ADDRESSING NETWORK CONGESTION IN RIAK CLUSTERS

Steve Vinoski

Architecture Group, Basho Technologies

Cambridge, MA USA

http://basho.com

@stevevinoski

vinoski@ieee.org

http://steve.vinoski.net/


COOL TALK, BRO

Riak

Riak

- A distributed

Riak

- A distributed highly available

Riak

- A distributed highly available eventually consistent

Riak

- A distributed highly available eventually consistent highly scalable

Riak

- A distributed highly available eventually consistent highly scalable open source

Riak

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

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
  - see annotated version of Dynamo paper with comparisons to Riak: http://docs.basho.com/riak/latest/references/dynamo/
- Also provides MapReduce, secondary indexes, and full-text search
- Built for operational ease

Riak Architecture

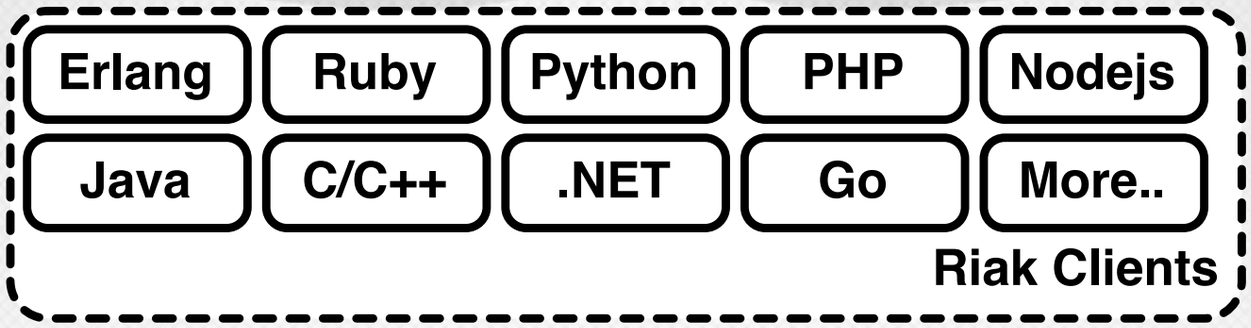


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

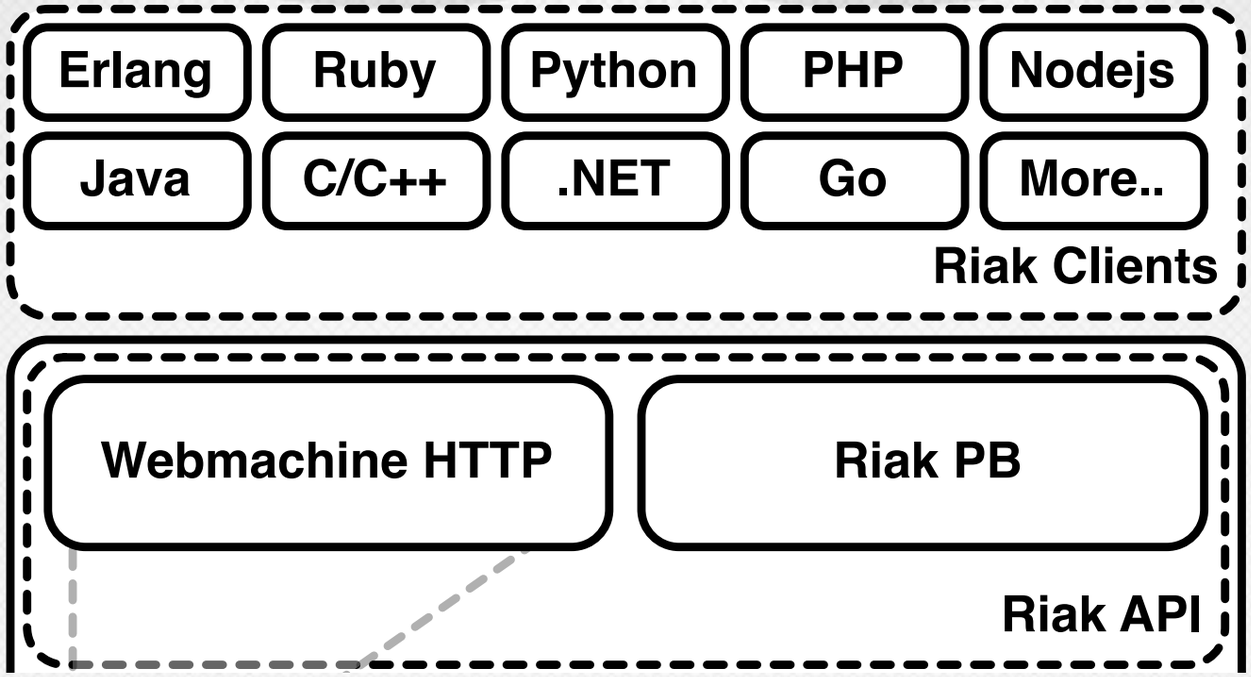


