Evented Ruby

vs

Node.js

http://zarious.deviantart.com/art/Spy-vs-Spy-WallPaper-2560X1024-115603200

KISS Performance

- Users happy => I'm happy
- Micro-optimizations are bad
- Easy to implement, easy to maintain
- Minimize code and server infrastructure changes

Overview

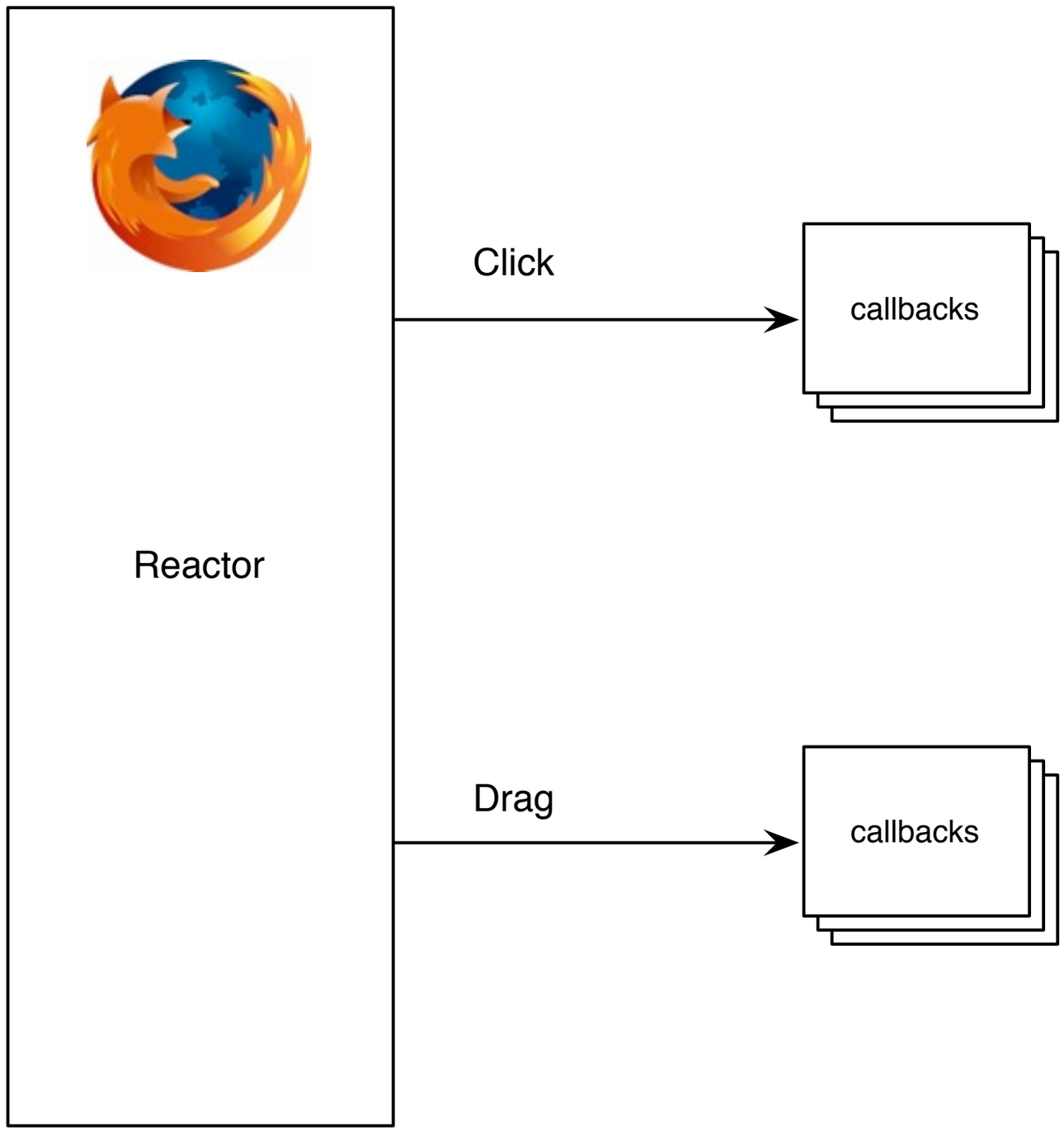
- Evented Programming
- Evented Web Backends
- Evented Ruby vs Evented Javascript
- Improving Rails Concurrency

Evented Programming

```
$("body").click(function(e) {
  $(this).css('color', 'red');
});
```

Event Callback

Reactor Pattern



This code is a simple example of the reactor pattern. In the pattern, a reactor is a system that listens for incoming events and then delivers those events to registered callbacks.

