Stateful services with Riak Core

Ben Tyler
Erlang User Conference 2016

What?

What?

"joy of cool tech"

THIS IS SO COOL

What?

"joy of cool tech"

primarily a perl (web) hacker

What?

"joy of cool tech"

primarily a perl (web) hacker

...but distributed systems are a big part of my work

preemptive credits

- Mariano Guerra (!)
- Mark Allen
- Heinz Giles

outline

- talk
- mixed talk/code
- code

Stateful

Memory that lasts for more than one request

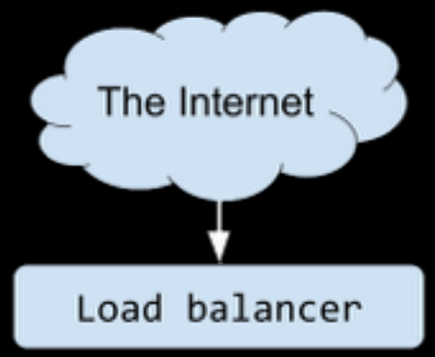
Stateful

(stateless?)

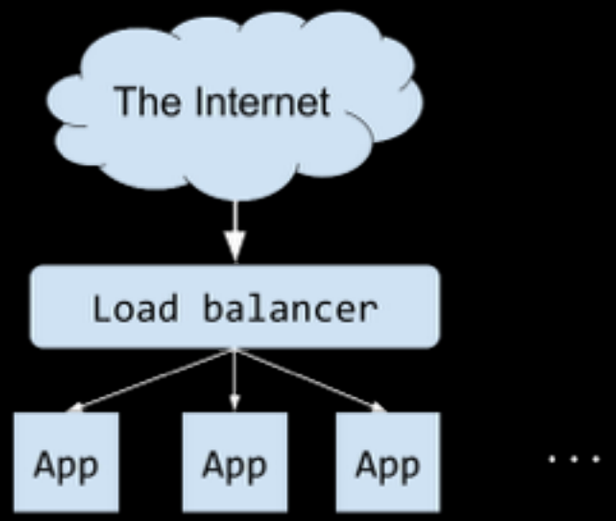
State*less*

State*less*

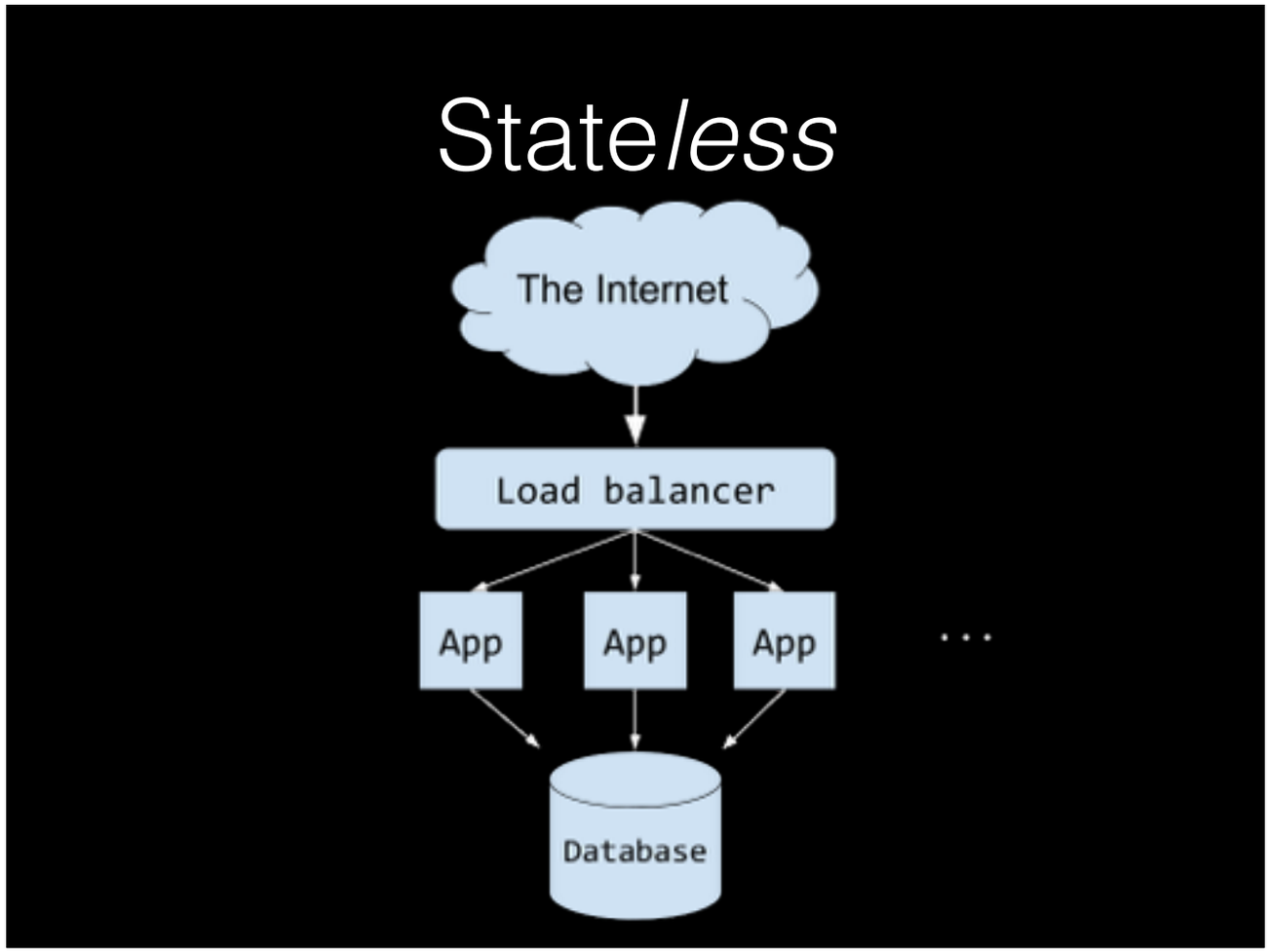
State*less*



State*less*

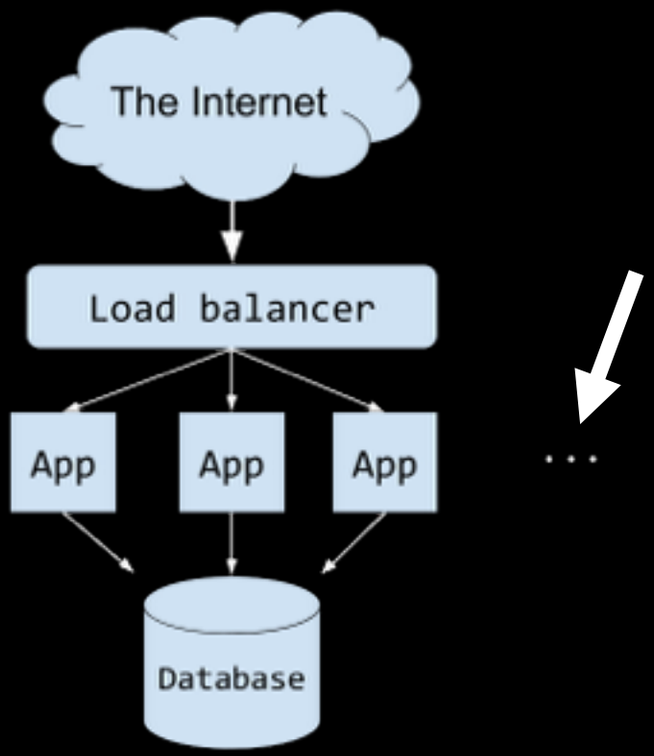


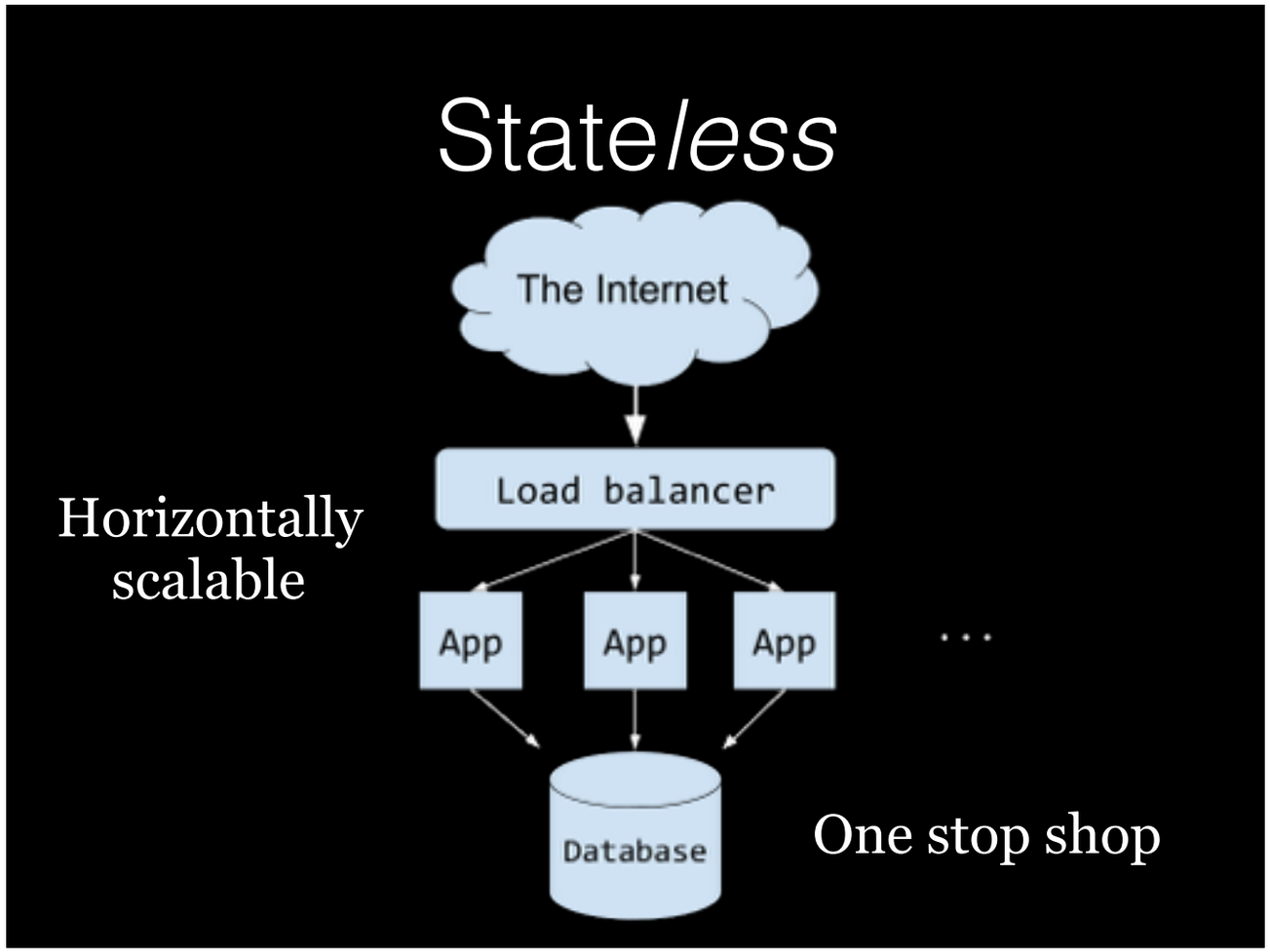
State*less*



State*less*

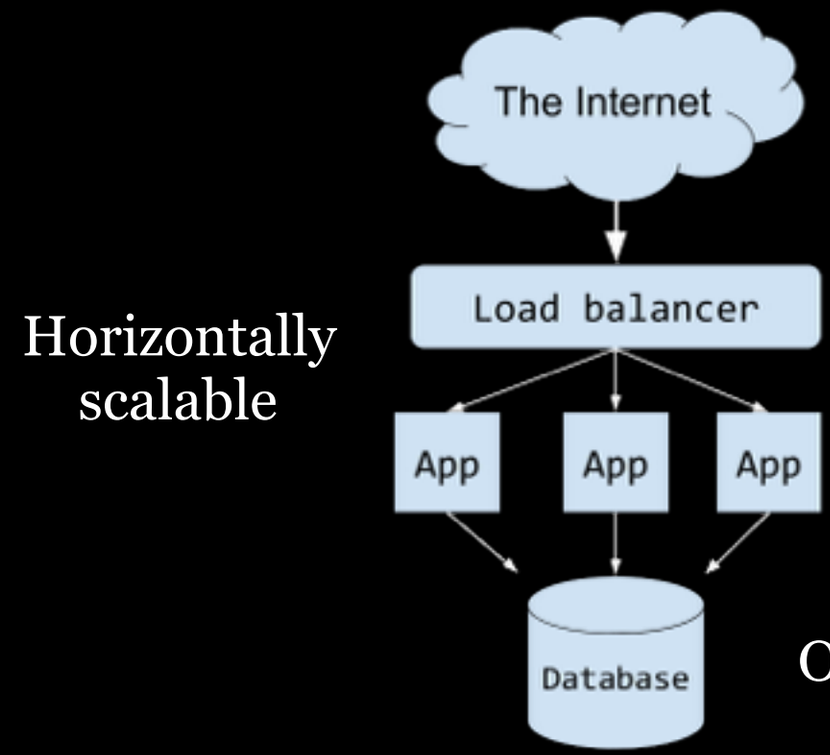
Horizontally scalable





you know exactly where to find all of your data: running reports that cover your entire dataset is straightforward

State*less*



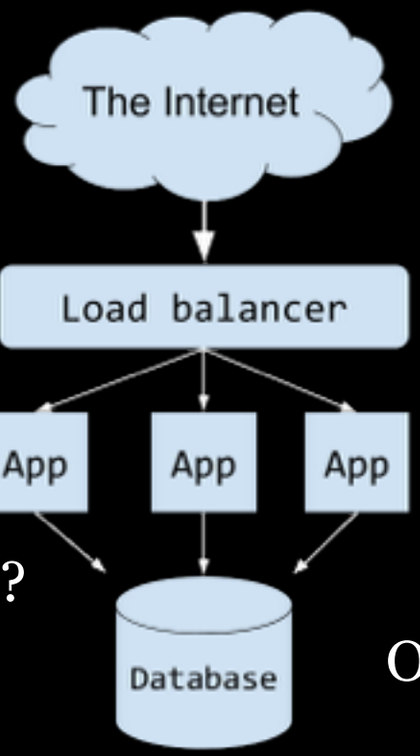
Horizontally scalable

Simple load balancing

...

One stop shop

State*less*



Horizontally scalable

Simple load balancing

...

Memory wasted?

One stop shop

Latency Numbers

CPU L1 cache reference	0:00:01

- *github.com/kofemann*

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20

- github.com/kofemann

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days

- github.com/kofemann

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days
Network round trip, same datacenter	11.5 days

- github.com/kofemann

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days
Network round trip, same datacenter	11.5 days
Read 1MB sequentially from disk	463 days

- github.com/kofemann

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days
Network round trip, same datacenter	11.5 days
Read 1MB sequentially from disk	463 days

- *github.com/kofemann*

So: reusing memory (even when it means some network calls) might be a very good thing. Anything to avoid hitting disk.

