Concurrent Ruby Application Servers

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

http://godfat.org/slide/2012-12-07-concurrent.pdf

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

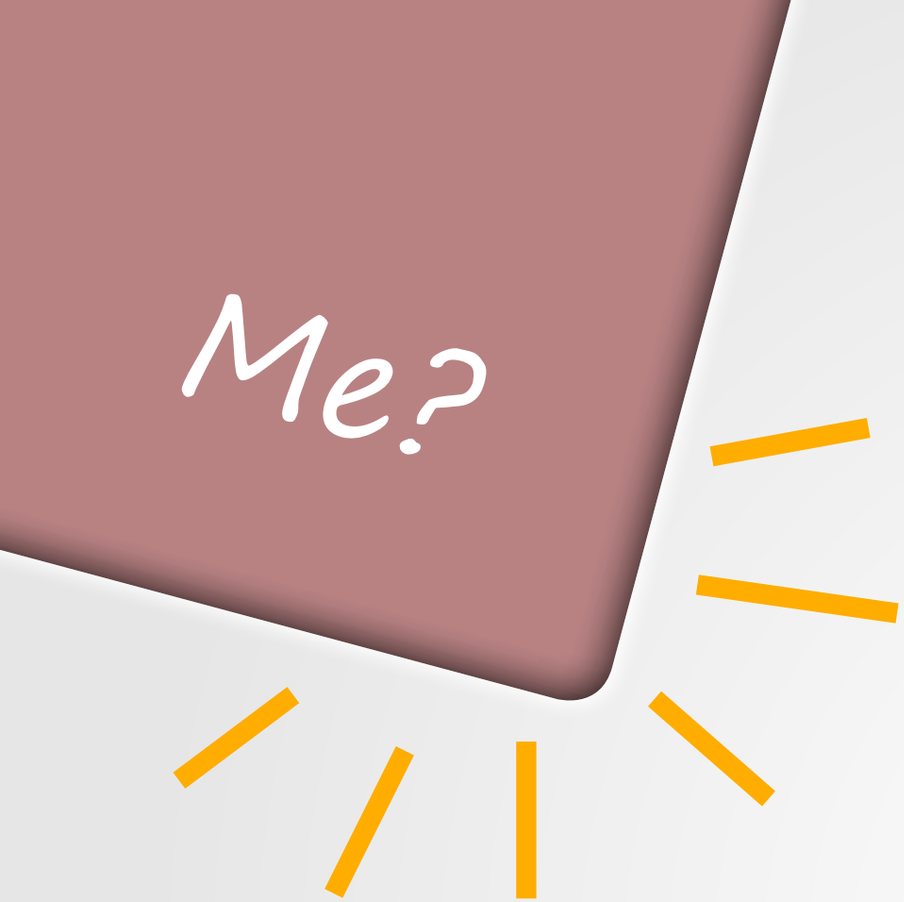
Me?

Concurrency?

What We Have?

Q?

App Servers?



Concurrency?

What We Have?

App Servers?

- Programmer at Cardinal Blue

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

- Programmer at Cardinal Blue
- Use Ruby from 2006

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

- Programmer at Cardinal Blue
- Use Ruby from 2006
- Interested in PL and FP (e.g. Haskell)

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

- Programmer at Cardinal Blue
- Use Ruby from 2006
- Interested in

Also concurrency recently

- Programmer at Cardinal Blue
- Use Ruby from 2006
- Interested in Also concurrency recently
- https://github.com/godfat

- Programmer at Cardinal Blue
- Use Ruby from 2006
- Interested in Also concurrency recently
- https://github.com/godfat
- https://twitter.com/godfat

PicCollage

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

- ~3.1k requests per minute
- Average response time: ~135 ms
- Average concurrent requests per process: ~7
- Total processes: 18

- Above data observed from NewRelic
- App server: Zbatery with EventMachine and thread pool on Heroku Cedar stack

PicCollage in 7 days

Recent Gems

- jellyfish - Pico web framework for building API-centric web applications
- rib - Ruby-Interactive-ruBy -- Yet another interactive Ruby shell
- rest-core - Modular Ruby clients interface for REST APIs
- rest-more - Various REST clients such as Facebook and Twitter built with rest-core

Special Thanks

ihower, ET Blue and Cardinal Blue

Lin Jen-Shin (godfat) @ Rubyconf.tw/2012

Me?

Concurrency?

What We Have?

App Servers?

Q?

Caveats

No disk I/O

No pipelined requests

No chunked encoding

No web sockets

No …

To make things simpler for now.

Caution It's not faster for a user

10 moms can't produce

1 child in 1 month

10 moms can produce

10 children in 10 months

Resource matters

Resource matters

We might not always have
enough processors

Resource matters

We might not always have enough processors

We need multitasking

Imagine 15 children
have to be context switched
amongst 10 moms

Multitasking is not free

Multitasking is not free

If your program context switches,
then actually it runs slower

Multitasking is not free

If your program context switches,
then actually it runs slower

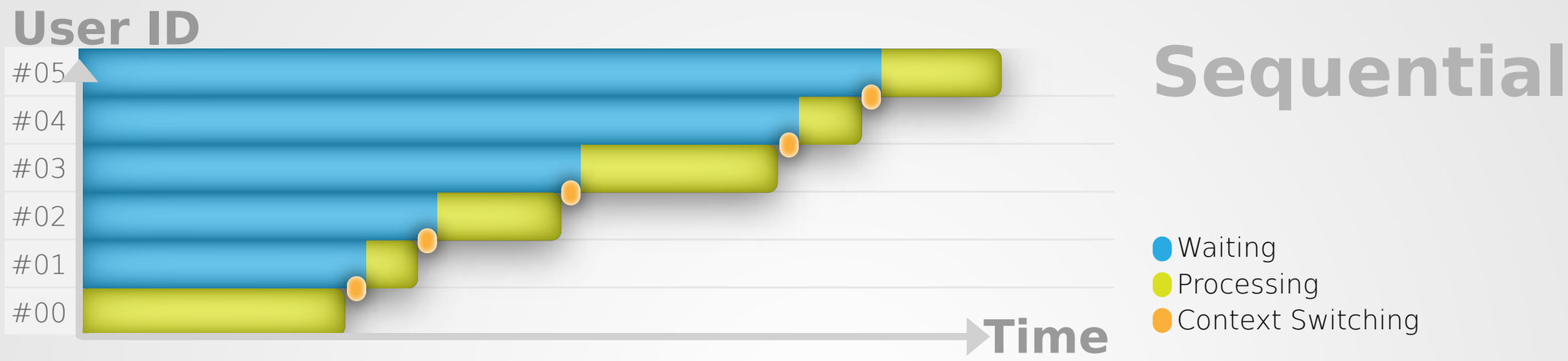
But it's more fair for users

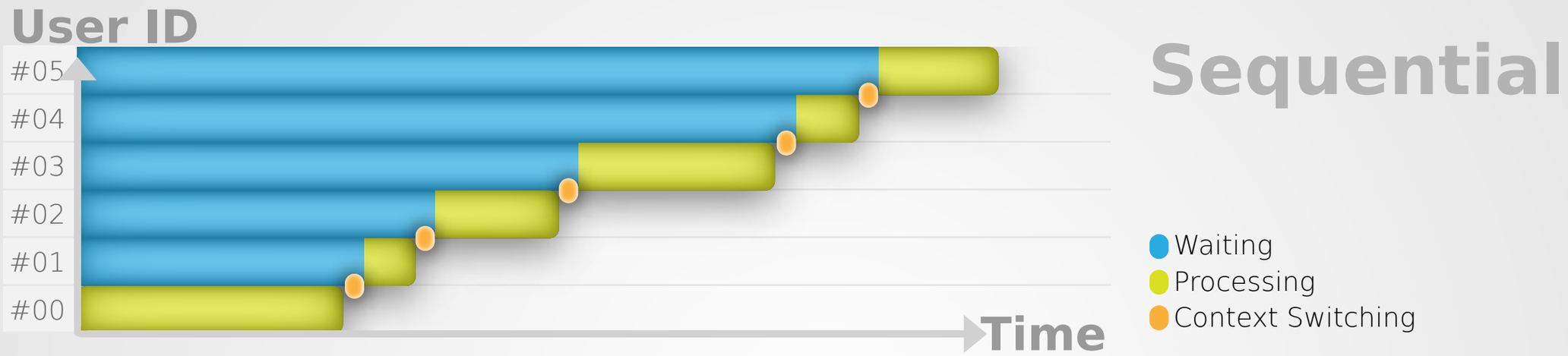
It's more fair if they are all served equally in terms of waiting time