- In our example, the browser is the reactor, and we can register callbacks for dom events.

- domain specific reactors are reusable and save time and plumbing

Reactor Pattern

- Events: keyboard, mouse, touch
- Reusable reactors: browser, game loop

Some domains lend themselves naturally to the reactor pattern.

Node.js

Node.js uses an **event-driven**, **non-blocking** **I/O model** that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Blocking I/O

If RAM was an F-18 Hornet with a max speed of **1,190 mph**, disk access speed is a banana slug with a top speed of **0.007 mph**

http://blog.scoutapp.com/articles/2011/02/10/understanding-disk-i-o-when-should-you-be-worried

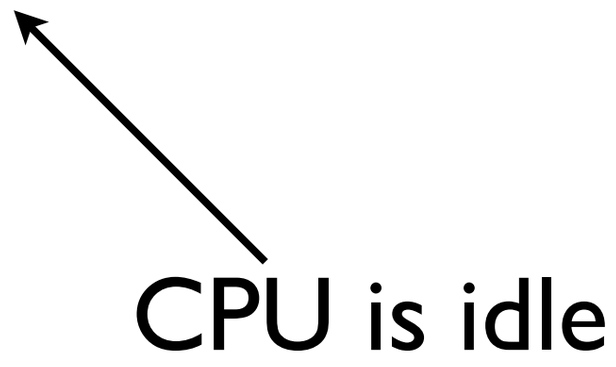
Blocking I/O is when slower devices can't get data fast enough for the CPU to process. CPUs are faster than memory, memory faster than disk and disk is faster than network access. If the CPU needs data from the disk, it has to be idled until that data is ready.