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days
Network round trip, same datacenter	11.5 days
Read 1MB sequentially from disk	463 days

- github.com/kofemann

can’t do much to manage this sort of thing as a high-level language user

Latency Numbers

CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days <- do this!
Network round trip, same datacenter	11.5 days
Read 1MB sequentially from disk	463 days

- github.com/kofemann

BUT maybe we can try to get more of our data into memory

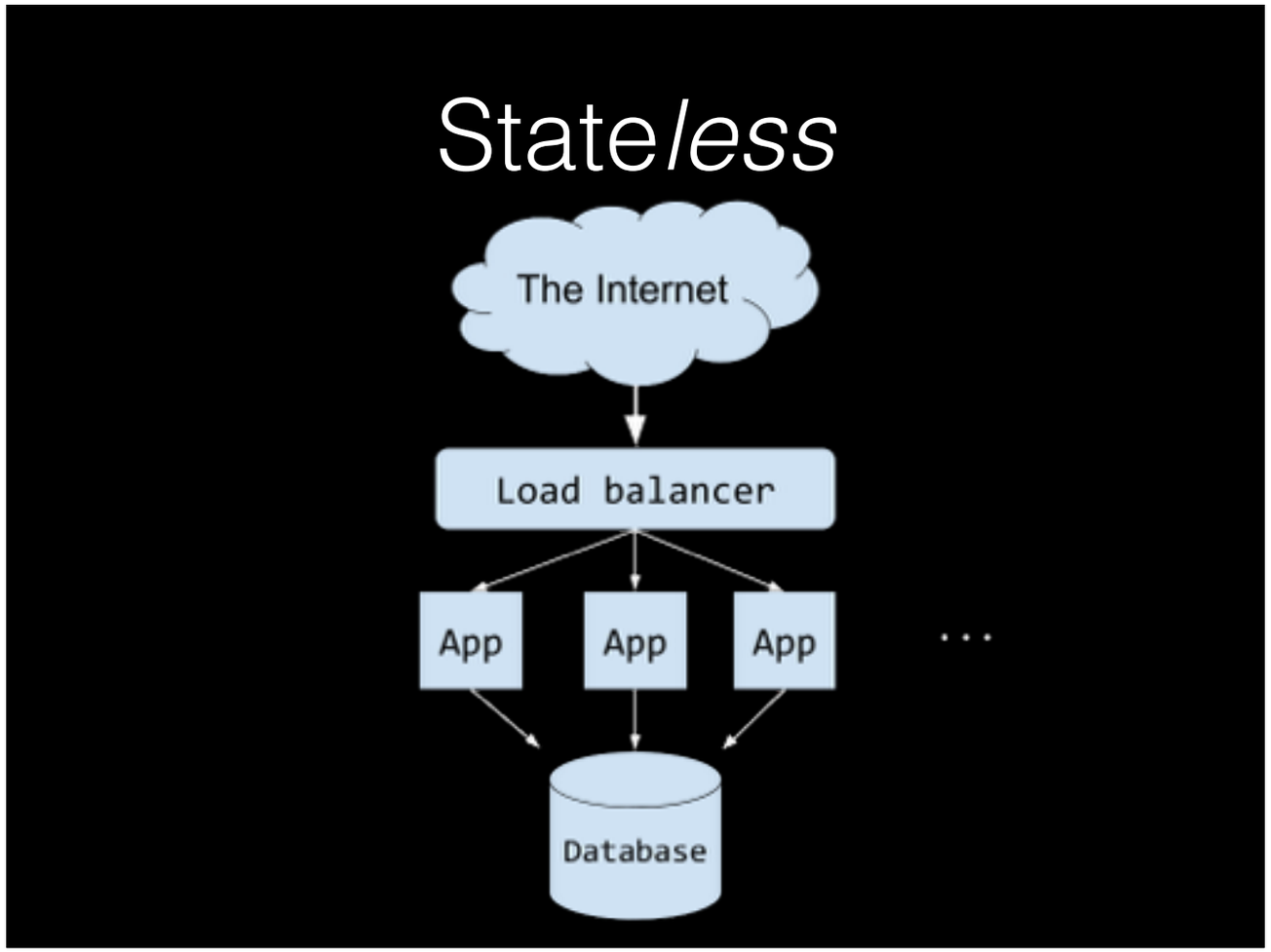
Latency Numbers

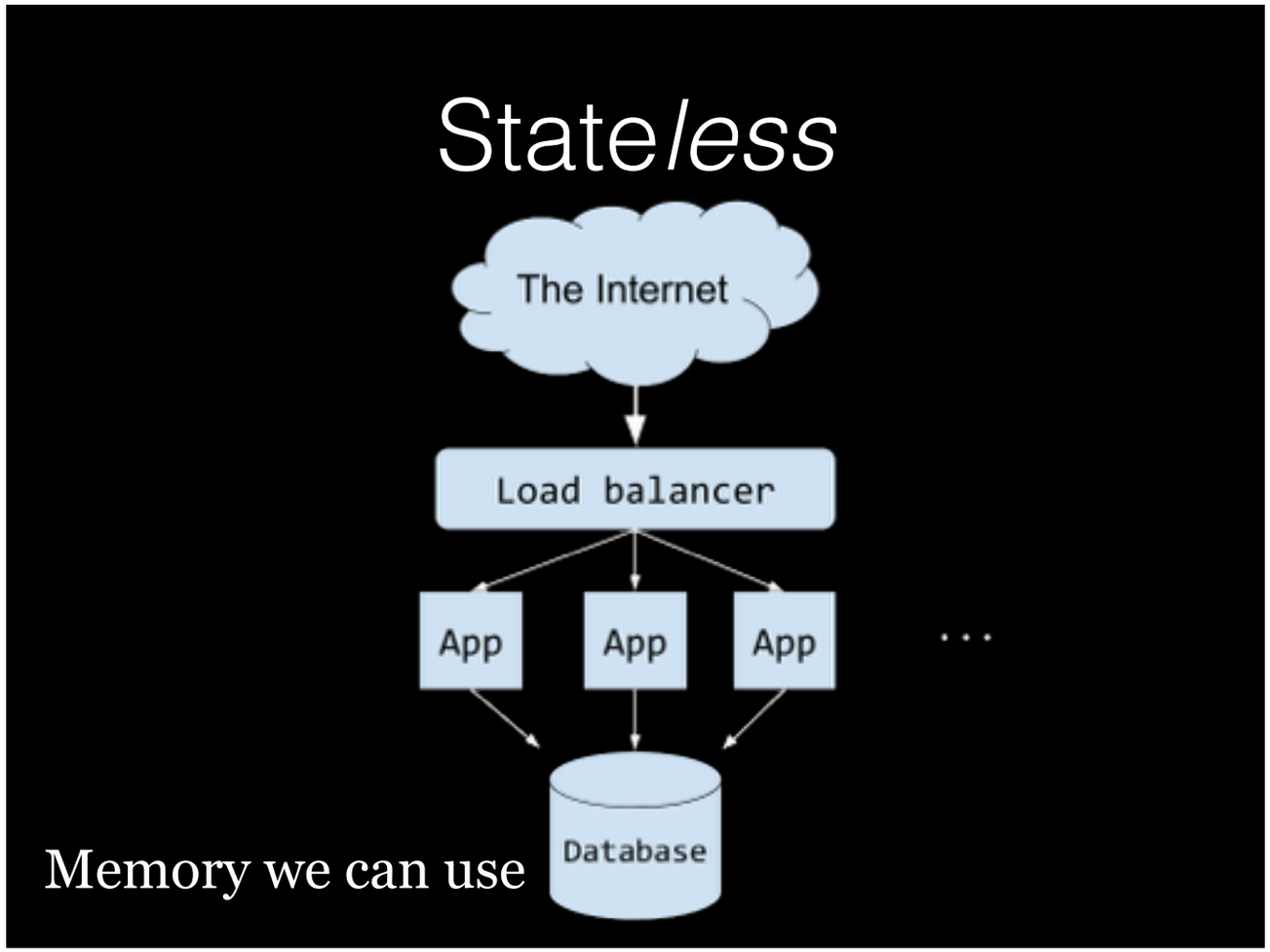
CPU L1 cache reference	0:00:01
Main memory reference	0:03:20
Read 1MB sequentially from memory	6 days <- do this!
Network round trip, same datacenter	11.5 days <- and this!
Read 1MB sequentially from disk	463 days

- *github.com/kofemann*

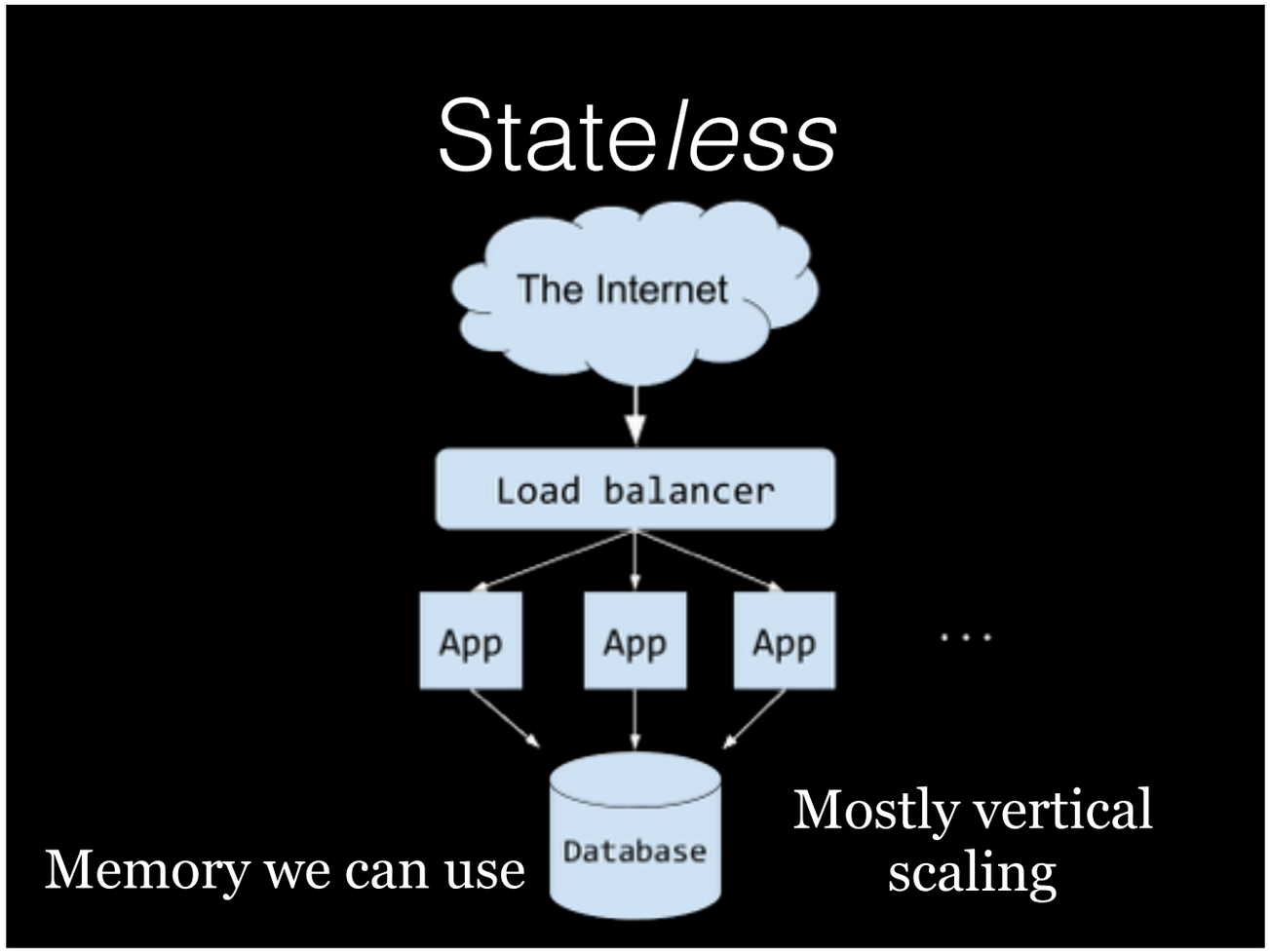
even if it means doing some extra network calls in order to manage that. this sort of change is similar to what Jose mentioned when he mentioned Moz's 'database-free architecture' yesterday.

State*less*





The memory that we can readily take advantage of in this architecture is the database (or we can involve dedicated memory servers like Redis or Memcached).



…and that tends to involve buying bigger database (or cache) servers. Even adding read-only replicas, if your data set is larger than the memory on a single box, you end up hoping that the particular bit of data you're querying for happens to be in the DB cache.

Horizontally scaling memory resources is one of the reasons why memcached is such a powerful tool.

Why state*less?*

So what drives us to keep using this architecture, if it has such constraints on how much memory we can leverage?

Why state*less?*

- Workload (HTTP)

Stateless protocol, stateless architecture. Checks out.

Why state*less?*

- Workload (HTTP)
- Hard to reuse memory

App server memory is really hard to re-use between requests, for a few reasons

Why state*less*?

- Workload (HTTP)
- Hard to reuse memory
  - Short lived programs

Originally CGI (process per request), these days workers that serve some number of requests before restarting (to avoid memory leaks, amusingly).

Why state*less*?

- Workload (HTTP)
- Hard to reuse memory
  - Short lived programs
  - Single threaded programs

Even when you have a long-running program serving many requests (see Event Loops, Twisted/Tornado/EventMachine/AnyEvent), Python/Ruby/Perl/PHP web apps tend to be single-threaded programs, so any memory sharing on the app involves communication between processes.

Why state*less*?

- Workload (HTTP)
- Hard to reuse memory
  - Short lived programs
  - Single threaded programs
  - Tricky to coordinate servers

Finally, it’s really tricky to coordinate between different app servers. Need message brokers or 0MQ, but those don’t fit into a standard web app all that well.

And for Erlang/Elixir*?*

And for Erlang/Elixir*?*

- Workload (HTTP)

And for Erlang/Elixir*?*

- Workload (HTTP) —> websockets

And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory

And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory
  - Short lived programs

And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory
  - Short lived programs —> BEAM!

And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory
  - Short lived programs —> BEAM!
  - Single threaded programs

And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory
  - Short lived programs —> BEAM!
  - Single threaded programs —> BEAM!

And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory
  - Short lived programs —> BEAM!
  - Single threaded programs —> BEAM!
  - Tricky to coordinate servers

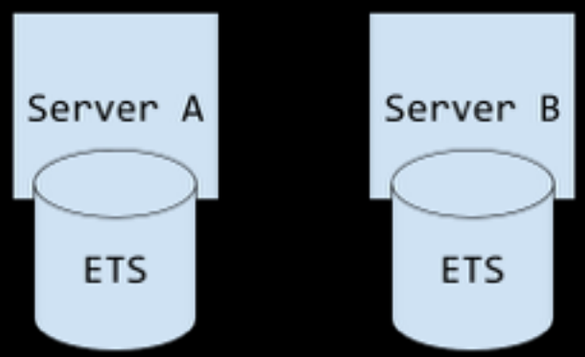
And for Erlang/Elixir?

- Workload (HTTP) —> websockets
- Hard to reuse memory
  - Short lived programs —> BEAM!
  - Single threaded programs —> BEAM!
  - Tricky to coordinate servers —> BEAM!

Awesome! Let's build
a stateful web app!

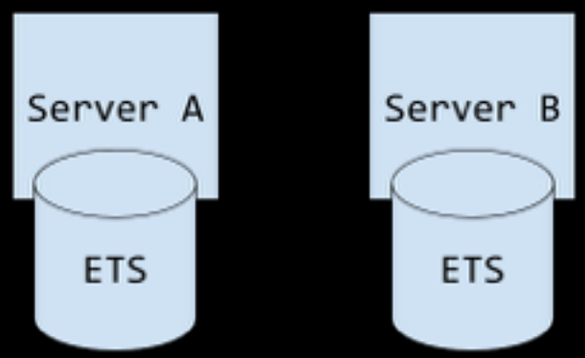
A stateful web app

A stateful web app



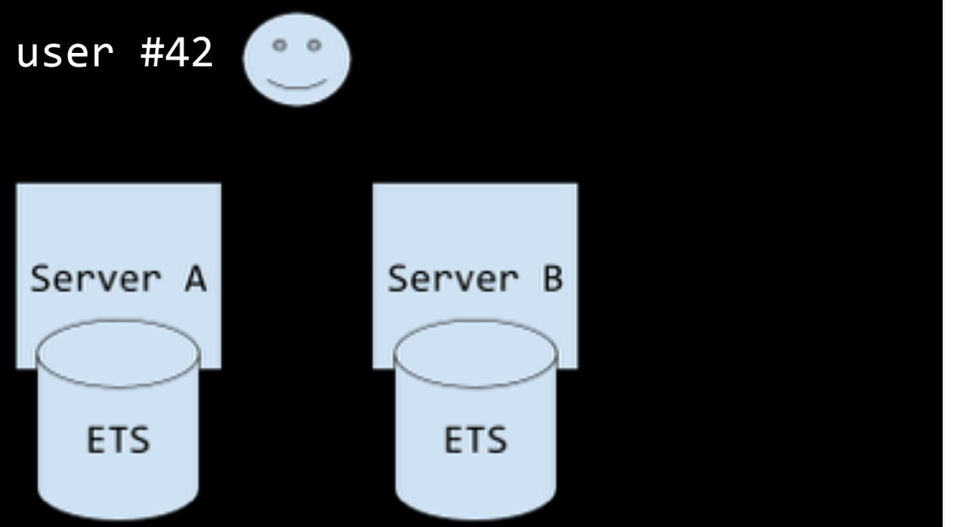
Storing our data in memory (using ETS) is what makes this state**ful**

