basho

Implementing Riak in Erlang: Benefits and Challenges

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

Erlang

Erlang

- Started in the mid-80's, Ericsson Computer Science Laboratories (CSL)
- Joe Armstrong began investigating languages for programming next-generation telecom equipment
- Erlang initially implemented in Prolog, with influence and ideas from ML, Ada, Smalltalk, other languages

Erlang

- Open sourced in 1998
- Available from http://erlang.org
- Latest release: R15B03 (Nov 2012)

Ericsson CSL Telecom Switch Requirements

Ericsson CSL Telecom Switch Requirements

- Large number of concurrent activities

Ericsson CSL Telecom Switch Requirements

- Large number of concurrent activities
- Large software systems distributed across multiple computers

Ericsson CSL Telecom Switch Requirements

- Large number of concurrent activities
- Large software systems distributed across multiple computers
- Continuous operation for years

Ericsson CSL Telecom Switch Requirements

- Large number of concurrent activities
- Large software systems distributed across multiple computers
- Continuous operation for years
- Live updates and maintenance

Ericsson CSL Telecom Switch Requirements

- Large number of concurrent activities
- Large software systems distributed across multiple computers
- Continuous operation for years
- Live updates and maintenance
- Tolerance for both hardware and software faults

Today's Data/Web/ Cloud/Service Apps

- Large number of concurrent activities
- Large software systems distributed across multiple computers
- Continuous operation for years
- Live updates and maintenance
- Tolerance for both hardware and software faults

Concurrency

Erlang Processes

- Lightweight, much lighter than OS threads
- Hundreds of thousands or even millions per Erlang VM instance

Concurrency For Reliability

Concurrency For Reliability

- Isolation: Erlang processes communicate only via message passing

Concurrency For Reliability

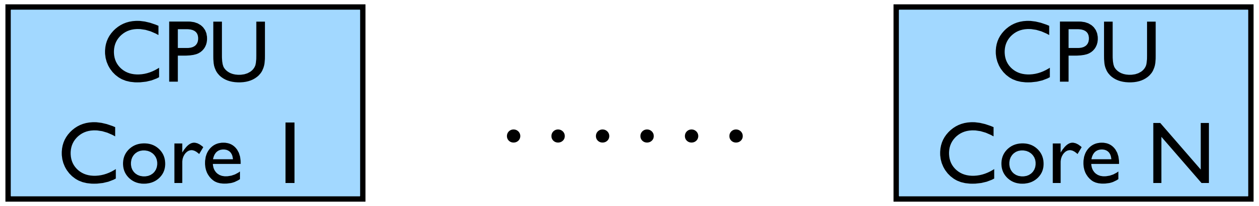
- Isolation: Erlang processes communicate only via message passing
- Distribution: Erlang process model works across nodes

Concurrency For Reliability

- Isolation: Erlang processes communicate only via message passing
- Distribution: Erlang process model works across nodes
- Monitoring/supervision: allow an Erlang process to take action when another fails

Erlang Process Architecture

Erlang Process Architecture

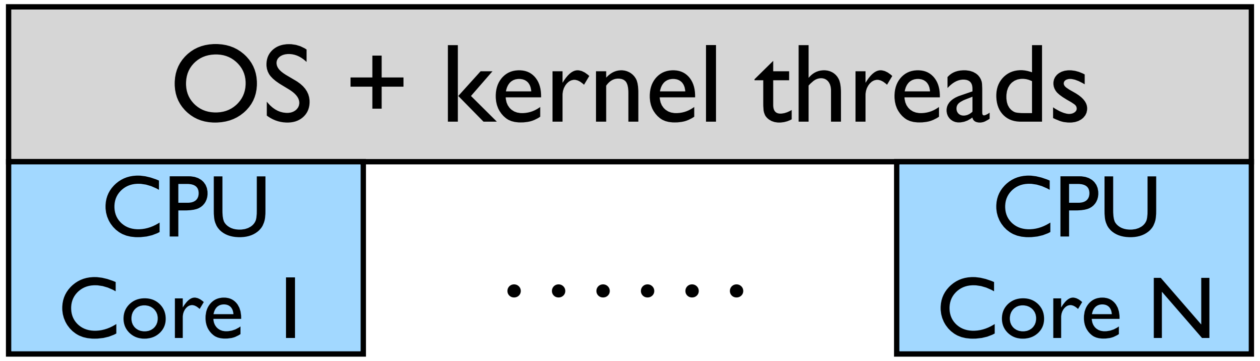


CPU Core 1

. . . . . .

CPU Core N

Erlang Process Architecture



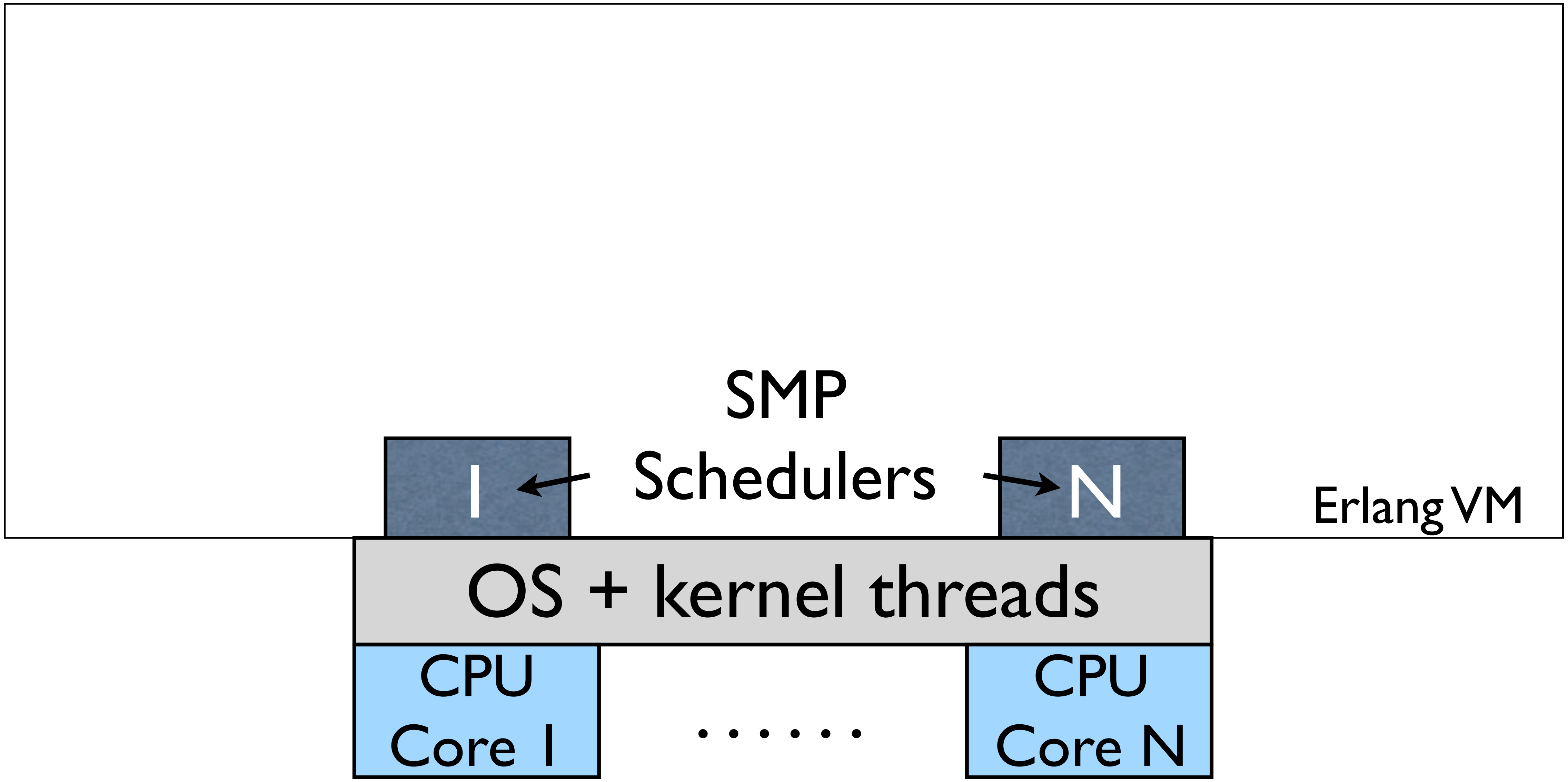
OS + kernel threads

CPU Core 1

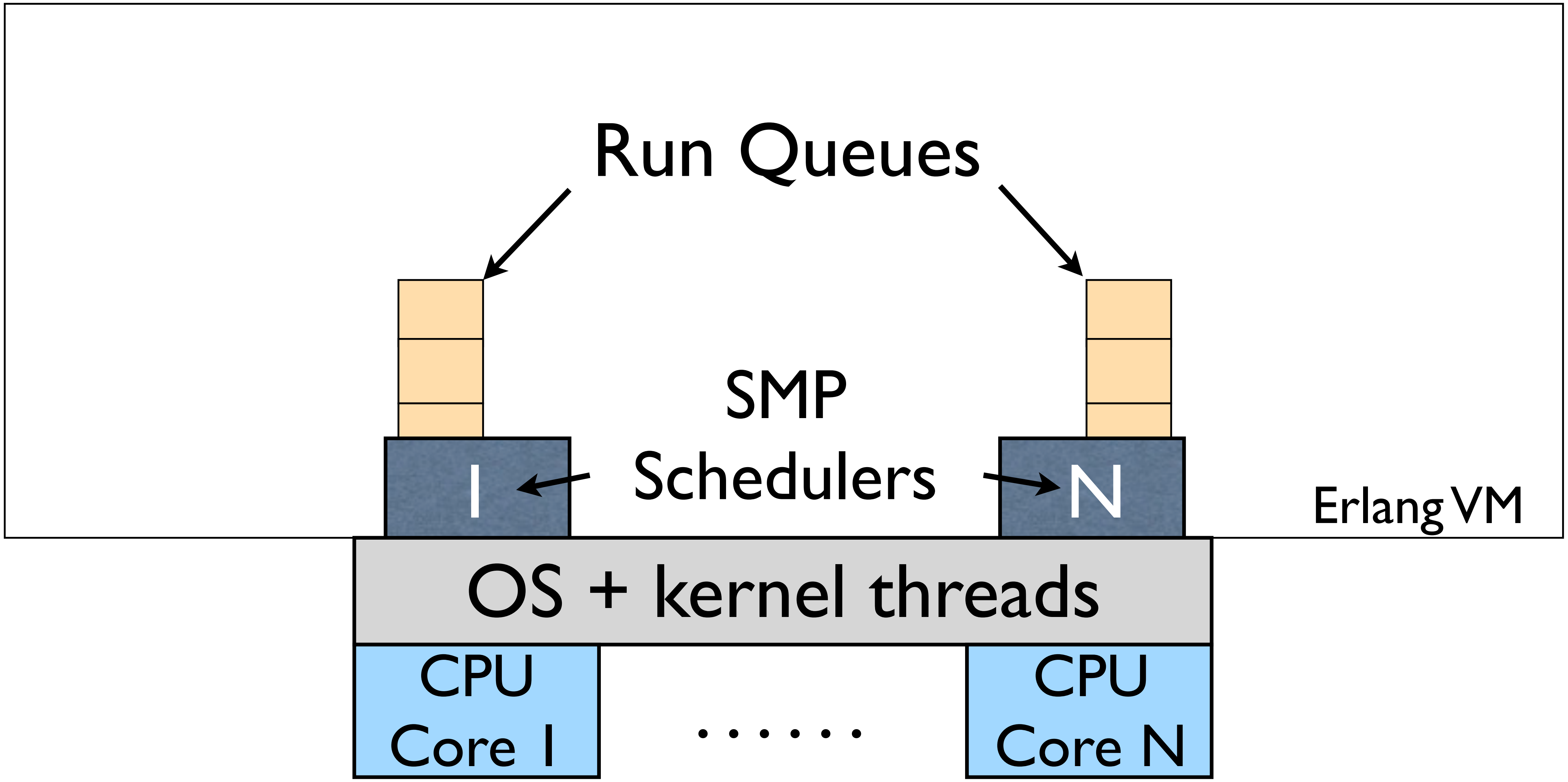
. . . . . .

CPU Core N

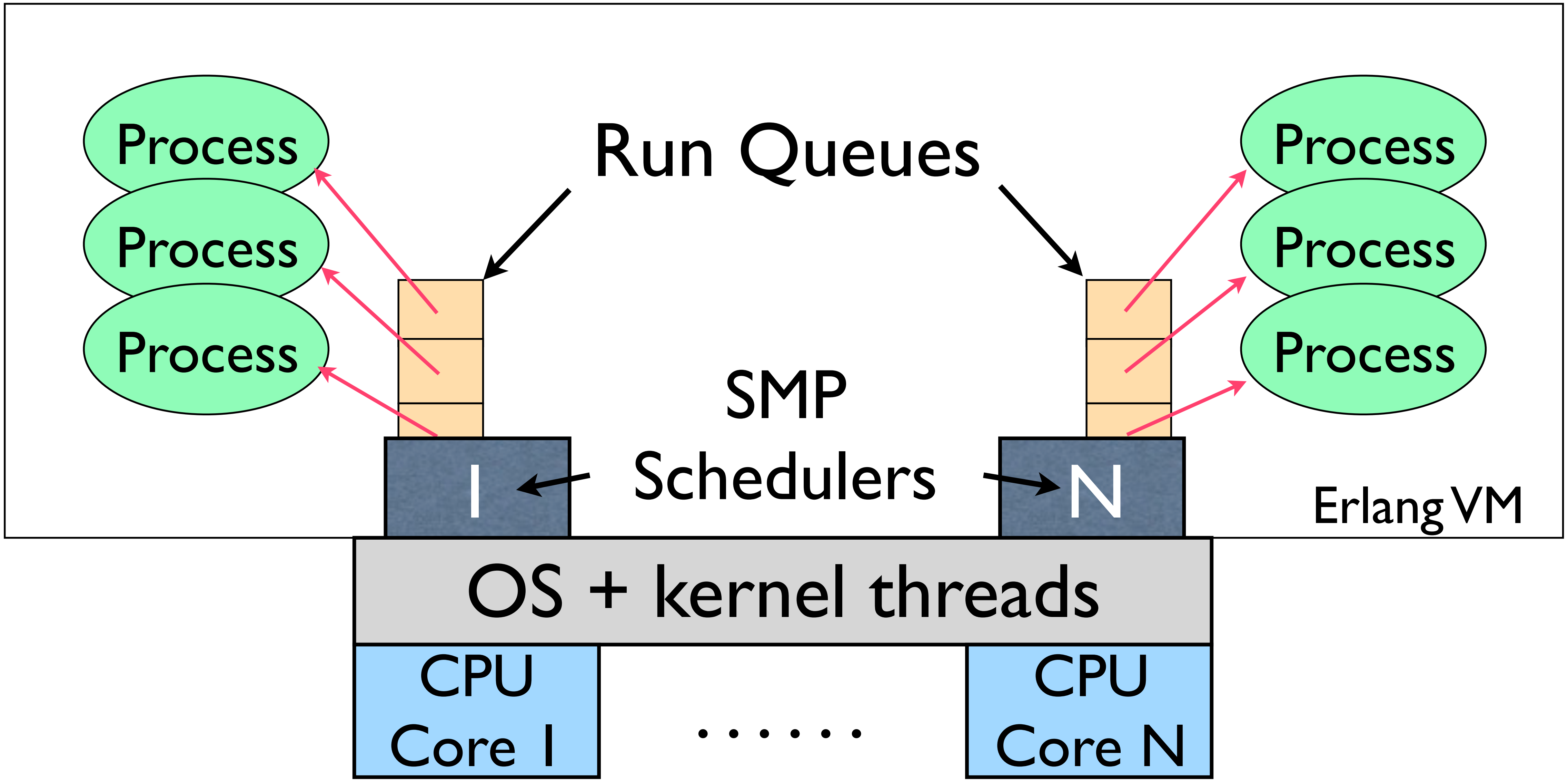
Erlang Process Architecture



Erlang Process Architecture



Erlang Process Architecture



A Small Language

- Erlang has just a few elements: numbers, atoms, tuples, lists, records, binaries, functions, modules
- Variables are single assignment, no globals
- Flow control via pattern matching, case, if, try-catch, recursion, messages

Easy To Learn

- Language size means developers become proficient quickly
- Code is typically small, easy to read, easy to understand
- Erlang's Open Telecom Platform (OTP) frameworks solve recurring problems across multiple domains

What is Riak?

What is Riak?

What is Riak?

- A distributed

What is Riak?

- A distributed
- highly available

What is Riak?

- A distributed
- highly available
- highly scalable

What is Riak?

- A distributed
- highly available
- highly scalable
- open source

What is Riak?

- A distributed
- highly available
- highly scalable
- open source
- key-value database

What is Riak?

- A distributed
- highly available
- highly scalable
- open source
- key-value database
- written mostly in Erlang.

What is Riak?

- Modeled after Amazon Dynamo
  - see Andy Gross's "Dynamo, Five Years Later" for more details
    https://speakerdeck.com/argv0/dynamo-five-years-later
- Also provides MapReduce, secondary indexes, and full-text search
- Built for operational ease

