

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

- 1980s: Ericsson Computer Science Laboratories (CSL) sought better ways to program telecom switches
- Joe Armstrong created Erlang based on Prolog, with influence from ML, Ada, Smalltalk, other languages
- Open sourced in 1998, available from <http://erlang.org>
- Ericsson develops and maintains it with community help
- Latest production release: R16B (Feb 2013)



Ericsson 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



They Come For The Concurrency...

- Erlang processes are very lightweight, much lighter than OS threads
- Hundreds of thousands or even millions of processes per Erlang VM instance

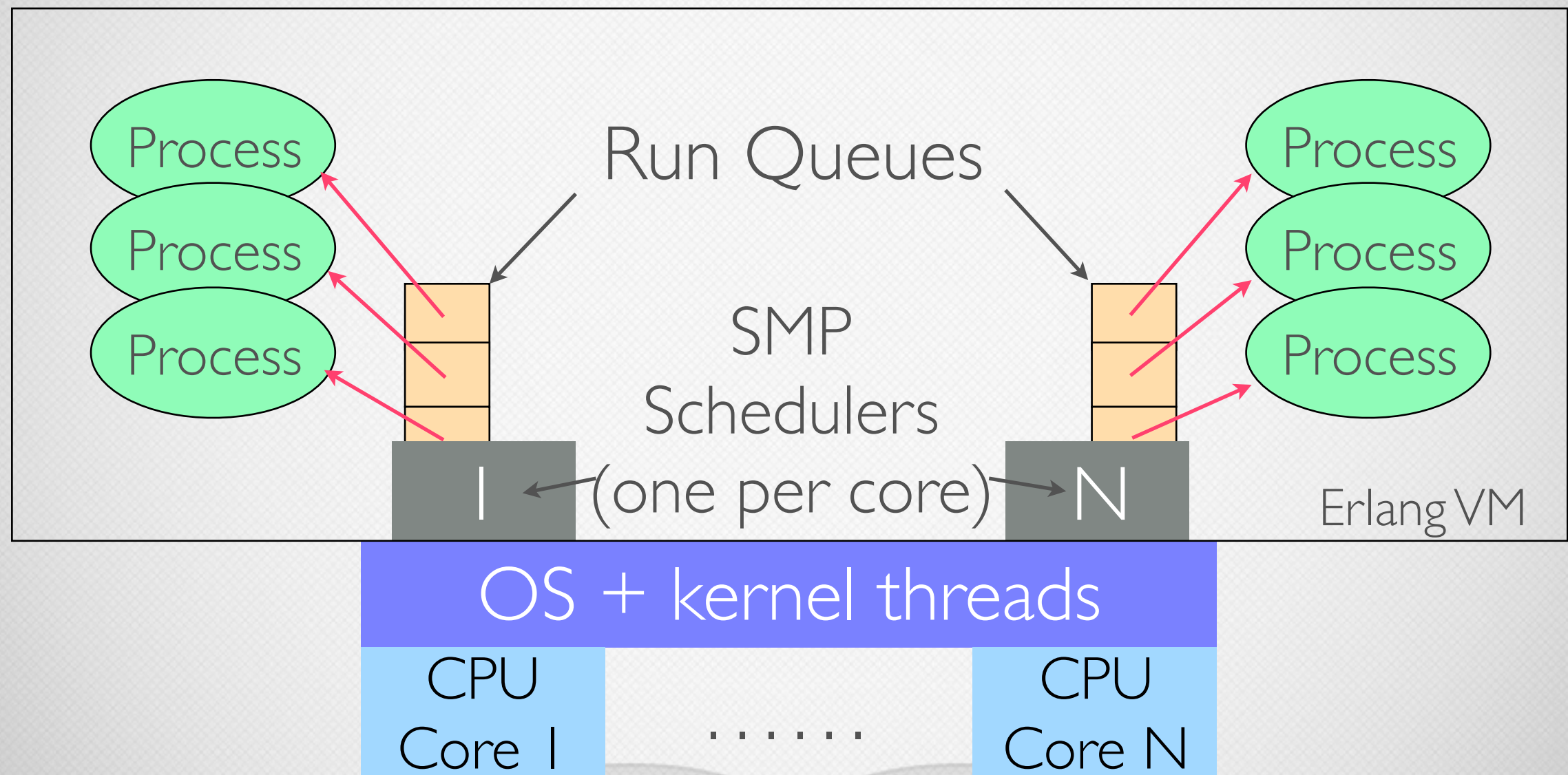


...But They Stay For The Reliability

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



Erlang Process Architecture



A Small Language

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



Concurrency Primitives

- No mutexes, condition variables, or other error-prone concurrency constructs
- All Erlang code runs within some process, always
 - processes are not “extra” like threads in other languages



Concurrency Primitives

- `spawn`: create a new Erlang process
- `!` (exclamation point) or `send`: send a message to another Erlang process, even on another node
- Messages can be any Erlang term
- Messages from A to B arrive in the order sent

```
Pid1 ! ok,  
Pid2 ! [{first, "John"}, {last, "Doe"}].
```



Concurrency Primitives

- Each process has a message queue
- `receive`: receive a message from another Erlang process
- **Selective** `receive` allows receiving specific messages from anywhere within the message queue

```
receive
    {ok, Reply} ->
        do_something(Reply);
    {error, Error} ->
        uh_oh(Error)
end.
```



Erlang Immutability



- Erlang assignment is pattern matching, not mutation
- Unbound variables get the value of the right-hand side and then can't be changed



Erlang Immutability

```
foo() ->  
    A = 2,    % A is bound to 2
```



Erlang Immutability

```
foo() ->
```

```
  A = 2,    % A is bound to 2
```

```
  A = 2,    % pattern match A to 2, result is 2
```



Erlang Immutability

```
foo() ->
```

```
  A = 2,      % A is bound to 2
```

```
  A = 2,      % pattern match A to 2, result is 2
```

```
  A = 3.      % pattern match A to 3, throw badmatch
```



Erlang Immutability

```
foo() ->
```

```
  A = 2,    % A is bound to 2
```

```
  A = 2,    % pattern match A to 2, result is 2
```

```
  A = 3.    % pattern match A to 3, throw badmatch
```