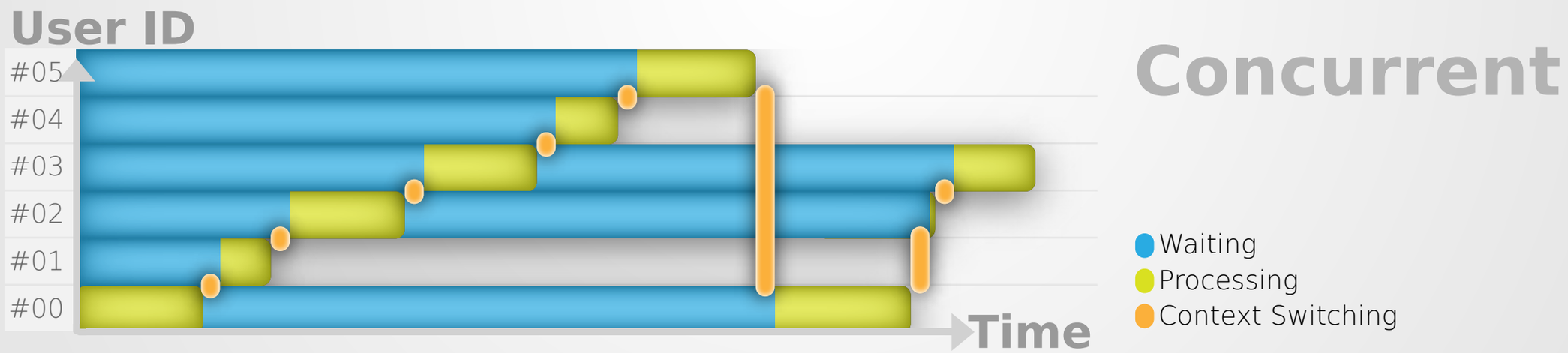

KEUKEN
ROLLEN
Echte
Boeren
Yoghurt
Petits pois extra fin

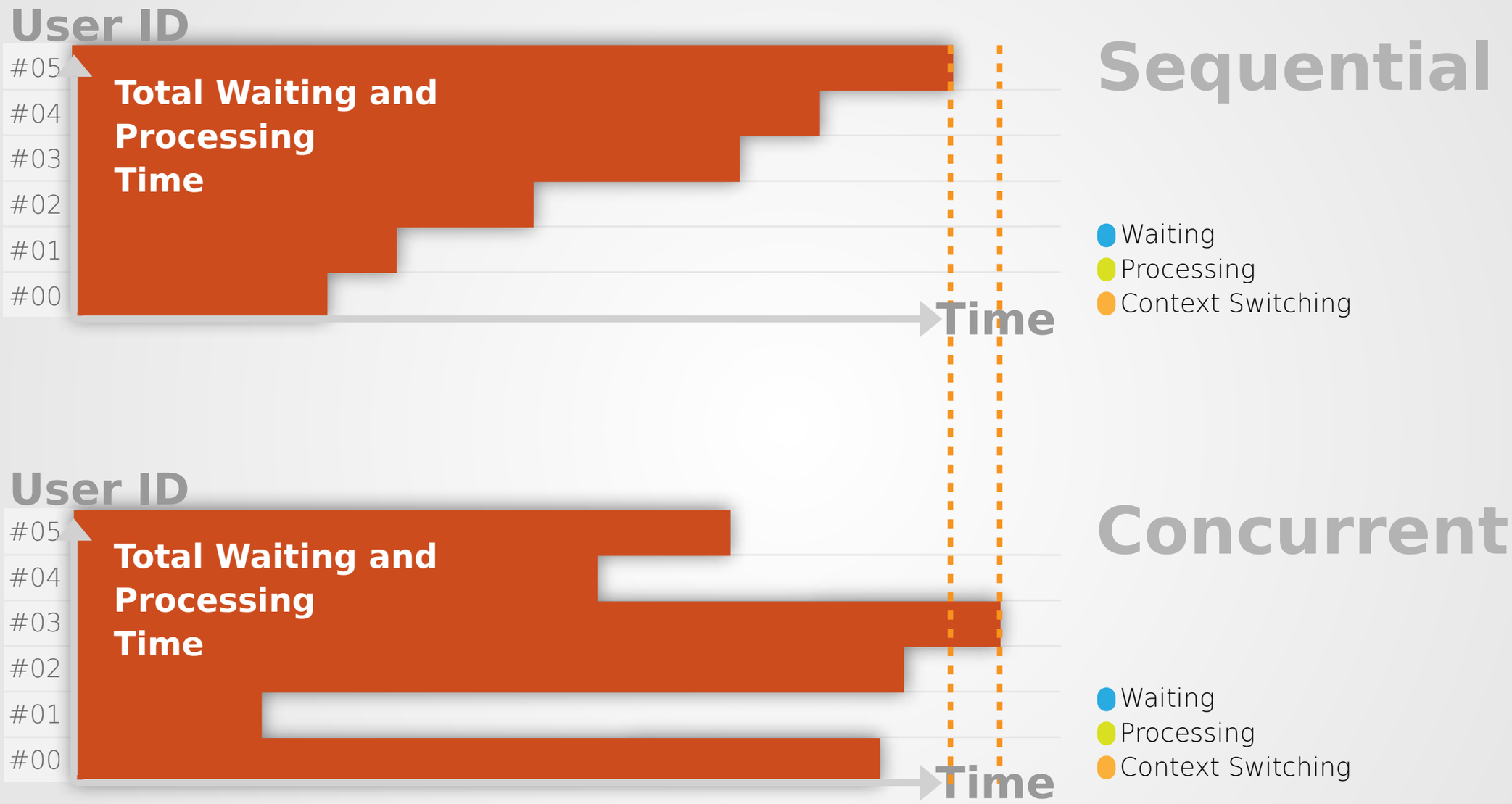
I just want a drink!

To illustrate the overall running time...




User ID
#05
#04
#03
#02
#01
#00
Time
Sequential
Waiting
Processing
Context Switching


User ID
#05
#04
#03
#02
#01
#00
Time
Concurrent
Waiting
Processing
Context Switching


User ID
#05
#04
#03
#02
#01
#00
Total Waiting and Processing Time
Time
Sequential
Waiting
Processing
Context Switching
User ID
#05
#04
#03
#02
#01
#00
Total Waiting and Processing Time
Time
Concurrent
Waiting
Processing
Context Switching

Scalable != Fast

Scalable == Less complaining

Scalable != Fast

Scalable == Less complaining

Rails is not fast

Rails might be scalable

So,

When do we want concurrency?

So,

When do we want concurrency?

When context switching cost is much cheaper than a single task

So,

When do we want concurrency?

When context switching cost is much cheaper than a single task

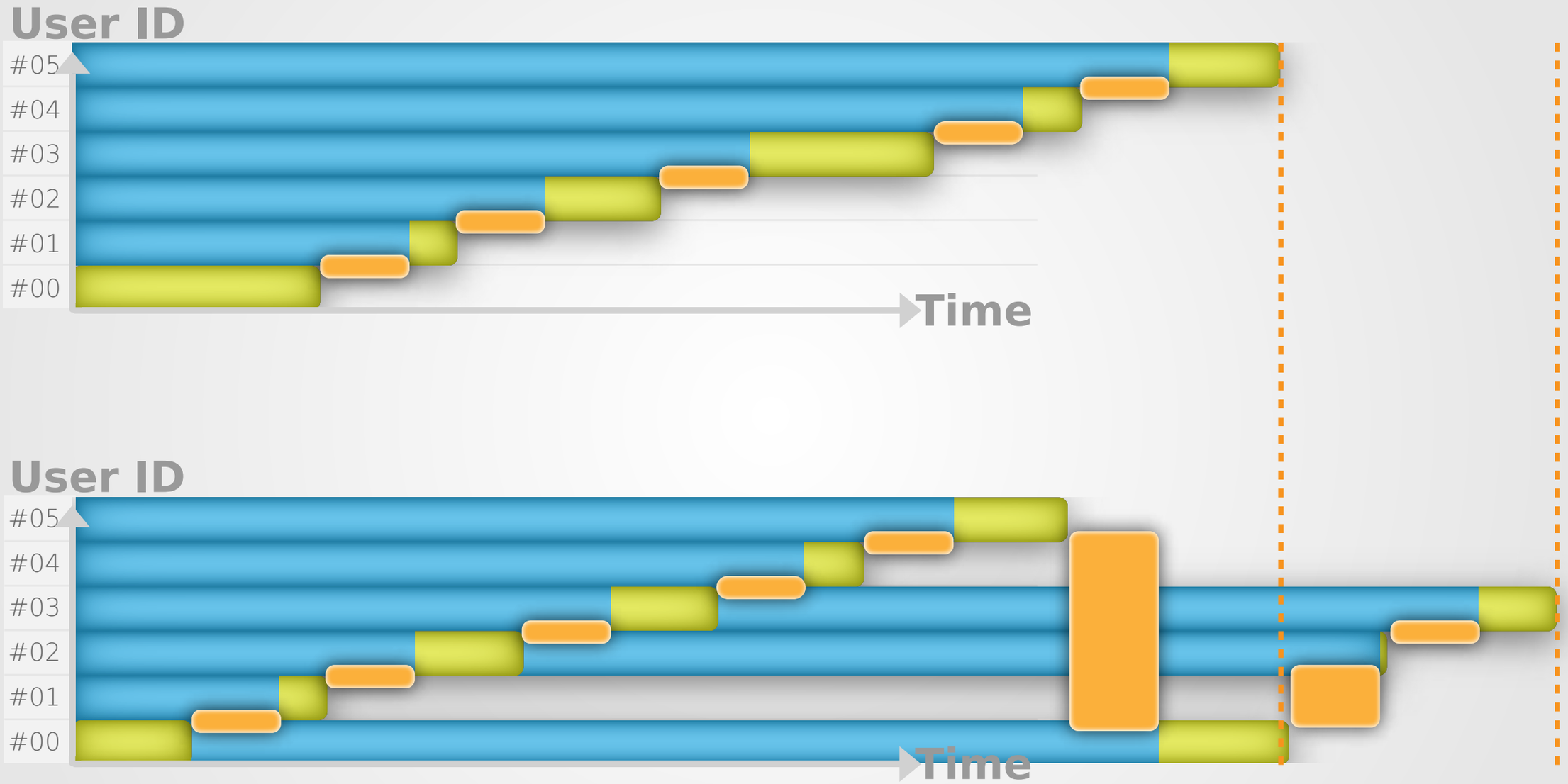
Or if you have much more cores than your clients (= no need for context switching)

If context switching cost is at about 1 second

If context switching cost is at about 1 second

It's much cheaper than 10 months, so it might be worth it

But if context switching cost is at about 5 months...



Do kernel threads context switch fast?

Do kernel threads context switch fast?

Do user threads context switch fast?

Do kernel threads context switch fast?

Do user threads context switch fast?

Do fibers context switch fast?

A ton of different concurrency models then invented

A ton of different concurrency models then invented

Each of them has different strengths to deal with different tasks

A ton of different concurrency models then invented

Each of them has different strengths to deal with different tasks

Also trade off, trading with the ease of programming and efficiency

So,

How many different types of tasks are there?

Two patterns of composite tasks

Linear data dependency

order_food -> eat

order_tea -> drink

Two patterns of composite tasks

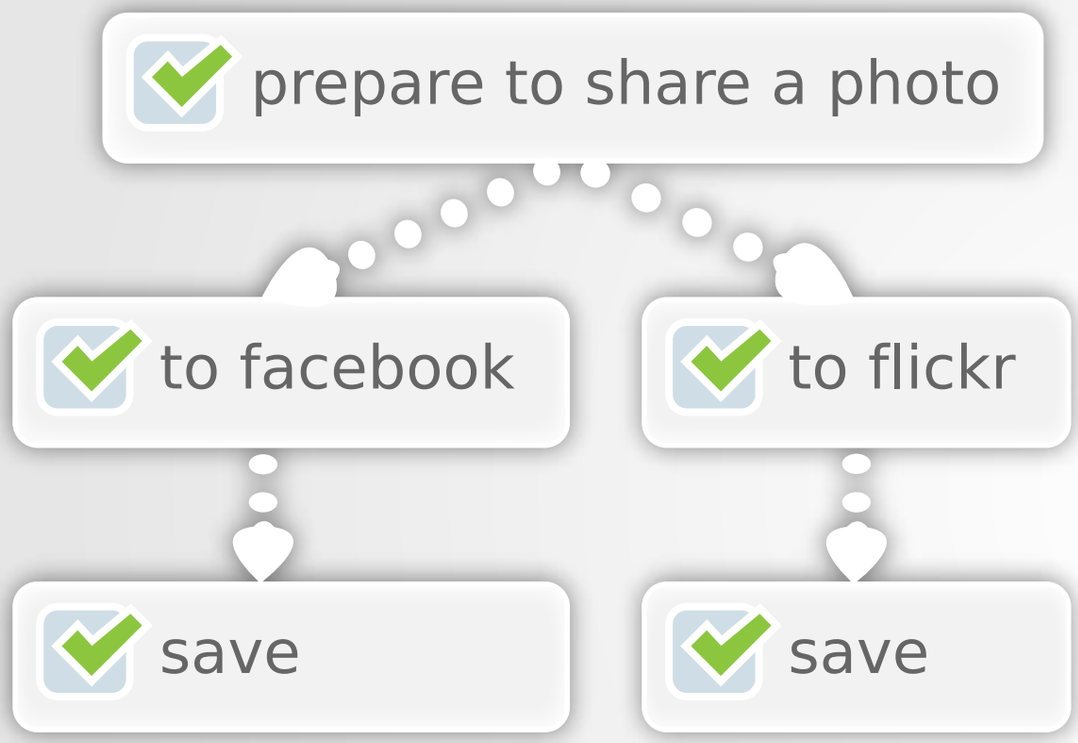
Linear data dependency


```
order_food -> eat
order_tea -> drink
```