A stateful web app

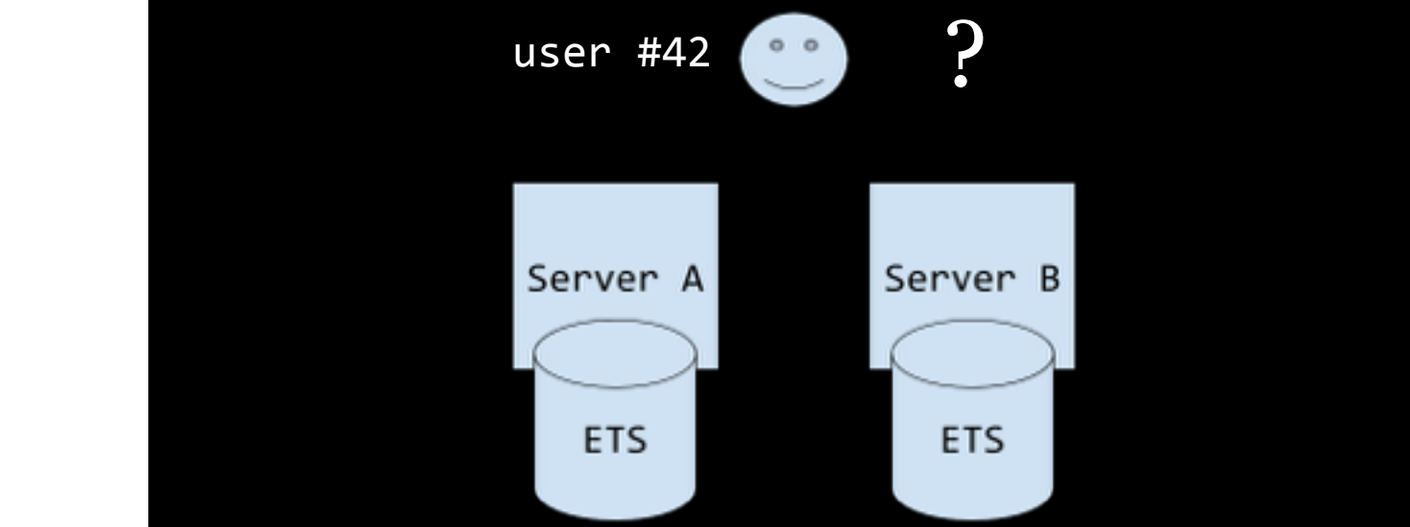



```
Data for user #42
```

A stateful web app

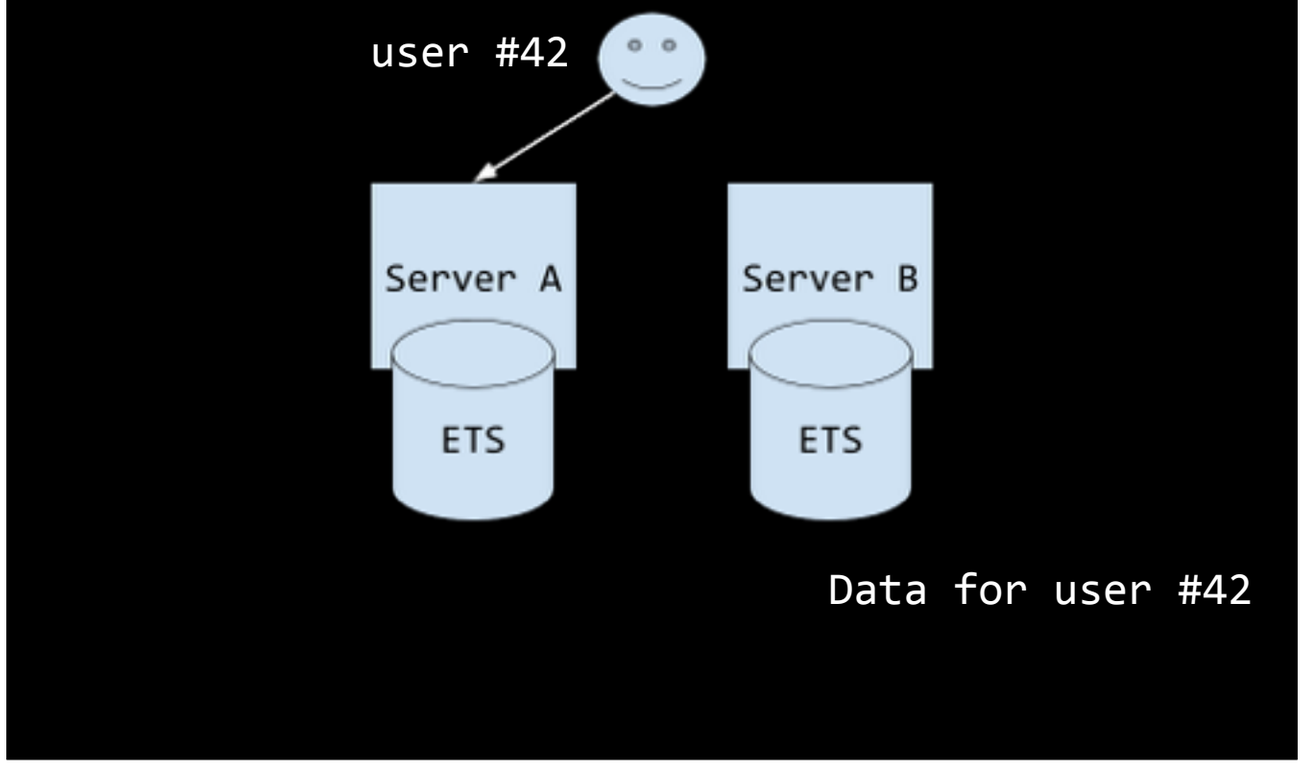


A stateful web app



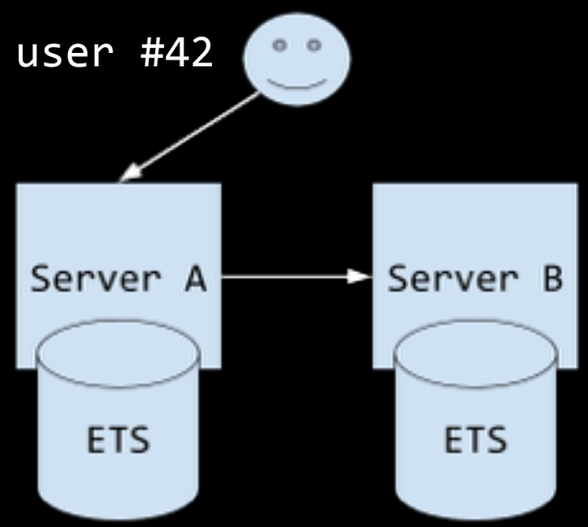
need to figure out where to route user #42: we _could_ introduce sticky load balancing, but that adds complexity and fragility

A stateful web app



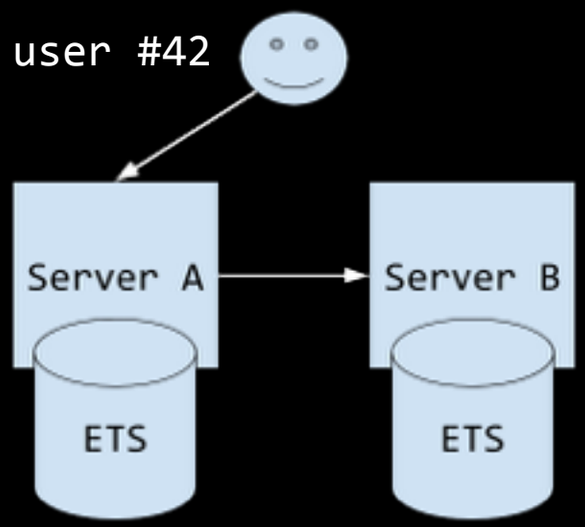
instead let’s stick with random load balancing, and just use distributed elixir to sort it out

A stateful web app



server A checks for user 42’s data; it doesn’t have it, need to ask server B

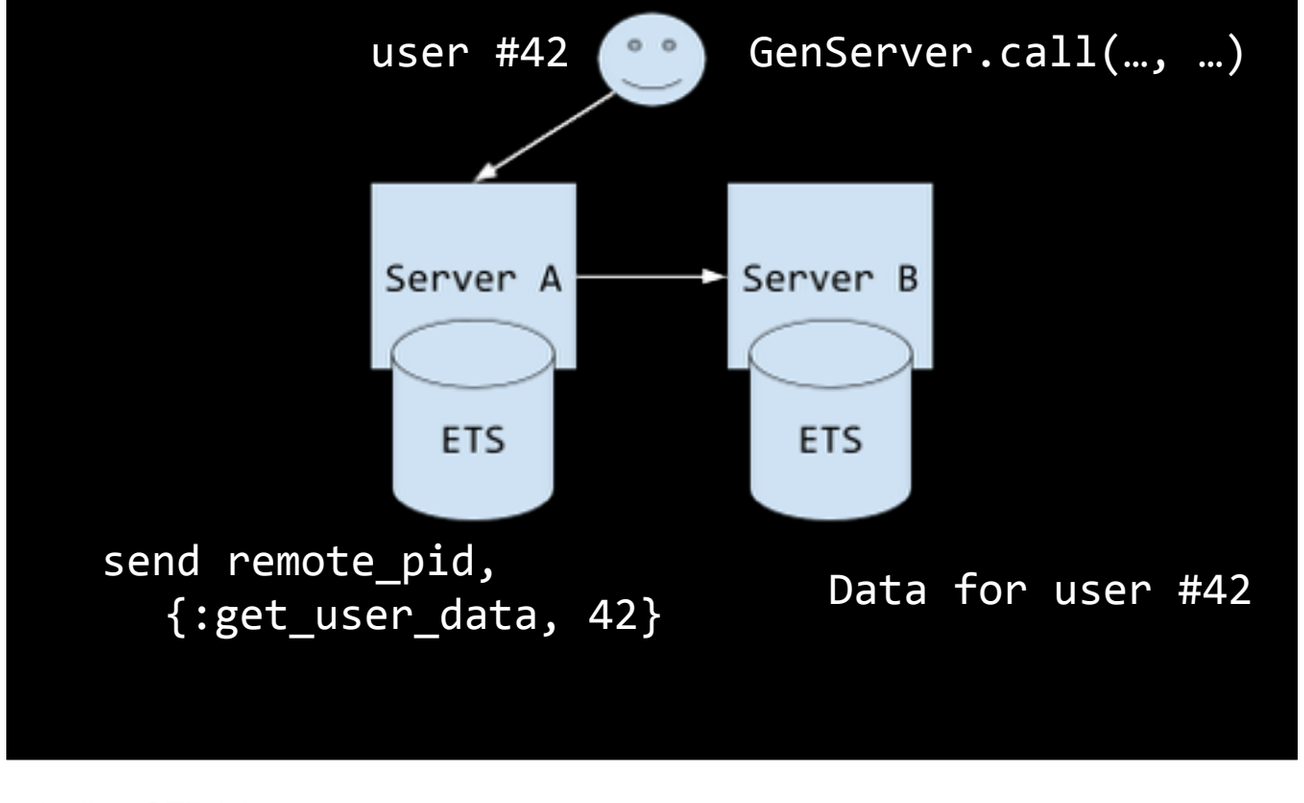
A stateful web app




```
send remote_pid,
   {:get_user_data, 42}
```

```
Data for user #42
```

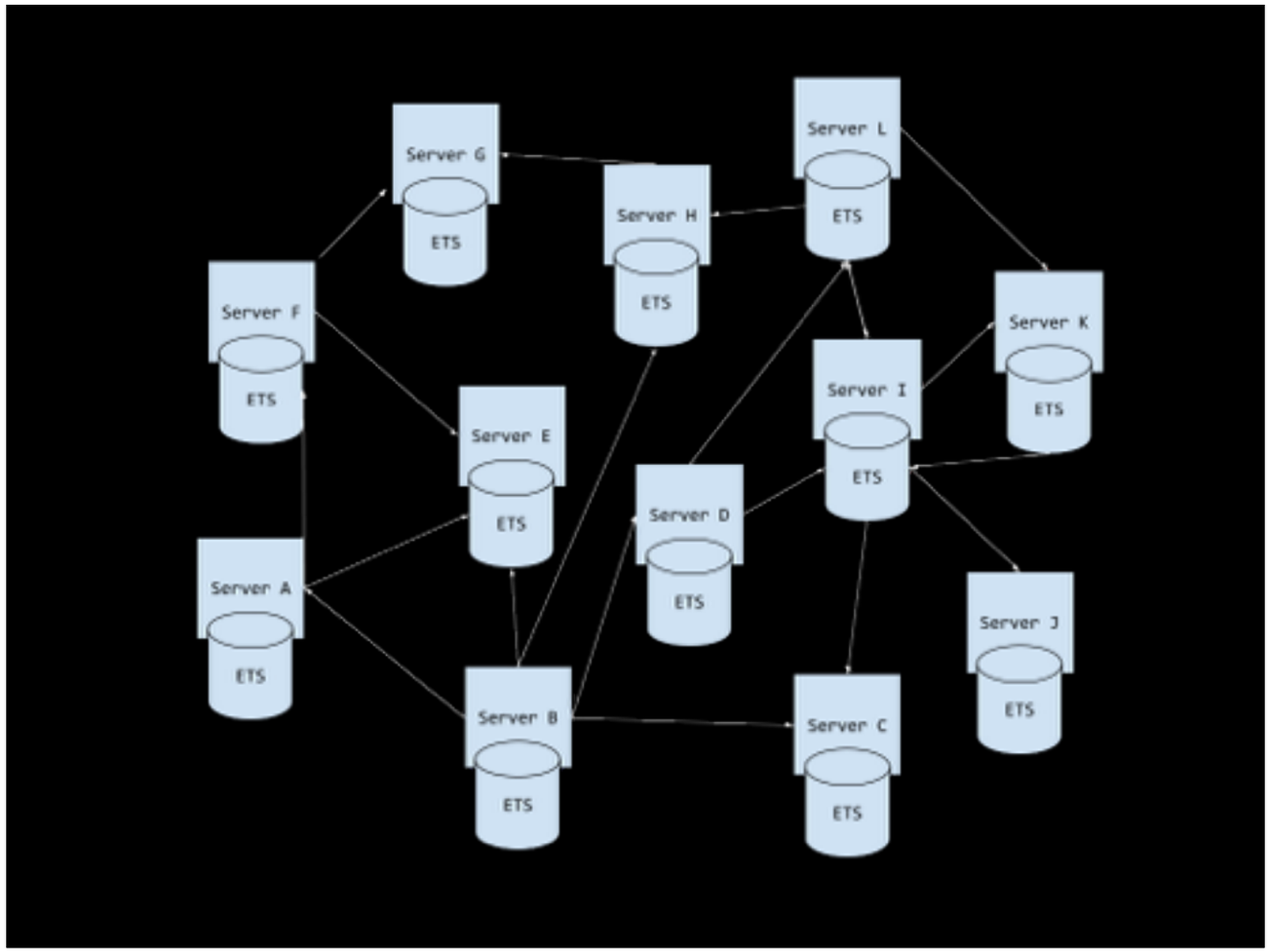
A stateful web app

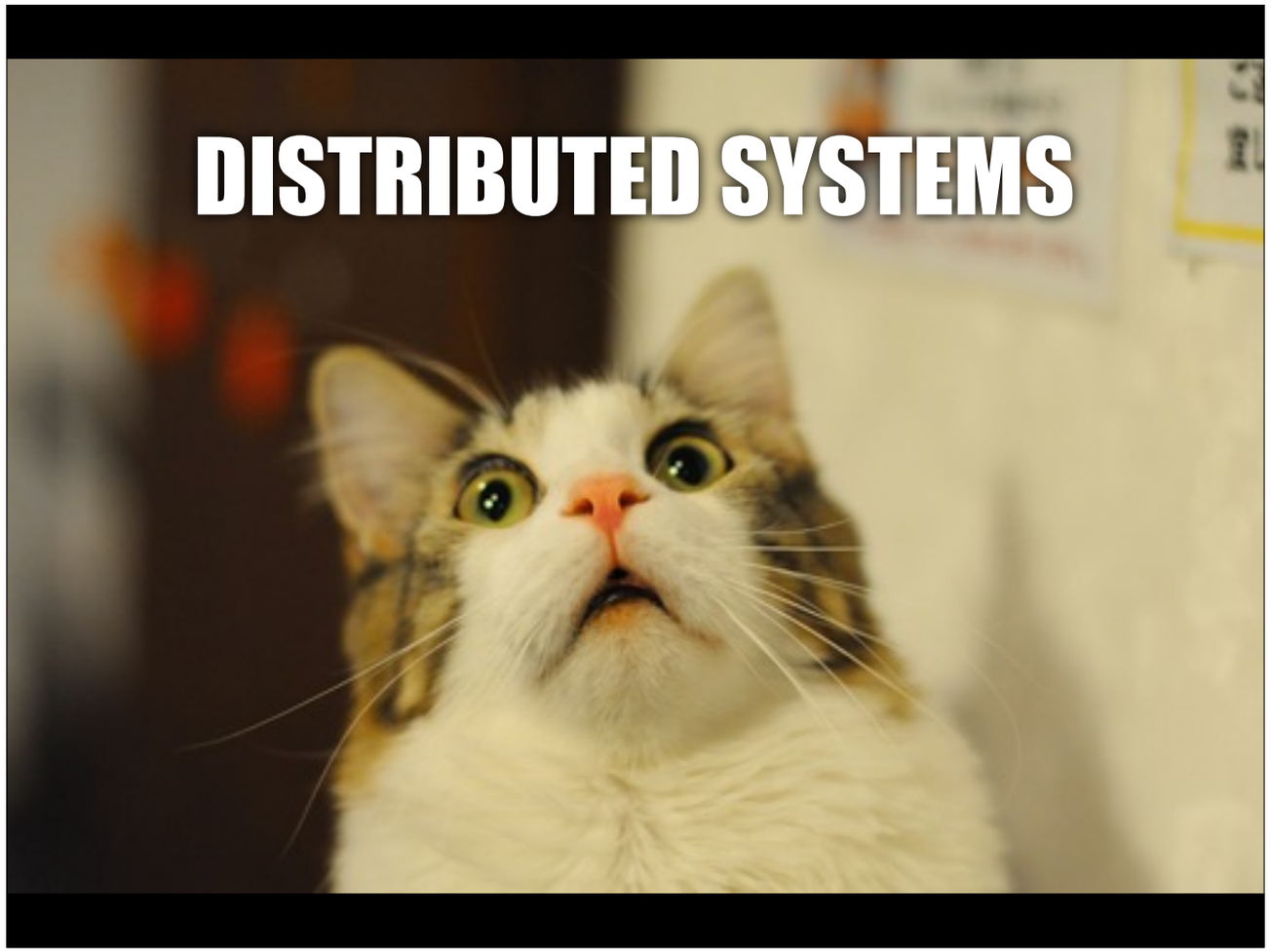


Or maybe use GenServer.call/cast if we’re feeling OTP-ish.

