wooga
world of gaming

Games for the Masses

How DevOps Affects Architecture

Jesper Richter-Reichhelm, @jrirei

wooga | Diamond Dash | Monster World | Bubble Island | Happy Hospital | Brain Buddies

Like 3M

608537 Add Coins
158 Add WooGoo
92 1201818

08:04:09

122

Plants | Products | Items

140	82	23	80
128	101	55	50

#	Name		
#1	Jesper	92	1,201,818
#2	Anke	91	1,135,517
#3	Sönke	90	1,025,391
#4	Fabian	89	943,727
	Invite		
#5	Manuela	87	755,036

Sandrine Valério is playing Adventure World – An Indiana Jones Game. about a minute ago

Sandrine Valério is playing Diamond Dash. about a minute ago

Jesper Richter-Reichhelm is playing Monster World. 4 minutes ago

Shaveer Mirpuri played Texas HoldEm Poker. Sponsored

Jesper Richter-Reichhelm has earned 14 of 301 achievements in Monster World.

Boril Boshnakov is playing Adventure World – An Indiana Jones Game. 7 minutes ago

Frank Ließner is playing Idle Worship. 9 minutes ago

Denise Engel is playing Adventure World – An Indiana Jones Game. 11 minutes ago

Sebastian Werner is playing Diamond Dash. 12 minutes ago

Sebastian Werner is playing Monster World. 13 minutes ago

Johannes Ippen played Adventure World – An Indiana Jones Game. 21 minutes ago

Florian Steinhoff played Zombie Island. 20 minutes ago

Boril Boshnakov is playing Adventure World – An Indiana Jones Game. 7 minutes ago

Frank Ließner is playing Idle Worship. 9 minutes ago

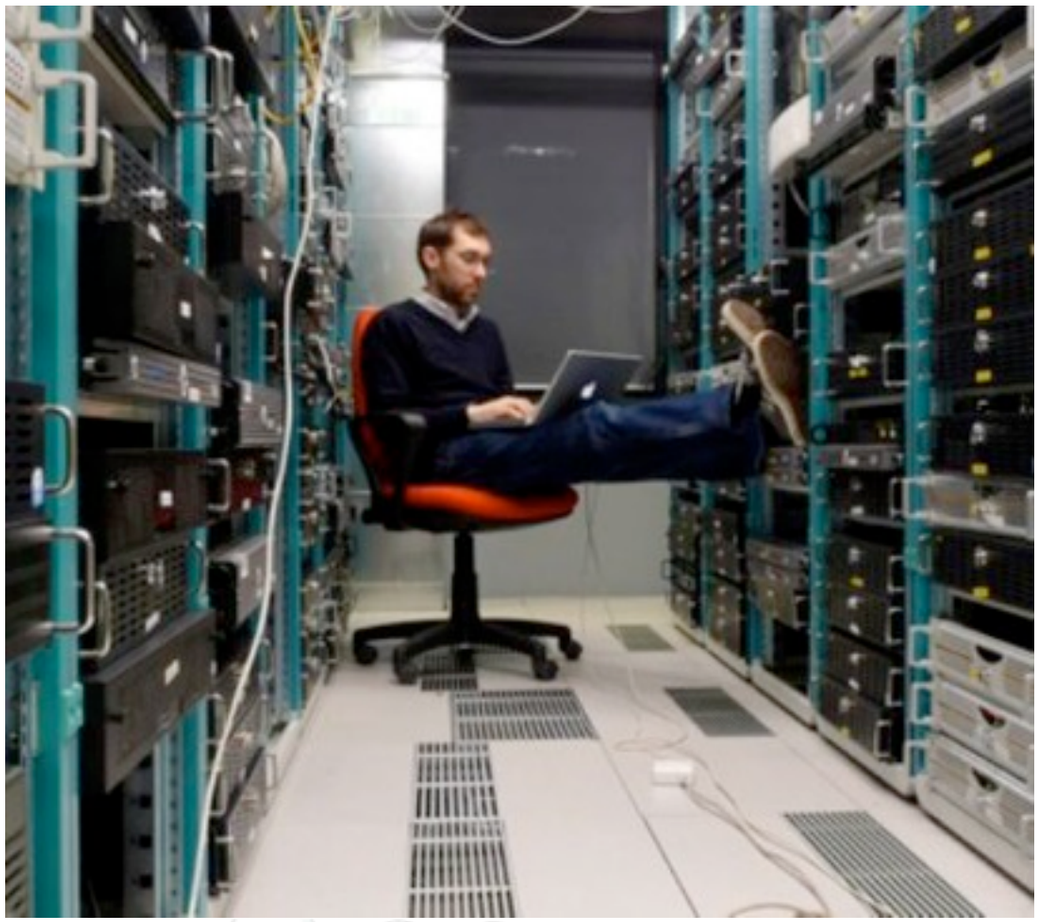
Denise Engel is playing Adventure World – An Indiana Jones Game. 11 minutes ago

