Concurrent Ruby Application Servers

Agenda

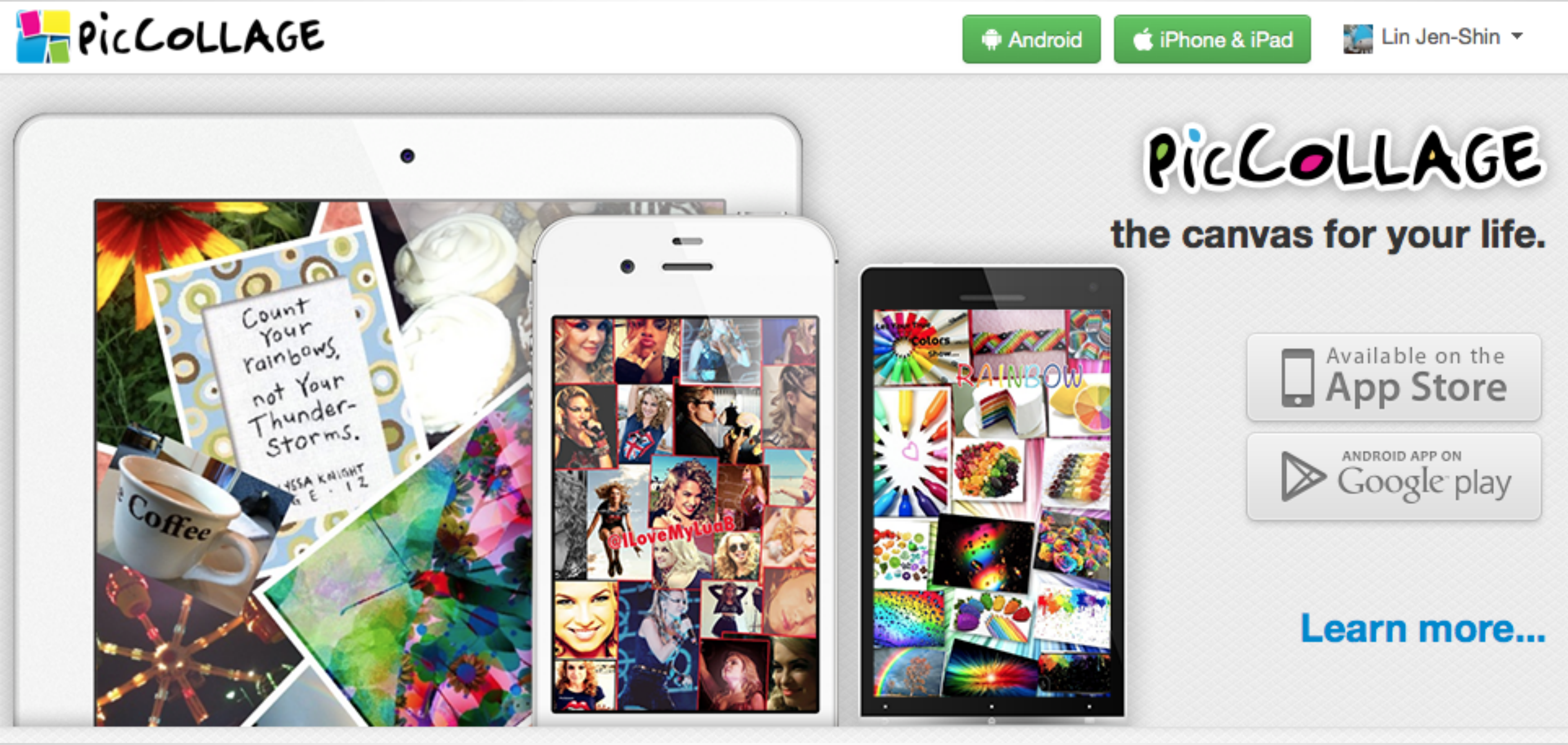
- Who? (2)
- Concurrency? (10)
- What we have? (15)
- App servers? (15)
- Q? (3)

Who?

* Programmer at Cardinal Blue
* Use Ruby from 2006
* Interested in programming languages and functional programming (e.g. Haskell)

* Programmer at Cardinal Blue
* Use Ruby from 2006
* Interested in programming languages and functional programming (e.g. Haskell)
* Also concurrency recently

PicCollage


PicCollage
Android
iPhone & iPad
Lin Jen-Shin
PicCollage
the canvas for your life.
Available on the
App Store
ANDROID APP ON
Google play
Learn more...

PicCollage

in 7 days

* ~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

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

Concurrency?

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 month

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

If your program context switches, then actually it runs slower

But it's more fair for users

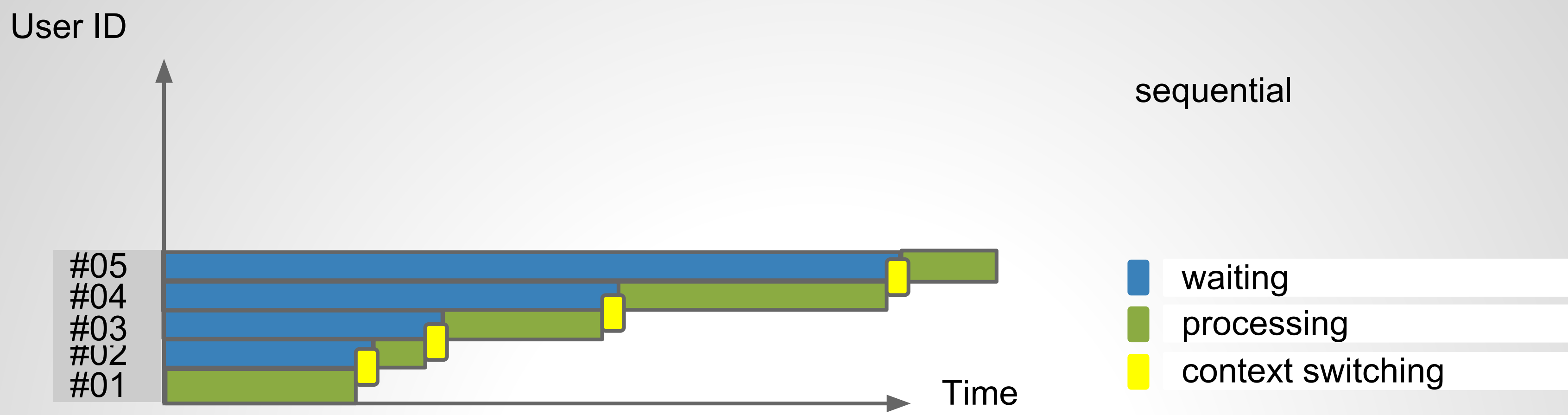

adidas

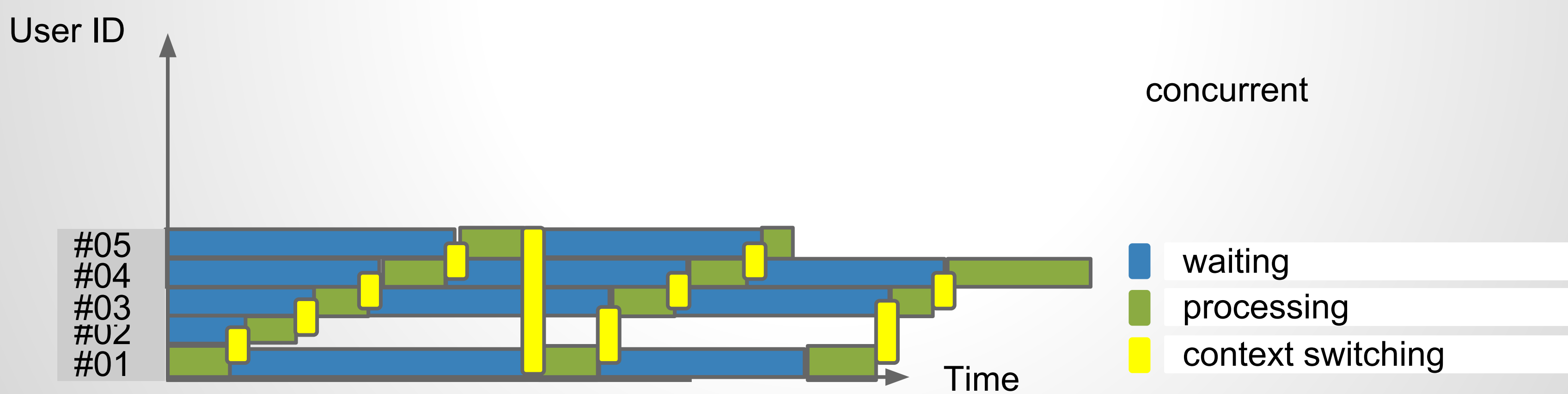
It's more fair if they are all served equally in terms of waiting time

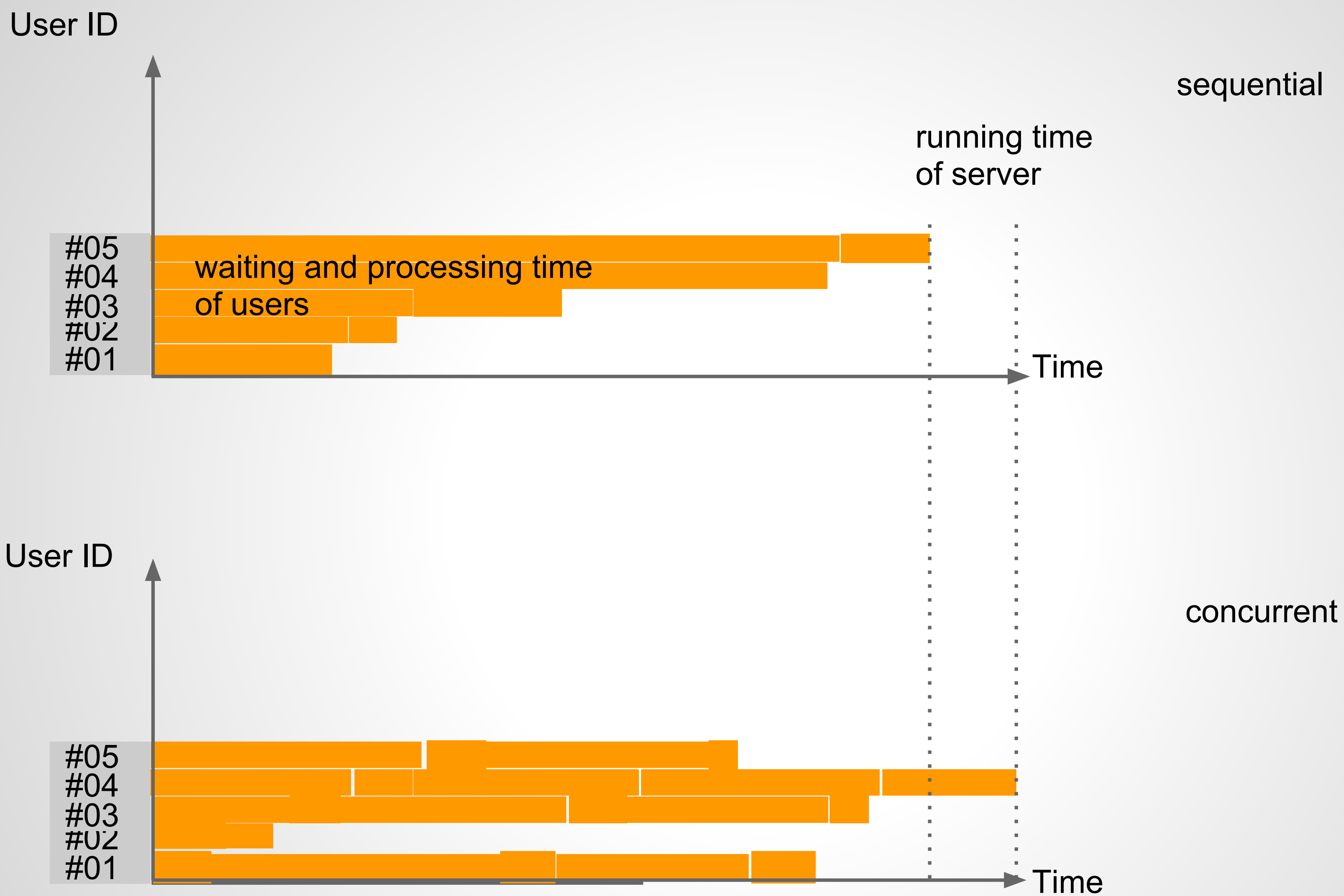

PROPIA
KEUKEN ROLLEN
WIT 2-LAAGS
Echte
Boer'n
COOLBEST
Petits pois extra fin
Froggy

I just want a drink!