Easy To Learn



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



RIAK



Riak

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



Riak

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



Riak Architecture

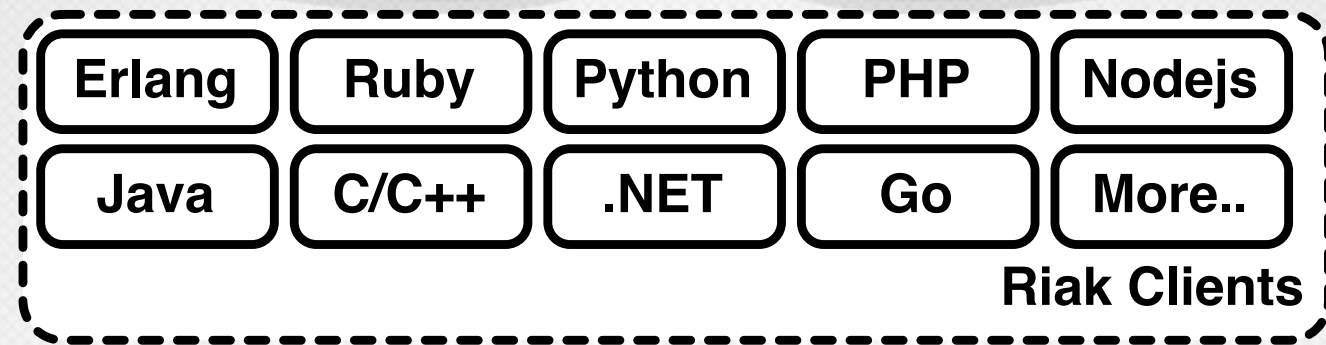


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/