Sebastian Werner is playing

Our games all look the same

Flash client

Backend

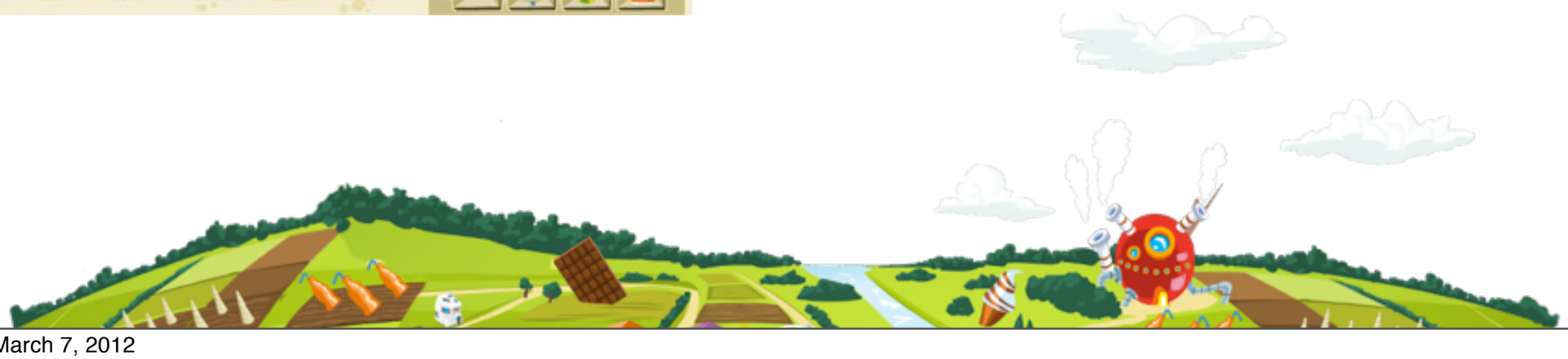


Our games all look the same

Flash client

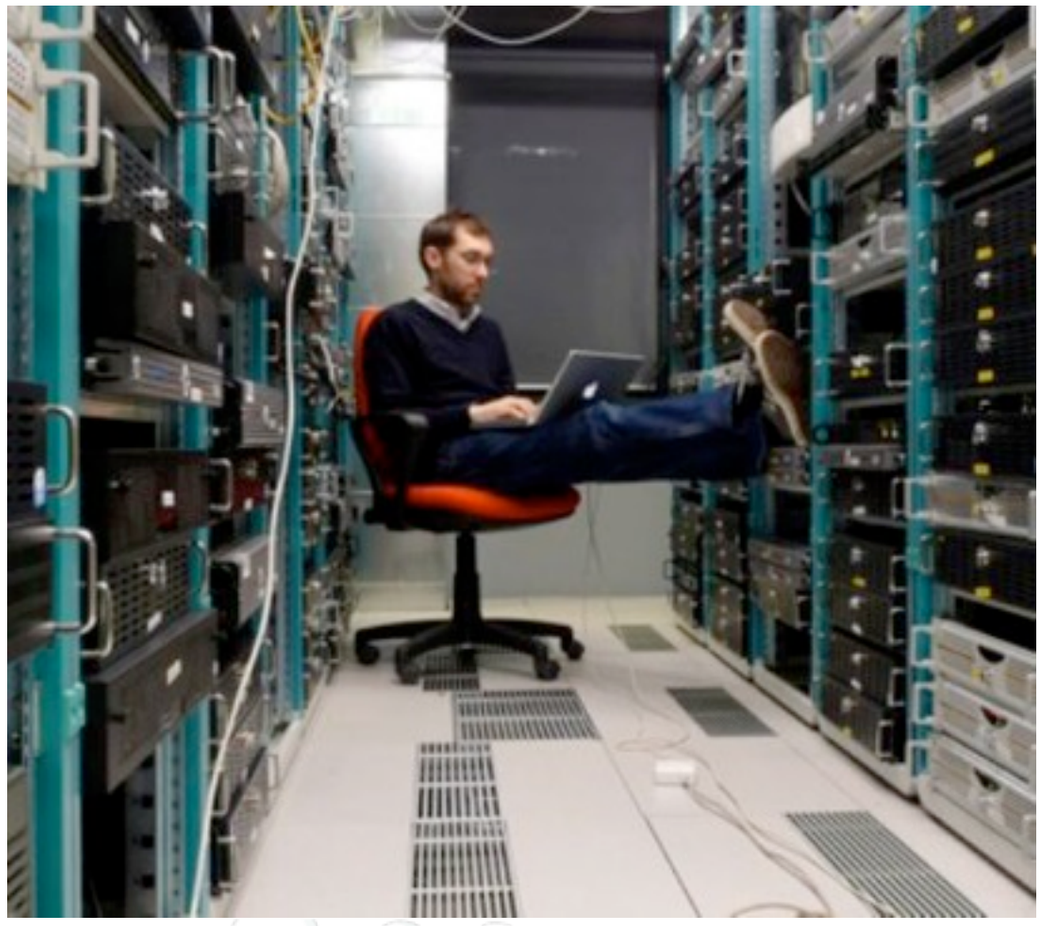
Game Session

Asynch. Communication



Our games all look the same

Backend



State Changes

Validation

Persistence

But the scale is interesting

14 billion requests / month

But the scale is interesting

Monster World

14 billion requests / month

Wikipedia
The Free Encyclopedia

But the scale is interesting

Monster World

14 billion requests / month

WIKIPEDIA
The Free Encyclopedia

Monster World

>100,000 DB operations / second

But the scale is interesting

14 billion requests / month

>100,000 DB operations / second

>50,000 DB updates / second

2 Developers to do it all

Typical team setup

4 product managers

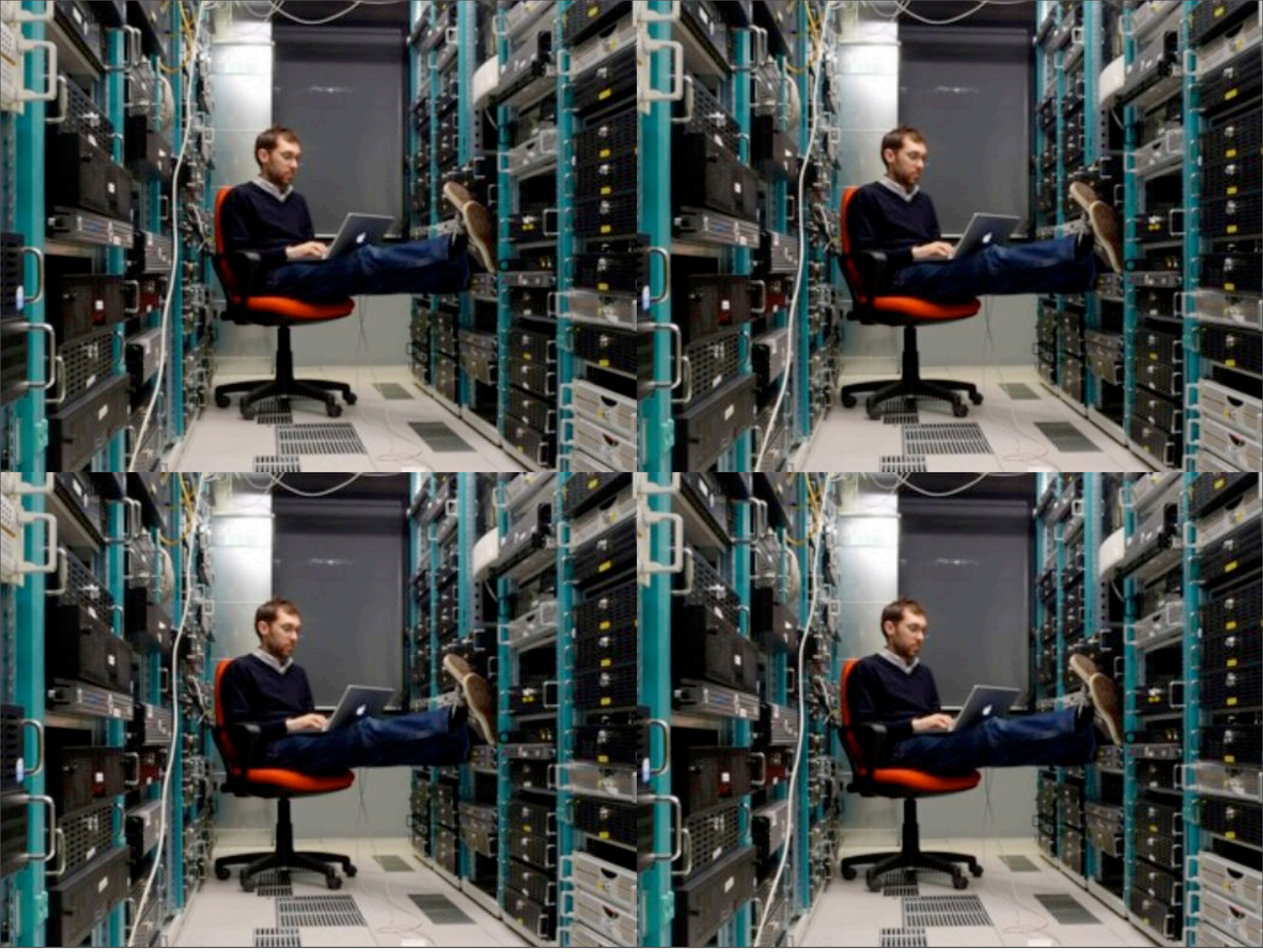
4 artists

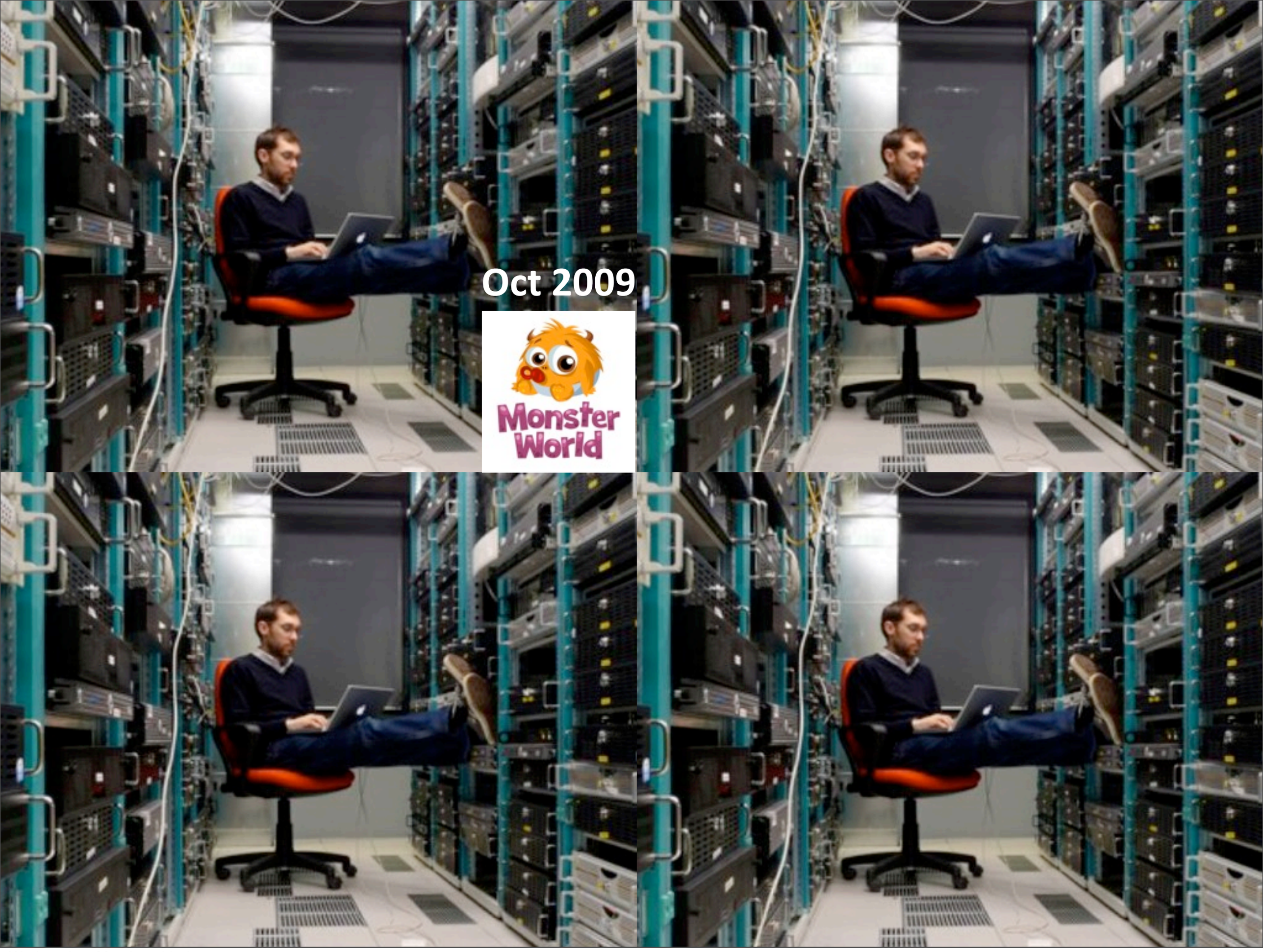
4 frontend engineers

2 backend engineers

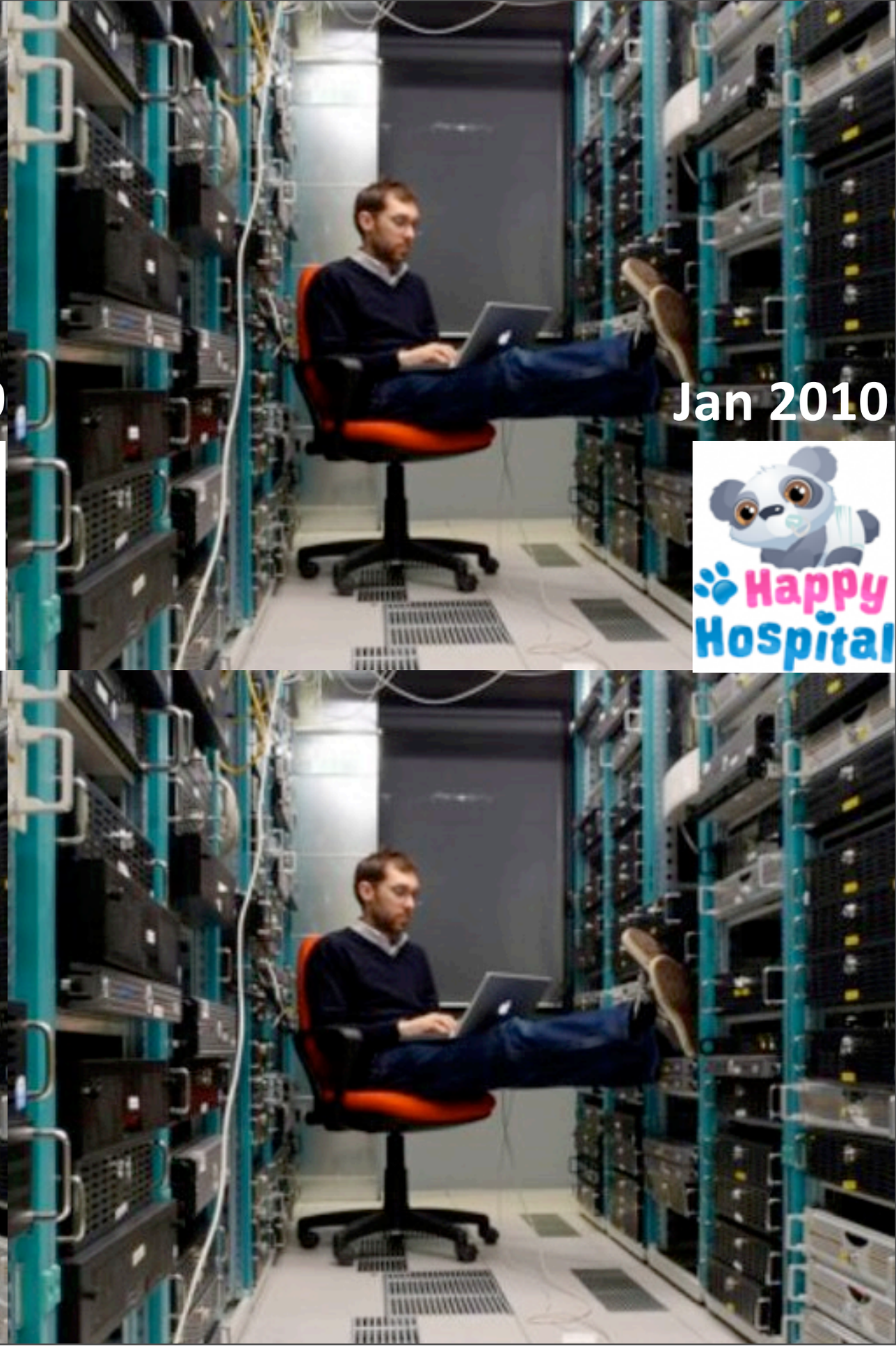
- design, implementation, operation

Wooga has dedicated game teams




Oct 2009
Monster World


Oct 2009
Monster World


Jan 2010
Happy Hospital

Oct 2009

Monster World

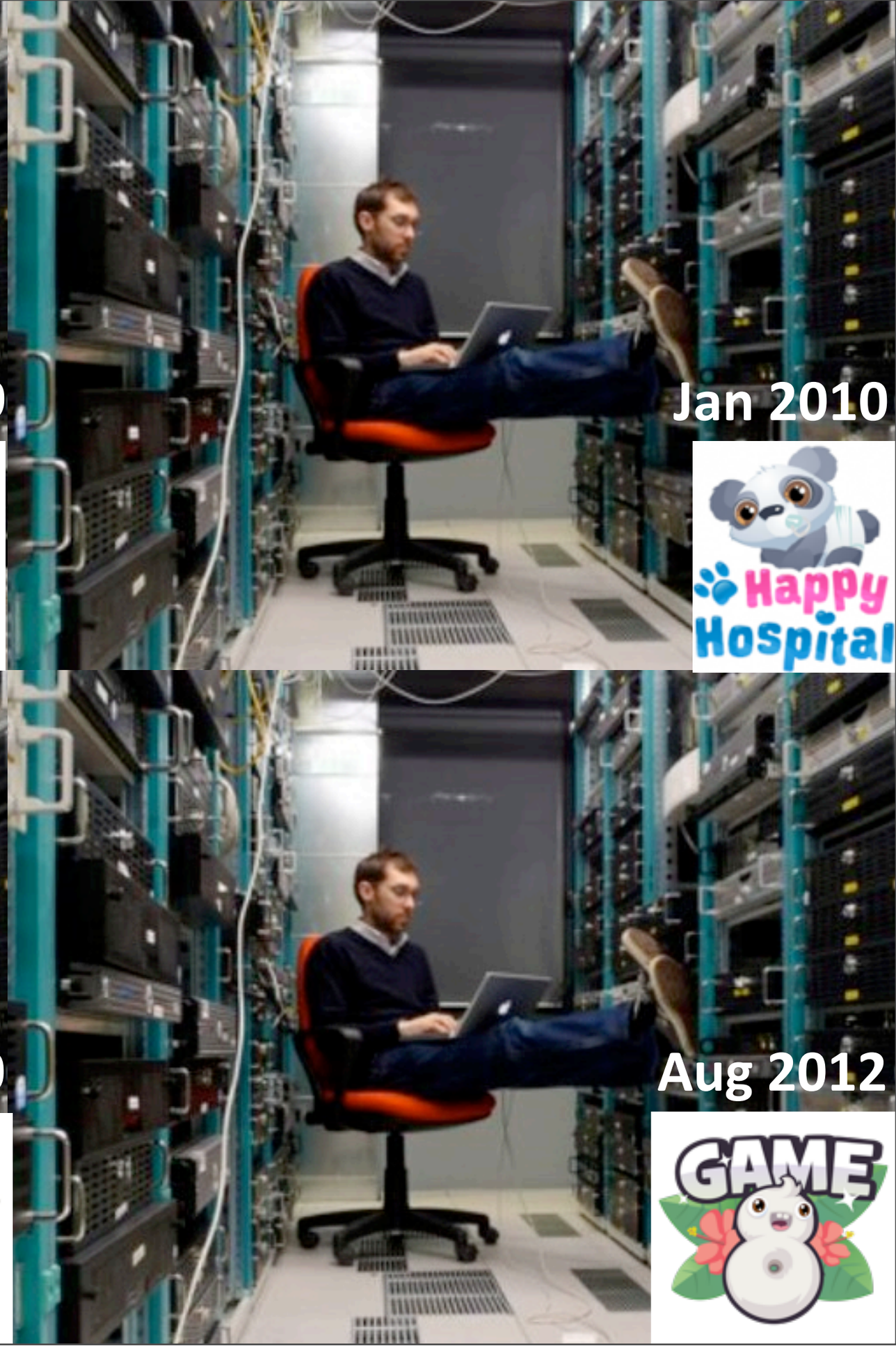
Jan 2010

Happy Hospital

Oct 2010

Magic Land


Oct 2009
Monster World


Jan 2010
Happy Hospital

Oct 2010
Magic Land

Aug 2012
GAME

Architecture Evolution at Wooga

The Start

The Next Step

Best of Two Worlds

Company Values

Oct 2009: 1st team wanted good code quality

Good code quality

- Easy to understand
- Easy to test
- Easy to refactor

Oct 2009: 1st team wanted good code quality

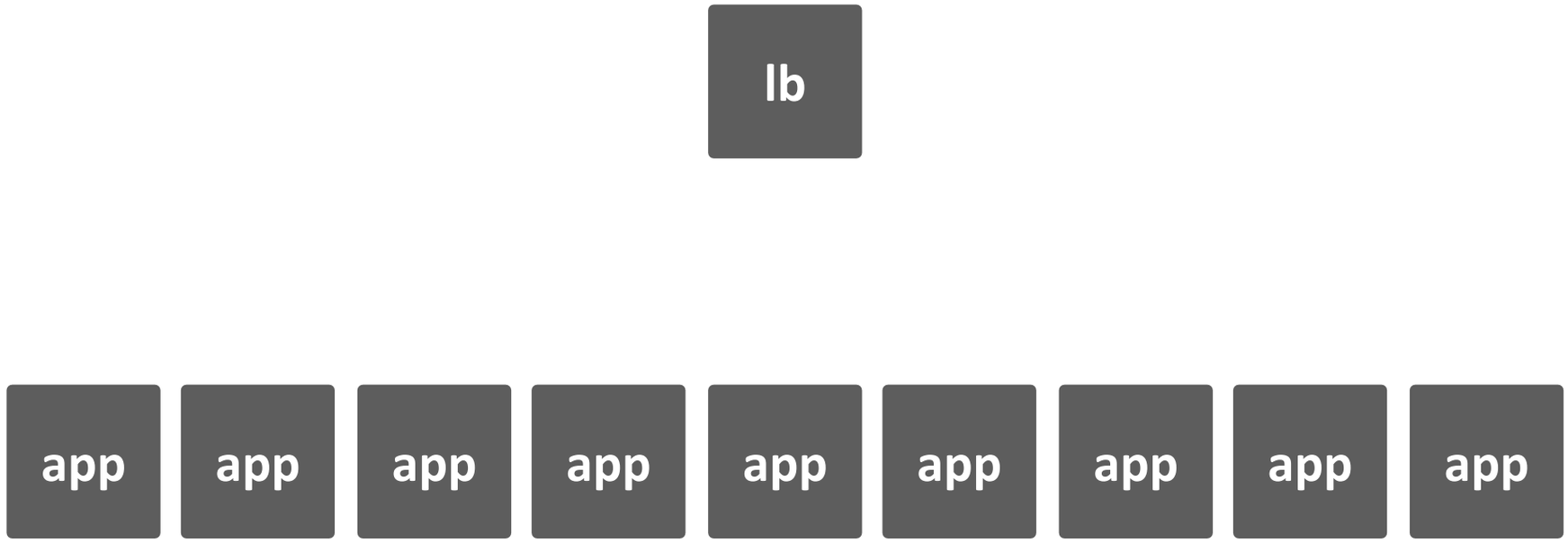
Good code quality

- Easy to understand
- Easy to test
- Easy to refactor

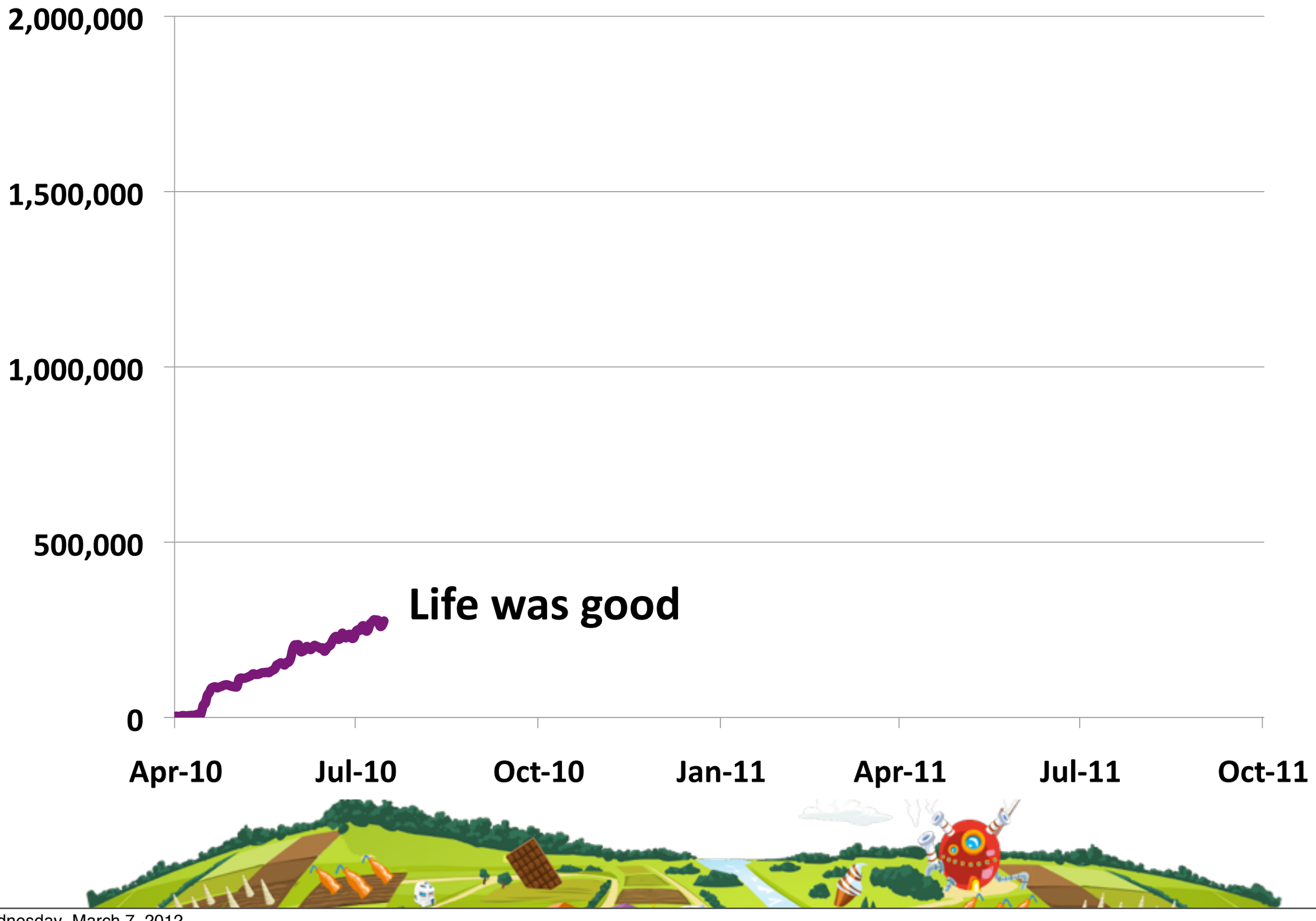
Evolution I: Use Ruby (on Rails)

Oct 2009

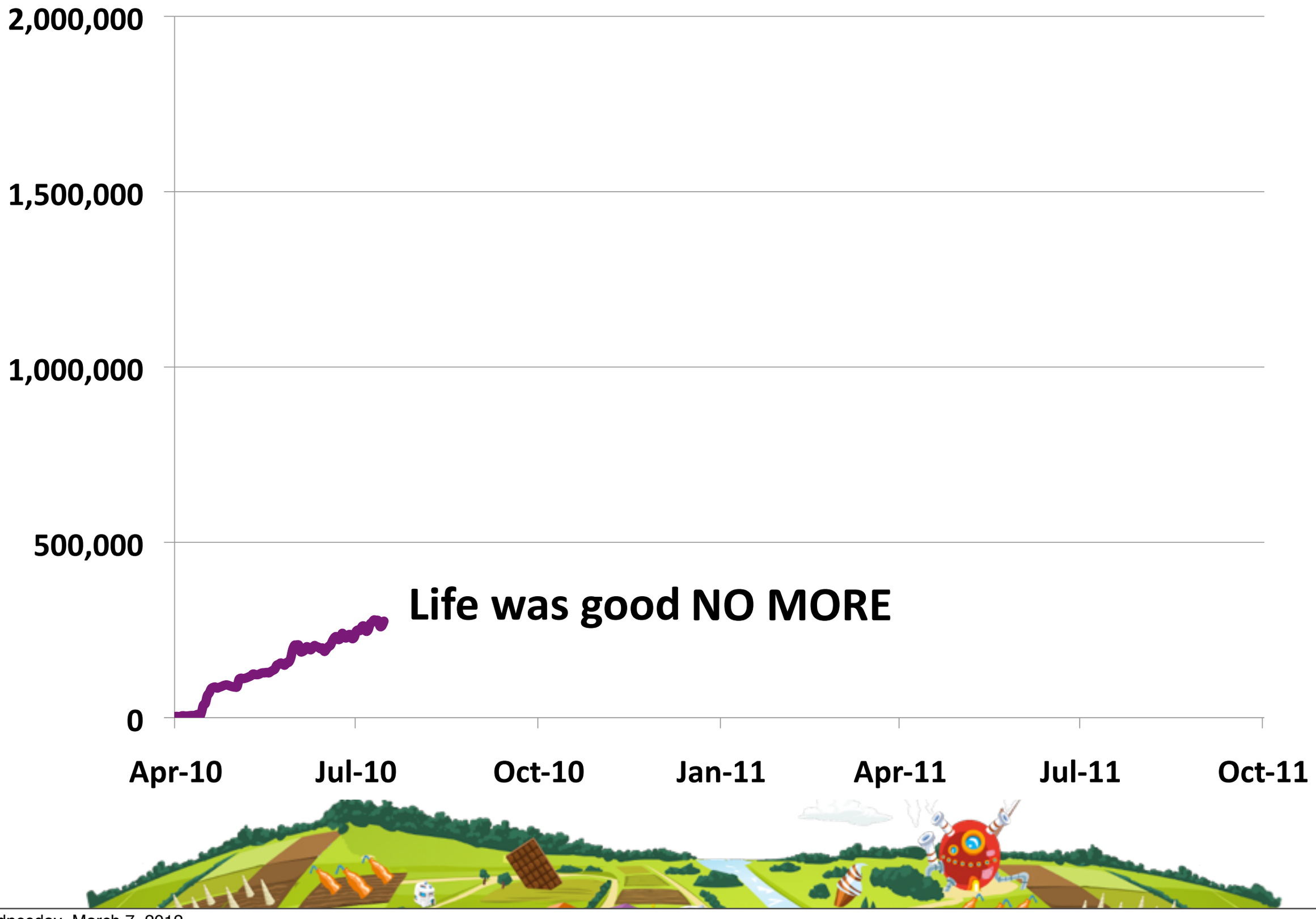
A basic setup using sharding worked fine



250K daily users



250K daily users



Welcome to 6 weeks of pain!

Heavy optimizations were necessary

Welcome to 6 weeks of pain!

Heavy optimizations were necessary

Numerous small fixes regarding DB config

Welcome to 6 weeks of pain!

Heavy optimizations were necessary

- Numerous small fixes regarding DB config
- More shards

Welcome to 6 weeks of pain!

Heavy optimizations were necessary

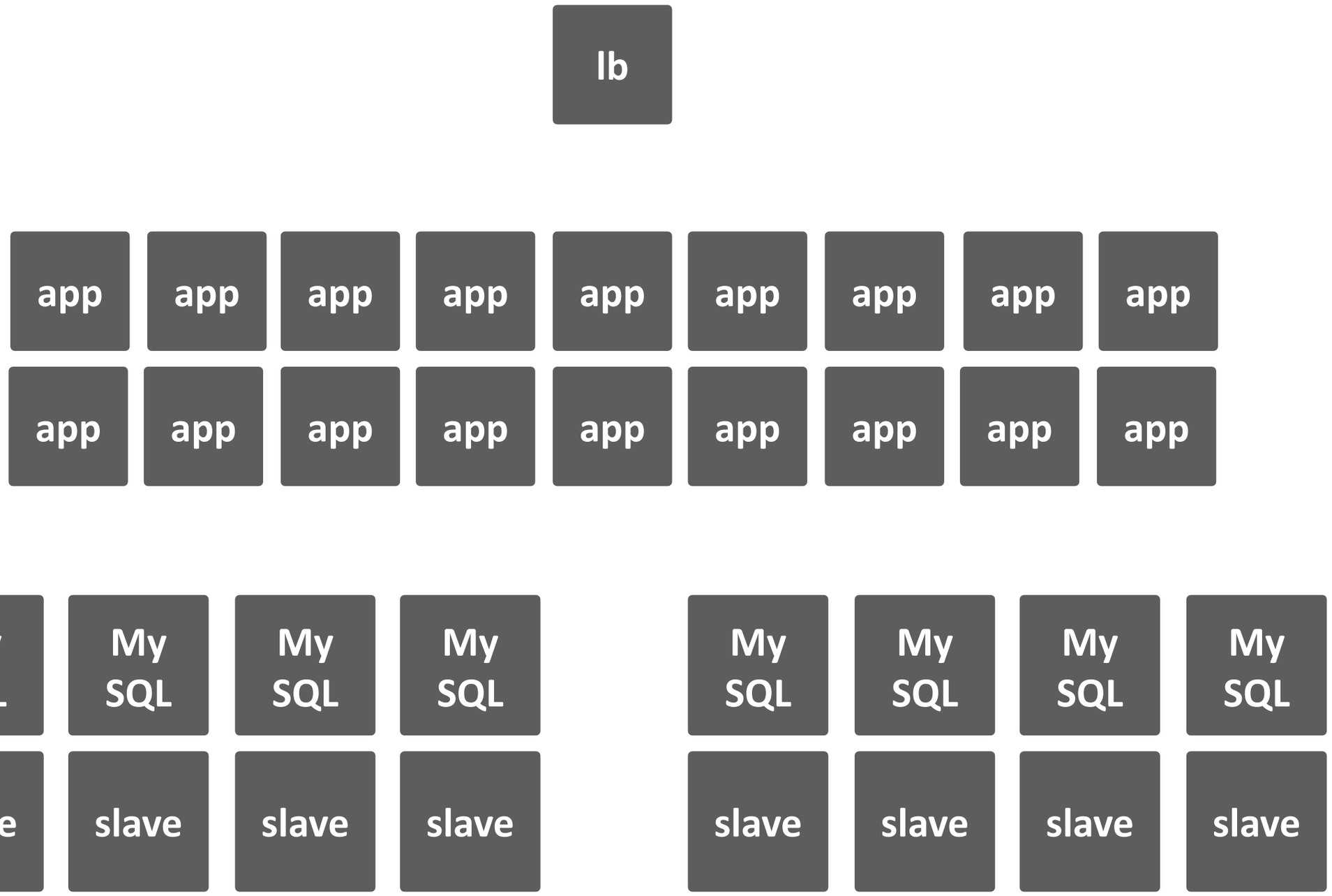
- Numerous small fixes regarding DB config
- More shards
- Even more shards

Welcome to 6 weeks of pain!

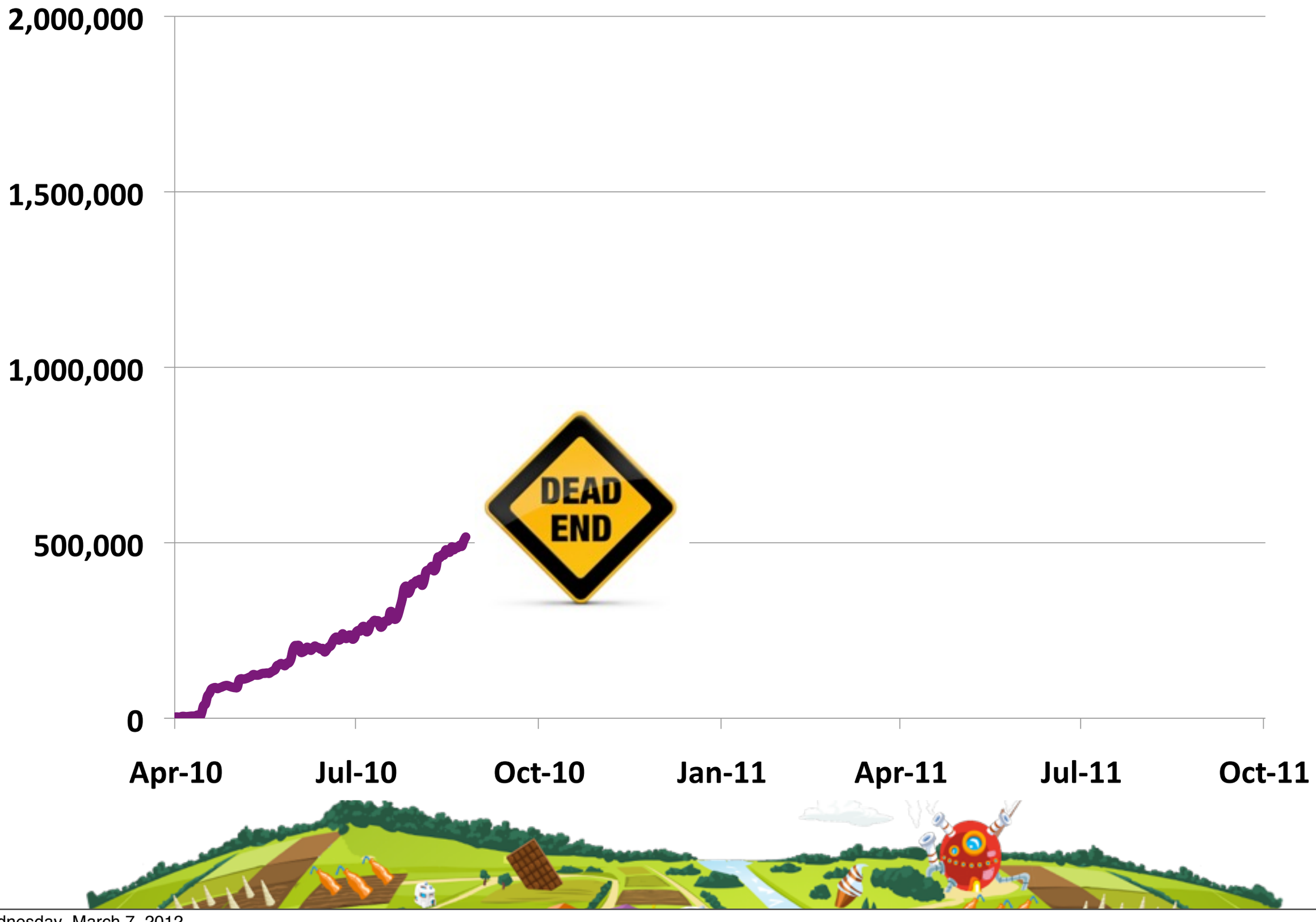
Heavy optimizations were necessary

- Numerous small fixes regarding DB config
- More shards
- Even more shards
- Splitting the model to get more shards

Early sharding hell: 8 master and 8 slaves



At 500K daily users we were at a dead end



OUCH!