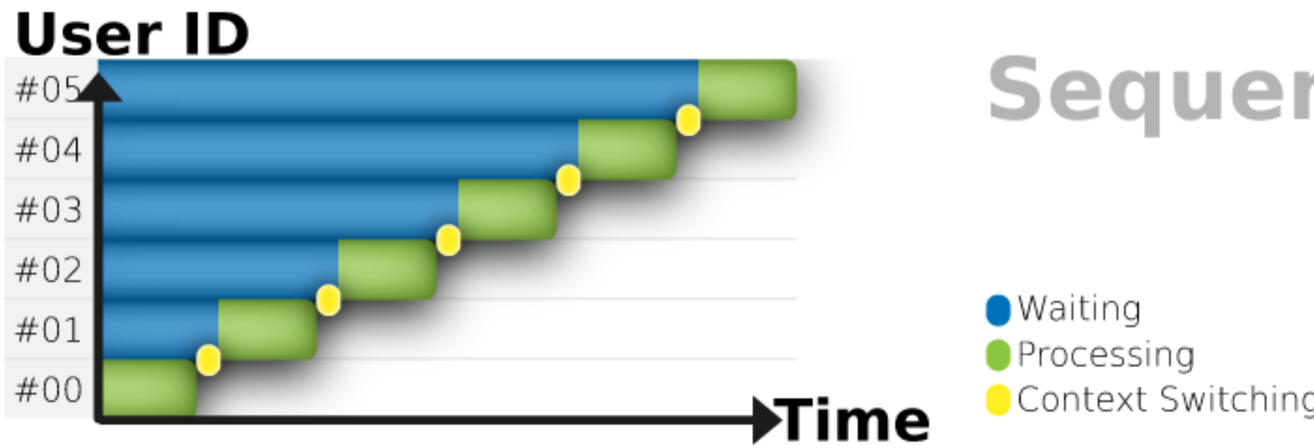
to illustrate the overall running time:

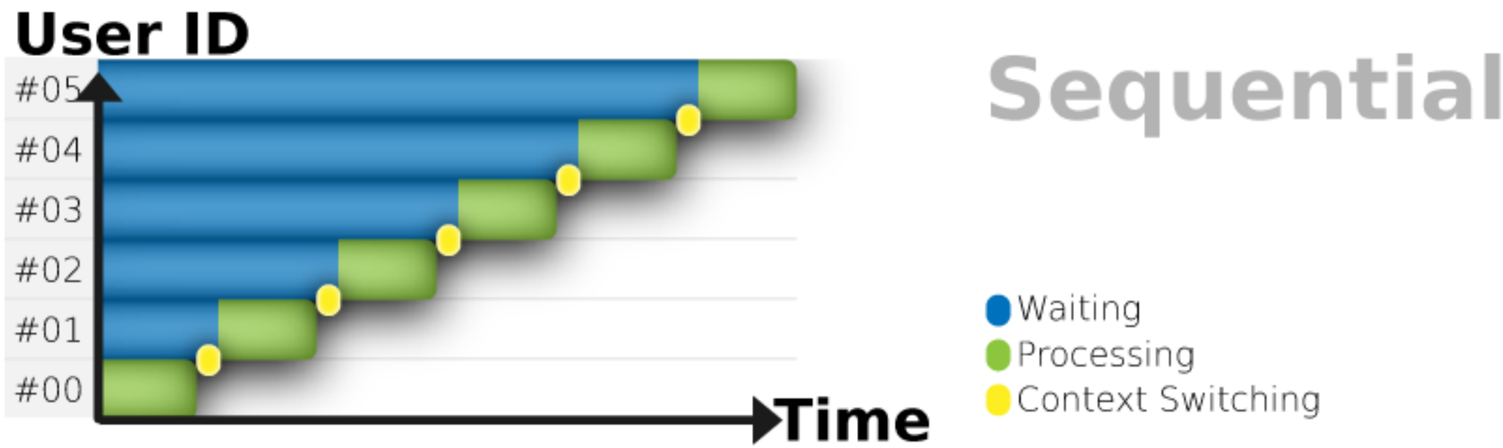

User ID
sequential
#05
#04
#03
#02
#01
Time
waiting
processing
context switching

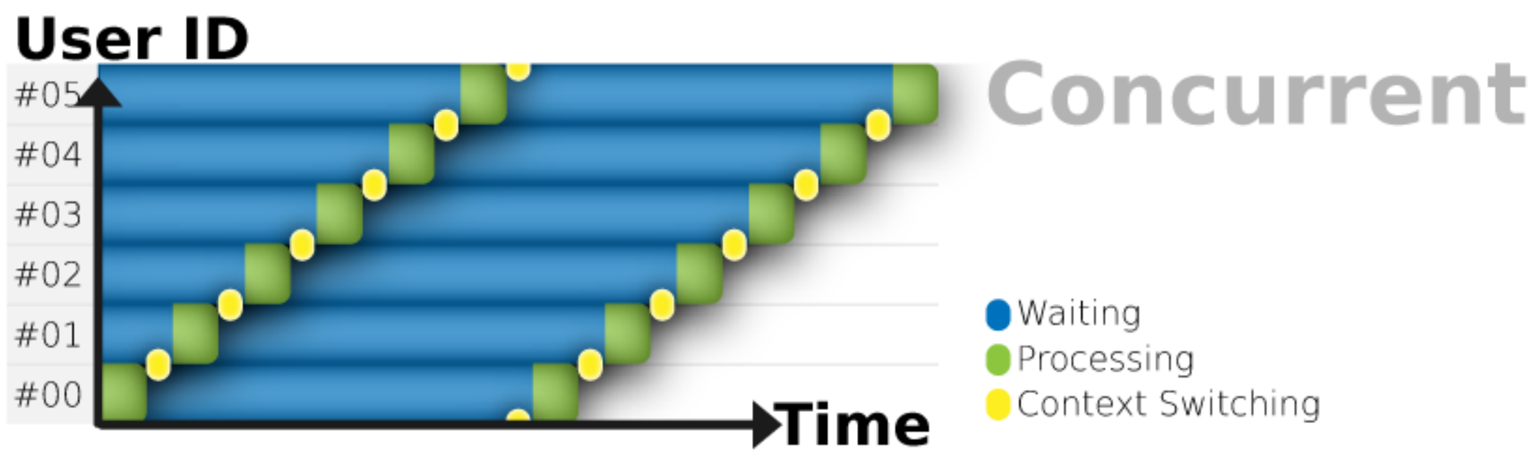

User ID
concurrent
#05
#04
#03
#02
#01
Time
waiting
processing
context switching


User ID
sequential
running time
of server
#05
#04
#03
#02
#01
waiting and processing time
of users
Time
User ID
concurrent
#05
#04
#03
#02
#01
Time

Sequential

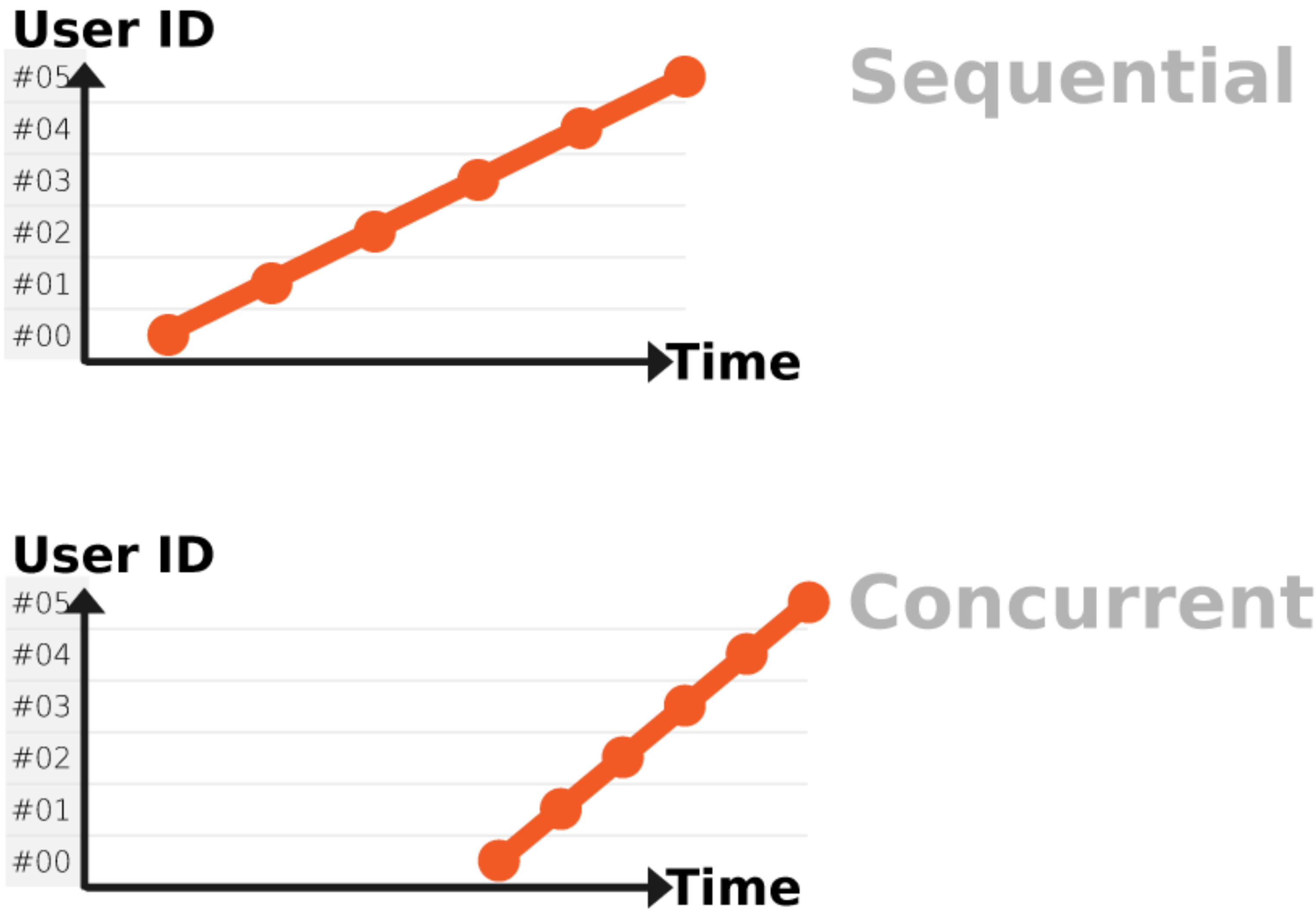

User ID
#05
#04
#03
#02
#01
#00
Time
Waiting
Processing
Context Switching


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

Sequential

Concurrent


User ID
#05
#04
#03
#02
#01
#00
Time
Sequential
User ID
#05
#04
#03
#02
#01
#00
Time
Concurrent

Scalable != Fast

Scalable != Fast

Scalable == Less complaining

Rails is not fast

Rails is not fast

Rails might be scalable

so when do we want concurrency?