Riak Architecture

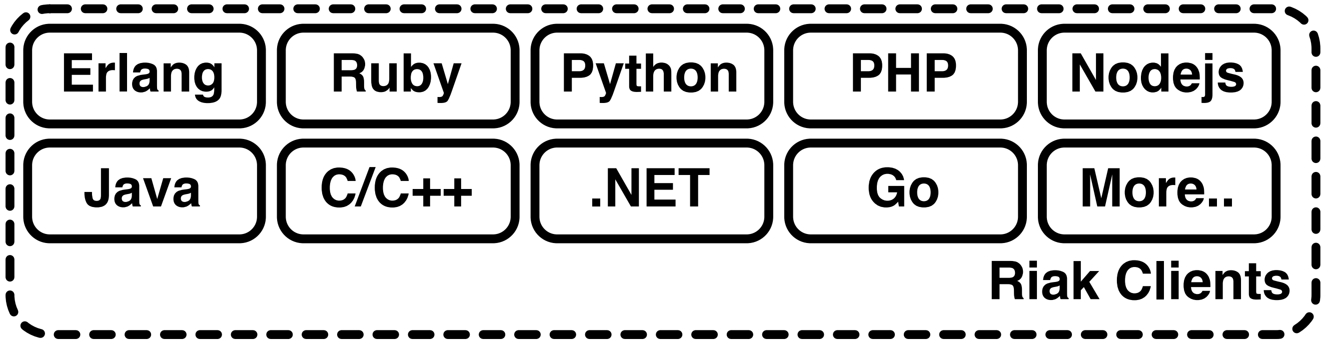


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

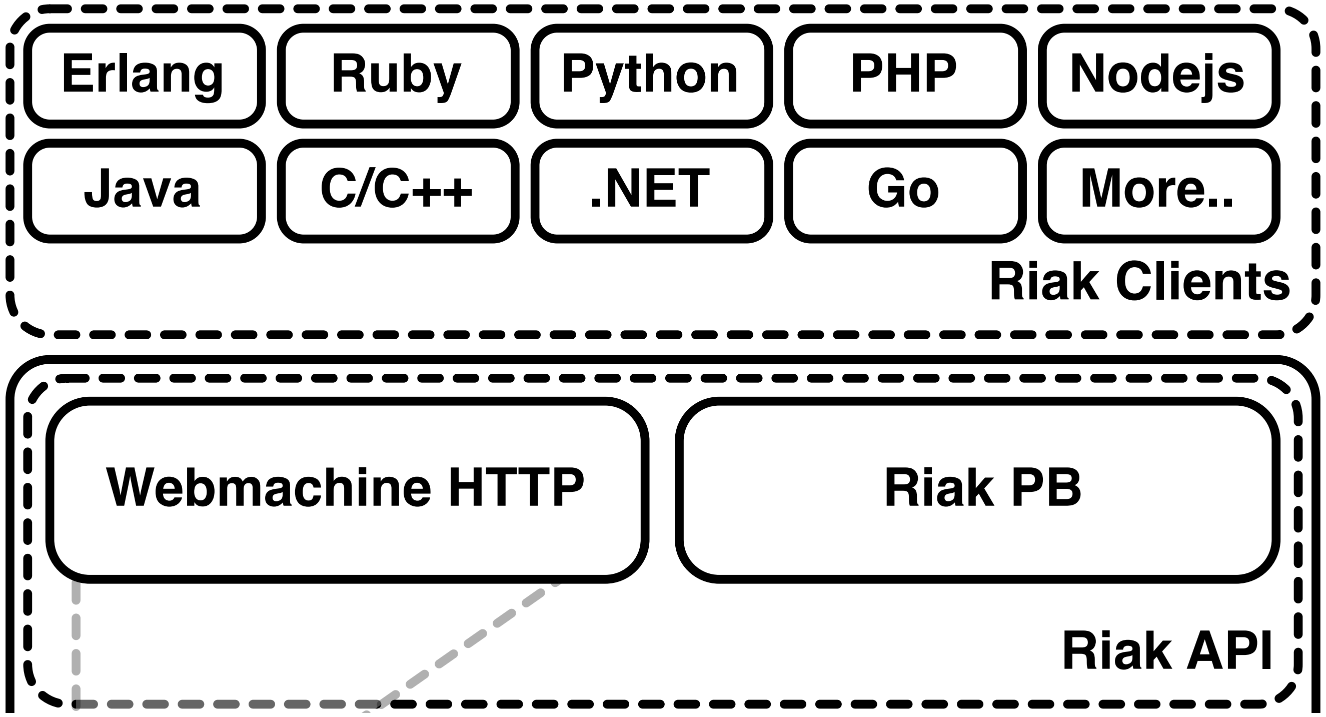


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

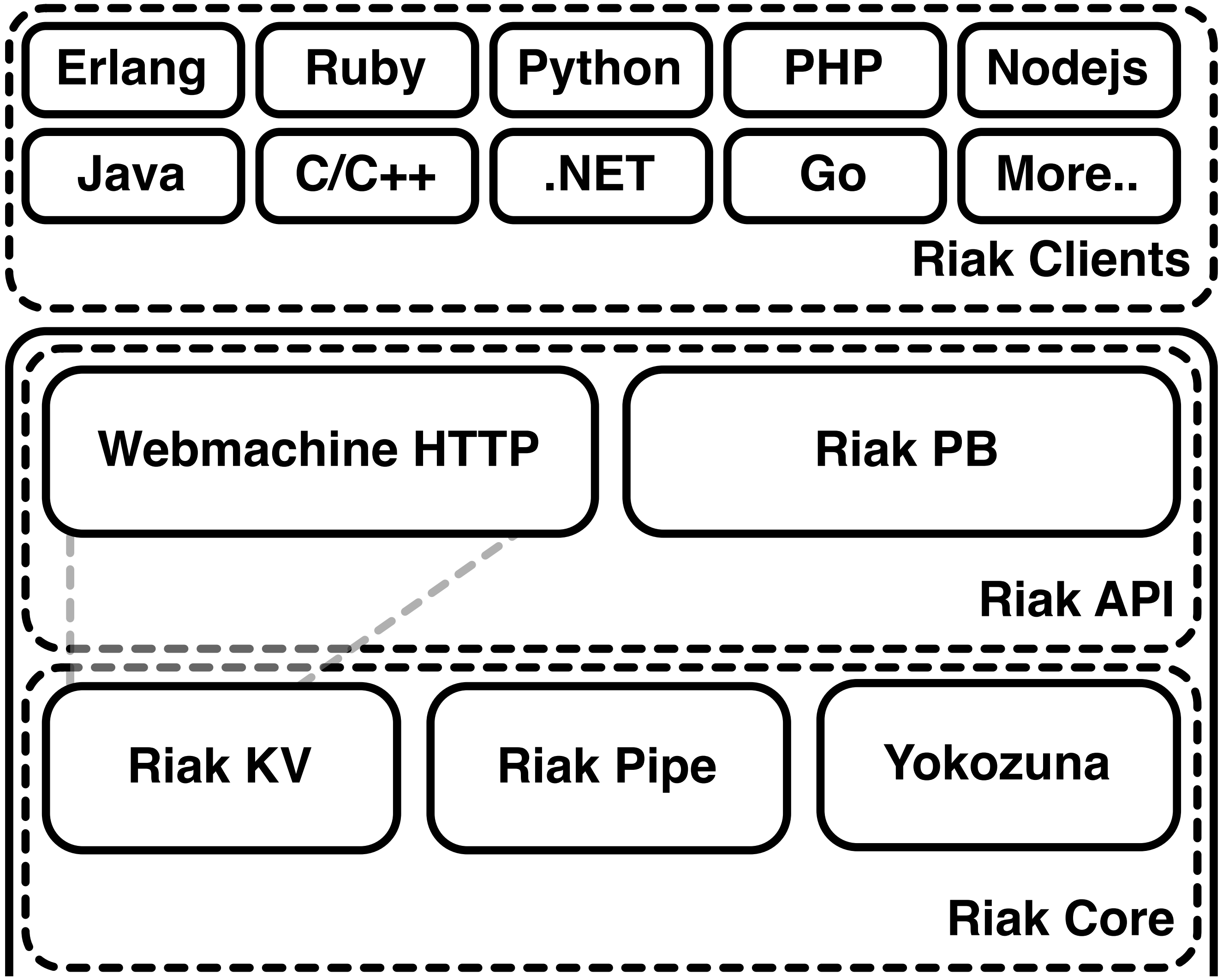


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

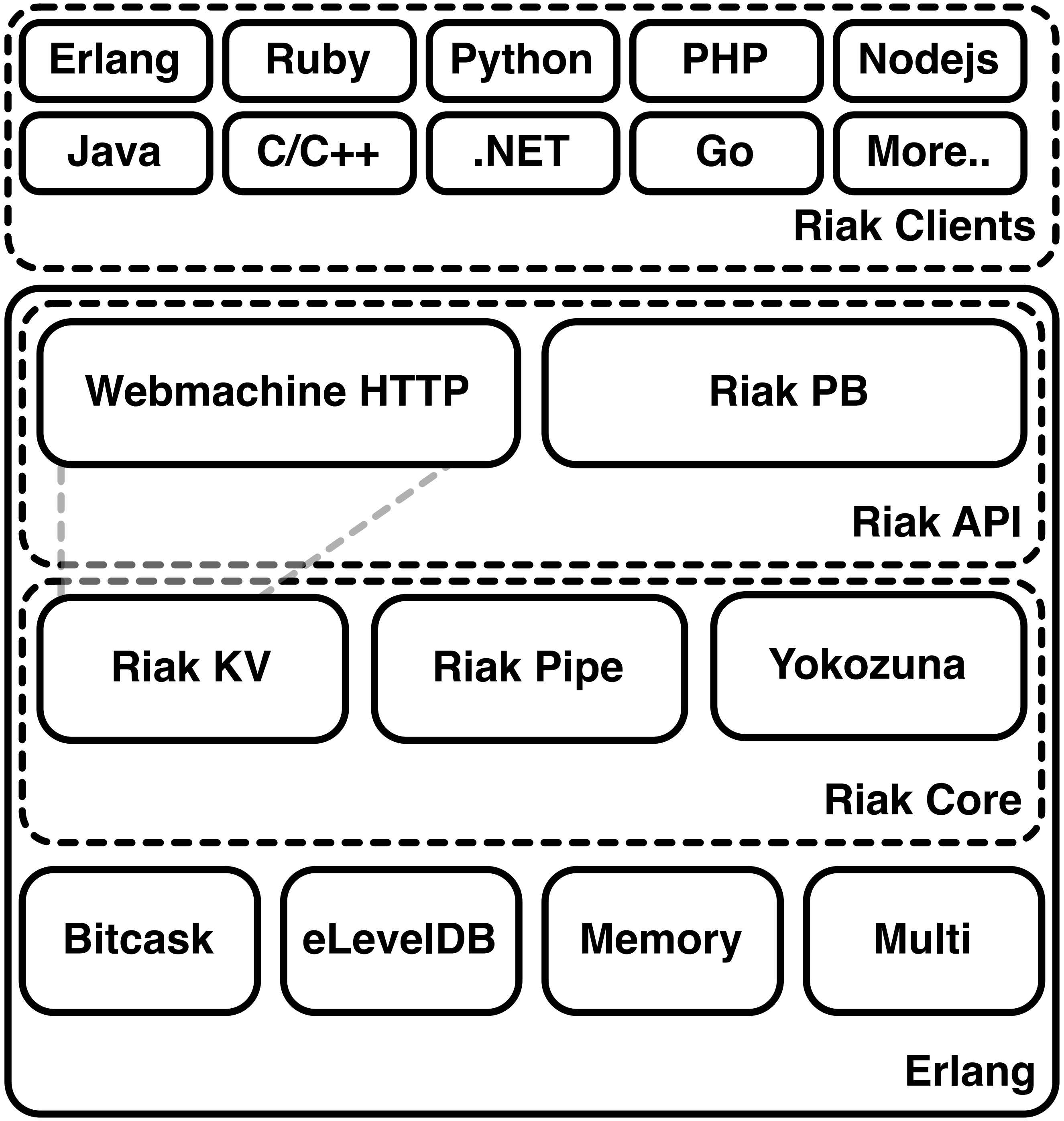
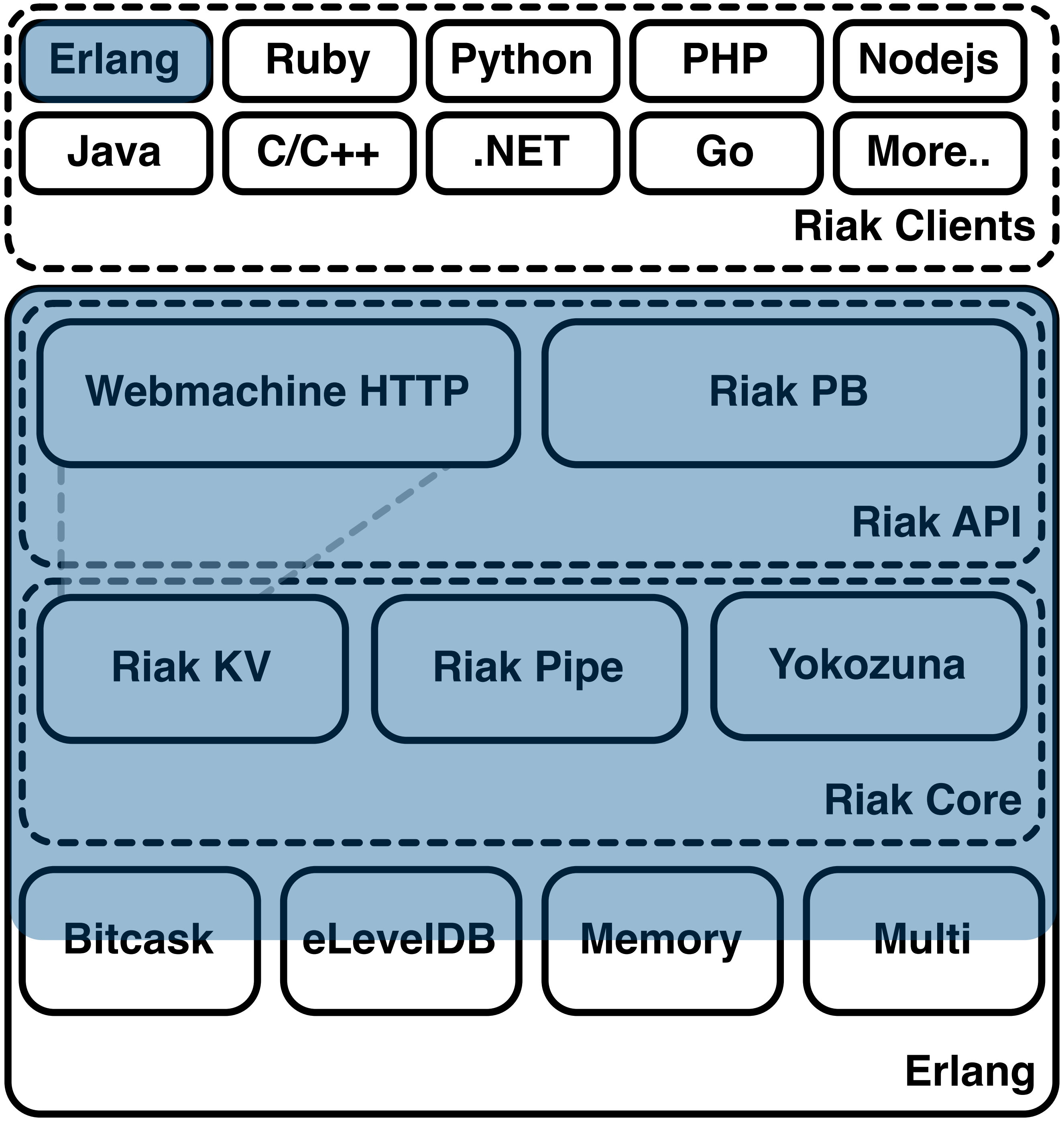


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

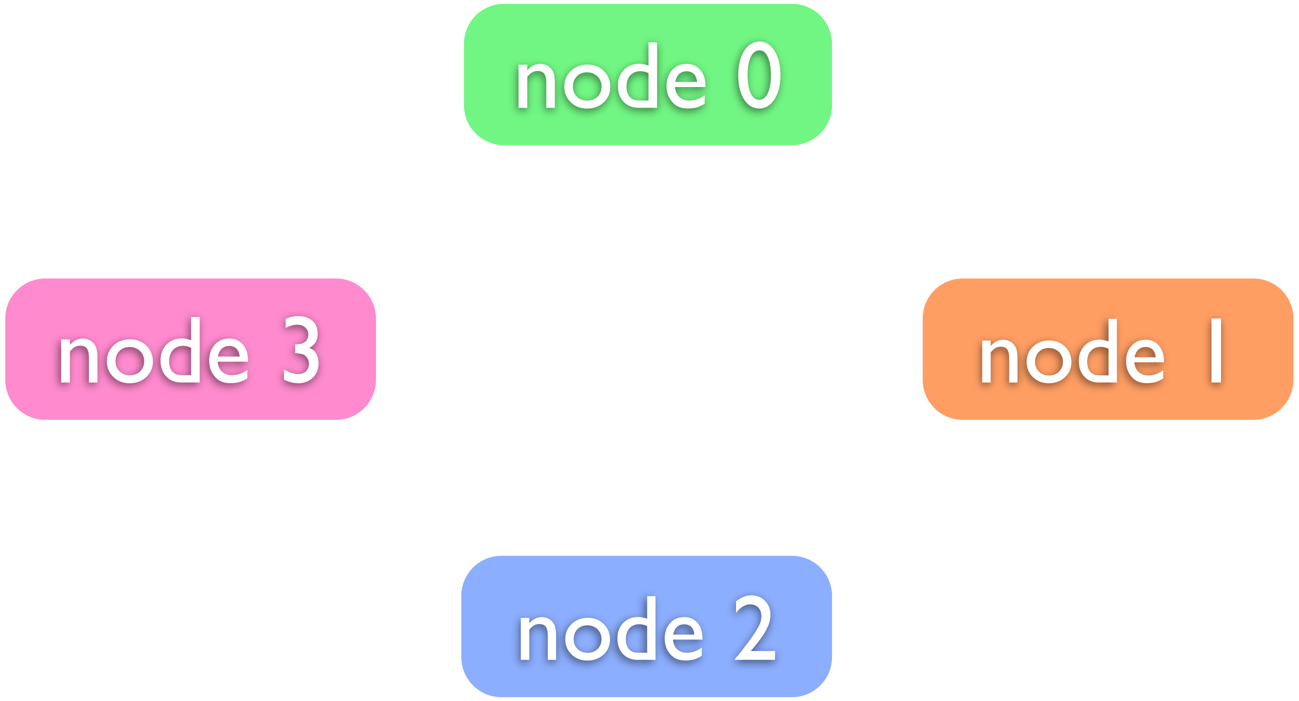
Riak Architecture



- Erlang parts

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

Riak Cluster



Distributing Data

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

node 0

node 1

node 2

node 3

Consistent Hashing

node 0

node 1

node 2

node 3

Consistent Hashing

- Riak uses SHA-1 as a hash function

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

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)

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 physical node in the cluster hosts multiple vnodes