http://www.flickr.com/photos/billue_the_bear/

Jan 2010: Meanwhile at the 2nd team

Don’t break the bank

- Make it faster
- Make it cheaper
- Make it simpler

Jan 2010: Meanwhile at the 2nd team

Don’t break the bank

- Make it faster
- Make it cheaper
- Make it simpler

Evolution II: Use Redis as main database

Oct 2009

Jan 2010

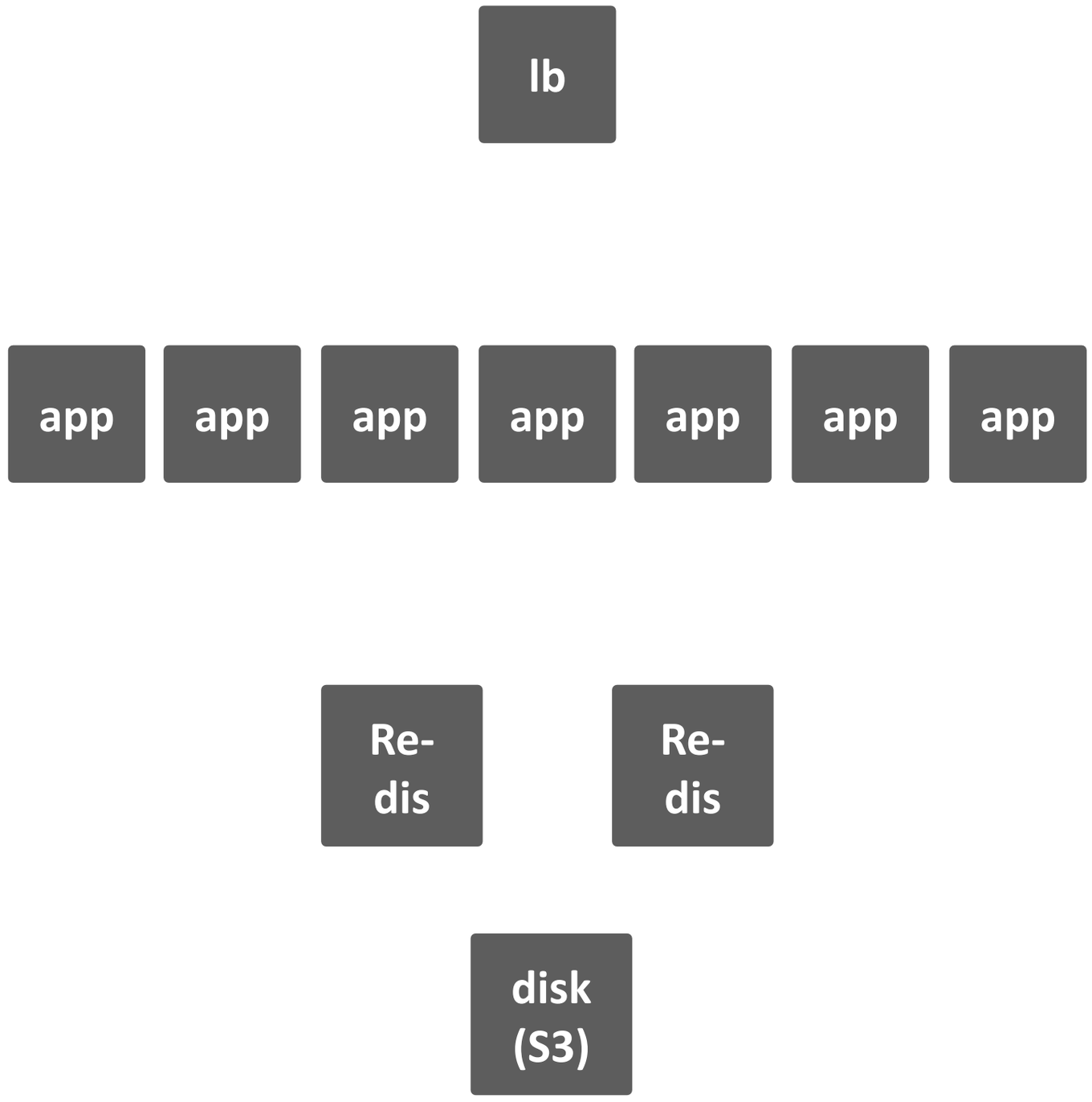
If MySQL is a truck

Fast enough **disk based** **robust**

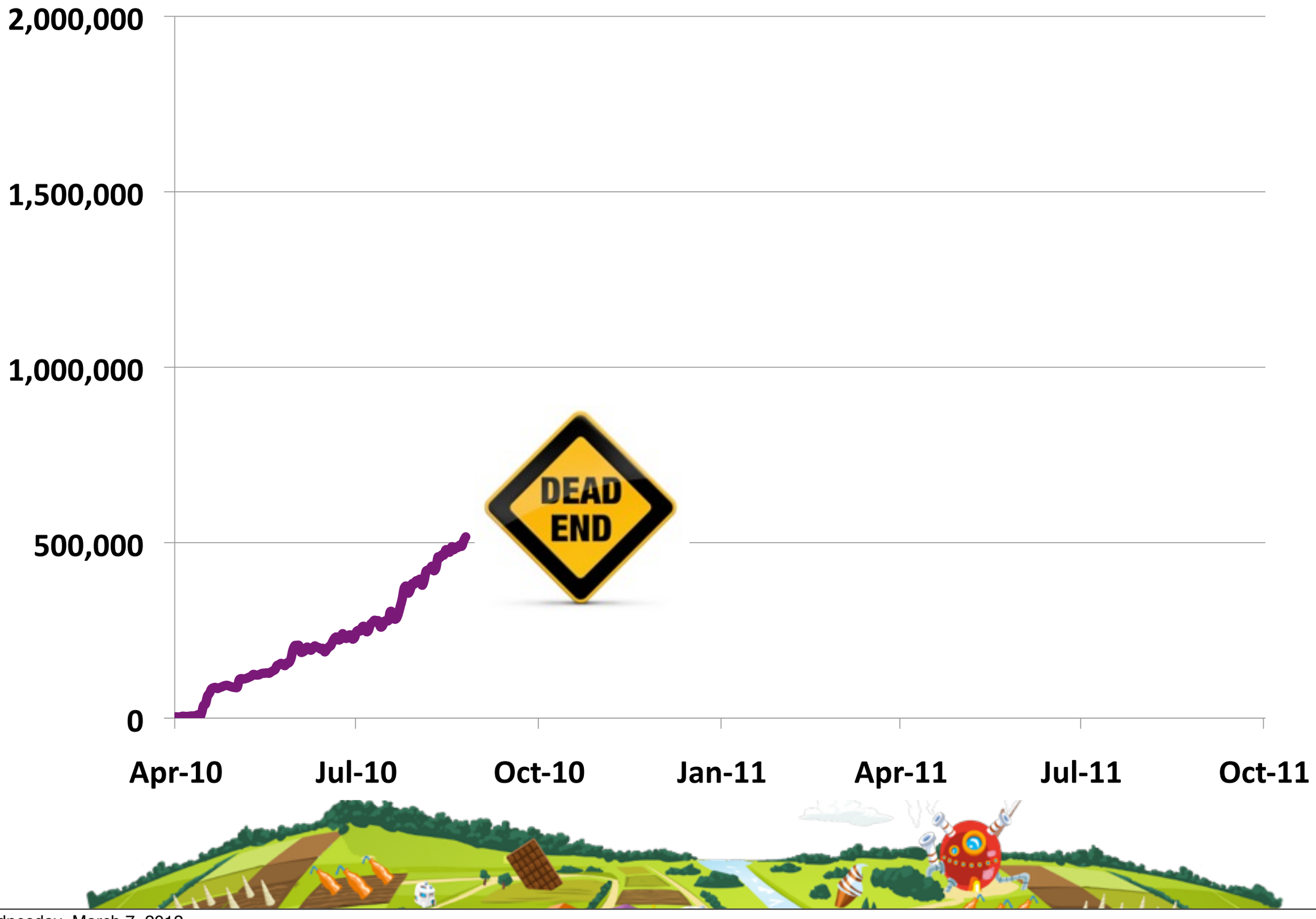
If MySQL is a truck, Redis is a race car

Super fast **RAM based** **fragile**

Bare metal for low latency!



How could we apply that knowledge?



On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end


def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end
```

```ruby
def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end


def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end



def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end



def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end



def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

On-demand migrations from MySQL to Redis

```ruby
def self.find(uid)

  if not_migrated?(uid)
    OldSQL.all(:conditions => {:uid=>uid}).each do |old|
      NewRedis.create_from_old!(old)
    end
  end

  # Load from Redis...
  # Create objects...
end



def not_migrated?(uid)
  redis.hsetnx('migrated_ids', uid)
end
```

Typical migration throughput over 3 days

Big and static data in MySQL, rest goes to Redis

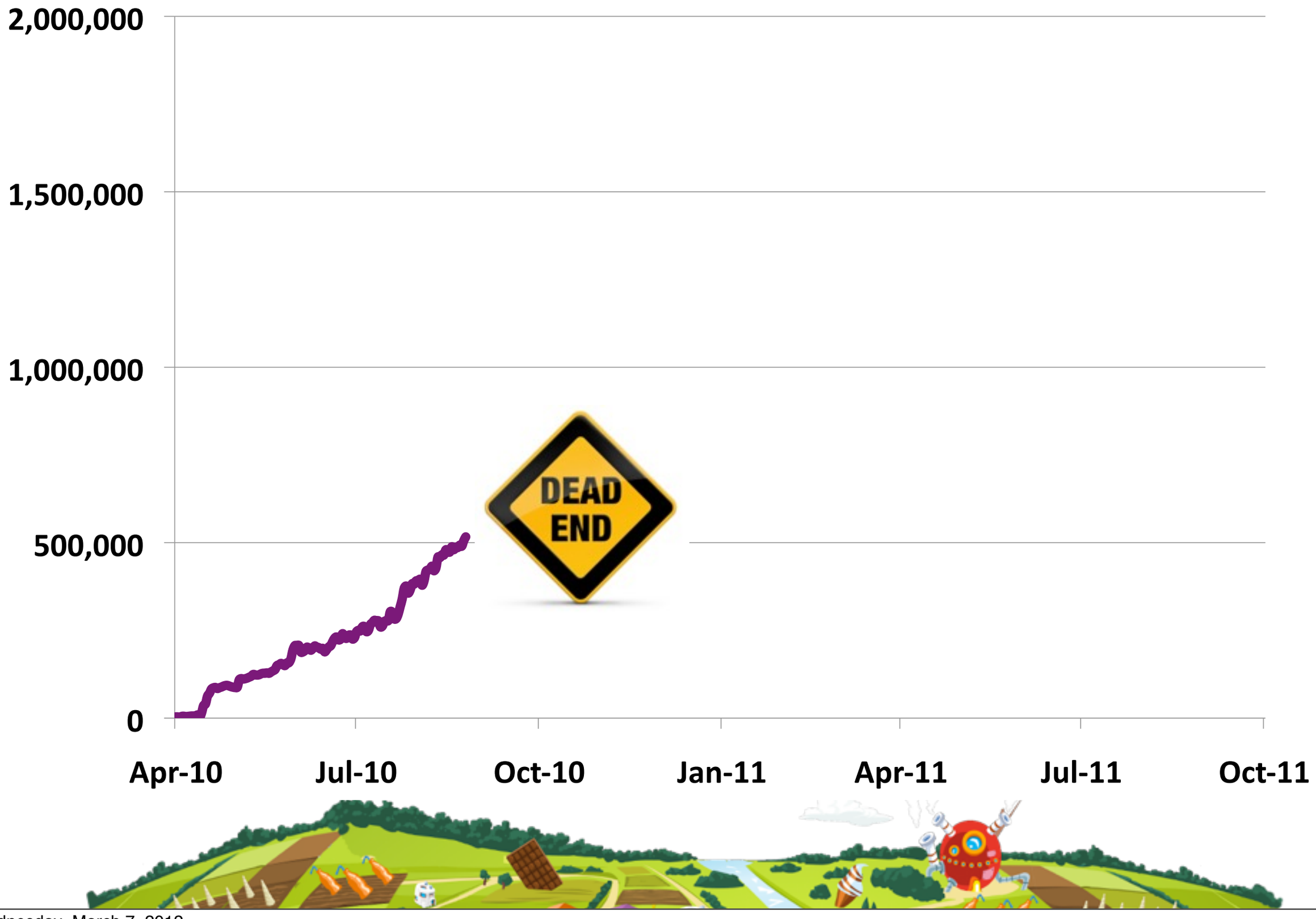
256 GB data

10% writes

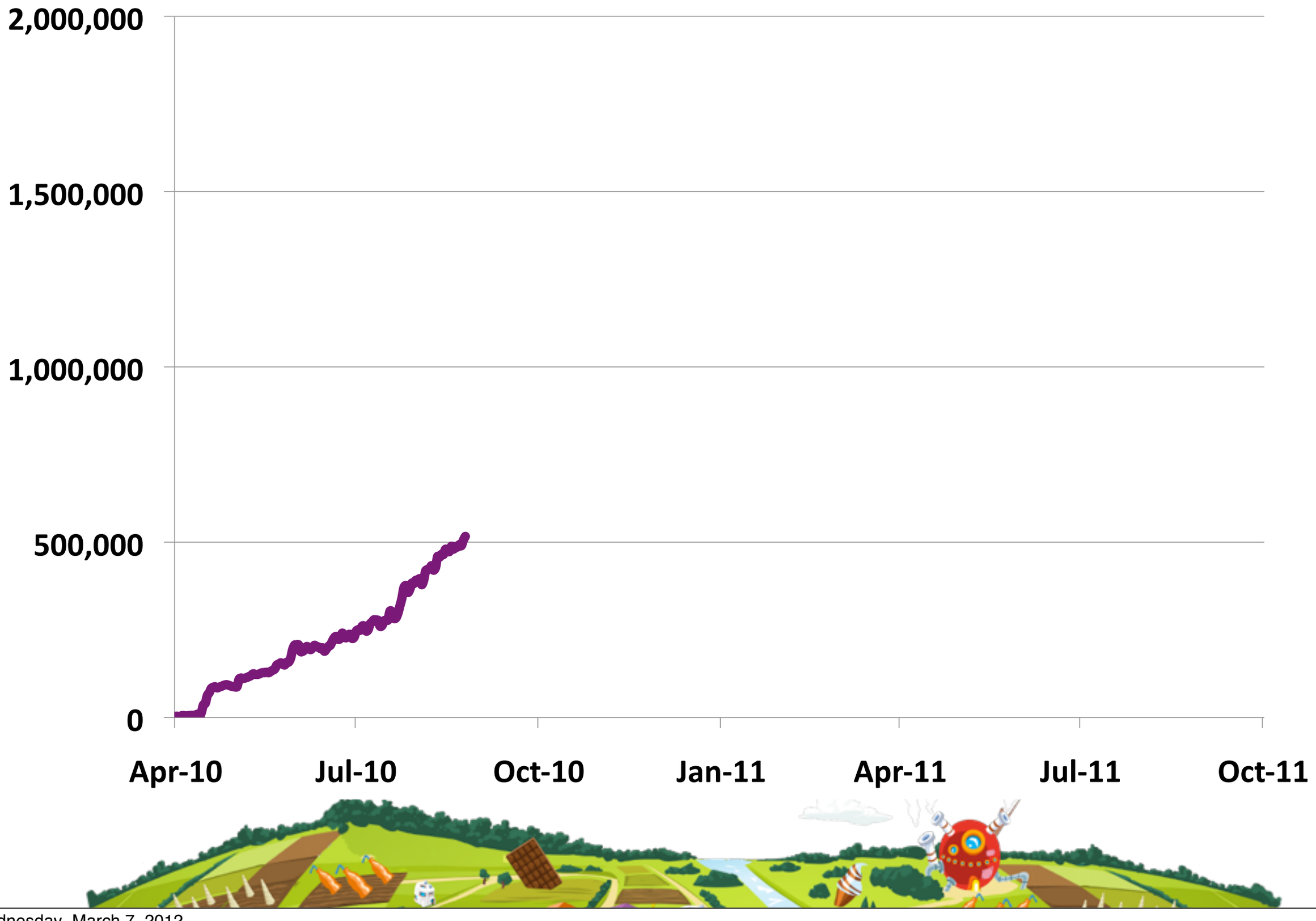
60 GB data

50% writes

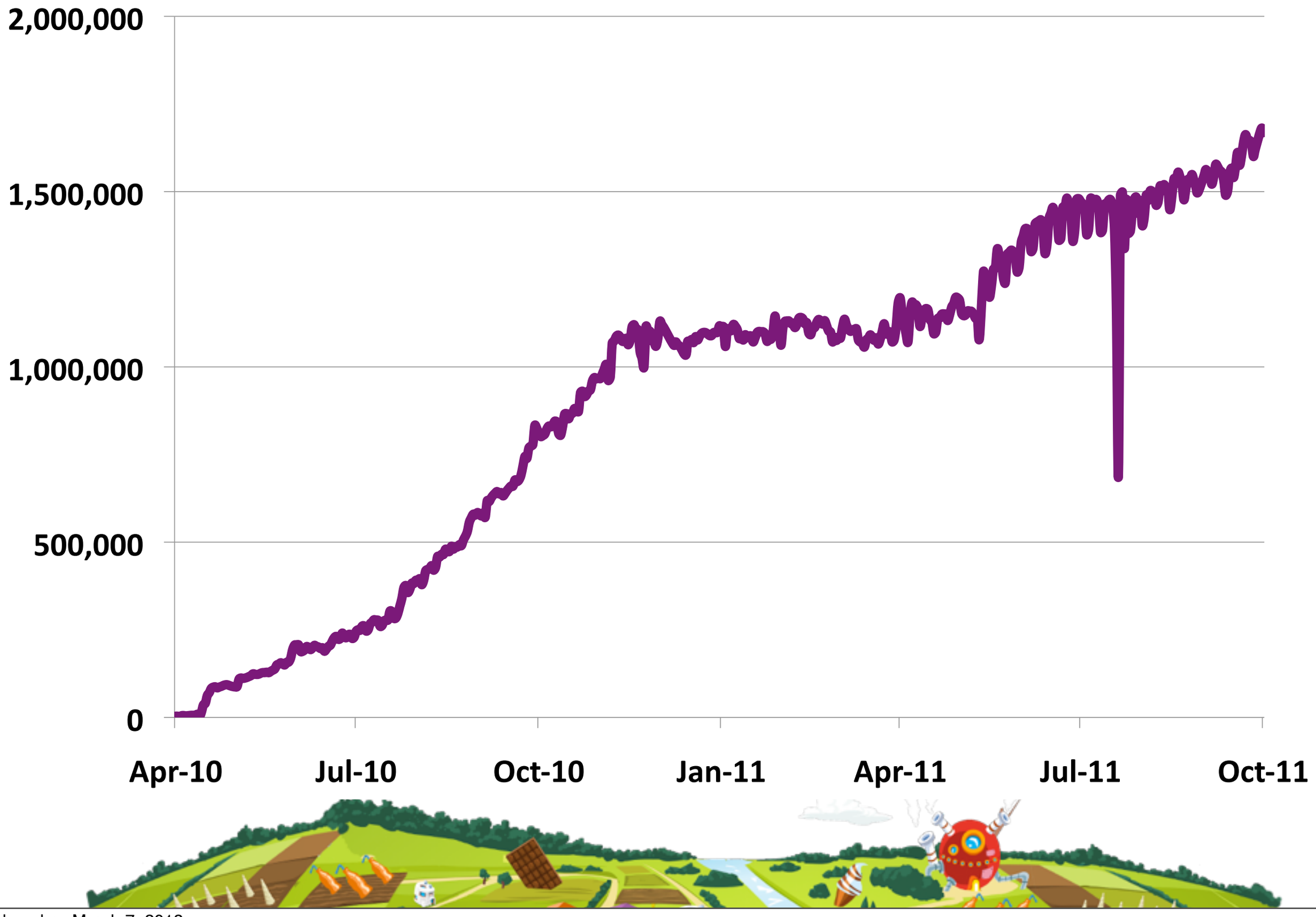
One team saved the other one



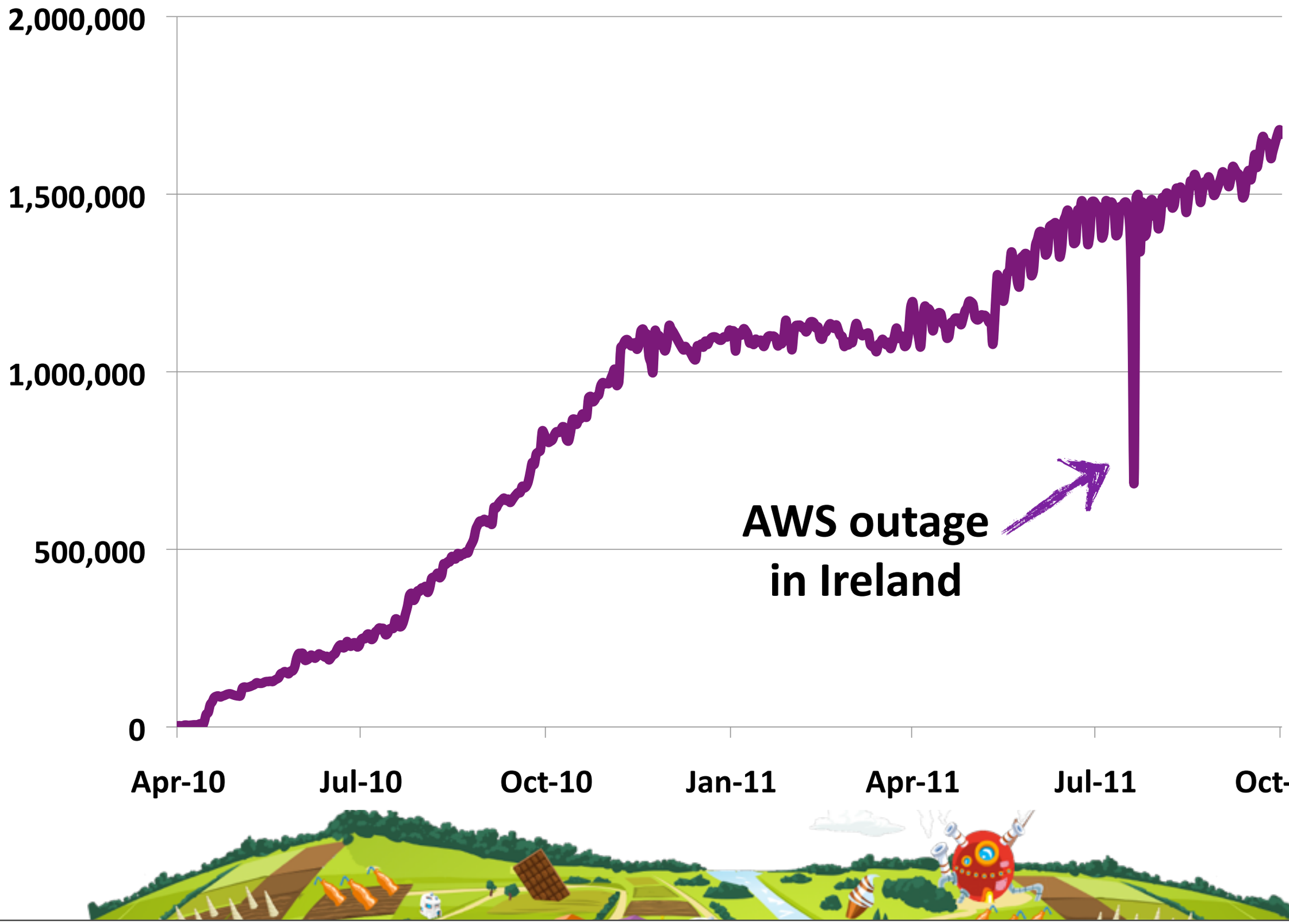
One team saved the other one



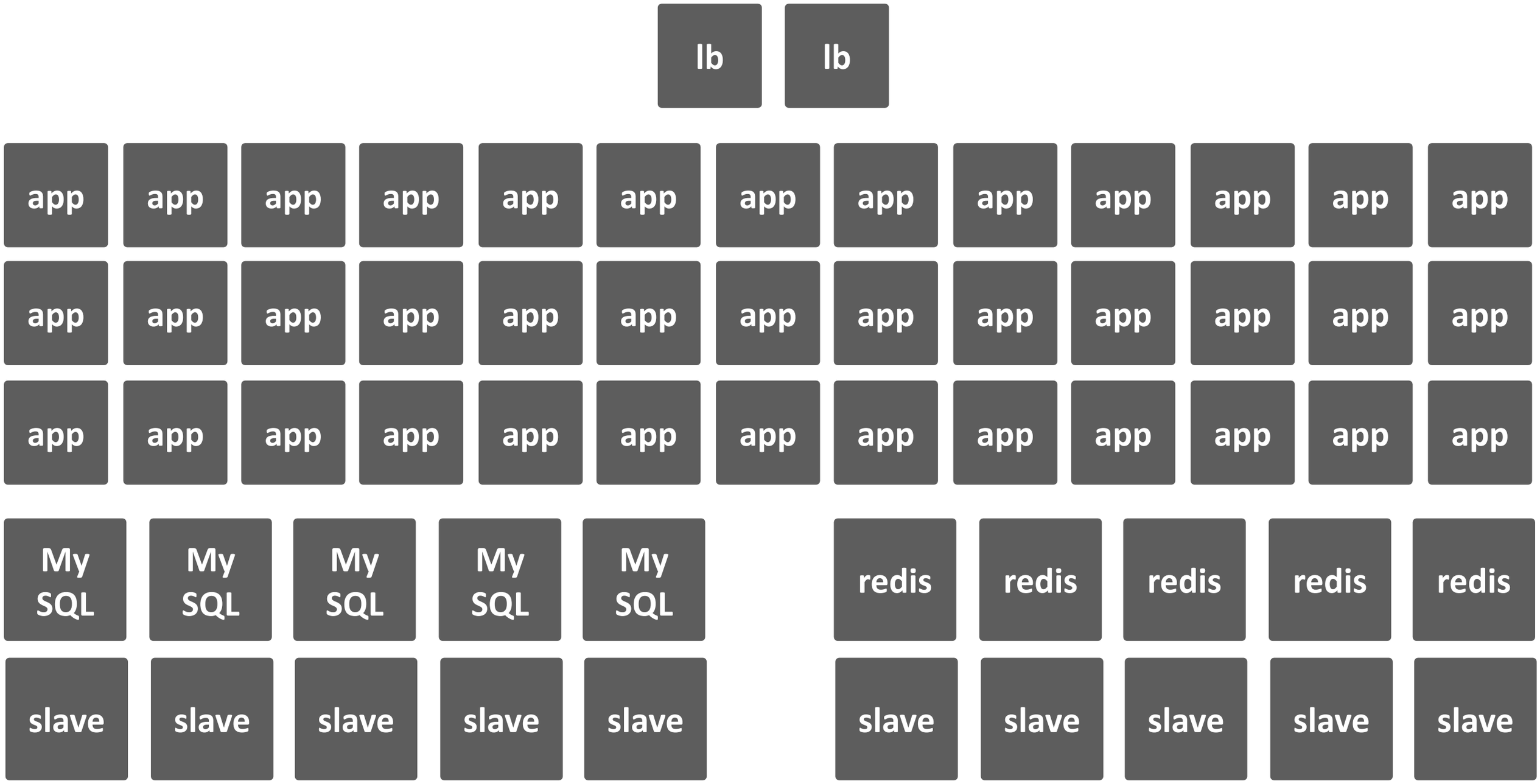
We now have more than 2 million users / day