Riak Architecture

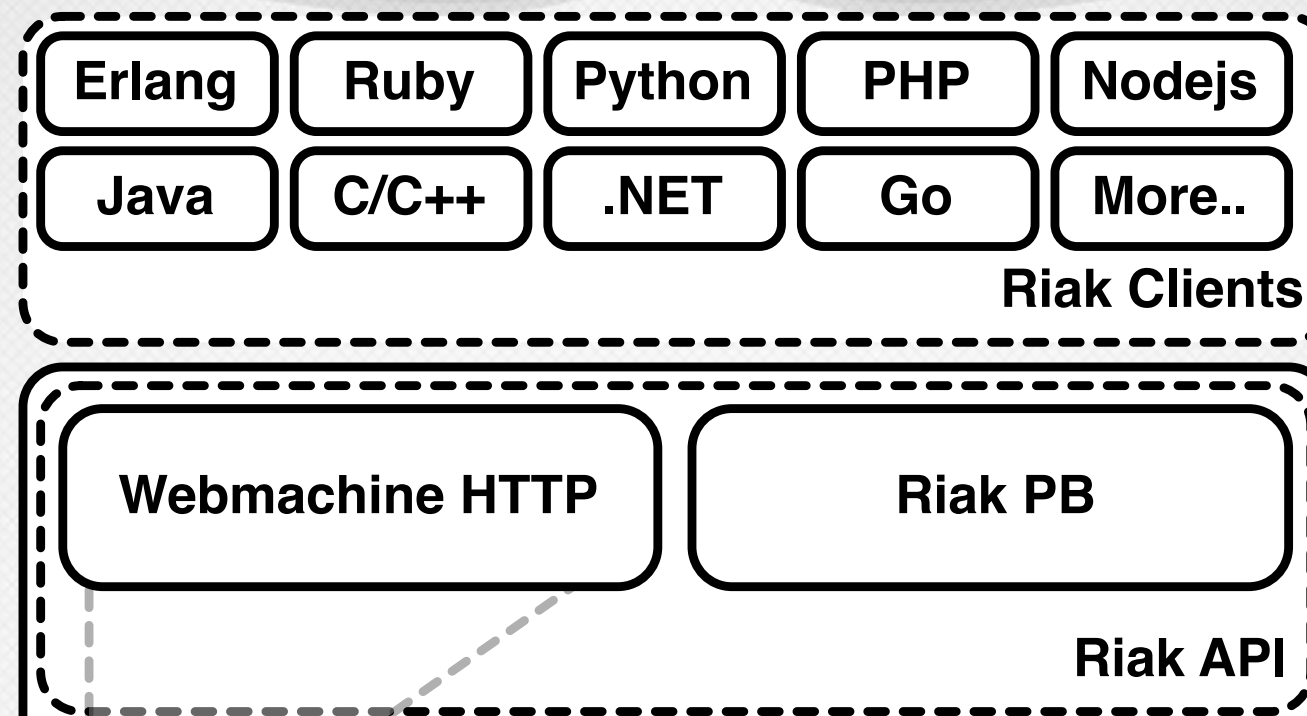


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/



Riak Architecture

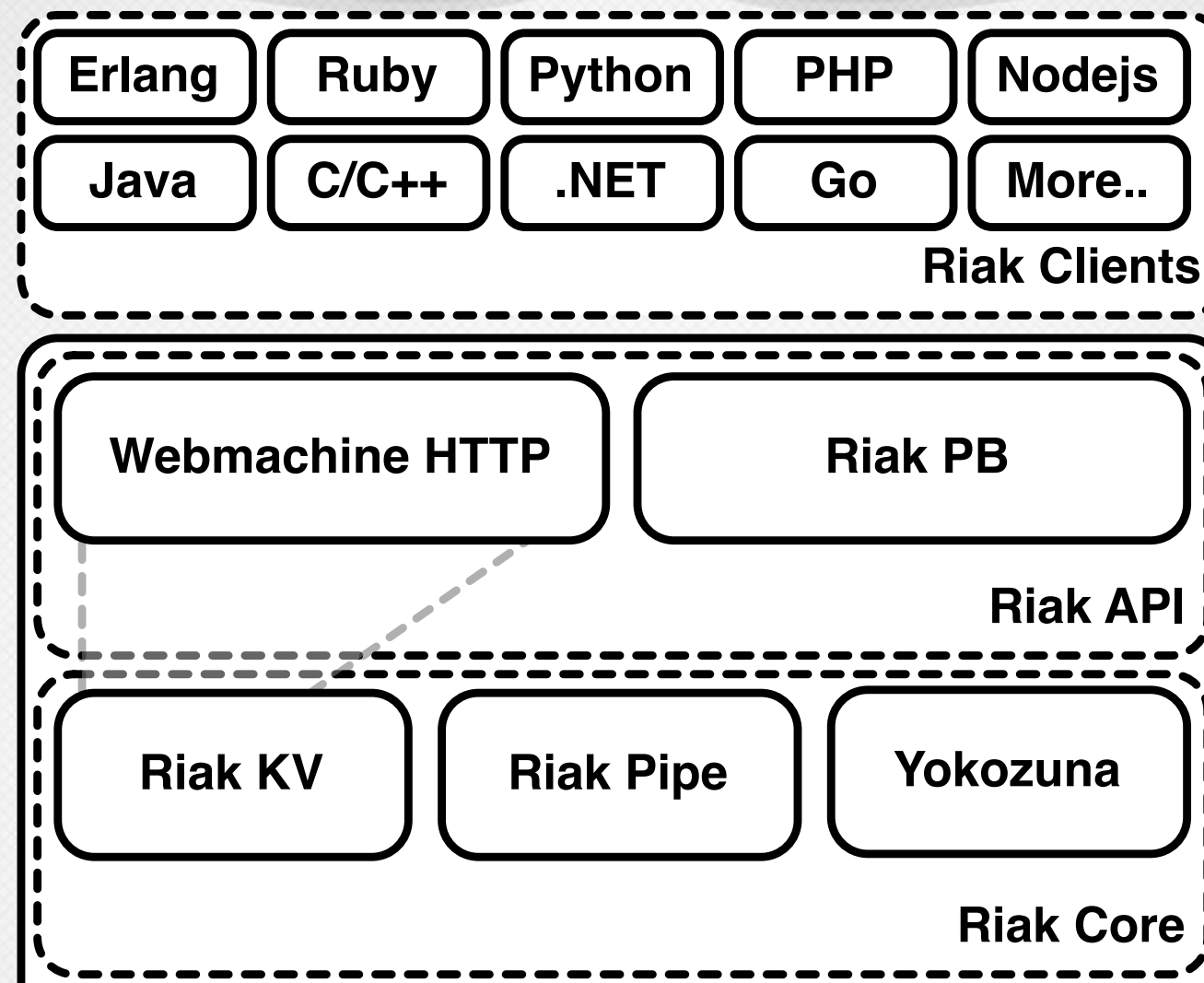


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/



Riak Architecture

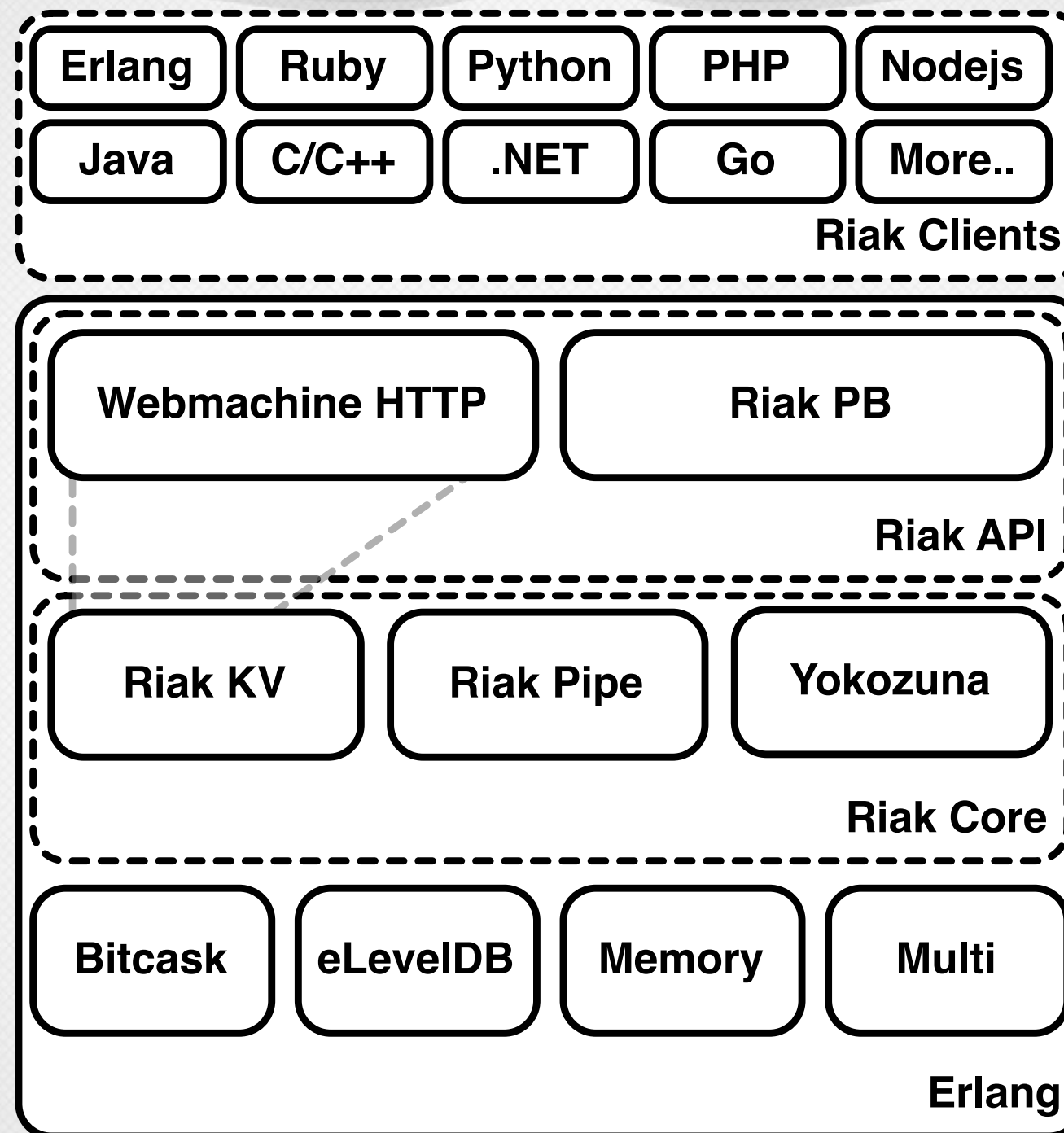
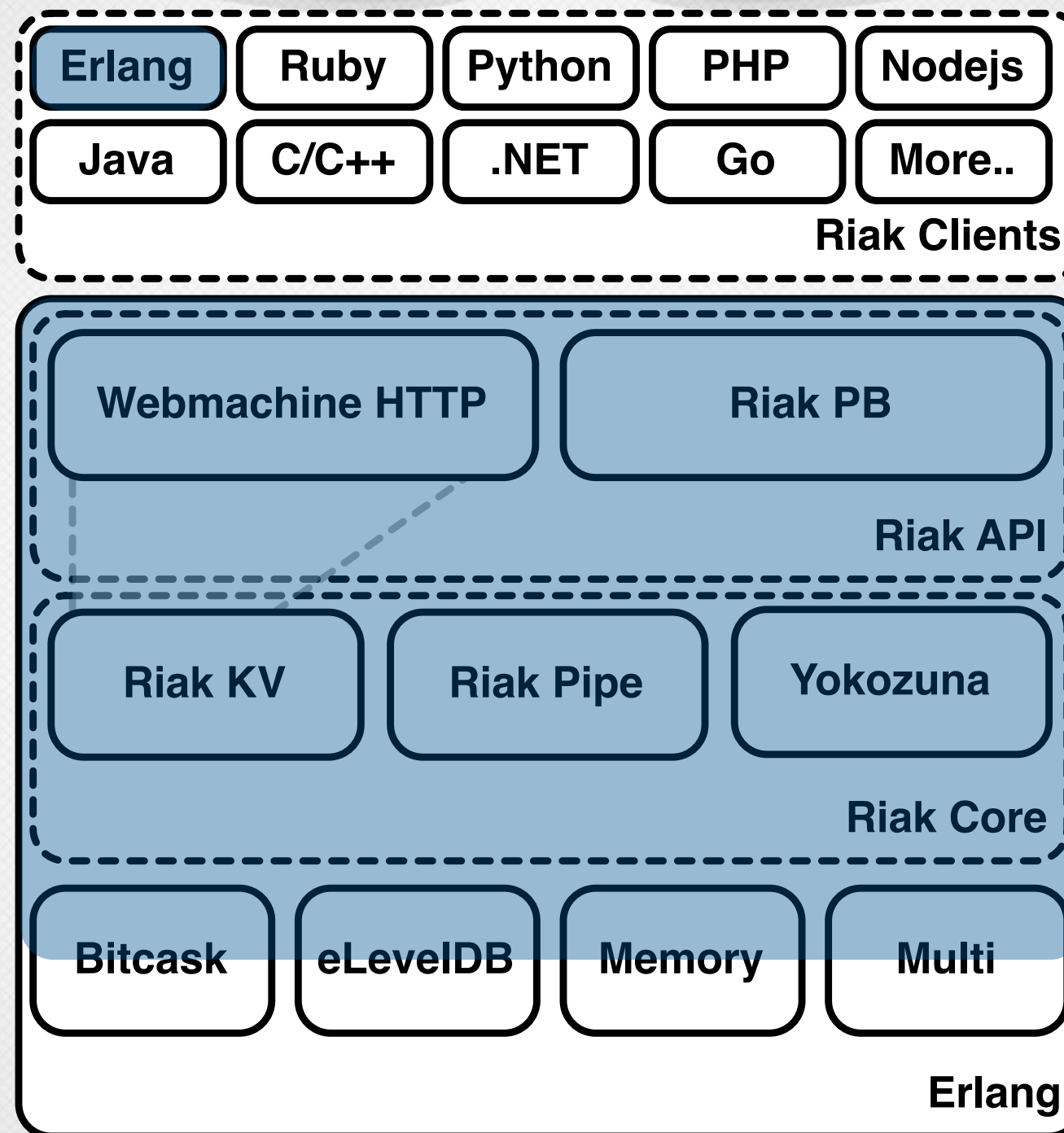