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

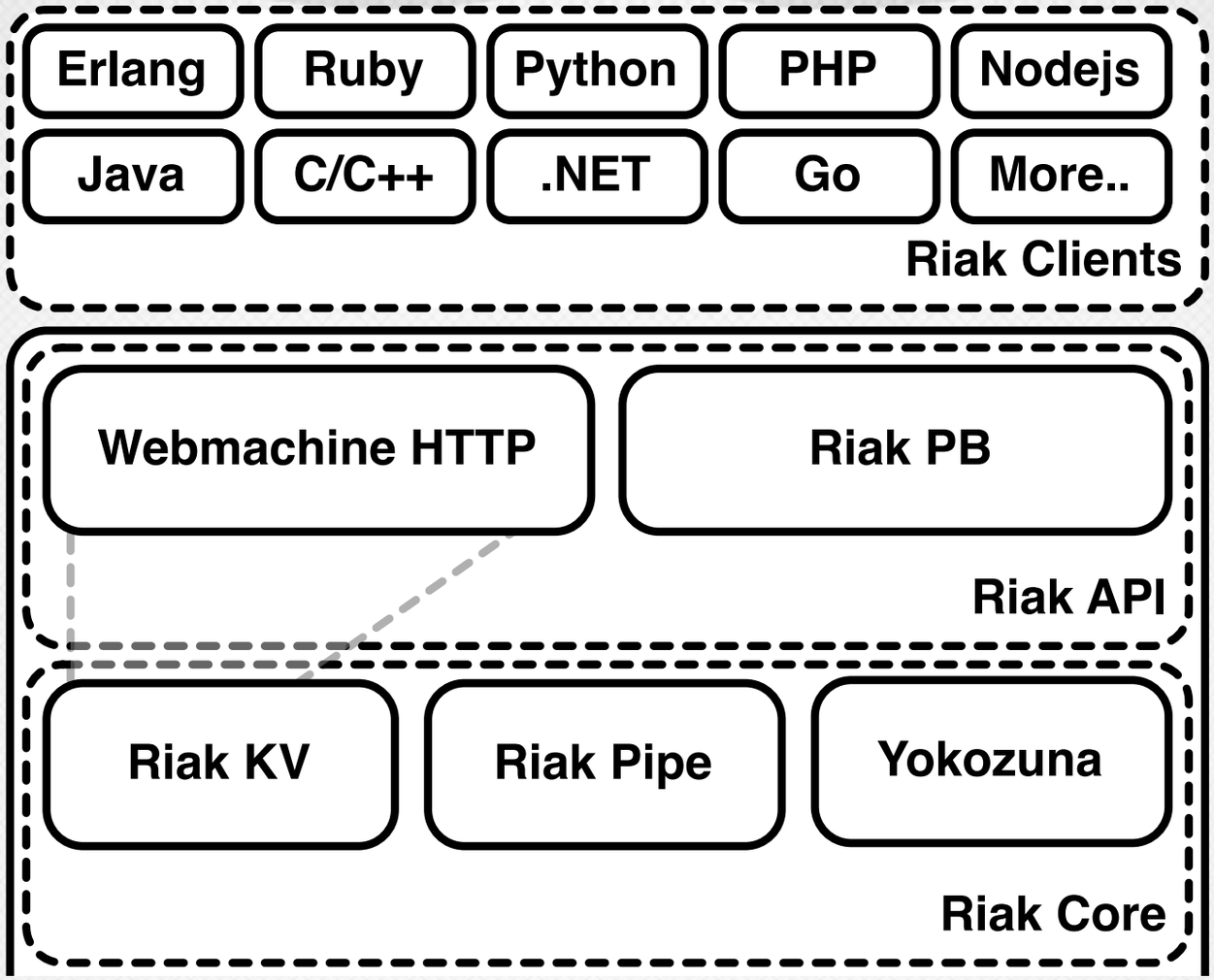


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture

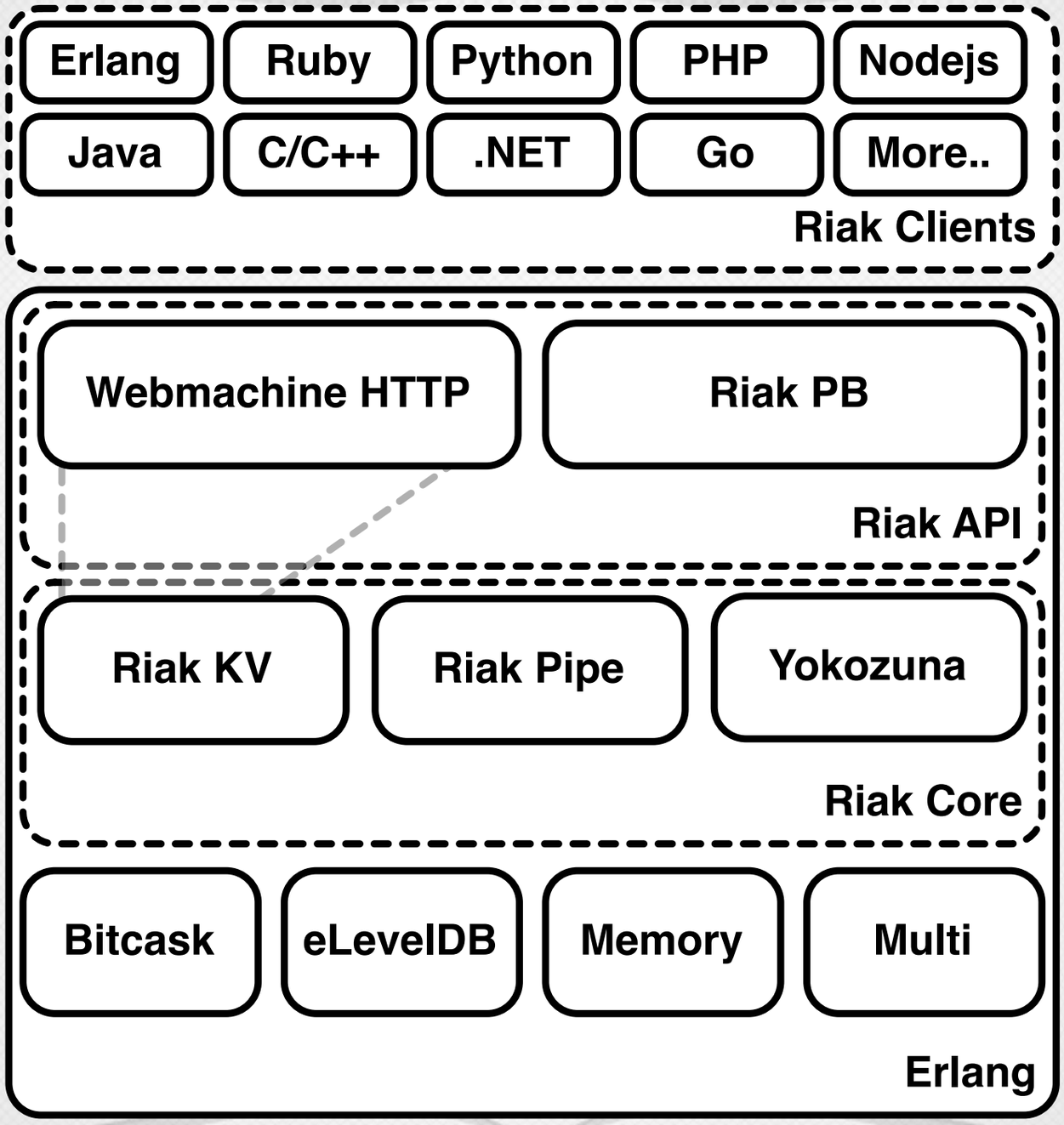
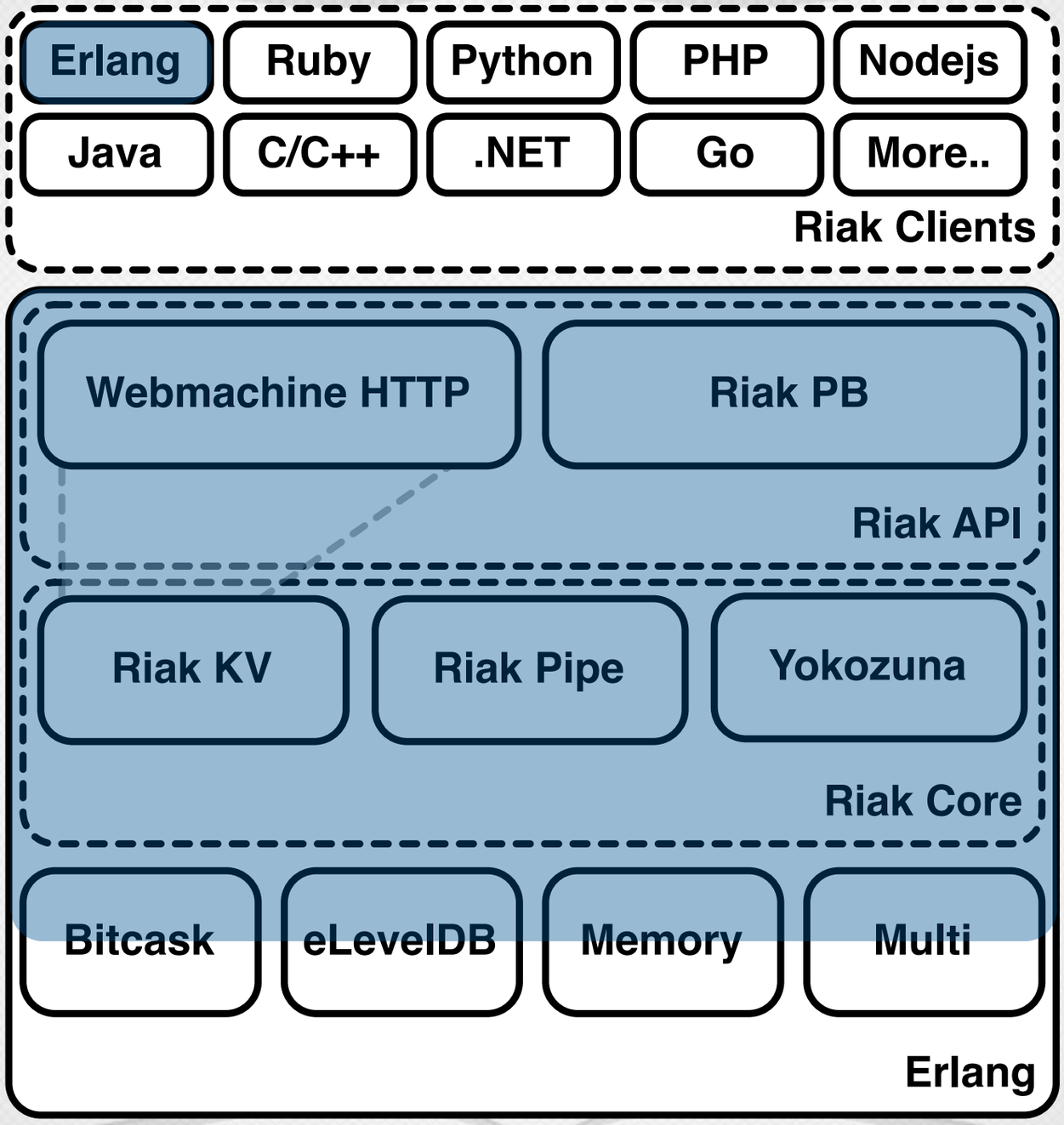


image courtesy of Eric Redmond, "A Little Riak Book" https://github.com/coderoshi/little_riak_book/

Riak Architecture



- Erlang parts

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

Riak Cluster

node 0

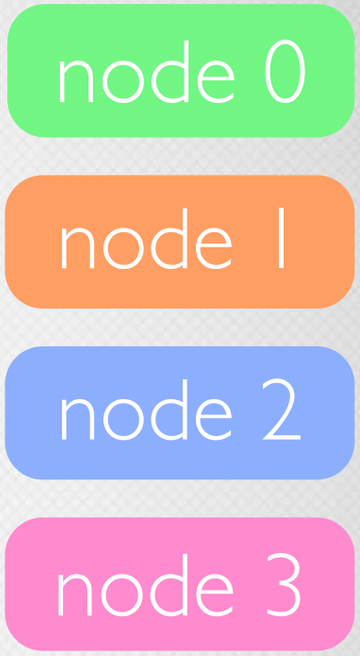
node 3

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



Consistent Hashing

node 0

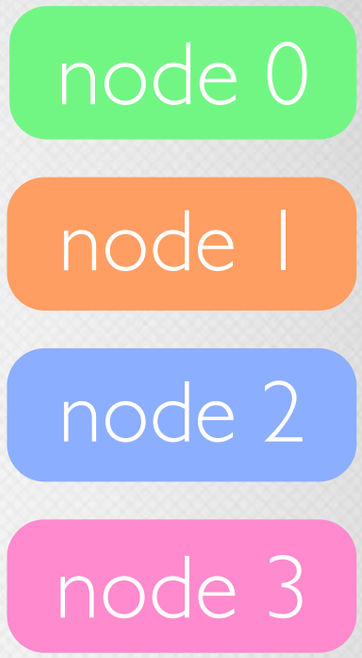
node 1

node 2

node 3

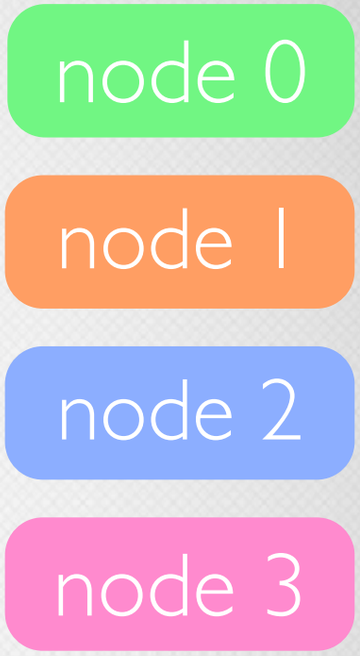
Consistent Hashing

- Riak uses SHA-1 as a hash function



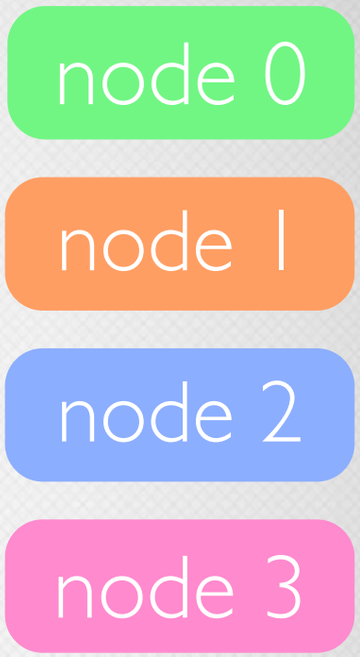
Consistent Hashing

- Riak uses SHA-1 as a hash function
- Treats its 160-bit value space as a ring



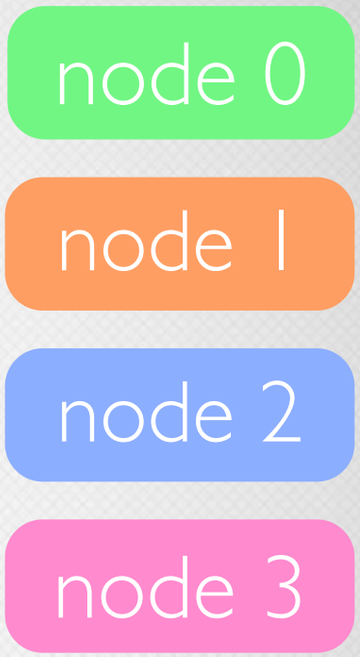
Consistent Hashing

- 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)