Blocking I/O

F = Fast F18 Hornet
S = Slow Banana Slug

```
data = File.read('file.txt')
```

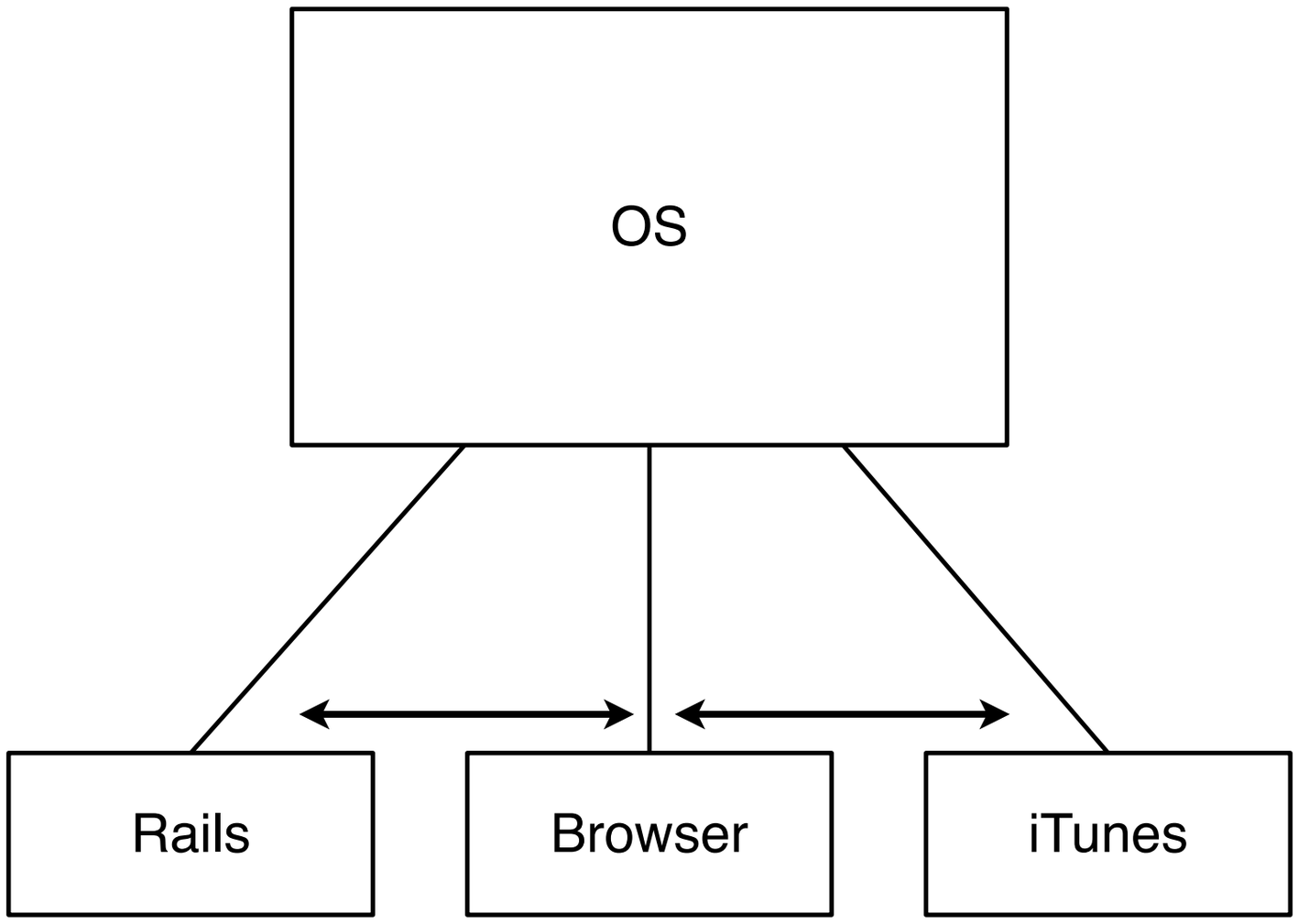
FSSSSSSSSSSSSSSSF



The operating system works with hardware caches to hide blocking I/O from us, so when we read a file, it looks like a single atomic operation.

If there was only 1 process running in the system, then if we read a file, the CPU would be idled the entire time it's waiting for the disk to return data.