Mixed data dependency


```
[order_food, order_spices]
-> add_spices -> eat
```

Linear data dependency



Mixed data dependency

One single task could be **CPU** or **IO** bound

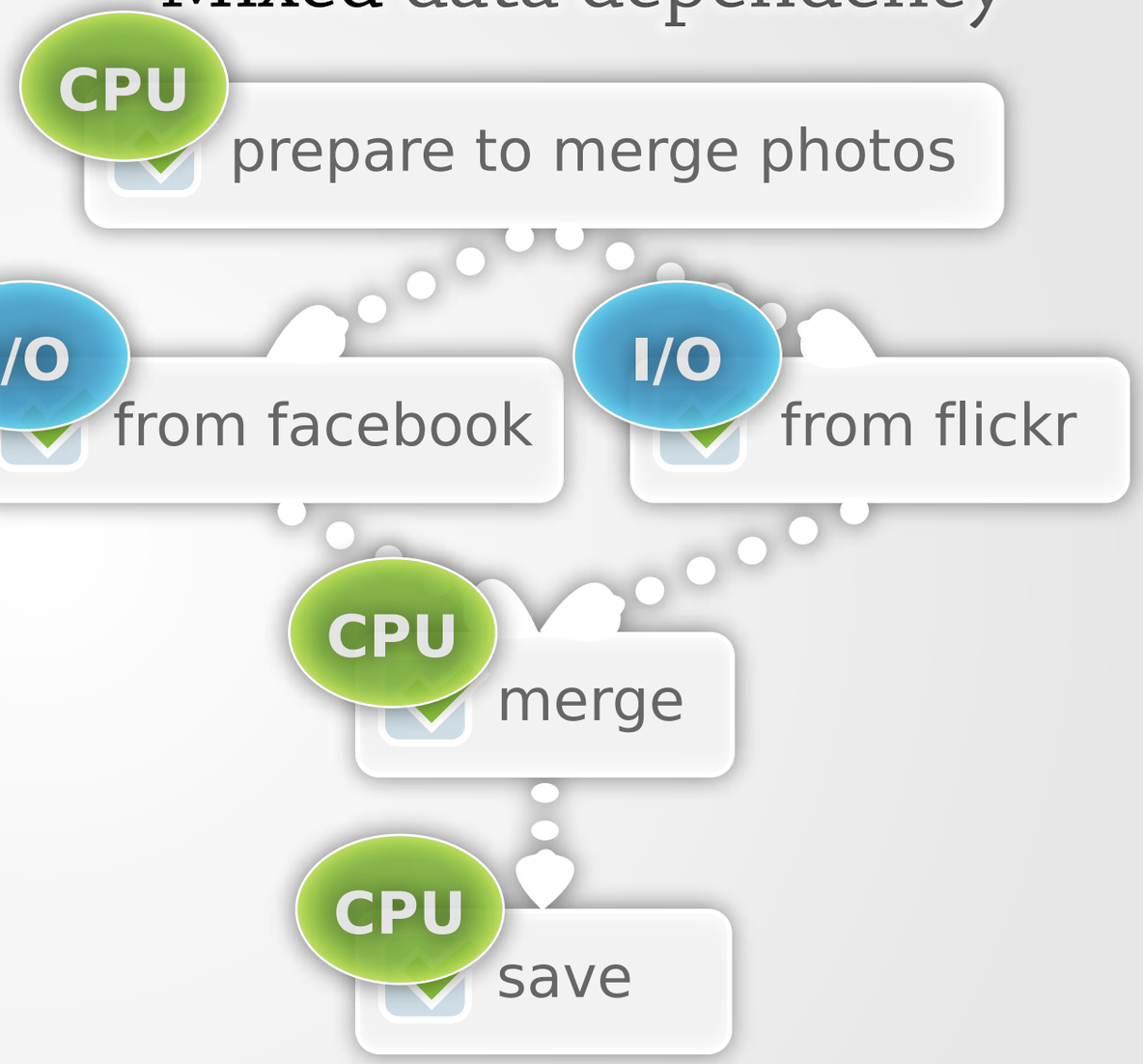
Linear data dependency

Mixed data dependency

One single task could be **CPU** or **IO** bound

Linear data dependency

Mixed data dependency



Me?

Concurrency?

What We Have?

App Servers?

Q?

Separation of Concerns

- Performance

Separation of Concerns

- Performance
- Ease of programming

Separation of Concerns

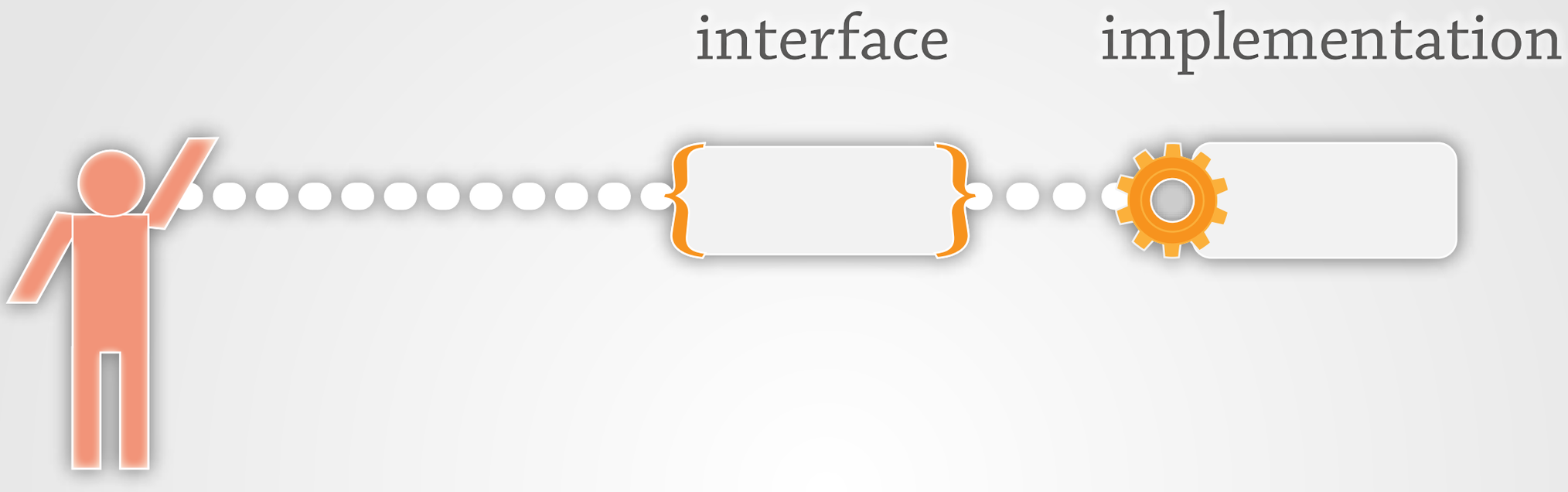
- Performance (implementation)
- Ease of programming (interface)

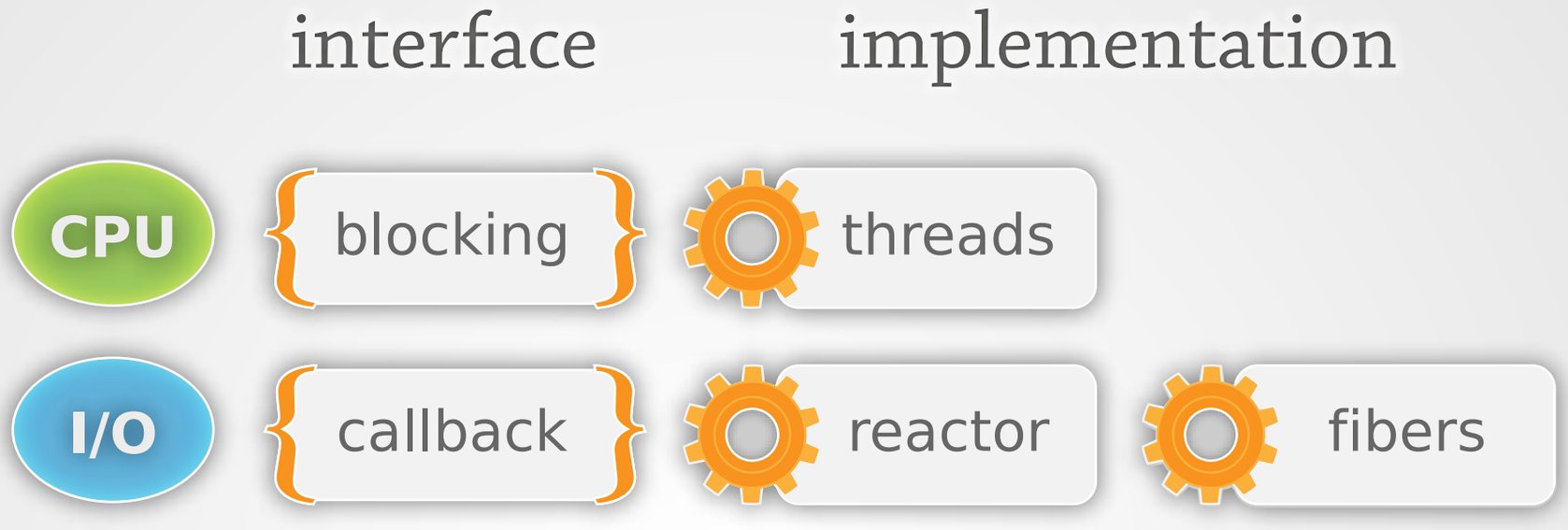
Advantages if interface is orthogonal to its implementation

- Change implementation with ease

Advantages if interface is orthogonal to its implementation

- Change implementation with ease
- Don't need to know impl detail in order to use this interface


interface
implementation


interface
implementation
CPU
blocking
threads
I/O
callback
reactor
fibers

Interface for Linear data dependency

Callback

```
order_food{ |food|
  eat(food)
}
order_tea{ |tea|
  drink(tea)
}
```

If we could control side-effect and do some static analysis...

```
eat(order_food)
drink(order_tea)
```

If we could control side-effect and do some static analysis...

```
eat(order_food)
drink(order_tea)
```

Not going to happen on Ruby though

Interface for Mixed data dependency

If callback is the only thing we have...

We don't want to do this

```
# order_food is blocking order_spices
order_food{ |food|
  order_spices{ |spices|
    eat(add_spices(spices, food))
  }
}
```

Tedious, but performs better

```
food, spices = nil
order_food{ |arrived_food|
  food = arrived_food
  start_eating(food, spices) if food && spices
}
order_spices{ |arrived_spices|
  spices = arrived_spices
  start_eating(food, spices) if food && spices
}
##
def start_eating food, spices
  superfood = add_spices(spices, food)
  eat(superfood)
end
```

Ideally, we could do this with futures

```
food = order_food
spices = order_spices
superfood = add_spices(spices, food)
eat(superfood)

# or one liner
eat(add_spices(order_spices, order_food))
```

but how?

Implementation

Implementation

Forget about
data dependency for now

Implementation: 2 main choices

If order_food is I/O bound

```
def order_food
   Thread.new{
      food = order_food_blocking
      yield(food)
   }
end
```

with a thread

If order_food is I/O bound

```
def order_food
  make_request('order_food'){ |food|
    yield(food)
  }
end
```

with a reactor

If order_food is I/O bound

```
def order_food
  buf = []
  reactor = Thread.current[:reactor]
  sock = TCPSocket.new('example.
com', 80)
  request = "GET / HTTP/1.0\r\n\r\n"
  reactor.write sock, request do
    reactor.read sock do |response|
      if response
        buf << response
      else
        yield(buf.join)
ennnnd
```

with a very simple reactor

https://github.com/godfat/ruby-server-exp/blob/master/sample/reactor.rb

If order_food is CPU bound

```
def order_food
  Thread.new{
    food = order_food_blocking
    yield(food)
  }
end
```

with a thread

If order_food is I/O bound

```
def order_food
  Thread.new{
    food = order_food_blocking
    yield(food)
  }
end
```

with a thread

You don't have to care whether it's
CPU bound or
I/O bound with a thread

And you won't want to process a CPU bound task inside a reactor blocking other clients.