以上 FREEZE *o*

When context switching cost is much cheaper than a single task

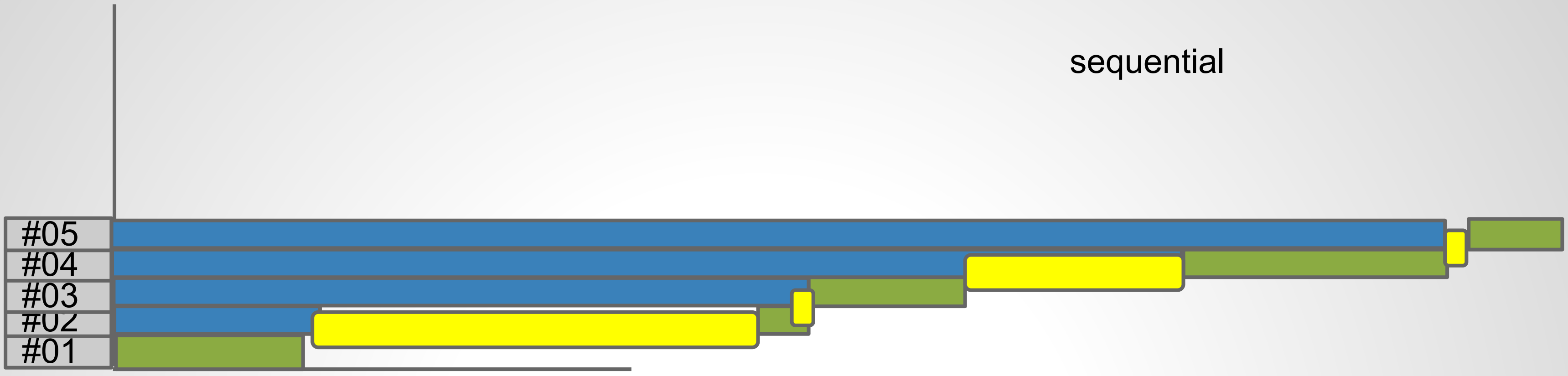
When context switching cost is much cheaper than a single task

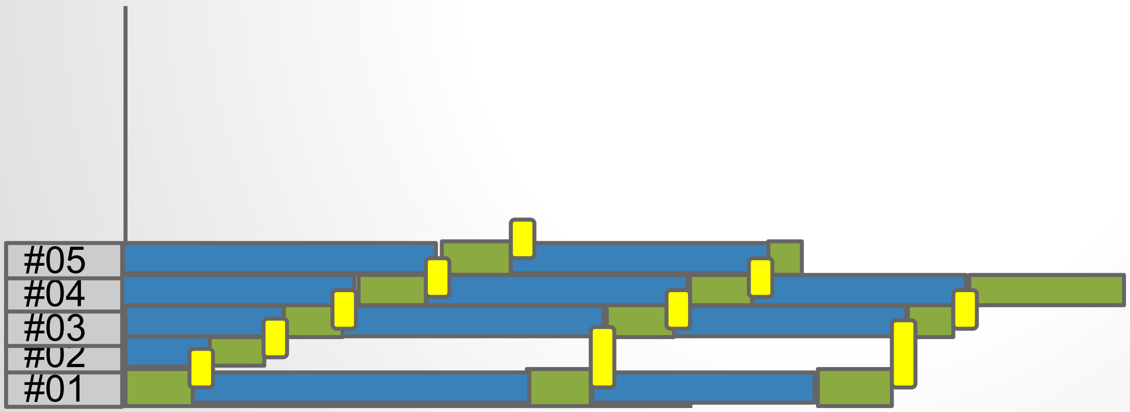
or if you have much more cores than your clients (= no context switching)

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, then it might not be worth it


sequential
#05
#04
#03
#02
#01


concurrent
#05
#04
#03
#02
#01

Do kernel threads context switch fast?

Do user threads context switch fast?

Do fibers context switch fast?

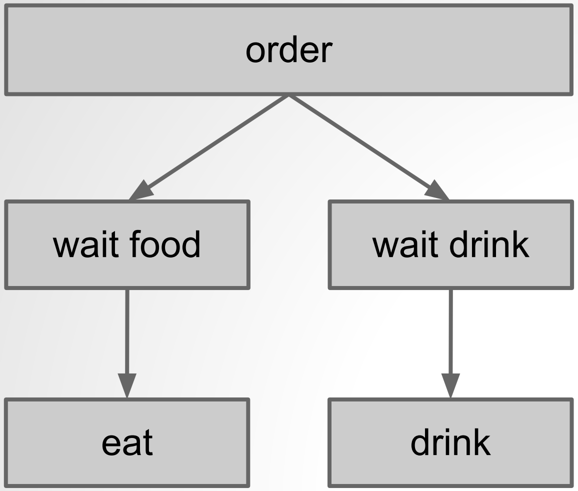
a ton of different concurrency models then invented

each of them has different strengths to deal with different tasks

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

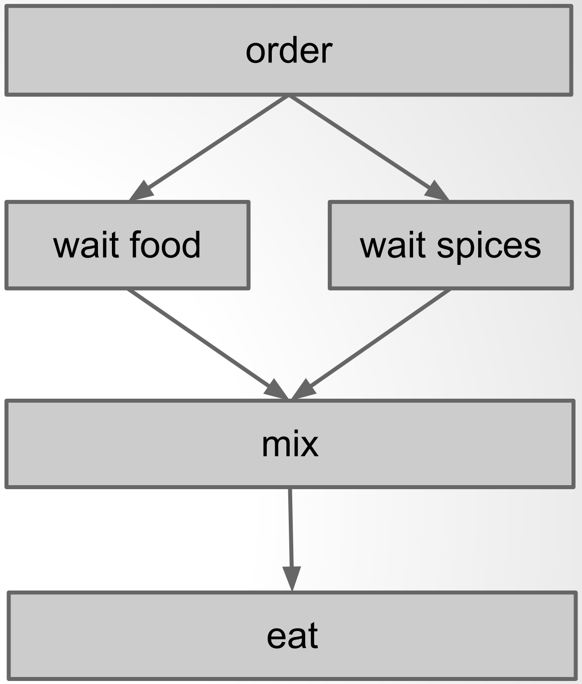
data dependency -- two patterns of composite tasks

