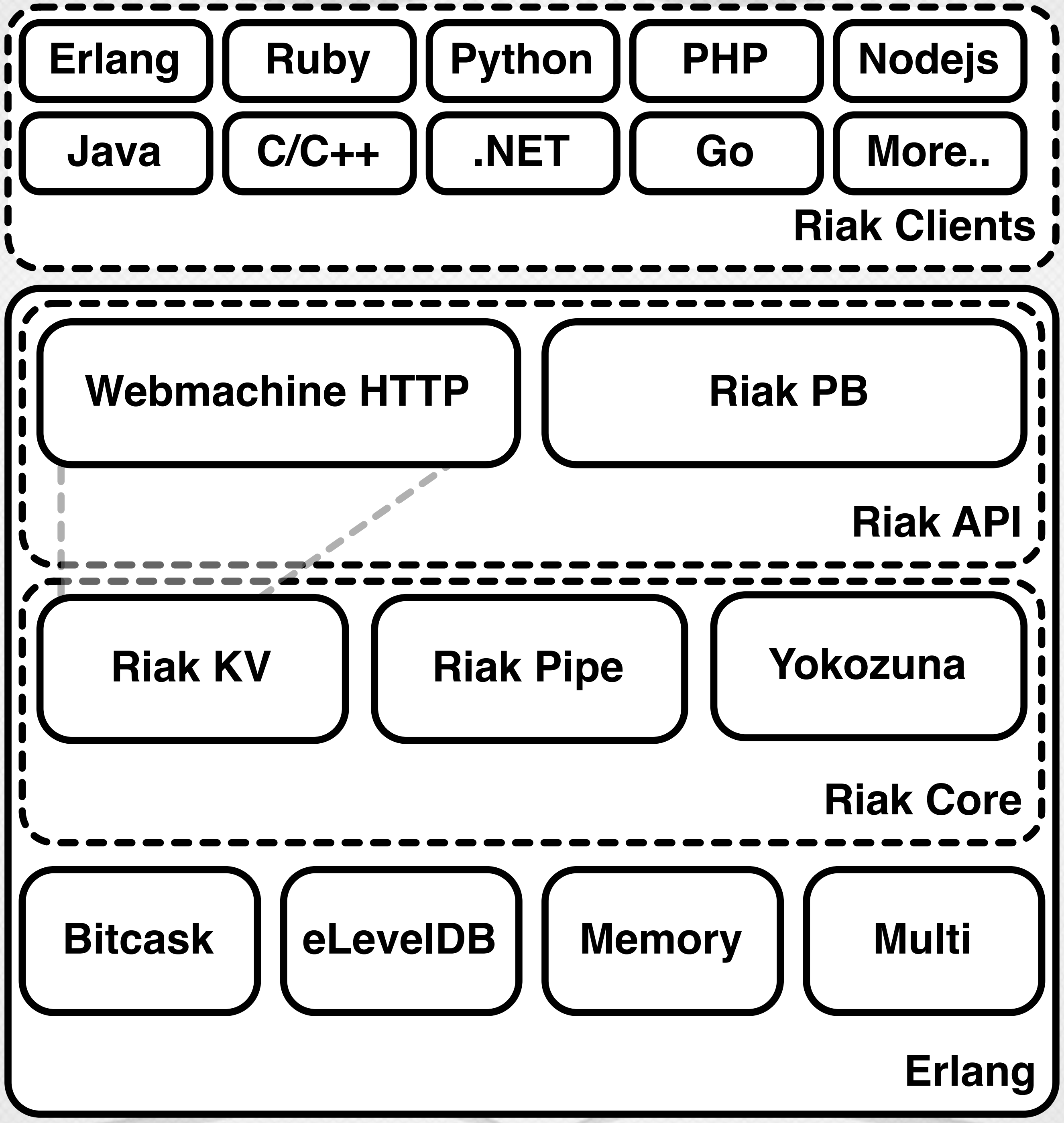
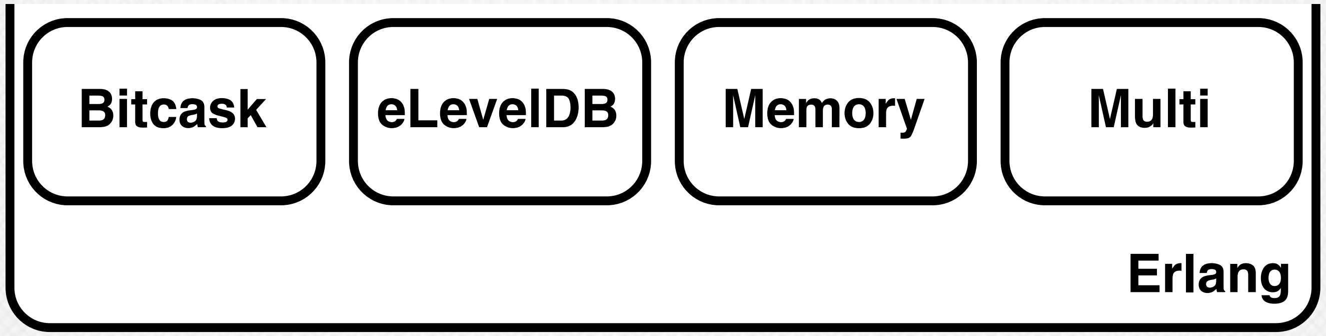