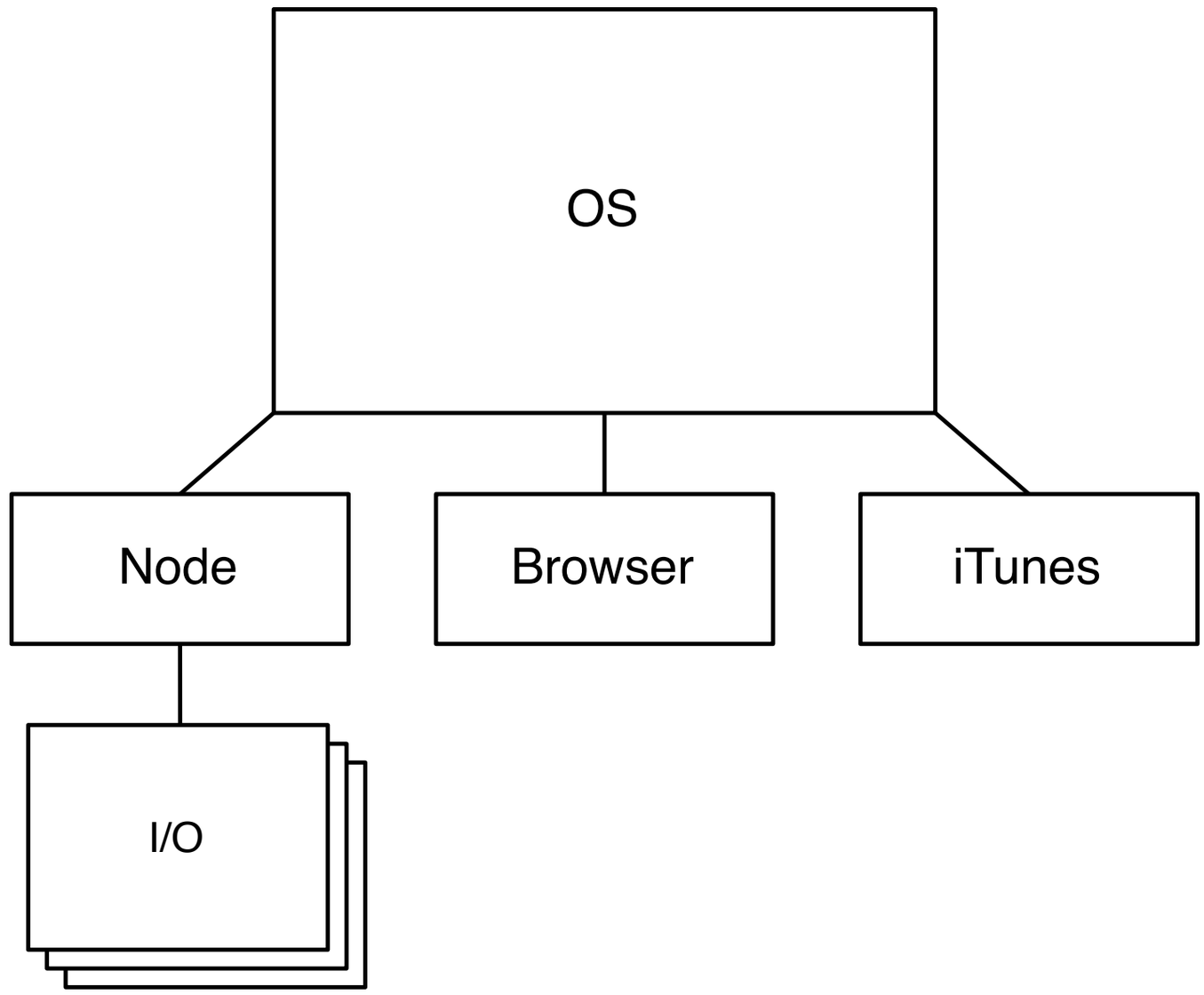
Blocking I/O



- Switches between busy processes
- OS and hardware caches hides complexity

Fortunately, the operating system is smart enough to let a CPU to work on other processes if blocking I/O is happening on one process.

Blocking I/O



Node.js switches between I/O within the same process

What Node.js does is optimize I/O one step further in the application layer. So rather than only having the OS switch between processes when the CPU is idle, Node runs a reactor and registers a I/O callback whenever it's about to start I/O.

- even if there was only one node process running on your system, it'd be able to internally dispatch I/O events so that your application code isn't blocked.

Web Apps

Up till now, we talked about how evented programming is used, and how it can be used for user events and for system events like blocking I/O.

But if the operating system already handles CPU switching when blocked on IO, why do we care about evented I/O for server side programming?

Web Apps

- blocking I/O decreases concurrency
- database, filesystem - disk
- S3, external APIs - network
- ImageMagick - shelling out

Up till now, we talked about how evented programming is used, and how it can be used for user events and for system events like blocking I/O.

But if the operating system already handles CPU switching when blocked on IO, why do we care about evented I/O for server side programming?

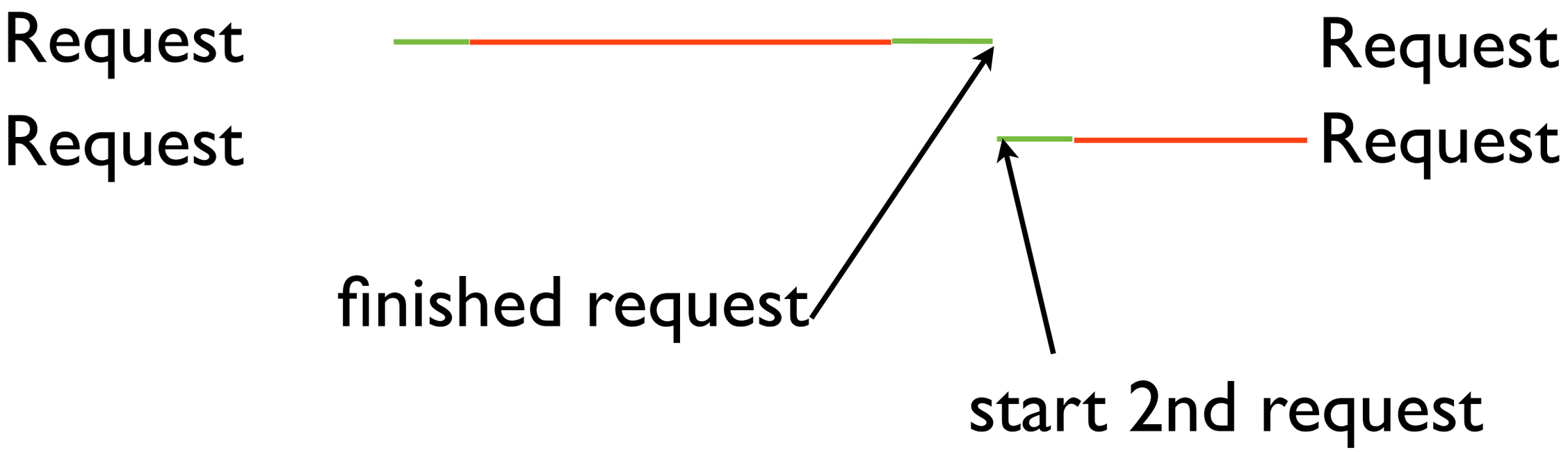
Rails Concurrency

```
tweet = Tweet.new(params["tweet"])   # 1.
tweet.shorten_links!                 # 2. network
tweet.save                           # 3. disk
```


A small example is if we were writing a controller action to create a new tweet...