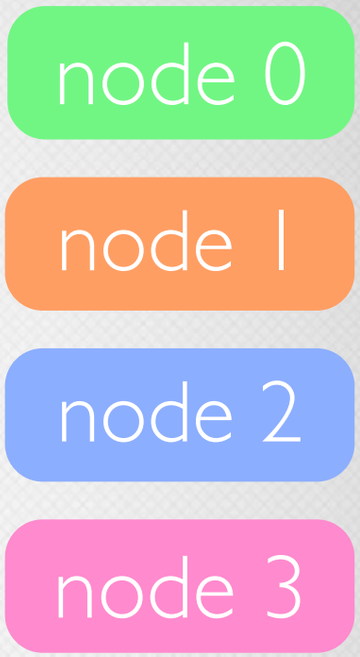
Consistent Hashing

- 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

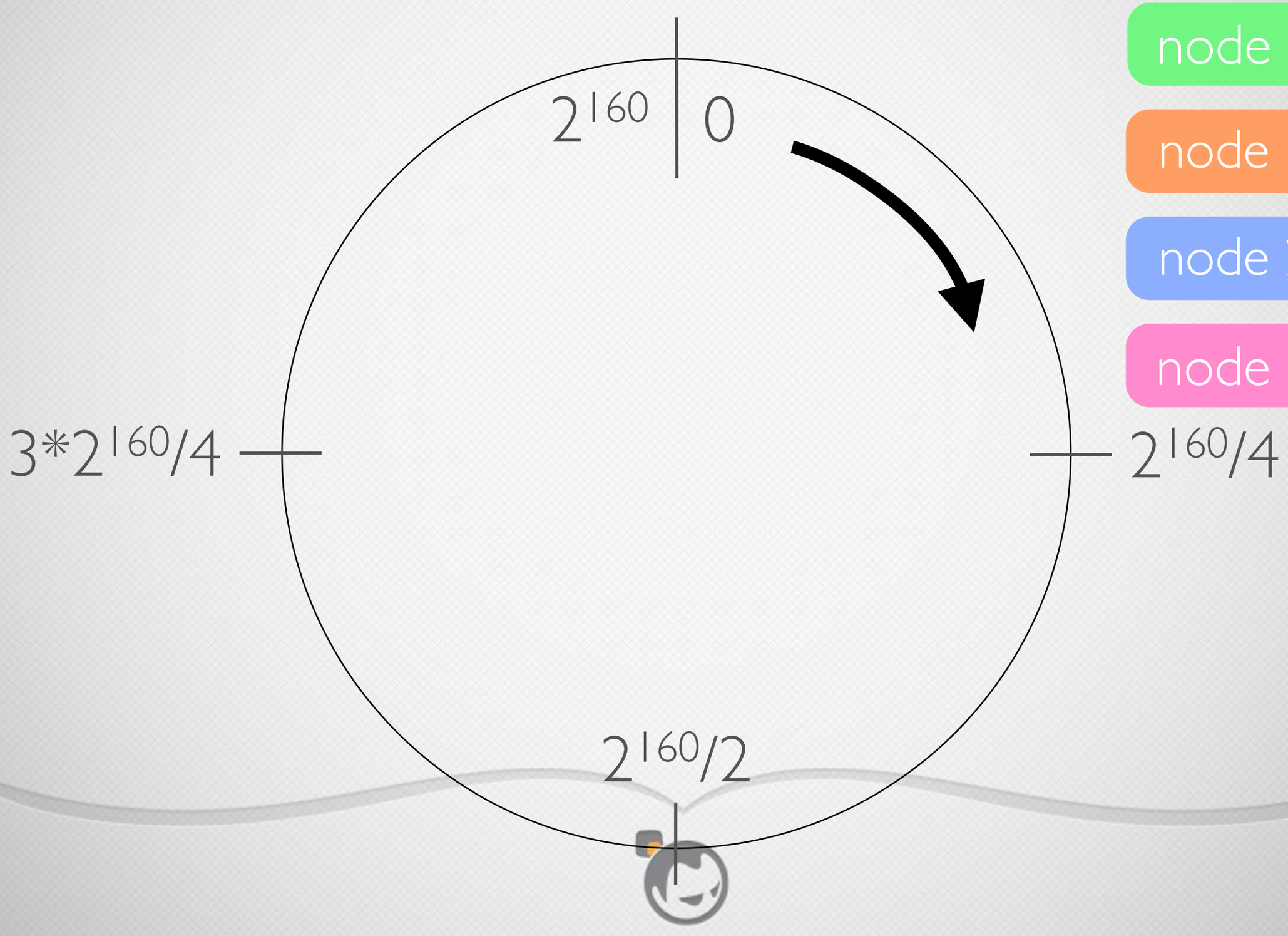


Consistent Hashing

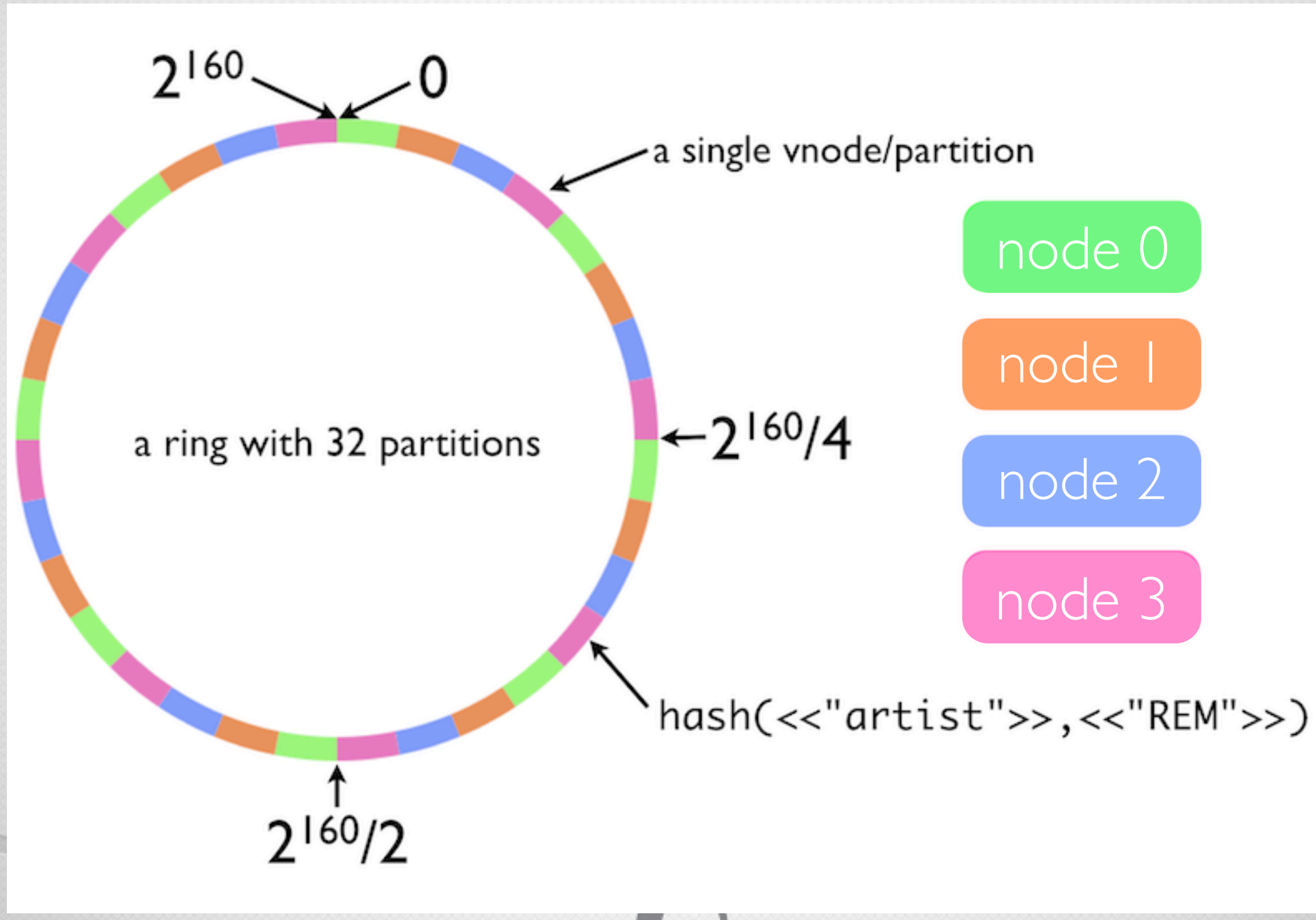
- Riak uses SHA-1 as a hash function
- Treats its 160-bit value space as a ring
- Divides the ring into partitions called "virtual nodes" or vnodes (default 64)
- Each vnode claims a portion of the ring space
- Each physical node in the cluster hosts multiple vnodes



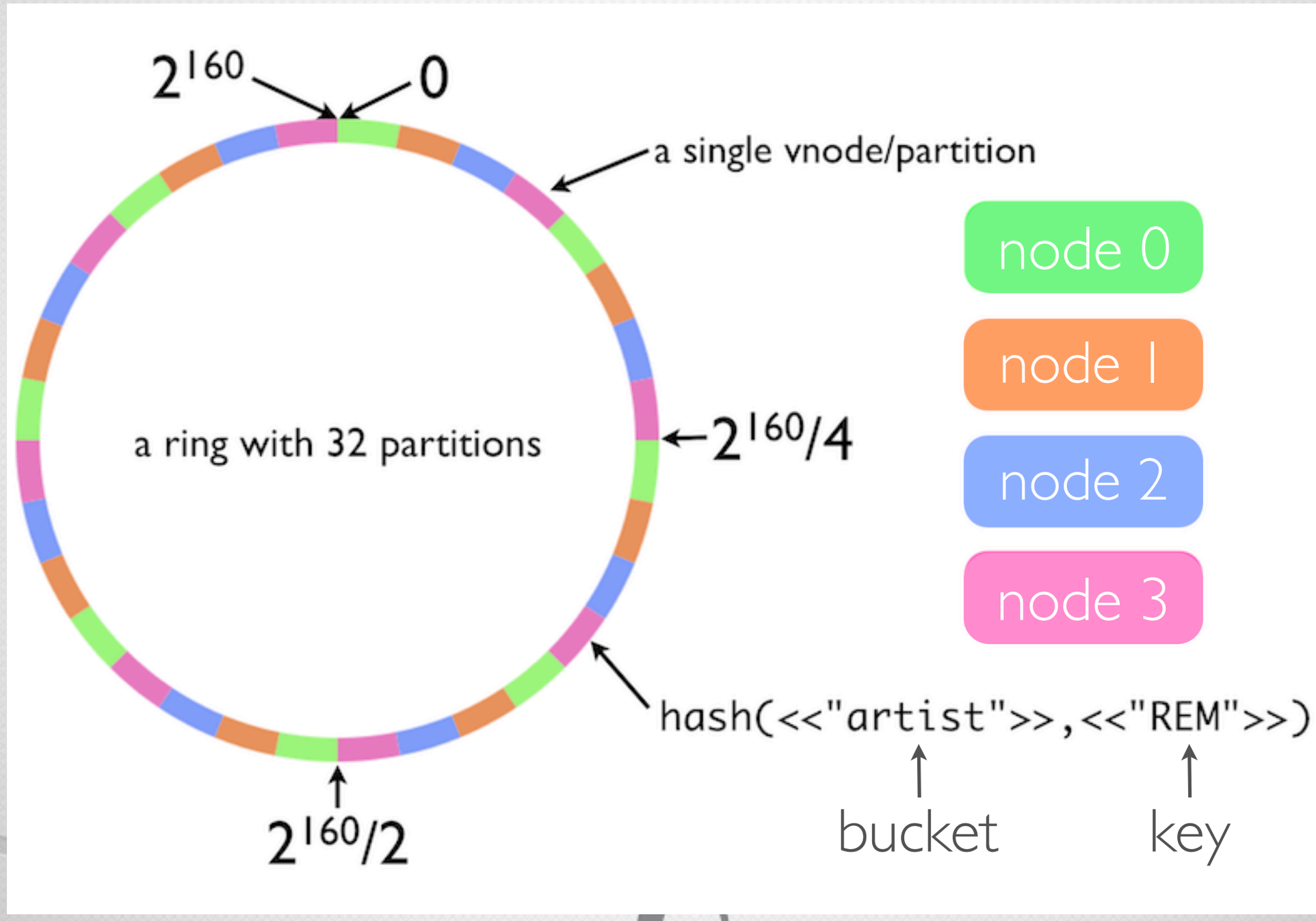
Hash Ring



Hash Ring


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

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/R/W Values

- N = number of replicas to store (default 3)