We now have more than 2 million users / day



10 single-points-of-failure - no fun at all!



http://www.flickr.com/photos/wolfsavard/

Architecture Evolution at Wooga

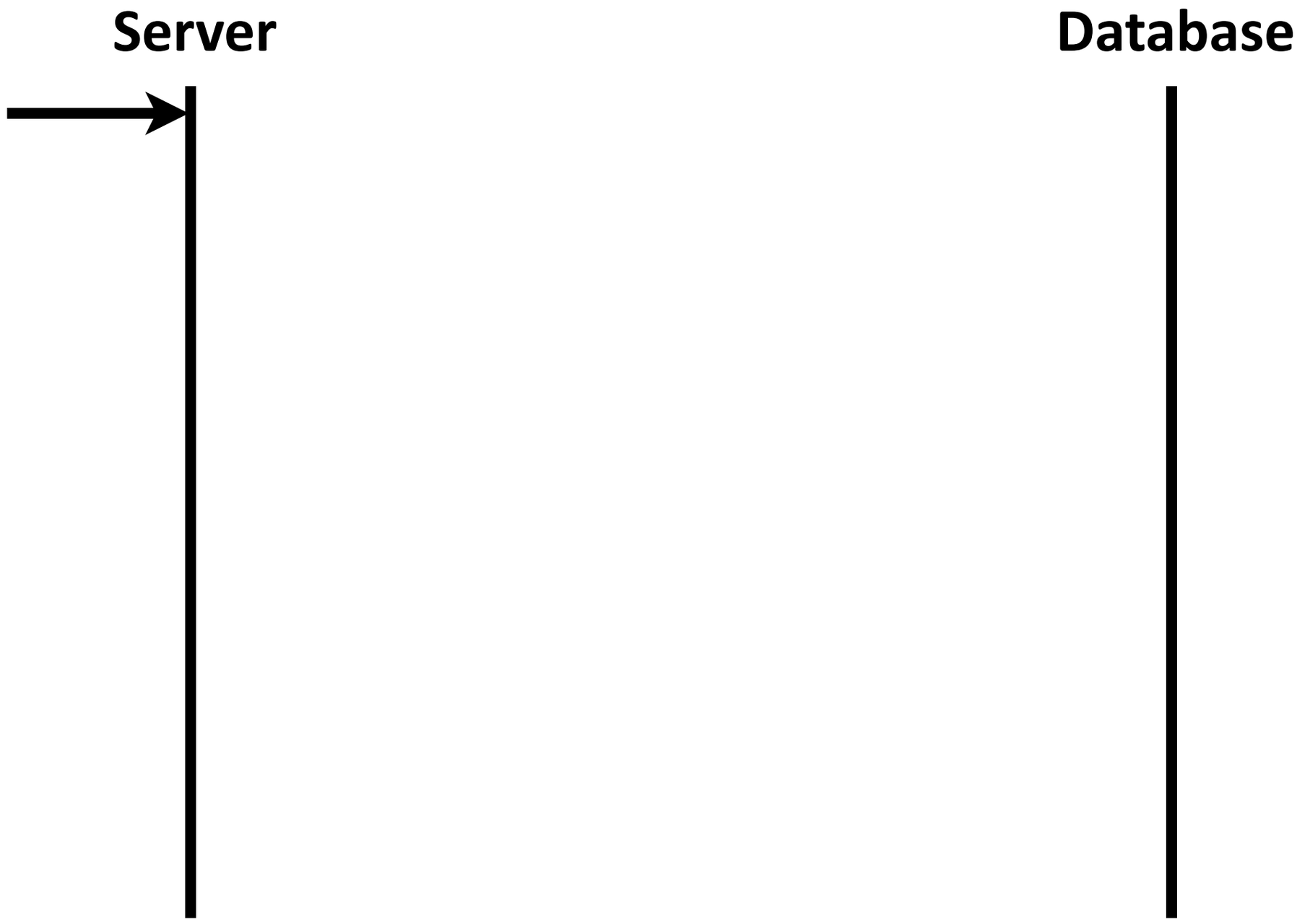
The Start: Ruby

The Next Step

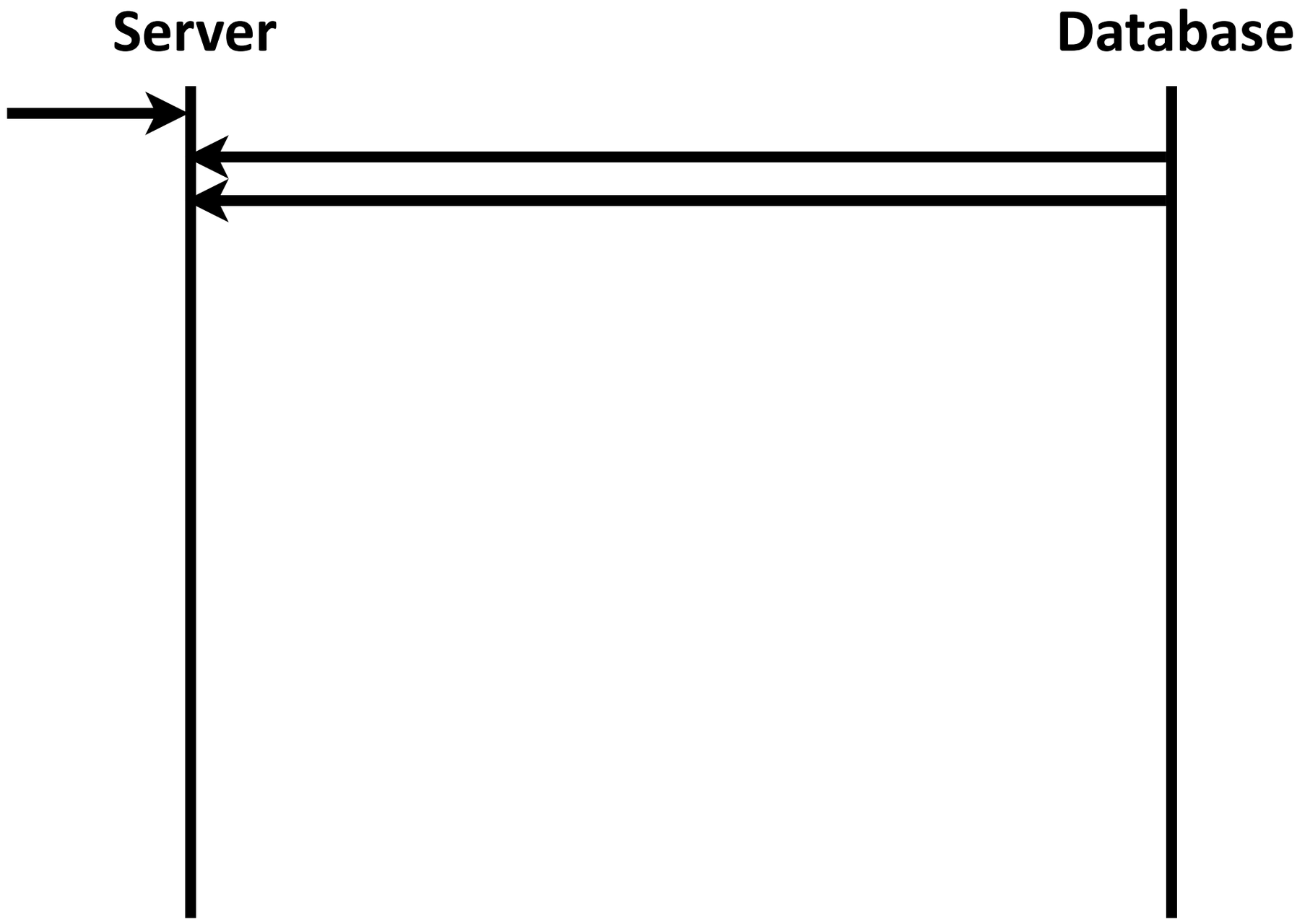
Best of Two Worlds

Company Values

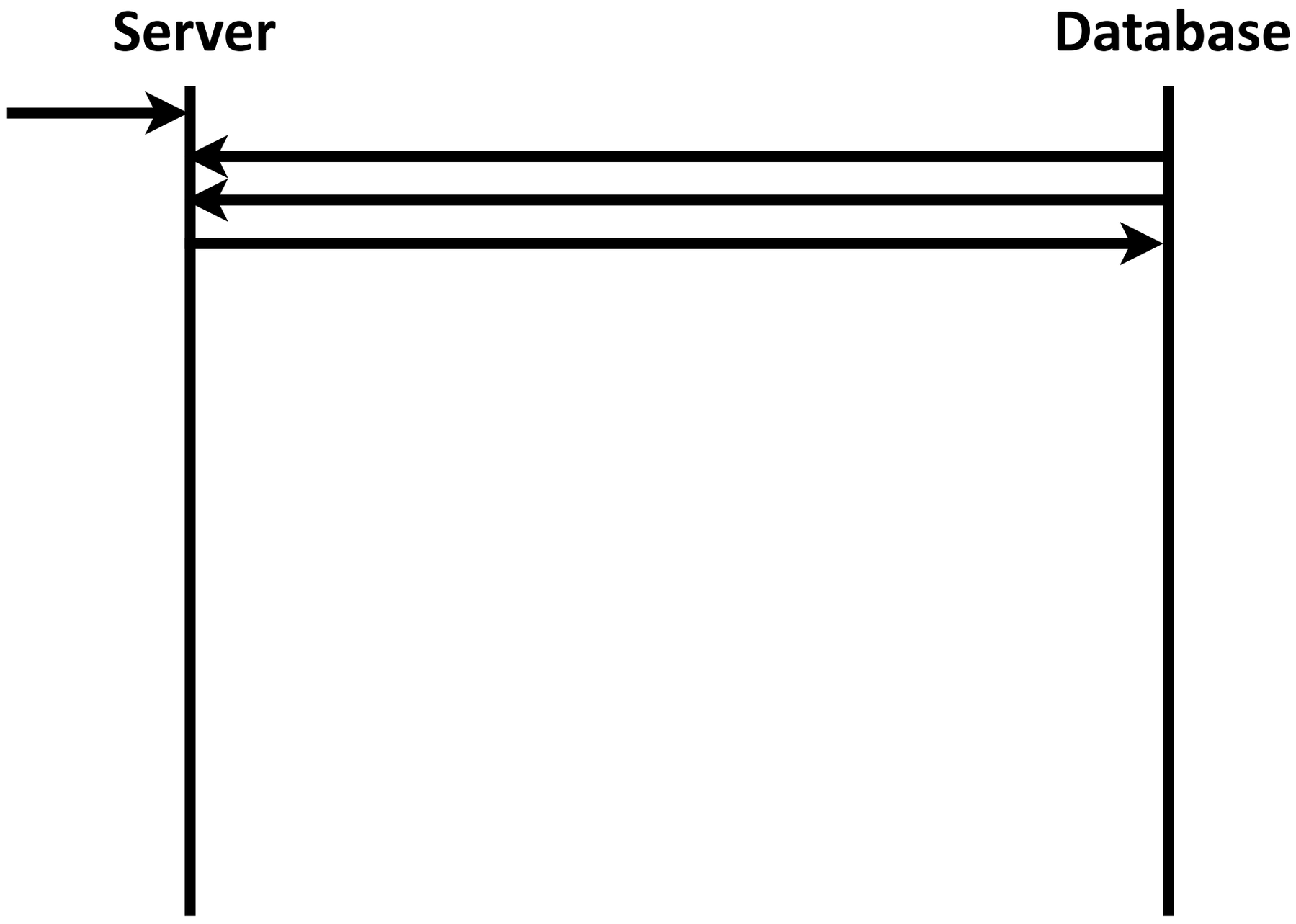
Stateless servers and DBs



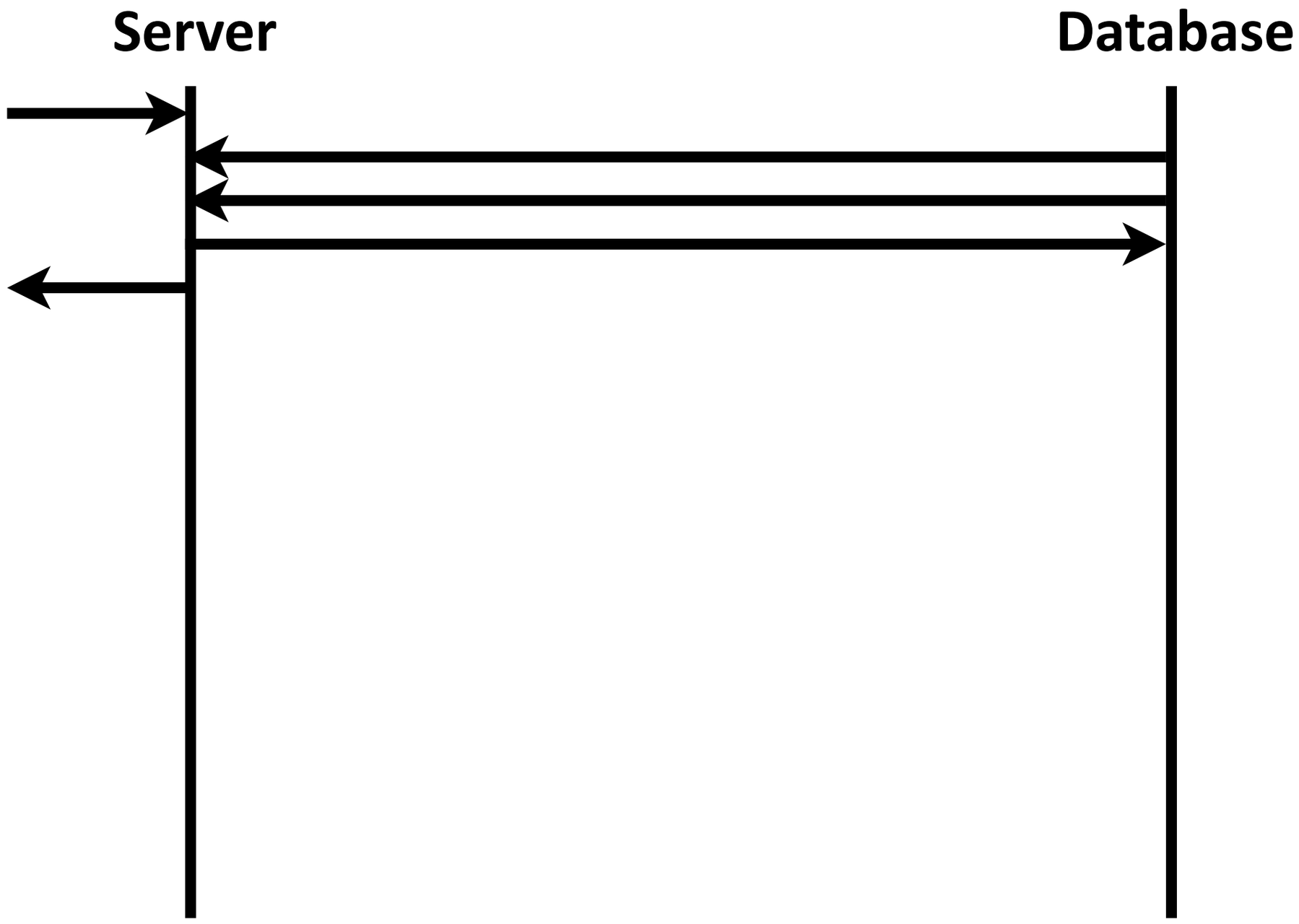
Stateless servers and DBs



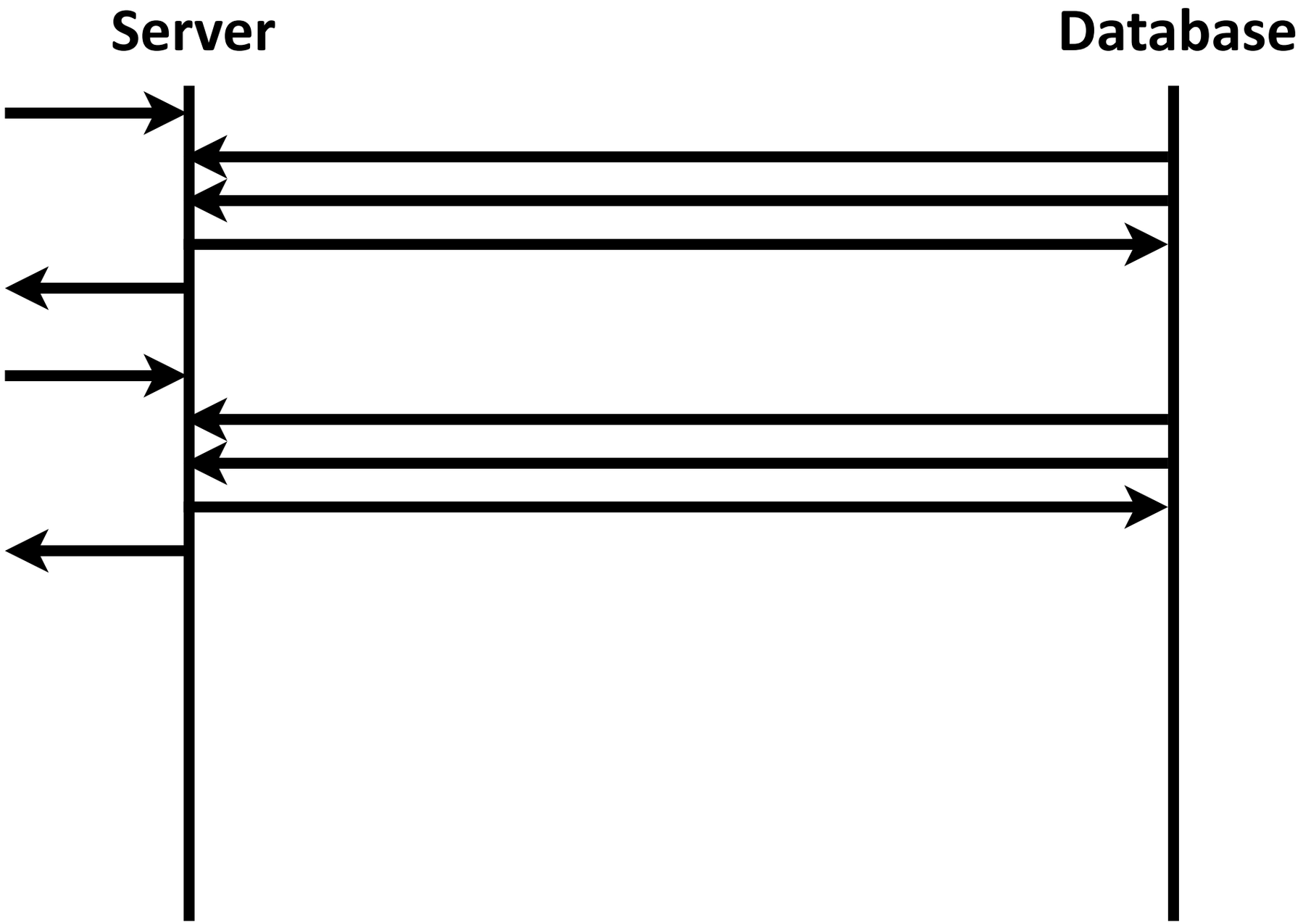
Stateless servers and DBs



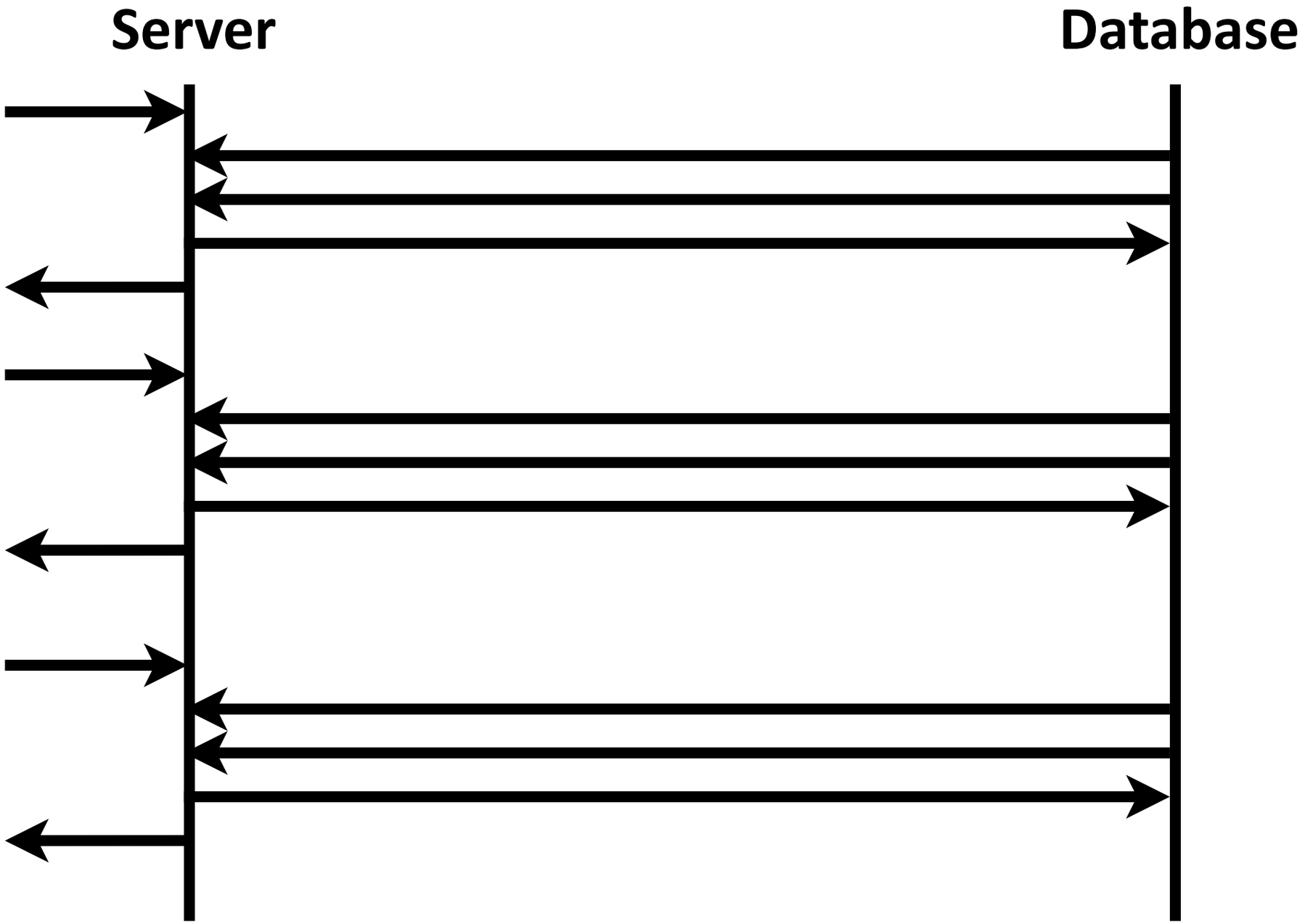
Stateless servers and DBs



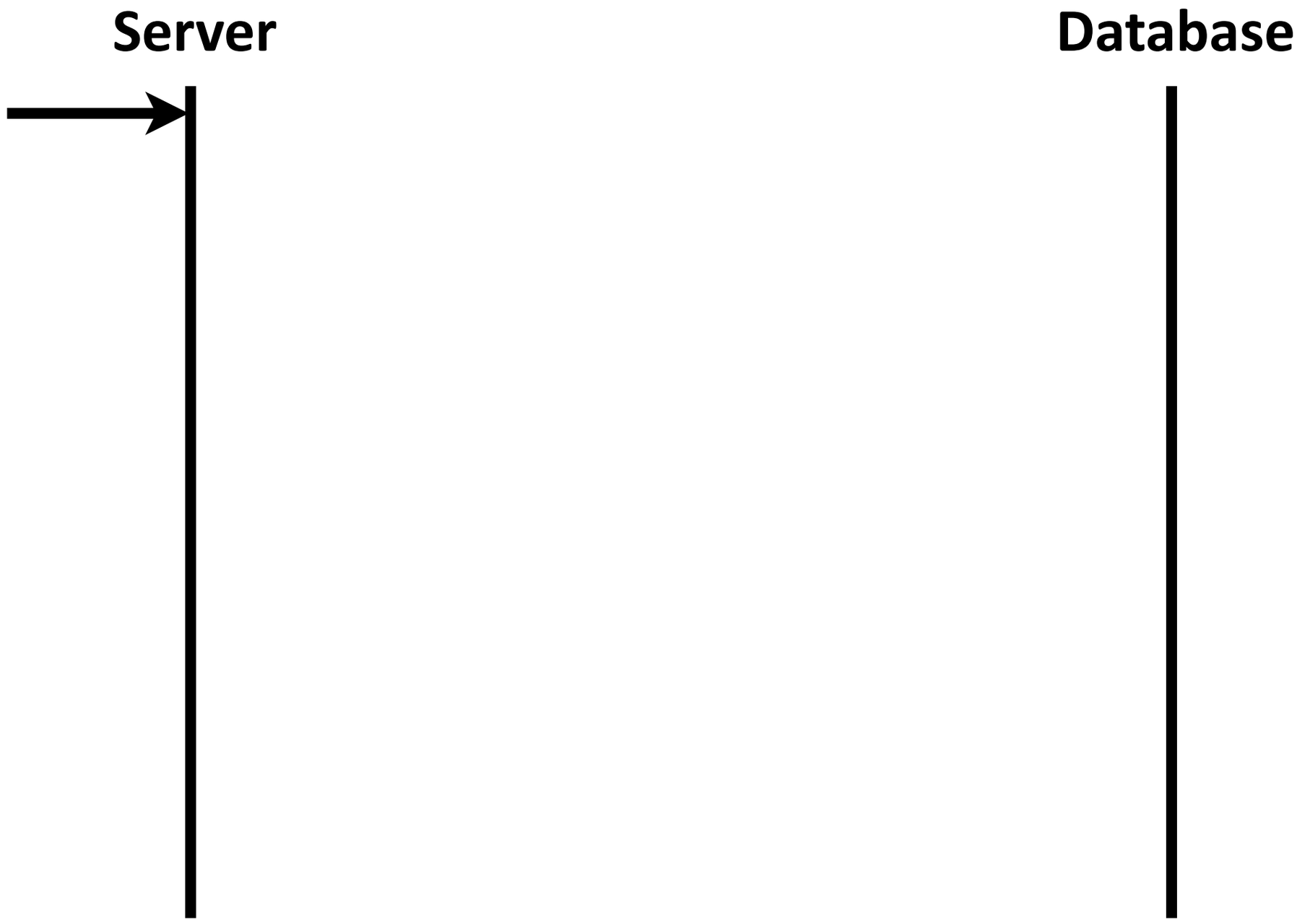
Stateless servers and DBs



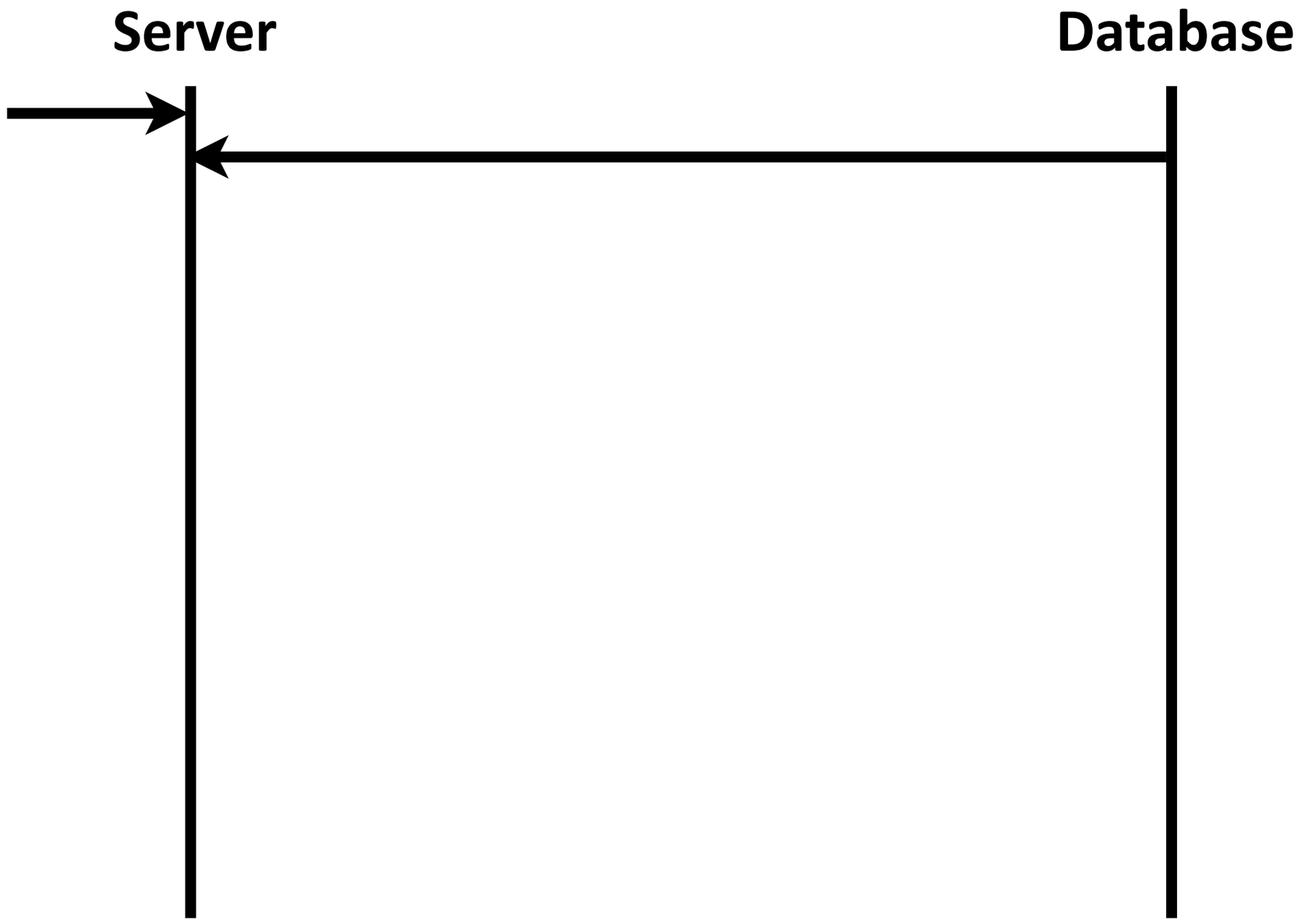