Summary

- Threads for CPU bound task
- Reactor for I/O bound task

Back to mixed data dependency

If we could have some other interface than callbacks

Threads


```
food, spices = nil
t0 = Thread.new{ food = order_food }
t1 = Thread.new{ spices = order_spices }
t0.join
t1.join
superfood = add_spices(spices, food)
eat(superfood)
```

We can do it with threads easily

what if we still want callbacks, since then we can pick either threads or reactors as the implementation detail?

we could use threads or fibers to remove the need for defining another callback (i.e. start_eating)


```
food, spices = nil
order_food{ |arrived_food|
  food = arrived_food
  start_eating(food, spices) if food && spices
}
order_spices{ |arrived_spices|
  spices = arrived_spices
  start_eating(food, spices) if food && spices
}
##
def start_eating food, spices
  superfood = add_spices(spices, food)
  eat(superfood)
end
```

instead of writing this...


```
food, spices = nil
order_food{ |arrived_food|
  food = arrived_food
  start_eating(food, spices) if food && spices
}
order_spices{ |arrived_spices|
  spices = arrived_spices
  start_eating(food, spices) if food && spices
}
##
def start_eating food, spices
  superfood = add_spices(spices, food)
  eat(superfood)
end
```

Threads Turn threads callback back to synchronized like

```
condv = ConditionVariable.new
mutex = Mutex.new
food, spices = nil
order_food{ |arrived_food|
  food = arrived_food
  condv.signal if food && spices
}
order_spices{ |arrived_spices|
  spices = arrived_spices
  condv.signal if food && spices
}
##
mutex.synchronize{ condv.wait(mutex) }
  superfood = add_spices(spices, food)
  eat(superfood)
```

Fibers

Turn reactor callback to synchronized style

```
fiber = Fiber.current

food, spices = nil
order_food{ |arrived_food|
  food = arrived_food
  fiber.resume if food && spices
}
order_spices{ |arrived_spices|
  spices = arrived_spices
  fiber.resume if food && spices
}
##
Fiber.yield
  superfood = add_spices(spices, food)
  eat(superfood)
```

Threads vs Fibers

threads if your request is wrapped inside a thread (e.g. thread pool strategy)

fibers if your request is wrapped inside a fiber (e.g. reactor + fibers)

we're using eventmachine + thread pool with thread synchronization

we used to run fibers, but it didn't work well with other libraries

- e.g. activerecord's connection pool didn't respect fibers, only threads

also, using fibers we're running a risk where we might block the event loop somehow we don't know

so using threads is easier if you consider thread-safety is easier than fiber-safety + potential risk of blocking the reactor

and we can even go one step further…

…into the futures!

this is also a demonstration that some interfaces are only available to some implementations

```
food = order_food
spices = order_spices
superfood = add_spices(spices, food)
eat(superfood)

# or one liner
eat(add_spices(order_spices, order_food))
```

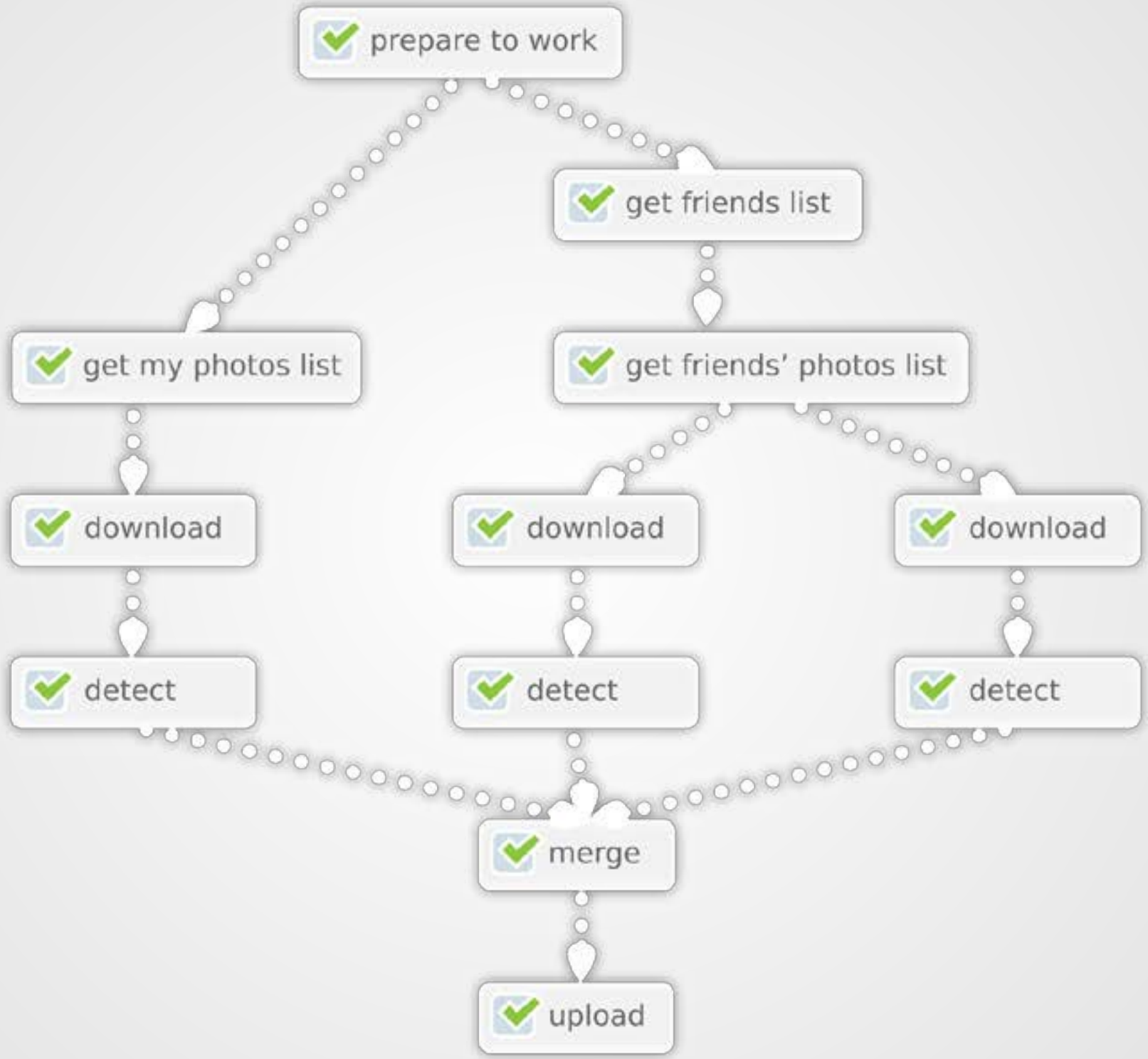
Who got futures?

- rest-core for HTTP futures
- celluloid for general futures
- also check celluloid-io for replacing eventmachine

a more complex (real world) example

- update friend list from facebook
- get photo list from facebook
- download 3 photos from the list
- detect the dimension of the 3 photos
- merge above photos
- upload to facebook

this example shows a mix model of linear and mixed data dependency


prepare to work
get friends list
get my photos list
get friends' photos list
download
download
download
detect
detect
detect
merge
upload

Me?

Concurrency?

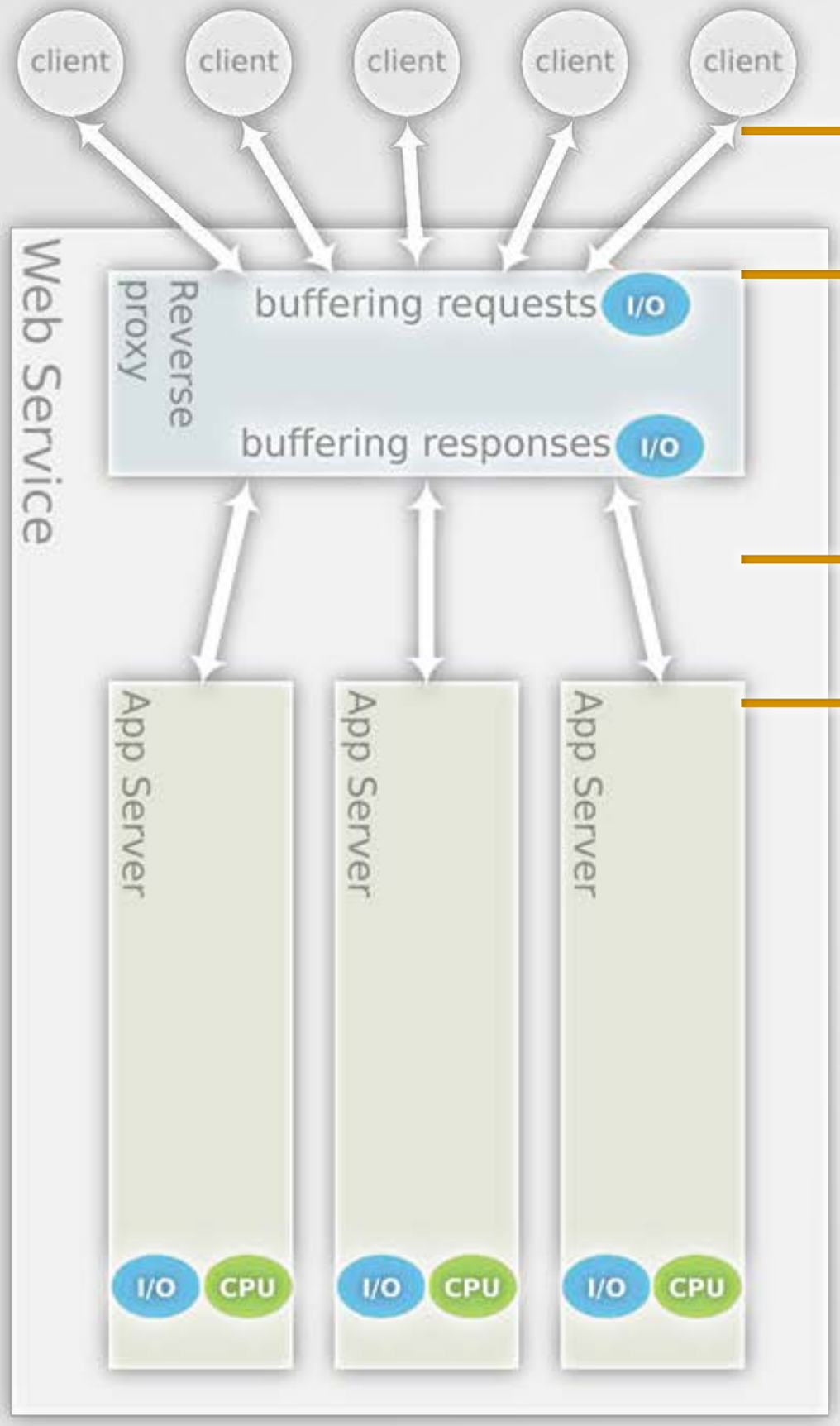
What We Have?

App Servers?

Q?

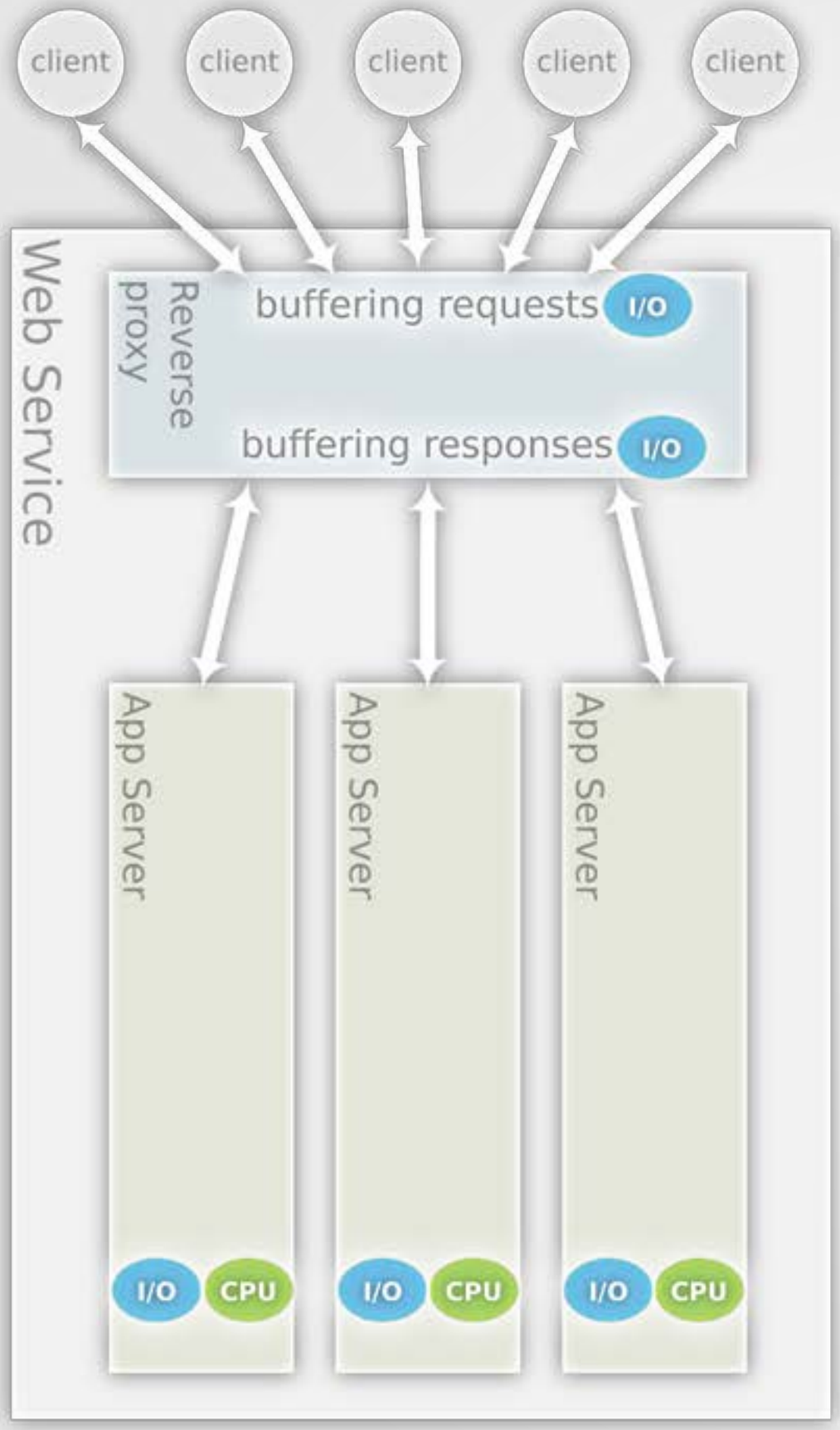
Again:

we don't talk about chunked encoding and web sockets or so for now; simply plain old HTTP 1.0