Sweet! All good! We’re done! …How much time do I have left?


ARE YOU SURE?


Server G
ETS
Server L
ETS
Server H
ETS
Server F
ETS
Server K
ETS
Server I
ETS
Server E
ETS
Server D
ETS
Server A
ETS
Server J
ETS
Server B
ETS
Server C
ETS


DISTRIBUTED SYSTEMS

Distributed Primitives and Patterns

Distributed Primitives and Patterns

`send(remote_pid, …), GenServer.call/cast`

distributed systems

Distributed Primitives and Patterns

`send(remote_pid, …), GenServer.call/cast`

distributed systems

≈

Distributed Primitives and Patterns

`send(remote_pid, …), GenServer.call/cast`

distributed systems

≈

`print("<div>")`

web development

Distributed Primitives and Patterns

distributed systems

≈

Distributed Primitives and Patterns

ad hoc message passing or RPC

distributed systems

≈

Distributed Primitives and Patterns

ad hoc message passing or RPC

distributed systems

≈

```
<?php
```

Distributed Primitives and Patterns

ad hoc message passing or RPC

distributed systems

≈

```
<?php
    $foo = mysql_query($my_cool_query);
```

Distributed Primitives and Patterns

ad hoc message passing or RPC

distributed systems

≈

```
<?php
    $foo = mysql_query($my_cool_query);
    echo "<div>$foo</div>";
```

Distributed Primitives and Patterns

ad hoc message passing or RPC

distributed systems

≈

```
<?php
    $foo = mysql_query($my_cool_query);
    echo "<div>$foo</div>";
?>
```

Distributed Primitives and Patterns

ad hoc message passing or RPC

distributed systems

≈

```
<?php
    $foo = mysql_query($my_cool_query);
    echo "<div>$foo</div>";
?>
```

web development

Distributed Primitives and Patterns

```
<?php
    $foo = mysql_query($my_cool_query);
    echo "<div>$foo</div>";
?>
```

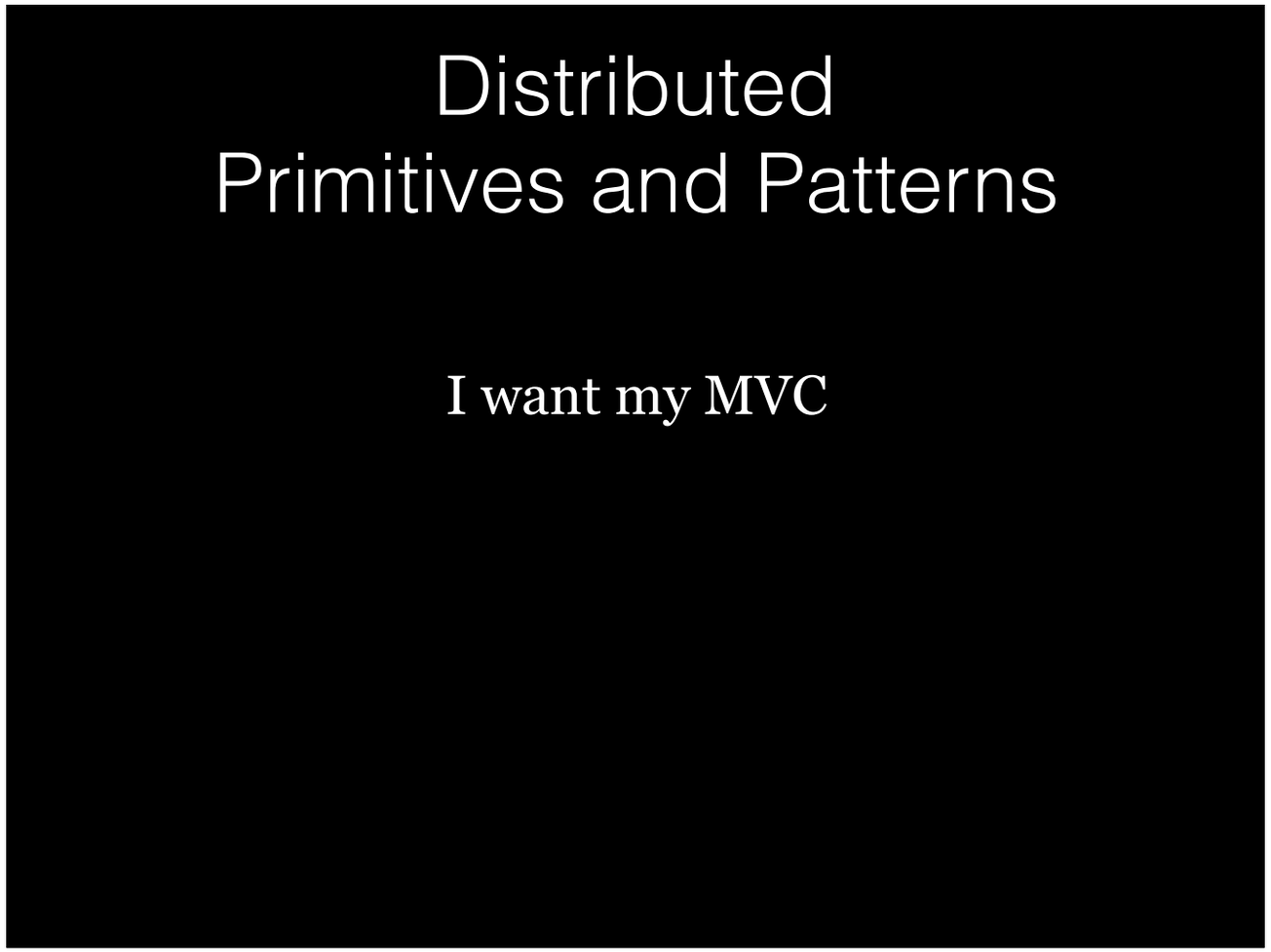
that isn’t to say that this bit of PHP is bad

Distributed Primitives and Patterns

$$$$$$$

```
<?php
    $foo = mysql_query($my_cool_query);
    echo "<div>$foo</div>";
?>
```

"unstructured" application development has generated _enormous_ amounts of value across the world. (hence all the $$ signs) — maybe more than all rails apps combined. everybody has written code to this effect at some point.



but that doesn’t stop us from reaching for more structure — we use approaches like MVC.

Distributed Primitives and Patterns

I want my MVC

what is the MVC of distributed systems?

so what’s our "MVC" — a guiding design principle — when it comes to distributed systems?

Distributed CAP

So the first thing you might think when you’re looking for a handy 3 letter acronym that describes some important principle is CAP. Super sloppy/quick overview

Distributed CAP

Consistency

I ask two servers the same question. Will they give me the same answer? (or will one be out of date, or will they be diverged, etc.)

Distributed CAP

Consistency

Availability

I ask a working server a question. Will it give me a (non-error) response? (in spite of server failures, network partitions, etc.)

Distributed CAP

Consistency

Availability

Partition tolerance

How well do you tolerate arbitrary network splits between members of your system?

Distributed CAP

Consistency

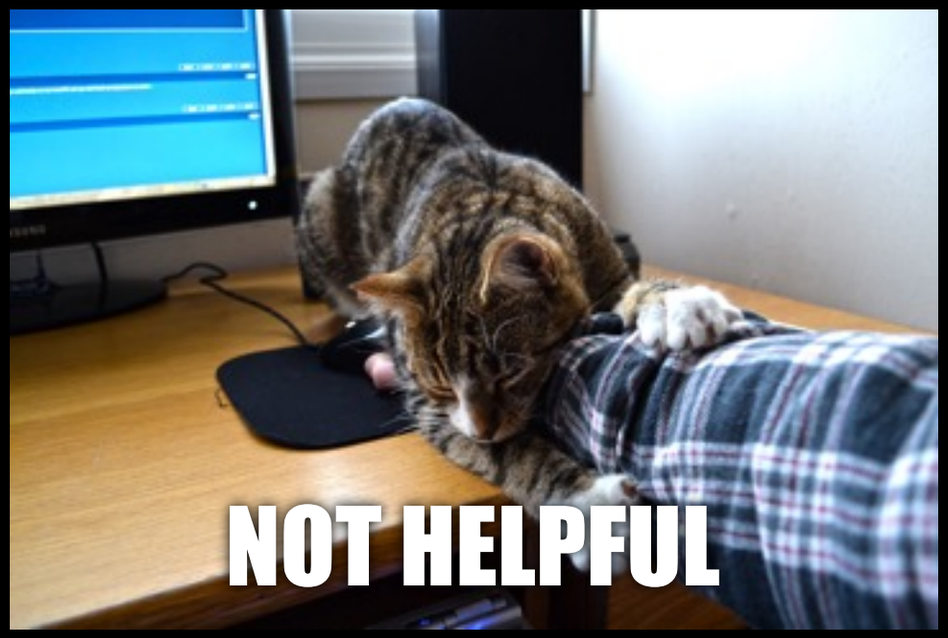
"Pick two"

Availability

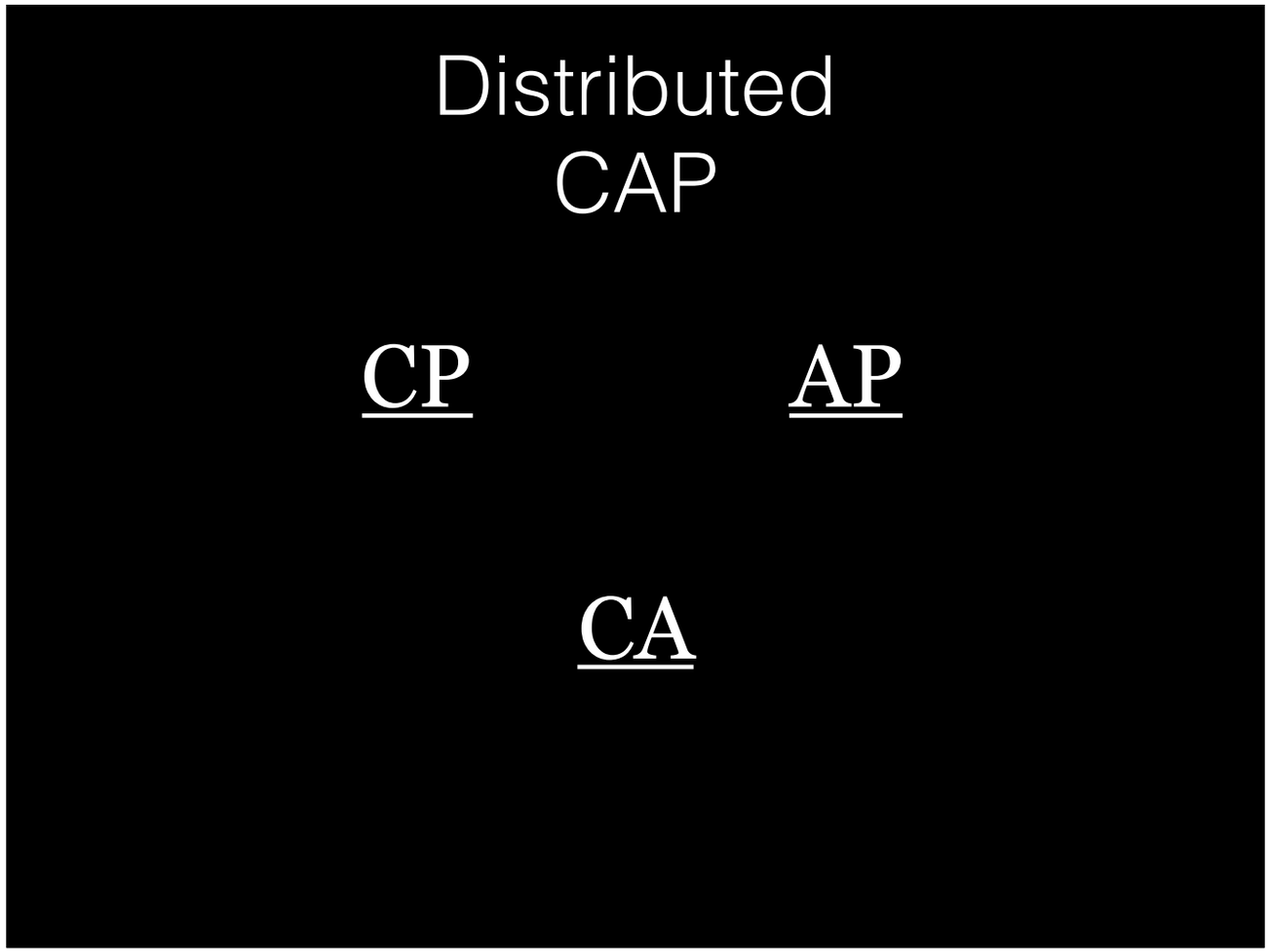
Partition tolerance

Basically. Kind of misleading, since lots of real systems end up with 1 or none, or 0.8/0.8/0.2 (deliberately less than 2, but some of each).

Distributed CAP



CAP doesn’t actually help us design or build much of anything. It doesn’t suggest how to structure applications. It’s not a pattern, like MVC. instead, it helps form the basis for a discussion about tradeoffs in distributed systems



Helps us categorise patterns: we can talk about them in terms of the tradeoffs they make. We've got 'consistent and partition tolerant'

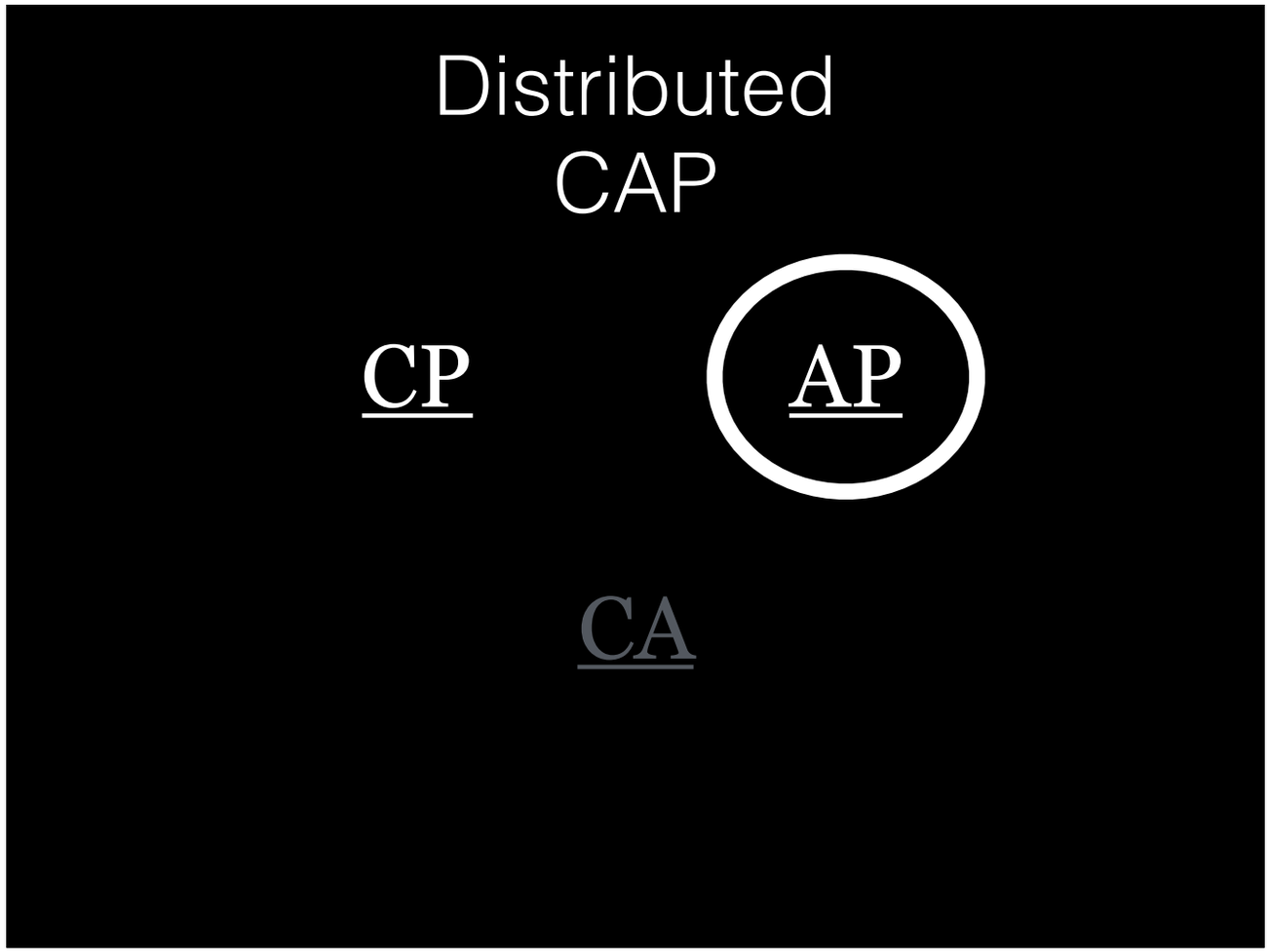
Distributed
CAP

CP

AP

CA

CA is questionable, because partitions are probably always going to happen



Today we’re going to focus on one group of patterns, 'AP'

Distributed
CAP

AP

Why AP? What kind of things are in there?