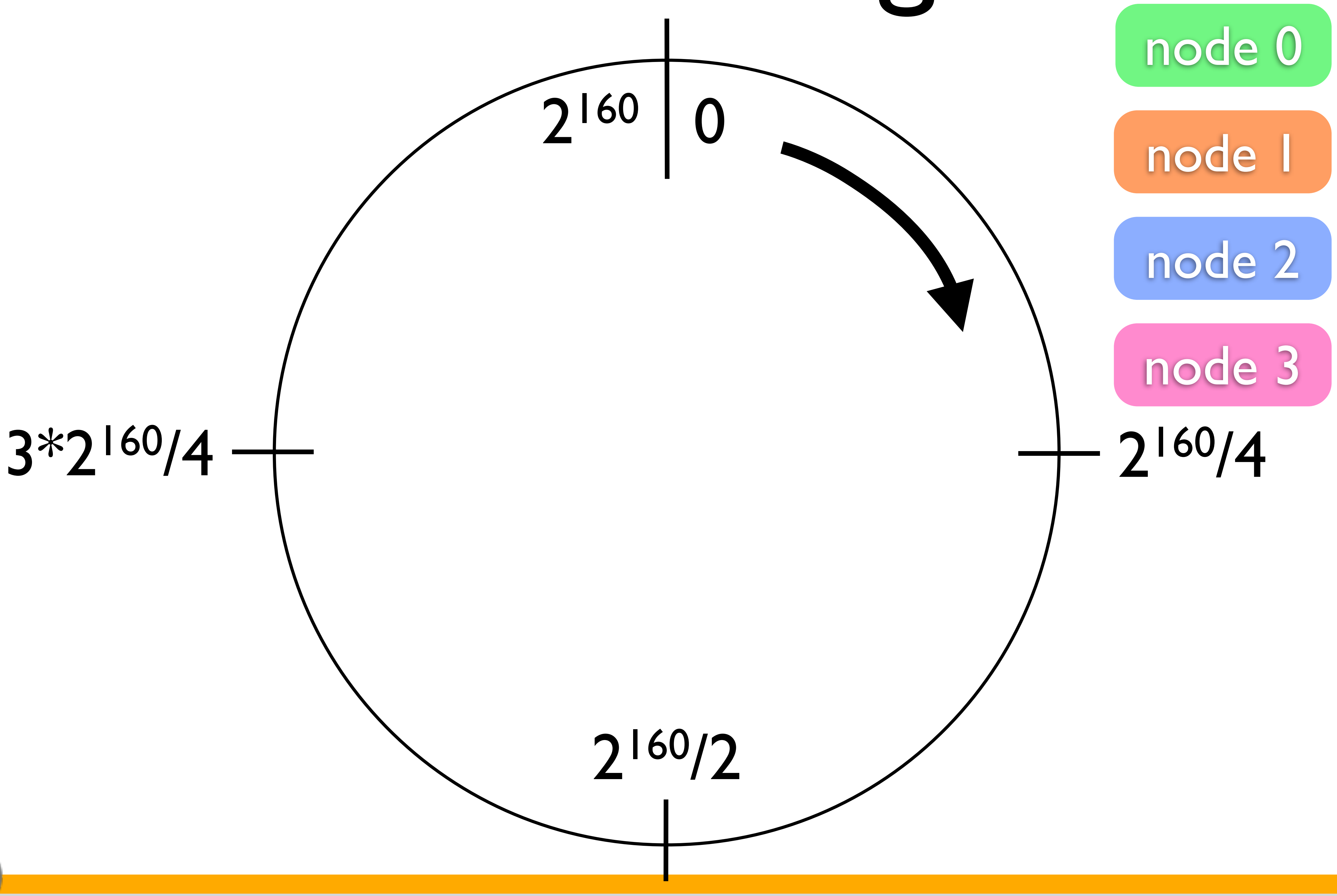
node 0

node 1

node 2

node 3

Hash Ring



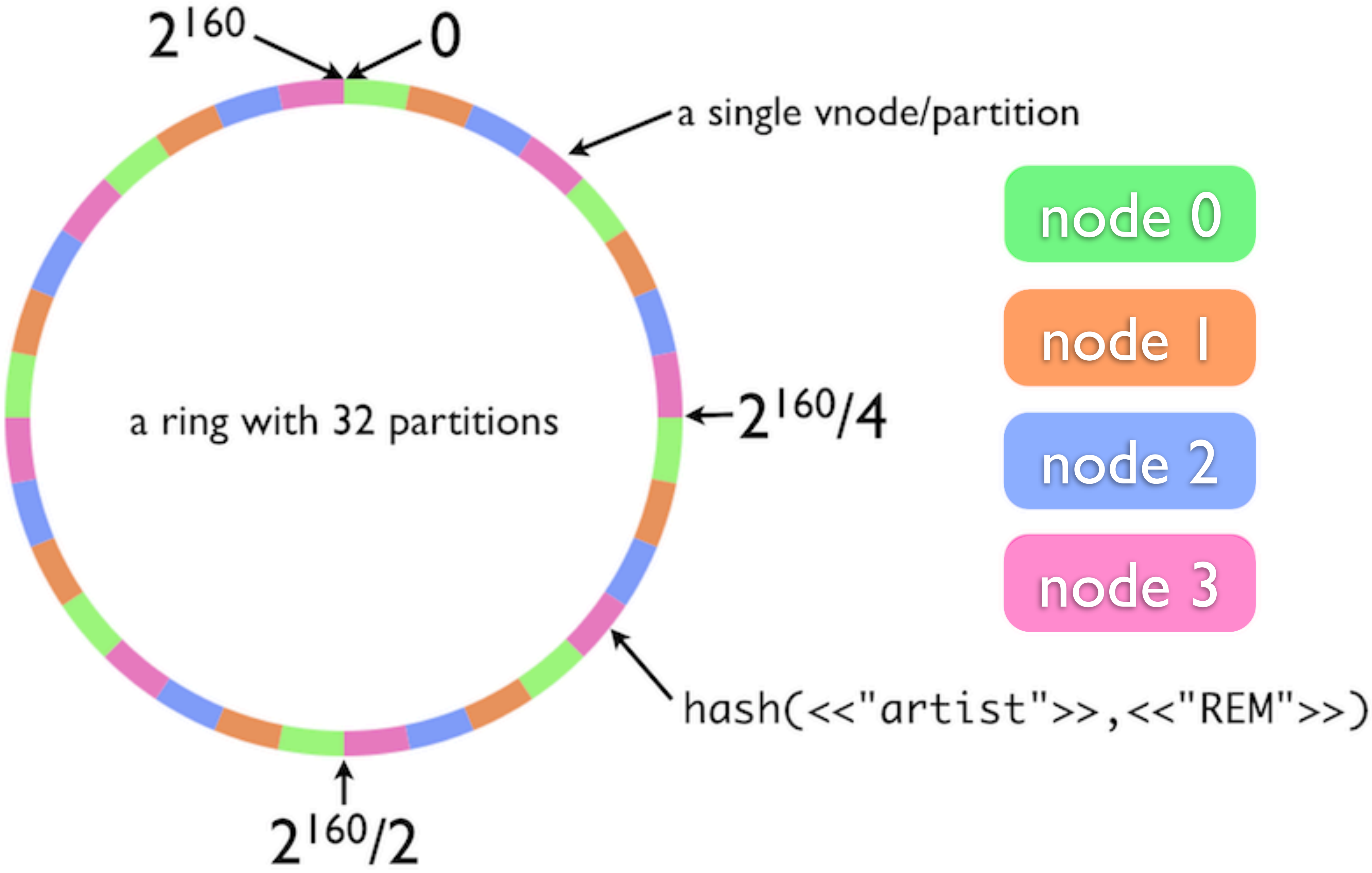
🟩 node 0

🟧 node 1

🟦 node 2

🟪 node 3

Hash Ring



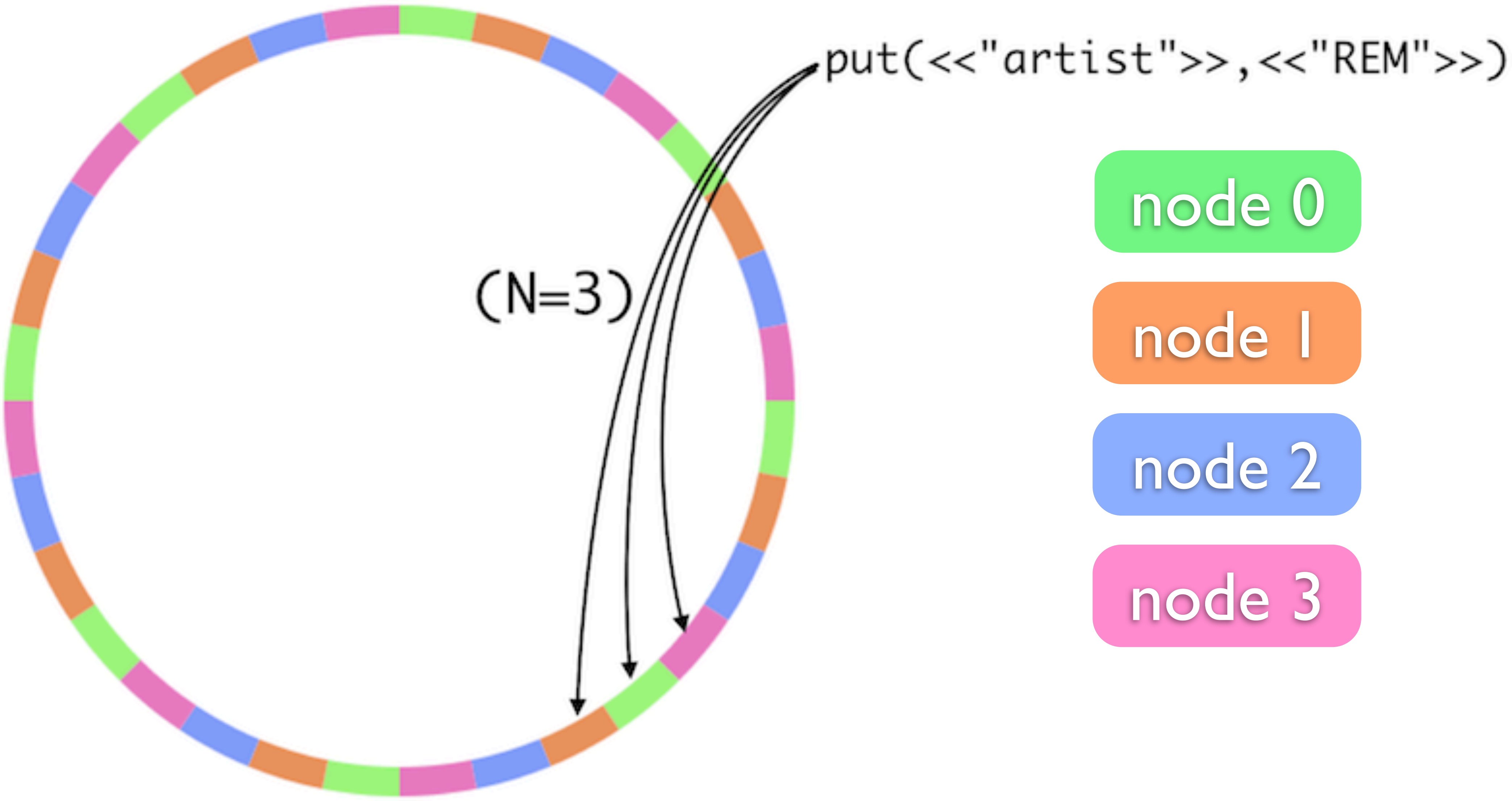
node 0

node 1

node 2

node 3

N/R/W Values



put(<<"artist">>,<<"REM">>)

(N=3)

node 0

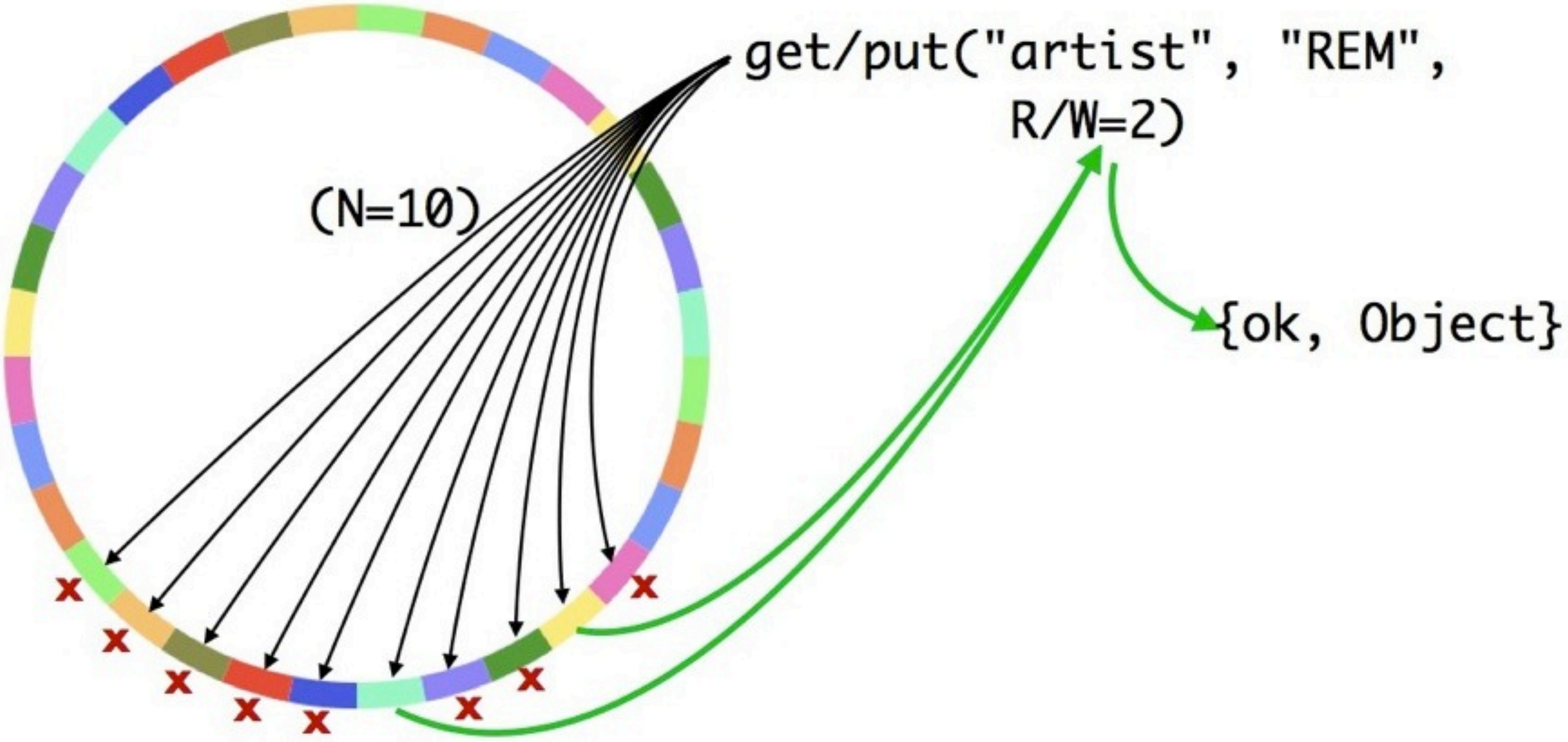
node 1

node 2

node 3

for details see http://docs.basho.com/riak/1.2.1/tutorials/fast-track/Tunable-CAP-Controls-in-Riak/

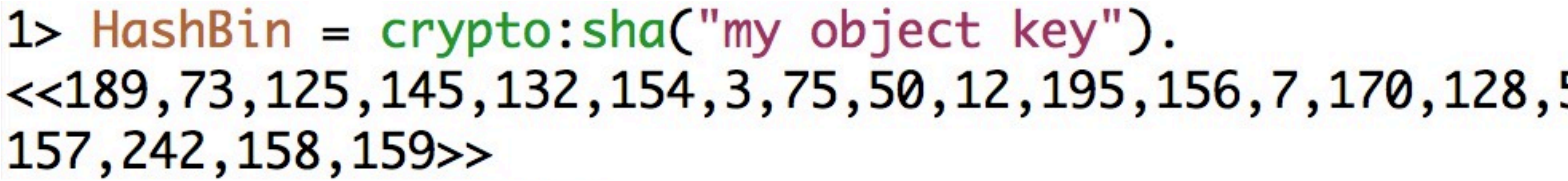
N/R/W Values



Implementing Consistent Hashing

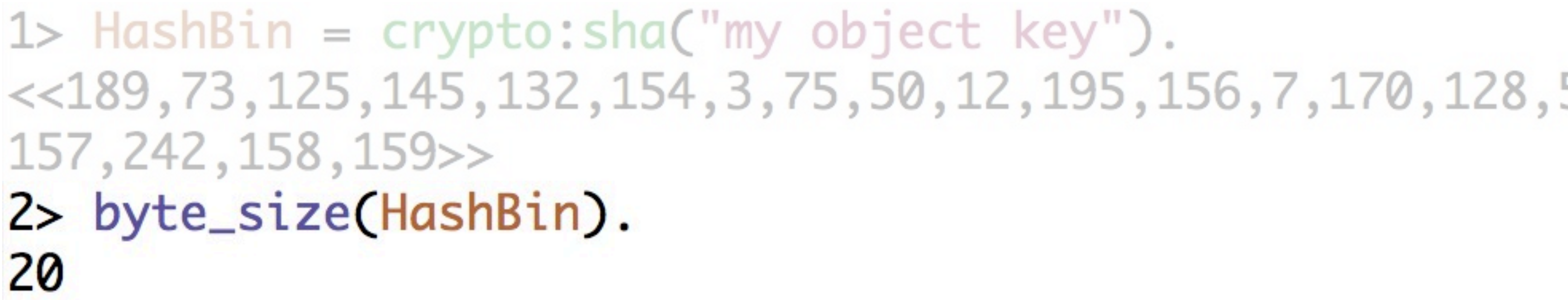
- Erlang's crypto module integration with OpenSSL provides the SHA-1 function
- Hash values are 160 bits
- But Erlang's integers are infinite precision
- And Erlang binaries store these large values efficiently

Implementing Consistent Hashing



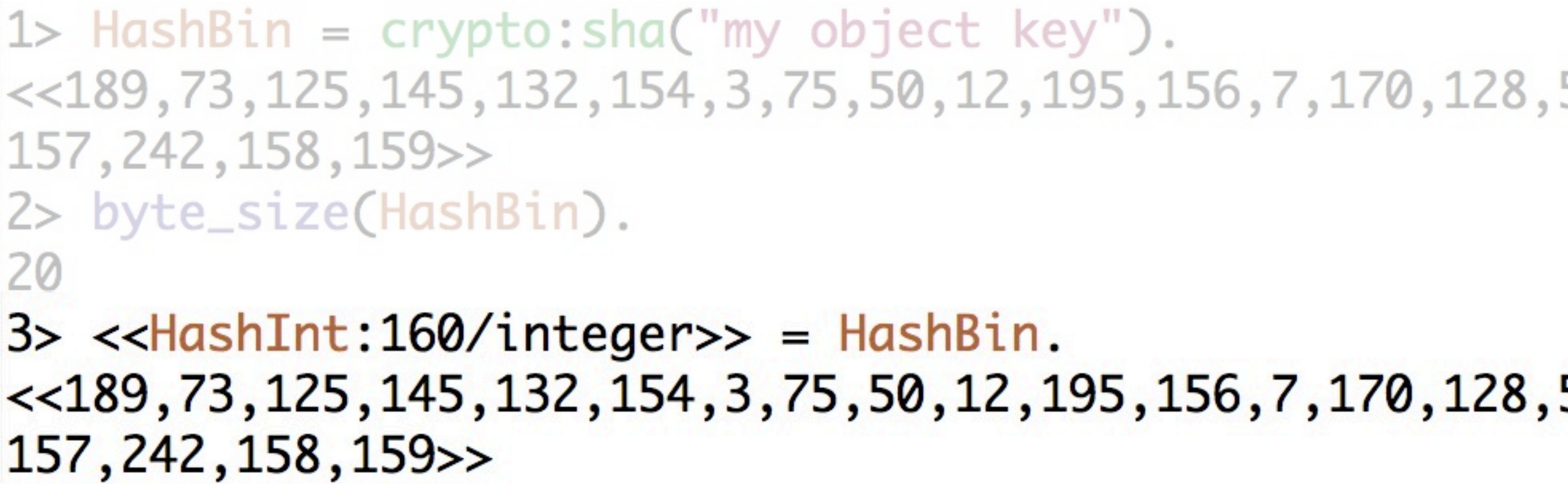
```
1> HashBin = crypto:sha("my object key").
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
```

Implementing Consistent Hashing



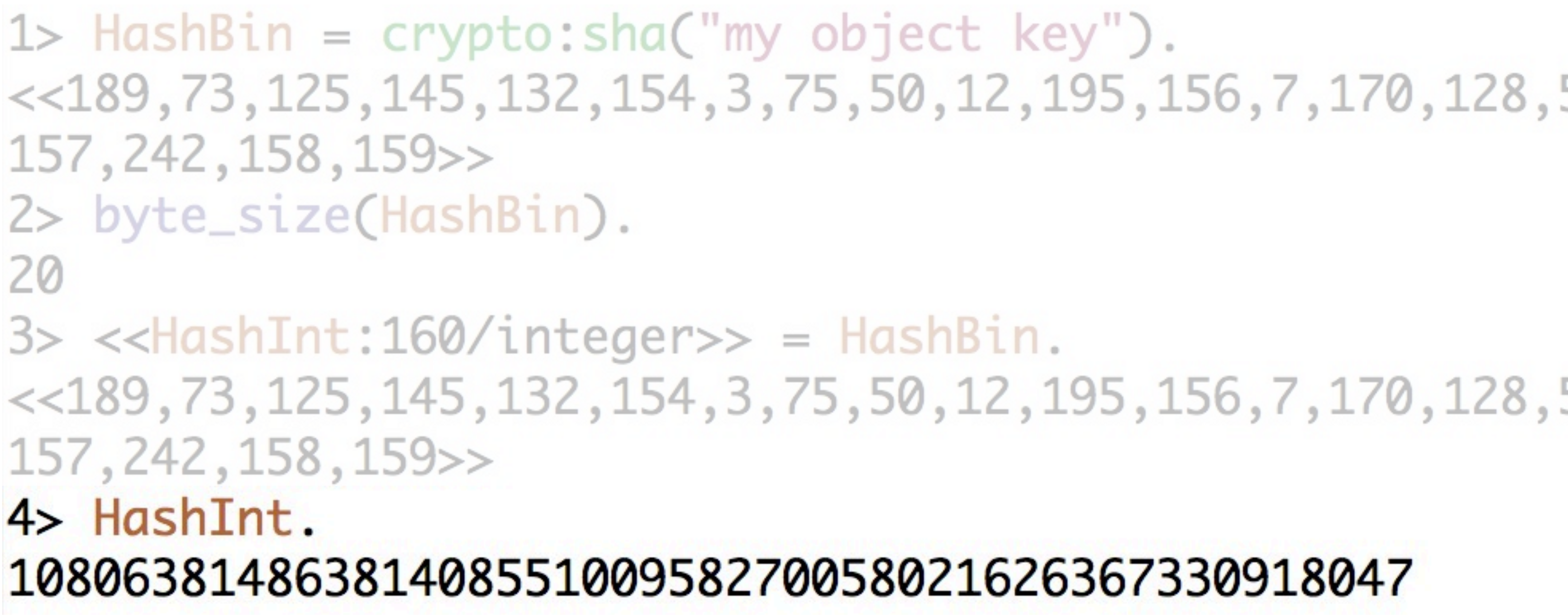
```
1> HashBin = crypto:sha("my object key").
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
2> byte_size(HashBin).
20
```