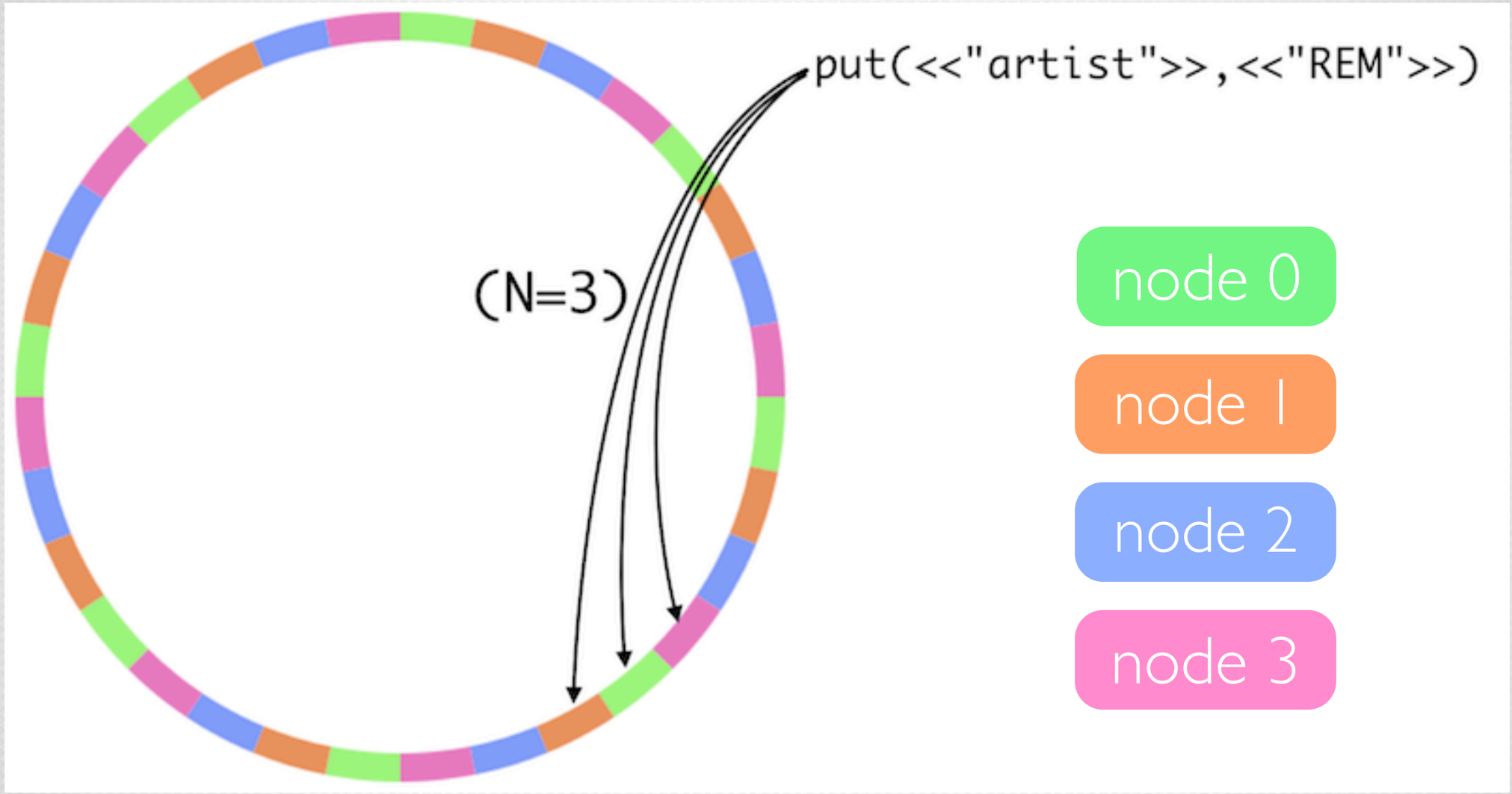
N/R/W Values

- N = number of replicas to store (default 3)
- R = read quorum = number of replica responses needed for a successful read (default N/2+1)

N/R/W Values

- N = number of replicas to store (default 3)
- R = read quorum = number of replica responses needed for a successful read (default N/2+1)
- W = write quorum = number of replica responses needed for a successful write (default N/2+1)

N/R/W Values



Riak TCP Traffic

Riak TCP Traffic

- Client requests: made to any node in the ring

Riak TCP Traffic

- Client requests: made to any node in the ring
- Coordination: node receiving client request coordinates the operation across the owning replicas

Riak TCP Traffic

- Client requests: made to any node in the ring
- Coordination: node receiving client request coordinates the operation across the owning replicas
- Gossip: Riak nodes share ring state via a gossip protocol

Riak TCP Traffic

- Client requests: made to any node in the ring
- Coordination: node receiving client request coordinates the operation across the owning replicas
- Gossip: Riak nodes share ring state via a gossip protocol
- Active Anti-Entropy: nodes actively verify and repair data consistency across the ring (new with Riak 1.3)

Riak TCP Traffic

- Client requests: made to any node in the ring
- Coordination: node receiving client request coordinates the operation across the owning replicas
- Gossip: Riak nodes share ring state via a gossip protocol
- Active Anti-Entropy: nodes actively verify and repair data consistency across the ring (new with Riak 1.3)
- Erlang: distributed Erlang nodes form a full mesh and do periodic node availability checks

Riak TCP Traffic

- Client requests: made to any node in the ring
- Coordination: node receiving client request coordinates the operation across the owning replicas
- Gossip: Riak nodes share ring state via a gossip protocol
- Active Anti-Entropy: nodes actively verify and repair data consistency across the ring (new with Riak 1.3)
- Erlang: distributed Erlang nodes form a full mesh and do periodic node availability checks
- Multi-Data Center Replication: sync data across multiple clusters (part of Riak Enterprise, see http://basho.com/riak-enterprise/)

Riak TCP Traffic

- Client requests: made to any node in the ring
- Coordination: node receiving client request coordinates the operation across the owning replicas
- Gossip: Riak nodes share ring state via a gossip protocol
- Active Anti-Entropy: nodes actively verify and repair data consistency across the ring (new with Riak 1.3)
- Erlang: distributed Erlang nodes form a full mesh and do periodic node availability checks
- Multi-Data Center Replication: sync data across multiple clusters (part of Riak Enterprise, see http://basho.com/riak-enterprise/)
- Handoff

Handoff

Handoff

- If primary vnode is unavailable, request goes to a fallback vnode

Handoff

- If primary vnode is unavailable, request goes to a fallback vnode
- Fallback vnode holds data on behalf of the unavailable primary

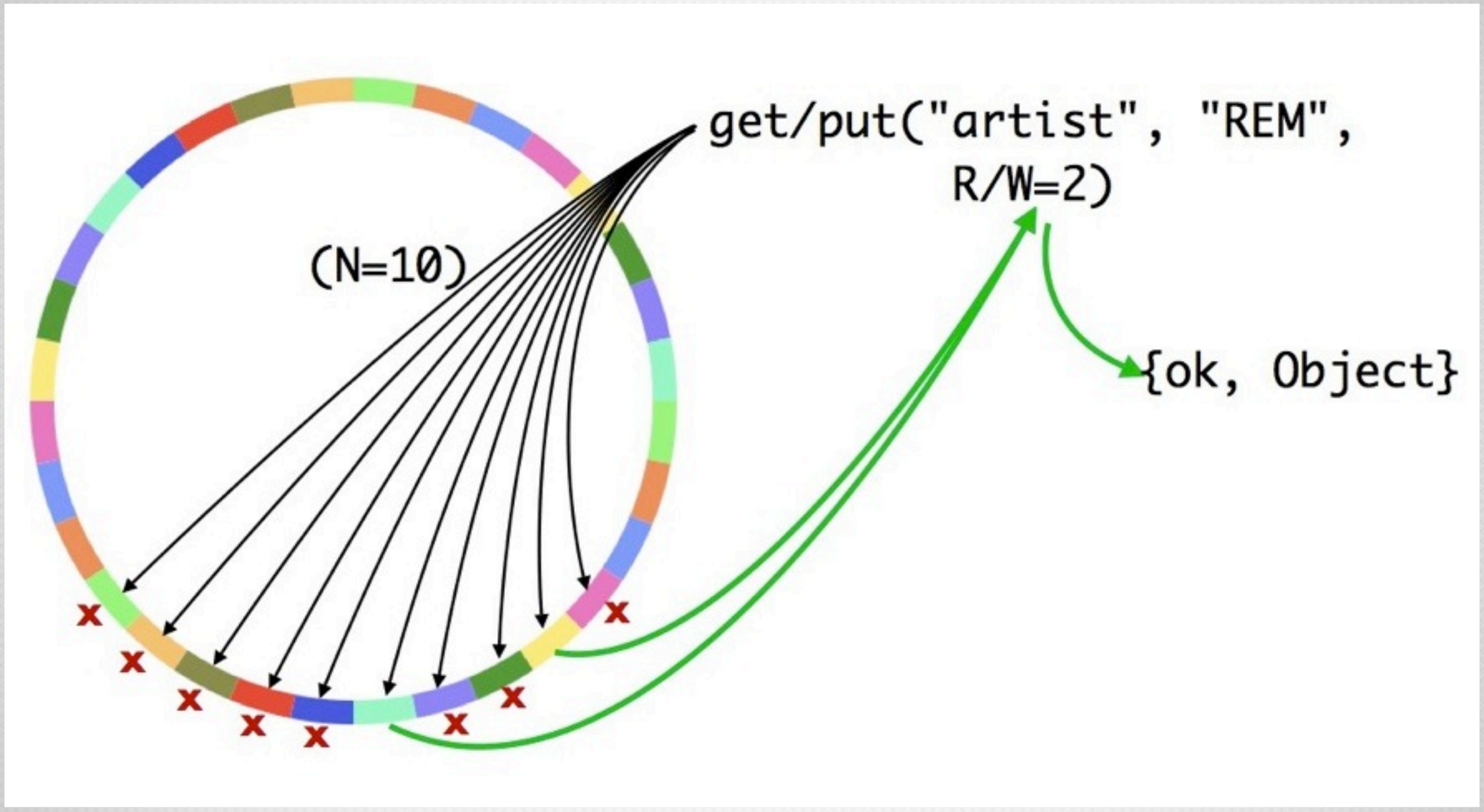
Handoff

- If primary vnode is unavailable, request goes to a fallback vnode
- Fallback vnode holds data on behalf of the unavailable primary
- Fallback vnode watches for return of primary vnode

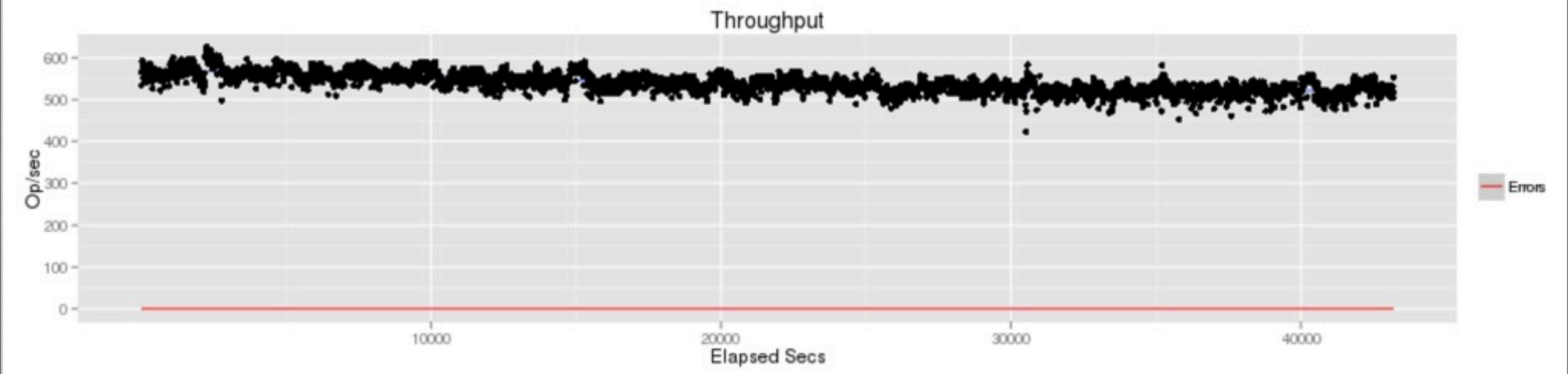
Handoff

- If primary vnode is unavailable, request goes to a fallback vnode
- Fallback vnode holds data on behalf of the unavailable primary
- Fallback vnode watches for return of primary vnode
- When the primary returns, the fallback performs a handoff to transfer data to it