Distributed
CAP

AP

gossip protocols

Gossip: "psst, I’m server B and I have user 42’s data. if anyone needs it, they should ask me"

Distributed
CAP

AP

gossip protocols

CRDTs

Conflict free replicated data types — Phoenix Presence

Distributed
CAP

AP

gossip protocols

CRDTs

distributed hash tables

distributed hash table: hash some piece of the data, that tells you which server has (or should have) it. That sounds good if our data is bigger than memory on a single box!

Distributed
CAP

AP

gossip protocols

CRDTs

distributed hash tables

distributed hash table: hash some piece of the data, that tells you which server has (or should have) it. That sounds good if our data is bigger than memory on a single box! (this is what memcached does)

Awesome! Let's build a
stateful, distributed web app
using a distributed hash table!

A stateful, distributed web app with a DHT

So I’m going to build a thing, what’s the first step? Reinvent…

A stateful, distributed web app with a DHT

Use a framework, don’t write one

…look for libraries! Just like we shouldn’t re-invent Rails, Django, or Phoenix for every web app, we should rely on existing tools (from Erlang, in this case)

Riak Core

Riak Core

"a toolkit for building distributed, scalable, fault-tolerant applications"

- riak_core README

GET READY TO HYPE HYPE HYPE

Riak Core

seriously advanced technology

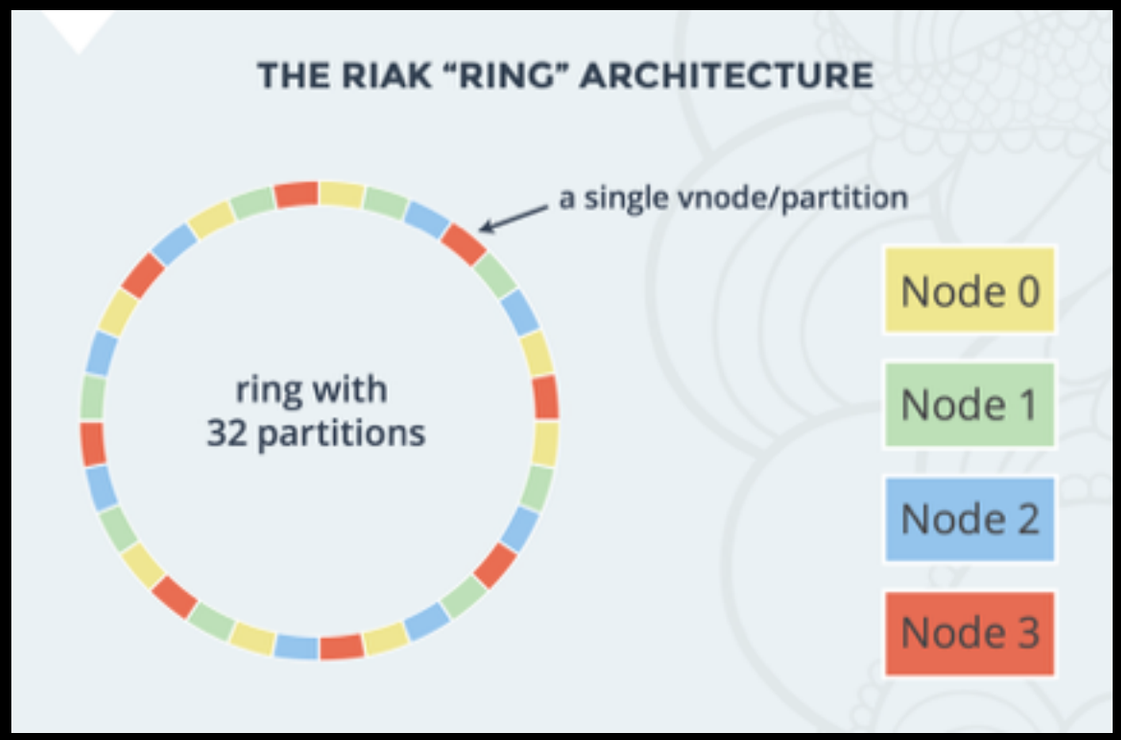
Riak Core

seriously advanced technology

without peer in other platforms

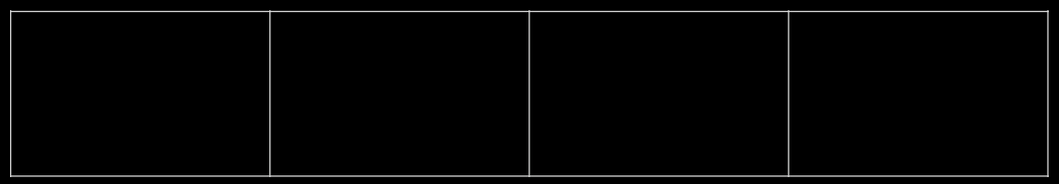
(practically). Akka + some .NET work, but proven maturity/reliability in Riak Core. and in tricky, tricky software like distributed databases, maturity is a HUGE bonus

Riak Core


THE RIAK "RING" ARCHITECTURE
a single vnode/partition
ring with
32 partitions
Node 0
Node 1
Node 2
Node 3

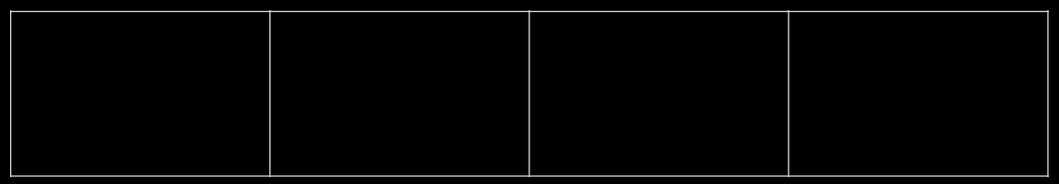
Riak Core
Hash Ring

`my_hash = {}`



Riak Core
Hash Ring

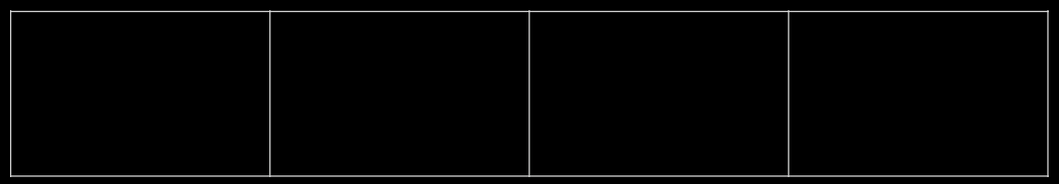
```
my_hash = {}
```



```
my_hash["answer"] = 42
```

Riak Core
Hash Ring

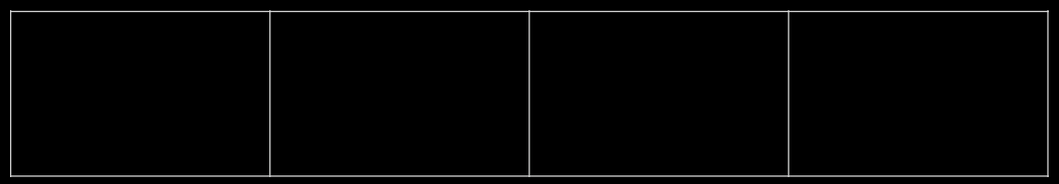
```
my_hash = {}
```



```
my_hash["answer"] = 42
hash("answer") -> 10403
```

Riak Core
Hash Ring

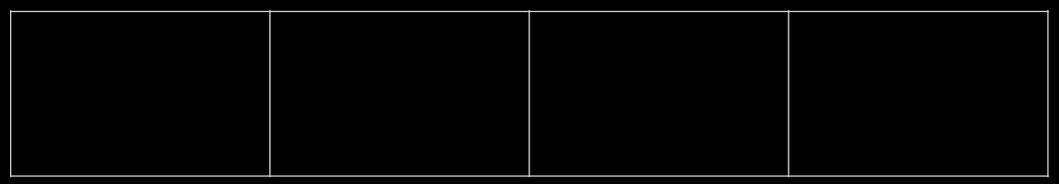
```
my_hash = {}
```



```
my_hash["answer"] = 42
hash("answer") -> 10403
index = 10403 % 4
```

Riak Core
Hash Ring

```
my_hash = {}
```



```
my_hash["answer"] = 42
hash("answer") -> 10403
index = 10403 % 4
# put 42 in index 3!
```

Riak Core
Hash Ring

```
my_hash = {"answer"=>42}
```

			"answer", 42

```
my_hash["answer"] = 42
hash("answer") -> 10403
index = 10403 % 4
# put 42 in index 3!
```

Riak Core
Hash Ring

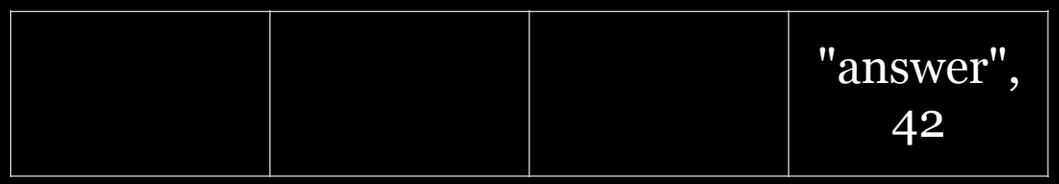
```
my_hash = {"answer"=>42}
```

			"answer", 42

Which server owns which bucket?

Riak Core
Hash Ring

```
my_hash = {"answer"=>42}
```

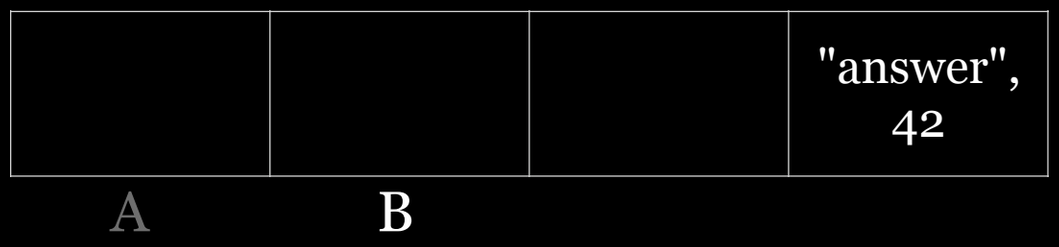


A

Which server owns which bucket?

Riak Core
Hash Ring

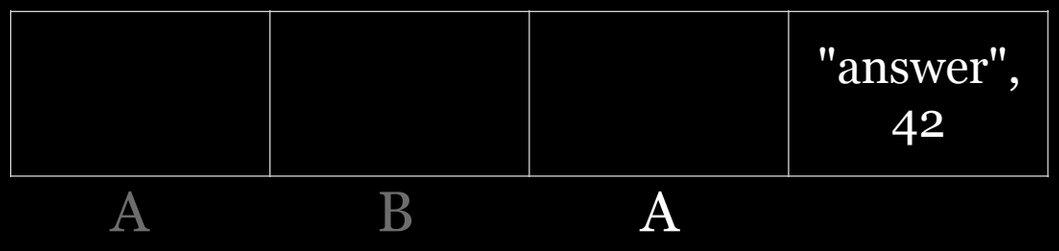
```
my_hash = {"answer"=>42}
```



Which server owns which bucket?

Riak Core
Hash Ring

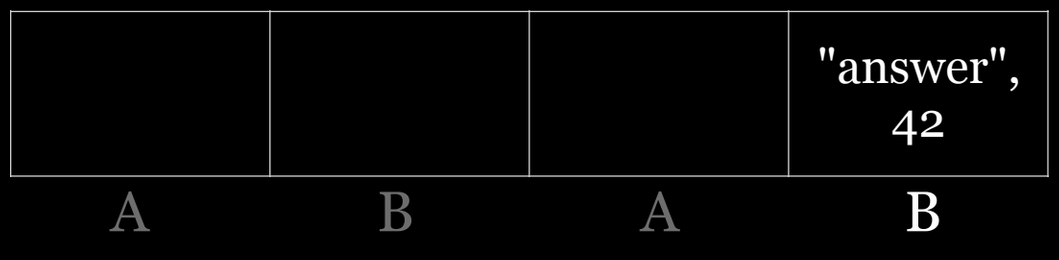
```
my_hash = {"answer"=>42}
```



Which server owns which bucket?

Riak Core
Hash Ring

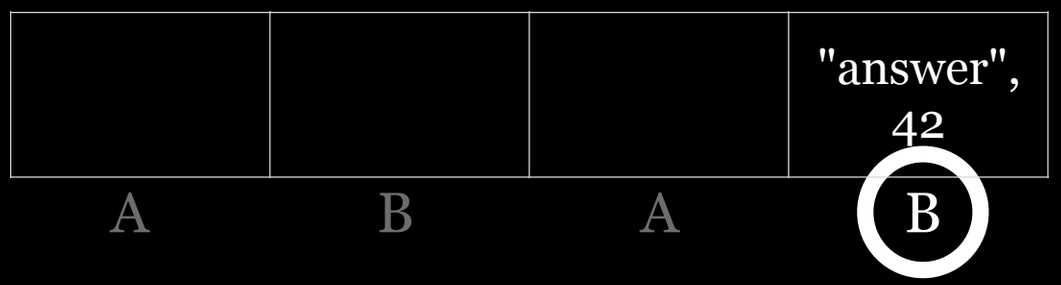
```
my_hash = {"answer"=>42}
```



Which server owns which bucket?