Stateless servers and DBs



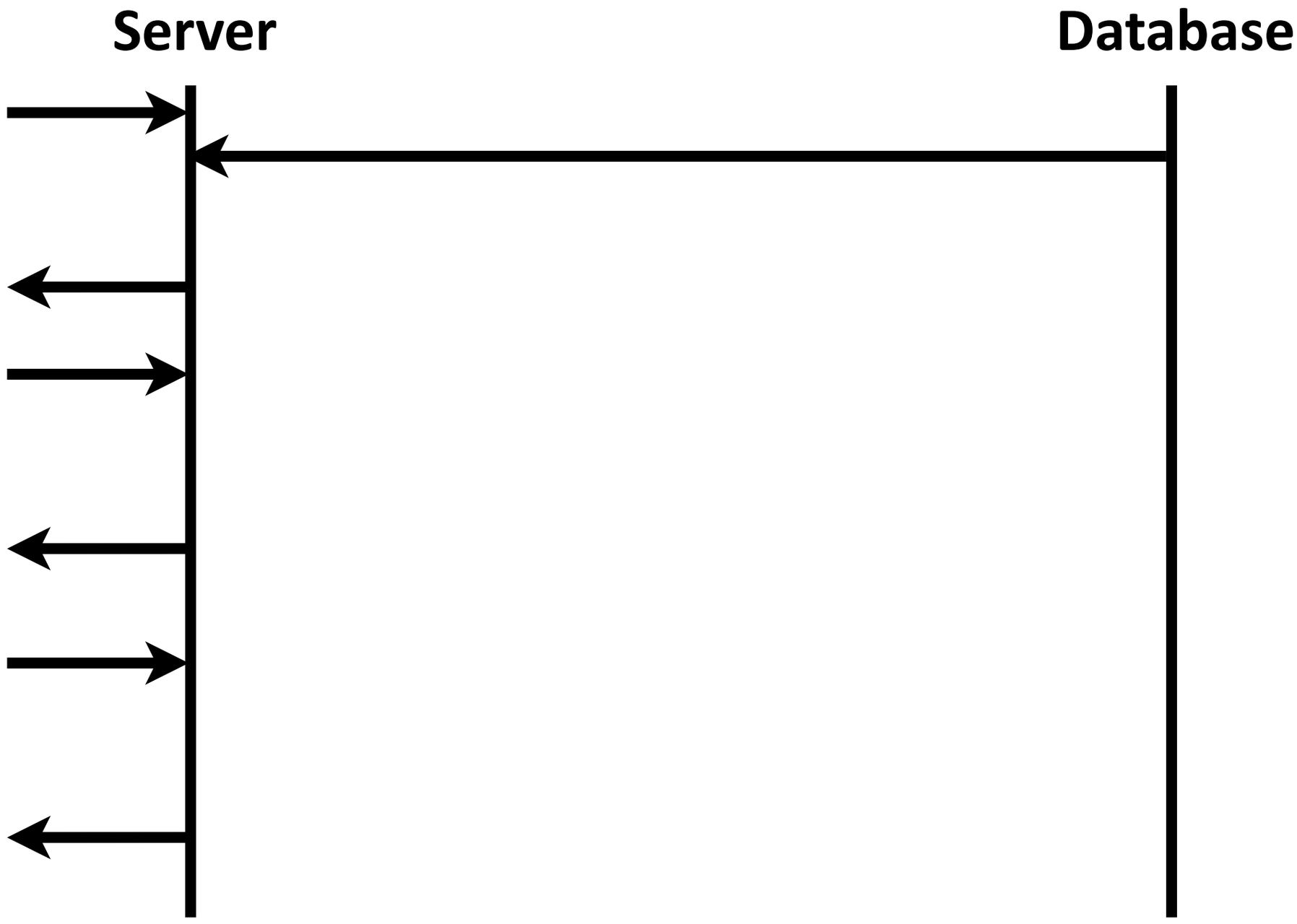
Stateful servers and DBs



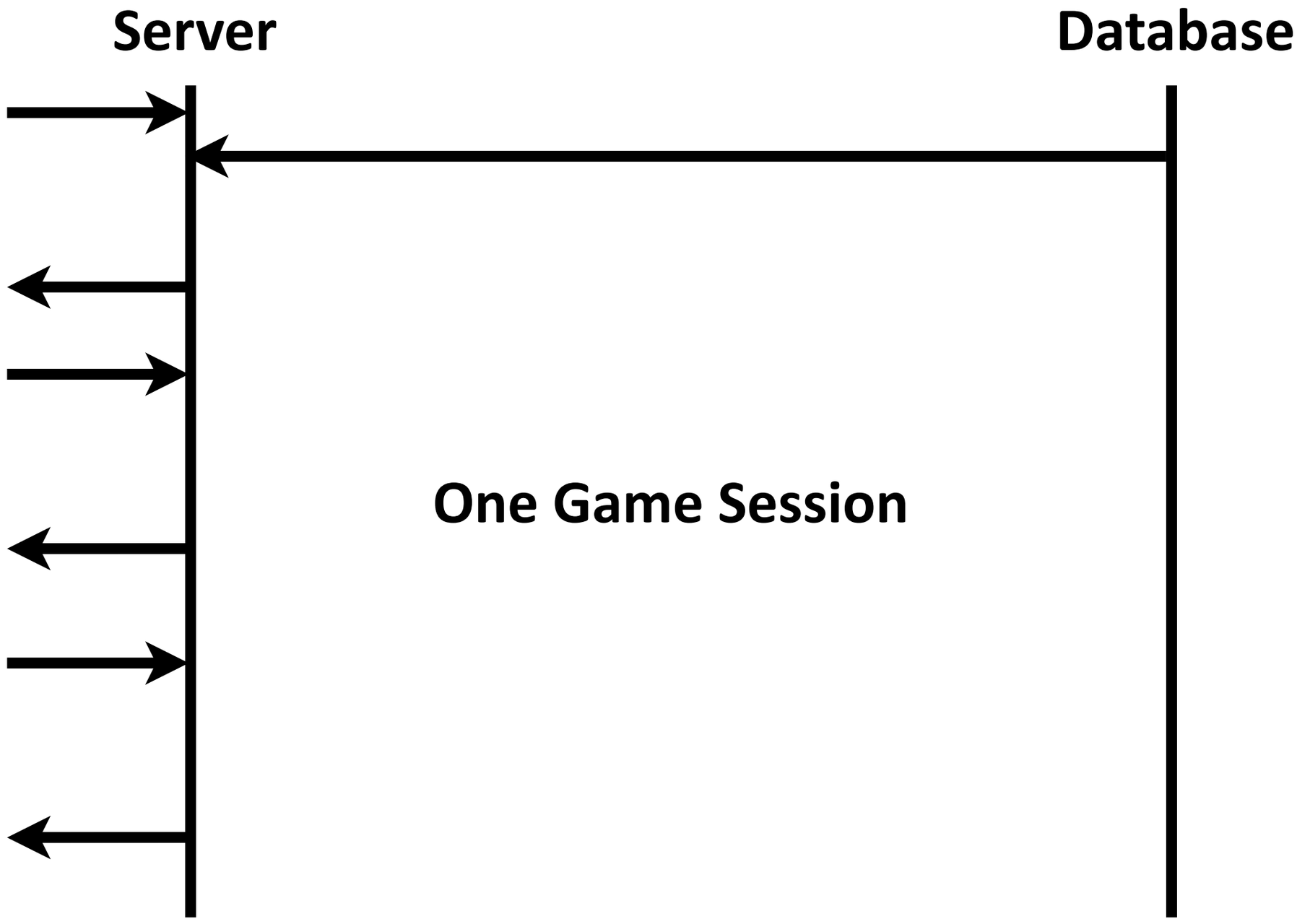
Stateful servers and DBs



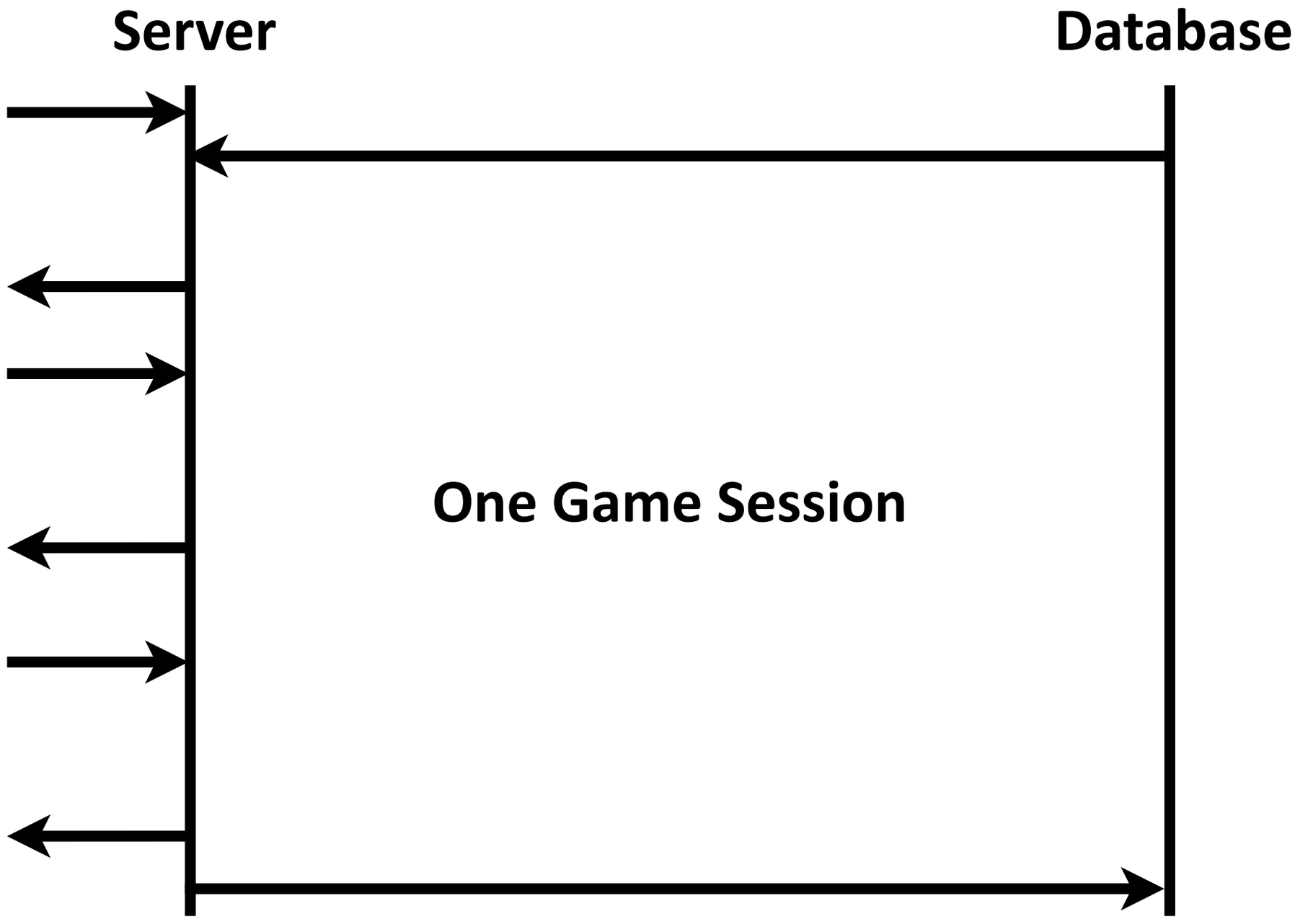
Stateful servers and DBs



Stateful servers and DBs



Stateful servers and DBs



Oct 2010: 3rd team used a stateful server

If DBs are the problem

Don’t use them

Store state in server

Need to be robust

Oct 2010: 3rd team used a stateful server

If DBs are the problem

Don’t use them

Store state in server

Need to be robust

Evolution III: Use Erlang for a stateful server

Magic Land

ERLANG

Oct 2010

Happy Hospital

Jan 2010

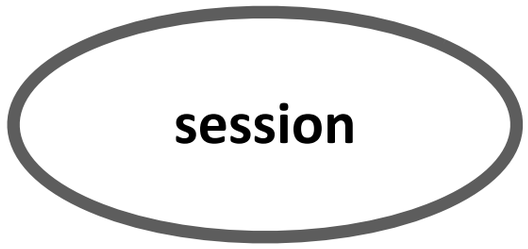
Monster World

Oct 2009

Bubble Island