- linear data dependency
- mixed data dependency

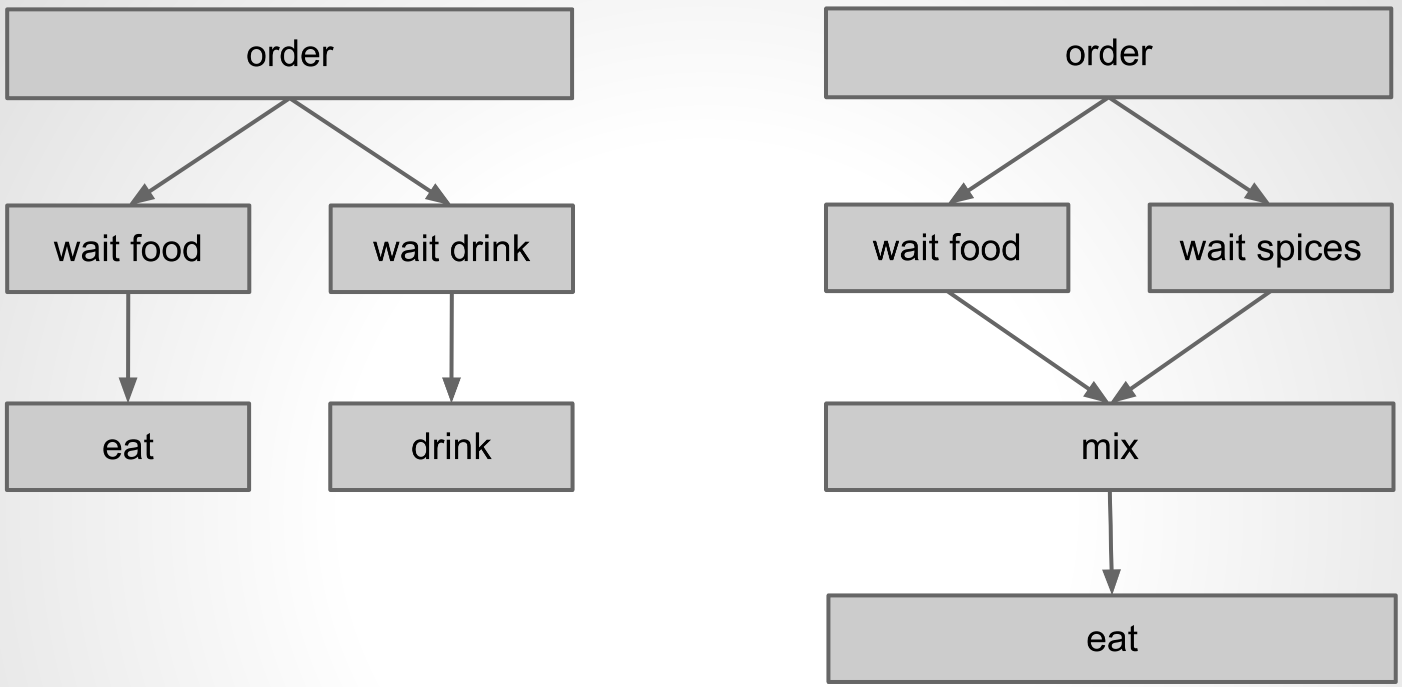


order_food -> eat

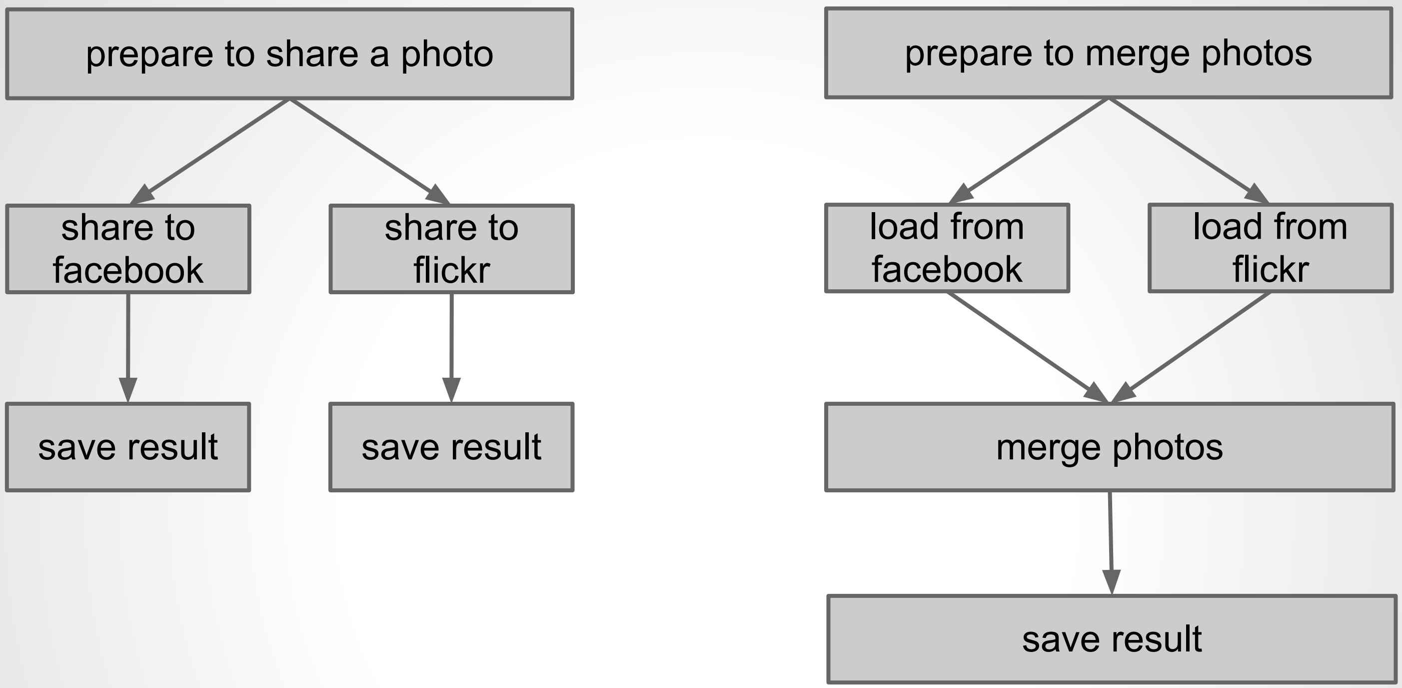
order_tea -> drink

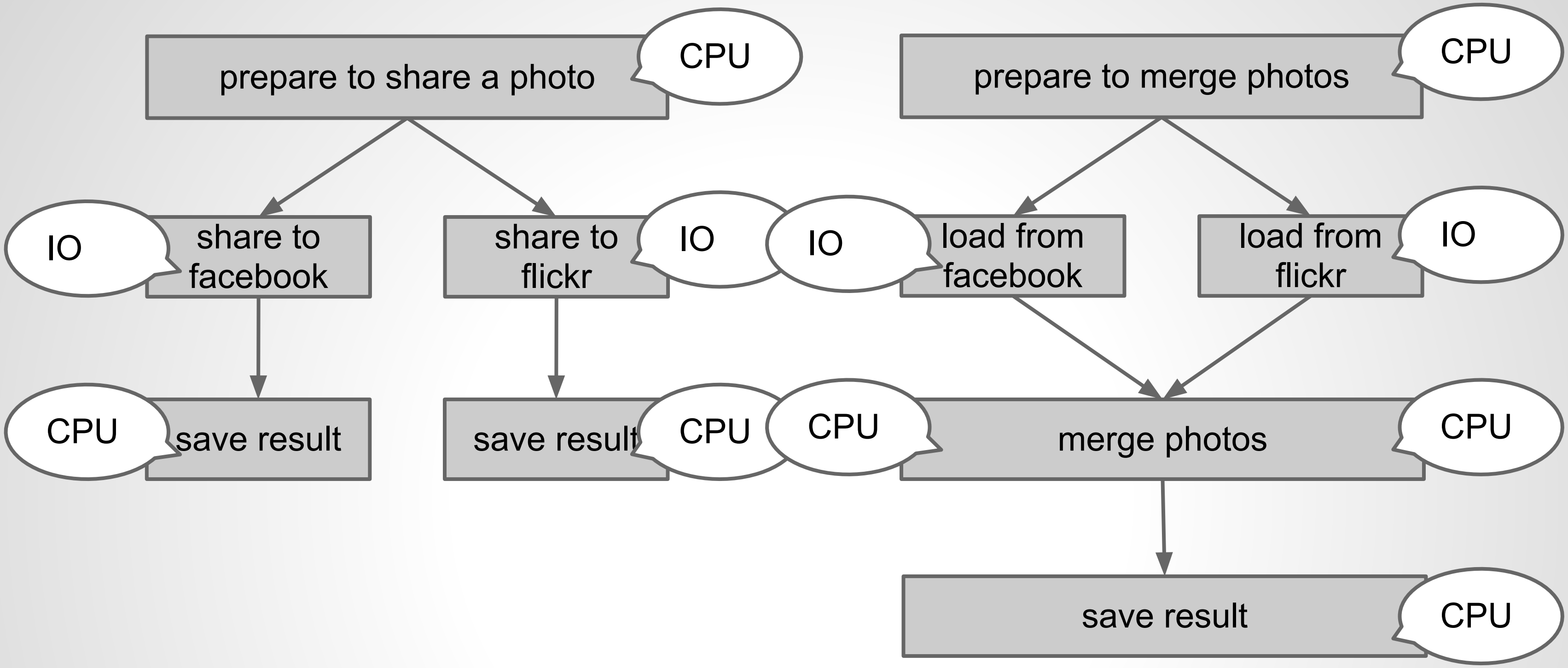


[order_food, order_spices] -> add_spices -> eat


order
wait food
wait drink
eat
drink
order
wait food
wait spices
mix
eat

two types of tasks:
CPU bound tasks
I/O bound tasks


prepare to share a photo
share to facebook
share to flickr
save result
save result
prepare to merge photos
load from facebook
load from flickr
merge photos
save result


prepare to share a photo
CPU
share to facebook
IO
share to flickr
IO
save result
CPU
save result
CPU
prepare to merge photos
CPU
load from facebook
IO
load from flickr
IO
merge photos
CPU
CPU
save result
CPU

what we have

we don't only talk about performance, we also talk about interface, since we human write programs the easy way, but not the hard way

it would be good if the interface we're using is orthogonal to its implementation

there are two advantages for this:

a) we don't have to change our code if the implementation is changed. (that said, we can also switch implementation to see how they work differently)

b) we don't need to really know the implementation detail in order to use this interface

linear data dependency

this is an easy one

interface -- callback

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

this might be bad, blocking drink while ordering food

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

but what if we could control side-effect and do static analysis?
not going to happen on ruby though

mixed data dependency

here comes the dragon

if the interface we only have is callbacks...

we don't want to do this:

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

we don't really want to do this either, but we have no choices if what we only have is callback and we want order_food and order_spices to run in a concurrent way

```
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 by how?

implementation

forget about data dependency for now, let's focus on implementation for a single task

basically we have two main choices:

a) threads with synchronous (blocking) interface

this could be used for either CPU bound or IO bound operations

b) reactor with asynchronous (callback) interface

this could only be used on I/O bound operations, since it is a

if order_food is I/O bound

so it could be done in either a thread or with a reactor the implementation -- how we define order_food with a thread

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

the implementation -- as for with a reactor...

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

the implementation -- how we define order_food with a reactor

```
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
```

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

if order_food is CPU bound

the implementation -- how we define order_food with a thread

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

yes, exactly the same

how about reactor?

sorry, you can't do that with a reactor. use a thread instead.

CPU: thread
(sockets and pipes) I/O: reactor

disk I/O: thread
see libev and libeio

back to mixed data dependency

if we could have some other interface than callbacks...

we can do it with threads easily

```
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)
```

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

can we do better?

can we do better?

YES!

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
```

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

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
```

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
```

Turn threads callback back to synchronized like using join

```
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)
```

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 or 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