Even in these 3 simple lines of code, we're doing blocking I/O by going to the network to shorten_links via an API and going to the disk to persist our data.

Rails Concurrency

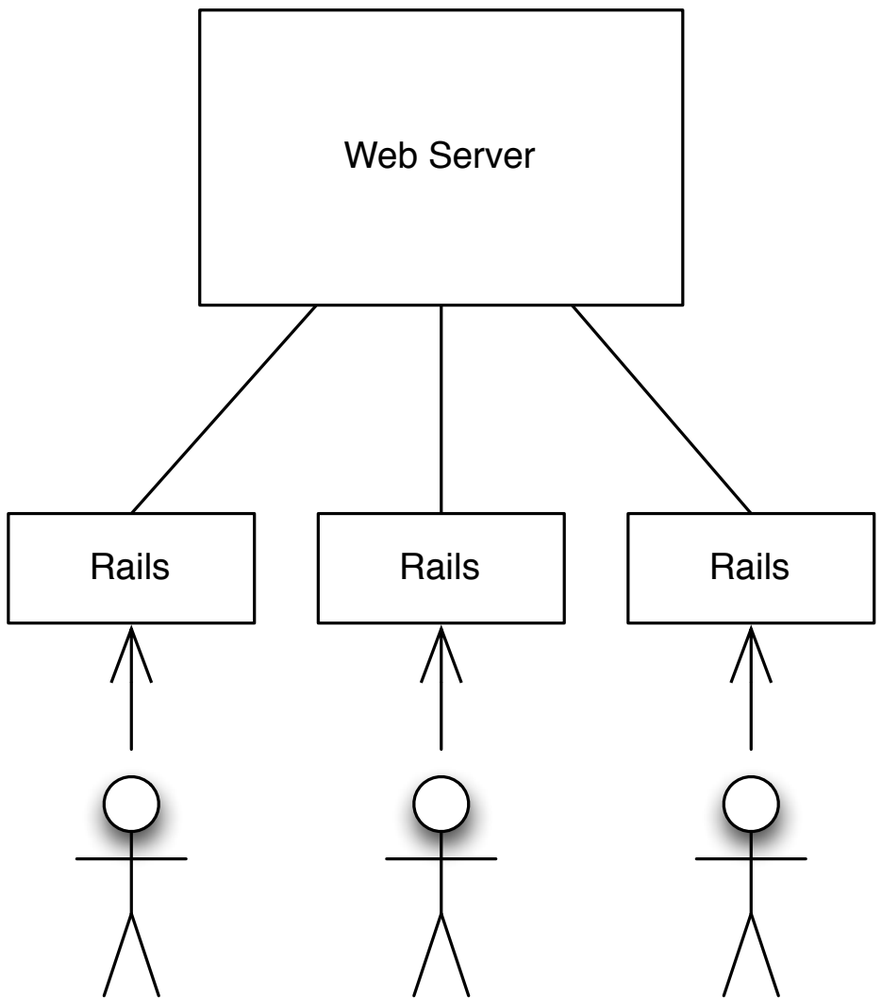


Green is executing thread

Red is waiting on I/O

From the CPU's perspective, when we have only one Rails process, we have to wait for a request to finish before we start the next request. All the time that's spent waiting for I/O is wasted.

Rails Concurrency



- Process concurrency
- Lots of memory

The way we handle multiple incoming requests at the same time today is to spawn multiple rails processes and have each of them handle one request a time. For an incoming request, if one process is busy, then the web server will know to route the request to a process that's free.

- The problem with this process concurrency model is that each of these large rails processes take up a lot of memory, and each of them can only do one request at a time.

- fork in linux is very efficient and copy-on-write should help w/ this, but the way ruby handles GC right now makes this wasteful. The next GC implementation should help with this.