Riak Core
Hash Ring

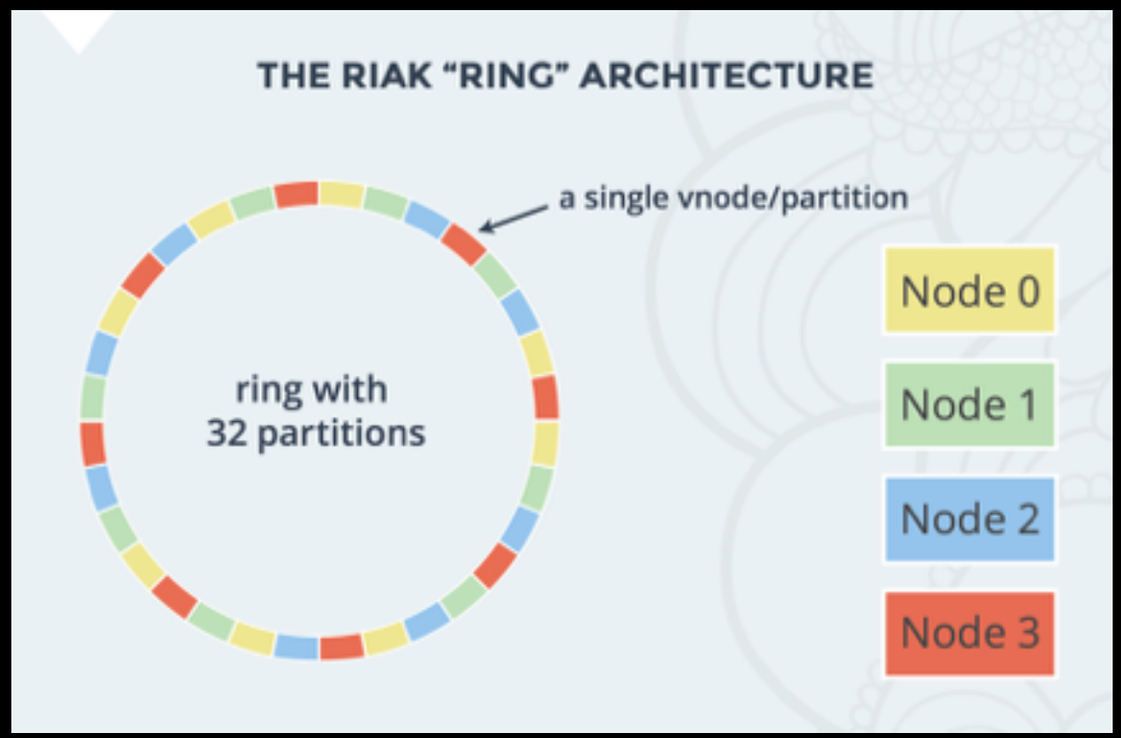
```
my_hash = {"answer"=>42}
```



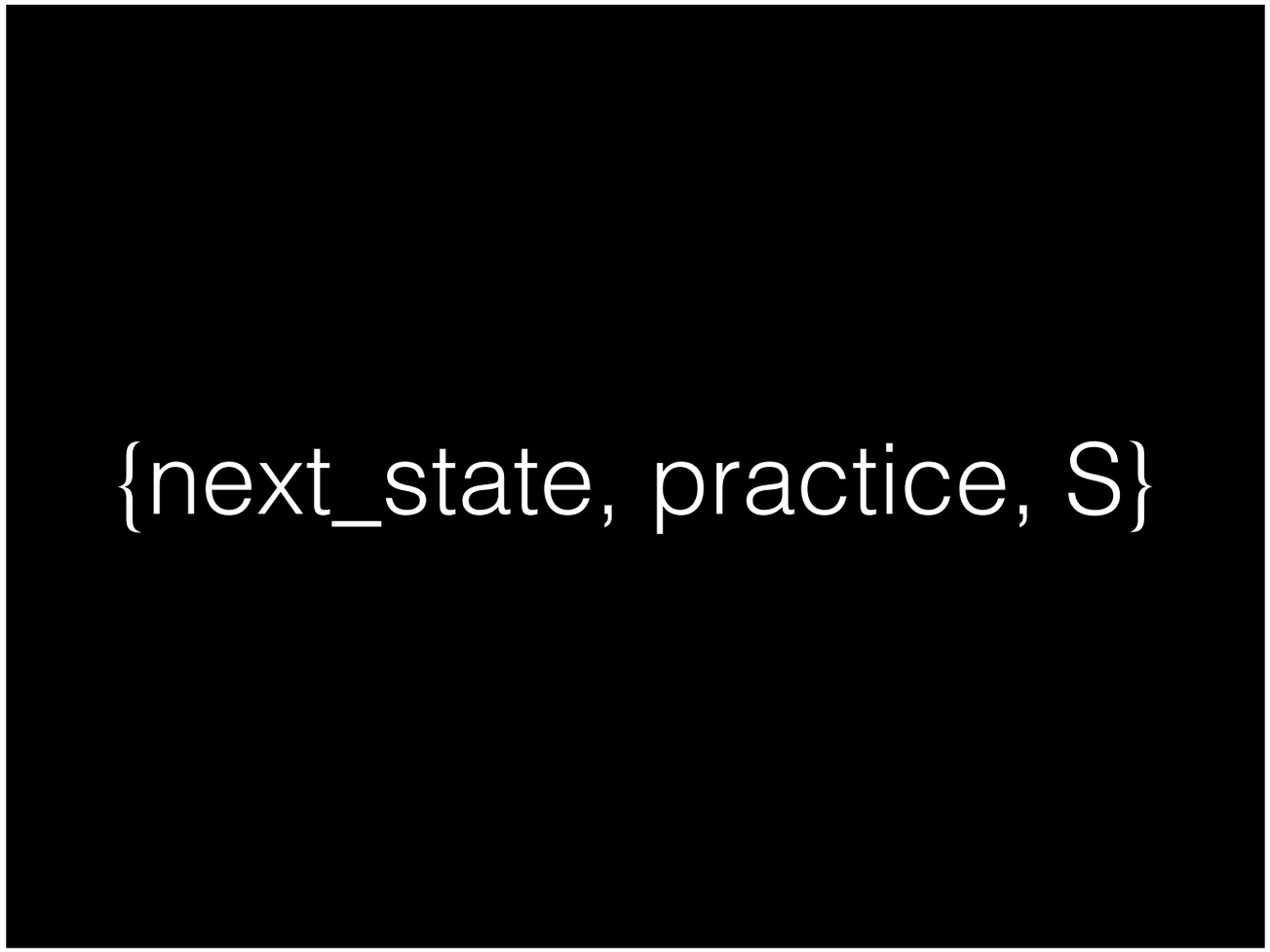
Which server owns which bucket?

So we know that server "B" is the place to go for this data! Nice!

Riak Core



Bucket == vnode. Just like Hash tables have tricks for resizing the array and buckets, riak has tricks for exchanging vnodes from one server to another



check if anyone would like to pair up: if you learn effectively by discussing with a partner/pair programming, go for it. flying solo also ok.

the coding exercises

a note on the coding exercises — I've set it up so that you can either build up a single project over the course of the workshop, or start fresh for each exercise and tinker with the code in an existing codebase. this is totally up to you — do whatever suits your learning style best. If you're pairing, I suggest starting with YakDB and building up from there so that it's easier for everyone to keep track of the current codebase.

the coding exercises

- Build up from scratch

a note on the coding exercises — I've set it up so that you can either build up a single project over the course of the workshop, or start fresh for each exercise and tinker with the code in an existing codebase. this is totally up to you — do whatever suits your learning style best. If you're pairing, I suggest starting with YakDB and building up from there so that it's easier for everyone to keep track of the current codebase.

the coding exercises

- Build up from scratch
- Tinker with existing code

a note on the coding exercises — I've set it up so that you can either build up a single project over the course of the workshop, or start fresh for each exercise and tinker with the code in an existing codebase. this is totally up to you — do whatever suits your learning style best. If you're pairing, I suggest starting with YakDB and building up from there so that it's easier for everyone to keep track of the current codebase.

the coding exercises

- Build up from scratch
- Tinker with existing code
- pair?

a note on the coding exercises — I've set it up so that you can either build up a single project over the course of the workshop, or start fresh for each exercise and tinker with the code in an existing codebase. this is totally up to you — do whatever suits your learning style best. If you're pairing, I suggest starting with YakDB and building up from there so that it's easier for everyone to keep track of the current codebase.

https://github.com/
kanatohodets/
riak_core_workshop_euc2016

Exercise 0:
build a cluster

```
(zap_chat2@127.0.0.1)18> riak_core_console:member_status([]).
================================ Membership ==========================
Status     Ring    Pending    Node
------------------------------------------------------------------------
valid      12.5%     50.0%    'zap_chat2@127.0.0.1'
valid      87.5%     50.0%    'zap_chat3@127.0.0.1'
------------------------------------------------------------------------
Valid:2 / Leaving:0 / Exiting:0 / Joining:0 / Down:0
ok
```

our first app

A Riak Core App: simple_hash - the app

A Riak Core App: simple_hash - the app

ping -> pong

A Riak Core App: simple_hash - the app

ping -> pong

hash(timestamp) -> vnode (bucket)

A Riak Core App: simple_hash - the app

ping -> pong

hash(timestamp) -> vnode (bucket)

example of distributing CPU work

A Riak Core App: simple_hash - the parts

A Riak Core App: simple_hash - the parts

service (high level API)

service: high level API for working with your ring application. our "Ping" public function will live here

A Riak Core App: simple_hash - the parts

service (high level API)

vnode (business logic)

process that represents part of the hash ring: a hash bucket with code attached. where rubber -> road: the functionality for "ping" will be here as a callback pattern match.

A Riak Core App: simple_hash - the parts

service (high level API)

vnode (business logic)

(supervisor)

does normal supervisor stuff, starts 'vnode_master' which manages the flock of vnode processes. not that interesting for us today

A Riak Core App: simple_hash - the parts

service (high level API)

vnode (business logic)

(supervisor)

(application)

starts the supervisor, registers vnode module with riak_core, registers the service with riak_core. not that interesting.

A Riak Core App: simple_hash - the parts

service (high level API)

vnode (business logic)

(supervisor)

(application)

let’s look at the service!

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do









  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx =                  hash_key(
                                                )
  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx =                    hash_key(
                                        :os.timestamp  )









  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
                                        :os.timestamp  )









  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})









  end
end
```

output of my hash function

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list =









  end
end
```

list of buckets (vnodes) that are indicated by this hash output (just like we transformed the hash of "answer" into a bucket index in our regular hash table)

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list =                get_primary_apl(
      doc_idx,                      )









  end
end
```

the list of buckets (vnodes) is called the active preference list

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list =                get_primary_apl(
      doc_idx,    SimpleHash.Service)




  end
end
```

service discovery!

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list =                get_primary_apl(
      doc_idx, 1, SimpleHash.Service)




  end
end
```

we only want to send a command to 1 vnode

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)




  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list



  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list

                              sync_spawn_command(
      index_node,                                 )
  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list

                              sync_spawn_command(
      index_node, :ping,                        )
  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list

                                sync_spawn_command(
      index_node, :ping, SimpleHash.Vnode_master)
  end
end
```

named process. has a funny name because

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list
    # riak core appends "_master" to SimpleHash.Vnode.
                              sync_spawn_command(
      index_node, :ping, SimpleHash.Vnode_master)
  end
end
```

of this

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list
    # riak core appends "_master" to SimpleHash.Vnode.
    :riak_core_vnode_master.sync_spawn_command(
      index_node, :ping, SimpleHash.Vnode_master)
  end
end
```

A Riak Core App: simple_hash - service

```
defmodule SimpleHash.Service do
  def ping do
    doc_idx = :riak_core_util.chash_key(
      {"ping", :erlang.term_to_binary(:os.timestamp)})

    pref_list = :riak_core_apl.get_primary_apl(
      doc_idx, 1, SimpleHash.Service)

    [{index_node, _type}] = pref_list
    # riak core appends "_master" to SimpleHash.Vnode.
    :riak_core_vnode_master.sync_spawn_command(
      index_node, :ping, SimpleHash.Vnode_master)
  end
end
```

A Riak Core App: simple_hash - the parts

service (high level API)

vnode (business logic)

(supervisor)

(application)

now for vnodes!

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do









end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode




end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup









end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init(      ), do: {:ok, %{          }}









end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([part]), do: {:ok, %{part: part}}









end
```

partition == bucket identifier. the portion of the number range in the hash output that starts with this number: this vnode owns it

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([part]), do: {:ok, %{part: part}}

  def handle_command(

  ) do

  end

end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([part]), do: {:ok, %{part: part}}

  def handle_command(
    :ping, _sender, %{part: part} = state
  ) do

  end

end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([part]), do: {:ok, %{part: part}}

  def handle_command(
    :ping, _sender, %{part: part} = state
  ) do
    {:reply, {:pong, part}, state}
  end

end
```

A Riak Core App: simple_hash - vnode

```
defmodule SimpleHash.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([part]), do: {:ok, %{part: part}}

  def handle_command(
    :ping, _sender, %{part: part} = state
  ) do
    {:reply, {:pong, part}, state}
  end
  # … other callbacks for :riak_core_vnode
end
```

Exercise 1
Ping 2

Add a second "ping" method that uses something other than timestamp for the hash input. Perhaps have it return information like what node that vnode is running on.

What about…state?

A Riak Core App:
silly_kv - the app

A Riak Core App: silly_kv - the app

store(key, data)

A Riak Core App: silly_kv - the app

store(key, data)

store in ETS

A Riak Core App: silly_kv - the app

store(key, data)

store in ETS

hash(key) -> vnode

A Riak Core App: silly_kv - service

```
def store(key, data) do




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx =                    hash_key(
                                        )
end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx =                    hash_key(
                                        key  )




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
                                        key  )




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})









end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list =                  get_primary_apl(
    doc_idx,                     )




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list =                get_primary_apl(
    doc_idx,    SillyKV.Service)




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list =                get_primary_apl(
    doc_idx, 1, SillyKV.Service)




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list = :riak_core_apl.get_primary_apl(
    doc_idx, 1, SillyKV.Service)




end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list = :riak_core_apl.get_primary_apl(
    doc_idx, 1, SillyKV.Service)

  [{index_node, _type}] = pref_list



end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list = :riak_core_apl.get_primary_apl(
    doc_idx, 1, SillyKV.Service)

  [{index_node, _type}] = pref_list
                          sync_spawn_command(
    index_node,
                        )
end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list = :riak_core_apl.get_primary_apl(
    doc_idx, 1, SillyKV.Service)

  [{index_node, _type}] = pref_list
                          sync_spawn_command(
    index_node, {:store, key, data},
                        )
end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list = :riak_core_apl.get_primary_apl(
    doc_idx, 1, SillyKV.Service)

  [{index_node, _type}] = pref_list
                          sync_spawn_command(
    index_node, {:store, key, data},
    SillyKV.Vnode_master)
end
```

A Riak Core App: silly_kv - service

```
def store(key, data) do
  doc_idx = :riak_core_util.chash_key(
    {"store", :erlang.term_to_binary(key)})

  pref_list = :riak_core_apl.get_primary_apl(
    doc_idx, 1, SillyKV.Service)

  [{index_node, _type}] = pref_list
  :riak_core_vnode_master.sync_spawn_command(
    index_node, {:store, key, data},
    SillyKV.Vnode_master)
end
```

A Riak Core App: silly_kv - vnode

```
defmodule SillyKV.Vnode do









end
```

A Riak Core App: silly_kv - vnode

```
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup









end
```

A Riak Core App: silly_kv - vnode

```
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([_part]) do


  end









end
```

A Riak Core App: silly_kv - vnode

```
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([_part]) do
    ets_handle = :ets.new(nil, [])
    {:ok, %{db: ets_handle}}
  end









end
```

A Riak Core App: silly_kv - vnode

```elixir
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([_part]) do
    ets_handle = :ets.new(nil, [])
    {:ok, %{db: ets_handle}}
  end
  def handle_command(

  ) do


  end

end
```

A Riak Core App: silly_kv - vnode

```
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([_part]) do
    ets_handle = :ets.new(nil, [])
    {:ok, %{db: ets_handle}}
  end
  def handle_command(
    {:store, key, data}, _sender, %{db: db} = state
  ) do


  end

end
```