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))
```

who got futures?

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

http://en.wikipedia.org/wiki/Futures_and_promises
https://github.com/cardinalblue/rest-core
https://github.com/celluloid/celluloid

a more complex (real world) example: (picture)
* 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 type Fork and type Diamond

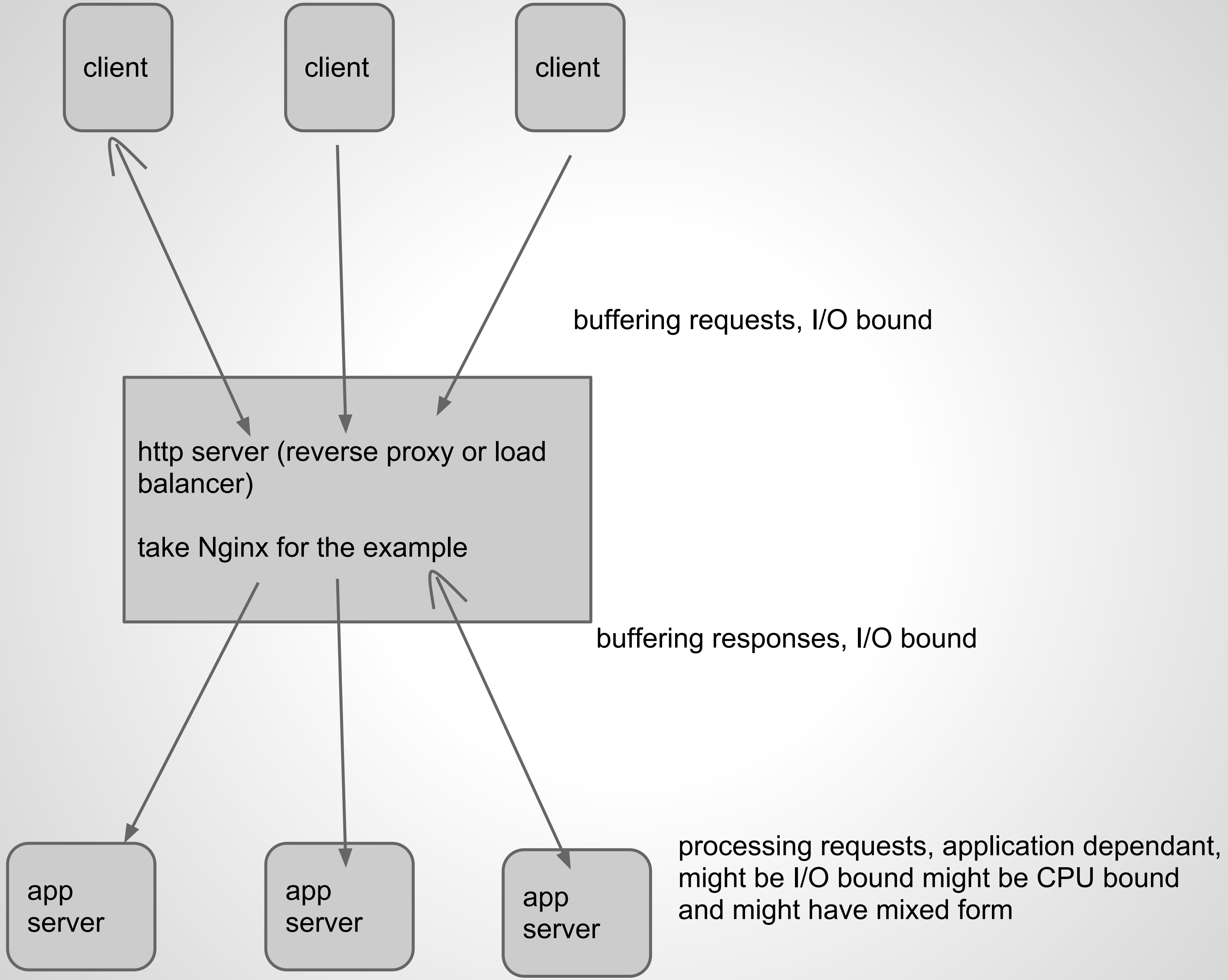
and how do we do that in a web application? we'll need to do the above example in a concurrent way. i.e. (last picture * 3)

application servers

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

two types of concurrency

- network concurrency
- application concurrency


client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
app server
app server
app server
processing requests, application dependant, might be I/O bound might be CPU bound and might have mixed form

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)

unicorn uses pre-forked workers, thin uses clusters (launch multiple processes), puma uses threads; while rainbows could do anything above and more. you can even use zbatery to avoid forking (such as saving memory, or make it work more like puma)

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

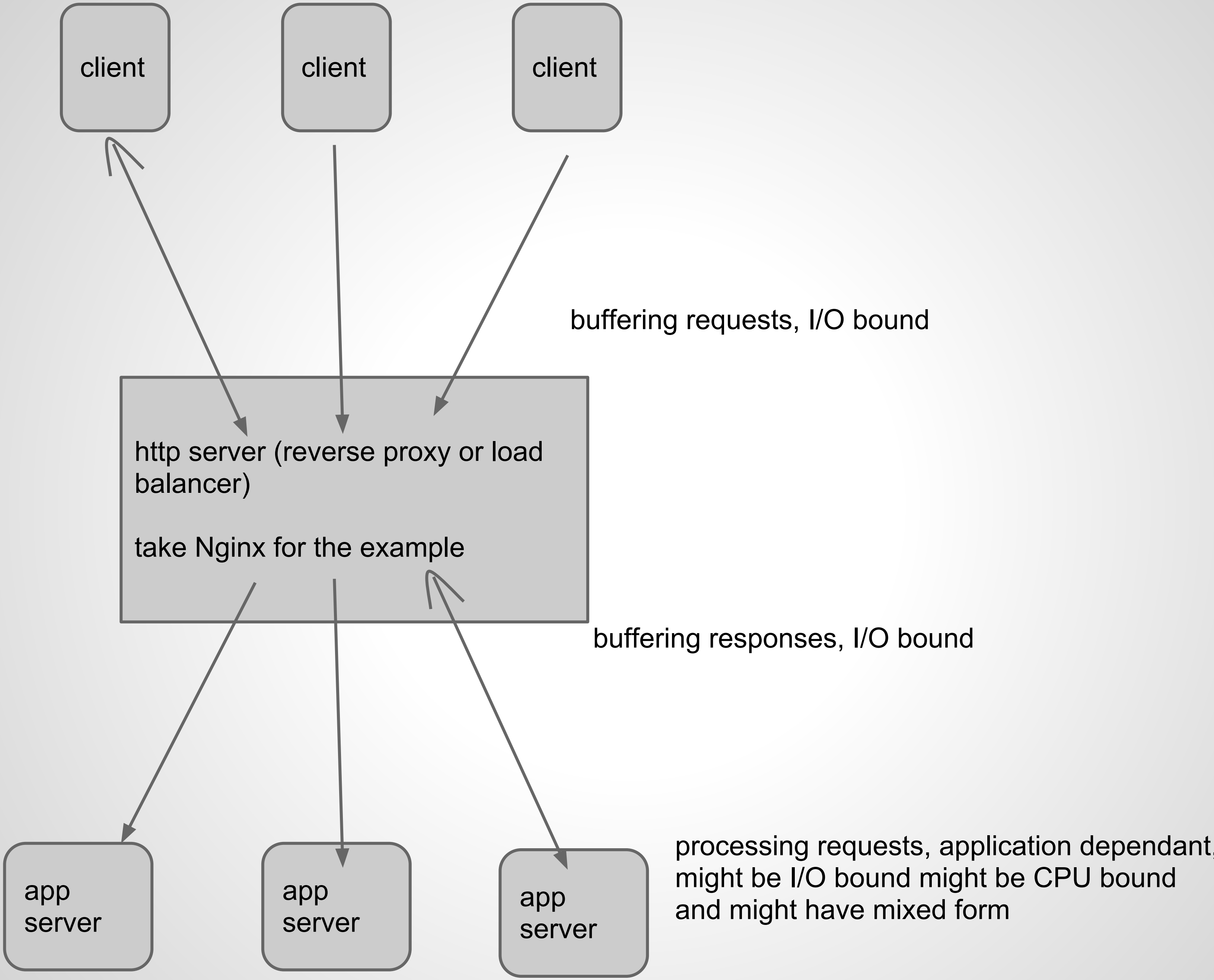
what if we want to make external requests to outside world? (e.g. facebook)

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

before we jump into the detail...

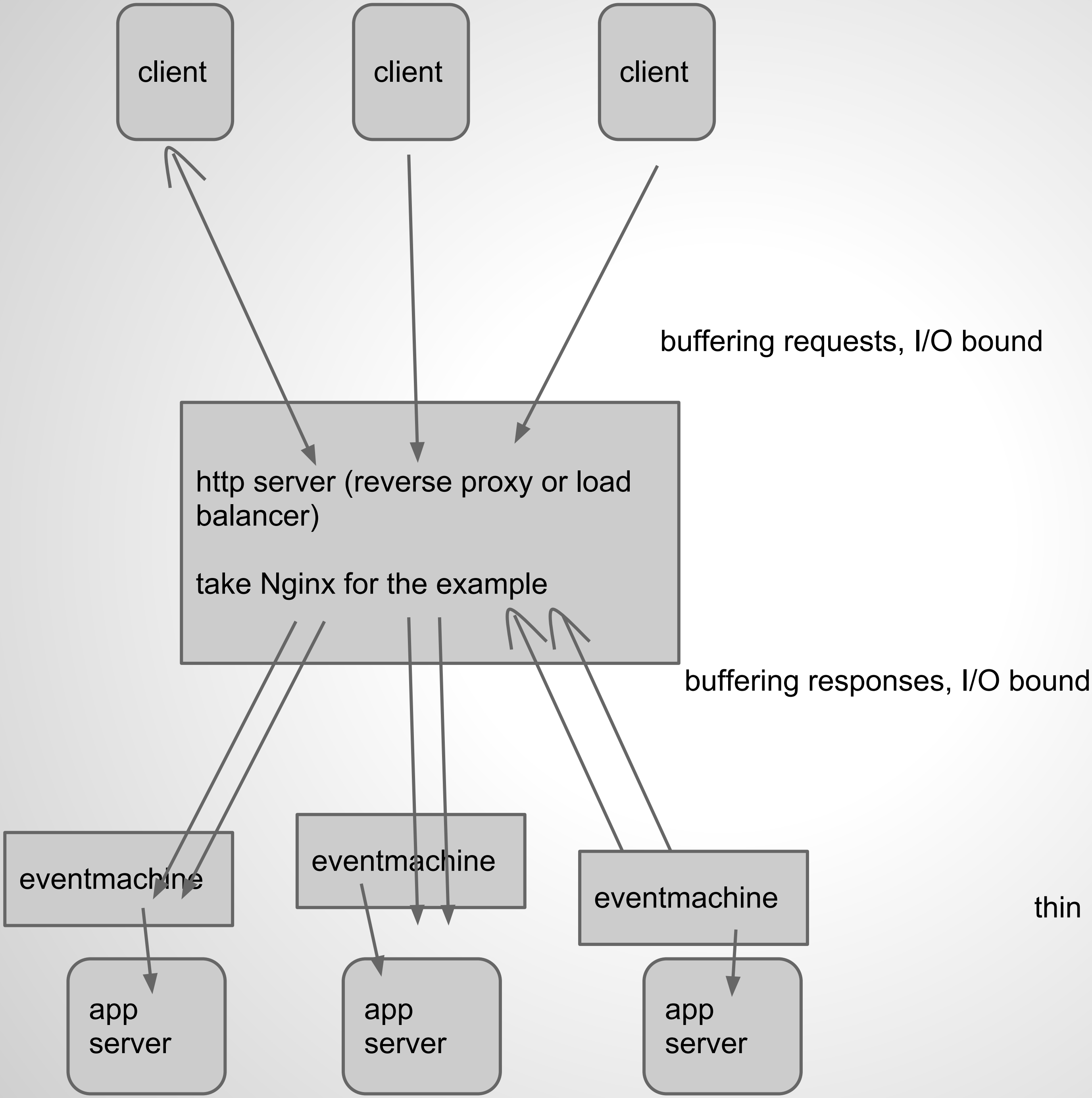
let's see some concurrent popular ruby application servers

thin, puma, unicorn family


client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
app server
app server
app server
processing requests, application dependant, might be I/O bound might be CPU bound and might have mixed form

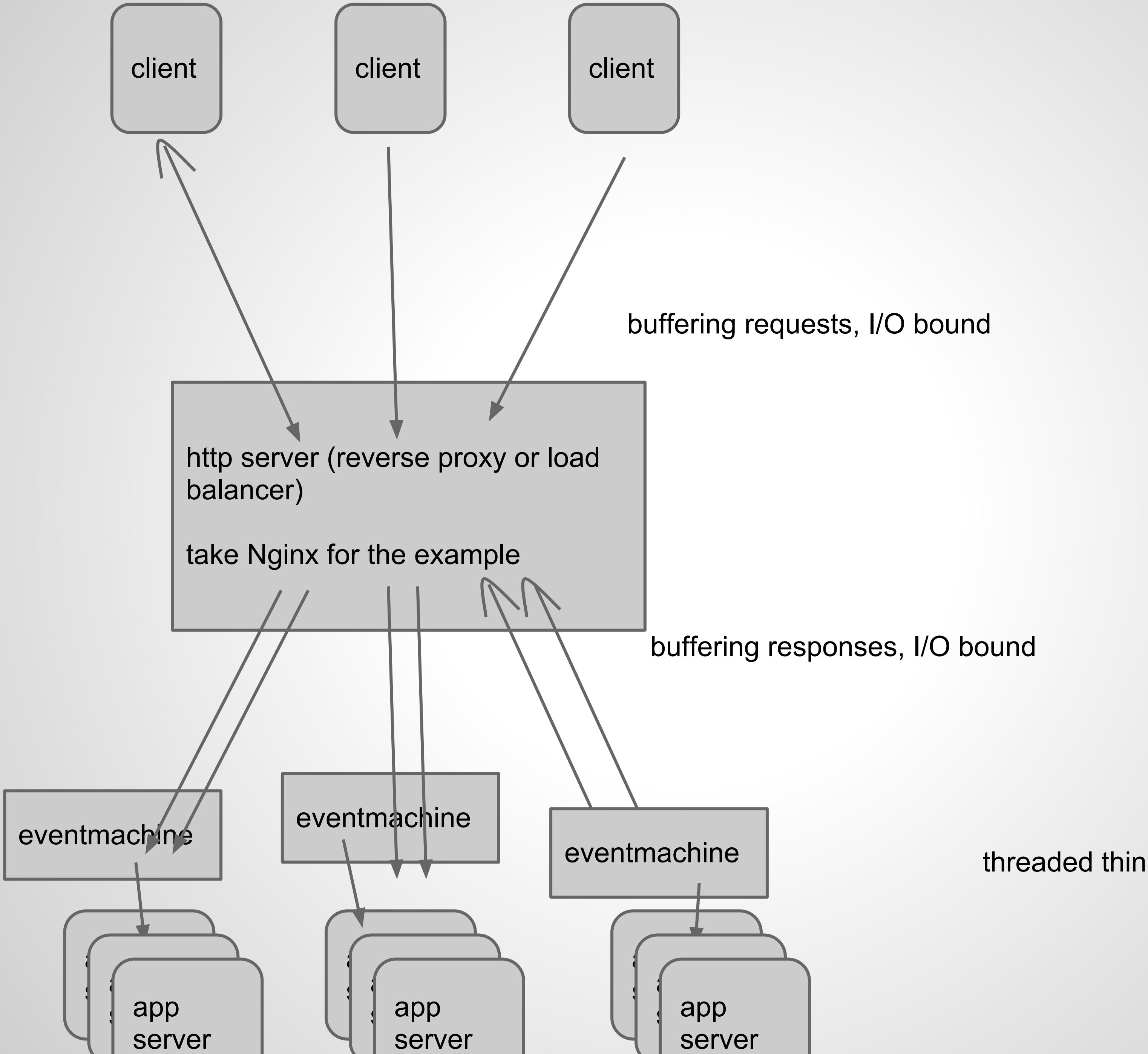
default thin: eventmachine (event loop) for buffering requests; no application concurrency

you can run thin cluster for application concurrency


client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
eventmachine
eventmachine
eventmachine
thin
app server
app server
app server

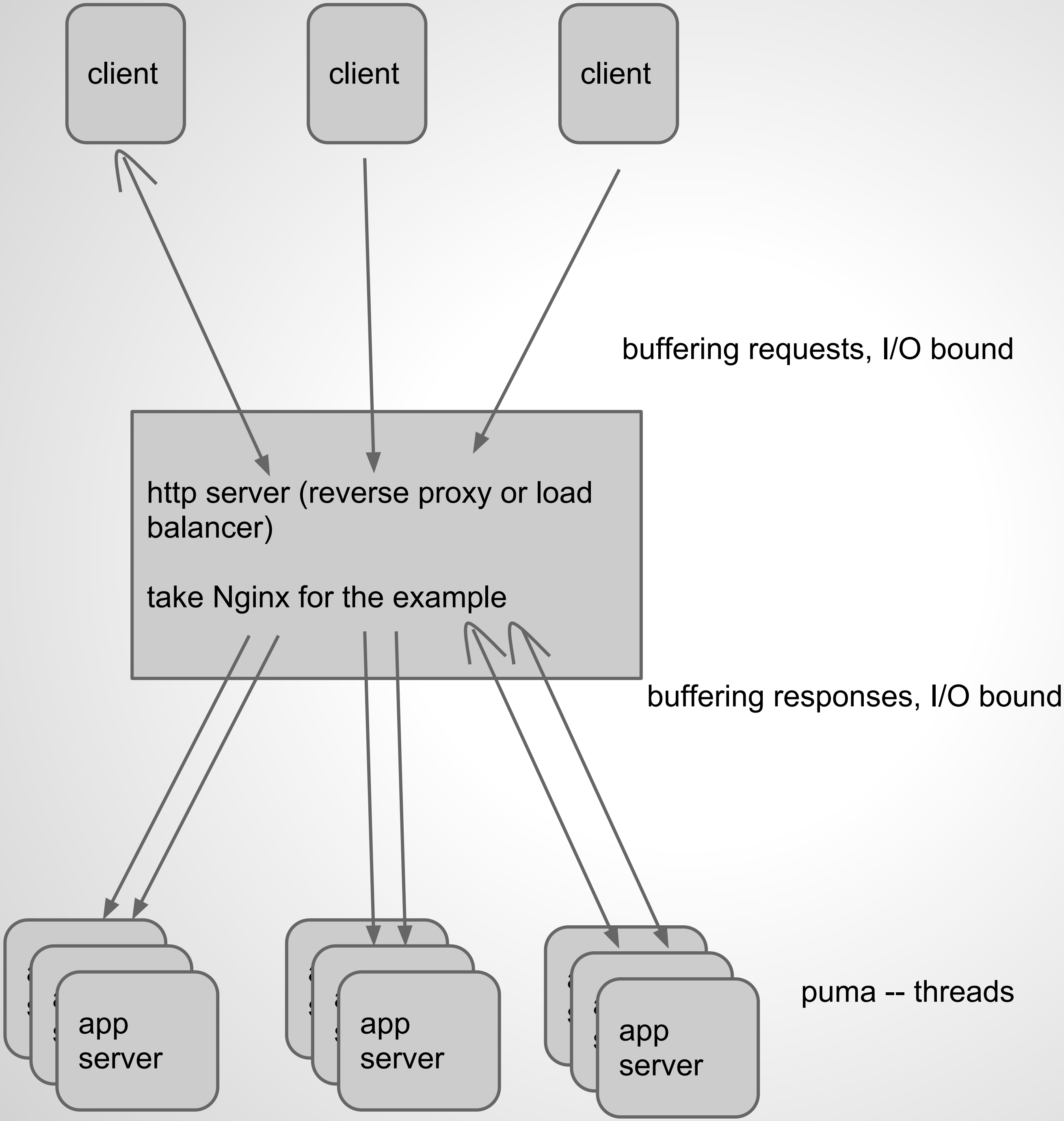
threaded thin: eventmachine (event loop) for buffering requests; thread pool to serve requests

you can of course run cluster for this

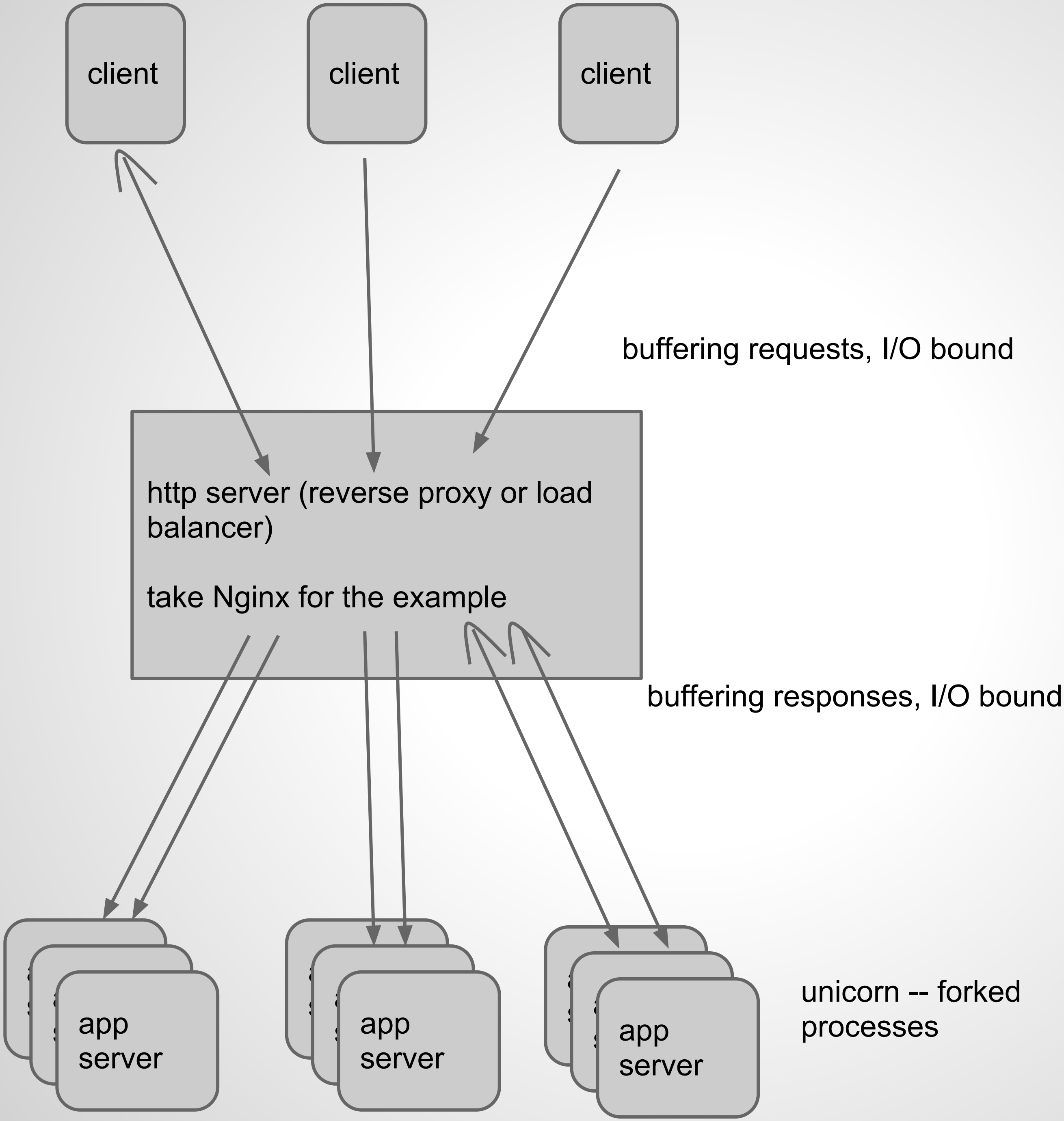

client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
eventmachine
eventmachine