Network concurrency

Application concurrency

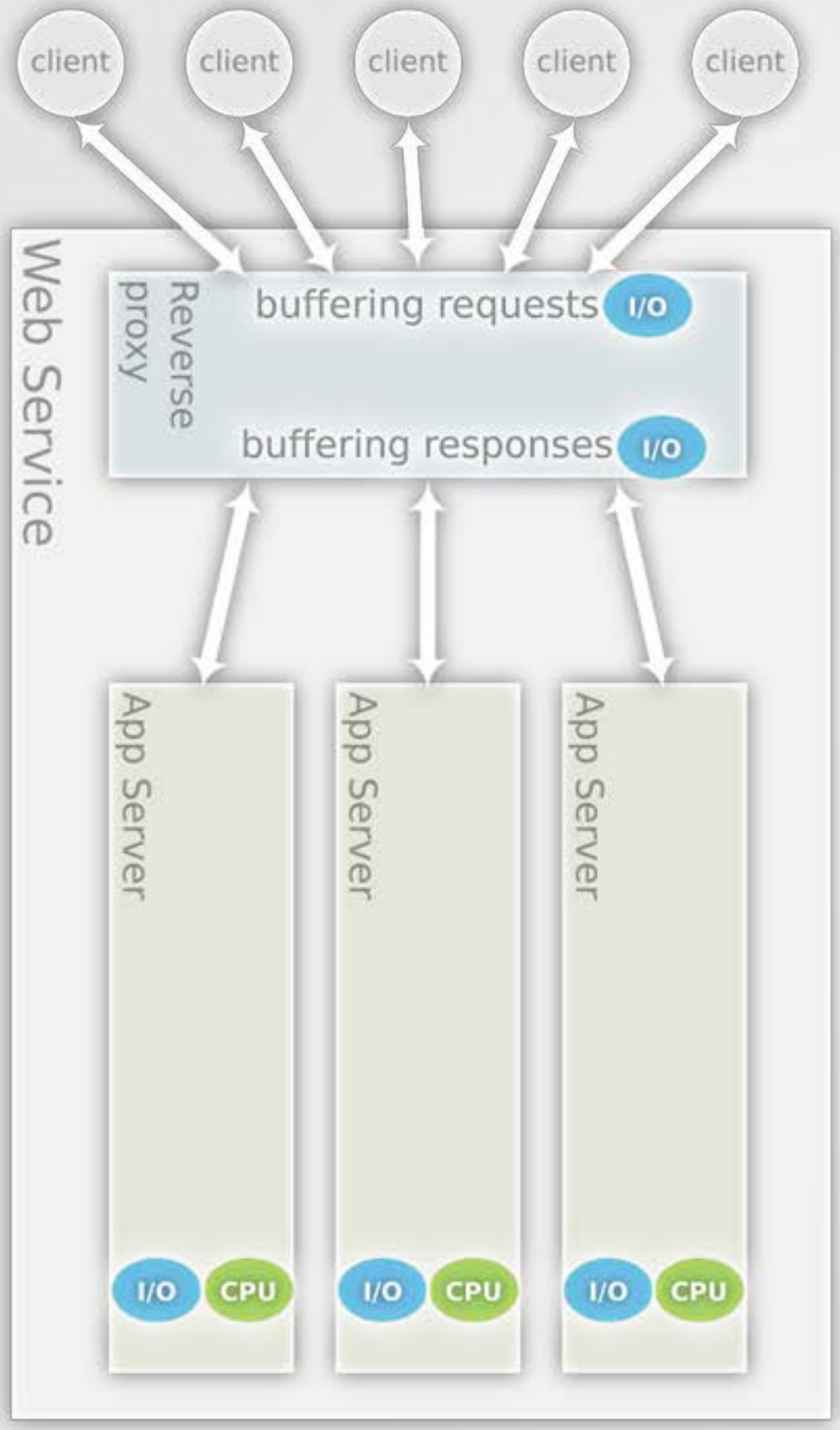


sockets I/O bound tasks would be ideal for an event loop to process them efficiently

nginx, eventmachine, libev, nodejs, etc.

however, CPU bound tasks should be handled in real hardware core

e.g. kernel process/thread



we can abstract the http server (reverse proxy) easily, since it only needs to do one thing and do it well (unix philosophy)

that is, using an event loop to buffer the requests

however, different application does different things, one strategy might work well for one application but not the other

we could have an universal concurrency model which could do averagely good, but not perfect for say, *your* application

that is why Rainbows provides all possible concurrency models for you to choose from

what if we want to make external requests to outside world?

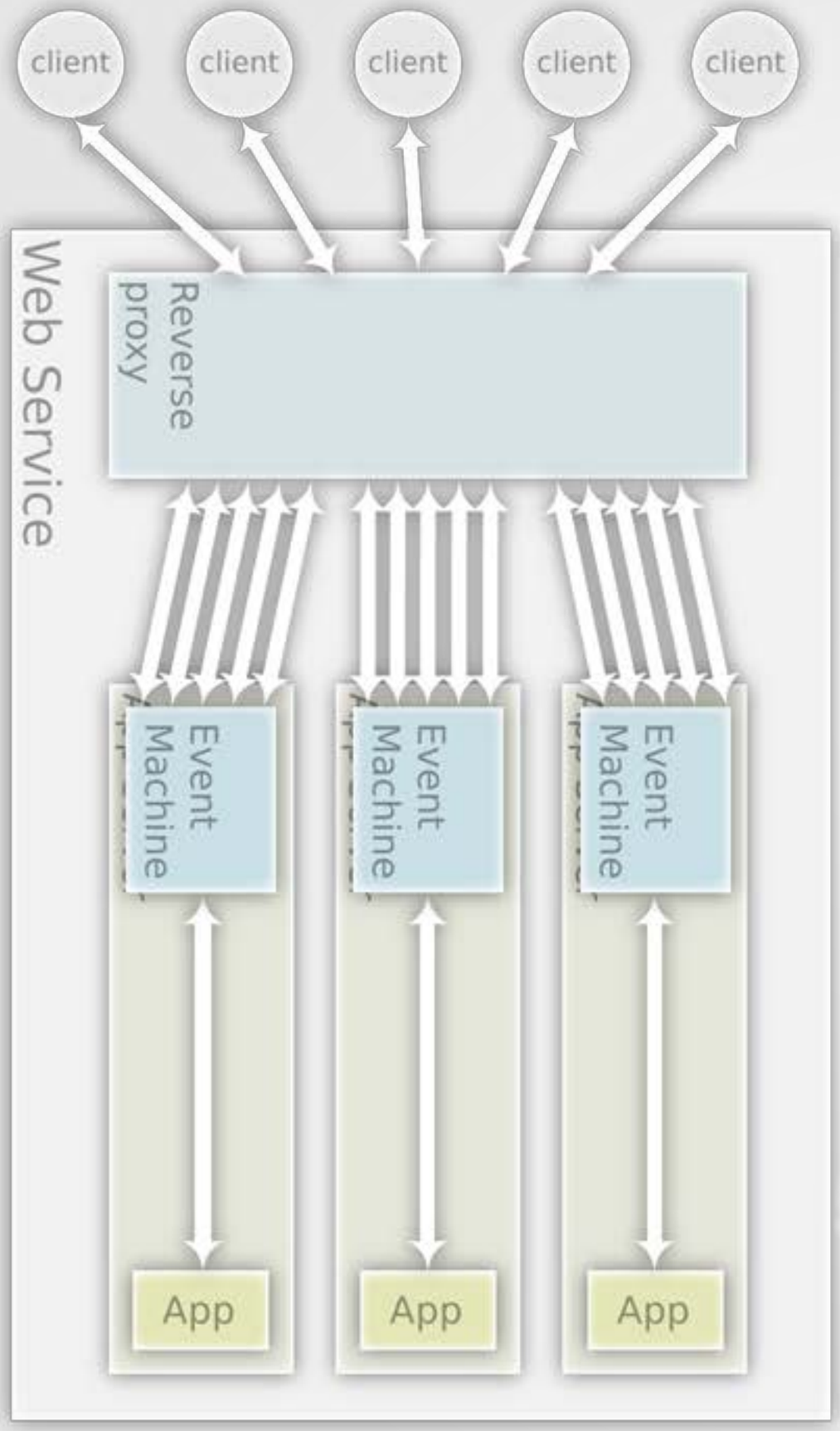
e.g. facebook

it's I/O bound, and could be the most significant bottleneck, much slower than your favorite database

before we jump into the detail...