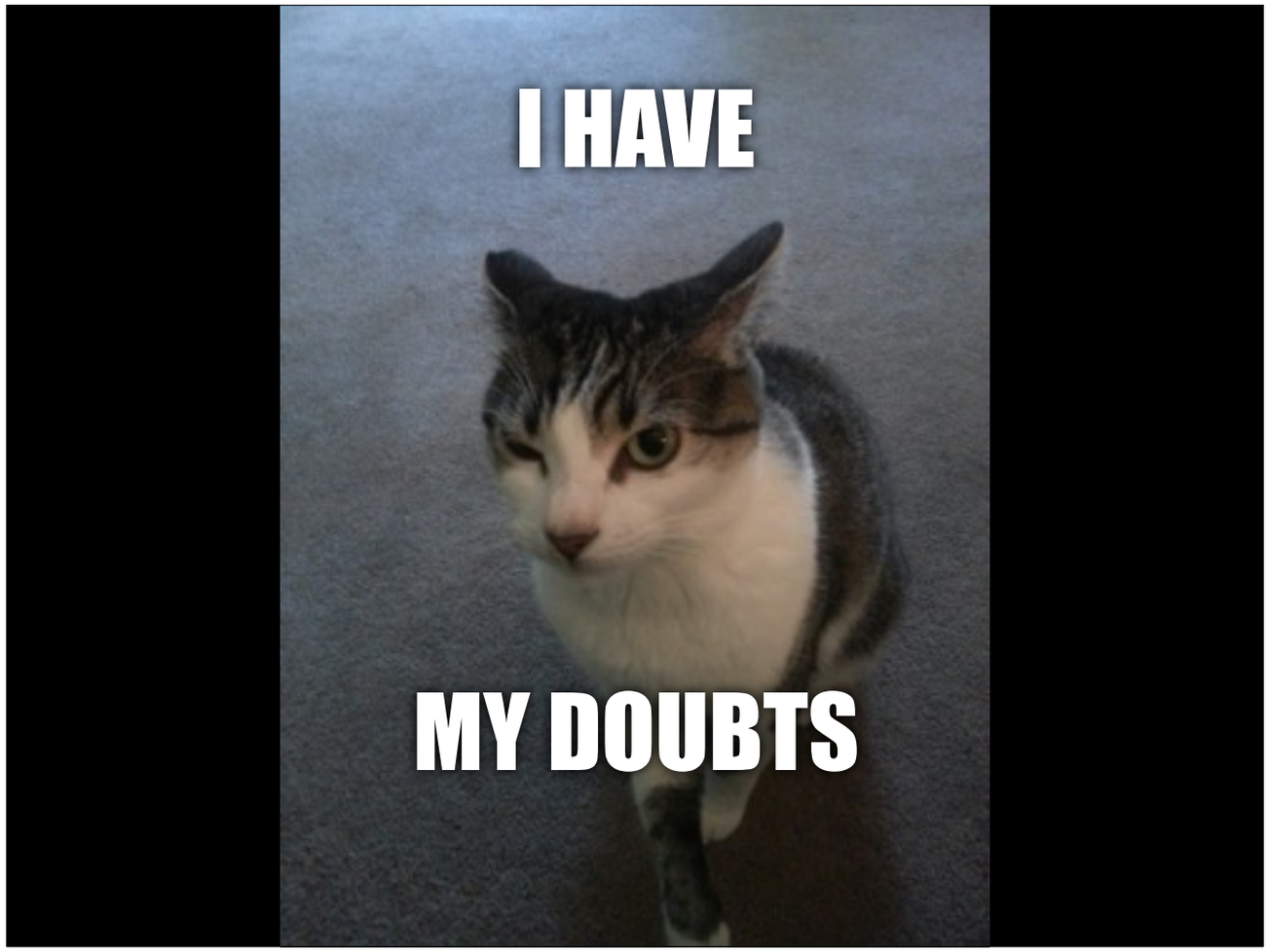
A Riak Core App: silly_kv - vnode

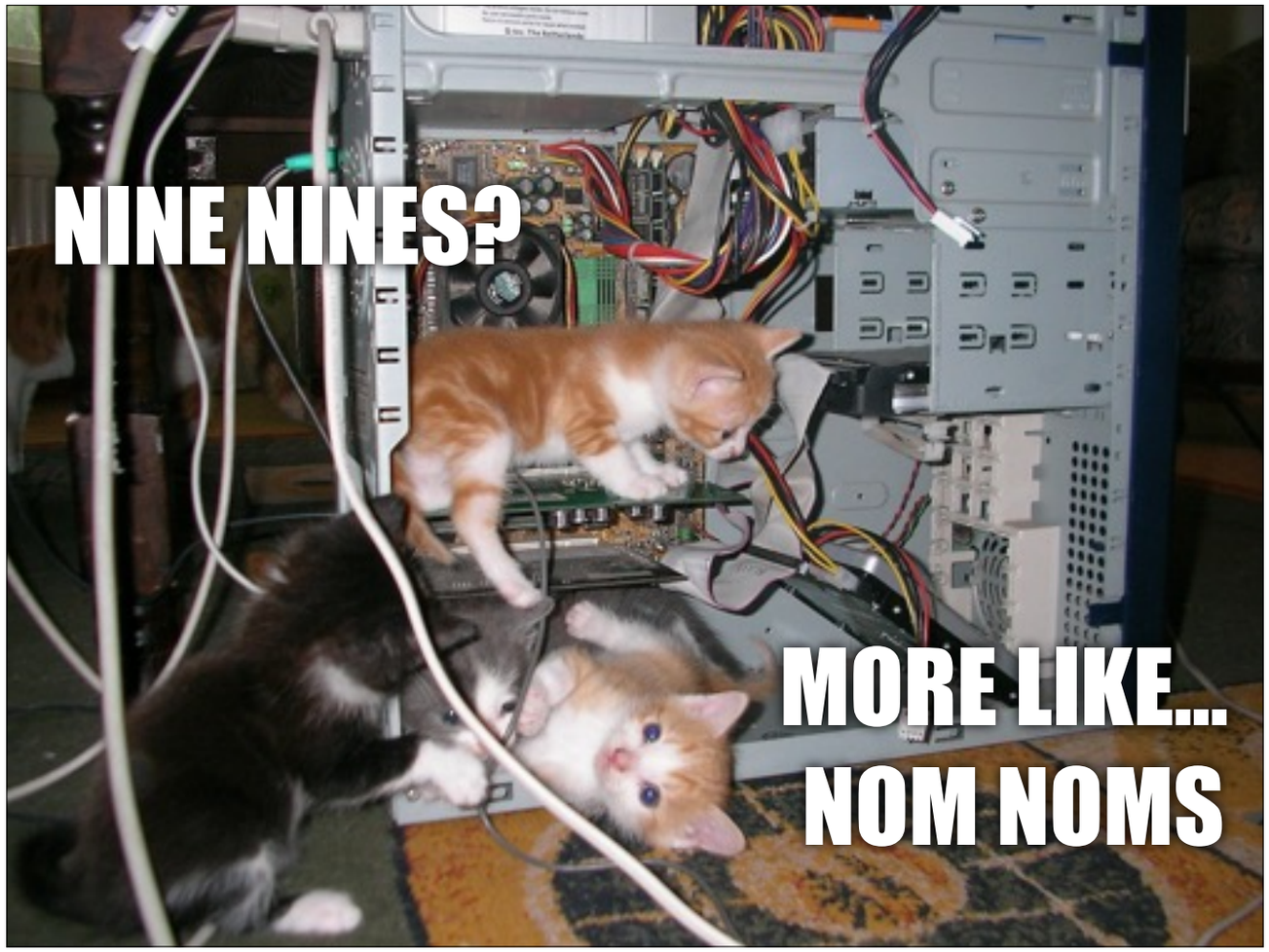
```
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([_part]) do
    ets_handle = :ets.new(nil, [])
    {:ok, %{db: ets_handle}}
  end
  def handle_command(
    {:store, key, data}, _sender, %{db: db} = state
  ) do
    result = :ets.insert(db, {key, data})
    {:reply, result, state}
  end

end
```

A Riak Core App: silly_kv - vnode

```
defmodule SillyKV.Vnode do
  @behaviour :riak_core_vnode
  # … some boilerplate for startup
  def init([_part]) do
    ets_handle = :ets.new(nil, [])
    {:ok, %{db: ets_handle}}
  end
  def handle_command(
    {:store, key, data}, _sender, %{db: db} = state
  ) do
    result = :ets.insert(db, {key, data})
    {:reply, result, state}
  end
  # … same for fetch, but :ets.lookup instead
end
```

Exercise 2
basic KV store/fetch


I HAVE
MY DOUBTS



Because all data is written to a single vnode, _any_ server crash or failure in the cluster will result in permanent data loss!

Fault Tolerance

Computers needed for fault tolerance?

Fault Tolerance

Computers needed for fault tolerance?

> 1

A Riak Core App: coordinated_kv - the parts

service

vnode

(supervisor)

(application)

adding a few new components to our application

A Riak Core App: coordinated_kv - the parts

service

vnode

(supervisor)

(application)

op coordinator (plus supervisor)

a op coordinator, plus its supervisor. this is custom code (not provided by riak core), but there’s a straightforward pattern. its job is to ensure that data is written to multiple vnodes.

it gets executed on multiple vnodes. that way, if some machines fail, you still have the data. if our application required it, we might also have a 'read' coordinator

The Op Coordinator

The Op Coordinator

executes commands on multiple vnodes

store data in multiple places — but how do I know these are on different computers?

The Op Coordinator

executes commands on multiple vnodes

riak_core takes care of spreading vnodes across servers

within reason: generally the advice is to have at least 5 members in your riak_core cluster, so that there's a good spread of vnodes across physical servers

The Op Coordinator

executes commands on multiple vnodes

riak_core takes care of spreading vnodes across servers

(as much as possible)

within reason: generally the advice is to have at least 5 members in your riak_core cluster, so that there's a good spread of vnodes across physical servers

Using the Op Coordinator

```
defmodule CoordKV.Service do



end
```

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data      ) do

  end

end
```

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do




  end

end
```

introducing ’n’ and 'w'. N is the number of vnodes we should store this data on; 'w' is the number of vnodes who must acknowledge before we consider this write successful. this is how we get our fault tolerance!

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do
    {:ok, req_id} = CoordKV.WCoord.do(
                                      )









  end

end
```

in order to make this work, we can’t do this action synchronously; we need to have some identifier for the operation. req_id

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do
    {:ok, req_id} = CoordKV.WCoord.do(
      key, {:store, key, data}, n, w)









  end

end
```

our "command" to the vnode has all the parts it needs to store this data

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do
    {:ok, req_id} = CoordKV.WCoord.do(
      key, {:store, key, data}, n, w)

    receive do


    after


    end
  end

end
```

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do
    {:ok, req_id} = CoordKV.WCoord.do(
      key, {:store, key, data}, n, w)

    receive do


    after
      5000 ->
        {:error, :timeout}
    end
  end

end
```

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do
    {:ok, req_id} = CoordKV.WCoord.do(
      key, {:store, key, data}, n, w)

    receive do
      {^req_id, value} ->
        {:ok, value}
    after
      5000 ->
        {:error, :timeout}
    end
  end

end
```

Using the Op Coordinator

```
defmodule CoordKV.Service do
  def store(key, data, n, w) do
    {:ok, req_id} = CoordKV.WCoord.do(
      key, {:store, key, data}, n, w)