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/



Riak Architecture



● Erlang parts

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



Riak Cluster

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

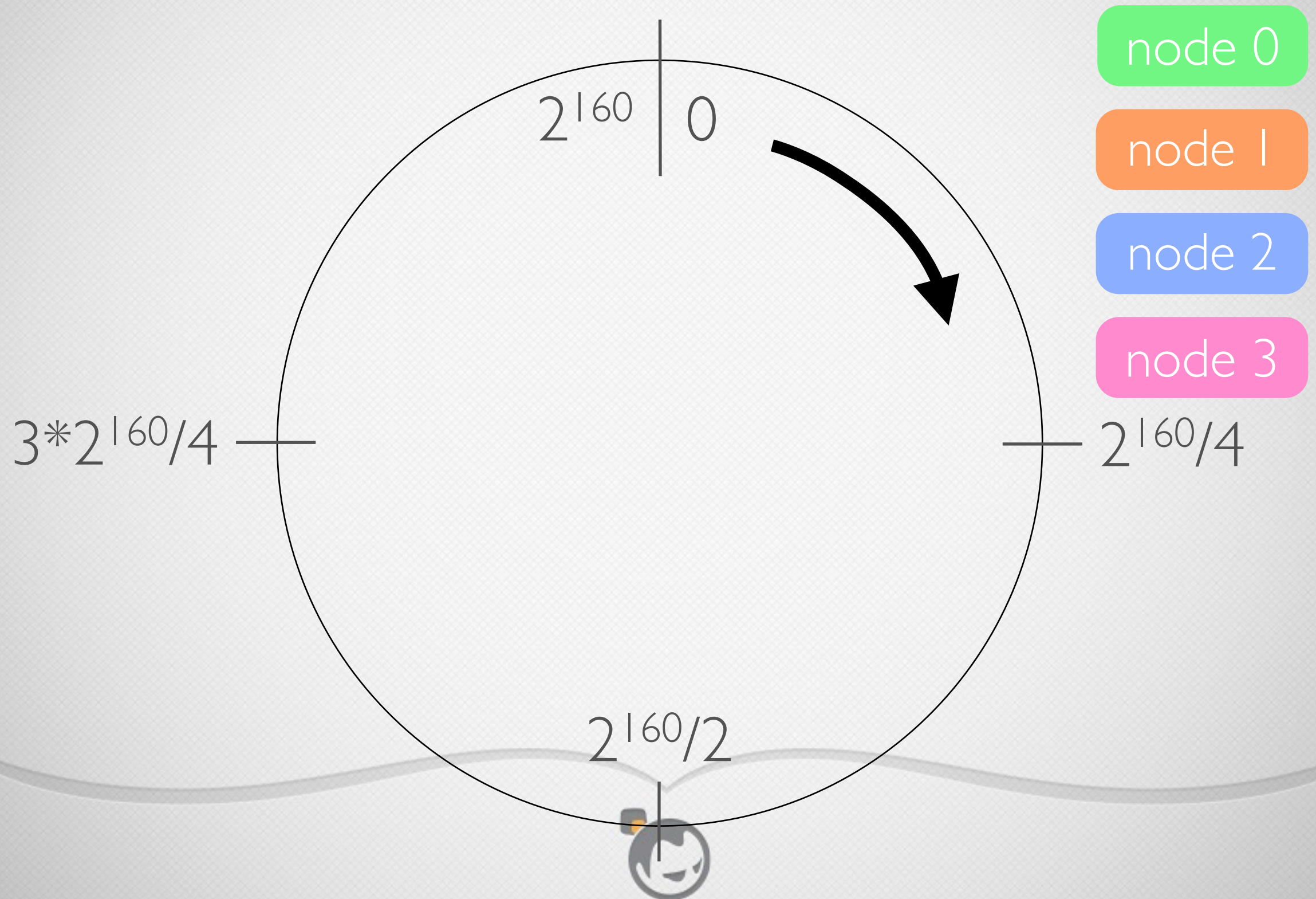
node 1

node 2

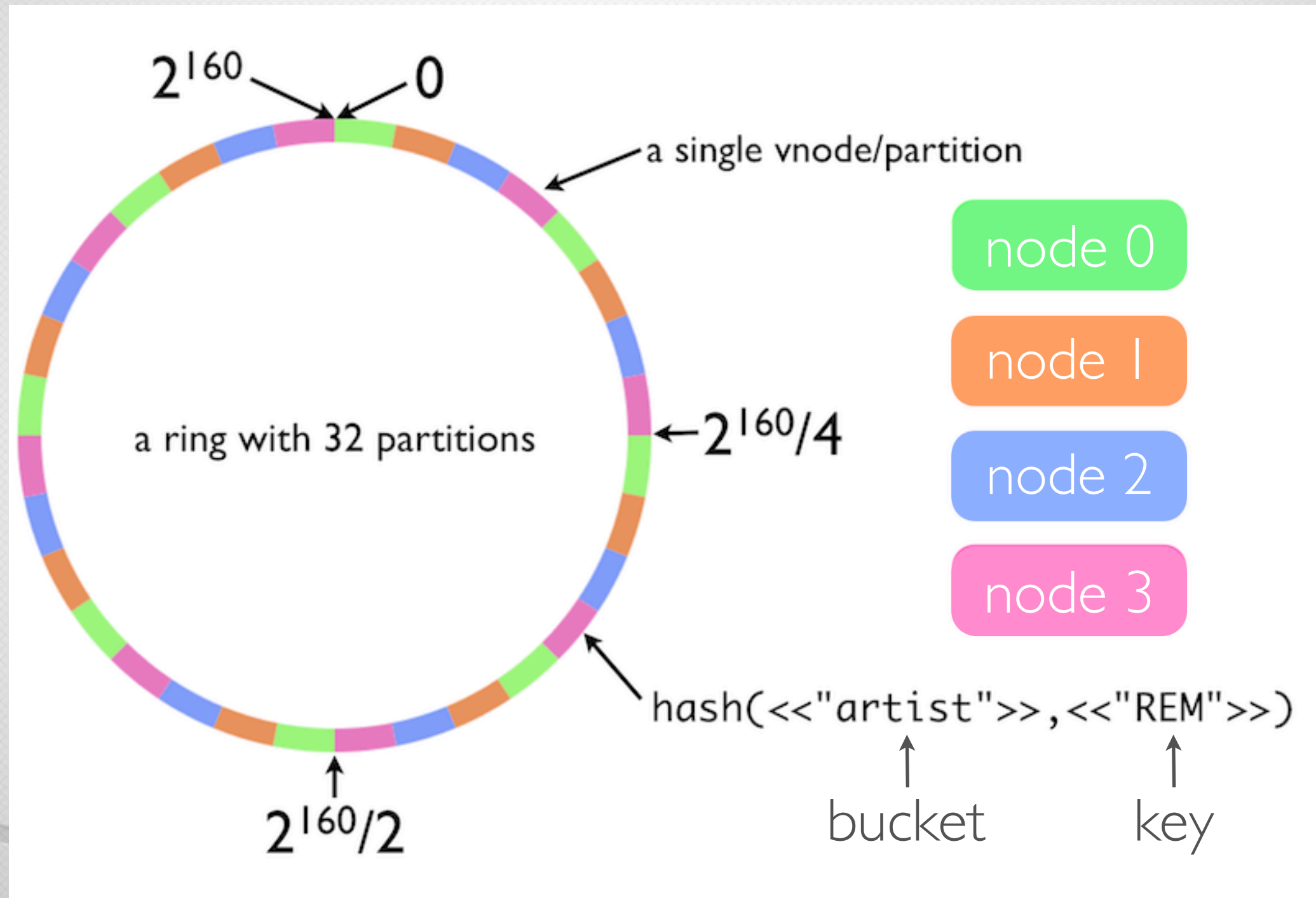
node 3



Hash Ring



Hash Ring

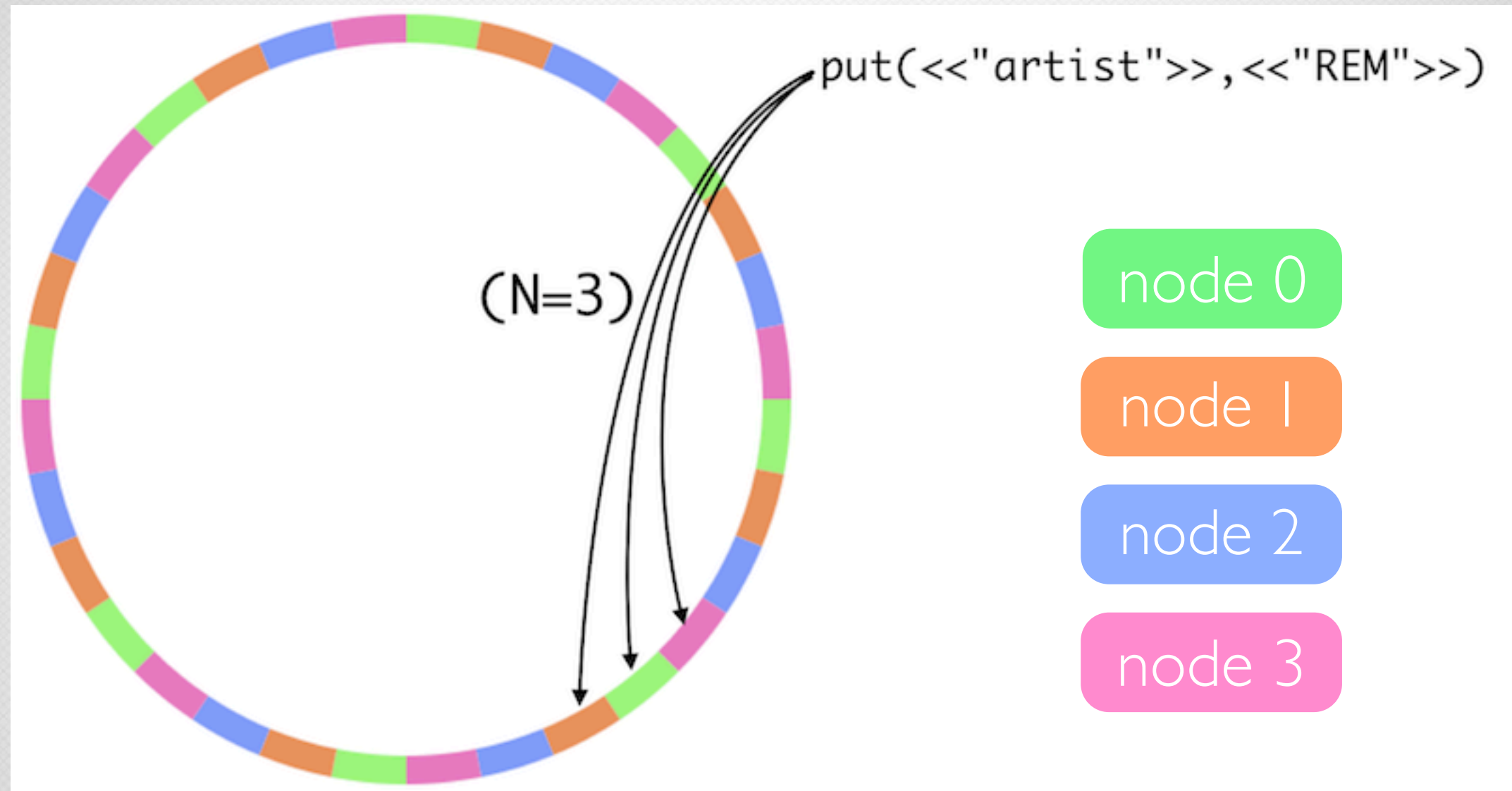


N/R/W Values

- 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)