Node Concurrency

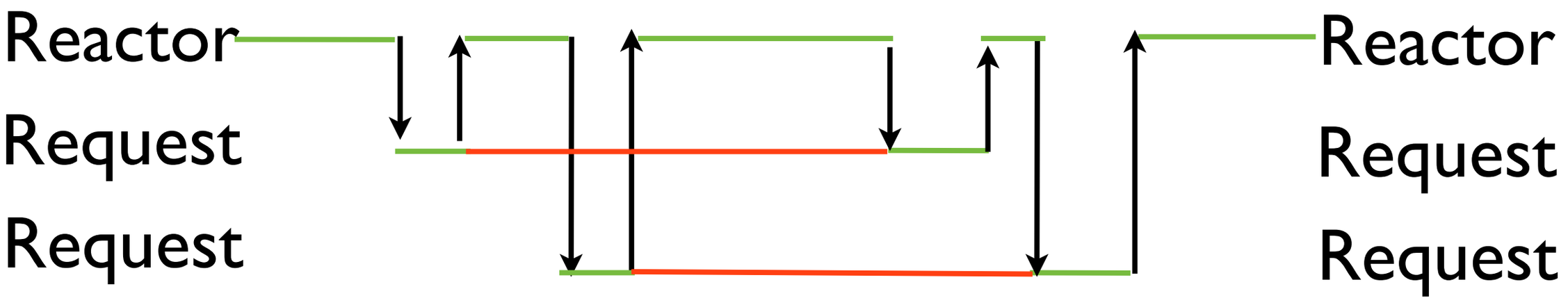
```
tweet = new Tweet();                          // 1.
tweet.shortenLinks(function(tweet) {          // 2. callback
  tweet.save(function(tweet) {                // 3. callback

  }
})
```


If we write this same code with node, the main difference is that all the blocking calls are registered as callback functions.

So instead of waiting for bitly to return a response before continuing, we just register a callback for when the response is ready, and instead of waiting for disk to finish, again, we register a callback for when the data is actually persisted.

Node Concurrency



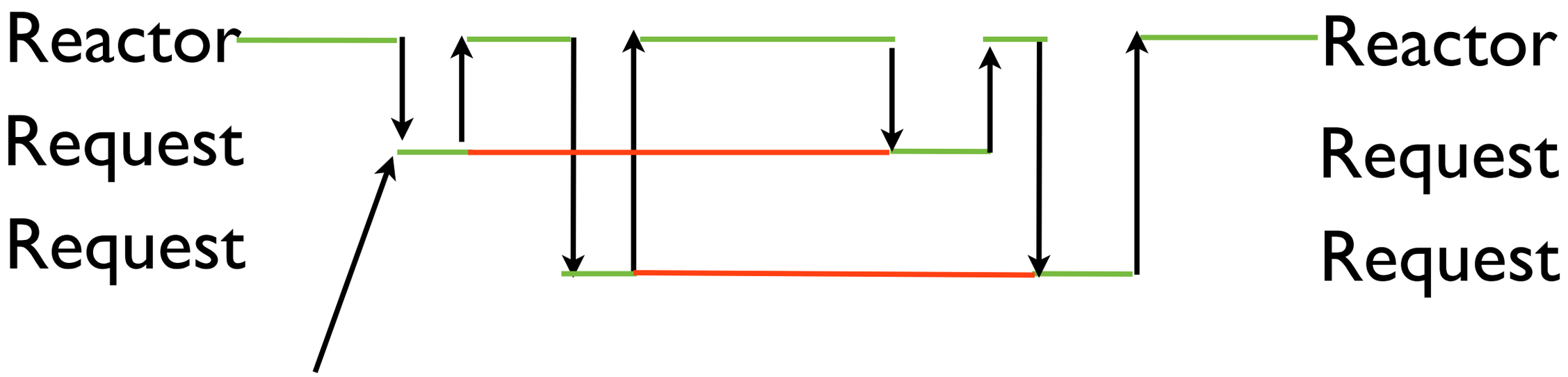
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