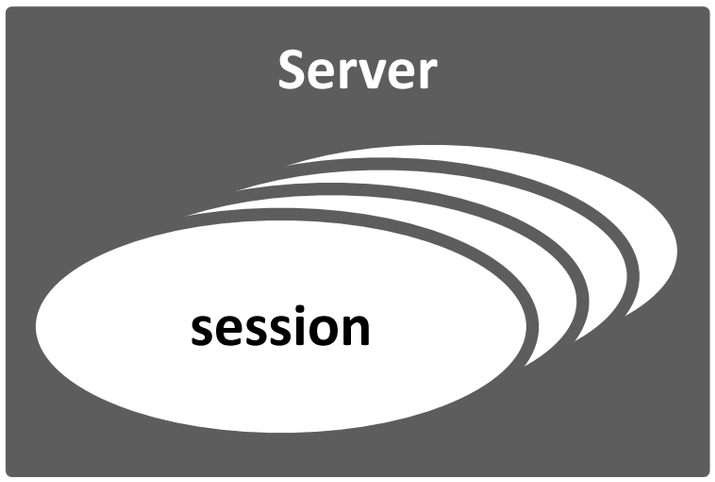
php

Stateful servers are not as hard as you think

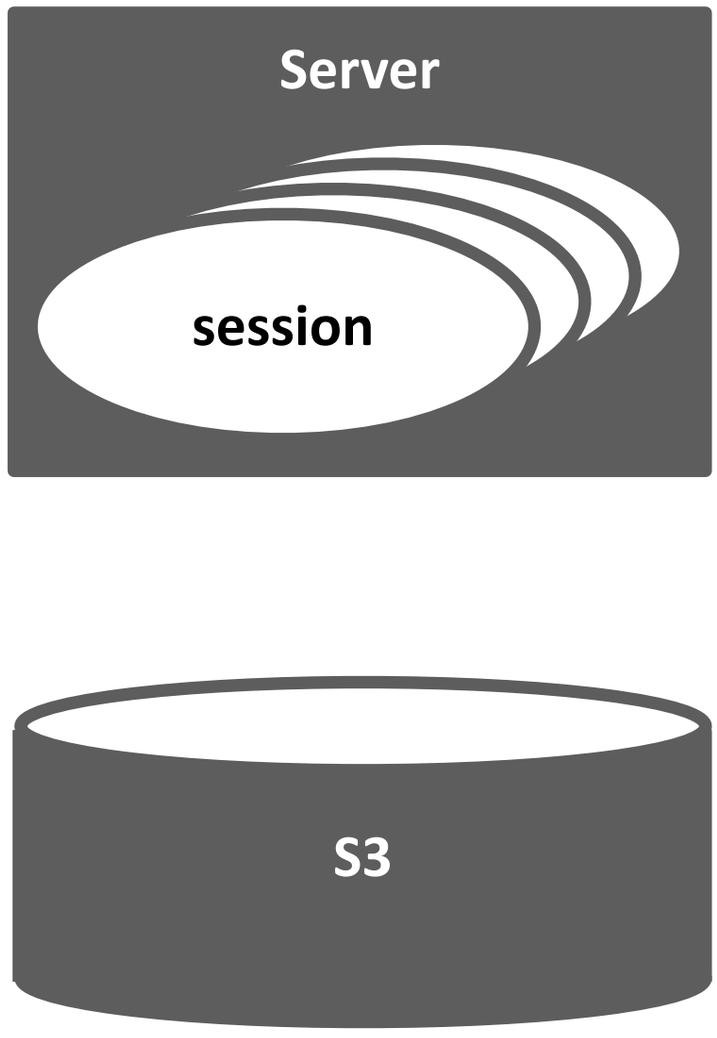


Stateful servers are not as hard as you think

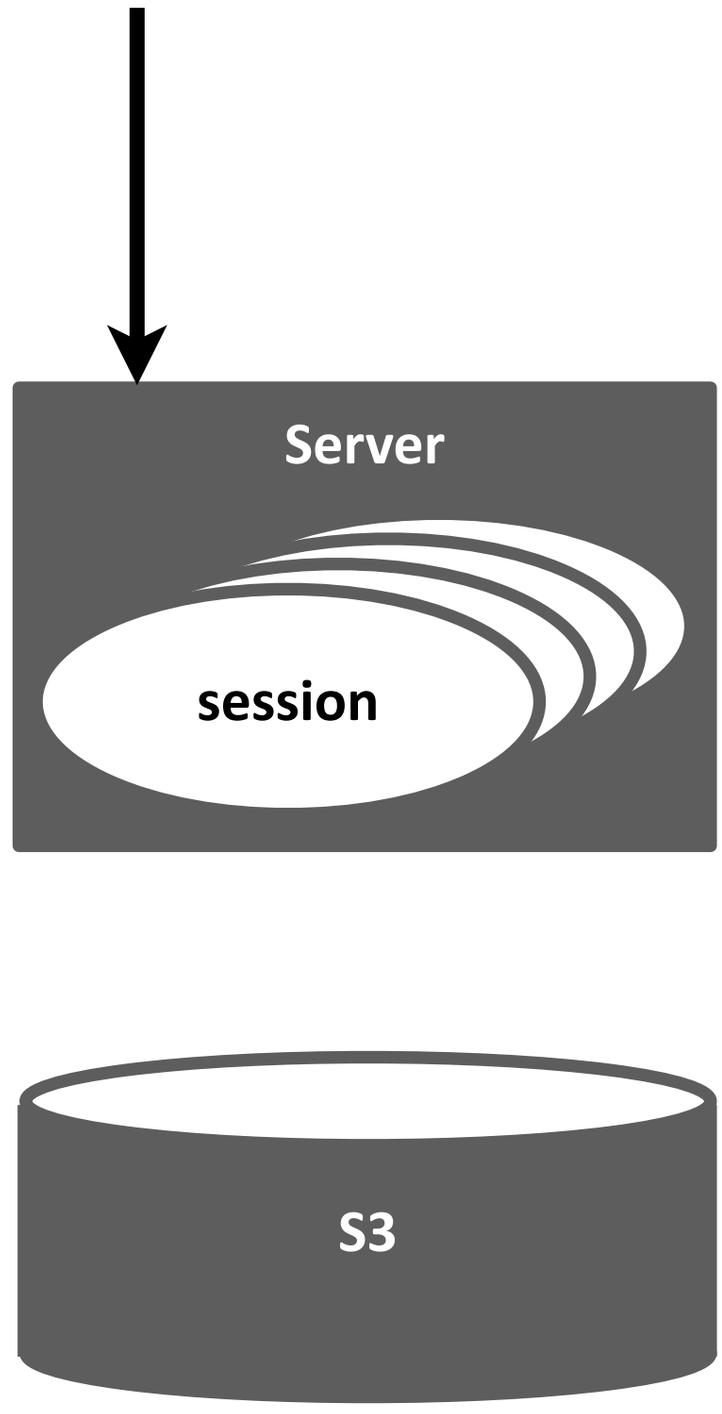
Stateful servers are not as hard as you think



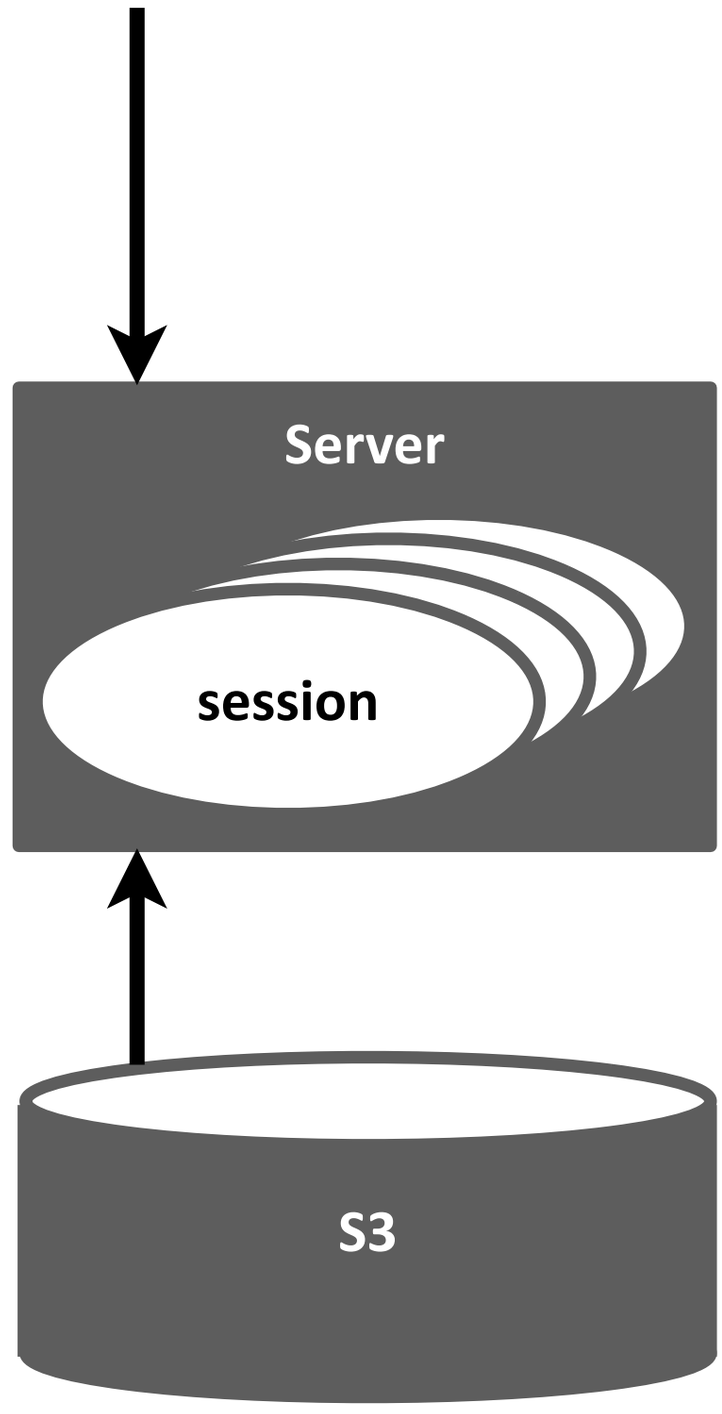
Stateful servers are not as hard as you think



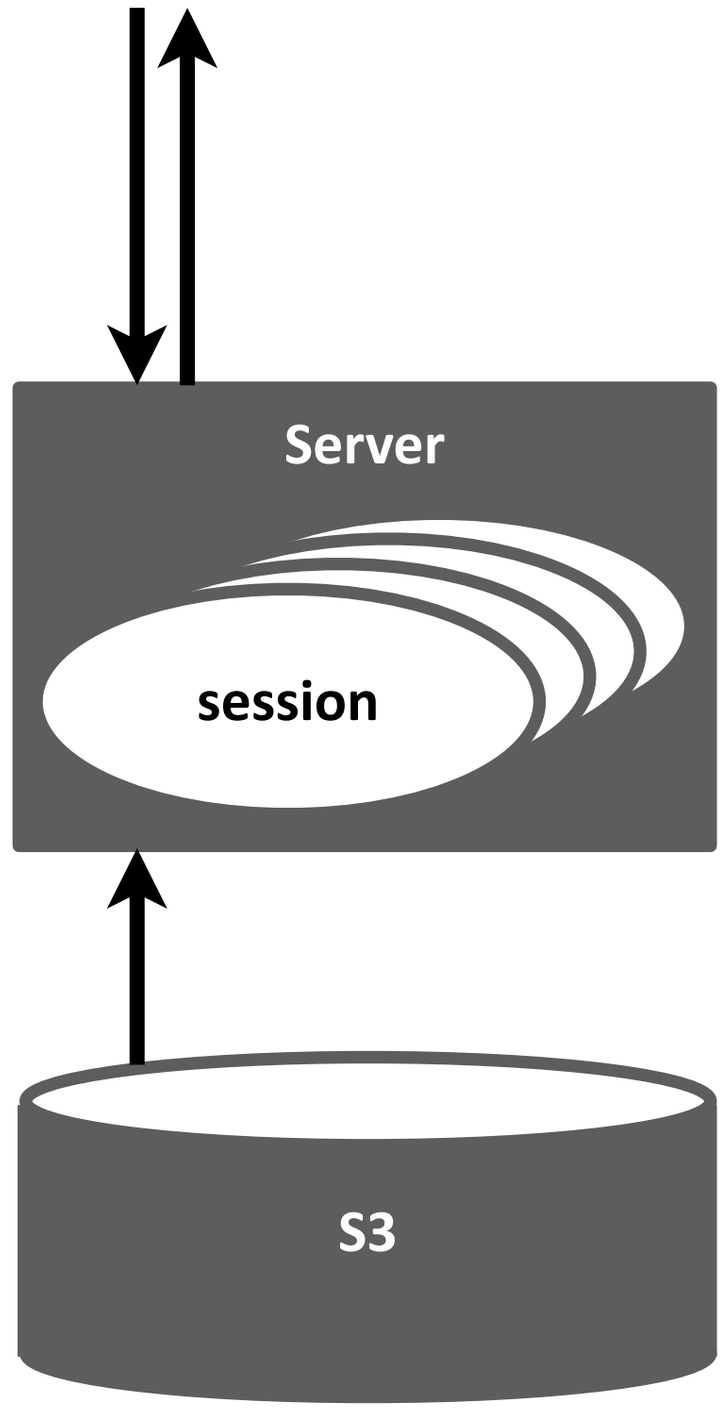
Stateful servers are not as hard as you think



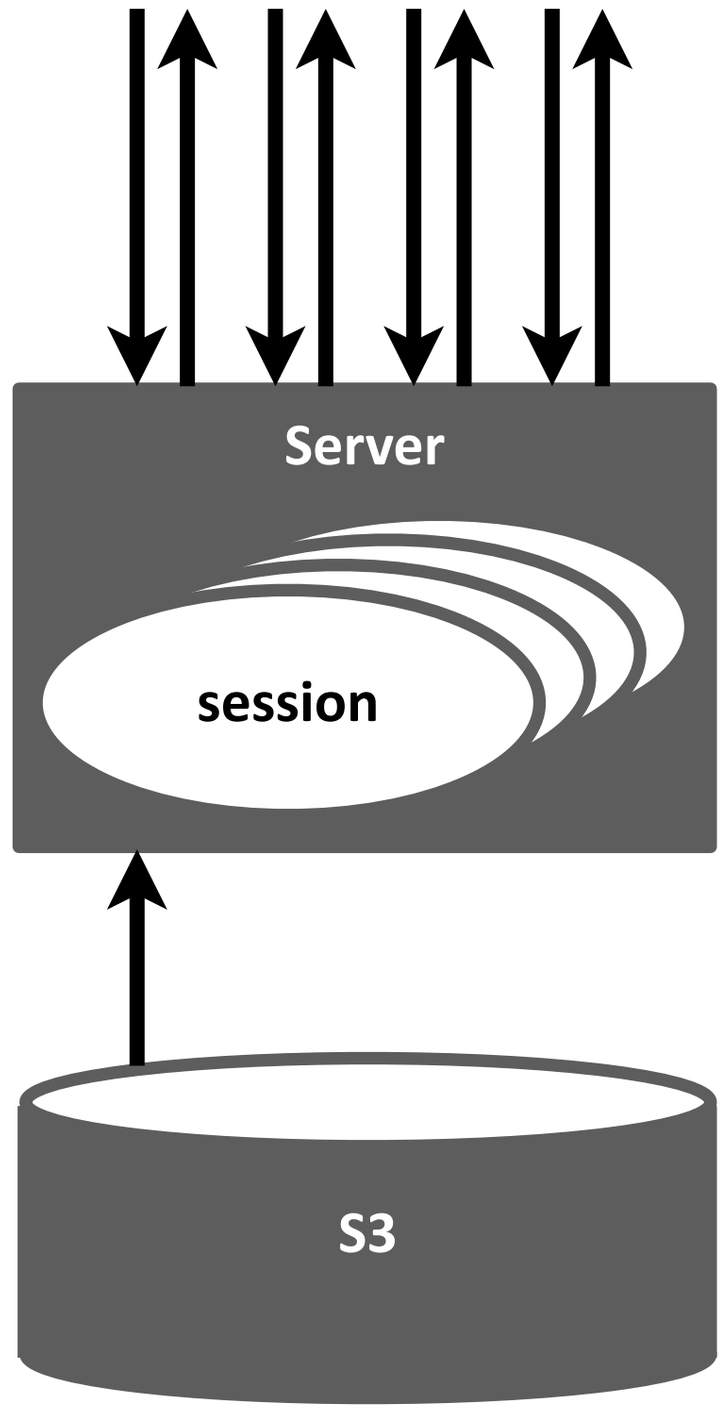
Stateful servers are not as hard as you think



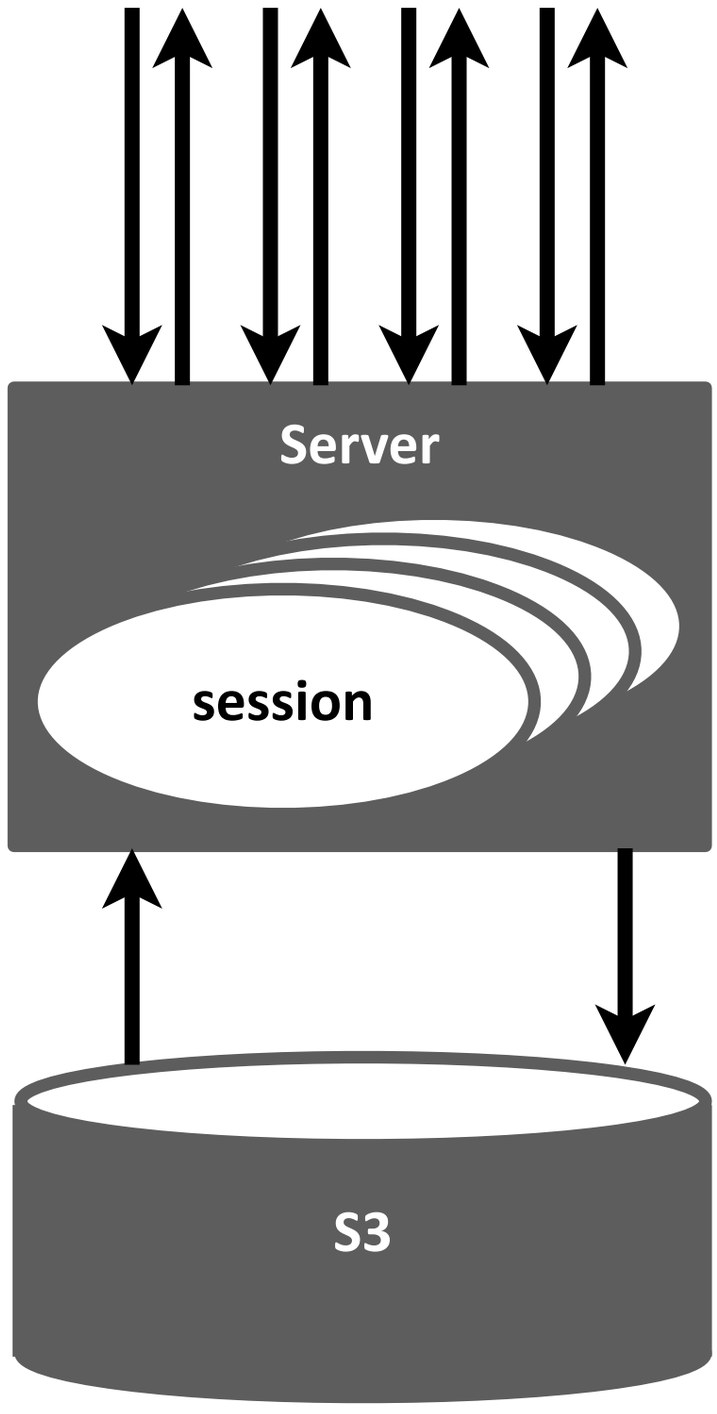
Stateful servers are not as hard as you think



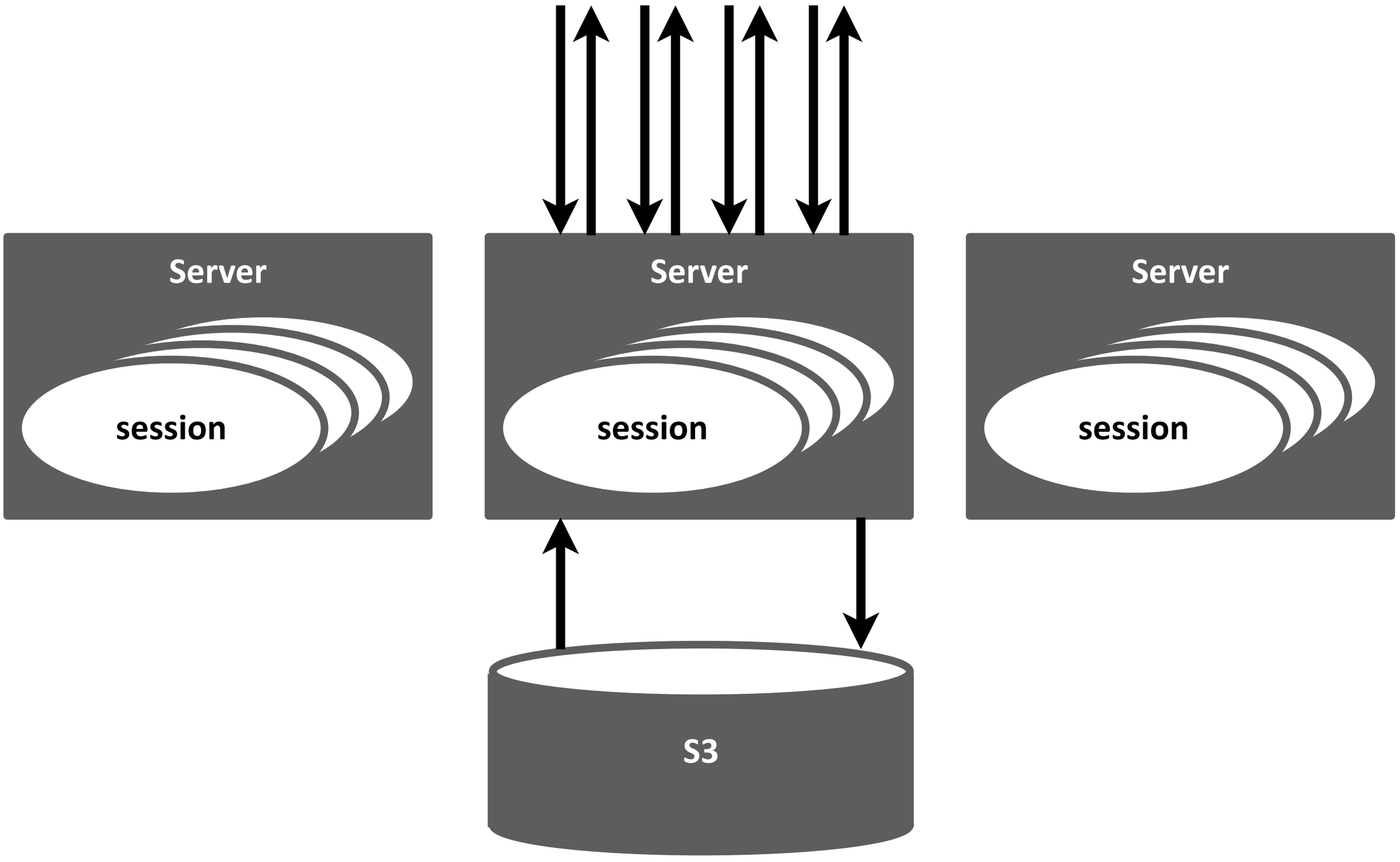
Stateful servers are not as hard as you think