Implementing Consistent Hashing



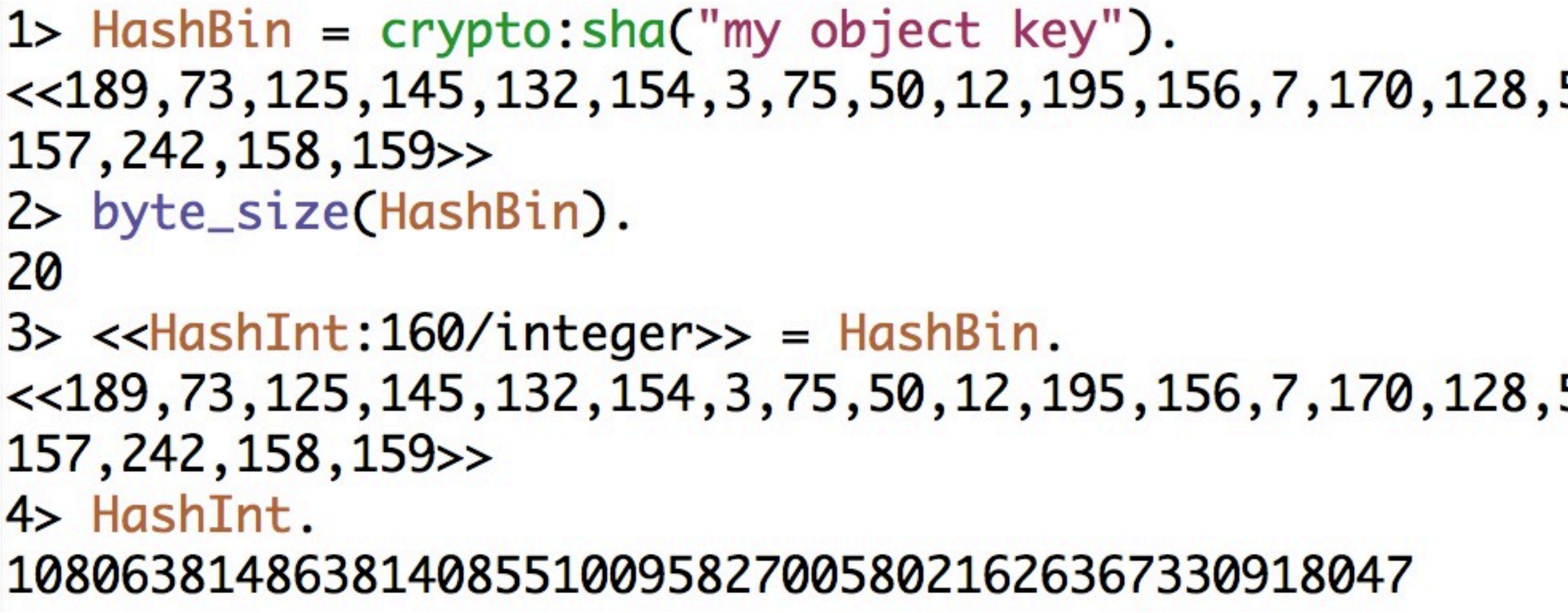
```
1> HashBin = crypto:sha("my object key").
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
2> byte_size(HashBin).
20
3> <<HashInt:160/integer>> = HashBin.
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
```

Implementing Consistent Hashing



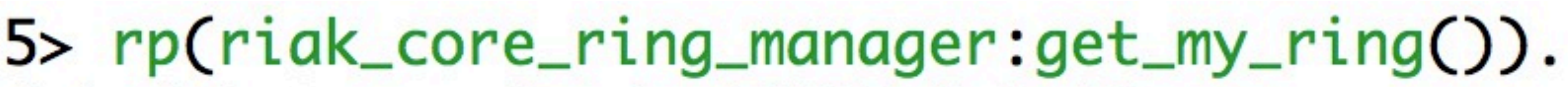
```
1> HashBin = crypto:sha("my object key").
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
2> byte_size(HashBin).
20
3> <<HashInt:160/integer>> = HashBin.
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
4> HashInt.
1080638148638140855100958270058021626367330918047
```

Implementing Consistent Hashing



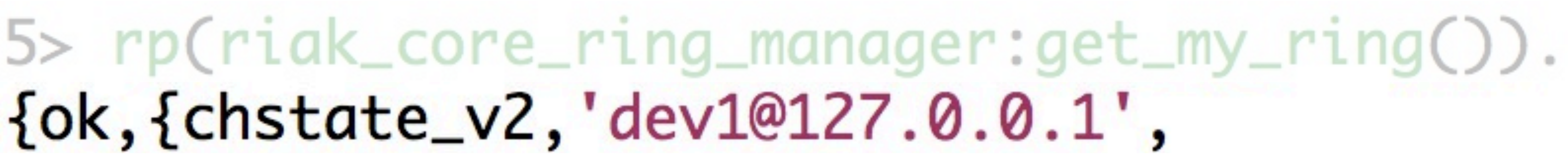
```
1> HashBin = crypto:sha("my object key").
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
2> byte_size(HashBin).
20
3> <<HashInt:160/integer>> = HashBin.
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,5
157,242,158,159>>
4> HashInt.
1080638148638140855100958270058021626367330918047
```

Riak's Ring



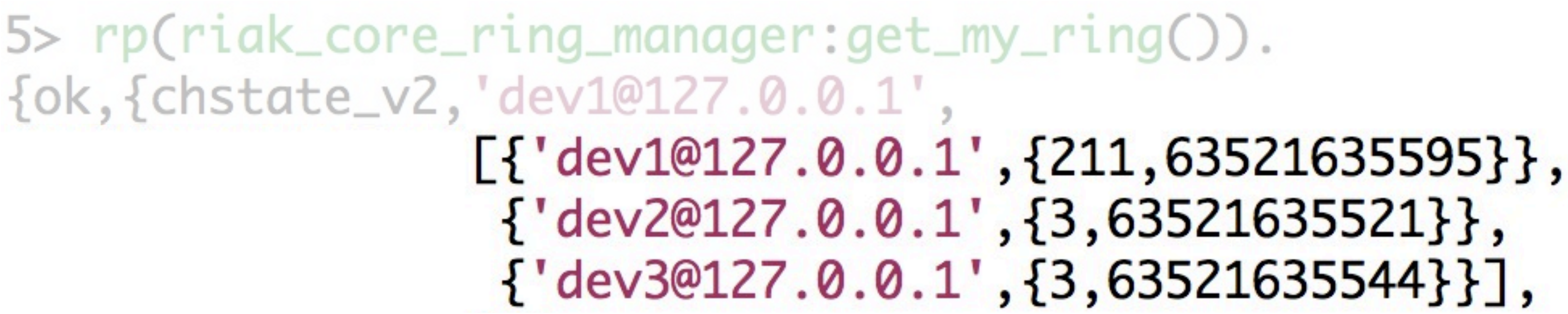
```
5> rp(riak_core_ring_manager:get_my_ring()).
```

Riak's Ring



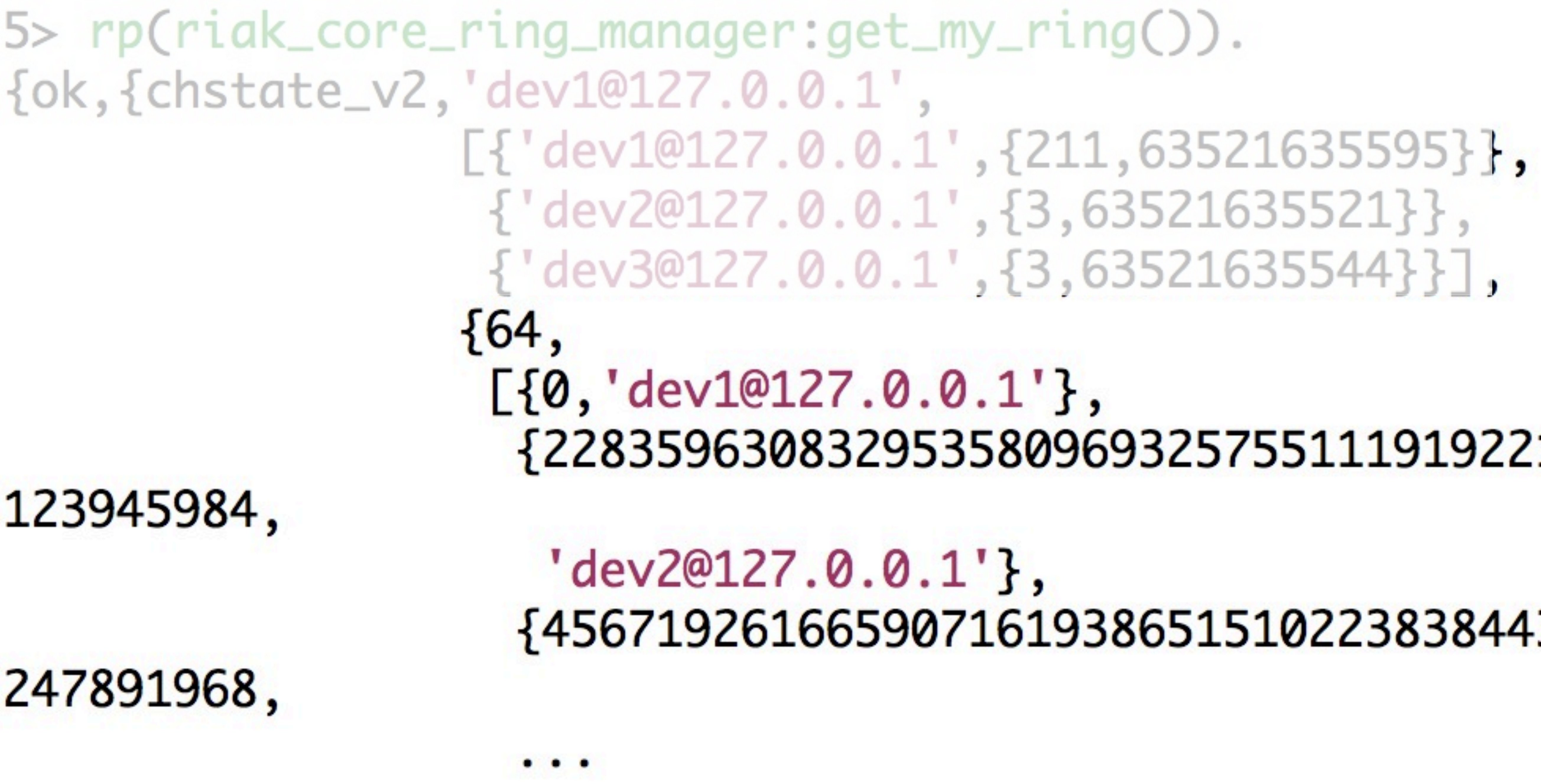
```
5> rp(riak_core_ring_manager:get_my_ring()).
{ok,{chstate_v2,'dev1@127.0.0.1',
```

Riak's Ring



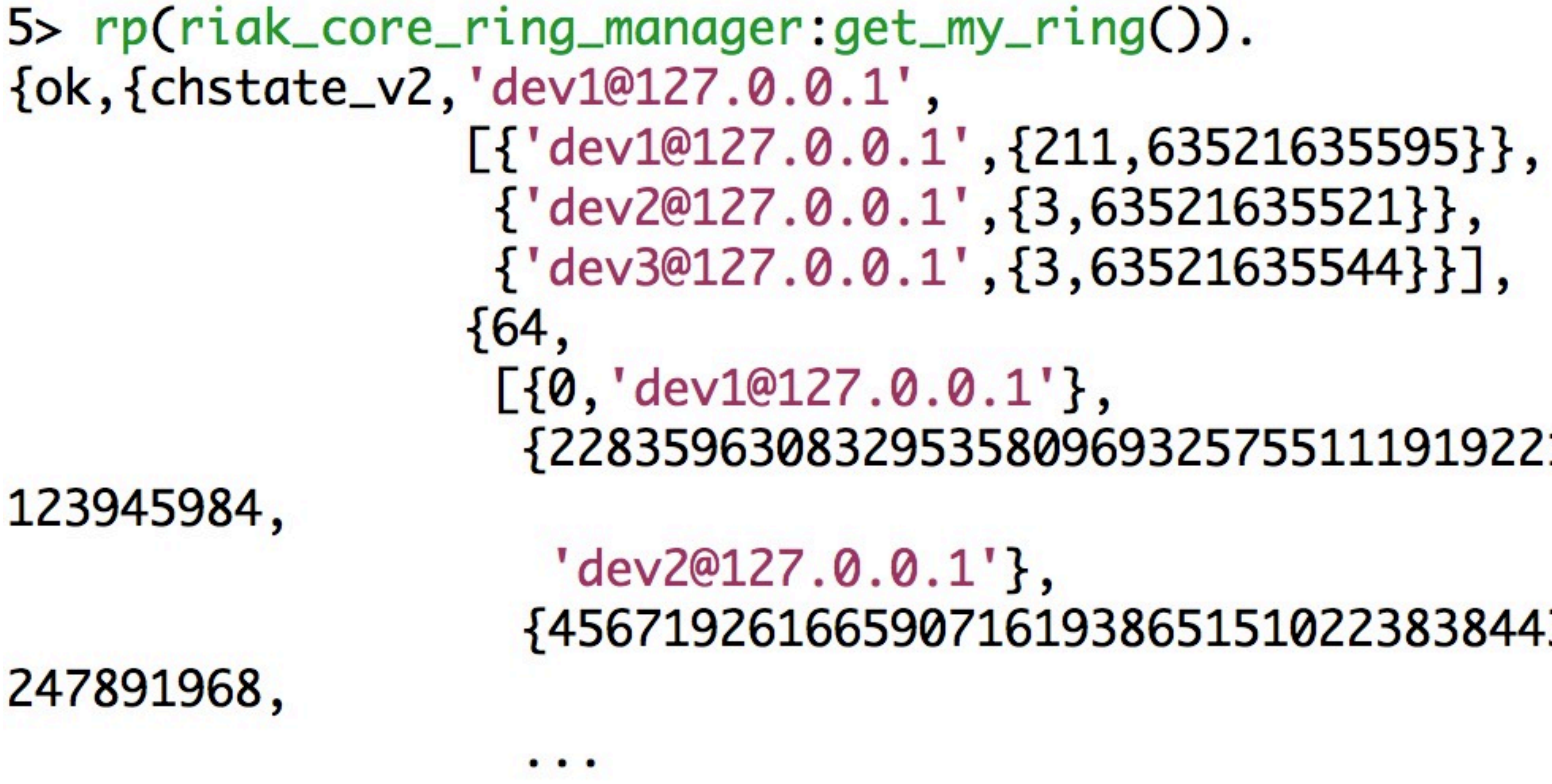
```
5> rp(riak_core_ring_manager:get_my_ring()).
{ok,{chstate_v2,'dev1@127.0.0.1',
                [{'dev1@127.0.0.1',{211,63521635595}},
                 {'dev2@127.0.0.1',{3,63521635521}},
                 {'dev3@127.0.0.1',{3,63521635544}}],
```

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'},
                  {22835963083295358096932575511191922
123945984,
                   'dev2@127.0.0.1'},
                  {45671926166590716193865151022383844
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'},
                  {22835963083295358096932575511191922
123945984,

                   'dev2@127.0.0.1'},
                  {45671926166590716193865151022383844
247891968,

                  ...
```

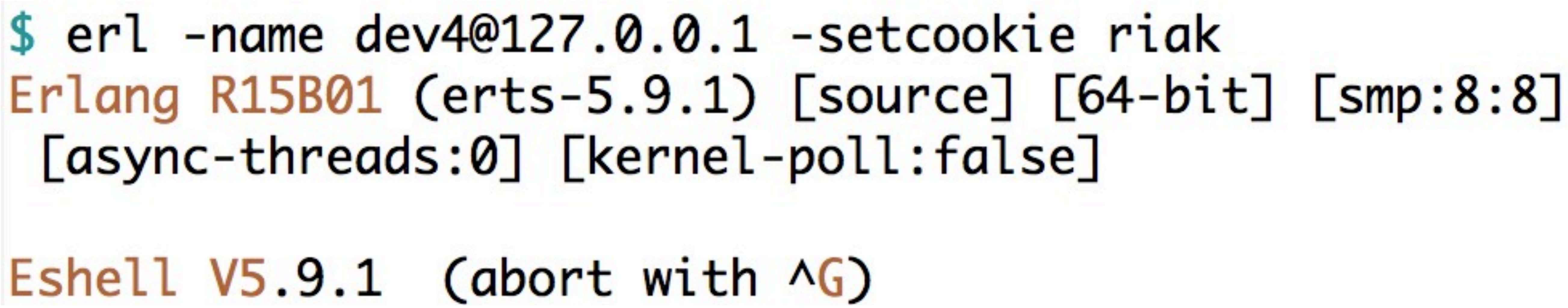
Ring State

- 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
  - required for reliability
- By default Erlang nodes form a mesh, every node knows about every other node
- Riak uses this for intra-cluster communication

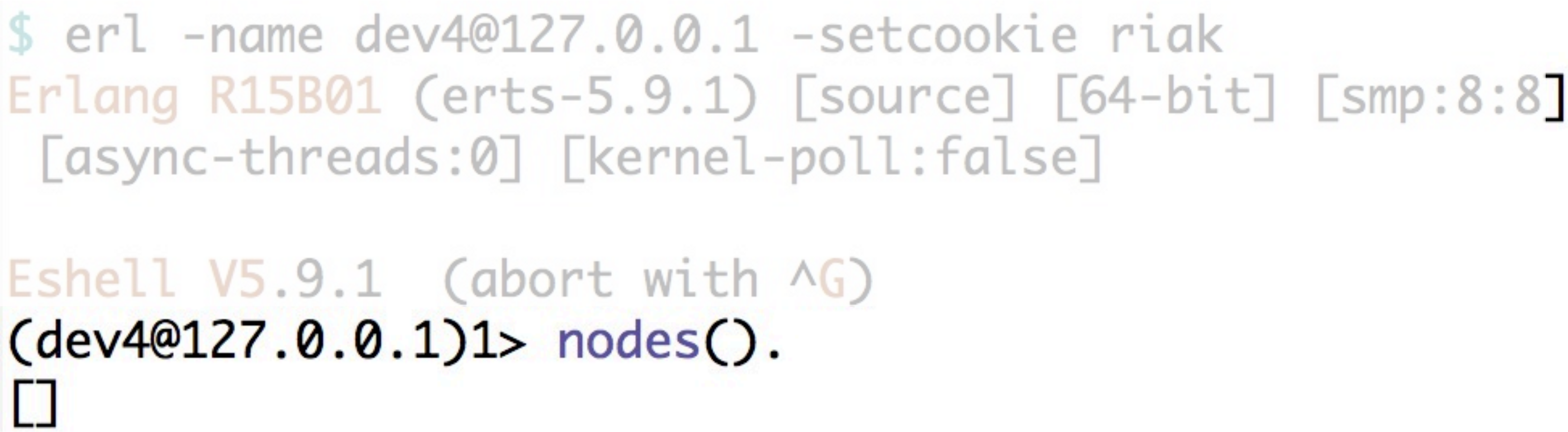
Distributed Erlang