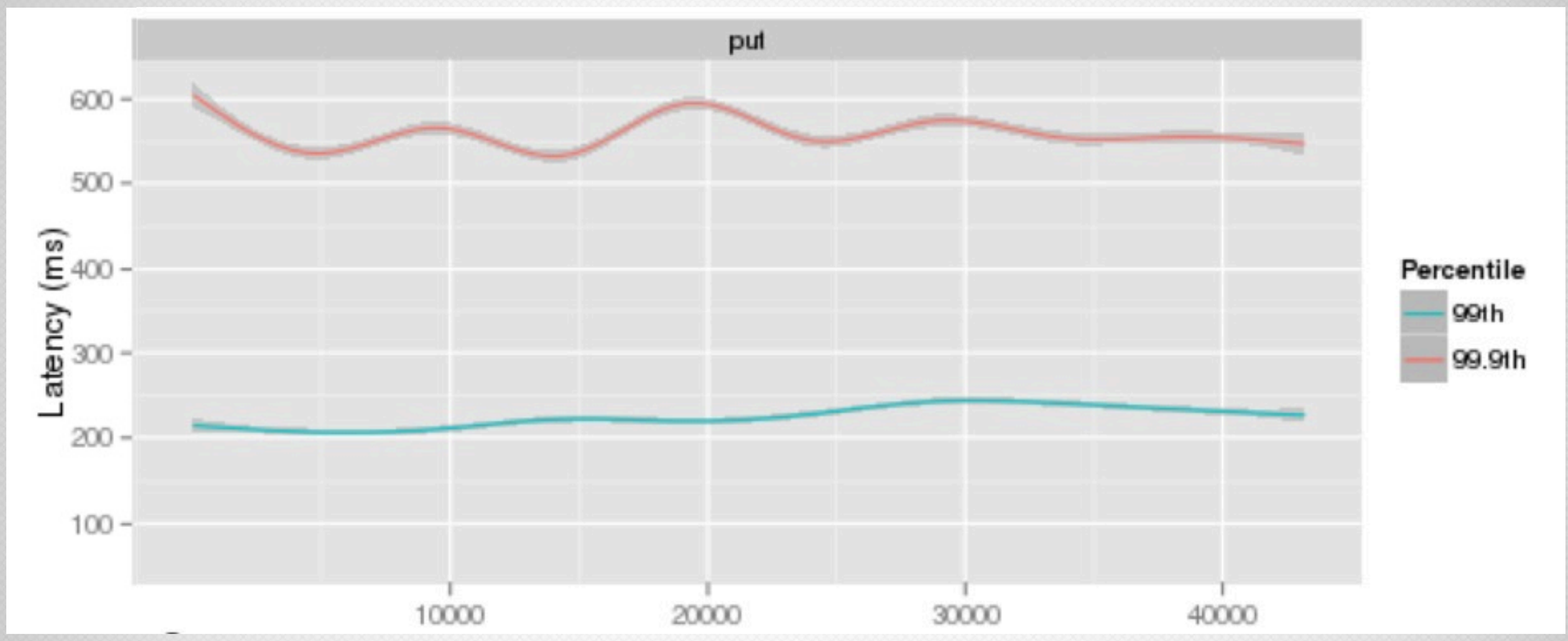
N/R/W Values



Cluster Throughput Under Extreme Load



Latency Of Puts Under Extreme Load



Let's Scale

Let's Scale

- Scaling up/down in Riak means adding/removing nodes

Let's Scale

- Scaling up/down in Riak means adding/removing nodes
- Adding: new nodes claim ring partitions

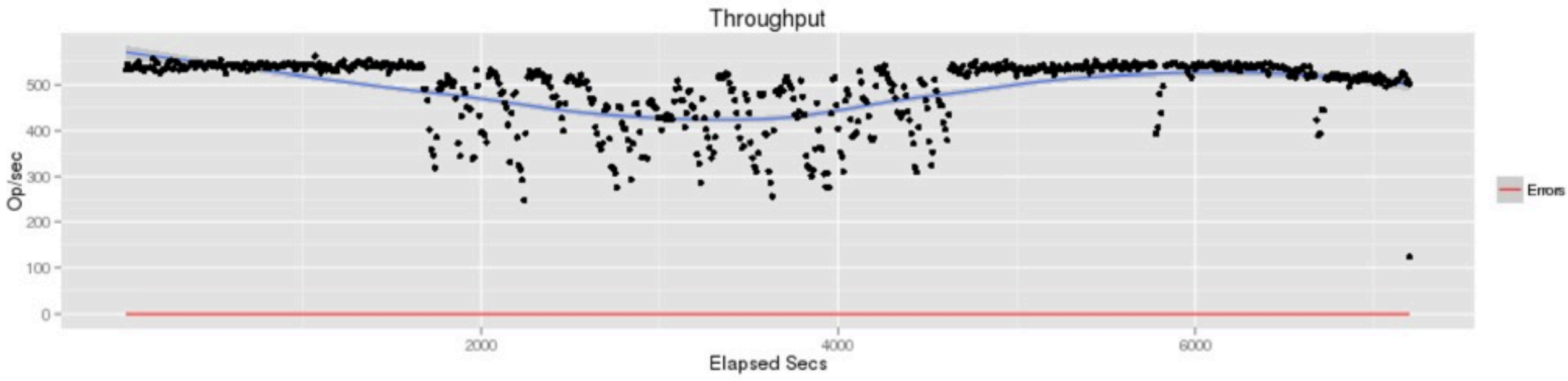
Let's Scale

- Scaling up/down in Riak means adding/removing nodes
- Adding: new nodes claim ring partitions
- Removing: existing nodes reclaim ring partitions from leaving nodes

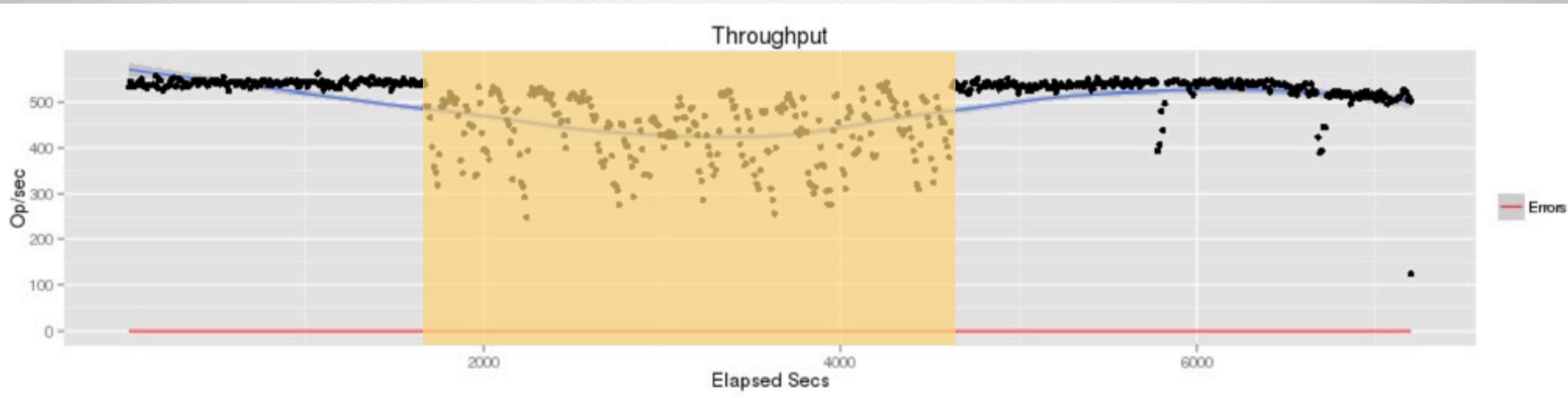
Let's Scale

- Scaling up/down in Riak means adding/removing nodes
- Adding: new nodes claim ring partitions
- Removing: existing nodes reclaim ring partitions from leaving nodes
- Handoff occurs to move data between nodes