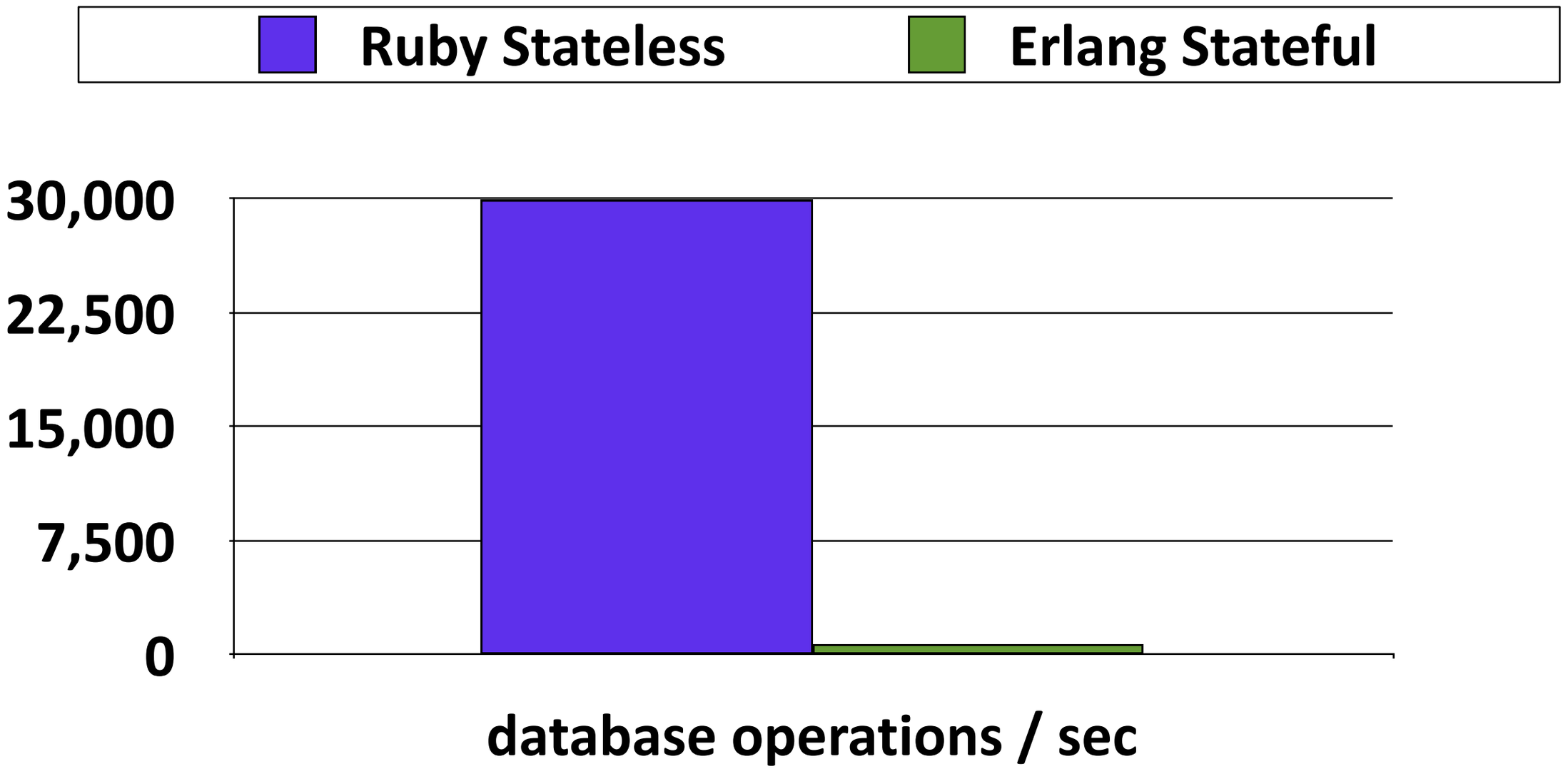
Stateful servers are not as hard as you think



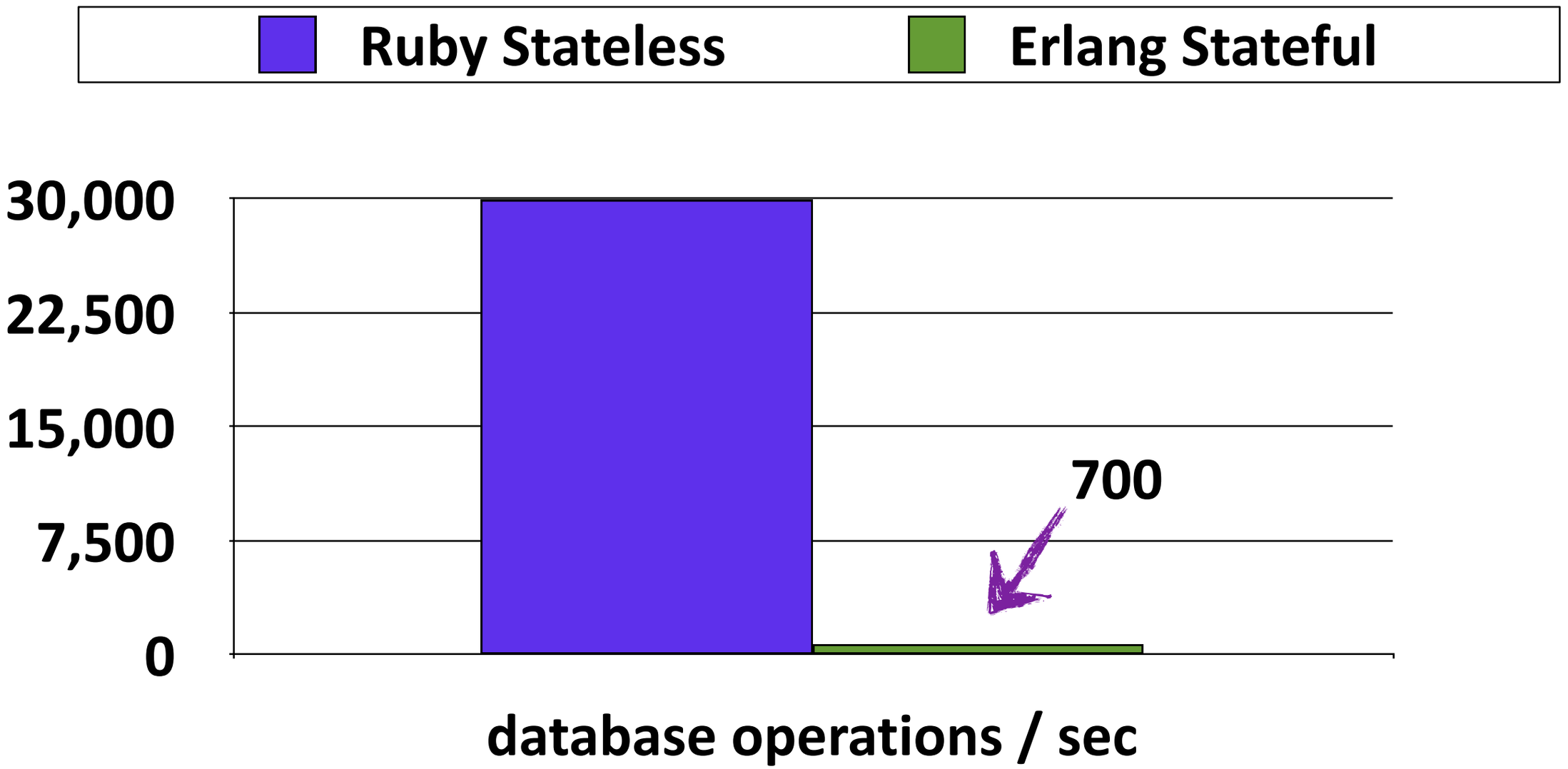
Stateful servers are not as hard as you think



With stateful server the DB is less used



With stateful server the DB is less used



Deploying with a stateful server

In order to bring up a new version

Deploying with a stateful server

In order to bring up a new version

Just deploy it

Hot code replacement is great!

There are even more advantages

Faster than Ruby (5,000 rps / node)

- CPU bound

There are even more advantages

Faster than Ruby (5,000 rps / node)

- CPU bound

Very few SPOFs

- ... and those are easy to recover

There are even more advantages

Faster than Ruby (5,000 rps / node)

- CPU bound

Very few SPOFs

- ... and those are easy to recover

Transactional logic

- Invariants instead of explicit error handling

Example “controller” in Erlang

```
%% Harvest or clear a seed. Check food capacity.
handle(?PLOTS, ActionA, Coords, Args, Game)
  when ActionA =:= harvest_seed;
       ActionA =:= clear_seed ->
    {Effects, NewMap} = k_map:handle(?PLOTS, ActionA, Coords, Args,
                                     map(Game)),

    {updated_game([fun (_G) -> {map, NewMap} end,
                   fun (G)  -> {user, k_user:refill(user(G))} end,
                   fun (G)  -> {user, k_user:update(Effects, user(G))} end,
                   fun (G)  -> check_food_cap(G) end],
                   Game),
     done};
```

Example “controller” in Erlang

```
%% Harvest or clear a seed. Check food capacity.
handle(?PLOTS, ActionA, Coords, Args, Game)
  when ActionA =:= harvest_seed;
       ActionA =:= clear_seed ->
    {Effects, NewMap} = k_map:handle(?PLOTS, ActionA, Coords, Args,
                                     map(Game)),

    {updated_game([fun (_G) -> {map, NewMap} end,
                   fun (G)  -> {user, k_user:refill(user(G))} end,
                   fun (G)  -> {user, k_user:update(Effects, user(G))} end,
                   fun (G)  -> check_food_cap(G) end],
                   Game),
     done};
```

Example “controller” in Erlang

```
%% Harvest or clear a seed. Check food capacity.
handle(?PLOTS, ActionA, Coords, Args, Game)
  when ActionA =:= harvest_seed;
       ActionA =:= clear_seed ->
    {Effects, NewMap} = k_map:handle(?PLOTS, ActionA, Coords, Args,
                                     map(Game)),

    {updated_game([fun (_G) -> {map, NewMap} end,
                   fun (G)  -> {user, k_user:refill(user(G))} end,
                   fun (G)  -> {user, k_user:update(Effects, user(G))} end,
                   fun (G)  -> check_food_cap(G) end],
                   Game),
     done};
```

Example “controller” in Erlang

```
%% Harvest or clear a seed. Check food capacity.
handle(?PLOTS, ActionA, Coords, Args, Game)
  when ActionA =:= harvest_seed;
       ActionA =:= clear_seed ->
    {Effects, NewMap} = k_map:handle(?PLOTS, ActionA, Coords, Args,
                                     map(Game)),
    {updated_game([fun (_G) -> {map, NewMap} end,
                   fun (G)  -> {user, k_user:refill(user(G))} end,
                   fun (G)  -> {user, k_user:update(Effects, user(G))} end,
                   fun (G)  -> check_food_cap(G) end],
                  Game),
     done};
```

Example “controller” in Erlang

```
%% Harvest or clear a seed. Check food capacity.
handle(?PLOTS, ActionA, Coords, Args, Game)
  when ActionA =:= harvest_seed;
       ActionA =:= clear_seed ->
    {Effects, NewMap} = k_map:handle(?PLOTS, ActionA, Coords, Args,
                                     map(Game)),

    {updated_game([fun (_G) -> {map, NewMap} end,
                   fun (G)  -> {user, k_user:refill(user(G))} end,
                   fun (G)  -> {user, k_user:update(Effects, user(G))} end,
                   fun (G)  -> check_food_cap(G) end],
                   Game),
     done};
```

Central handling of effects

Example “controller” in Erlang

```
%% Harvest or clear a seed. Check food capacity.
handle(?PLOTS, ActionA, Coords, Args, Game)
  when ActionA =:= harvest_seed;
       ActionA =:= clear_seed ->
    {Effects, NewMap} = k_map:handle(?PLOTS, ActionA, Coords, Args,
                                     map(Game)),

    {updated_game([fun (_G) -> {map, NewMap} end,
                   fun (G)  -> {user, k_user:refill(user(G))} end,
                   fun (G)  -> {user, k_user:update(Effects, user(G))} end,
                   fun (G)  -> check_food_cap(G) end],
                   Game),
     done};
```

Central handling of effects

Transactional behavior

Example model in Erlang

```
harvest(#seed{id = Id} = Seed, BoostPercentage) ->
    verify_seed_ready(Seed),
    verify_not_withered(Seed),

    ContractReward = k_seed_config:reward(Id),
    Reward = k_utils:ceil(ContractReward * (1 + BoostPercentage / 100)),

    UserEffects = [?GAIN_FOOD(Reward),
                   ?ADD_XP(k_seed_config:xp_reward(Id))],
{UserEffects, undefined}.
```

Example model in Erlang

```
harvest(#seed{id = Id} = Seed, BoostPercentage) ->
    verify_seed_ready(Seed),
    verify_not_withered(Seed),

    ContractReward = k_seed_config:reward(Id),
    Reward = k_utils:ceil(ContractReward * (1 + BoostPercentage / 100)),

    UserEffects = [?GAIN_FOOD(Reward),
                   ?ADD_XP(k_seed_config:xp_reward(Id))],
    {UserEffects, undefined}.
```

Example model in Erlang

```
harvest(#seed{id = Id} = Seed, BoostPercentage) ->
    verify_seed_ready(Seed),
    verify_not_withered(Seed),

    ContractReward = k_seed_config:reward(Id),
    Reward = k_utils:ceil(ContractReward * (1 + BoostPercentage / 100)),

    UserEffects = [?GAIN_FOOD(Reward),
                   ?ADD_XP(k_seed_config:xp_reward(Id))],
{UserEffects, undefined}.
```

Example model in Erlang

```
harvest(#seed{id = Id} = Seed, BoostPercentage) ->
    verify_seed_ready(Seed),
    verify_not_withered(Seed),

    ContractReward = k_seed_config:reward(Id),
    Reward = k_utils:ceil(ContractReward * (1 + BoostPercentage / 100)),

    UserEffects = [?GAIN_FOOD(Reward),
                   ?ADD_XP(k_seed_config:xp_reward(Id))],
{UserEffects, undefined}.
```

Example model in Erlang

```
harvest(#seed{id = Id} = Seed, BoostPercentage) ->
    verify_seed_ready(Seed),
    verify_not_withered(Seed),

    ContractReward = k_seed_config:reward(Id),
    Reward = k_utils:ceil(ContractReward * (1 + BoostPercentage / 100)),

    UserEffects = [?GAIN_FOOD(Reward),
                   ?ADD_XP(k_seed_config:xp_reward(Id))],
    {UserEffects, undefined}.
```

Example model in Erlang

```
harvest(#seed{id = Id} = Seed, BoostPercentage) ->
    verify_seed_ready(Seed),
    verify_not_withered(Seed),

    ContractReward = k_seed_config:reward(Id),
    Reward = k_utils:ceil(ContractReward * (1 + BoostPercentage / 100)),

    UserEffects = [?GAIN_FOOD(Reward),
                   ?ADD_XP(k_seed_config:xp_reward(Id))],
{UserEffects, undefined}.
```

Erlang code is not that hard to read, isn't it?

http://www.flickr.com/photos/hotreactor/

Architecture Evolution at Wooga

The Start: Ruby

The Next Step: Erlang

Best of Two Worlds

Company Values

Aug 2011: 4th team wanted both

Erlang is great

Concurrency, robustness

Great for operation

Aug 2011: 4th team wanted both

Erlang is great

Concurrency, robustness

Great for operation

Ruby is great

Concise, expressive, testable

Great for development

Aug 2011: 4th team wanted both

Erlang is great

Concurrency, robustness

Great for operation

Ruby is great

Concise, expressive, testable

Great for development

Aug 2011: 4th team wanted both

Erlang is great

Concurrency, robustness

Great for operation

Ruby is great

Concise, expressive, testable

Great for development

Evolution IV: The best out of two worlds

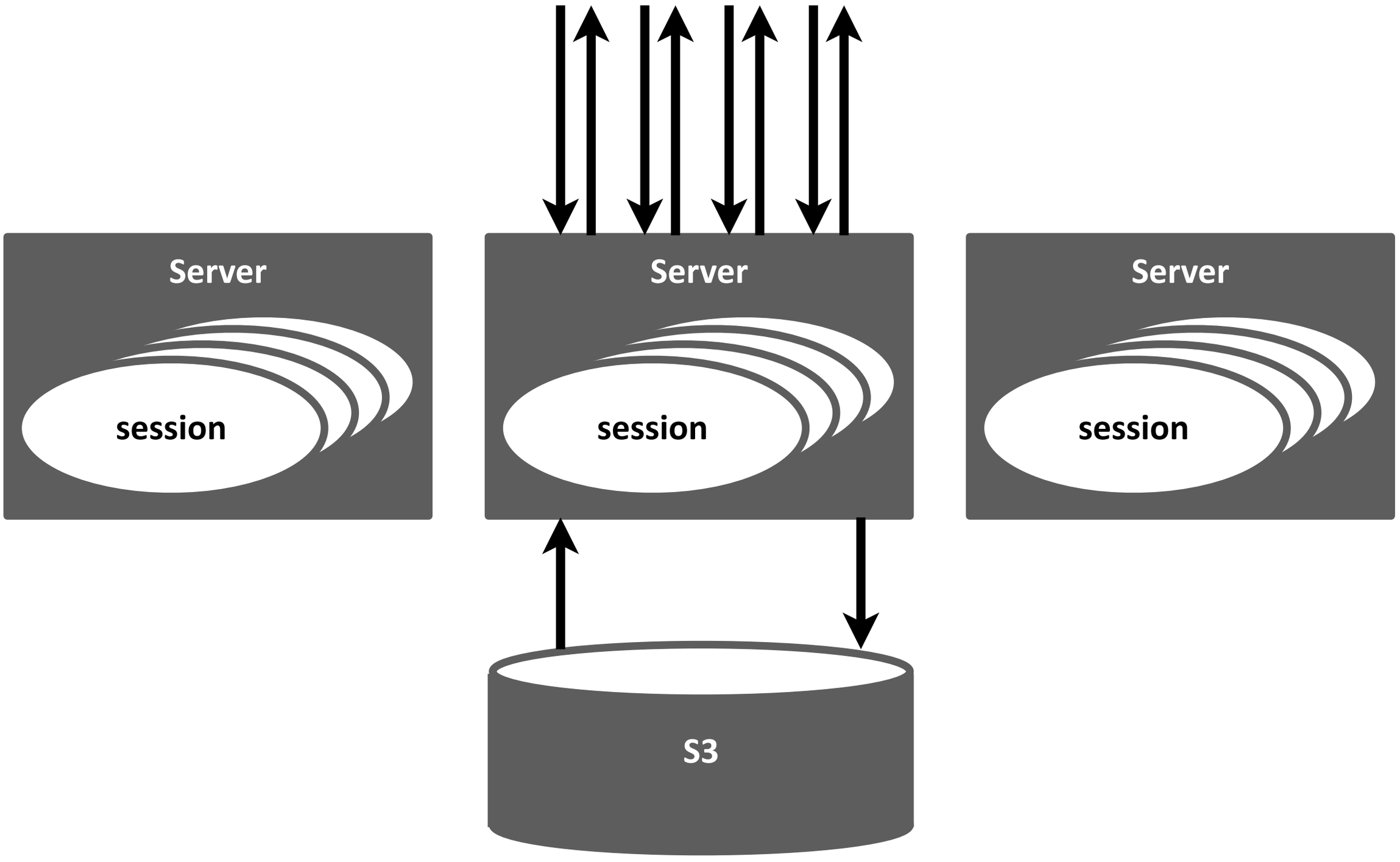
Aug 2011

Oct 2010

Jan 2010

Oct 2009

The basic setup looks exactly like before



Example controller in Ruby

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Example controller in Ruby

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Example controller in Ruby

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Example controller in Ruby

```ruby
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Example controller in Ruby

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

DSL-like definition of game action

Example controller in Ruby

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

DSL-like definition of game action

Skinny as controllers should be

Example model in Ruby

```ruby
class FruitTree < Tree

  property :last_shake_time, :type => Integer, :default => 0
  property :collectable_fruit_count, :type => Integer, :default => 0

  def shake
    raise G8::Error::Validation, "no fruit!" unless carries_fruit?

    session.user.xp += 1
    session.user.energy -= 1
    self.last_shake_time = game_time
    self.collectable_fruit_count = config.fruit_count
  end

end
```

Example model in Ruby

```ruby
class FruitTree < Tree

  property :last_shake_time, :type => Integer, :default => 0
  property :collectable_fruit_count, :type => Integer, :default => 0

  def shake
    raise G8::Error::Validation, "no fruit!" unless carries_fruit?

    session.user.xp += 1
    session.user.energy -= 1
    self.last_shake_time = game_time
    self.collectable_fruit_count = config.fruit_count
  end

end
```

Example model in Ruby

```ruby
class FruitTree < Tree

  property :last_shake_time, :type => Integer, :default => 0
  property :collectable_fruit_count, :type => Integer, :default => 0

  def shake
    raise G8::Error::Validation, "no fruit!" unless carries_fruit?

    session.user.xp += 1
    session.user.energy -= 1
    self.last_shake_time = game_time
    self.collectable_fruit_count = config.fruit_count
  end

end
```

Example model in Ruby

```ruby
class FruitTree < Tree

  property :last_shake_time, :type => Integer, :default => 0
  property :collectable_fruit_count, :type => Integer, :default => 0

  def shake
    raise G8::Error::Validation, "no fruit!" unless carries_fruit?

    session.user.xp += 1
    session.user.energy -= 1
    self.last_shake_time = game_time
    self.collectable_fruit_count = config.fruit_count
  end

end
```

Example model in Ruby

```ruby
class FruitTree < Tree

  property :last_shake_time, :type => Integer, :default => 0
  property :collectable_fruit_count, :type => Integer, :default => 0

  def shake
    raise G8::Error::Validation, "no fruit!" unless carries_fruit?