```
$ 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)
```

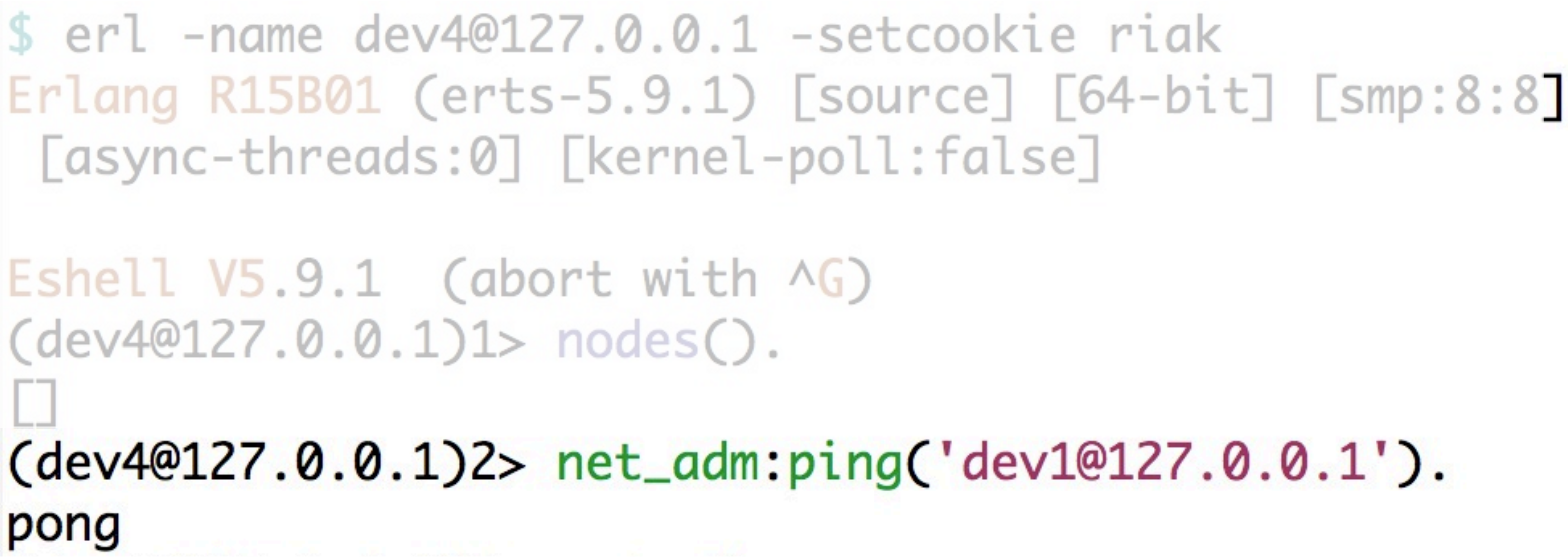
Distributed Erlang



```
$ 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().
[]
```

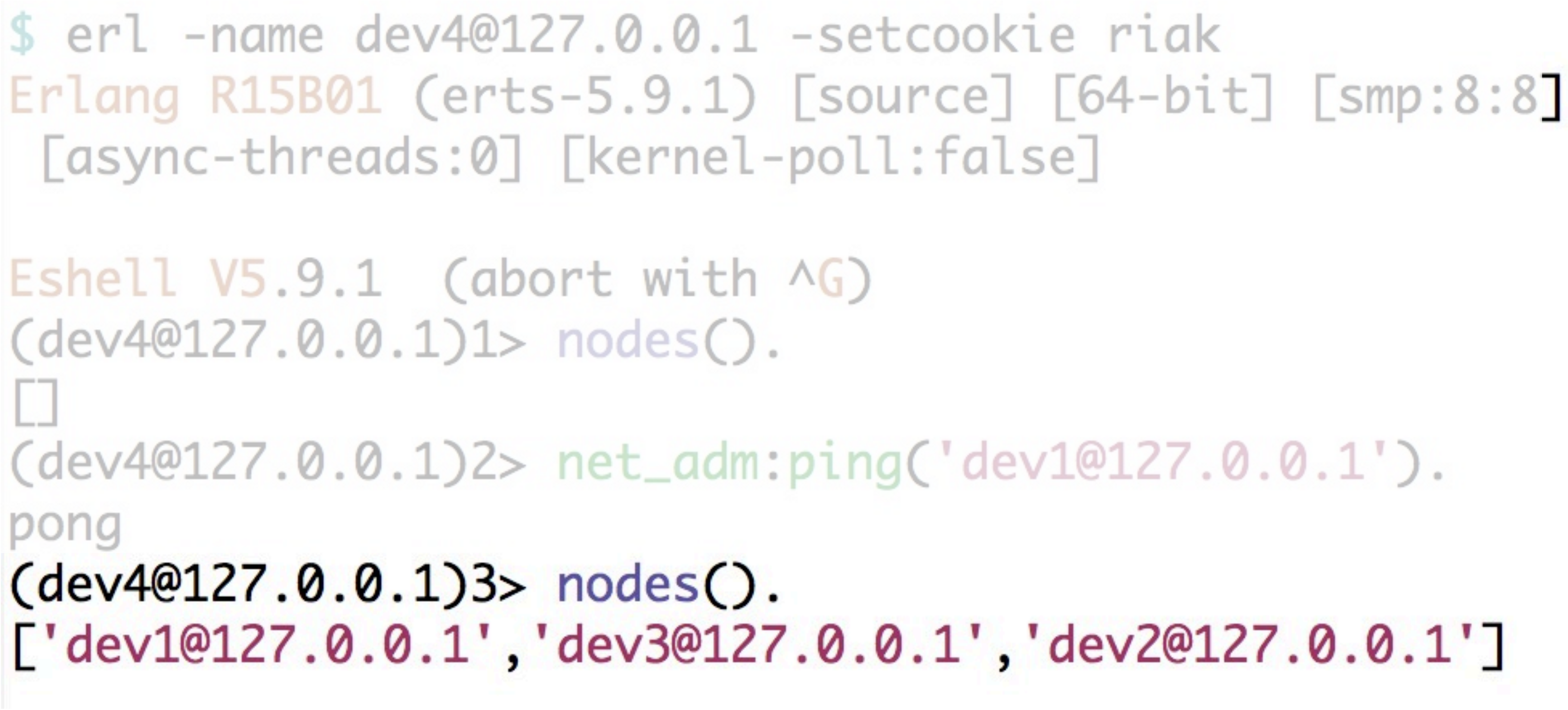
Distributed Erlang



```
$ 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
```

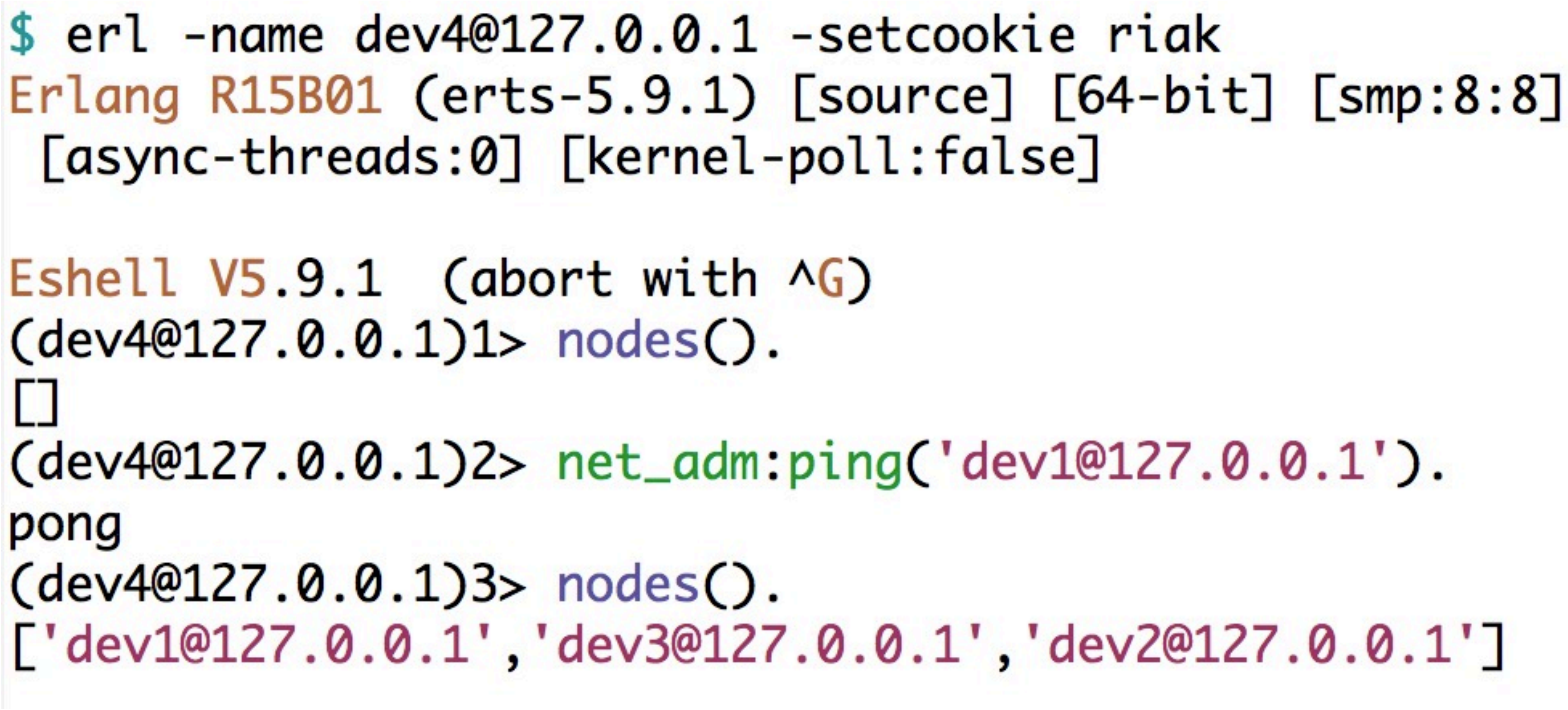
Distributed Erlang



```
$ 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']
```

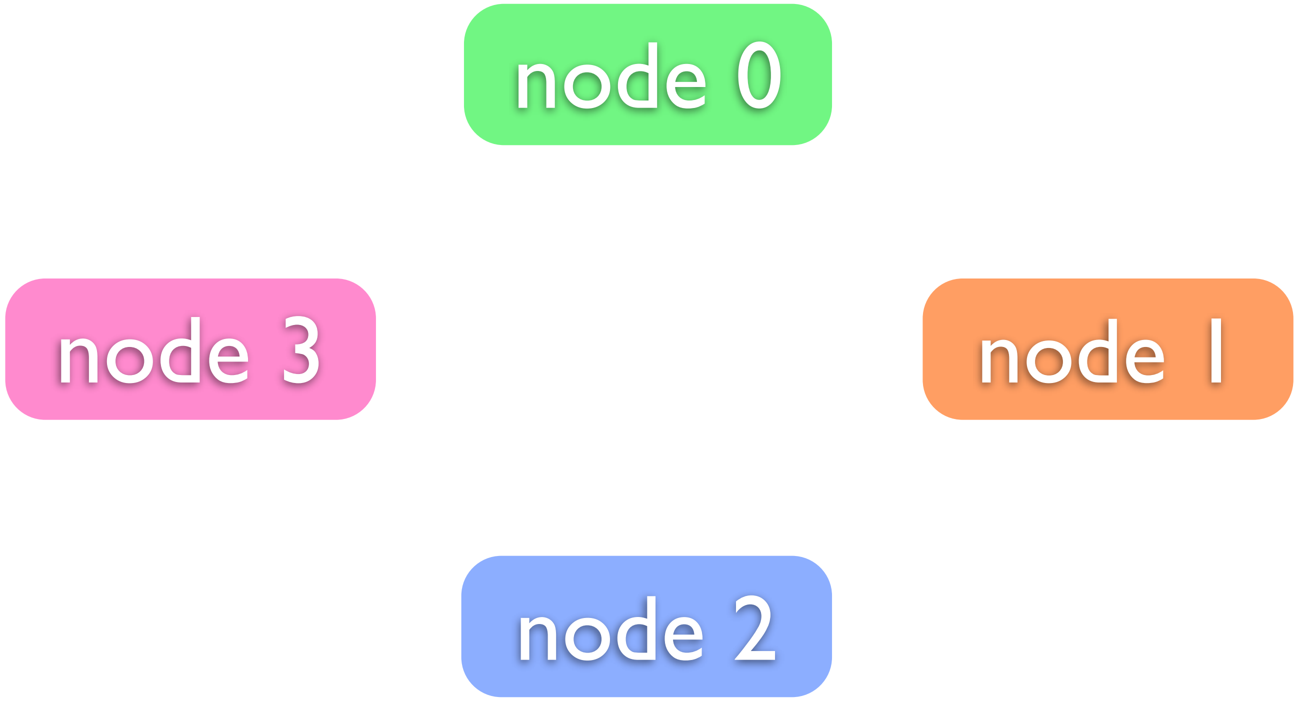
Distributed Erlang



```
$ 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



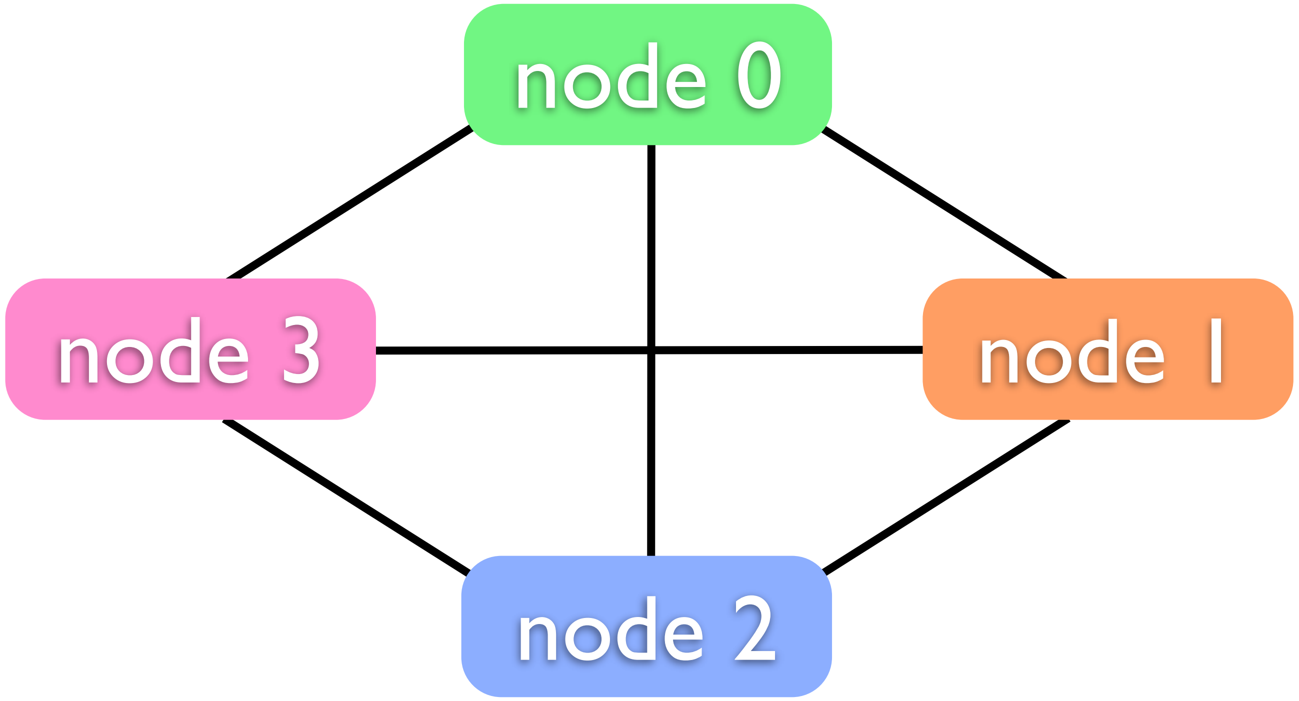
node 0

node 3

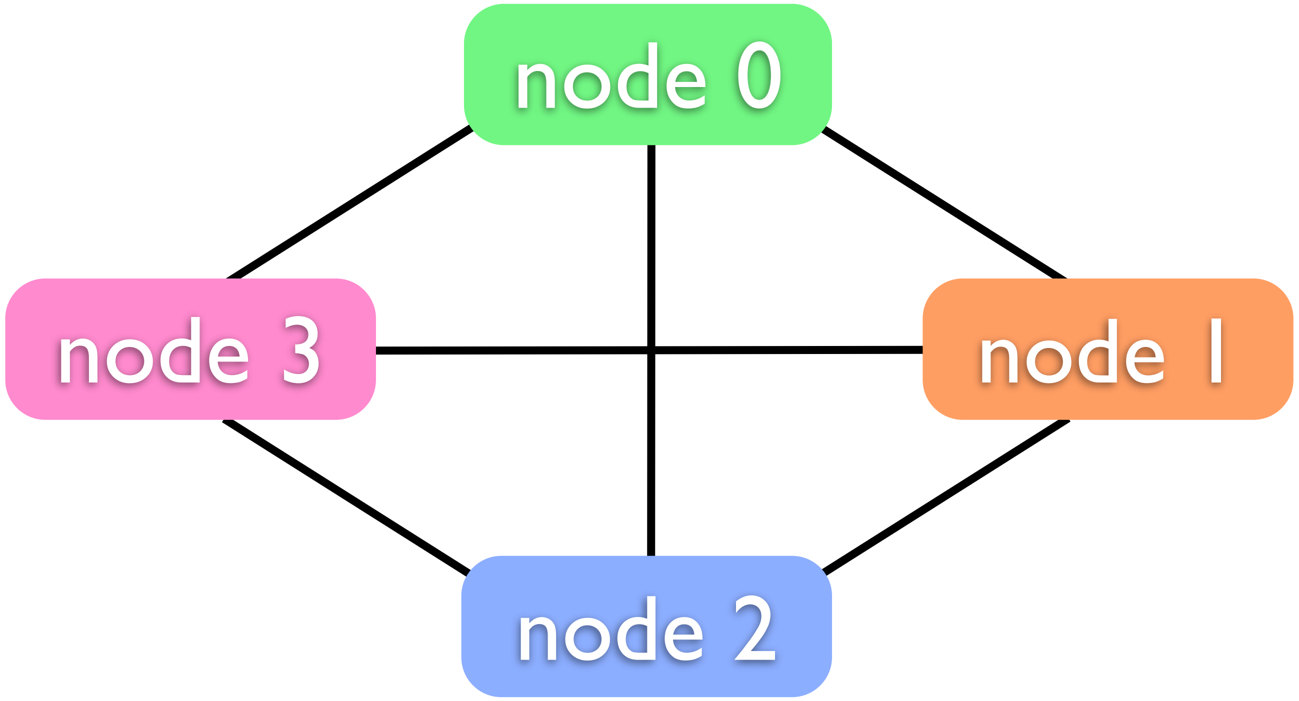
node 1

node 2

Distributed Erlang Mesh



Distributed Erlang Mesh

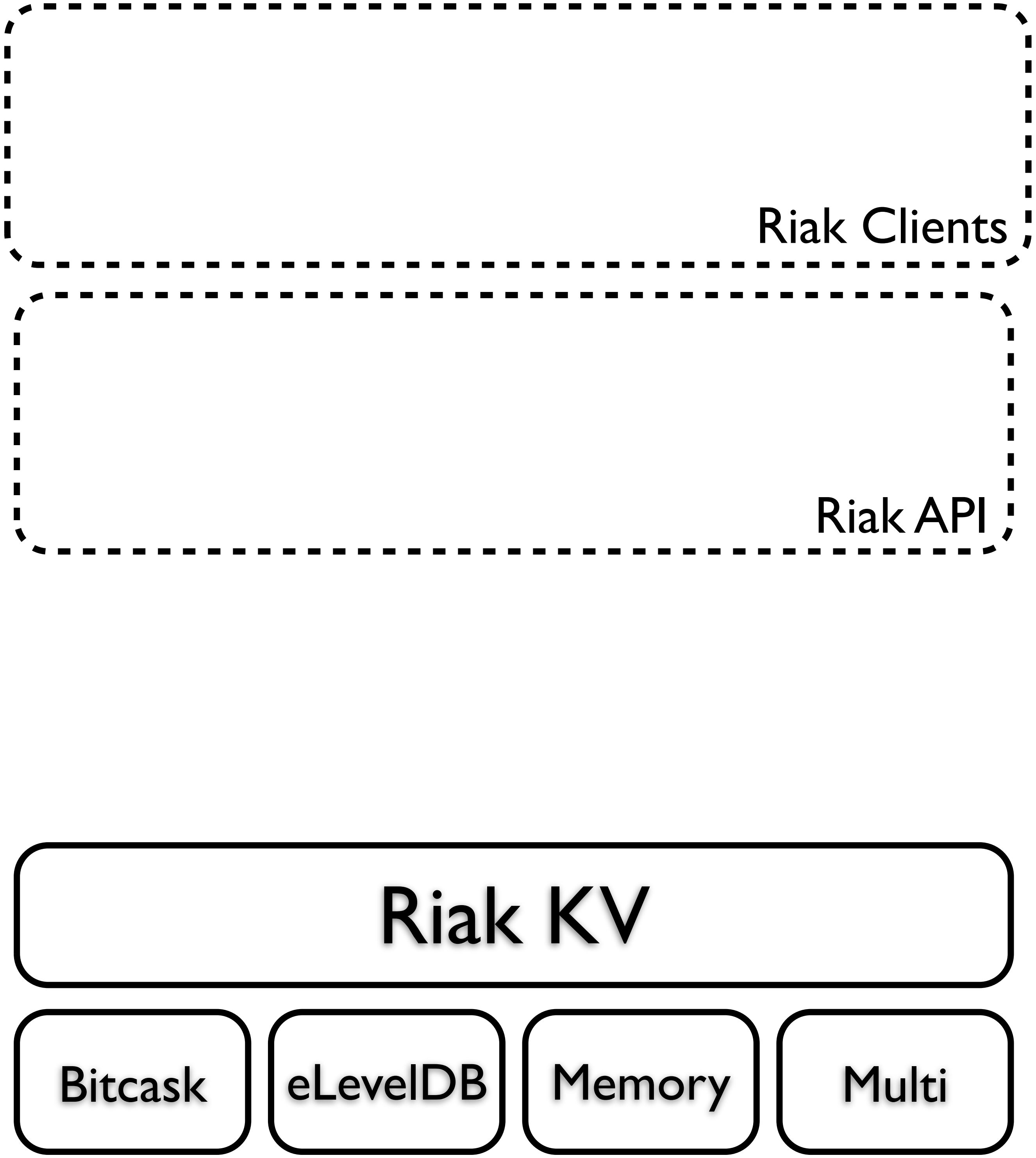


- Caveat: mesh housekeeping runs into scaling issues as the cluster grows large

Gossip

- Nodes periodically send their understanding of the ring state to other randomly chosen nodes
- Gossip module also provides an API for sending ring state to specific nodes