let's see some concurrent popular ruby application servers

Thin, Puma, Unicorn family



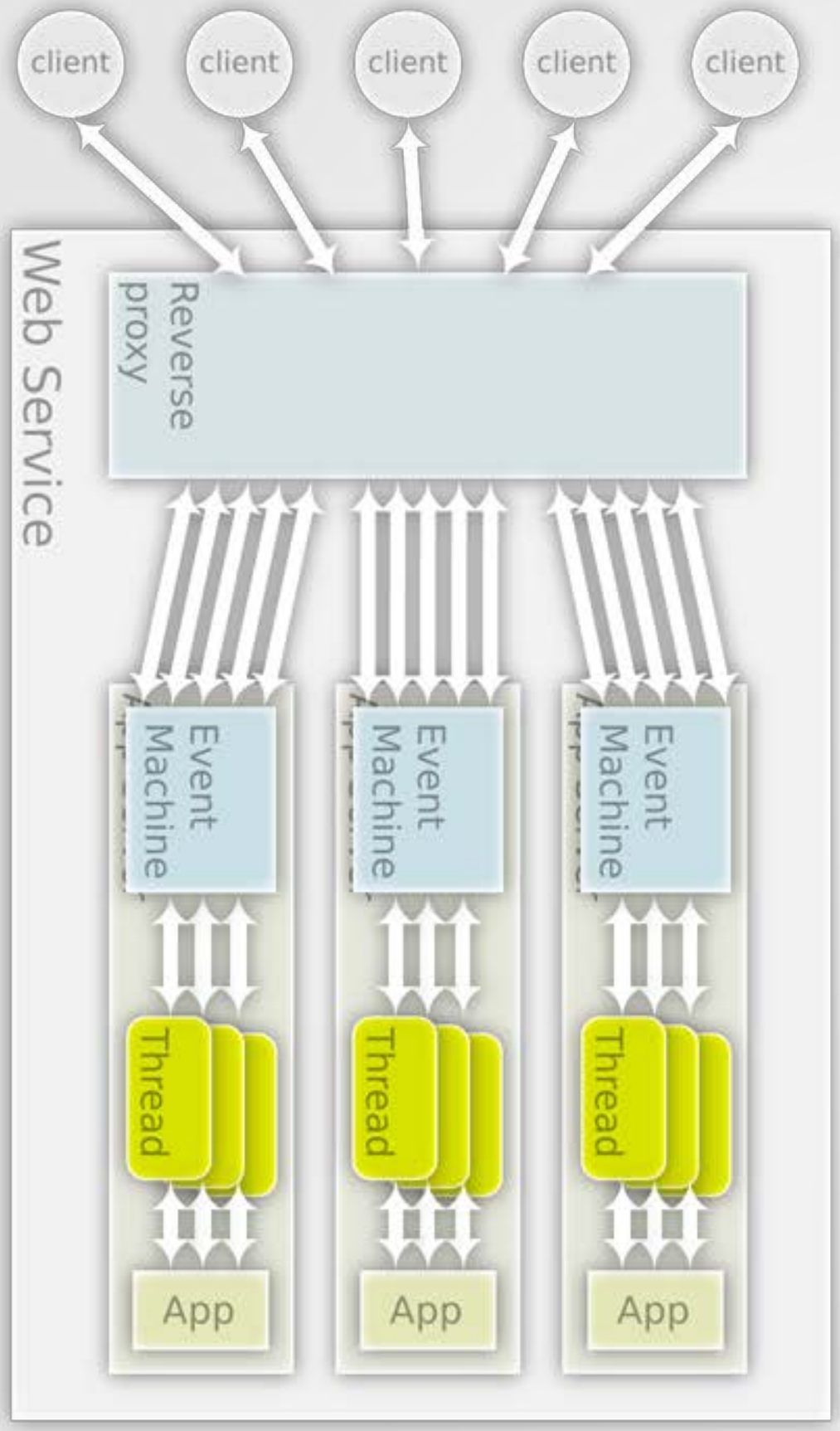
Default Thin

eventmachine (event loop) for buffering requests

no application concurrency

you can run thin cluster for application concurrency

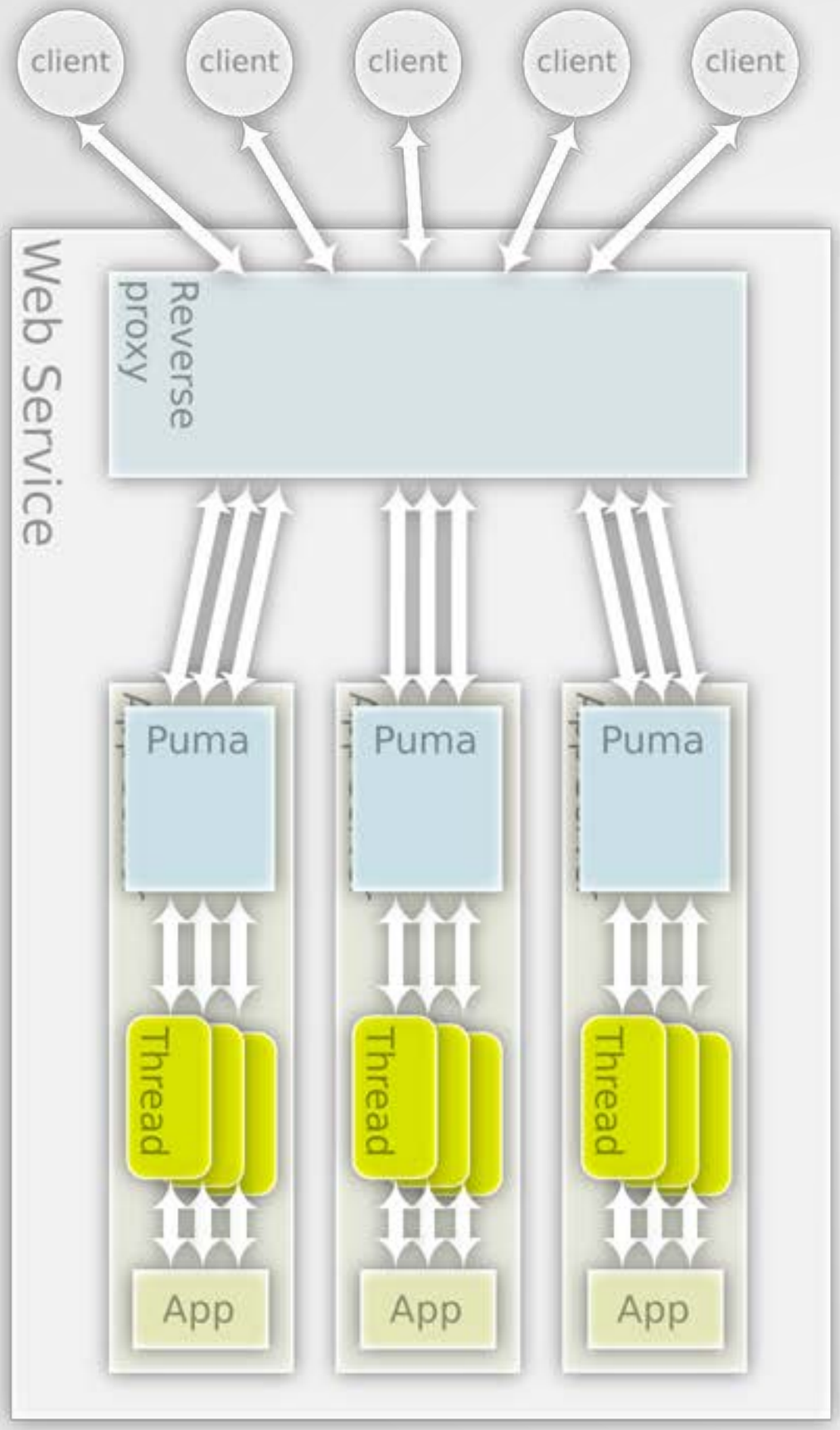
Threaded Thin



eventmachine (event loop) for buffering requests

thread pool to serve requests