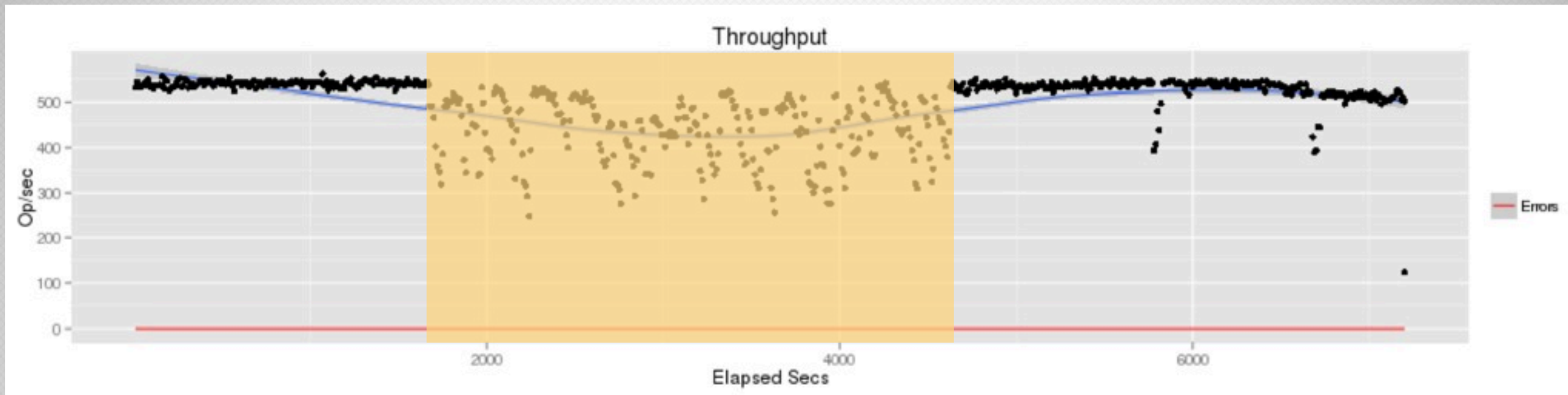
Throughput With Node Join/Leave



Throughput With Node Join/Leave

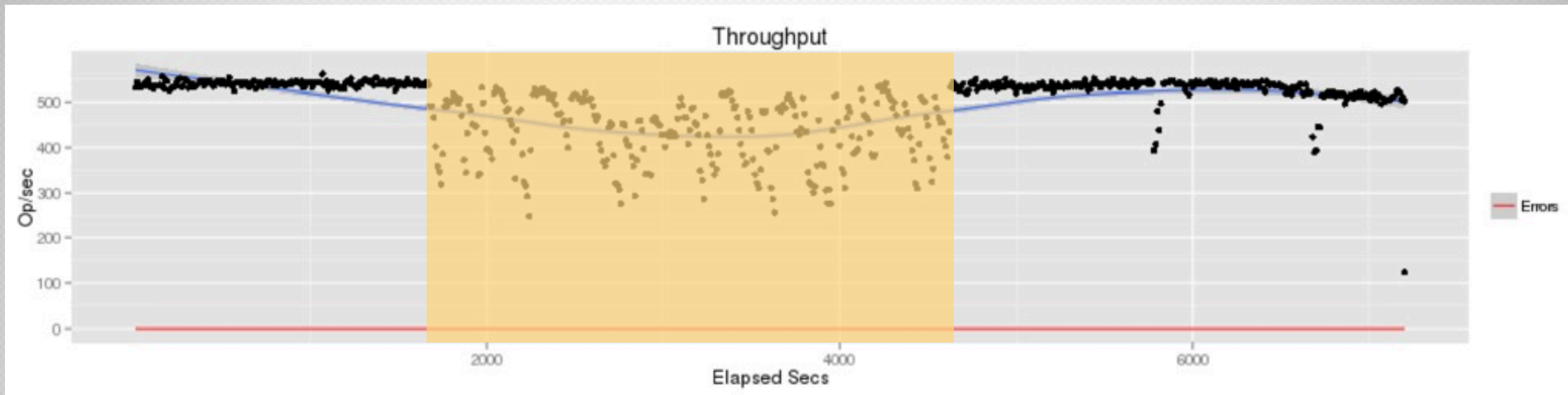


Throughput With Node Join/Leave



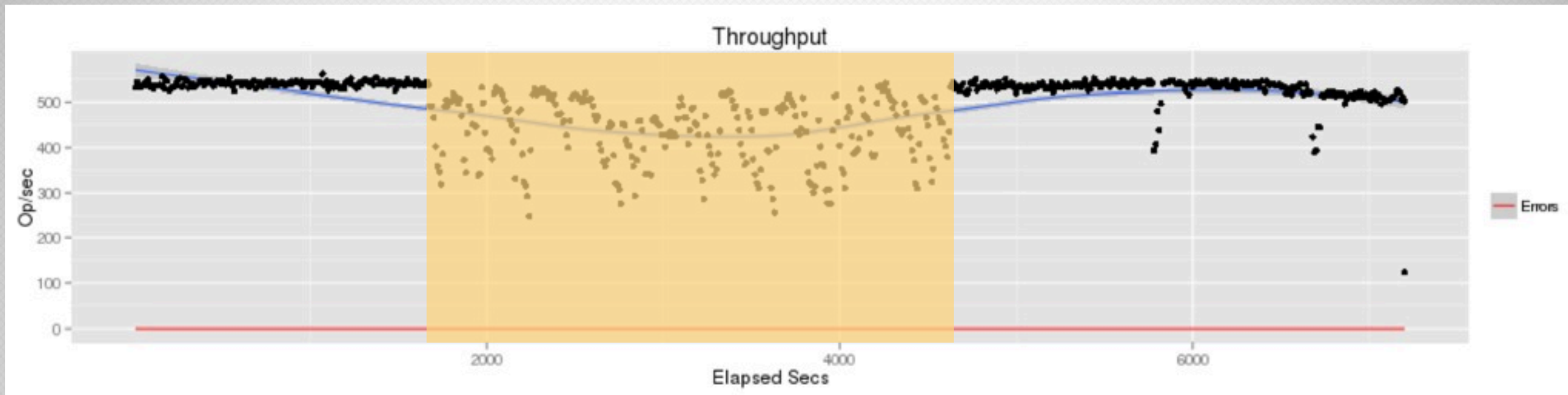
- Latencies also increase

Throughput With Node Join/Leave



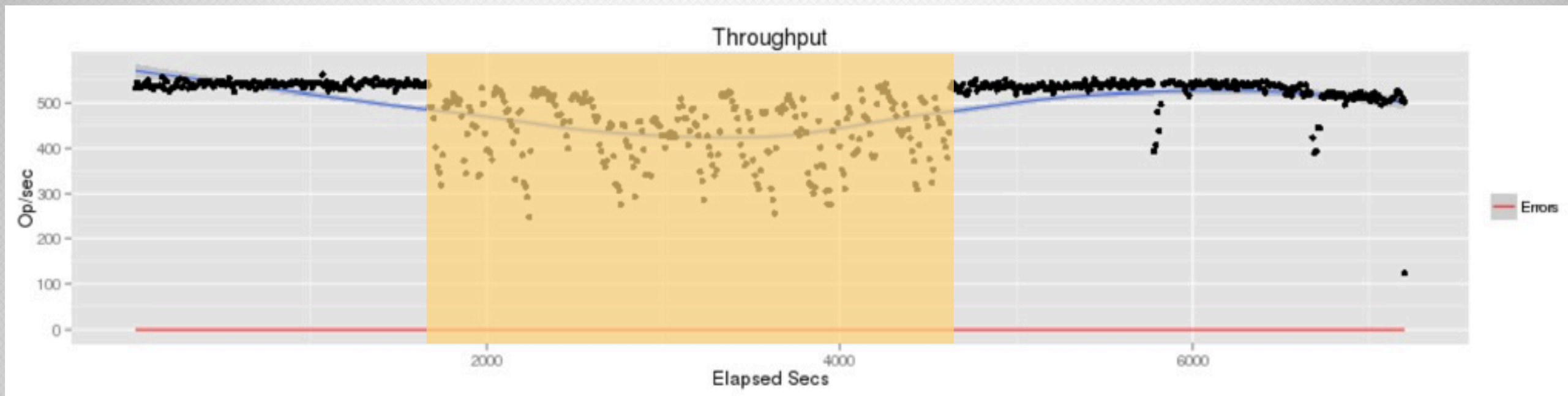
- Latencies also increase
- Increases in I/O, CPU, network congestion

Throughput With Node Join/Leave



- Latencies also increase
- Increases in I/O, CPU, network congestion
- Potential for TCP incast problems

Throughput With Node Join/Leave



- Latencies also increase
- Increases in I/O, CPU, network congestion
- Potential for TCP incast problems
- Potential for client timeouts

TCP Incast

TCP Incast

- Affects many-to-one operations in datacenters

TCP Incast

- Affects many-to-one operations in datacenters
- In microbursts, senders overrun switch buffers, packets are dropped, senders back off and slow down

TCP Incast

- Affects many-to-one operations in datacenters
- In microbursts, senders overrun switch buffers, packets are dropped, senders back off and slow down
- Result is significant throughput collapse

TCP Incast

- Affects many-to-one operations in datacenters
- In microbursts, senders overrun switch buffers, packets are dropped, senders back off and slow down
- Result is significant throughput collapse
- Affects systems like Riak because multiple vnodes (the many) often send messages nearly simultaneously to a coordinator (the one)

LEDBAT

LEDBAT

- Low Extra Delay Background Transport (RFC 6817, experimental, Dec. 2012)

LEDBAT

- Low Extra Delay Background Transport (RFC 6817, experimental, Dec. 2012)
- Quick reacting delay-based congestion control

LEDBAT

- Low Extra Delay Background Transport (RFC 6817, experimental, Dec. 2012)
- Quick reacting delay-based congestion control
- Uses one-way delay measurements to estimate data path queuing

LEDBAT

- Low Extra Delay Background Transport (RFC 6817, experimental, Dec. 2012)
- Quick reacting delay-based congestion control
- Uses one-way delay measurements to estimate data path queuing
- Adds low extra queuing delay to minimize interference with other flows

LEDBAT

- Low Extra Delay Background Transport (RFC 6817, experimental, Dec. 2012)
- Quick reacting delay-based congestion control
- Uses one-way delay measurements to estimate data path queuing
- Adds low extra queuing delay to minimize interference with other flows
- Suitable for "background" tasks like bulk data transfer

Micro Transport Protocol (uTP)

Micro Transport Protocol (uTP)

- LEDBAT congestion control, precedes the RFC

Micro Transport Protocol (uTP)

- LEDBAT congestion control, precedes the RFC
- Created in Internet2 research, implemented by Plicto, acquired by Bittorrent in 2006

Micro Transport Protocol (uTP)

- LEDBAT congestion control, precedes the RFC
- Created in Internet2 research, implemented by Plicto, acquired by Bittorrent in 2006
- Bittorrent has been using uTP since 2009

Micro Transport Protocol (uTP)

- LEDBAT congestion control, precedes the RFC
- Created in Internet2 research, implemented by Plicto, acquired by Bittorrent in 2006
- Bittorrent has been using uTP since 2009
- Their C++ library implementation is on github: https://github.com/bittorrent/libutp