eventmachine
threaded thin
app server
app server
app server

puma: thread pool

zbatery + ThreadPool = puma


client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
app server
app server
app server
puma -- threads

unicorn: no network concurrency; worker process application concurrency


client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
app server
app server
app server
unicorn -- forked processes

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?

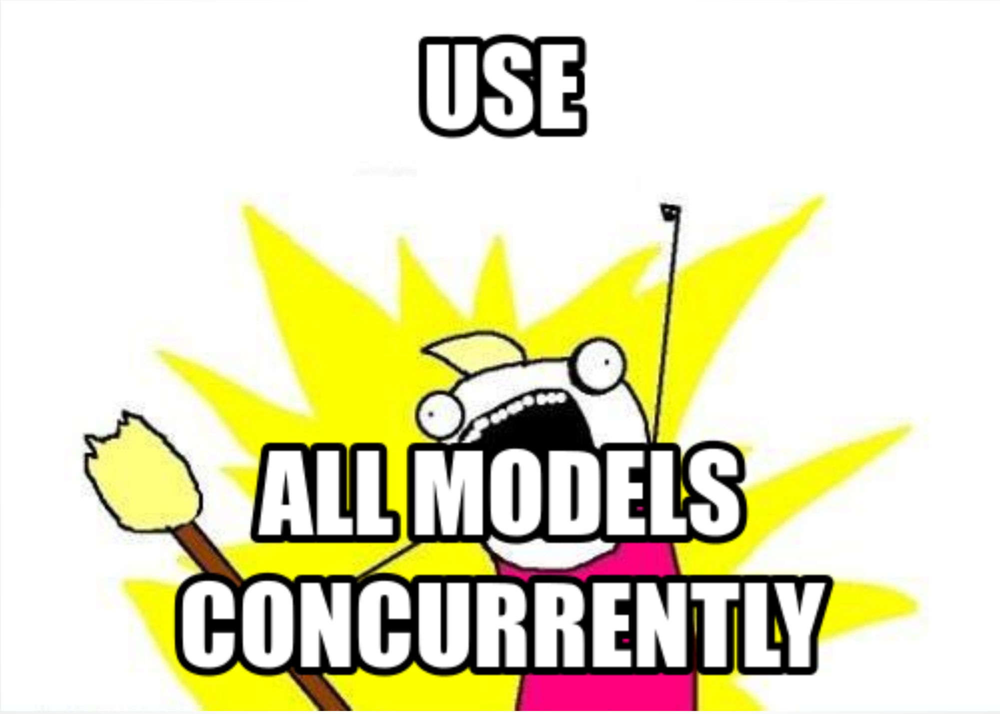
it's of course CPU bound, and should be handled in a real core/CPU

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

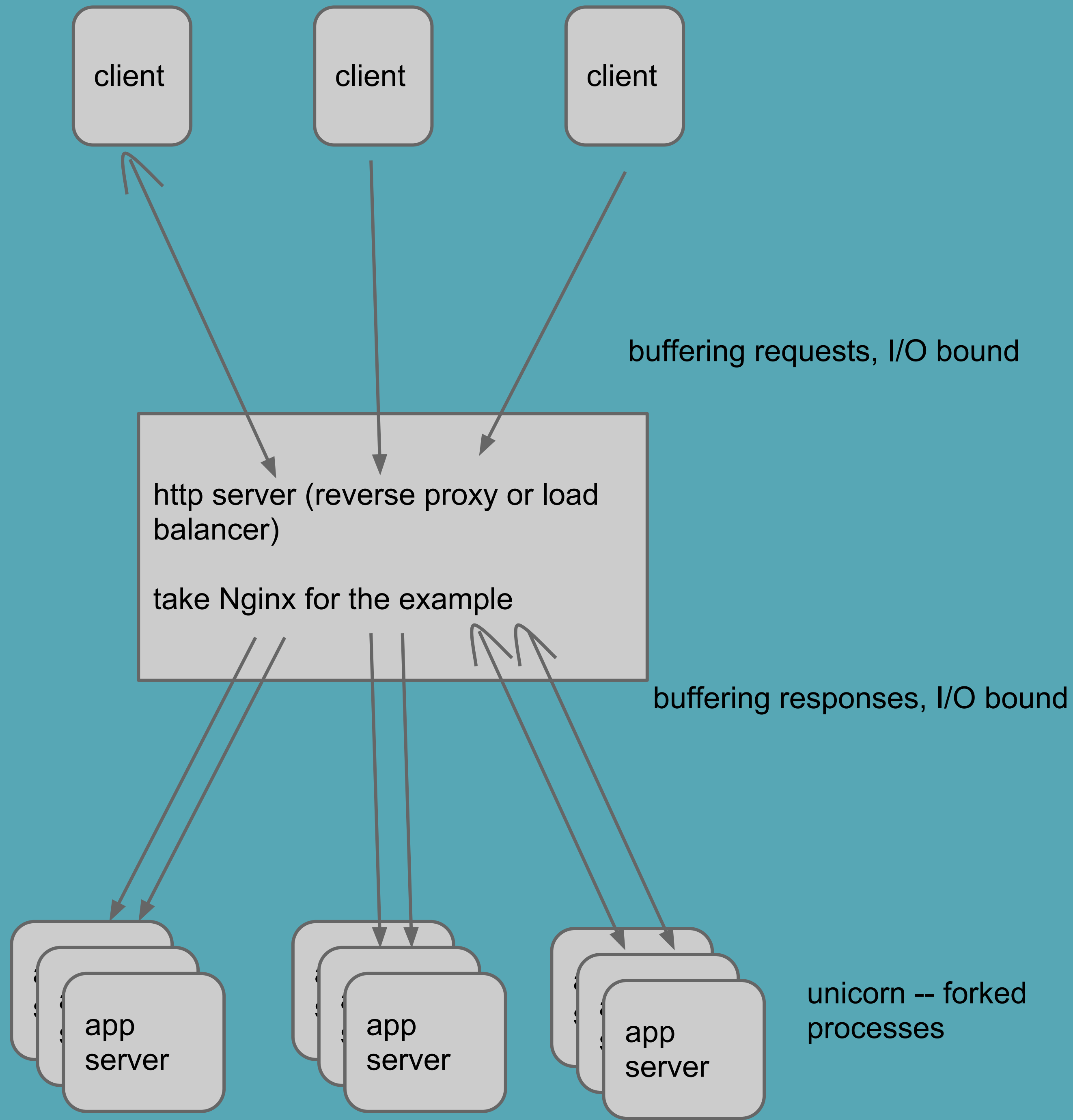
USE EVERYTHING



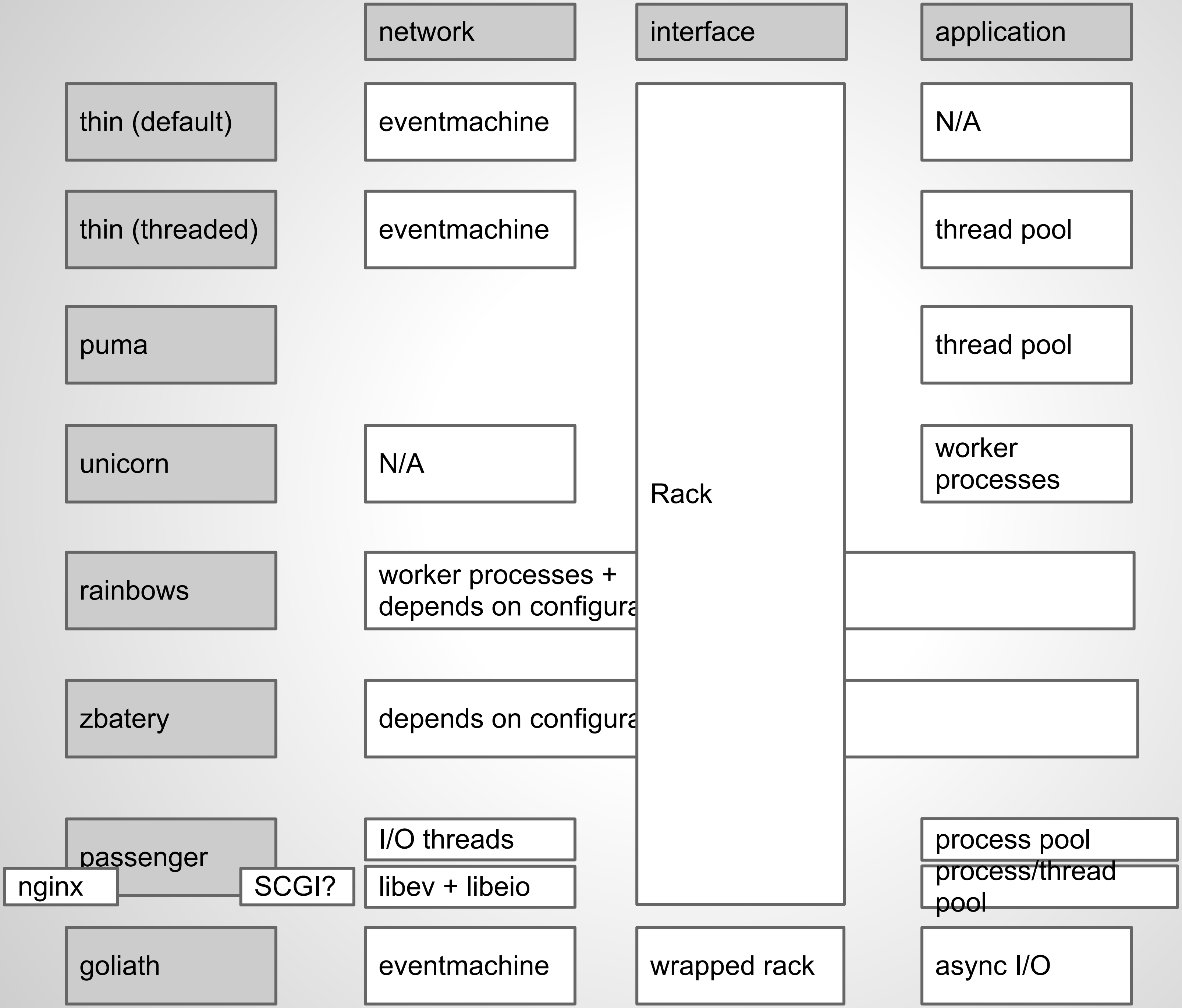
ultimate solution

for what i can think of right now

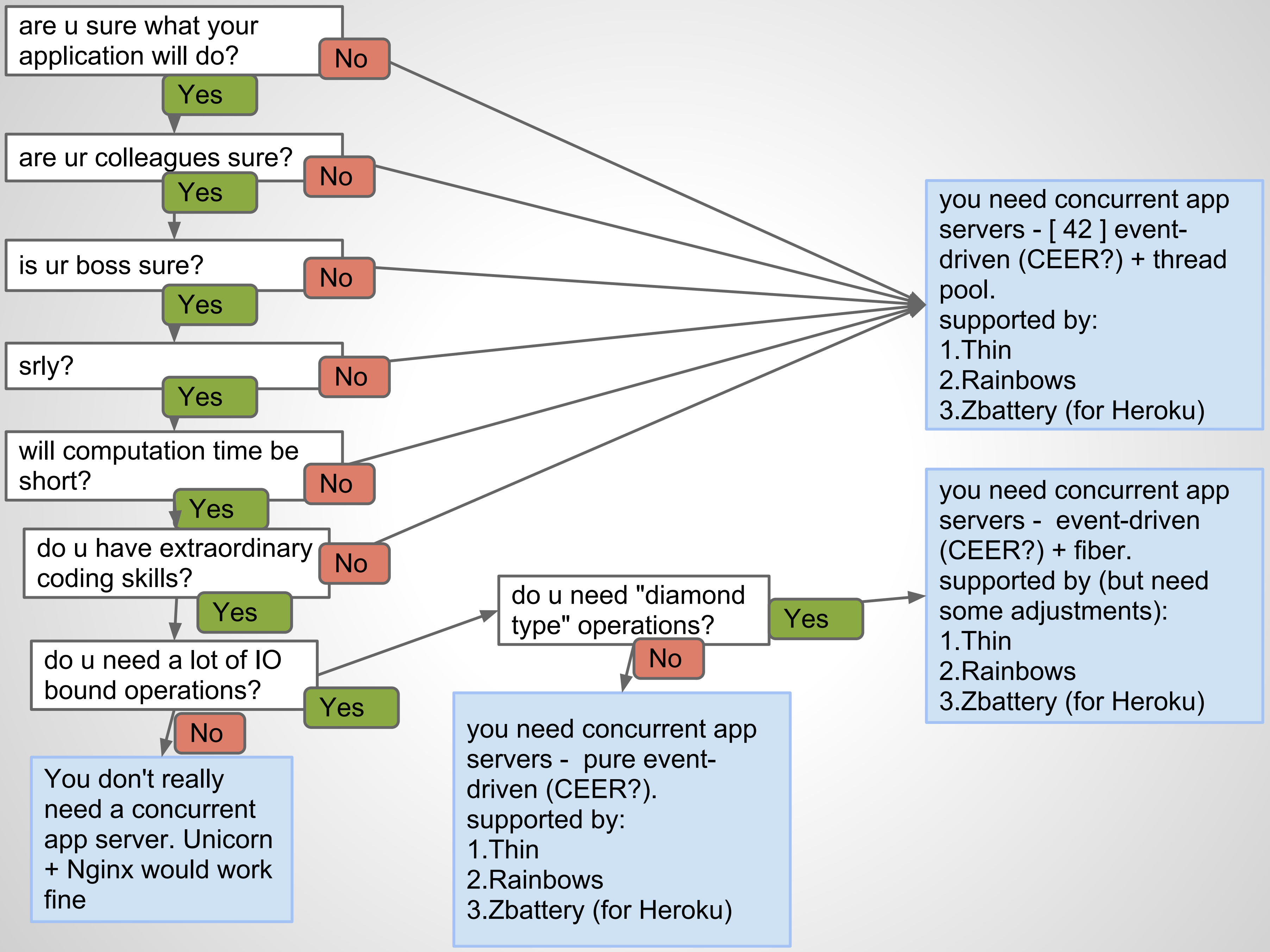
rainbows + eventmachine + thread pool + futures!


client
client
client
buffering requests, I/O bound
http server (reverse proxy or load balancer)
take Nginx for the example
buffering responses, I/O bound
app server
app server
app server
unicorn -- forked processes

and how do we do that in a web application?
we'll need to do the above example in a concurrent way. i.e. (last picture * 3)


network
interface
application
thin (default)
eventmachine
N/A
thin (threaded)
eventmachine
thread pool
puma
thread pool
unicorn
N/A
worker processes
Rack
rainbows
worker processes + depends on configura
zbatery
depends on configura
passenger
I/O threads
process pool
nginx
SCGI?
libev + libeio
process/thread pool
goliath
eventmachine
wrapped rack
async I/O

conclusion: your choice


are u sure what your application will do?
No
Yes
are ur colleagues sure?
No
Yes
is ur boss sure?
No
Yes
srly?
No
Yes
will computation time be short?
No
Yes
do u have extraordinary coding skills?
No
Yes
do u need a lot of IO bound operations?
Yes
No
You don't really need a concurrent app server. Unicorn + Nginx would work fine
do u need "diamond type" operations?
Yes
No
you need concurrent app servers - [42] event-driven (CEER?) + thread pool.
supported by:
1.Thin
2.Rainbows
3.Zbattery (for Heroku)
you need concurrent app servers - event-driven (CEER?) + fiber.
supported by (but need some adjustments):
1.Thin
2.Rainbows
3.Zbattery (for Heroku)
you need concurrent app servers - pure event-driven (CEER?).
supported by:
1.Thin
2.Rainbows
3.Zbattery (for Heroku)

Q?

EXTRA

some free talk

Reinvent the wheel, not the car