Riak Core



Riak Clients

Riak API

Riak KV

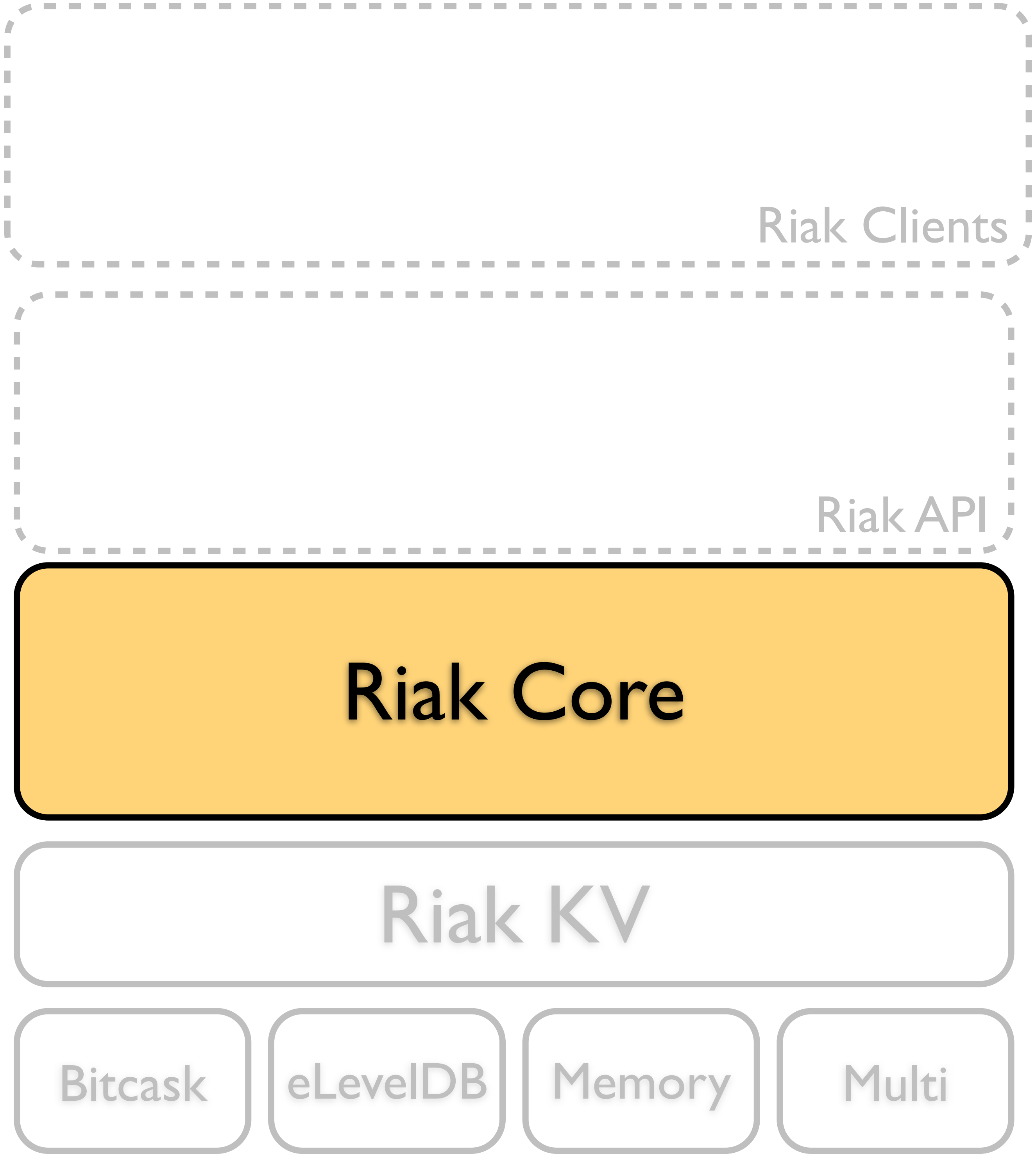
Bitcask

eLevelDB

Memory

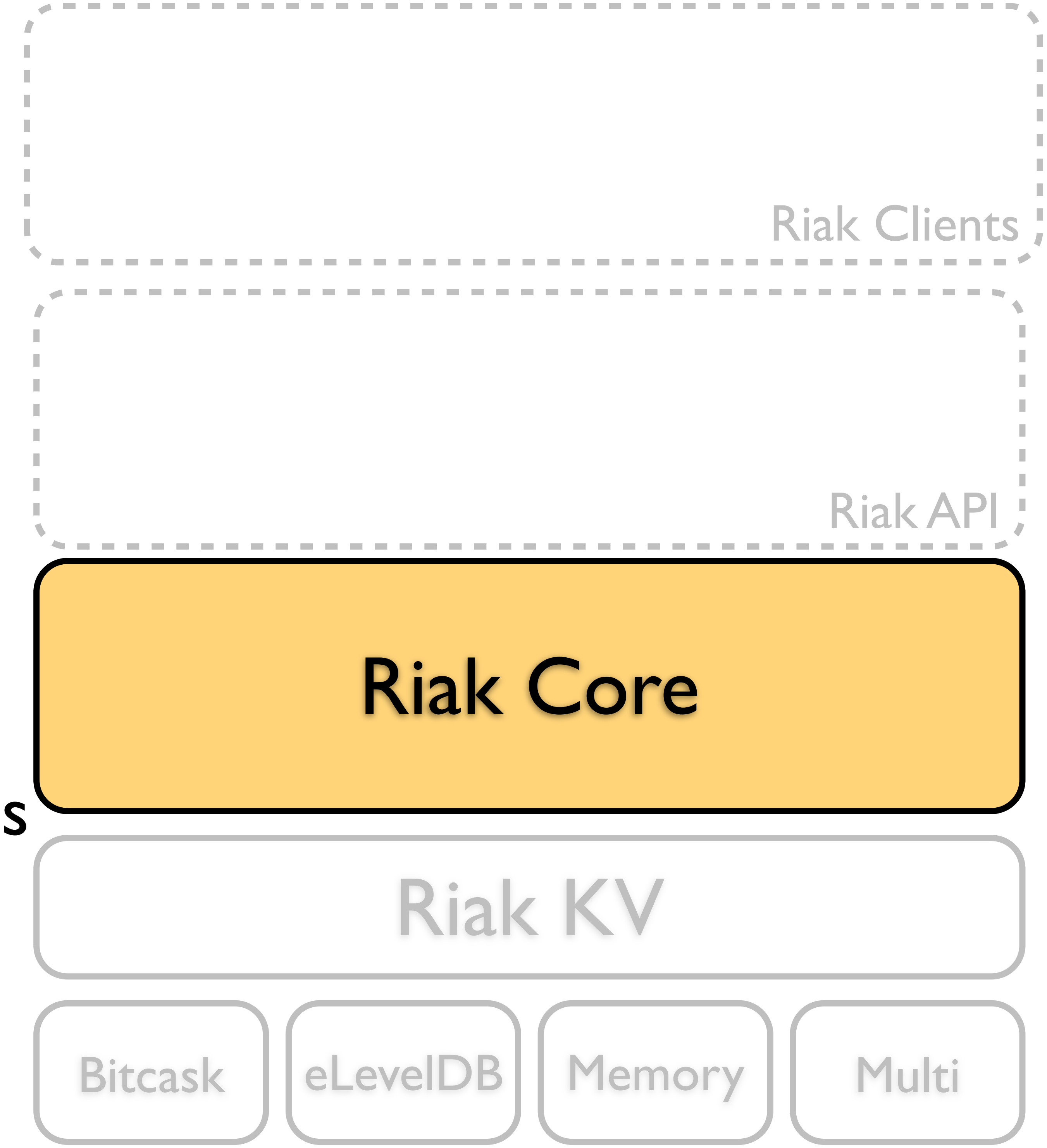
Multi

Riak Core


Riak Clients
Riak API
Riak Core
Riak KV
Bitcask
eLevelDB
Memory
Multi

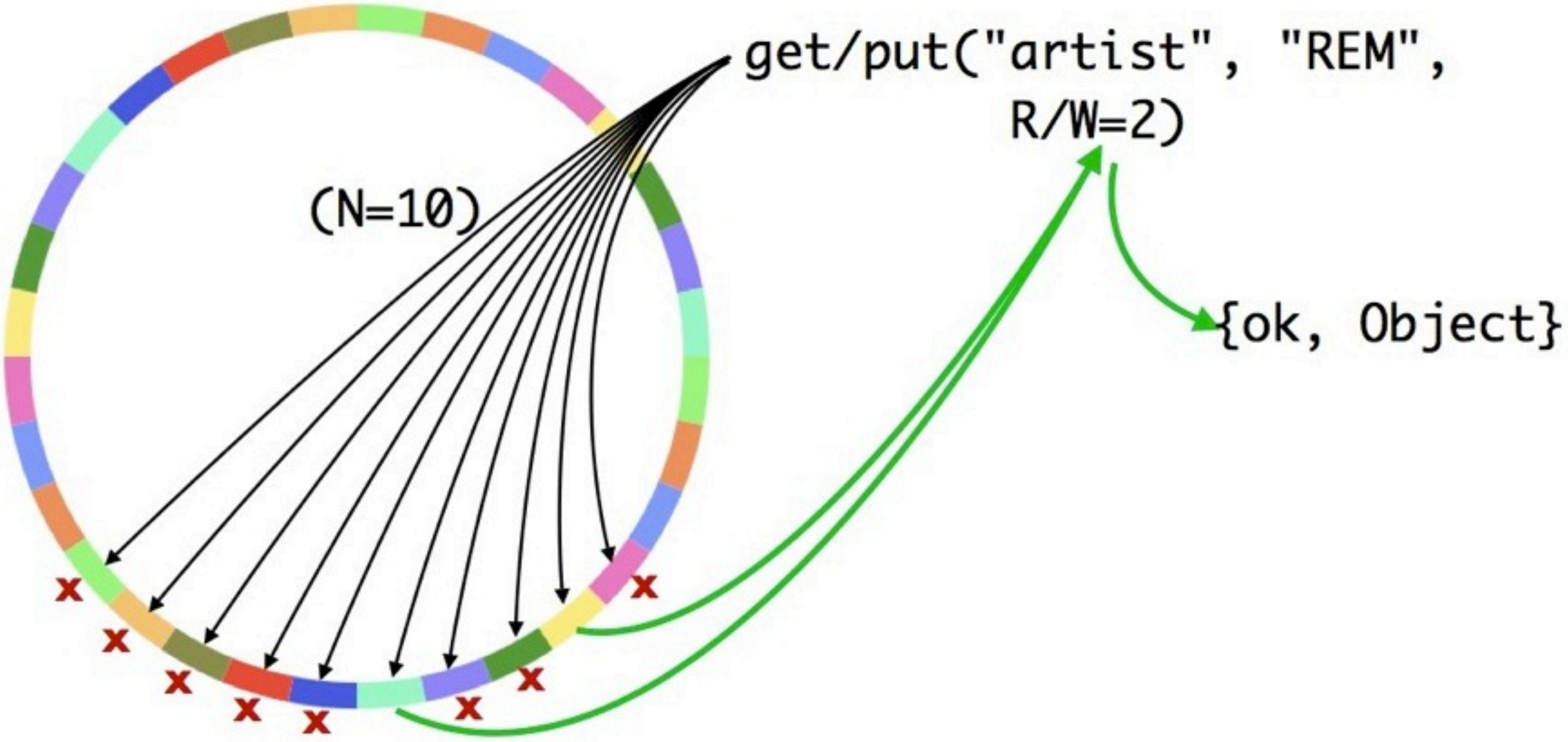
Riak Core

- consistent hashing
- vector clocks
- sloppy quorums



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

N/R/W Values



Hinted Handoff

Hinted Handoff

- Fallback vnode holds data for unavailable actual vnode

Hinted Handoff

- Fallback vnode holds data for unavailable actual vnode
- Fallback vnode keeps checking for availability of actual vnode

Hinted Handoff

- Fallback vnode holds data for unavailable actual vnode
- Fallback vnode keeps checking for availability of actual vnode
- Once actual vnode becomes available, fallback hands off data to it

Old Issue with Handoff

- Handoff can require shipping megabytes of data over the network
- Used to be a hard-coded 128kb limit in the Erlang VM for its distribution port buffer
- Hitting the limit caused VM to de-schedule sender until the dist port cleared
- Basho's Scott Fritchie submitted an Erlang patch that allows the dist port buffer size to be configured (Erlang version R14B01)

Read Repair

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

Core Protocols

- Gossip, handoff, read repair, etc. all require intra-cluster protocols
- Erlang features help significantly with protocol implementations

Binary Handling

- Erlang's binaries make working with network packets easy
- For example, deconstructing a TCP message (from Cesarini & Thompson “Erlang Programming”)

Binary Handling

```
TcpBuf.
```

Binary Handling

```
<<SourcePort:16, DestinationPort:16,
  SequenceNumber:32, AckNumber:32,
  DataOffset:4, _Rsrvd:4, Flags:8,
  WindowSize:16, Checksum:16,
  UrgentPtr:16,
  Data/binary>> = TcpBuf.
```

Binary Handling

```
<<SourcePort:16, DestinationPort:16,
  SequenceNumber:32, AckNumber:32,
  DataOffset:4, _Rsrvd:4, Flags:8,
  WindowSize:16, Checksum:16,
  UrgentPtr:16,
  Data/binary>> = TcpBuf.
```

Protocols with OTP

- OTP provides libraries of standard modules
- And also **behaviours:** implementations of common patterns for concurrent, distributed, fault-tolerant Erlang apps

OTP Behaviour Modules

- A behaviour is similar to an abstract base class in OO terms, providing:
  - a message handling loop
  - integration with underlying OTP system (for code upgrade, tracing, process management, etc.)

OTP Behaviors

- application
- supervisor
- gen_server
- gen_fsm
- gen_event

gen_server

- Generic server behaviour 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

- Behaviour supporting finite state machines (FSMs)
- Same tail-call loop for maintaining state as 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_*

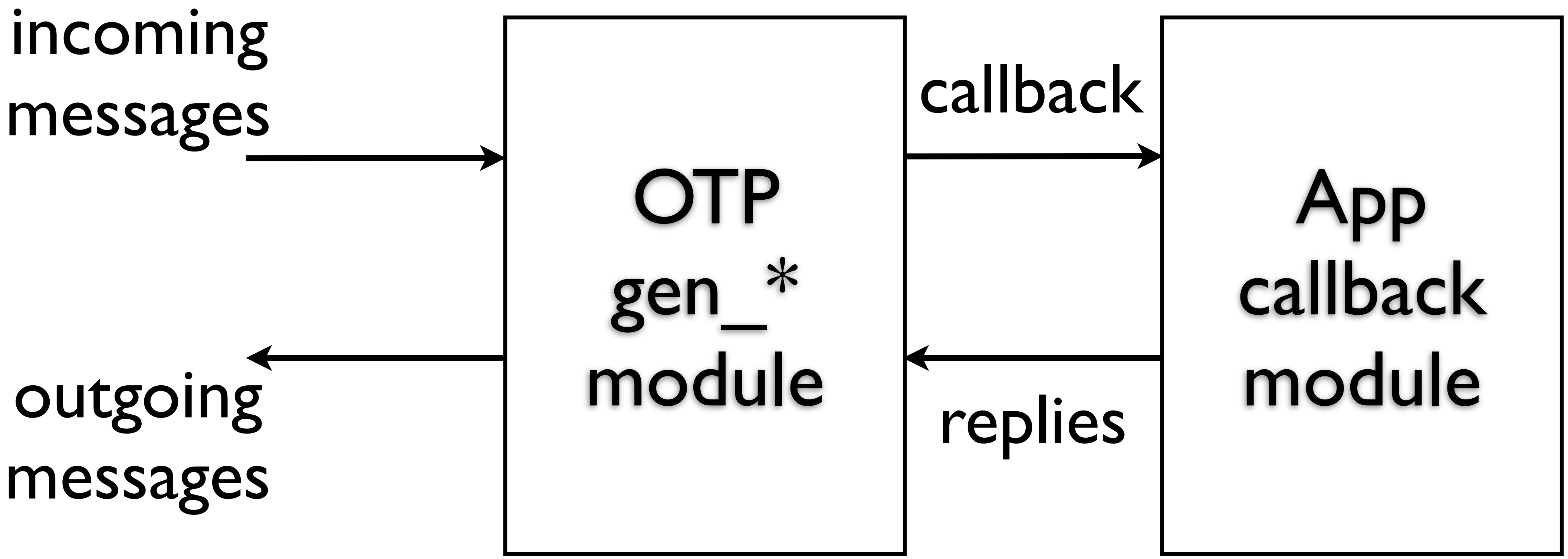
- Riak makes heavy use of these behaviours, e.g.:
  - FSMs for get and put operations
  - Vnode FSM
  - Gossip module is a gen_server

Behaviour Benefits

- Standardized frameworks providing common patterns, common vocabulary
- Used by pretty much all non-trivial Erlang systems
- Erlang developers understand them, know how to read them

Behaviour Benefits

- Separate a lot of messaging, debugging, tracing support, system concerns from business logic



application Behaviour

- Provides an entry point for an OTP-compliant app
- Allows multiple Erlang components to be combined into a system
- Erlang apps can declare their dependencies on other apps
- A running Riak system comprises about 30 applications

App Startup Sequence

- Hierarchical sequence
- Erlang system application controller starts the app
- App starts supervisor(s)
- Each supervisor starts workers
- Workers are typically instances of OTP behaviors