Integrating Libutp Into Riak

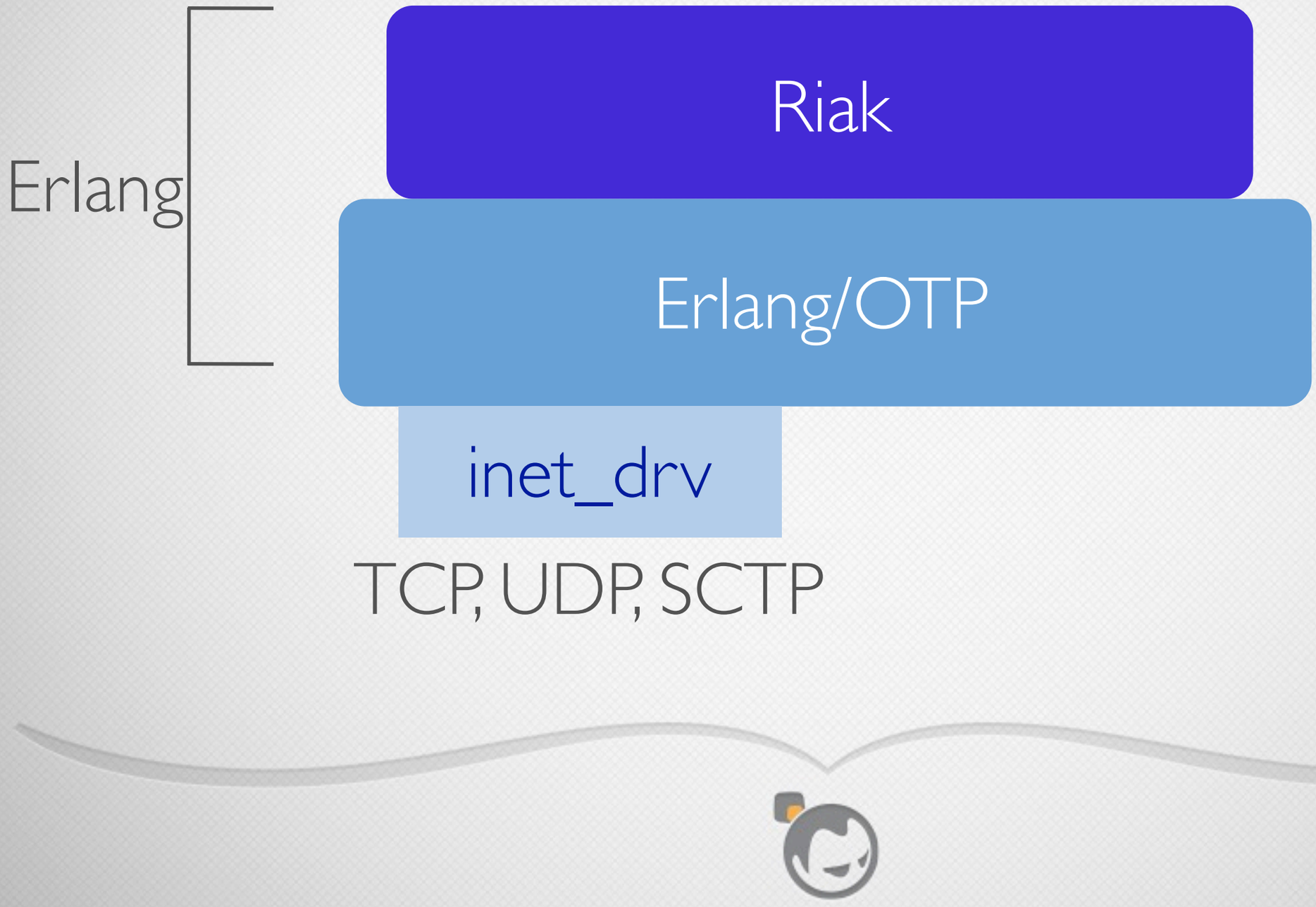
Integrating Libutp Into Riak

Riak

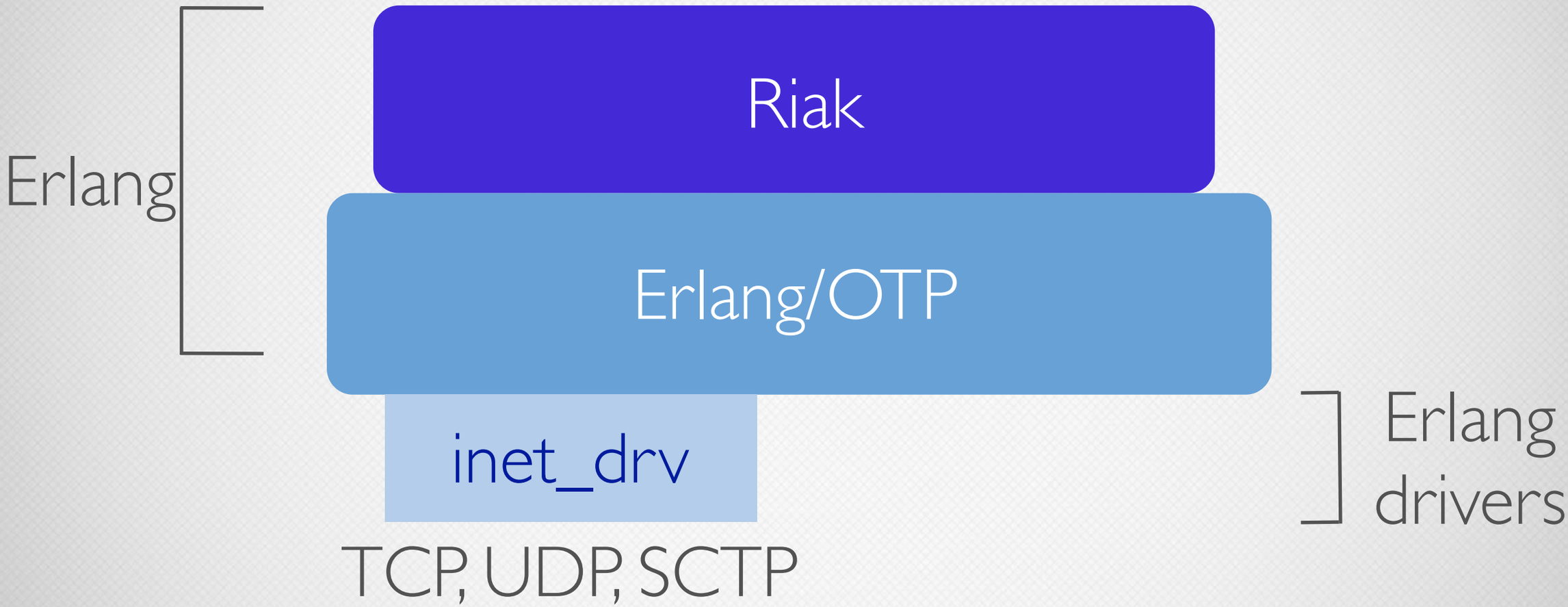
Integrating Libutp Into Riak

Integrating Libutp Into Riak

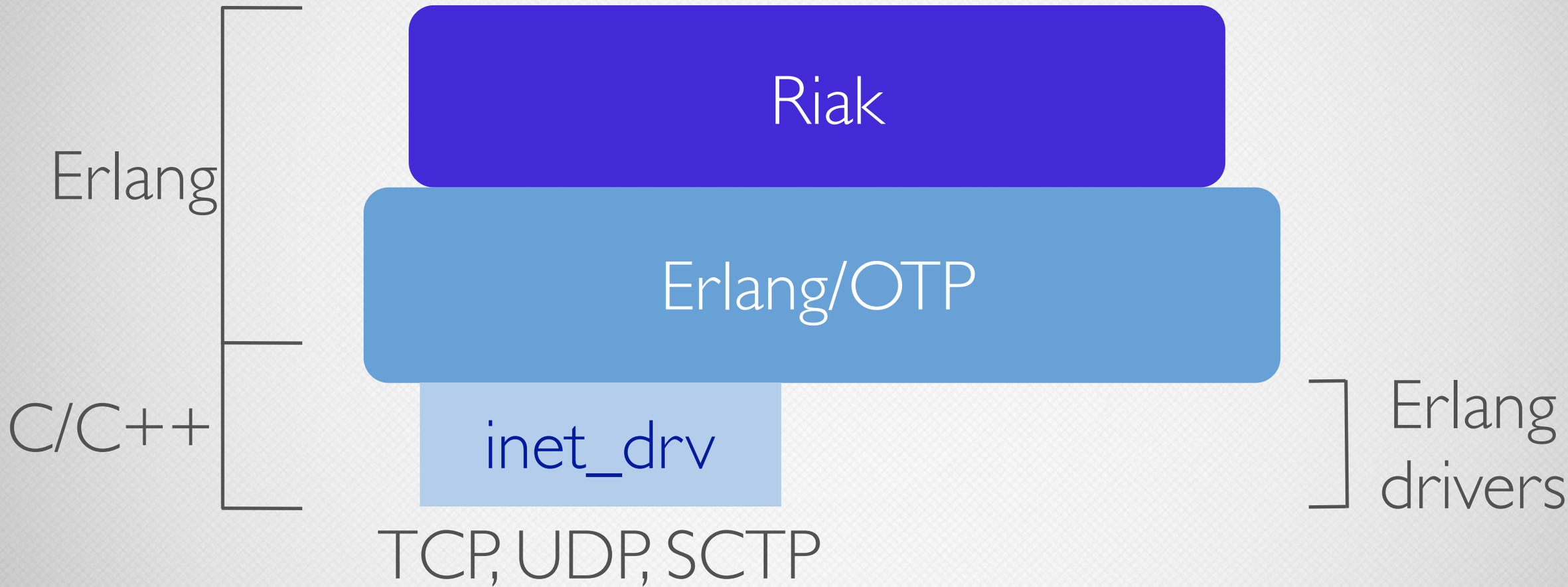
Integrating Libutp Into Riak



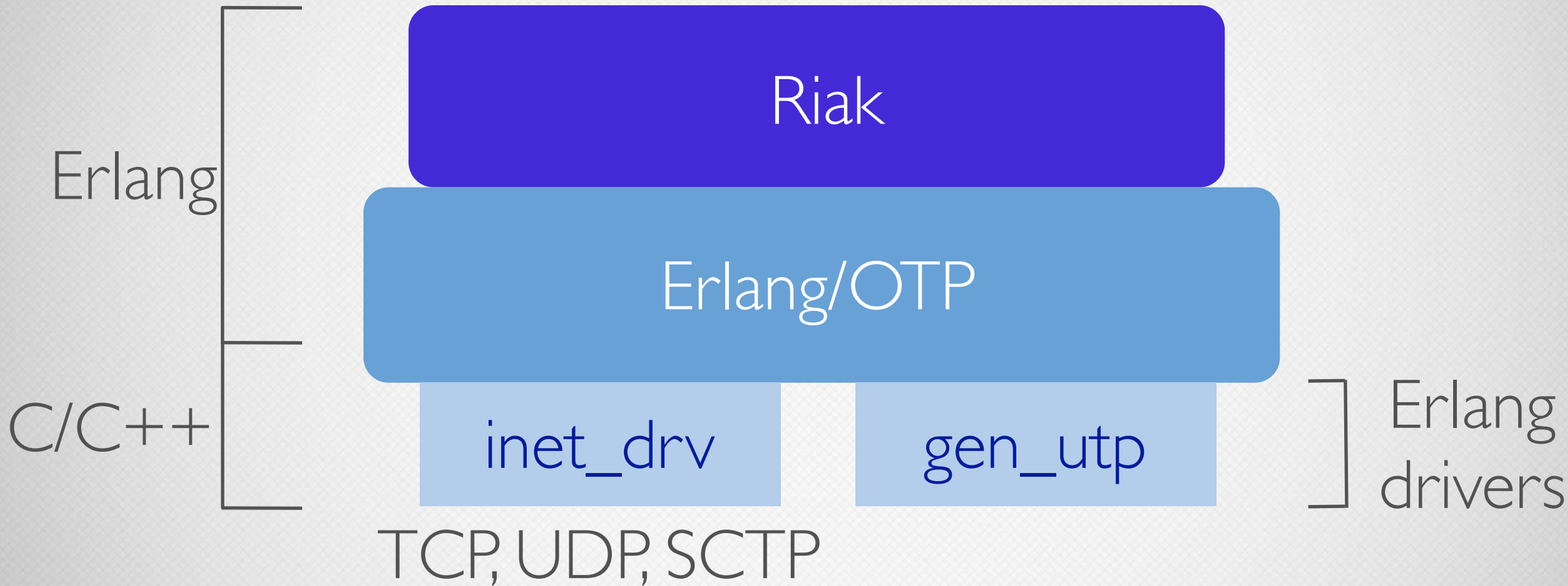
Integrating Libutp Into Riak



Integrating Libutp Into Riak

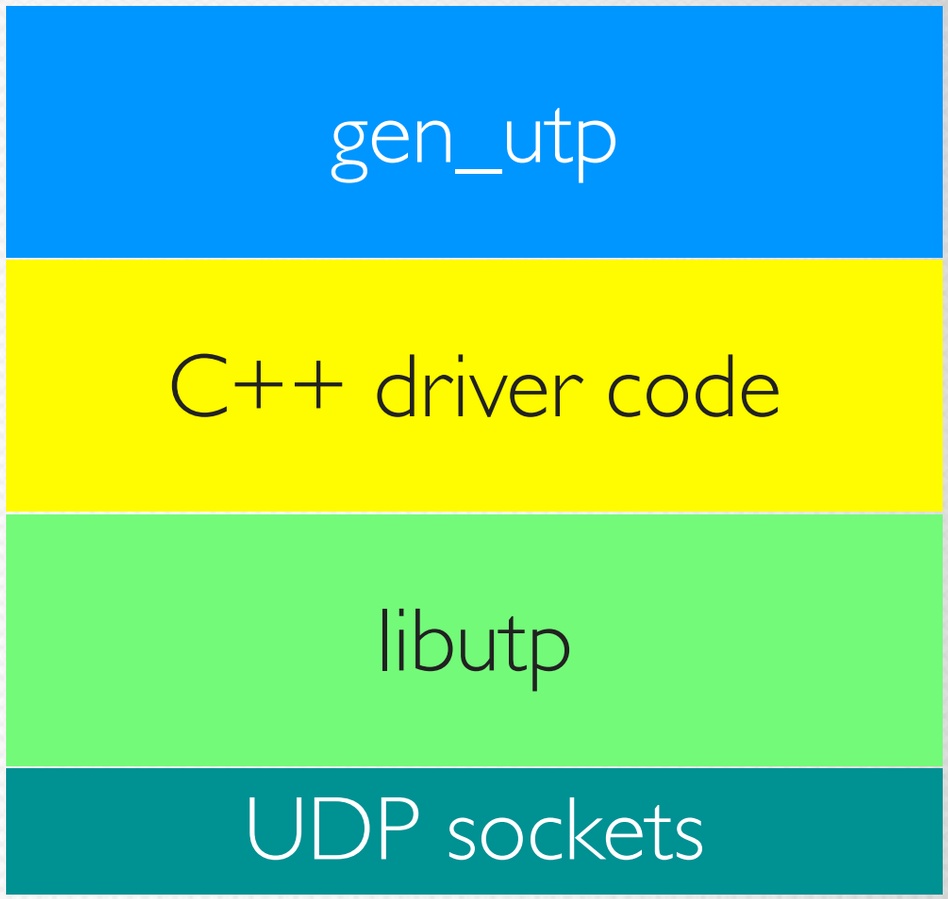


Integrating Libutp Into Riak



Gen_utp

- Erlang interface matches standard library gen_tcp
- gen_utp module wraps access to the driver
- C++ driver code wraps libutp
- C++ driver also manages underlying UDP sockets



Gen_utp Features

Gen_utp Features

- Connection-oriented protocol

Gen_utp Features

- Connection-oriented protocol
- uTP sockets represented via Erlang ports, same as for TCP and UDP

Gen_utp Features

- Connection-oriented protocol
- uTP sockets represented via Erlang ports, same as for TCP and UDP
- Active modes: false, true, once

Gen_utp Features

- Connection-oriented protocol
- uTP sockets represented via Erlang ports, same as for TCP and UDP
- Active modes: false, true, once
- Binary or list data delivery

Gen_utp Features

- Connection-oriented protocol
- uTP sockets represented via Erlang ports, same as for TCP and UDP
- Active modes: false, true, once
- Binary or list data delivery
- Supports sending iolists

Gen_utp Features

- Connection-oriented protocol
- uTP sockets represented via Erlang ports, same as for TCP and UDP
- Active modes: false, true, once
- Binary or list data delivery
- Supports sending iolists
- IPv4 and IPv6

Gen_utp Features

Gen_utp Features

- Packet option (raw, 0, 1, 2, 4)

Gen_utp Features

- Packet option (raw, 0, 1, 2, 4)
- Message headers (first N bytes of each message delivered as a list)

Gen_utp Features

- Packet option (raw, 0, 1, 2, 4)
- Message headers (first N bytes of each message delivered as a list)
- Network interface binding