```
fdata = RC::Facebook.new.get('me')
tdata = RC::Twitter.new.get('me')
t0 = Thread.new{ merge_photos fdata, tdata }
t1 = Thread.new{ mix_photos fdata, tdata }
t0.join; t1.join
merge_final_photo
Thread.new{ do_some_fancy_stuffs }
RC::Facebook.new.post('me', final) # non-blocking
RC::Twitter.new.post('me', final).tap{} # blocking
```

network concurrency VS application concurrency

network concurrency = buffering client request

nginx could do this well

we need reactor pattern, event loop, or whatever you call it

it's a full I/O issue

application concurrency = do the real business

might be I/O bound or CPU bound, it depends on your business. they are two different things default thin server = event loop network concurrency + no application concurrency

threaded thin server = event loop network concurrency + threaded application concurrency

if you have nginx in front, then there's no much point for an event loop, or is there?

do you make external network request?

if so, then it matters. if not, then it doesn't matter ruby people, please learn about threads

don't easily trust the hype

be conscious

trust the old good things

threads are old...

it's not that hard

threads are hard, if you don't try to understand it

cores are growing, threads would be more and more important in the future. threading is not simply a technique, it's a concept... which we have to overcome in the end, don't be afraid of it thin = zbatery + eventmachine

thin clusters = rainbows + eventmachine

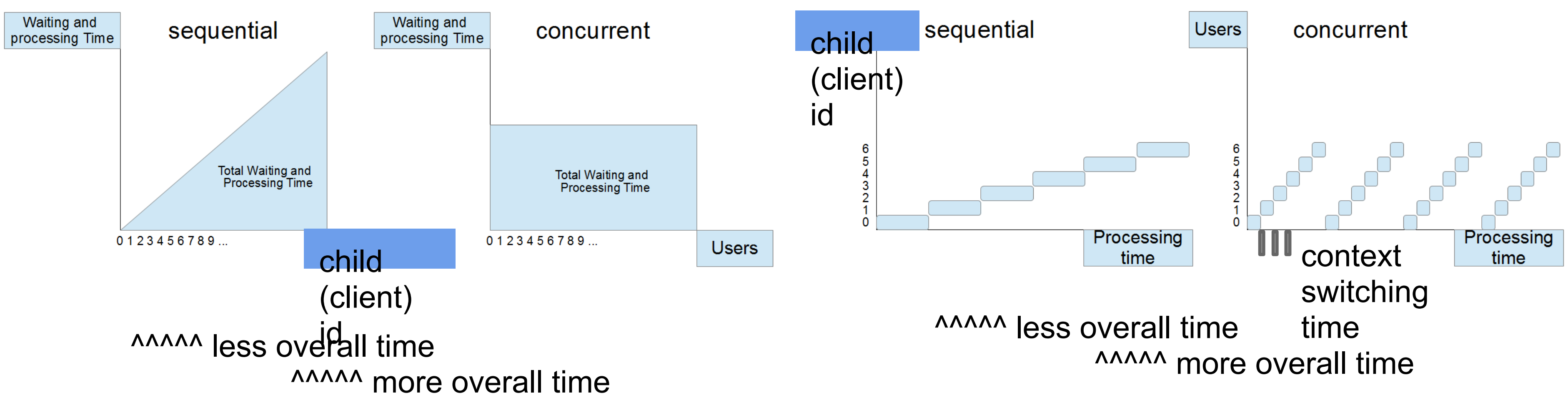
puma = zbatery + thread pool

bonus content? not sure if i would have time to talk about this, since it would be the toughest content

application concurrency

（把上一張的圖中的食物替換回 I/O & CPU）

considering one mom (thread/process)



considering one mom (thread/process)

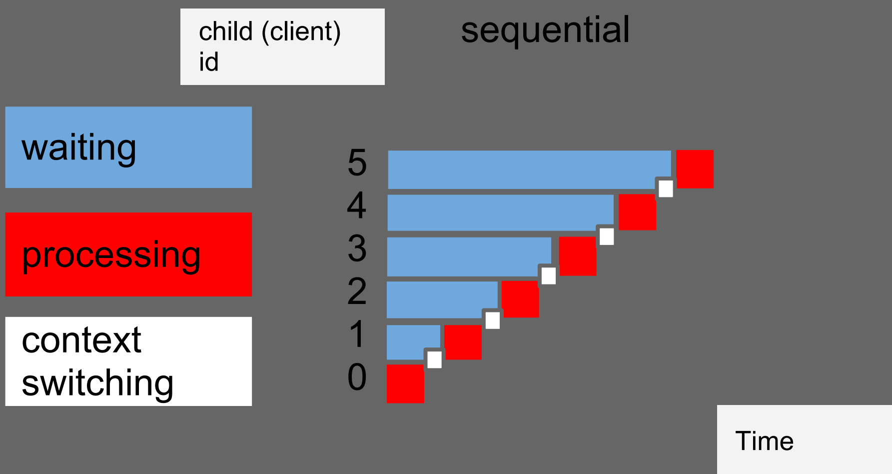
sequential

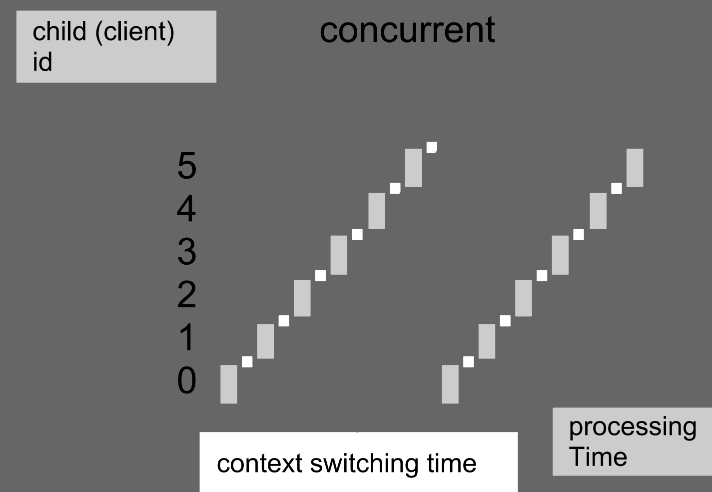
^^^^^ less overall time

concurrent

^^^^^ more overall time

considering one mom (thread/process)





^^^^^ less overall time

^^^^^ more overall time

(使用前&使用後(1))

（圖一：總等待時間，用了 concurrency 以後圖中的面積反而會變大）

X: 個別 client

Y: 等待時間

(使用前&使用後(2))

(圖二:甘特圖, 執行不同使用者的 request 的時候, 用了 concurrency 會讓單一使用者的處理時間變的不連續)

X: 時間

Y: 個別client

（可以用 Concurrency 處理的問題類型, Data Dependency）

（圖一, 叉子型:等待兩個不同來源的資料, 各自獨立, 先到的就先處理 (food + drink), 像是用手機同時分享照片到FB跟Twitter）

（圖二, 鑽石型:兩個來源的資料都等到了才能處理(food + spices), 像是抓FB跟Flickr的照片來合成一張Collage） fork shaped

tasks (eat, drink) depend on independent tasks (order_food, order_tea)

this is ideal for parallels computing (if we're not enforcing task resolution ordering), and it is quite easy to solve for whatever methods

order_food -> eat
order_tea -> drink

diamond shaped

task (add_spices) depends on multiple independent tasks (order_food, order_spices)

[order_food, order_spices] -> add_spices -> eat if we're not using concurrency, all clients would need to wait for a different period of time, and some unfortunate users might need to wait for a long long time, since (s)he needs to wait for all the preceding users had done their requests.

we don't want that, we want all users wait for the same time, eliminate unfortunate users, making our program "fair". 公平正義

but making our program run concurrently, would actually make our program run slower, in terms of overall processing time. that's the trade of 公平正義. 公平正義的政府效率比較差

Why Concurrency

（解釋 concurrency 這個東西其實並不會讓處理速度變快，相反的還會變慢，只是他可以避免某個特定使用者等超久的情況出現。）

（所以...的情況適合採用 concurrency，但...的情況就不適合 concurrency）

（接下來幾張秀code：鑽石型的實做方式）

1. synchronized I/O + thread
2. asynchronized I/O + callback
3. asynchronized I/O + callback + thread (coroutine)

synchronized CPU - 依序執行
asynchronized CPU - threads 輪流執行

synchronized I/O - 依序
asynchronized I/O -

a simple reactor