    receive do
      {^req_id, value} ->
        {:ok, value}
    after
      5000 ->
        {:error, :timeout}
    end
  end
  # … fetch implementation
end
```

The Op Coordinator

The Op Coordinator

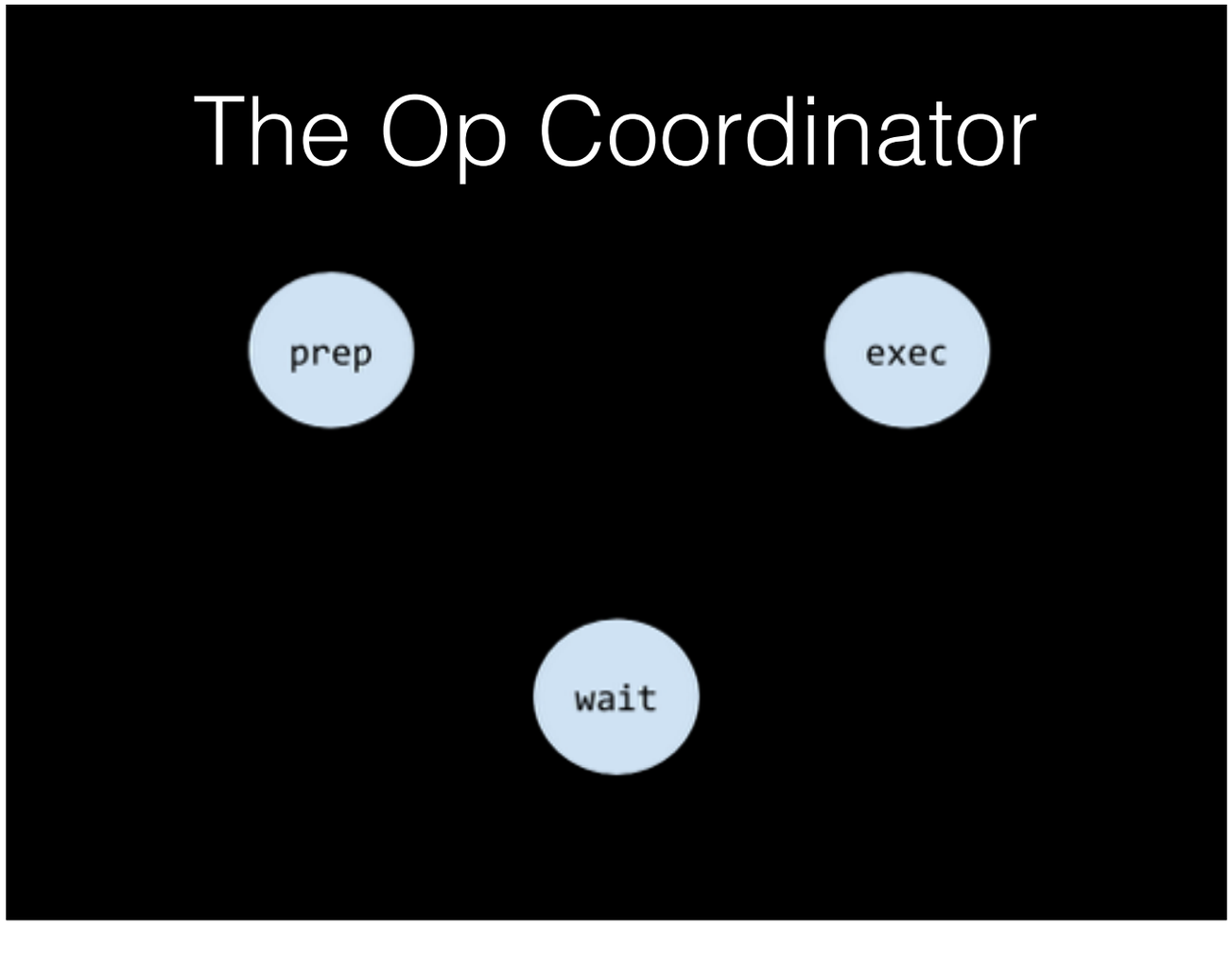
:gen_fsm

The Op Coordinator

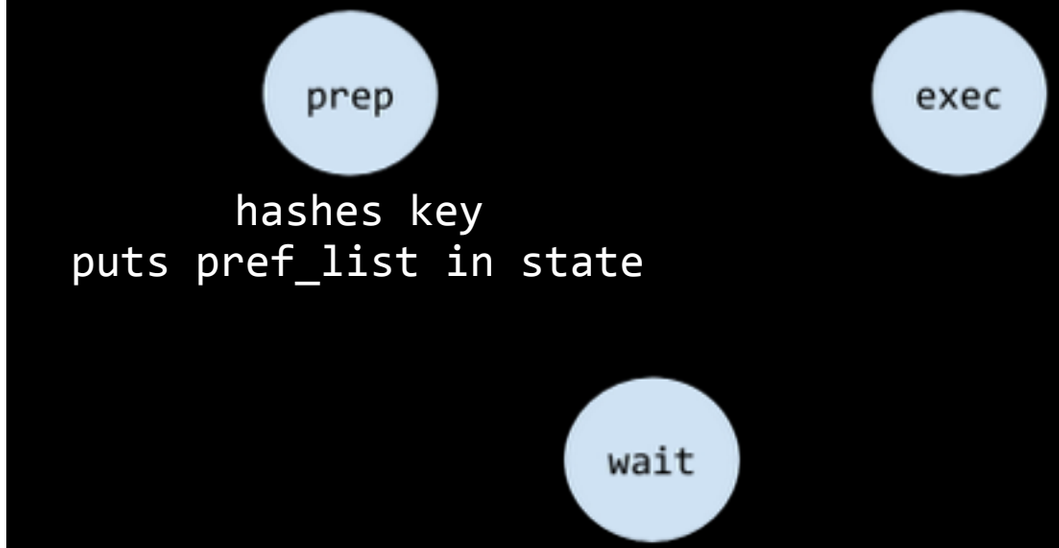
:gen_fsm

spawned on demand (:simple_one_for_one)

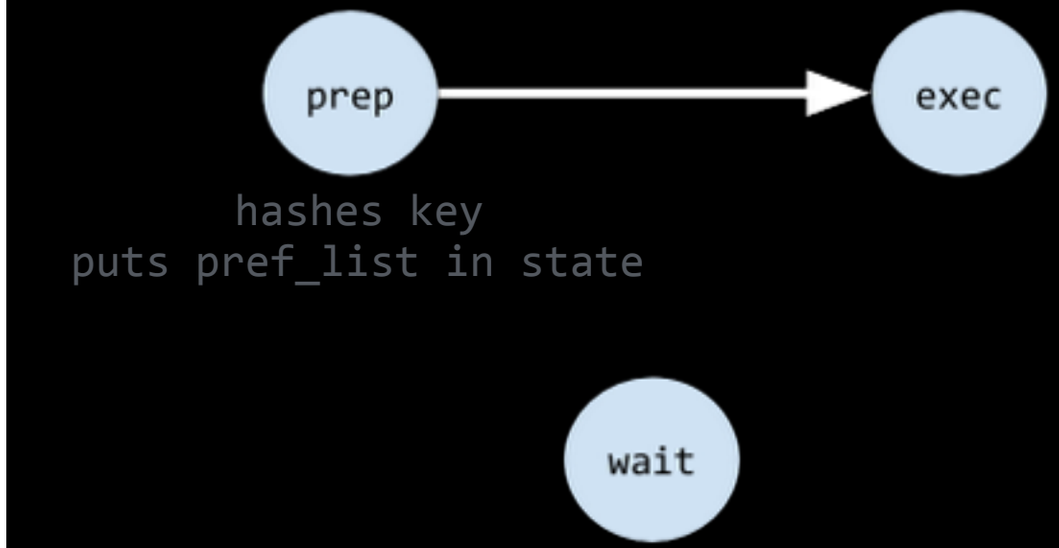
The Op Coordinator



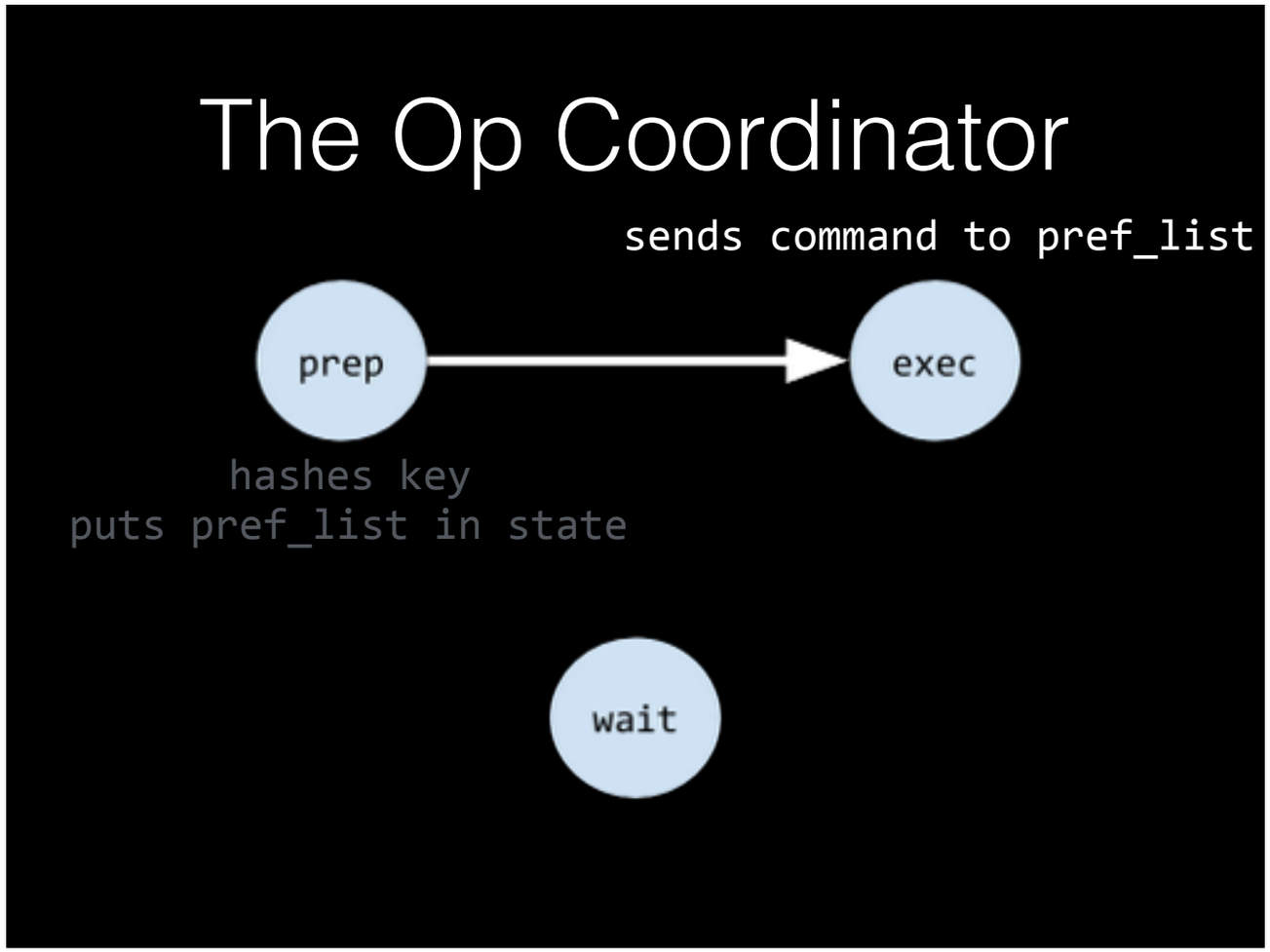
The Op Coordinator



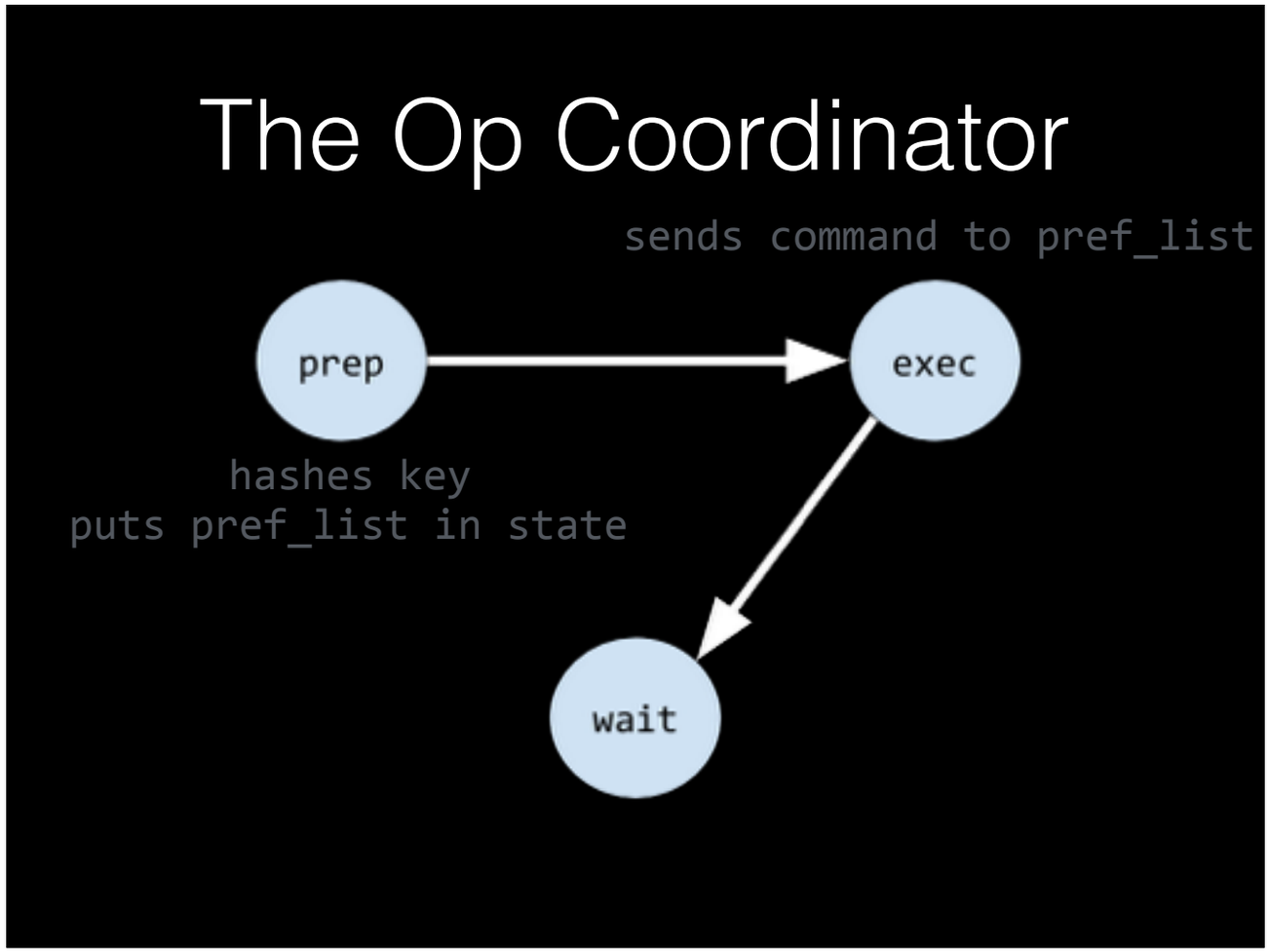
The Op Coordinator



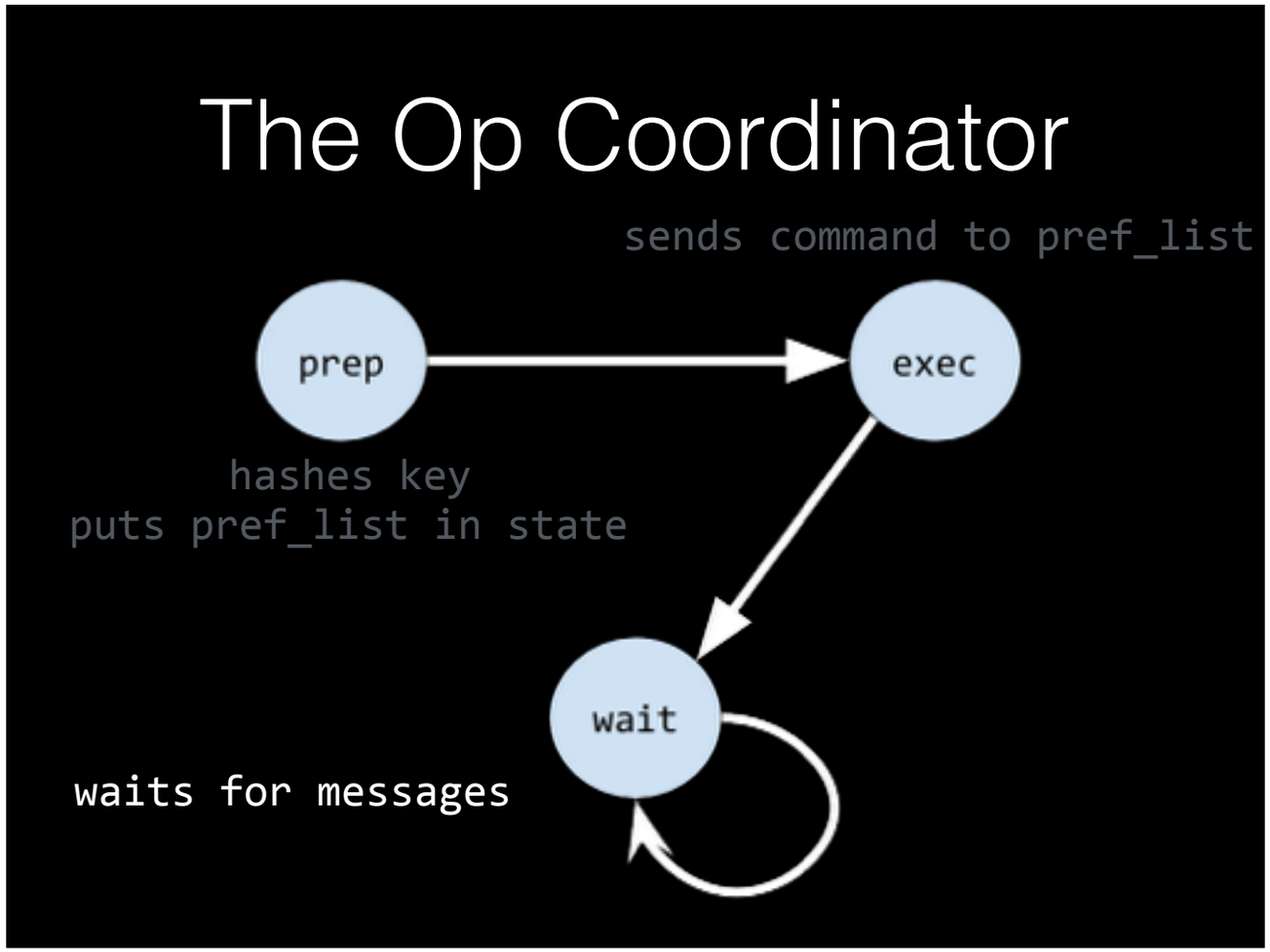
The Op Coordinator



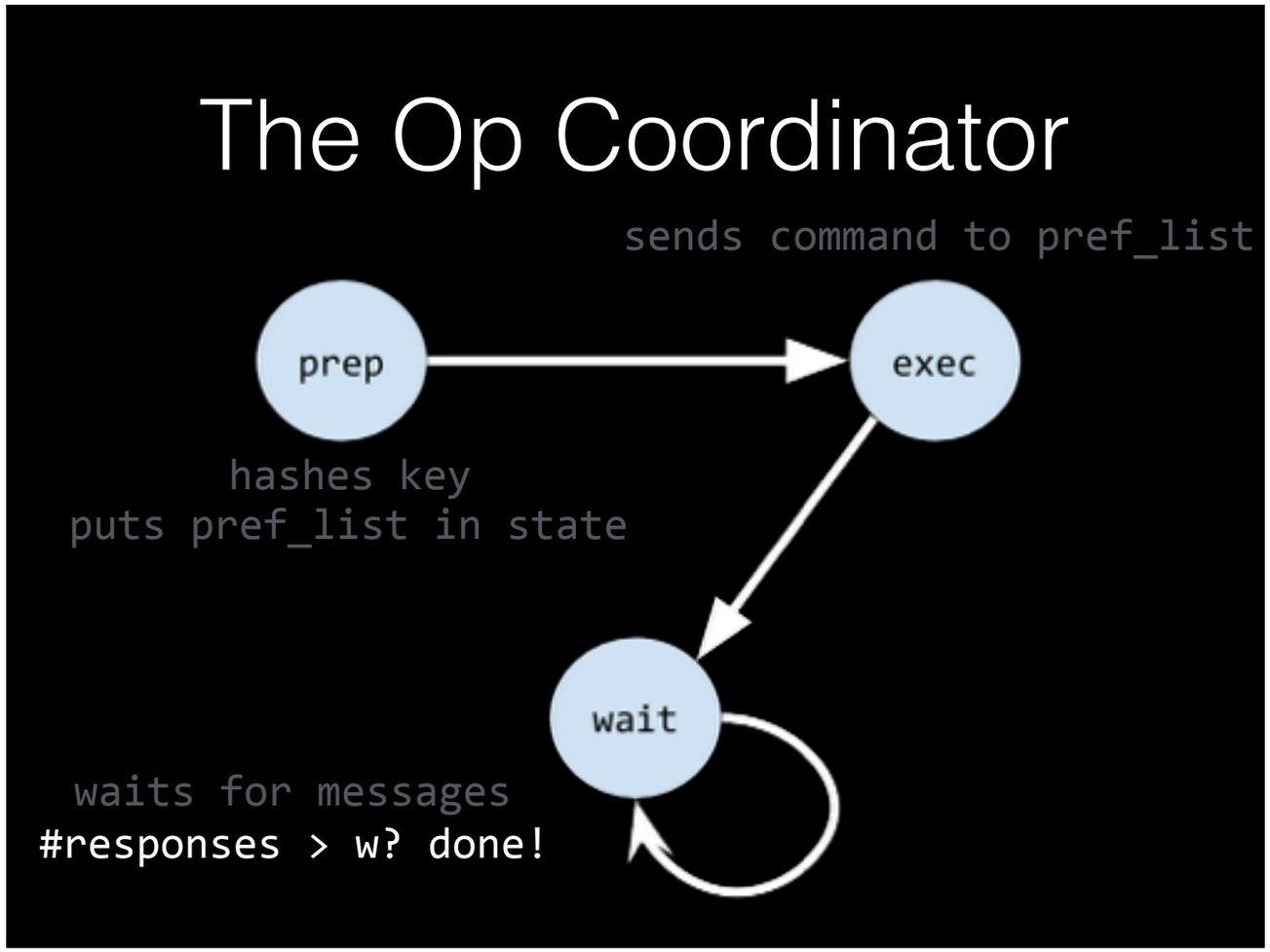
The Op Coordinator



The Op Coordinator



The Op Coordinator



Exercise 3
op coordinator

study code, play with various values of N, W, R. implement read?

how about scaling?

Scaling/Fault Tolerance handoff

Scaling/Fault Tolerance handoff

what if we add (or lose) servers?

Scaling/Fault Tolerance handoff

what if we add (or lose) servers?

"handing off" responsibility for vnode

Scaling/Fault Tolerance handoff

what if we add (or lose) servers?

"handing off" responsibility for vnode

series of callbacks in Vnode module

Scaling/Fault Tolerance handoff

what if we add (or lose) servers?

"handing off" responsibility for vnode

series of callbacks in Vnode module

mostly a matter of serialisation

Exercise 4
handoff

implement/study code. trigger it by adding/removing nodes from cluster, add print statements as desired.

putting the "service" in
"stateful service"

Phoenix + Riak Core

Phoenix + Riak Core

just use an umbrella!

Phoenix + Riak Core

just use an umbrella!

then use the Service API in your Phoenix app somewhere

Phoenix + Riak Core

just use an umbrella!

then use the Service API in your Phoenix app somewhere

```
scope "/api", MyApp do
    pipe_through :api
    put "/store/:key", StoreController, :store
    get "/store/:key", StoreController, :fetch
end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
```

```
defmodule MyApp.StoreController do
  use Phoenix.Controller
  use MyApp.Web, :controller









end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(


    ) do



    end

  end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(
      %Plug.Conn{body_params: data}=conn,

    ) do



    end

  end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(
      %Plug.Conn{body_params: data}=conn,
      %{"key" => key}=params
    ) do



    end

  end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(
      %Plug.Conn{body_params: data}=conn,
      %{"key" => key}=params
    ) do
      n = 3


    end

  end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
```

```
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(
      %Plug.Conn{body_params: data}=conn,
      %{"key" => key}=params
    ) do
      n = 3
      result = HandoffKV.Service.store(key, data, n)

    end

  end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(
      %Plug.Conn{body_params: data}=conn,
      %{"key" => key}=params
    ) do
      n = 3
      result = HandoffKV.Service.store(key, data, n)
      render conn, store: result
    end

  end
```

Phoenix + Riak Core

```
curl -XPUT -d '{"a":"b"}' localhost:4000/api/store/my_key
  defmodule MyApp.StoreController do
    use Phoenix.Controller
    use MyApp.Web, :controller
    def store(
      %Plug.Conn{body_params: data}=conn,
      %{"key" => key}=params
    ) do
      n = 3
      result = HandoffKV.Service.store(key, data, n)
      render conn, store: result
    end
    # … similar for fetch/2
  end
```

So…isn’t that just terribly normal? Our interaction with our Riak Core application might as well be any other database or service.

exercise 5
HTTP API

expose a few methods of our YakDB over an HTTP API

Rolling our own crypto

Riak KV provides a _ton_ of features on top of riak core specific to being a KV database (version vectors, map reduce, anti-entropy, etc, etc.). If your riak core app is converging on being essentially a KV store, you're almost certainly better off running Riak KV.

Rolling our own crypto

- it isn't the worst clone of Riak KV

Riak KV provides a _ton_ of features on top of riak core specific to being a KV database (version vectors, map reduce, anti-entropy, etc, etc.). If your riak core app is converging on being essentially a KV store, you're almost certainly better off running Riak KV.

Rolling our own crypto

- it isn't the worst clone of Riak KV
- beware converging on bad clones of KV

Riak KV provides a _ton_ of features on top of riak core specific to being a KV database (version vectors, map reduce, anti-entropy, etc, etc.). If your riak core app is converging on being essentially a KV store, you're almost certainly better off running Riak KV.

Rolling our own crypto

- it isn't the worst clone of Riak KV
- beware converging on bad clones of KV
- do other things!

Riak KV provides a _ton_ of features on top of riak core specific to being a KV database (version vectors, map reduce, anti-entropy, etc, etc.). If your riak core app is converging on being essentially a KV store, you're almost certainly better off running Riak KV.

Zap Chat

web scale telnet chat

web scale telnet chat

```
nc localhost 4040
```

web scale telnet chat

```
nc localhost 4040
```

```
set-name Alex
```

web scale telnet chat

```
nc localhost 4040

set-name Alex

join CoolRoom
```

web scale telnet chat

```
nc localhost 4040

set-name Alex

join CoolRoom

say CoolRoom HI!
```

exercise 6
zap chat