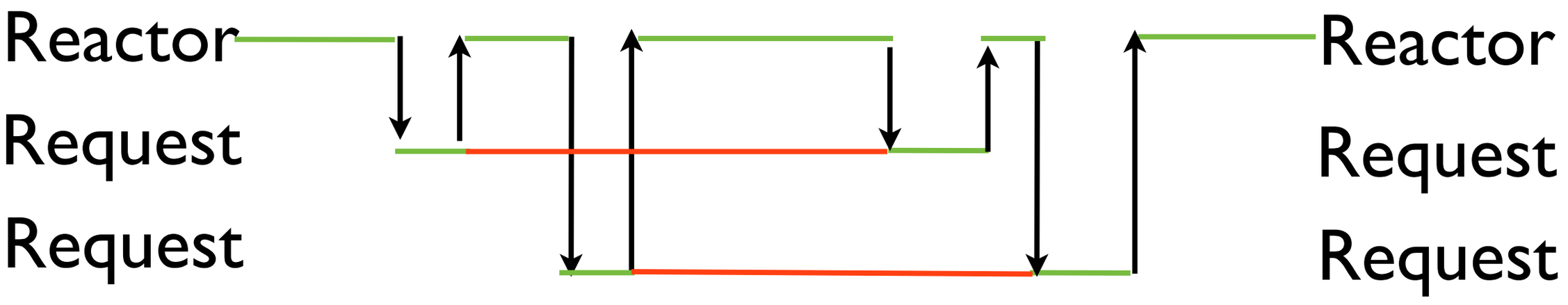
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



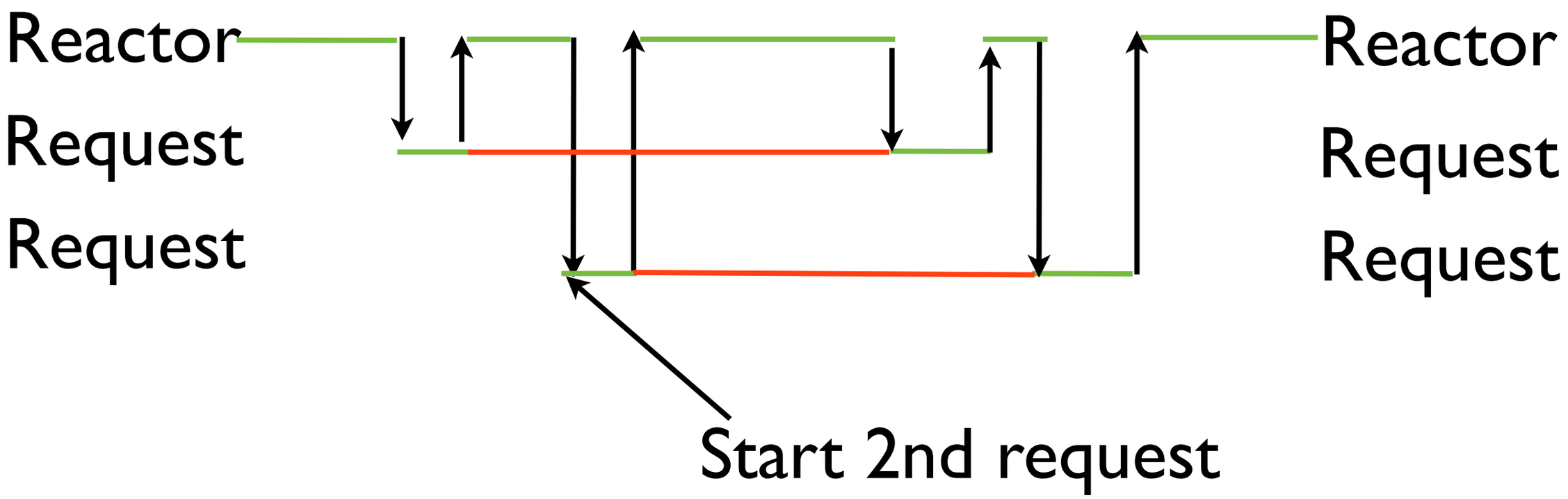
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



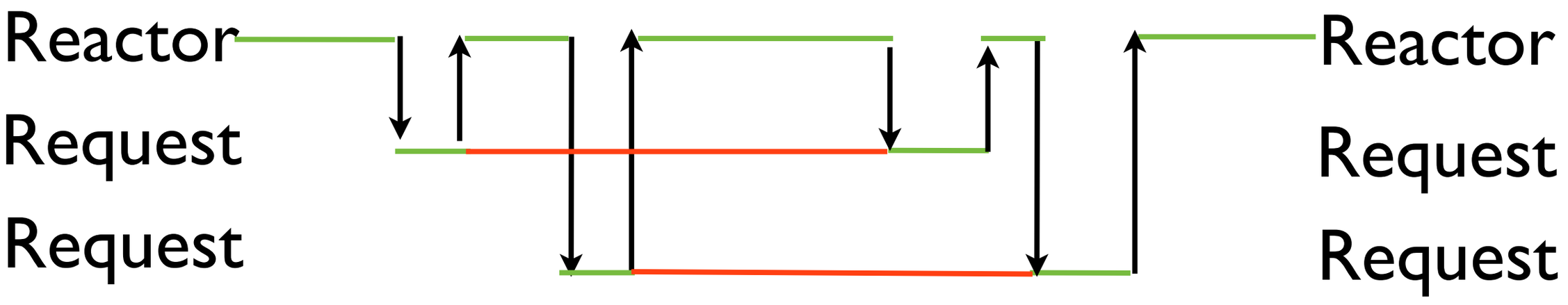
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