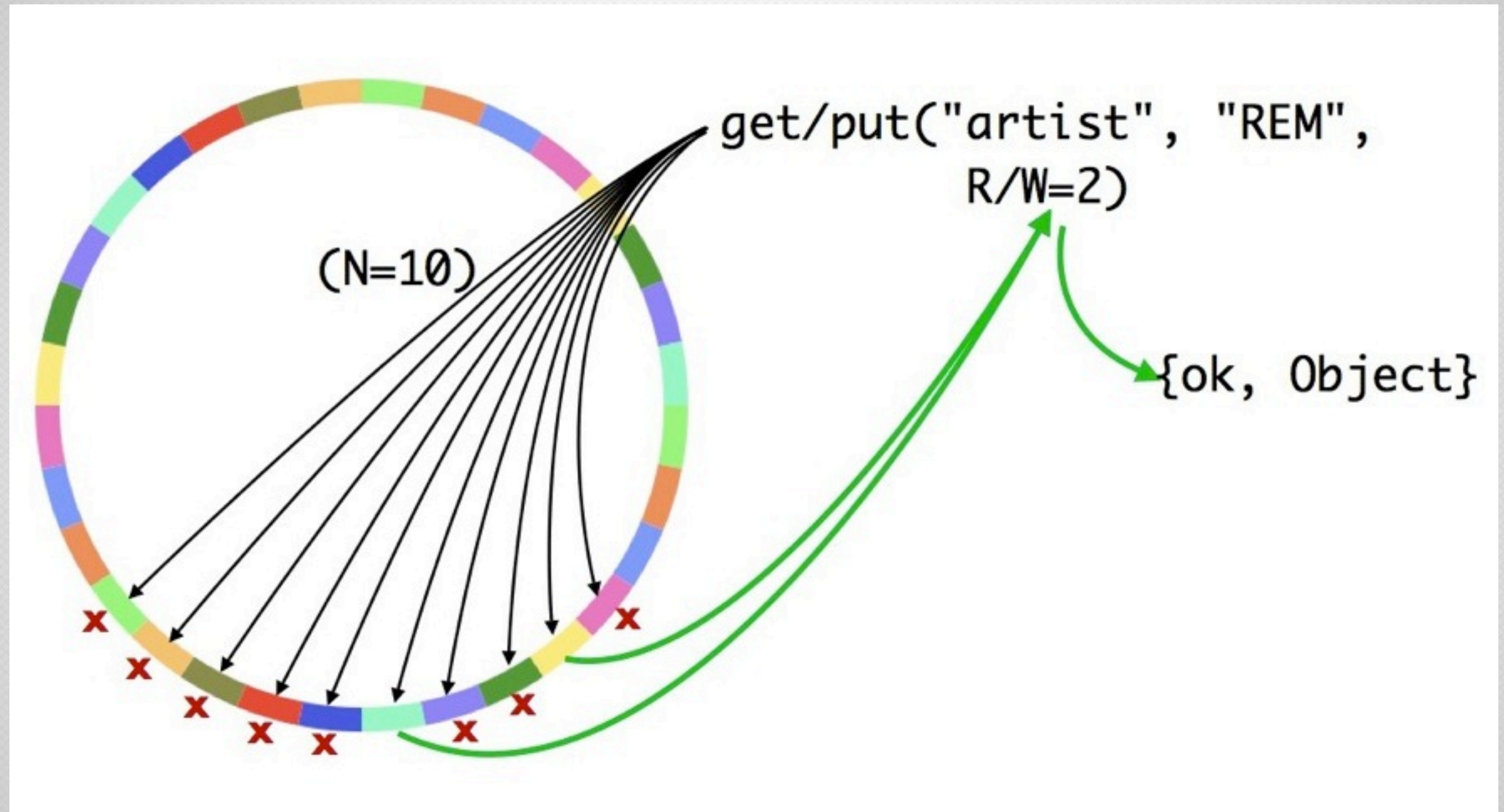
N/R/W Values



for details see <http://docs.basho.com/riak/1.3.1/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 that's OK, Erlang's integers are infinite precision
- And Erlang binaries store these large values efficiently



Implementing Consistent Hashing

```
1> HashBin = crypto:sha("my object key").
```



Implementing Consistent Hashing

```
1> HashBin = crypto:sha("my object key").  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>
```



Implementing Consistent Hashing

```
1> HashBin = crypto:sha("my object key").  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>  
2> byte_size(HashBin).  
20
```



Implementing Consistent Hashing

```
1> HashBin = crypto:sha("my object key").  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>  
2> byte_size(HashBin).  
20  
3> <<HashInt:160/integer>> = HashBin.  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>
```



Implementing Consistent Hashing

```
1> HashBin = crypto:sha("my object key").  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>  
2> byte_size(HashBin).  
20  
3> <<HashInt:160/integer>> = HashBin.  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>  
4> HashInt.  
1080638148638140855100958270058021626367330918047
```



Implementing Consistent Hashing

```
1> HashBin = crypto:sha("my object key").  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
157,242,158,159>>  
2> byte_size(HashBin).  
20  
3> <<HashInt:160/integer>> = HashBin.  
<<189,73,125,145,132,154,3,75,50,12,195,156,7,170,128,52,  
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'},
          {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](https://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

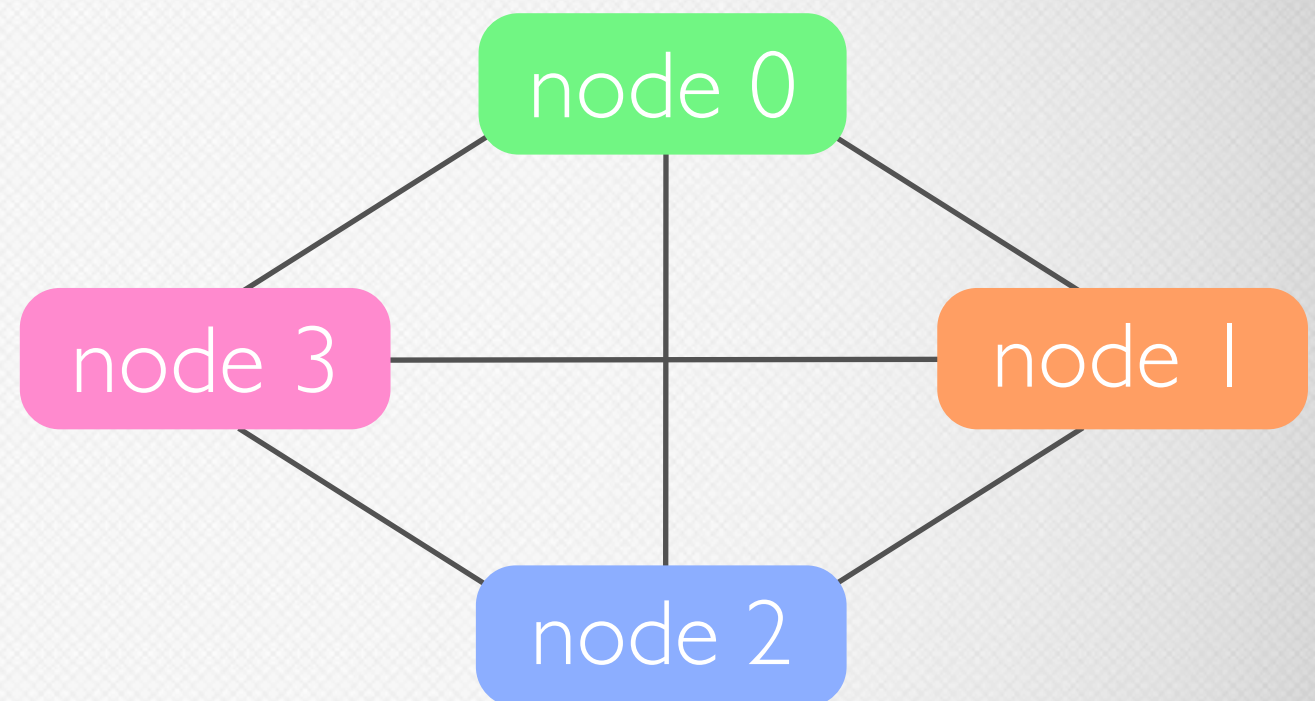
```
$ erl -name dev4@127.0.0.1 -setcookie riak
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]
[async-threads:0] [kernel-poll:false]