Workers & Supervisors

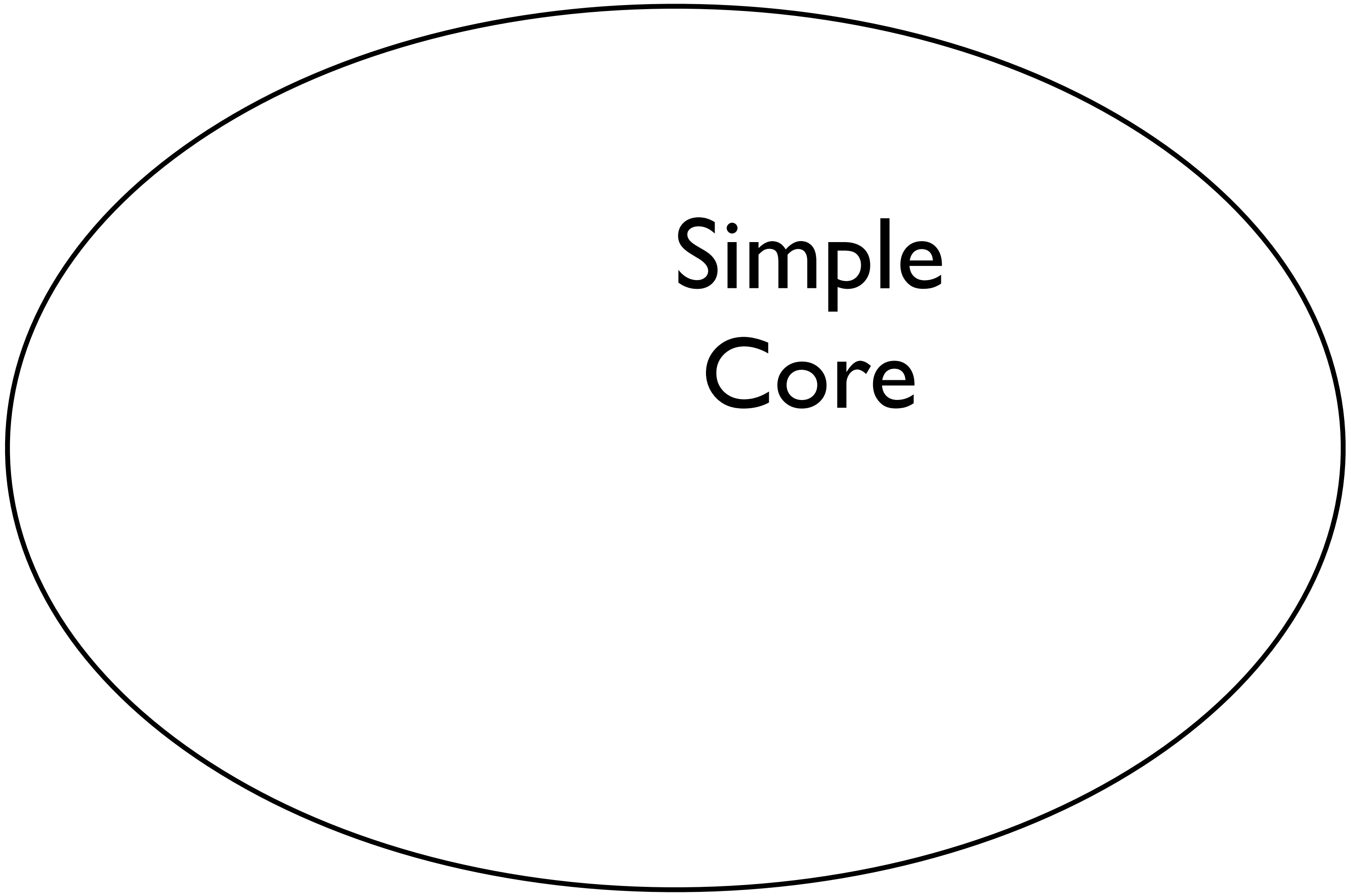
- Workers implement application logic
- Supervisors:
  - start child workers and sub-supervisors
  - link to the children and trap child process exits
  - take action when a child dies, typically restarting one or more children

Let It Crash

- In his doctoral thesis, Joe Armstrong, creator of Erlang, wrote:
  - *Let some other process do the error recovery.*
  - *If you can't do what you want to do, die.*
  - *Let it crash.*
  - *Do not program defensively.*

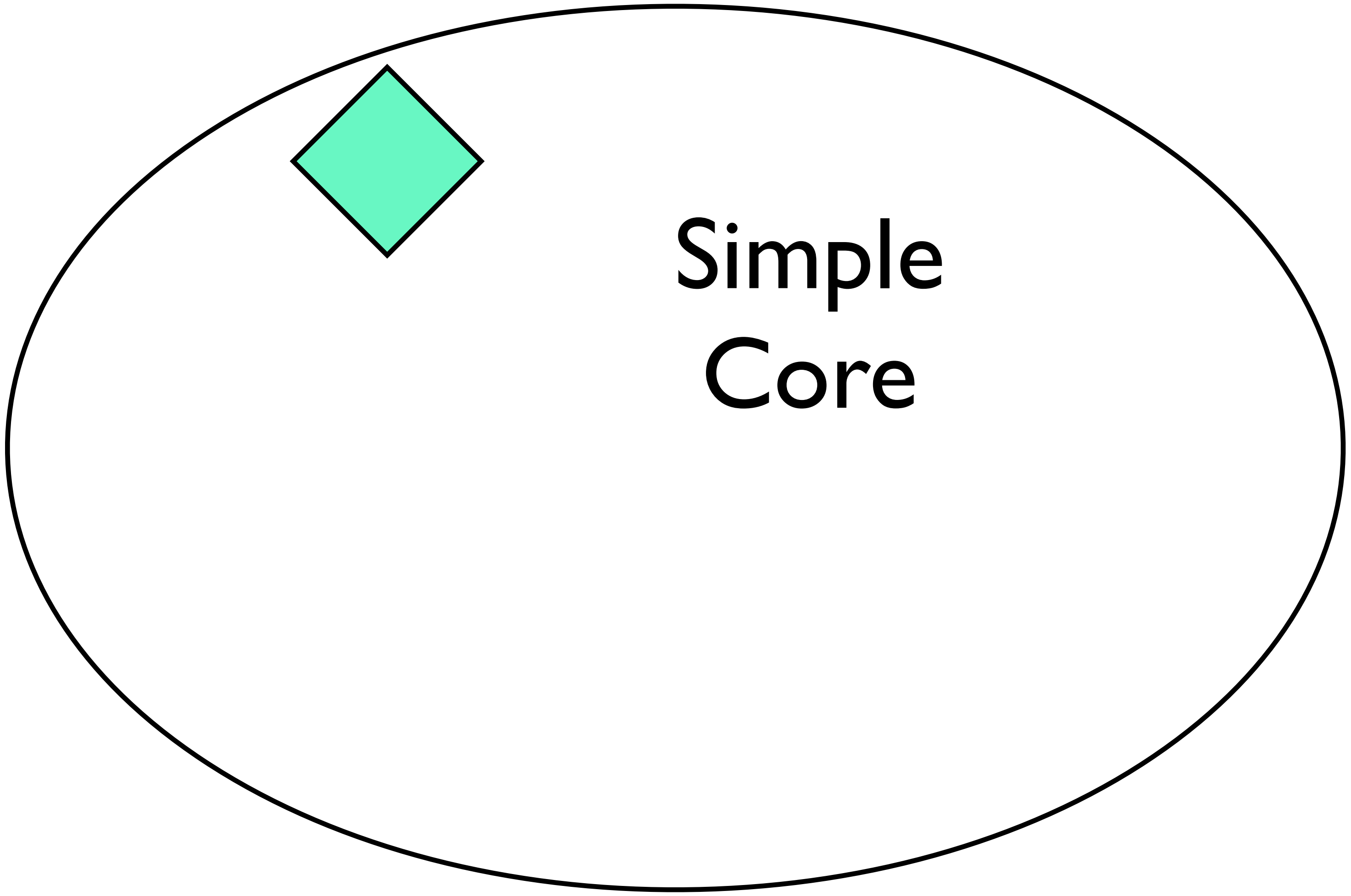
see http://www.erlang.org/download/armstrong_thesis_2003.pdf

Application, Supervisors, Workers



Application, Supervisors, Workers

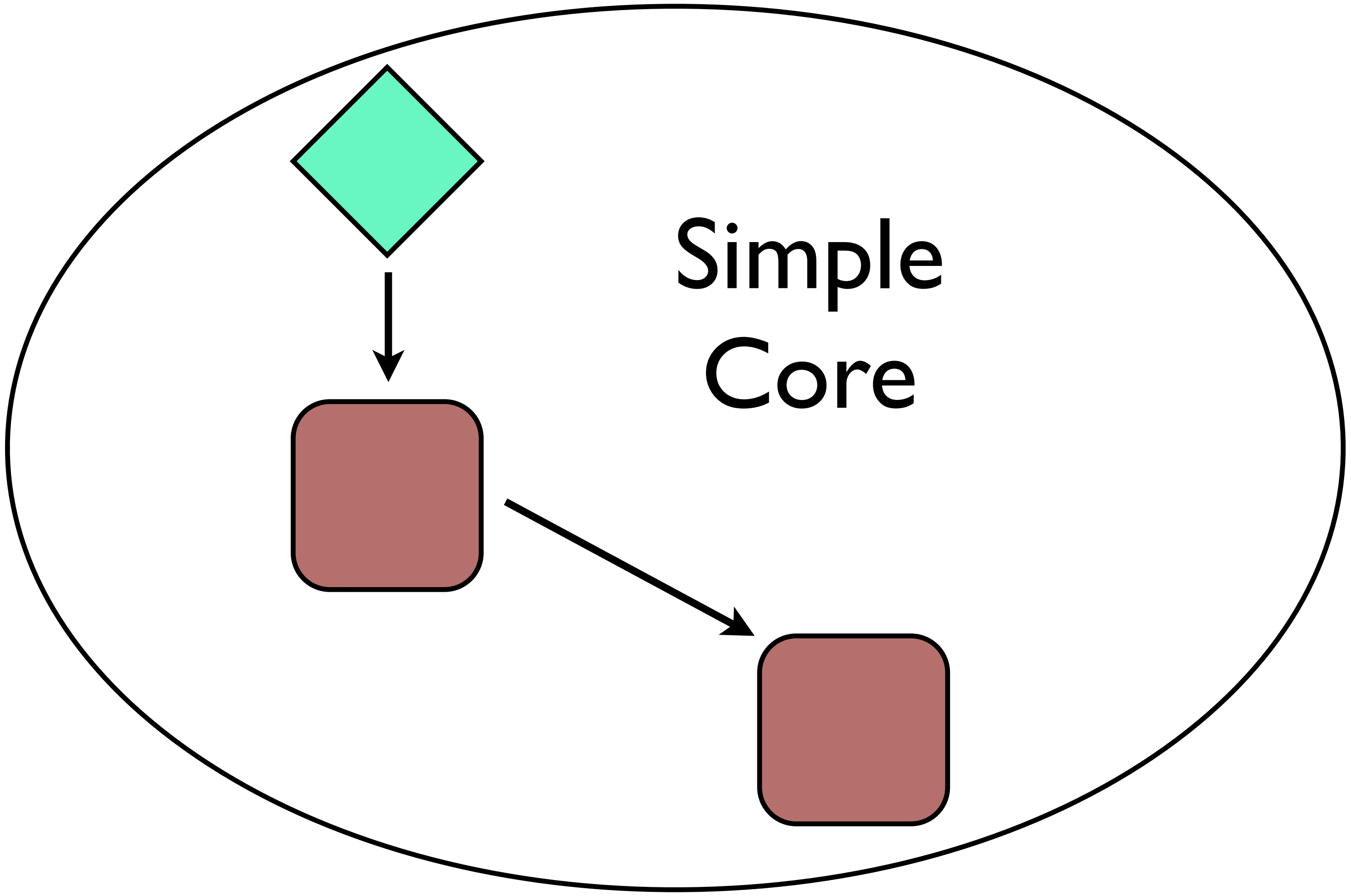
Application



Application, Supervisors, Workers

Application

Supervisors

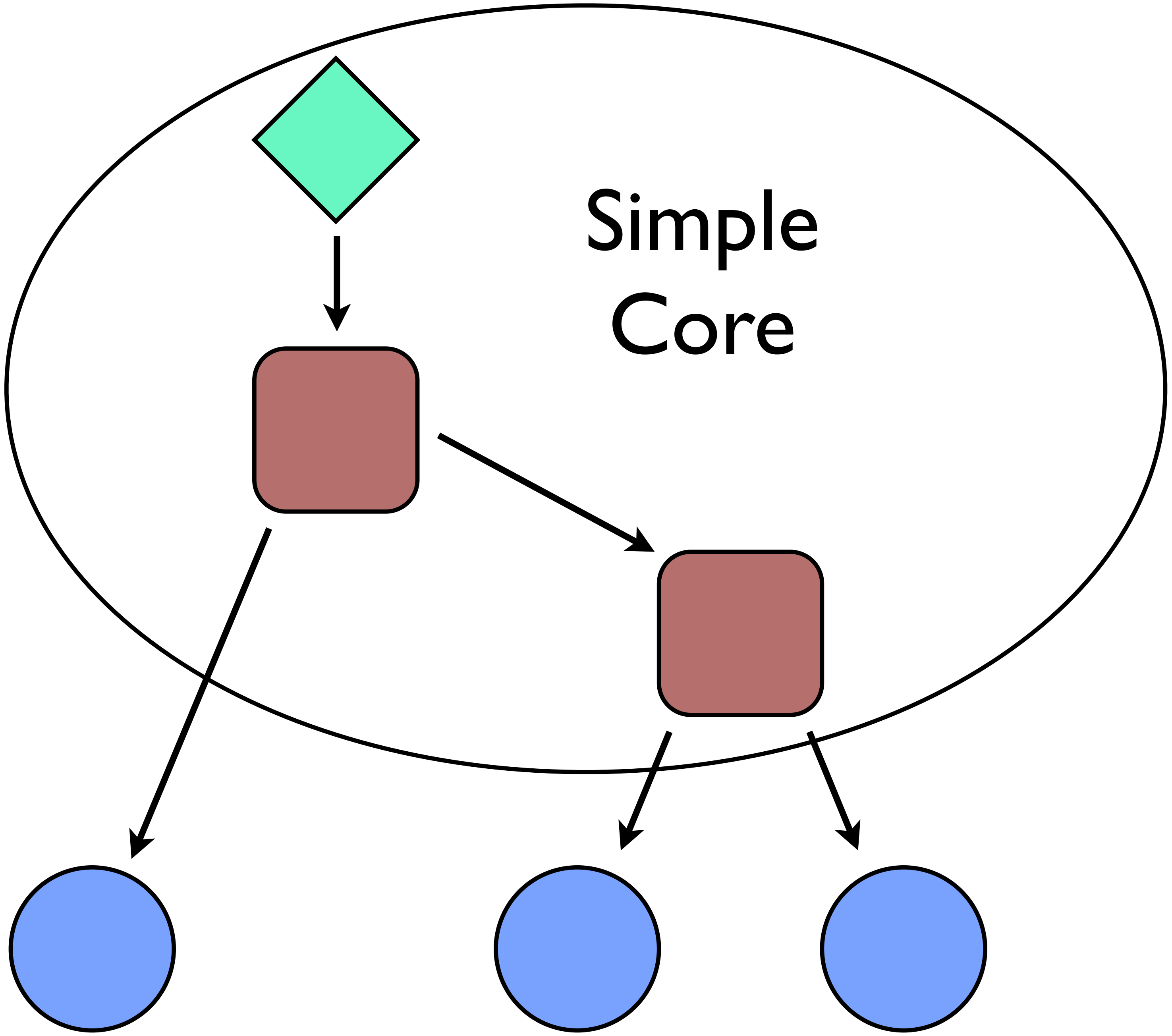


Application, Supervisors, Workers

Application

Supervisors

Workers



Simple Core

OTP System Facilities

OTP System Facilities

- Status

OTP System Facilities

- Status
- Process info

OTP System Facilities

- Status
- Process info
- Tracing

OTP System Facilities

- Status
- Process info
- Tracing
- The above work with OTP-compliant behaviours, very useful for debug

OTP System Facilities

- Status
- Process info
- Tracing
- The above work with OTP-compliant behaviours, very useful for debug
- Releases

OTP System Facilities

- Status
- Process info
- Tracing
- The above work with OTP-compliant behaviours, very useful for debug
- Releases
- Live upgrades