```
class Reactor
  def run
    until read_socks.empty? &&
            write_socks.empty?
    rs, ws = IO.select(read_socks, write_socks,
                                  [], 0.05)
    read_data(rs)  if rs
    write_data(ws) if ws
ennnd
```

a simple reactor

```
class Reactor
  def read sock, &callback
    read_socks << sock
    read_calls[sock.object_id] = callback
  end
  def write sock, data, &callback
    write_socks << sock
    write_pairs[sock.object_id] = [data, callback]
 ennd
```

a simple reactor

```
class Reactor
  def read_data rs
    rs.each do |r|
      begin
        read_calls[r.object_id].call(r.read_nonblock(81
      rescue Errno::EAGAIN, ::IO::WaitReadable
      rescue Errno::ECONNRESET, EOFError
        read_socks.delete(r)
ennnd
```

a simple reactor

```
class Reactor
  def write_data ws
    ws.each do |w|
      data, callback = write_pairs[w.object_id]
      begin
        data.slice!(0, w.write_nonblock(data))
        raise EOFError if data.empty?
      rescue ::IO::WaitWritable
      rescue EOFError
        write_pairs.delete(w.object_id)
        write_socks.delete(w)
        callback.call(w)
```

it doesn't seem to be a good pattern

but it's so ugly and tedious!

what if we want 42 kinds of different spices?

yes, yes, i know nodejs guys have some solution for this, but why invent a new method while the old good synchronous programming does do the job elegantly? you can also consider that as a DSL for synchronous programming let's see how to make your code synchronous

there are two approaches, depending on the architecture, you can use either threads or fibers to do that.

but note that fibers only work in an event loop (or say single threaded asynchronous programming) still looks ugly? we can go further with futures.

it can make your code exactly the same as synchronous one, but actually running asynchronous underneath.

（賣藥時間）

using rest-core to make concurrent requests with futures

maybe try to use celluloid to implement futures in the futures? that way, we can have a more consistent way to deal with either I/O or CPU bound operations.

i shouldn't create my own futures -- there's no futures for me (?)

confused?

interface / implementation	synchronized	asynchronized
thread	O	O
reactor	X	O
reactor + fiber	O	X
+futures reactor + fiber	O	O