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
- But communication overhead means it doesn't scale to large clusters > 150 nodes (yet)



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

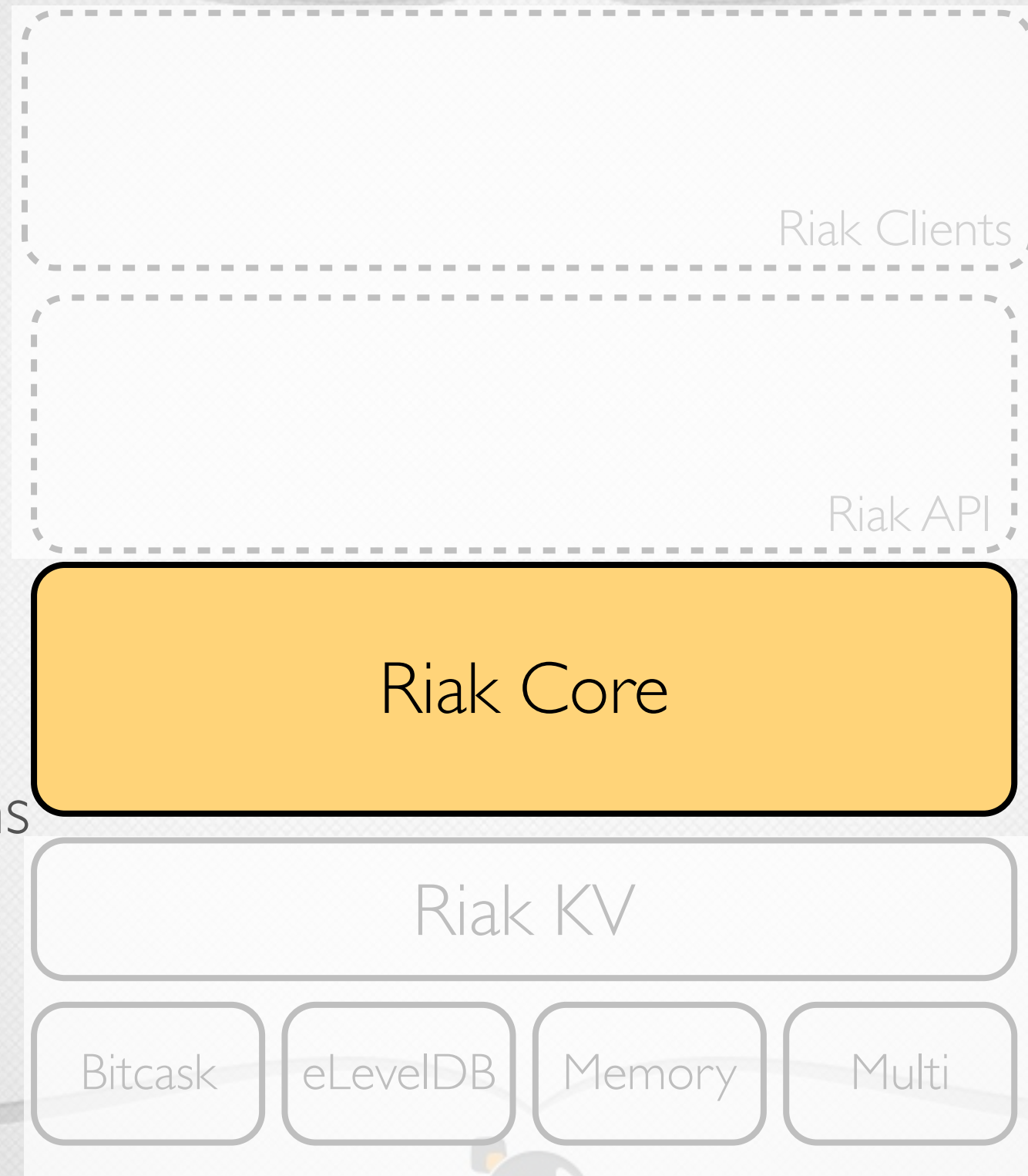


Control Vs. Data

- Distributed Erlang: good for control plane, not so good for data plane
- Sending large data can cause busy distribution ports and head-of-line blocking
- Use TCP, UDP, etc. directly for data plane traffic
- Don't mix control plane and data plane traffic
 - unfortunately Riak currently still does this in a few places



Riak Core

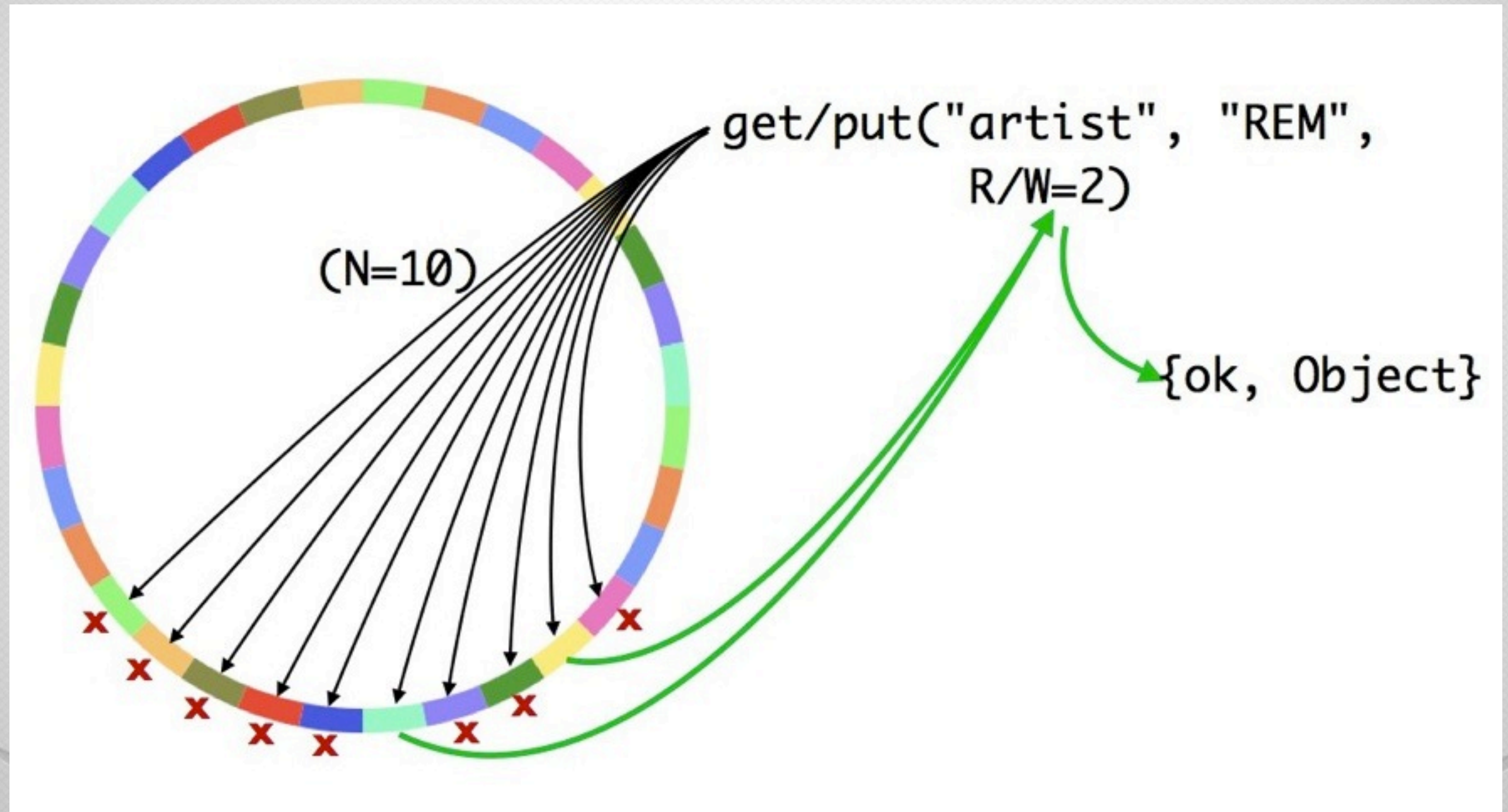


- consistent hashing
- vector clocks
- sloppy quorums

- 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



Binary Handling

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

TCP Header																																		
Offsets Octet		0								1								2								3								
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
0	0	Source port																Destination port																
4	32	Sequence number																																
8	64	Acknowledgment number (if ACK set)																																
12	96	Data offset	Reserved 0 0 0			N S		C W R E G	E C R E G	U R C K	A C K	P S H	R S T	S S T N	F I N N	Window Size																		
16	128	Checksum																Urgent pointer (if URG set)																

source: http://en.wikipedia.org/wiki/Transmission_Control_Protocol



Binary Handling

TcpBuf.