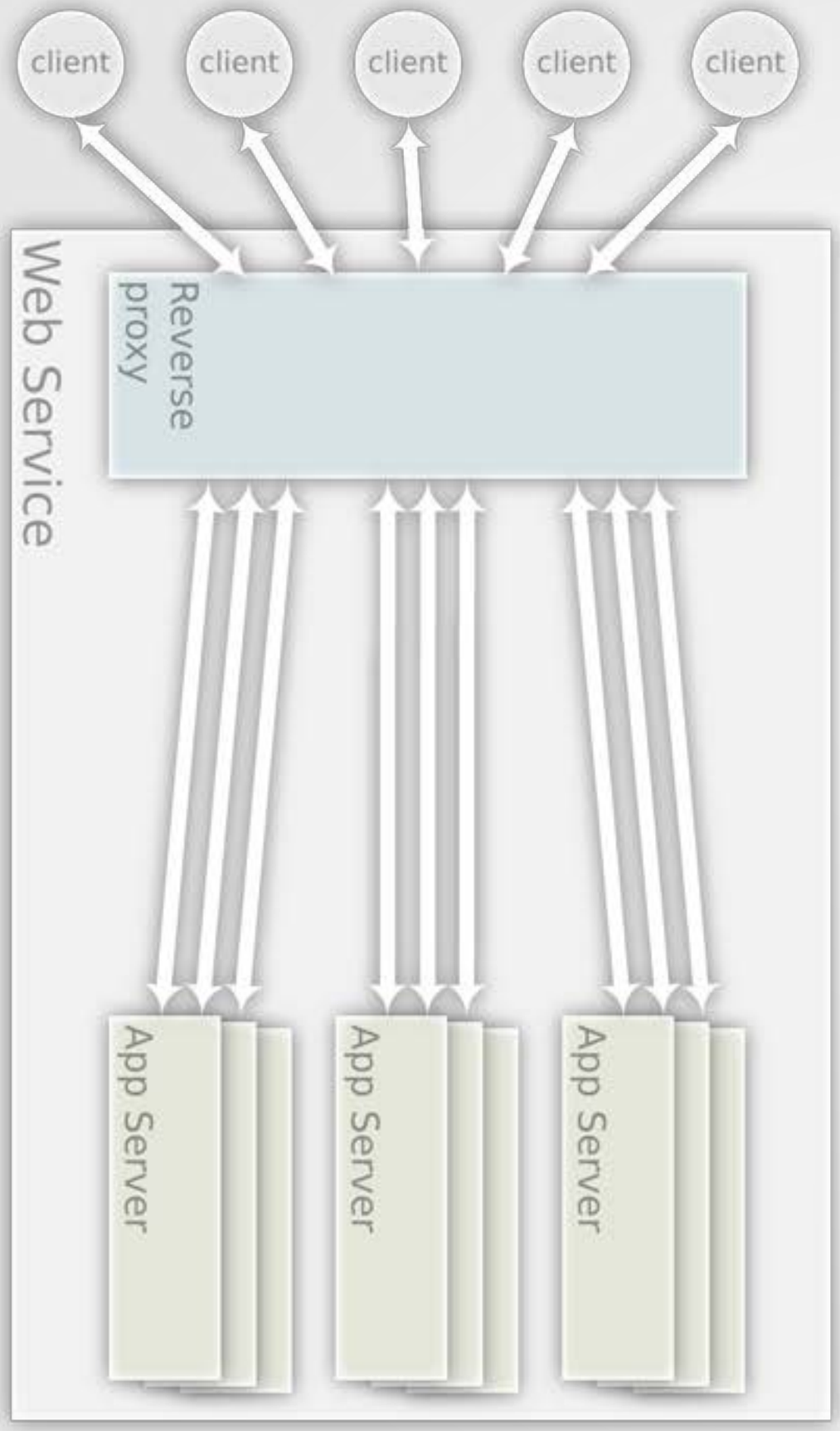
you can of course run cluster for this



Puma

zbatery + ThreadPool

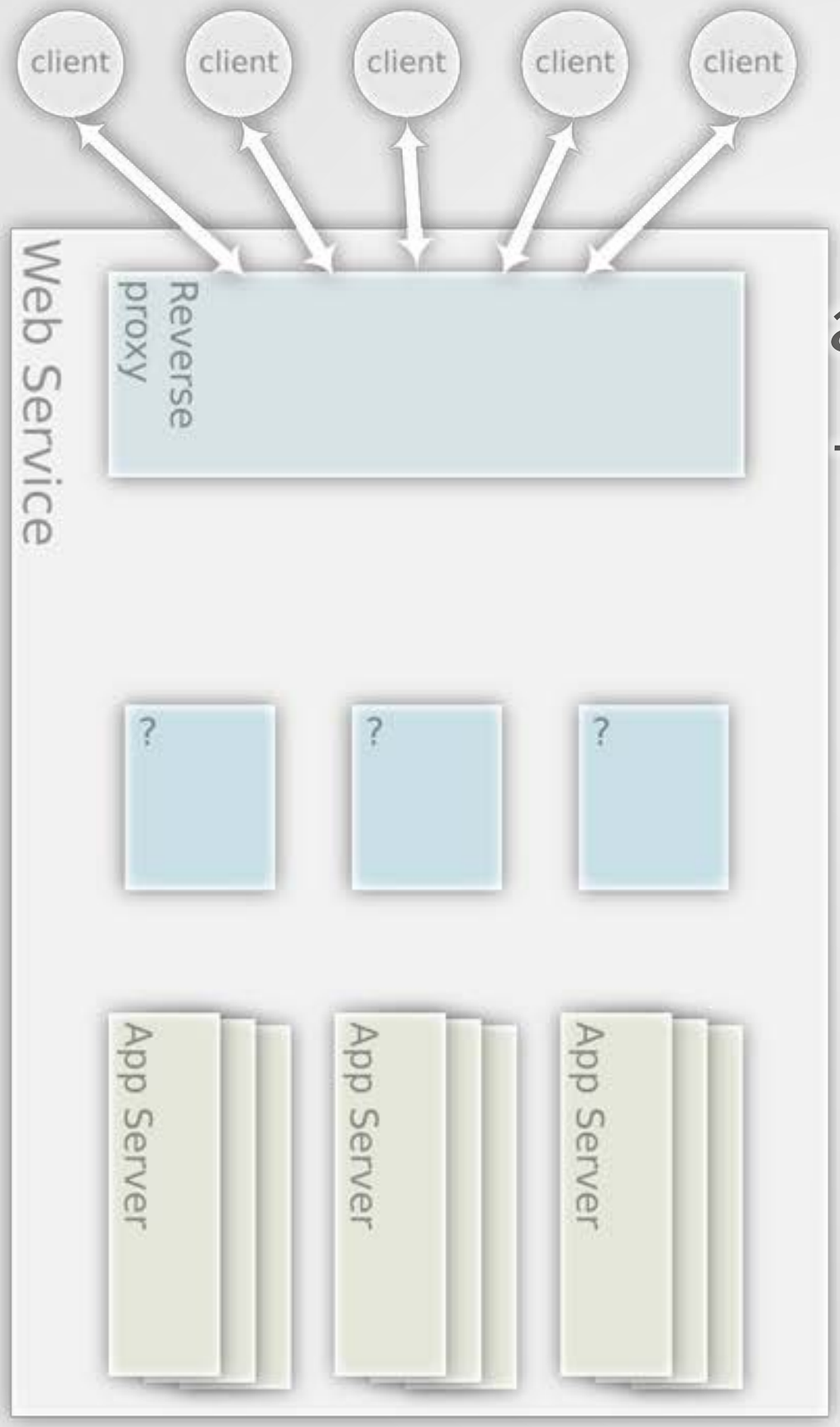
= puma



Unicorn

no network concurrency

worker process
application concurrency



Rainbows

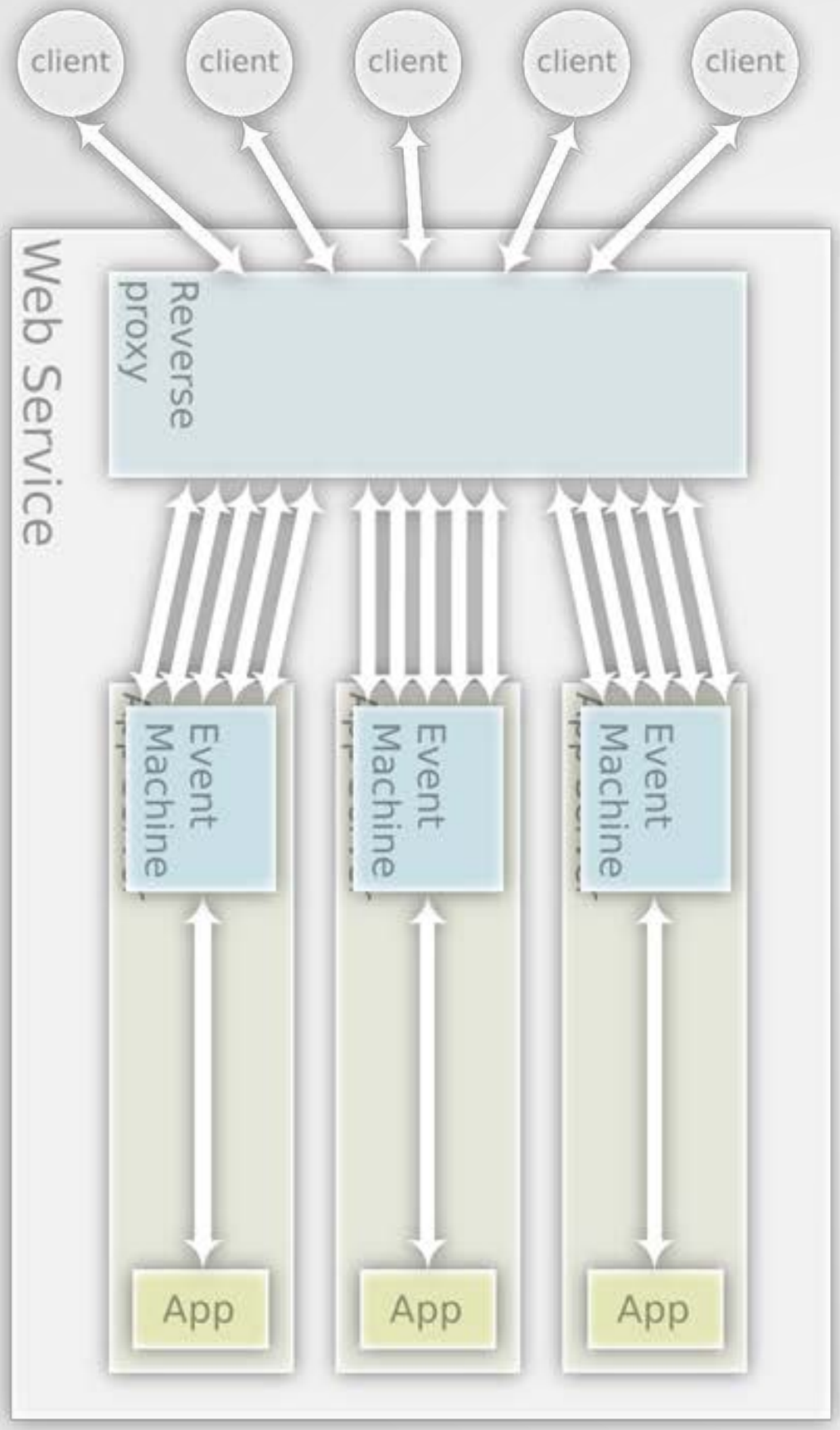
another concurrency model

+ unicorn

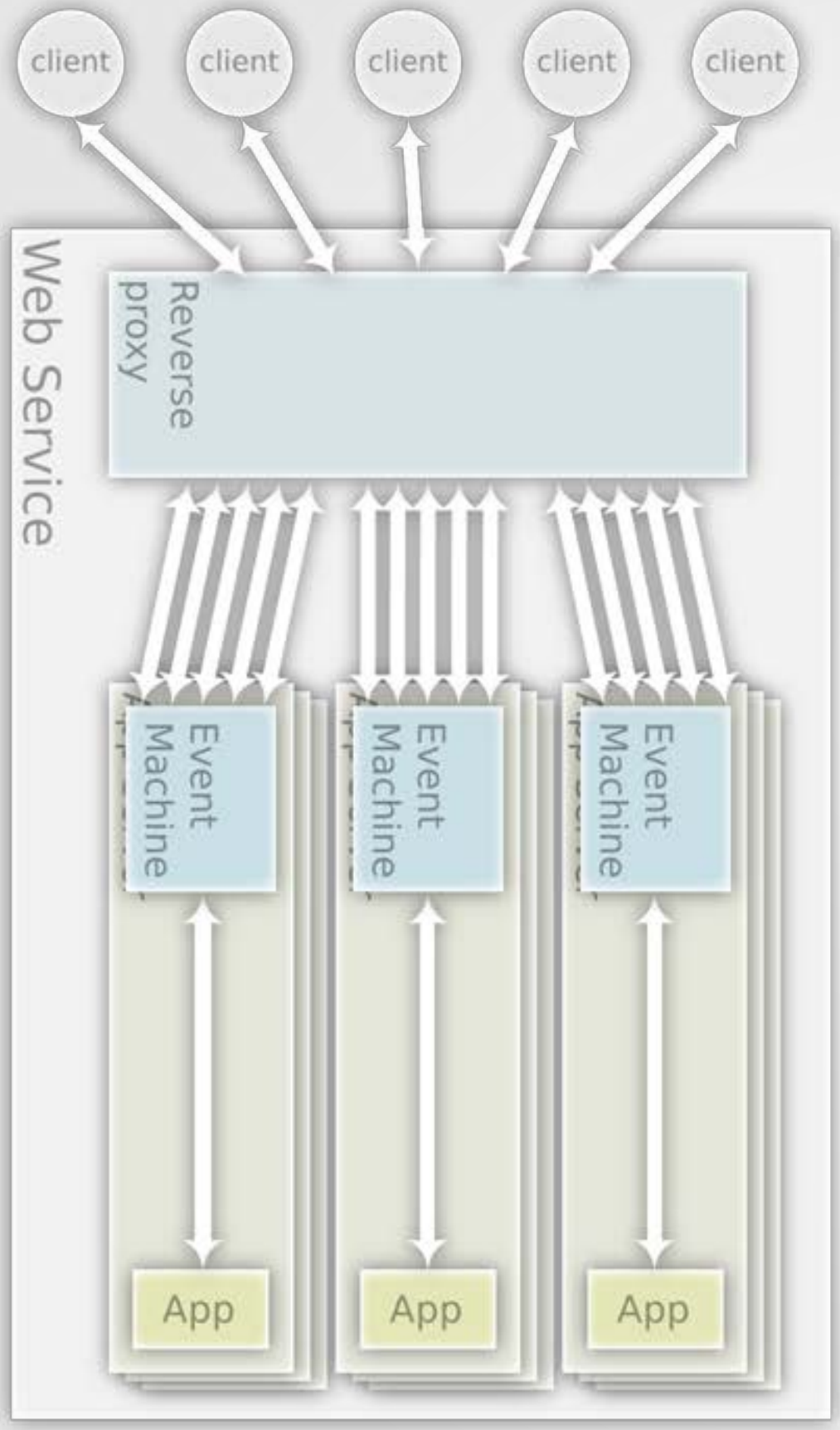
Zbatery

rainbows with single unicorn (no fork)