Gen_utp Features

- Packet option (raw, 0, 1, 2, 4)
- Message headers (first N bytes of each message delivered as a list)
- Network interface binding
- Attach to already-open UDP socket file descriptor

Gen_utp Functions

Gen_utp Functions

- listen, accept, async_accept

Gen_utp Functions

- listen, accept, async_accept
- connect

Gen_utp Functions

- listen, accept, async_accept
- connect
- send, recv

Gen_utp Functions

- listen, accept, async_accept
- connect
- send, recv
- close

Gen_utp Functions

- listen, accept, async_accept
- connect
- send, recv
- close
- sockname, peername, port

Gen_utp Functions

- listen, accept, async_accept
- connect
- send, recv
- close
- sockname, peername, port
- setopts, getopts

Gen_utp Functions

- listen, accept, async_accept
- connect
- send, recv
- close
- sockname, peername, port
- setopts, getopts
- controlling_process

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
22> {ok,ClientSock} = gen_utp:connect("localhost", ServerPort, [{active,true}]).
{ok,#Port<0.617>}
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
22> {ok,ClientSock} = gen_utp:connect("localhost", ServerPort, [{active,true}]).
{ok,#Port<0.617>}
23> receive {utp_async,ListenSock,Ref,{ok,ServerSock}} -> ServerSock end.
#Port<0.618>
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
22> {ok,ClientSock} = gen_utp:connect("localhost", ServerPort, [{active,true}]).
{ok,#Port<0.617>}
23> receive {utp_async,ListenSock,Ref,{ok,ServerSock}} -> ServerSock end.
#Port<0.618>
24> gen_utp:send(ClientSock, "this is a test").
ok
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
22> {ok,ClientSock} = gen_utp:connect("localhost", ServerPort, [{active,true}]).
{ok,#Port<0.617>}
23> receive {utp_async,ListenSock,Ref,{ok,ServerSock}} -> ServerSock end.
#Port<0.618>
24> gen_utp:send(ClientSock, "this is a test").
ok
25> {ok,Msg} = gen_utp:recv(ServerSock, 0).
{ok,<<"this is a test">>}
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
22> {ok,ClientSock} = gen_utp:connect("localhost", ServerPort, [{active,true}]).
{ok,#Port<0.617>}
23> receive {utp_async,ListenSock,Ref,{ok,ServerSock}} -> ServerSock end.
#Port<0.618>
24> gen_utp:send(ClientSock, "this is a test").
ok
25> {ok,Msg} = gen_utp:recv(ServerSock, 0).
{ok,<<"this is a test">>}
26> gen_utp:send(ServerSock, <<"this is also a test">>).
ok
```

Gen_utp Example

```
19> {ok,ListenSock} = gen_utp:listen(0, [{active,false},binary]).
{ok,#Port<0.616>}
20> {ok,Ref} = gen_utp:async_accept(ListenSock).
{ok,#Ref<0.0.0.177>}
21> {ok,{ServerIP,ServerPort}} = gen_utp:sockname(ListenSock).
{ok,{{0,0,0,0},60704}}
22> {ok,ClientSock} = gen_utp:connect("localhost", ServerPort, [{active,true}]).
{ok,#Port<0.617>}
23> receive {utp_async,ListenSock,Ref,{ok,ServerSock}} -> ServerSock end.
#Port<0.618>
24> gen_utp:send(ClientSock, "this is a test").
ok
25> {ok,Msg} = gen_utp:recv(ServerSock, 0).
{ok,<<"this is a test">>}
26> gen_utp:send(ServerSock, <<"this is also a test">>).
ok
27> flush().
Shell got {utp,#Port<0.617>,"this is also a test"}
ok
```

Gen_utp Internals

- libutp is a C++ library, so the Erlang driver is also C++
- libutp works via callbacks
- libutp implements the uTP protocol, you have to supply all socket handling
- Sockets are UDP, libutp adds the protocol reliability

Gen_utp Internals

- Master branch has a C++ class hierarchy of Handlers
- Handlers implement socket handling, uTP handling, and Erlang port handling
- Development branch (not yet working) breaks these into parallel hierarchies of Handlers and Ports

Handler Classes

Handler Classes

- SocketHandler: handles UDP sockets

Handler Classes

- SocketHandler: handles UDP sockets
  - a listener uses a SocketHandler

Handler Classes

- SocketHandler: handles UDP sockets
  - a listener uses a SocketHandler
- UtpHandler: handles libutp callbacks

Handler Classes

- SocketHandler: handles UDP sockets
  - a listener uses a SocketHandler
- UtpHandler: handles libutp callbacks
  - derived from SocketHandler

Handling Events

- UDP sockets are registered in the Erlang runtime's polling set
- Erlang runtime calls SocketHandlers when sockets have input
- libutp also has a timeout check that the uTP driver calls every 10ms

Port Classes

Port Classes

- DrvPort: abstract base class for all Port classes

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver
  - implements listen and connect calls

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver
  - implements listen and connect calls
- SocketPort: base class for ports dealing with SocketHandlers

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver
  - implements listen and connect calls
- SocketPort: base class for ports dealing with SocketHandlers
- ListenPort: port returned from listen calls

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver
  - implements listen and connect calls
- SocketPort: base class for ports dealing with SocketHandlers
  - ListenPort: port returned from listen calls
- UtpPort: base class for ports dealing with UtpHandlers

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver
  - implements listen and connect calls
- SocketPort: base class for ports dealing with SocketHandlers
  - ListenPort: port returned from listen calls
- UtpPort: base class for ports dealing with UtpHandlers
  - AcceptPort: port returned from accept calls

Port Classes

- DrvPort: abstract base class for all Port classes
- MainPort: implements initial port into uTP driver
  - implements listen and connect calls
- SocketPort: base class for ports dealing with SocketHandlers
  - ListenPort: port returned from listen calls
- UtpPort: base class for ports dealing with UtpHandlers
  - AcceptPort: port returned from accept calls
  - ConnectPort: port returned from connect calls

Implementing Accept

Implementing Accept

- A uTP client "connects" to a uTP listener, but it's really connectionless UDP underneath

Implementing Accept

- A uTP client "connects" to a uTP listener, but it's really connectionless UDP underneath
- TCP accept means "give me a new socket connected to that client", and we want the same semantics

Implementing Accept

Implementing Accept

- For incoming connection requests:

Implementing Accept

- For incoming connection requests:
  - open a new accept socket sharing the listen port (using SO_REUSEADDR or SO_REUSEPORT)

Implementing Accept

- For incoming connection requests:
  - open a new accept socket sharing the listen port (using SO_REUSEADDR or SO_REUSEPORT)
  - connect(2) the UDP accept socket to the client (yes, connect works for UDP too)

Implementing Accept

- For incoming connection requests:
  - open a new accept socket sharing the listen port (using SO_REUSEADDR or SO_REUSEPORT)
  - connect(2) the UDP accept socket to the client (yes, connect works for UDP too)
  - any subsequent traffic from that client is seen only by the accept socket

Implementing Accept

- For incoming connection requests:
  - open a new accept socket sharing the listen port (using SO_REUSEADDR or SO_REUSEPORT)
  - connect(2) the UDP accept socket to the client (yes, connect works for UDP too)
  - any subsequent traffic from that client is seen only by the accept socket
  - all sends on the accept socket go only to that client (i.e., using send vs. sendto)

Implementing Accept

- Unlike inet_drv, the uTP driver uses the driver queue for reads, not writes
- Implementing {active,false} or {active,once} for TCP just means deselecting the socket
- uTP driver always has to read all incoming messages to check if they're uTP messages, so it never deselects
  - driver queue stores read messages not yet delivered up through gen_utp

Shortcomings

- No good way to implement a listen queue
  - uTP client will just timeout if nobody's accepting
- uTP is slow when closing a socket, seems to want to exchange a bunch of messages
- libutp is not thread-safe, all access must be serialized
- Getting lifetimes of sockets, Erlang ports, and C++ handler instances right is hard
  - hoping the Handler/Port split will help

Gen_utp Testing

- Definitely a work in progress!
- Integrated with Riak some months ago on a branch
- successfully performed small-scale handoff
- but no large-scale Riak testing yet

Gen_utp Testing

Gen_utp Testing

- With direct Ethernet connection between two systems:

Gen_utp Testing

- With direct Ethernet connection between two systems:
  - same throughput as gen_tcp at 10baseT

Gen_utp Testing

- With direct Ethernet connection between two systems:
  - same throughput as gen_tcp at 10baseT
  - same throughput as gen_tcp at 100baseT

Gen_utp Testing

- With direct Ethernet connection between two systems:
  - same throughput as gen_tcp at 10baseT
  - same throughput as gen_tcp at 100baseT
  - 2x slower than gen_tcp at 1000baseT

Gen_utp Testing

- With direct Ethernet connection between two systems:
  - same throughput as gen_tcp at 10baseT
  - same throughput as gen_tcp at 100baseT
  - 2x slower than gen_tcp at 1000baseT
  - gen_utp shows higher CPU in all cases, most likely due to copying forced by libutp callback interface

Gen_utp Testing

Gen_utp Testing

- Lower throughput on fast networks could be a showstopper, since datacenter LANs are usually fast

Gen_utp Testing

- Lower throughput on fast networks could be a showstopper, since datacenter LANs are usually fast
- Always deferring to TCP flows might not always be desirable, for example:

Gen_utp Testing

- Lower throughput on fast networks could be a showstopper, since datacenter LANs are usually fast
- Always deferring to TCP flows might not always be desirable, for example:
  - when adding nodes to scale a cluster that's struggling to keep up with load

Gen_utp Testing

- Lower throughput on fast networks could be a showstopper, since datacenter LANs are usually fast
- Always deferring to TCP flows might not always be desirable, for example:
  - when adding nodes to scale a cluster that's struggling to keep up with load
  - you want data transfer to happen as quickly as possible so the new nodes help manage load

Current Status

- gen_utp available at https://github.com/basho-labs/gen_utp
- It mostly works but:
  - recent updates for Erlang R16B introduced bugs on master related to binary vs. list delivery
  - current development branch (Handler/Port split) still needs work

Next Steps

Next Steps

Next Steps

- Next step: testing on a Riak cluster under load

Next Steps

- Next step: testing on a Riak cluster under load
- Redesign the driver to work with Erlang's prim_inet layer

Next Steps

- Next step: testing on a Riak cluster under load
- Redesign the driver to work with Erlang's prim_inet layer
  - this should allow SSL to work over uTP

Next Steps

- Next step: testing on a Riak cluster under load
- Redesign the driver to work with Erlang's prim_inet layer
  - this should allow SSL to work over uTP
- If it doesn't help with congestion, consider using it for Riak Enterprise multi-datacenter syncing over WANs

Related Work

Related Work

- Jesper Louis Andersen (@jlouis) wrote a partial pure Erlang implementation of uTP: https://github.com/jlouis/erlang-utp

Related Work

- Jesper Louis Andersen (@jlouis) wrote a partial pure Erlang implementation of uTP: https://github.com/jlouis/erlang-utp
- He basically reverse engineered libutp

Related Work

- Jesper Louis Andersen (@jlouis) wrote a partial pure Erlang implementation of uTP: https://github.com/jlouis/erlang-utp
- He basically reverse engineered libutp
- I didn't use it because I thought libutp would make things easier, and because I can layer SSL over a driver

Related Work

- Jesper Louis Andersen (@jlouis) wrote a partial pure Erlang implementation of uTP: https://github.com/jlouis/erlang-utp
- He basically reverse engineered libutp
- I didn't use it because I thought libutp would make things easier, and because I can layer SSL over a driver
- He's stopped work on it but is willing to entertain pull requests :)

THANKS

http://basho.com
@stevevinoski