Binary Handling

TCP header fields



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

TCP data payload



Binary Handling

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



Protocols With OTP

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



OTP Behavior Modules

- A 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



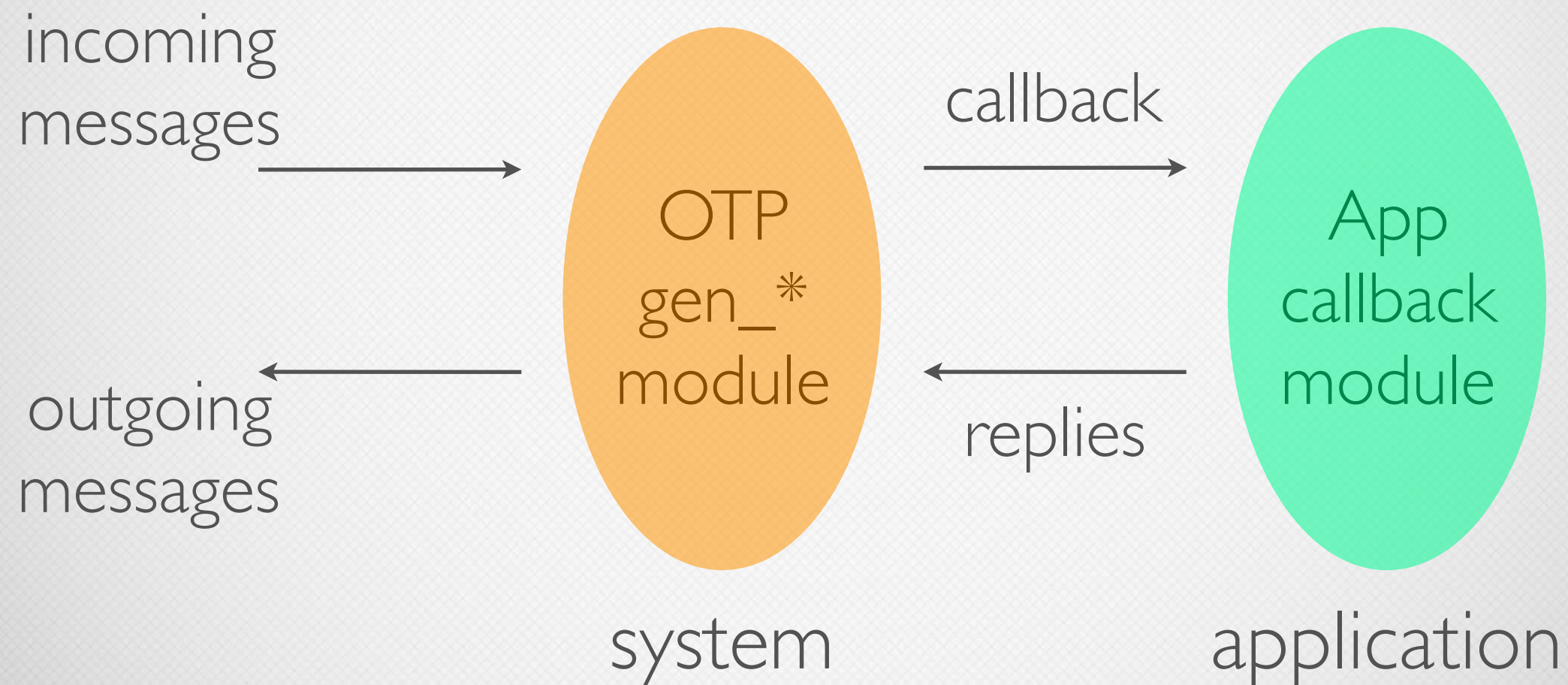
Behavior Benefits

- 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



Behavior Benefits

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



Application Behavior

- 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
- Erlang runtime starts apps in reverse dependency order
- A running Riak system comprises about 30 different 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

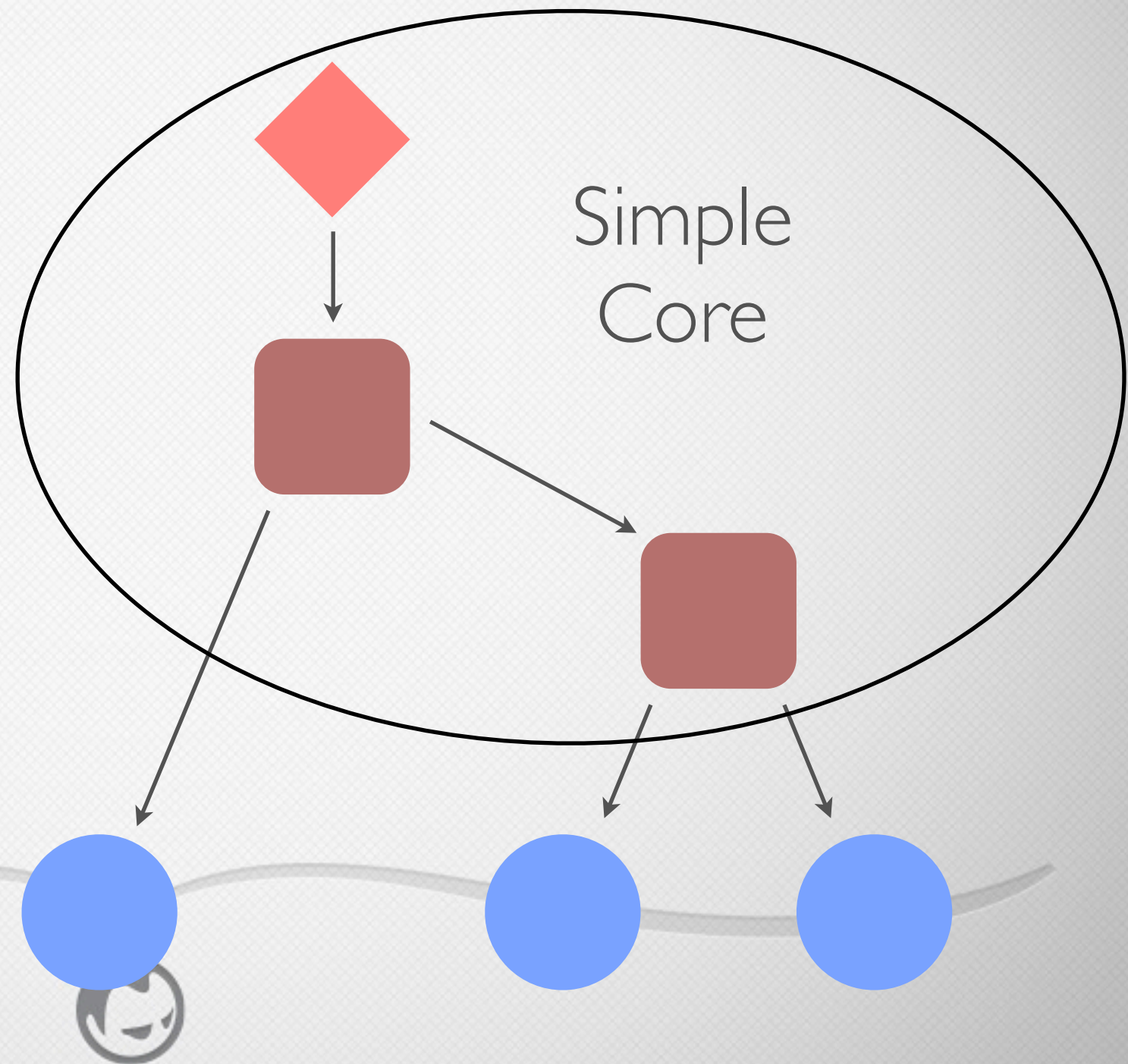


Application, Supervisors, Workers

Application

Supervisors

Workers



Erlang/OTP System Facilities

- Get status of an OTP process
- Get process info for any process
- Trace function calls, messages
- Releases
- Live upgrades