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

Erlang on top



C/C++ on the bottom

image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Linking With C/C++

- Erlang provides the ability to dynamically link C/C++ libraries into the VM
- One way is through the linked-in port driver interface
  - for example the VM supplies network and file system facilities via drivers
- Another way is through Native Implemented Functions (NIFs)

Native Implemented Functions (NIFs)

- Lets C/C++ functions operate as Erlang functions
- Erlang module serves as entry point
- When module loads it dynamically loads its NIF shared library, overlaying its Erlang functions with C/C++ replacements

Example: Eleveldb

- NIF wrapper around Google's LevelDB C++ database
- Erlang interface plugs in underneath Riak KV
- Based on riak_kv_backend storage backend behavior

NIF Features

- Easy to convert arguments and return values between C/C++ and Erlang
- Ref count binaries to avoid data copying where needed
- Portable interface to OS multithreading capabilities (threads, mutexes, cond vars, etc.)

TESTING

Eunit

- Erlang's unit testing facility
- Support for asserting test results, grouping tests, setup and teardown, etc.
- Used heavily in Riak

QuickCheck

- Property-based testing product from Quviq, invented by John Hughes (a co-inventor of Haskell)
- Create a model of the software under test
- QuickCheck runs randomly-generated tests against it
- When it finds a failure, QuickCheck automatically shrinks the testcase to a minimum for easier debugging
- Used heavily in Riak, especially to test various protocols and interactions

MISCELLANEOUS

Memory

- Process message queues have no limits, can cause out-of-memory conditions if a process can't keep up
- By design, VM dies if it runs out of memory
- Apps like Riak run Erlang memory monitors that help notify about potential out-of-memory conditions

Interactive Erlang Shell

- Hard to imagine working without it
- Huge help during development and debug

Hot Code Loading

- It really works
- Use it all the time during development
- We've also used it to load repaired code into live production systems to help customers

VM Knowledge

- Running high-scale high-load systems like Riak requires knowledge of Erlang VM internals
- No different than working with the JVM or other language runtimes

For More Riak Info

- "A Little Riak Book" by Basho's Eric Redmond
  http://littleriakbook.com
- Mathias Meyer's "Riak Handbook"
  http://riakhandbook.com
- Eric Redmond's "Seven Databases in Seven Weeks"
  http://pragprog.com/book/rwdata/seven-databases-in-seven-weeks

For More Riak Info

- Basho documentation
  http://docs.basho.com

- Basho blog
  http://basho.com/blog/

- Basho's github repositories
  https://github.com/basho
  https://github.com/basho-labs

THANKS

http://basho.com
@stevevinoski