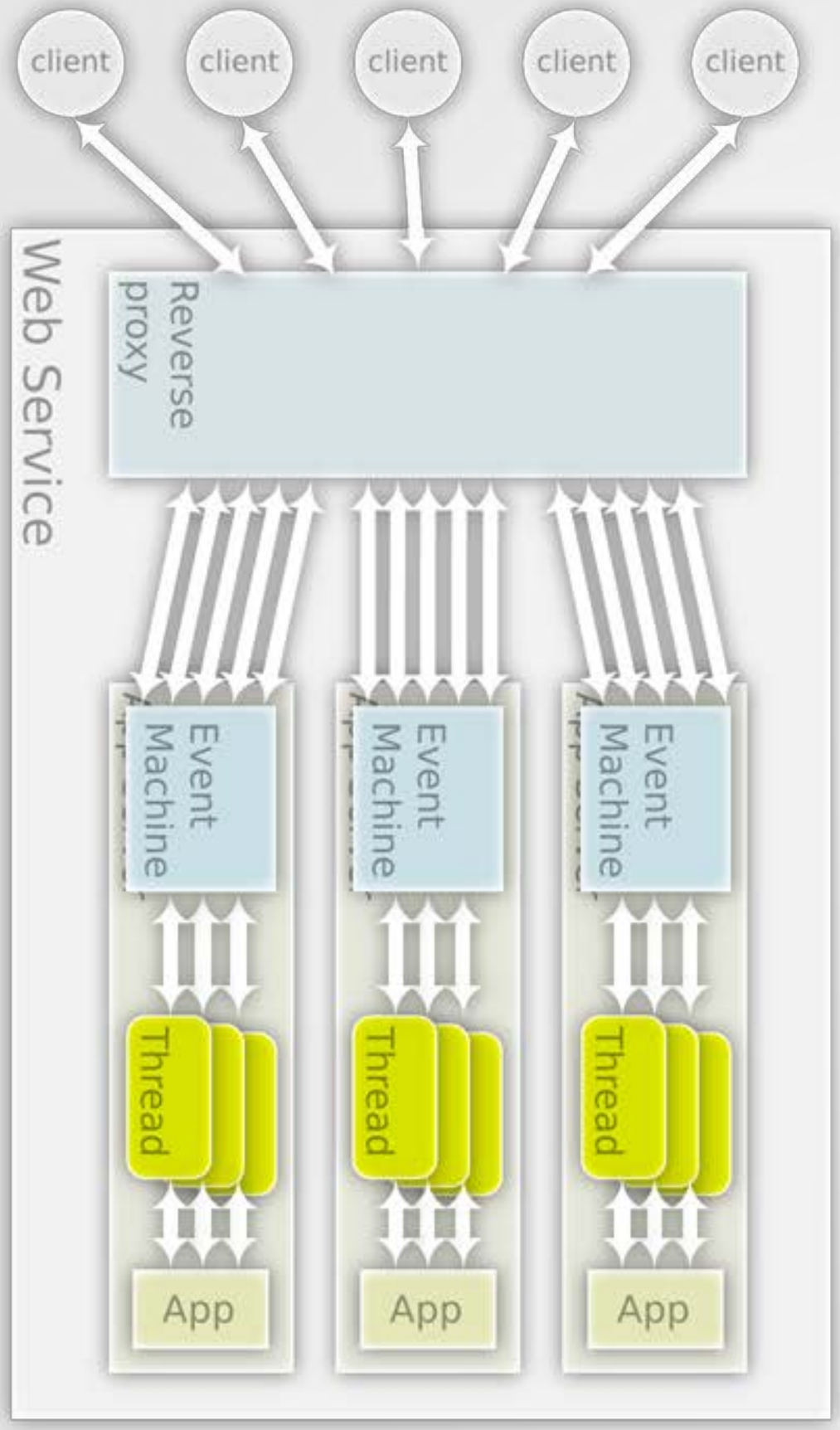
saving memories



Zbatery + EventMachine = default Thin



Rainbows + EventMachine = cluster default Thin



Zbatery + EventMachine + TryDefer (thread pool) = threaded Thin

Each model has its strength to deal with different task

Remember? threads for cpu operations, reactor for I/O operations

What if we want to resize images, encode videos?

it's of course CPU bound, and should be handled in a real core/CPU

What if we want to do both? what if we first request facebook, and then encode video, or vice versa?

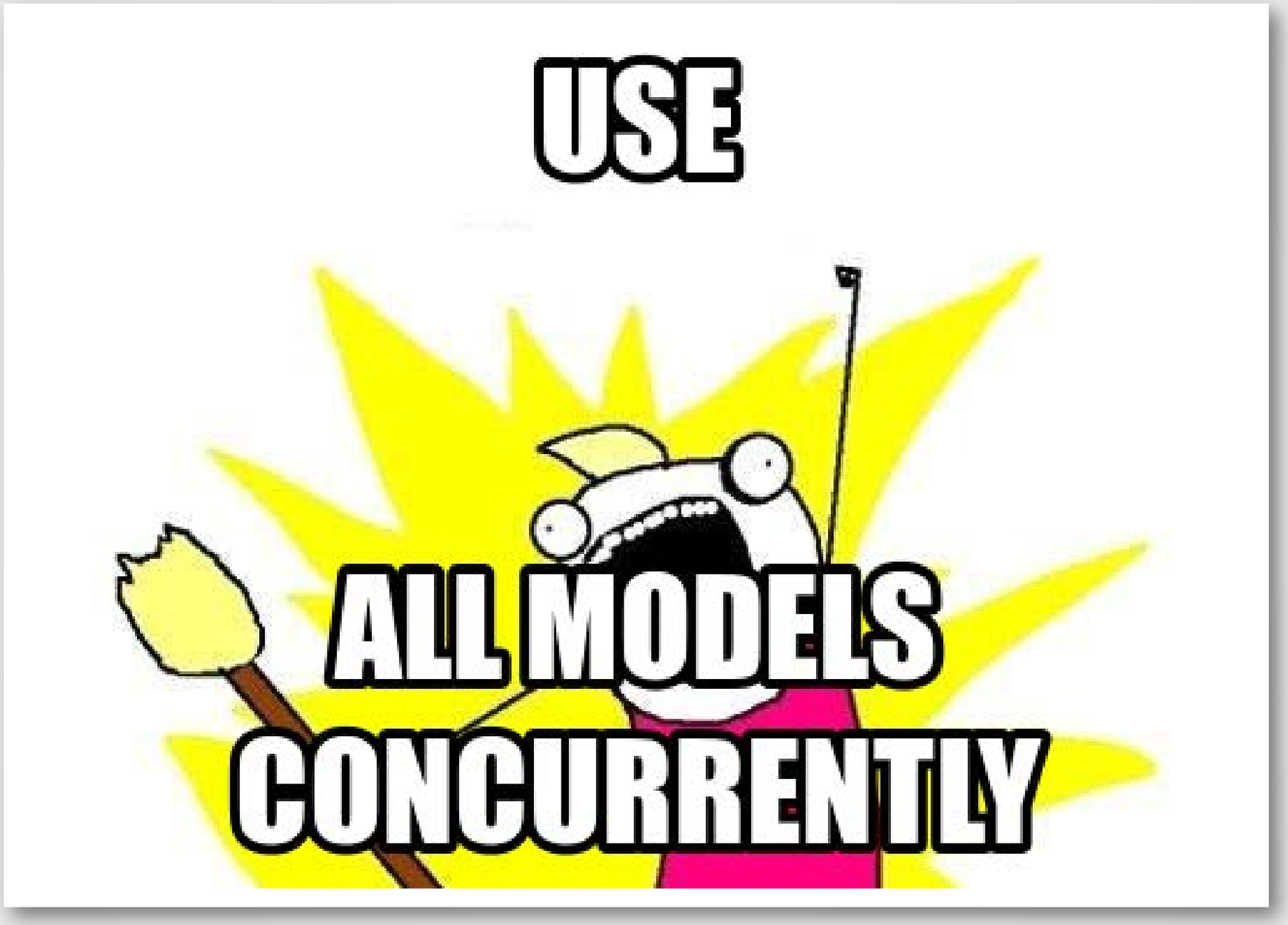
or we need to request facebook and encode videos and request facebook again?

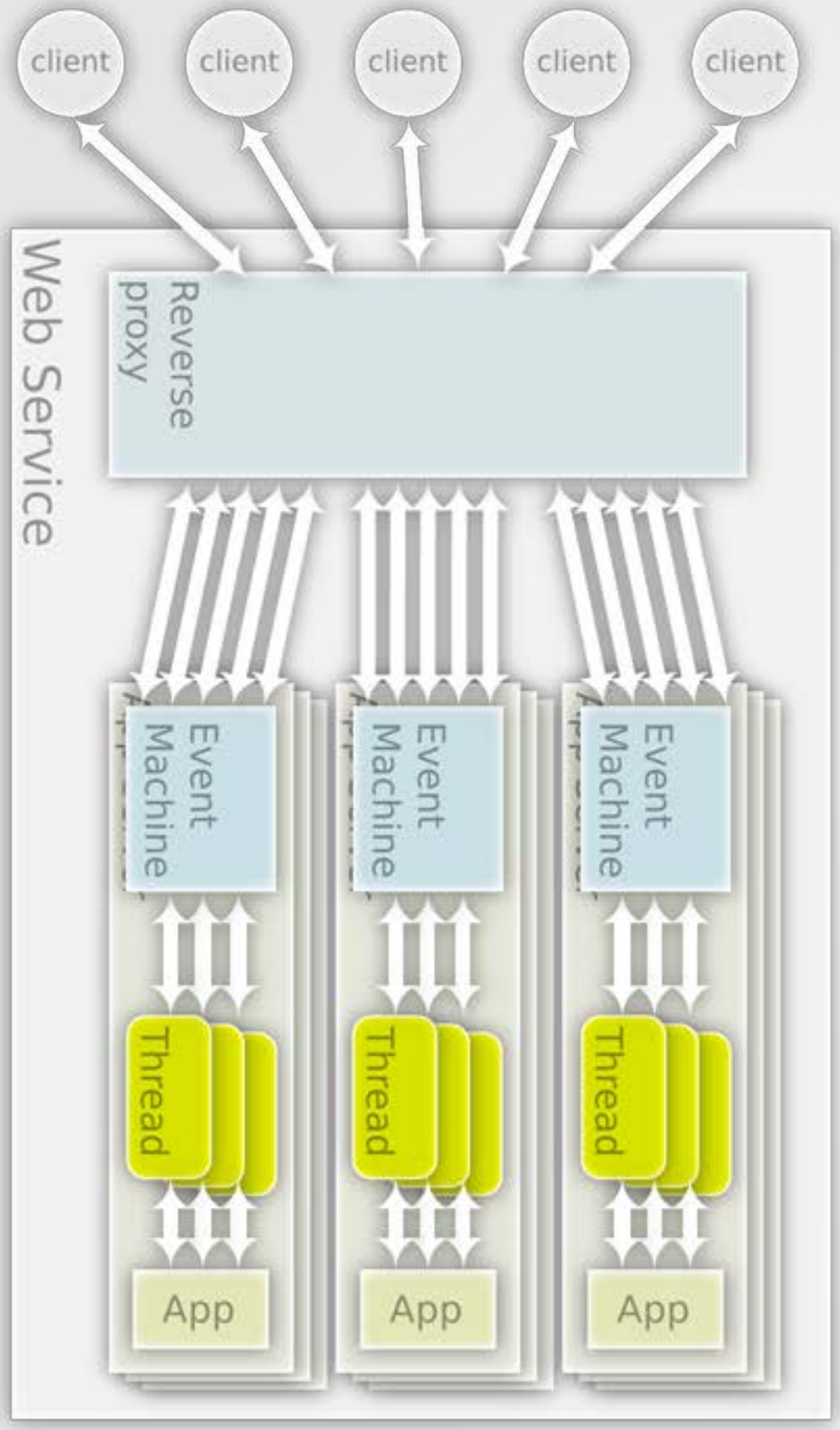
The reactor could be used for http concurrency and also making external requests

Ultimate solution

for what i can think of right now

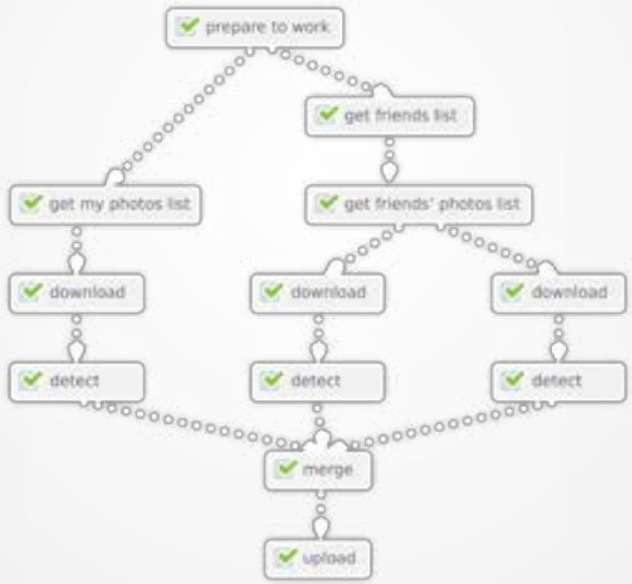
USE EVERYTHING





Rainbows + EventMachine + Thread pool + Futures!

And how do we do that in a web application? We'll need to do the above example in a concurrent way. i.e.



To compare all the application servers above

	Network	Interface	Application
Thin (default)	EventMachine	Rack	N/A
Thin (threaded)	EventMachine	Rack	Thread pool
Puma	Thread pool	Rack	Thread pool
Unicorn	Worker processes	Rack	Worker processes
Rainbows	Worker processes +	depends on Rack	configurations
Zbatery	Depends on configurations	Rack	
Passenger	I/O threads libev	Rack	Process pool Process/thread pool
Goliath	EventMachine	Wrapped Rack	N/A

Me?

Concurrency?

What We Have?

Q?

App Servers?