    session.user.xp += 1
    session.user.energy -= 1
    self.last_shake_time = game_time
    self.collectable_fruit_count = config.fruit_count
  end

end
```

Easily unit testable

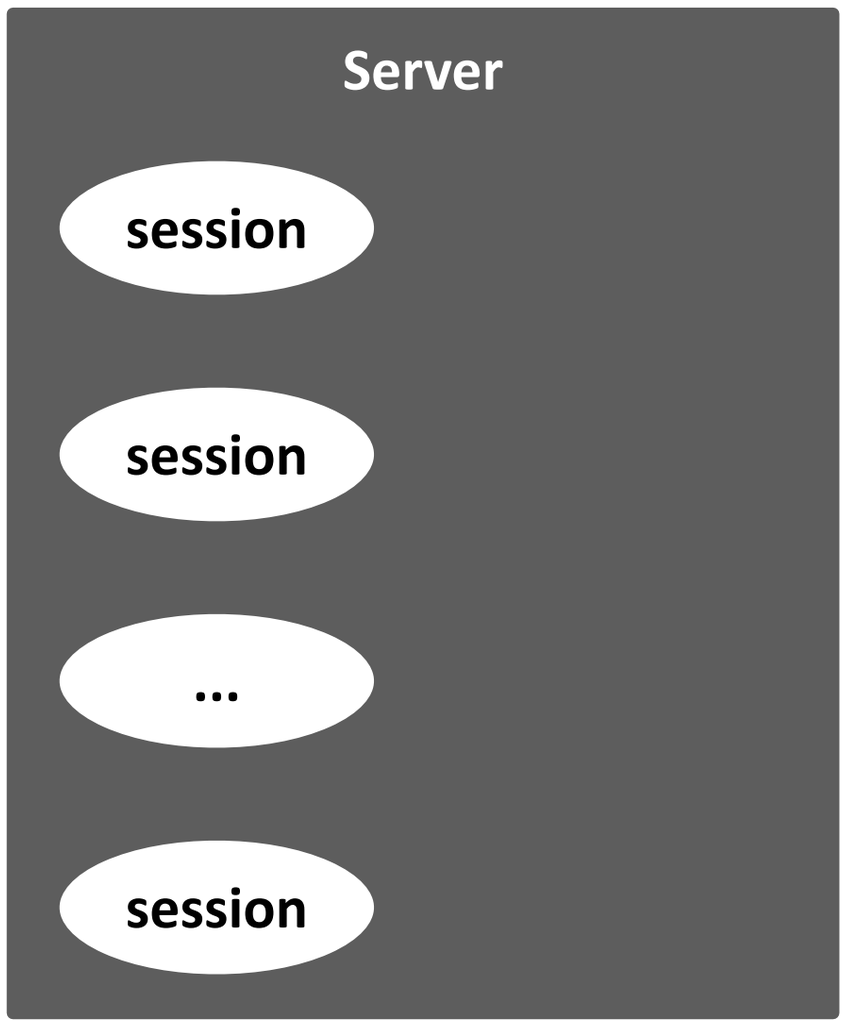
Example model in Ruby

```ruby
class FruitTree < Tree

  property :last_shake_time, :type => Integer, :default => 0
  property :collectable_fruit_count, :type => Integer, :default => 0

  def shake
    raise G8::Error::Validation, "no fruit!" unless carries_fruit?

    session.user.xp += 1
    session.user.energy -= 1
    self.last_shake_time = game_time
    self.collectable_fruit_count = config.fruit_count
  end

end
```

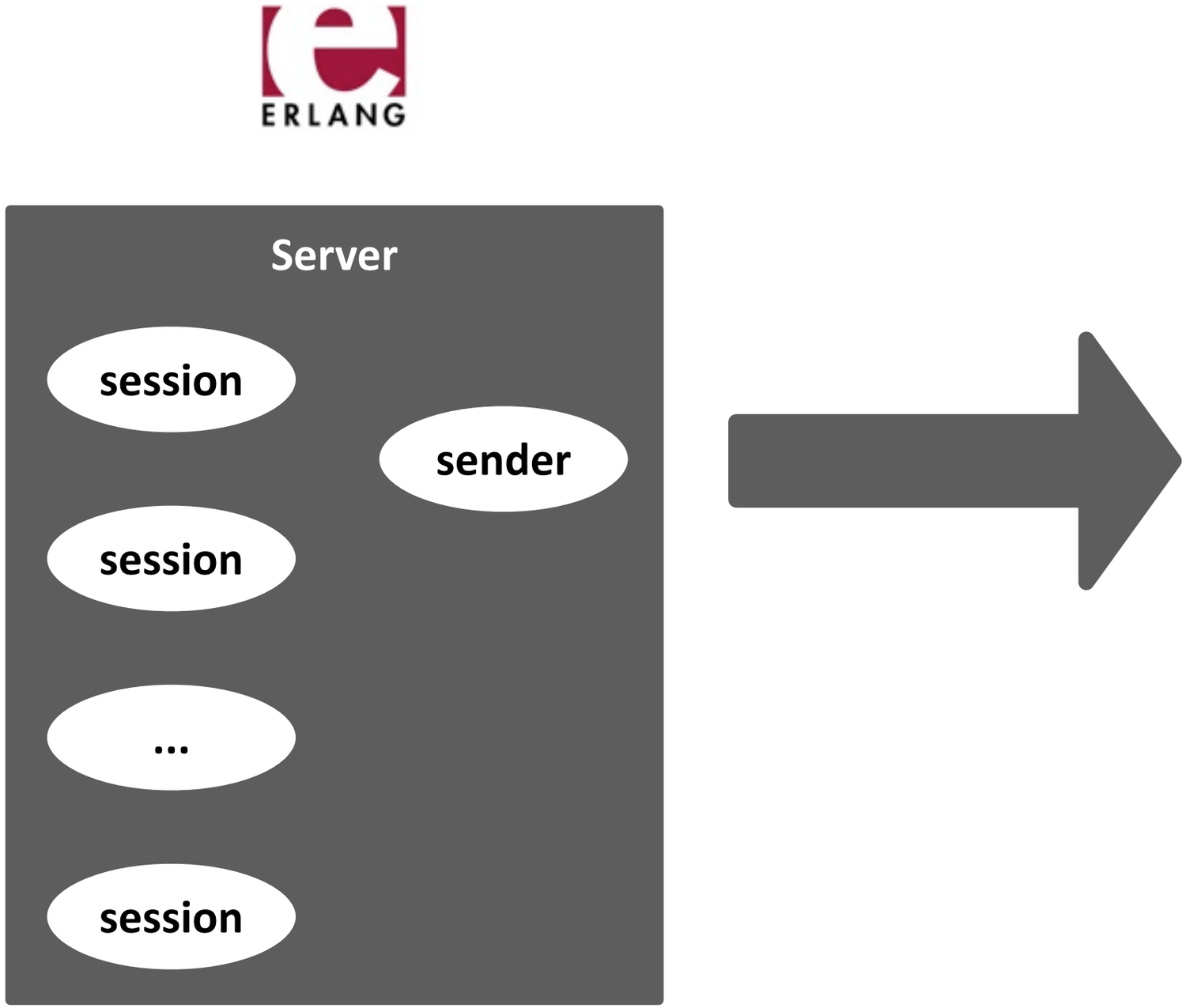
Easily unit testable

Minimal amount of code

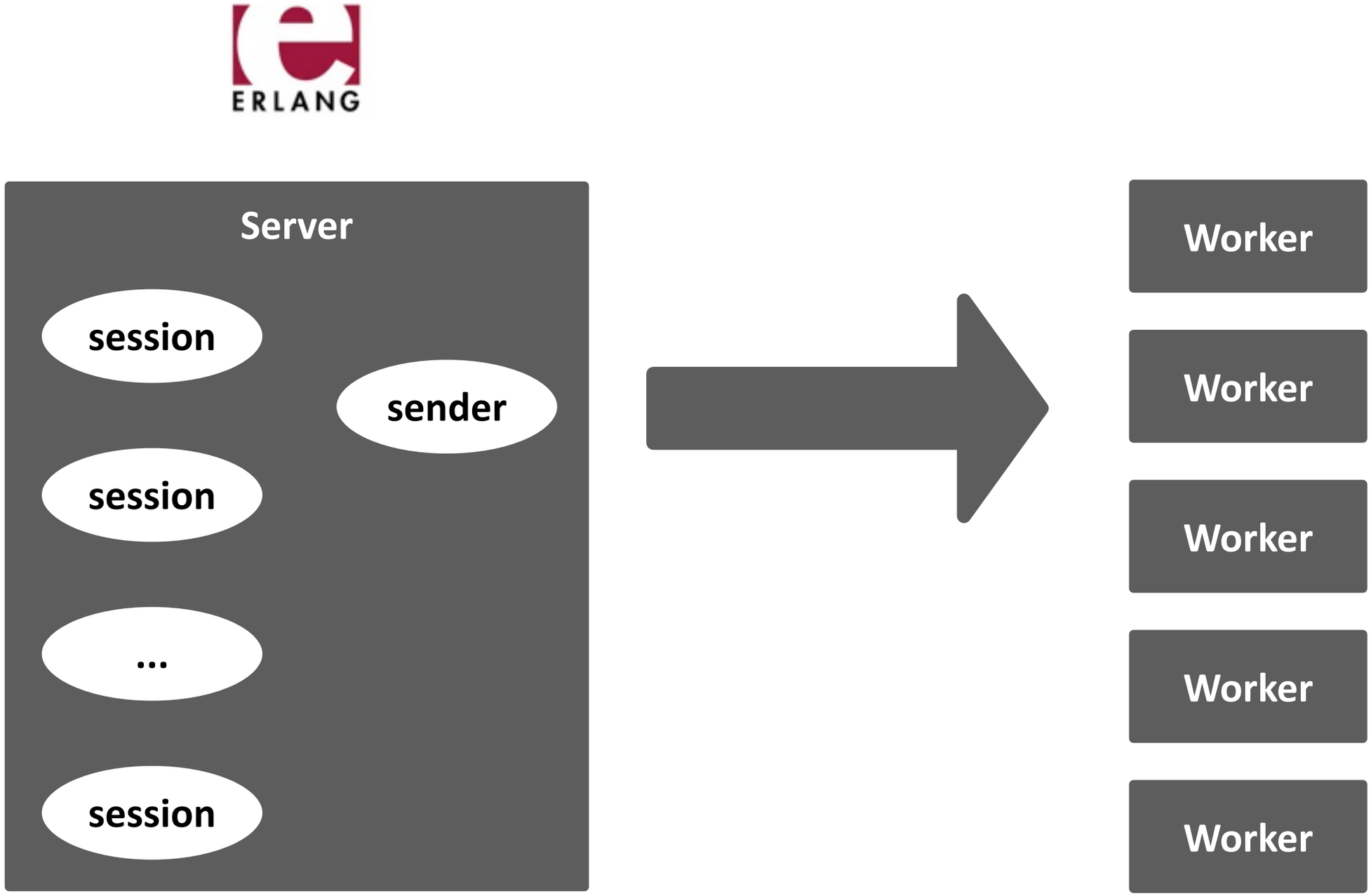
Bringing 2 worlds together



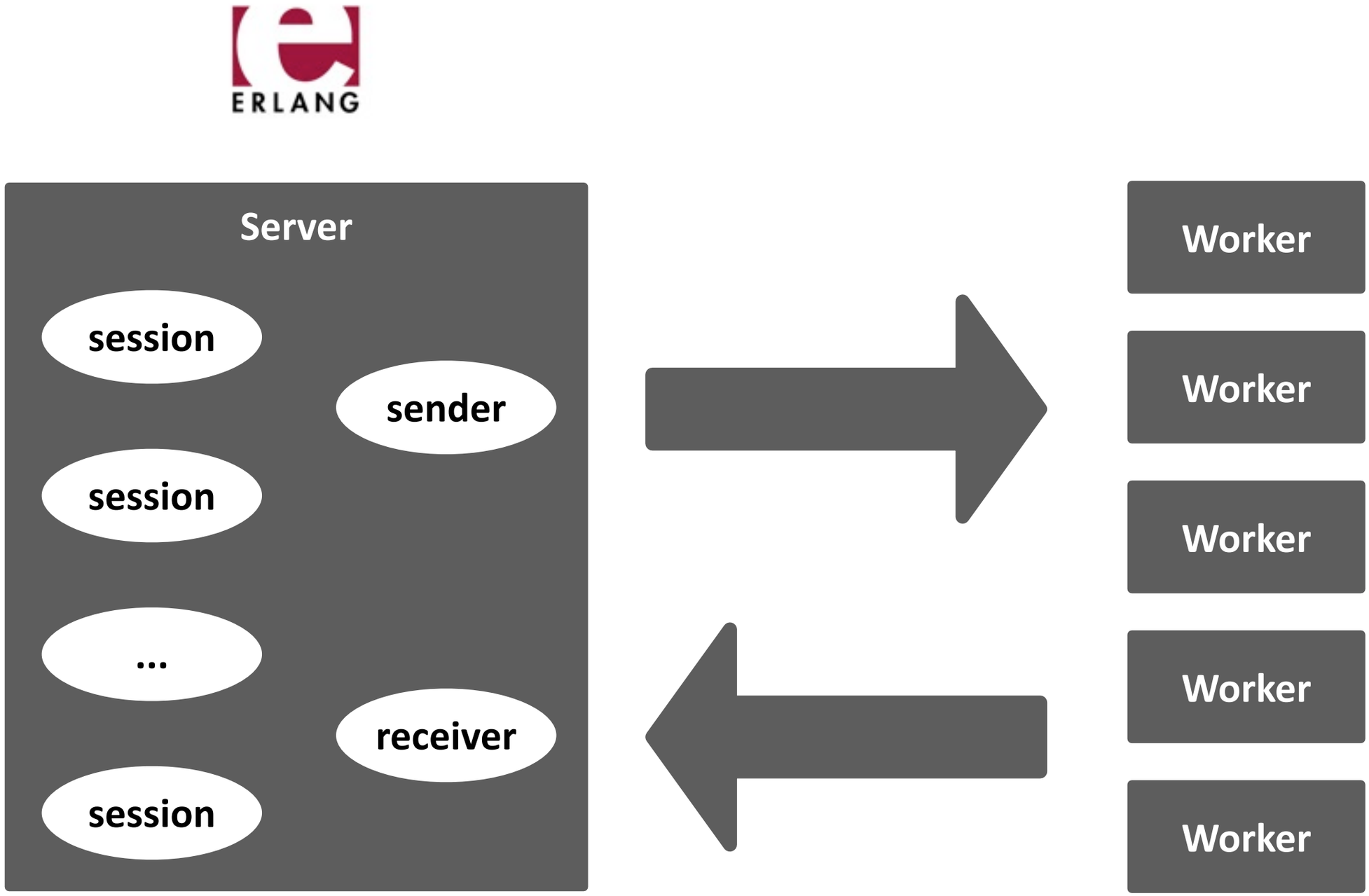
Bringing 2 worlds together



Bringing 2 worlds together



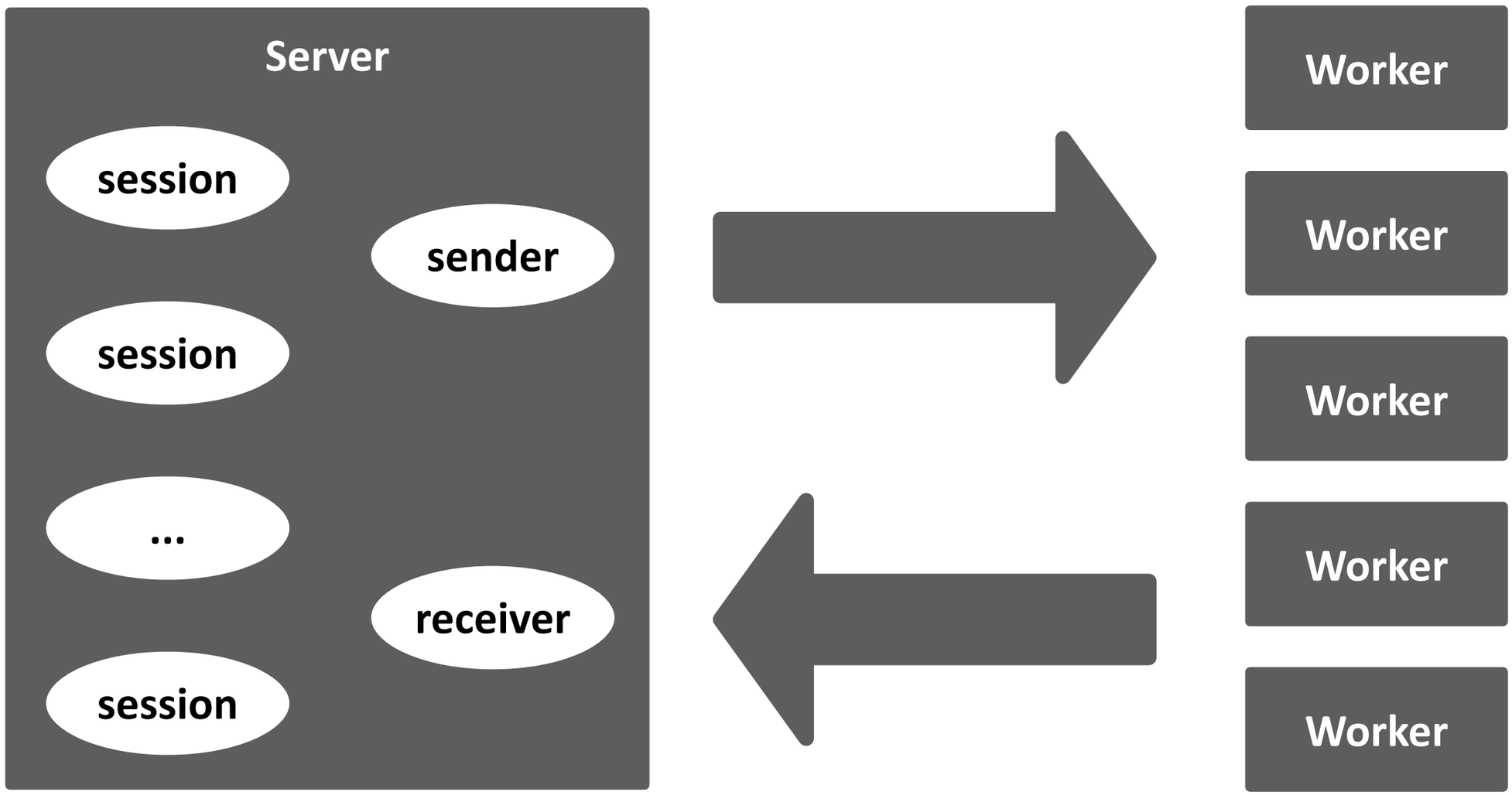
Bringing 2 worlds together



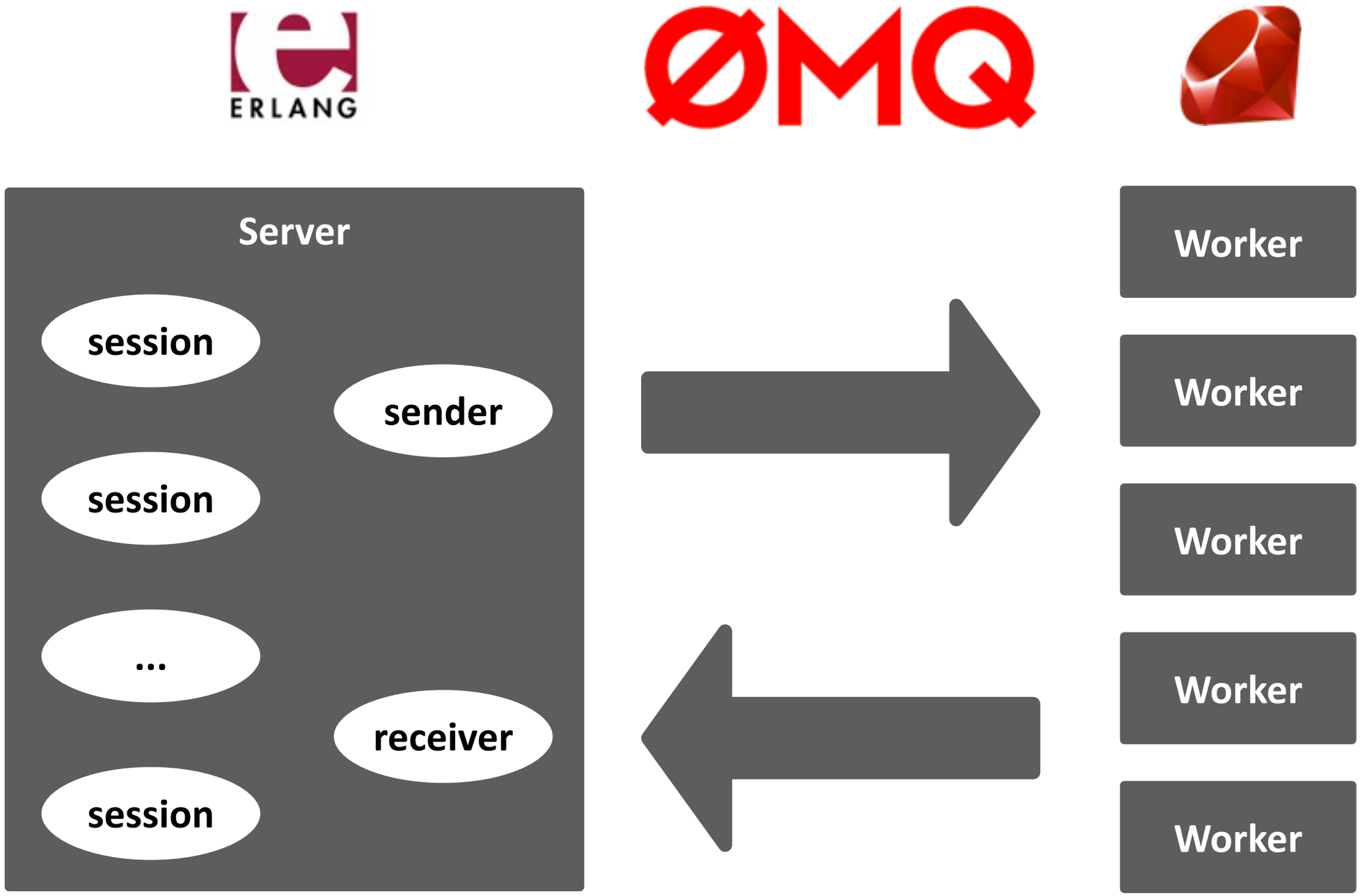
Bringing 2 worlds together

ERLANG

ØMQ



Bringing 2 worlds together



Game state

Game state is split in multiple parts

user, map, fruit_trees etc.

Game state

Game state is split in multiple parts

user, map, fruit_trees etc.

Erlang does not care about content

Serialized Ruby objects

Game state

Game state is split in multiple parts

user, map, fruit_trees etc.

Erlang does not care about content

Serialized Ruby objects

Erlang does know mapping of state parts to URLs

Mapping provided by Ruby on startup

Looking back at the game action

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Looking back at the game action

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Looking back at the game action

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Mapping of state parts to game actions

Looking back at the game action

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Mapping of state parts to game actions

Worker knows mapping

Looking back at the game action

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Mapping of state parts to game actions

- Worker knows mapping
- Worker pushes mapping to Erlang on startup

Looking back at the game action

```
game_action '/:actor/fruit_tree/self/shake',
            :affects => [:fruit_trees, :user] do |response|

  x, y = params[:x], params[:y]
  fruit_trees[x, y].shake
end
```

Mapping of state parts to game actions

- Worker knows mapping
- Worker pushes mapping to Erlang on startup
- Erlang can query mapping if needed

NICE!

http://www.flickr.com/photos/aigle_dore/

Architecture Evolution at Wooga

The Start: Ruby

The Next Step: Erlang

Best of Two Worlds

Company Values

Each new game brought us innovation

Bubble Island

php

Oct 2009

Monster World

Happy Hospital

Jan 2010

Magic Land

ERLANG

Oct 2010

GAME

ERLANG

Aug 2011

We've learned to value

Small teams over big teams

We've learned to value

Collaboration over competition

We've learned to value

Generalists over specialists

We've learned to value

Effort reduction

over

cost reduction

We've learned to value

Innovation

over

risk mitigation

A good value system

We've learned to value

Small teams	over	Big teams
Collaboration	over	Competition
Generalists	over	Specialists
Effort reduction	over	Cost reduction
Innovation	over	Risk mitigation

It works!

It works!

Be fast, be bold!

Questions?

Jesper Richter-Reichhelm
@jrirei

slideshare.net/wooga
wooga.com/jobs

wooga
world of gaming