Integration

Riak Architecture

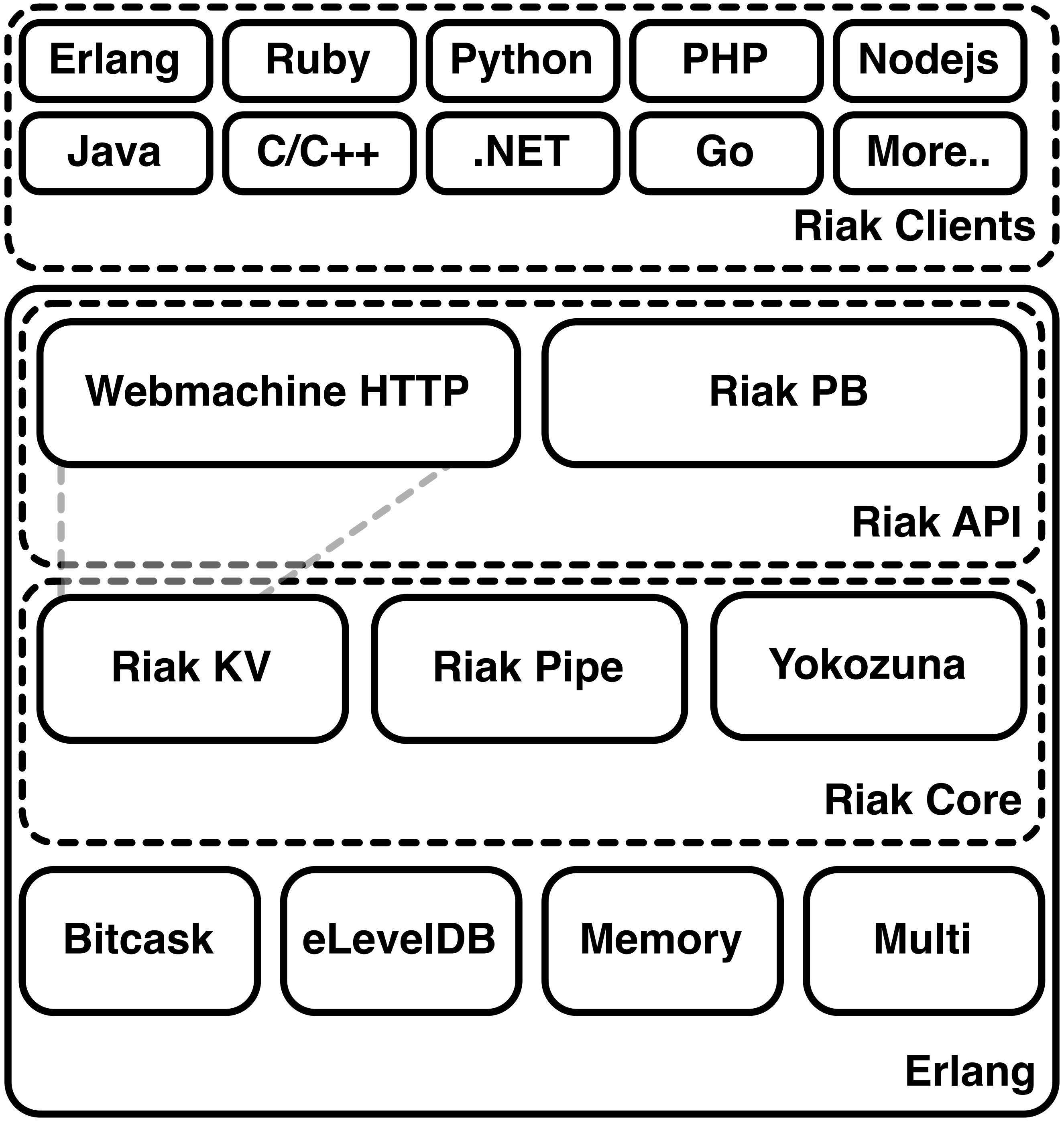


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

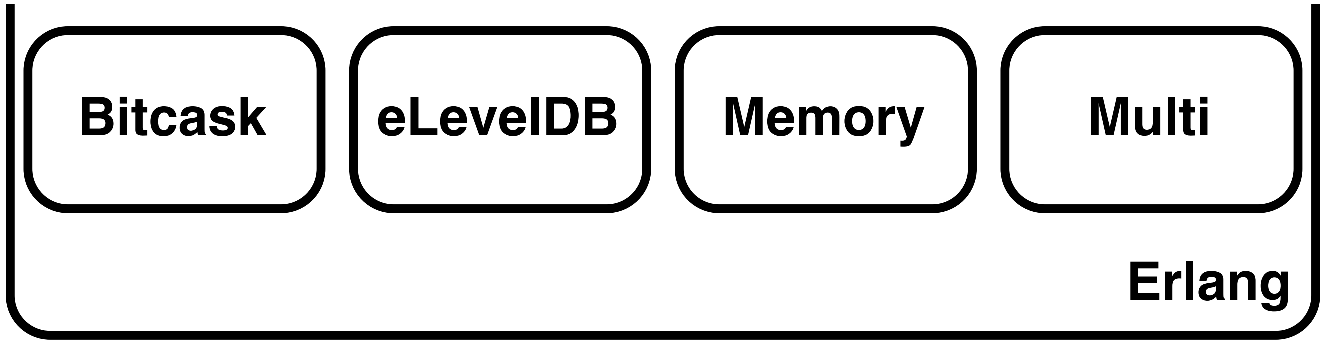
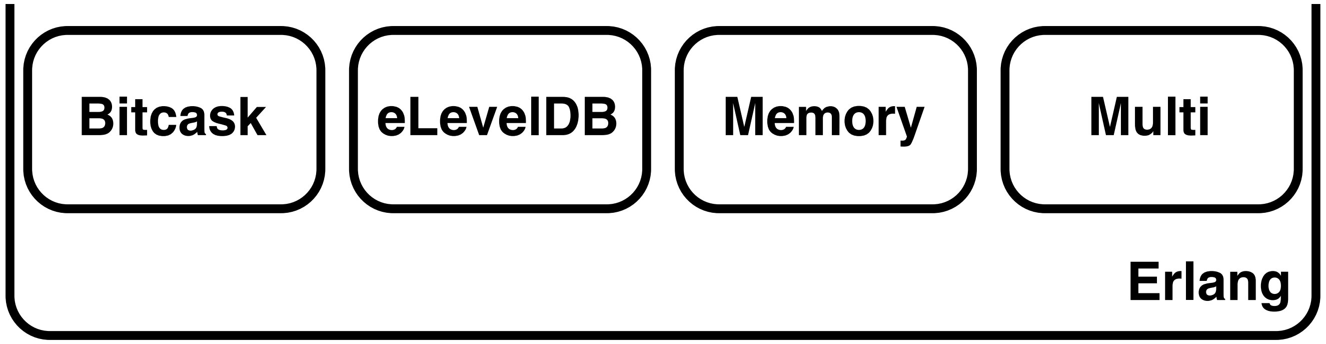


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

Erlang 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 driver interface
  - for example the VM supplies network and file system facilities via drivers
- Another way is through Native Implemented Functions (NIFs)

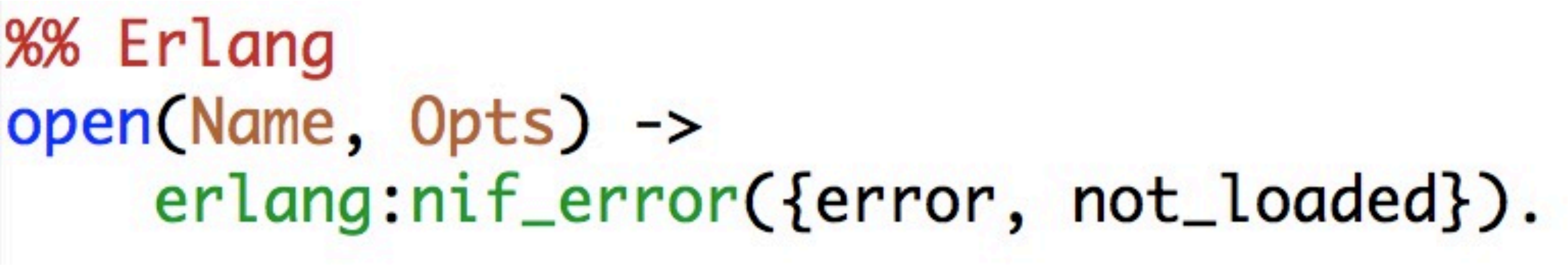
Native Implemented Functions (NIFs)

- 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

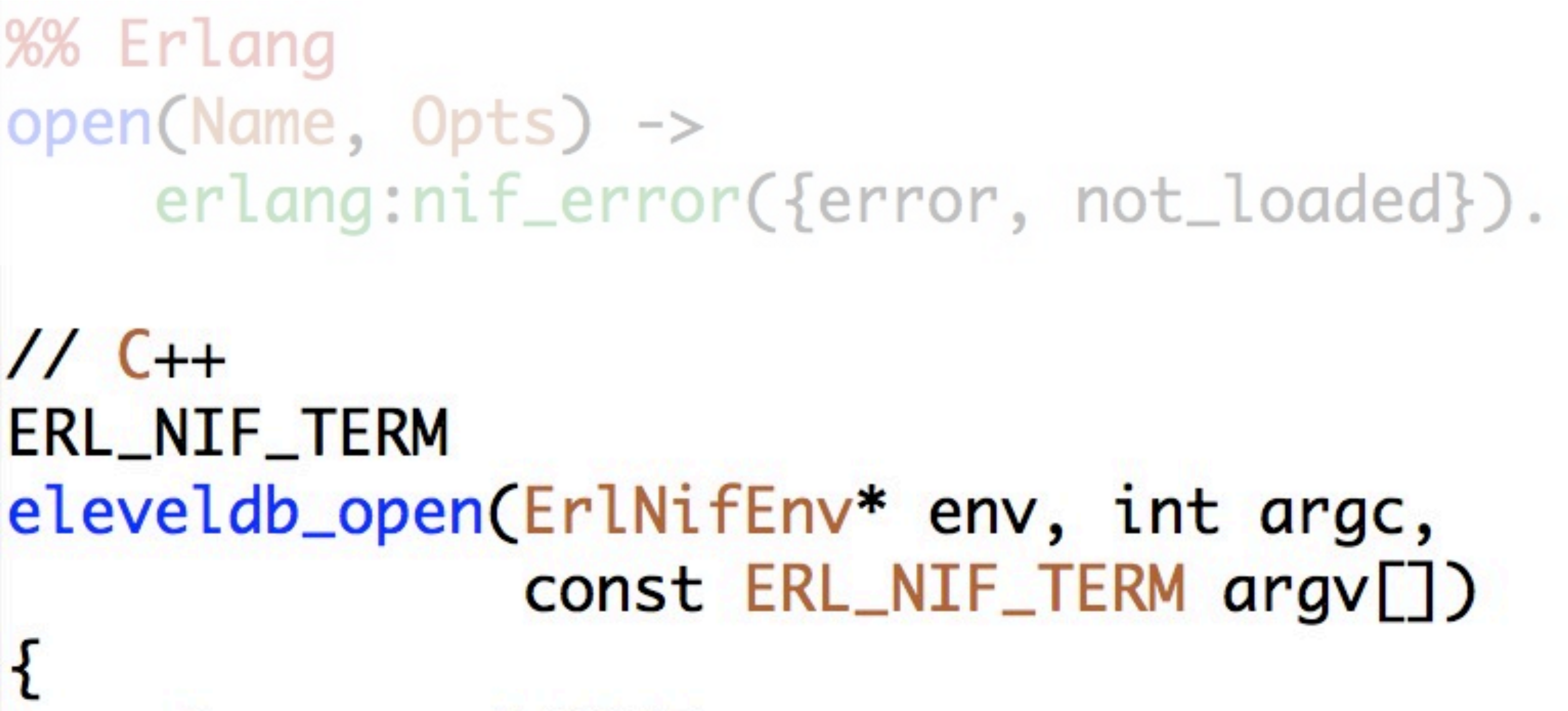
- NIF wrapper around Google's LevelDB C++ database
- Erlang interface plugs in underneath Riak KV

Example: eleveldb



```
%% Erlang
open(Name, Opts) ->
    erlang:nif_error({error, not_loaded}).
```

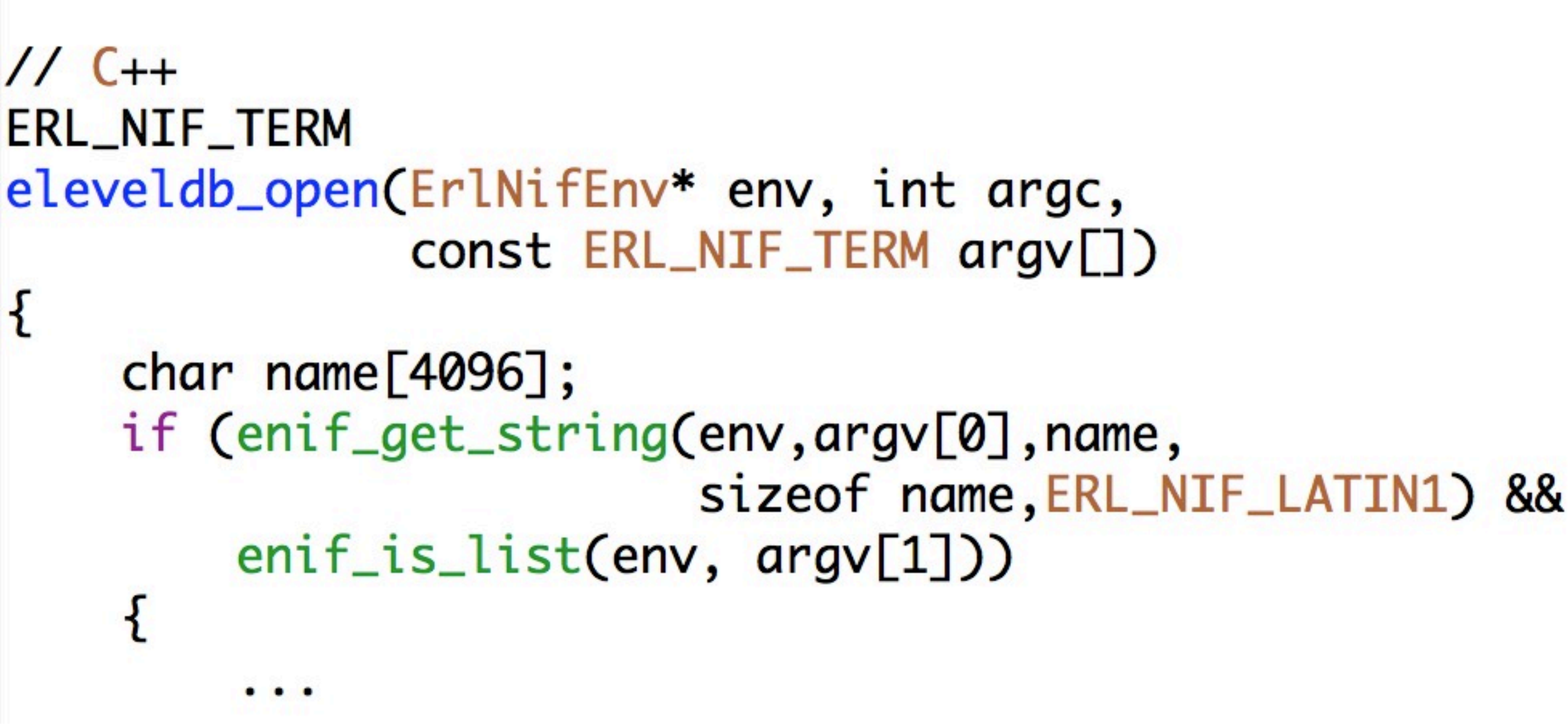
Example: eleveldb



```
%% Erlang
open(Name, Opts) ->
    erlang:nif_error({error, not_loaded}).

// C++
ERL_NIF_TERM
eleveldb_open(ErlNifEnv* env, int argc,
              const ERL_NIF_TERM argv[])
{
```

Example: eleveldb



```
// C++
ERL_NIF_TERM
eleveldb_open(ErlNifEnv* env, int argc,
              const ERL_NIF_TERM argv[])
{
    char name[4096];
    if (enif_get_string(env,argv[0],name,
                        sizeof name,ERL_NIF_LATIN1) &&
        enif_is_list(env, argv[1]))
    {
        ...
```

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.)

NIF Caveats

NIF Caveats

- Crashes in your linked-in C/C++ kill the whole VM

NIF Caveats

- Crashes in your linked-in C/C++ kill the whole VM
- Lesson: use NIFs and drivers only when needed, and don't write crappy code

NIF Caveats

NIF Caveats

- NIF calls execute within a VM scheduler thread

NIF Caveats

- NIF calls execute within a VM scheduler thread
- If the NIF blocks, the scheduler thread blocks

NIF Caveats

- NIF calls execute within a VM scheduler thread
- If the NIF blocks, the scheduler thread blocks
- THIS IS VERY BAD

NIF Caveats

- NIF calls execute within a VM scheduler thread
- If the NIF blocks, the scheduler thread blocks
- THIS IS VERY BAD
- NIFs should block for no more than 1 millisecond

NIF Caveats

NIF Caveats

- Basho found "scheduler anomalies" where

NIF Caveats

- Basho found "scheduler anomalies" where
  - the VM would put most of its schedulers to sleep, by design, under low load

NIF Caveats

- Basho found "scheduler anomalies" where
    - the VM would put most of its schedulers to sleep, by design, under low load
    - but would fail to wake them up as load increased

NIF Caveats

- Basho found "scheduler anomalies" where
  - the VM would put most of its schedulers to sleep, by design, under low load
  - but would fail to wake them up as load increased
- Believe it's caused by NIF calls that were taking multiple seconds in some cases

NIF Caveats

- Basho found "scheduler anomalies" where
  - the VM would put most of its schedulers to sleep, by design, under low load
  - but would fail to wake them up as load increased
- Believe it's caused by NIF calls that were taking multiple seconds in some cases
- Lesson: put long-running activities in their own threads

Testing

Eunit

- Erlang's unit testing facility
- Support for asserting test results, grouping tests, setup and teardown, etc.
- Unit tests typically live in the same module as the code they test, but are conditionally compiled in only for testing
- Used heavily in Riak

QuickCheck

- Property-based testing product from Quviq
- John Hughes will be giving a talk about this later today, you should definitely attend

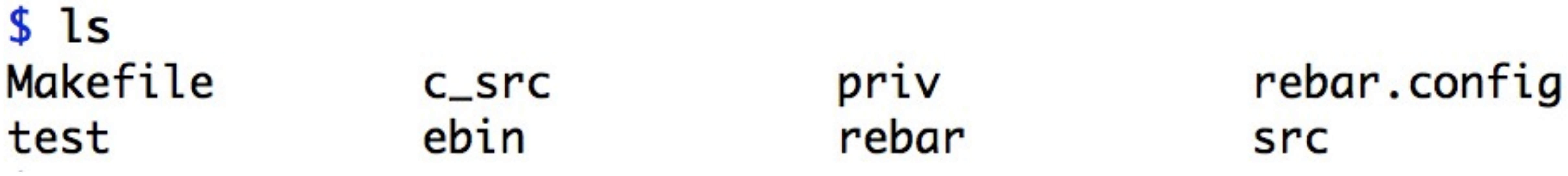
QuickCheck

- 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 quite heavily in Riak, especially to test various protocols and interactions

Build and Release

Application Directories

- Erlang applications tend to use a standard directory layout
- Certain tools expect to find this layout



```
$ ls
Makefile         c_src            priv             rebar.config
test             ebin             rebar            src
```

Rebar

- A tool created by Dave "Dizzy" Smith (formerly of Basho) to manage Erlang apps
- Manages dependencies, builds, runs tests, generates releases
- Now the de facto app build and release tool

Miscellaneous

Miscellaneous

- Memory
- Erlang shell
- Hot code loading
- Logging
- VM knowledge
- Hiring

Memory

- Process message queues have no limits, can cause out-of-memory conditions if a process can't keep up
- VM dies by design if it runs out of memory
- Riak runs a memory monitor to help log out-of-memory conditions

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 for customers

Logging

- Non-Erlang folks have a hard time reading Erlang logs
- Andrew Thompson of Basho wrote Lager to help address this
- Lager translates Erlang logging into something regular people can deal with
  - also logs original Erlang to keep all the details
- But does more than that, see https://github.com/basho/lager for details

VM Knowledge

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

Hiring

- Erlang is easy to learn
- Not really a problem to hire Erlang programmers
- Basho hires great developers, those who need to learn Erlang just do it
- BTW we're hiring, see http://bashojobs.theresumator.com

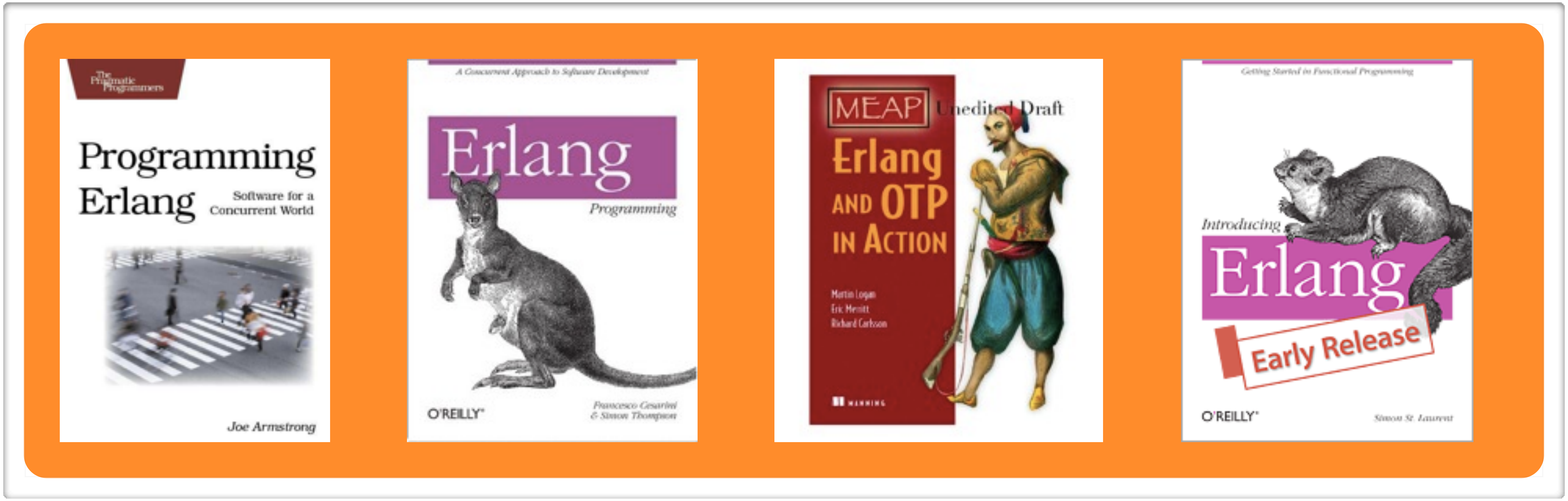
Summary

- Erlang/OTP is an amazing system for developing distributed systems like Riak
- It's very much a DSL for distributed concurrent systems
- It does what it says on the tin

Summary

- Erlang code is relatively small, easy to read, write, and maintain
- Tools support the entire software lifecycle
- Erlang community is friendly and fantastic

For More Erlang Info



Also: http://learnyousomeerlang.com/

For More Riak Info

- "A Little Riak Book" by Basho's Eric Redmond
  https://github.com/coderoshi/little_riak_book/
- 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