INTEGRATION



Riak Architecture

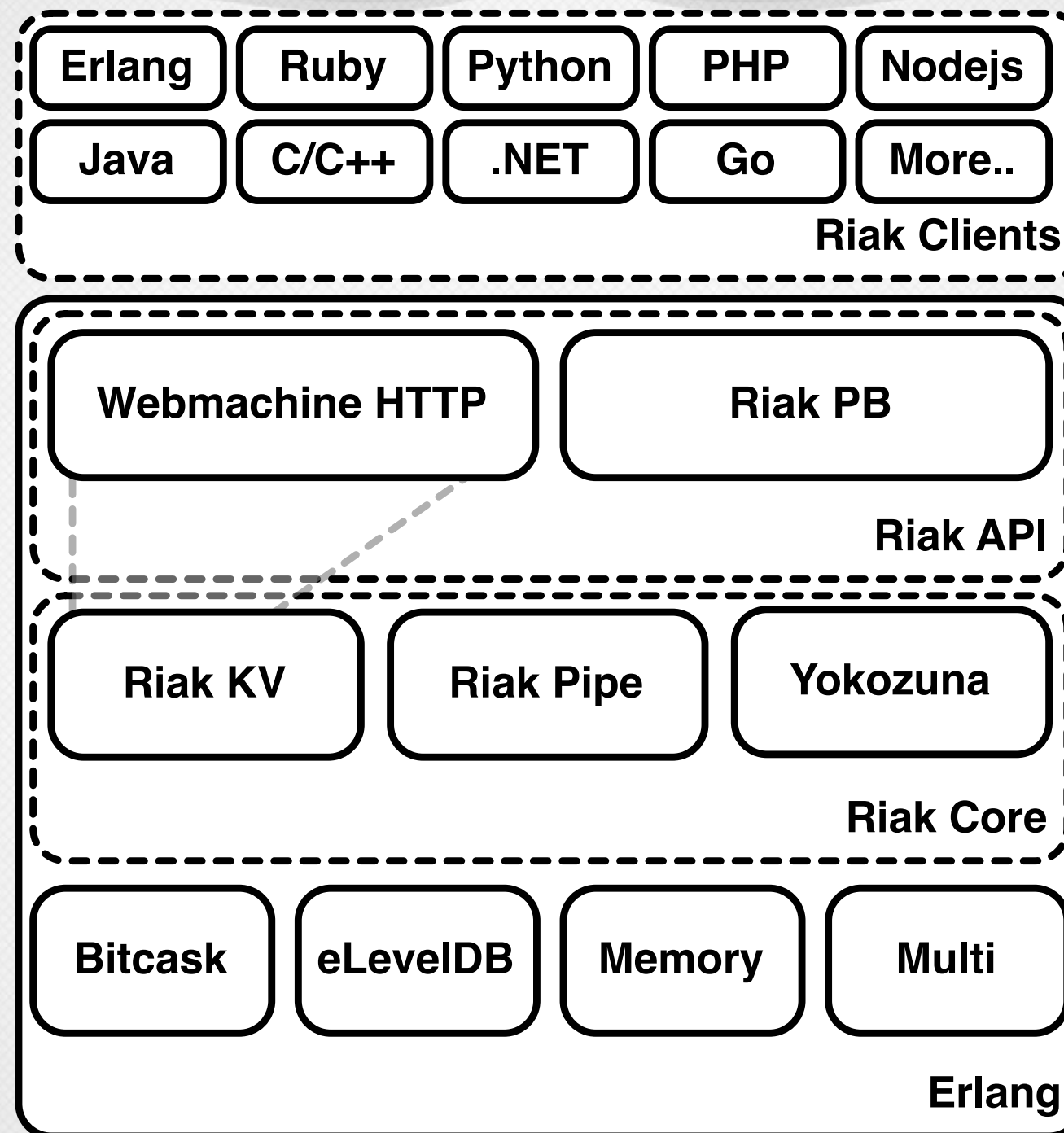
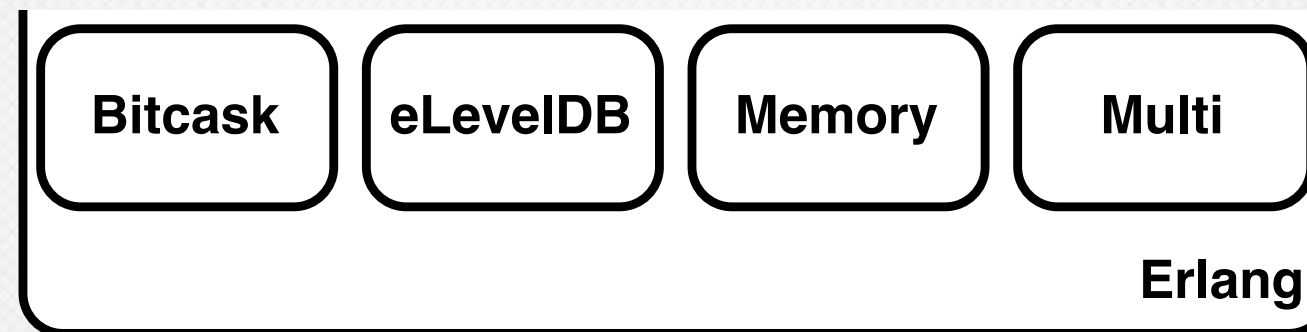


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

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 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

- 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 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

- In 2012 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
- Caused by NIF and BIF calls that were taking multiple seconds in some cases
- Lesson: put long-running activities in their own threads or break up long calls into multiple calls
- Or wait for "dirty" schedulers, targeted for Erlang R17



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



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 standard Erlang build and release tool



MISCELLANEOUS



Miscellaneous

- Memory
- Erlang shell
- Hot code loading
- VM knowledge
- Finding Erlang developers



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 log and notify about looming 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 for customers (with their permission of course)



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



Finding Erlang Devs

- 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

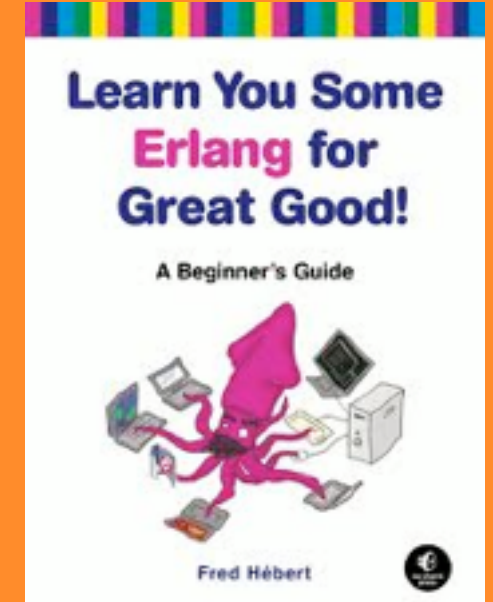
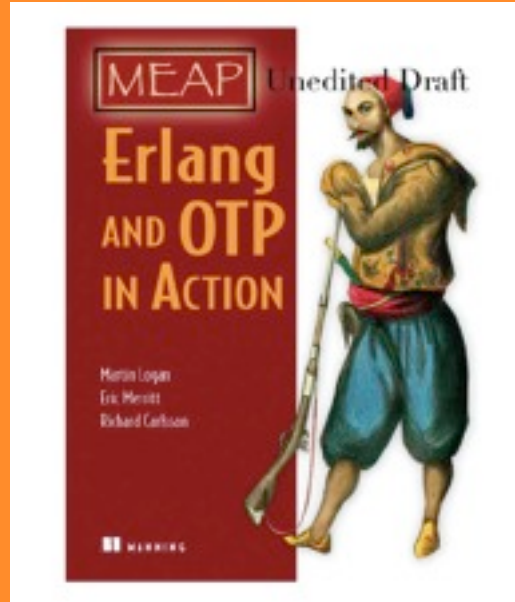


Summary: Why Erlang For Riak?

- Distributed systems features
 - sort of a "distributed systems DSL"
- Concurrency features
- Reliability features
- Runtime introspection capabilities
- Individual developer and team productivity



For More Erlang Info



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