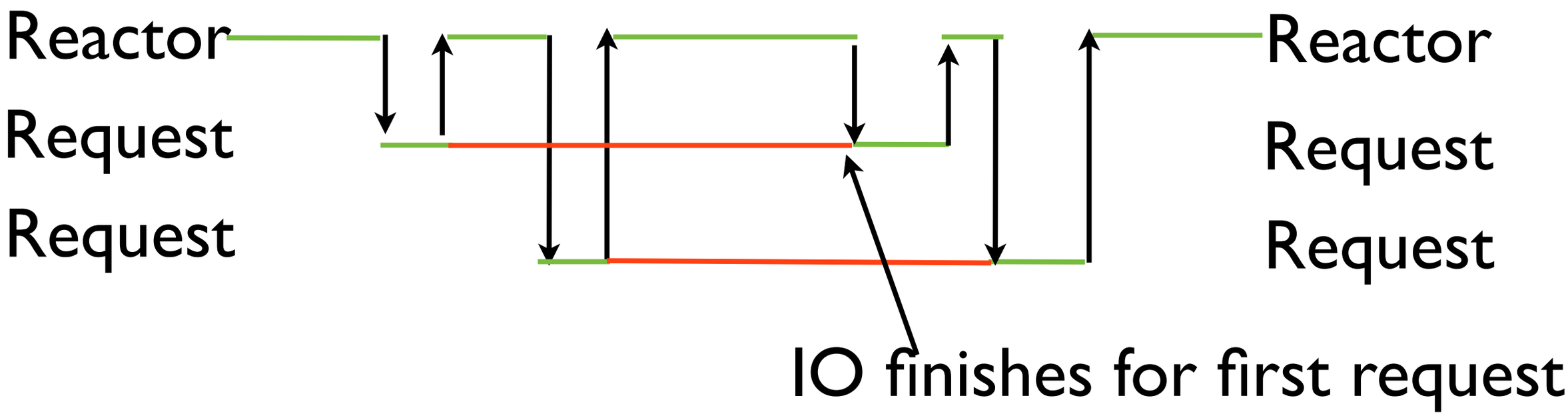
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



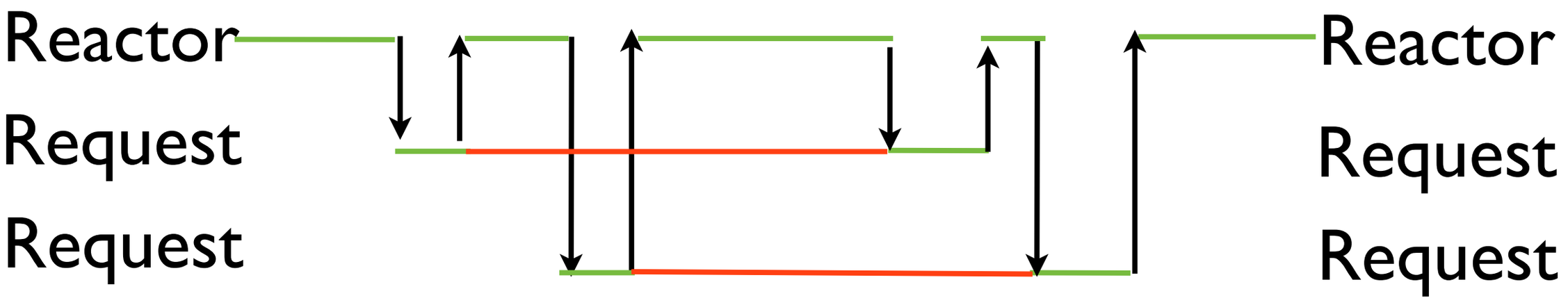
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



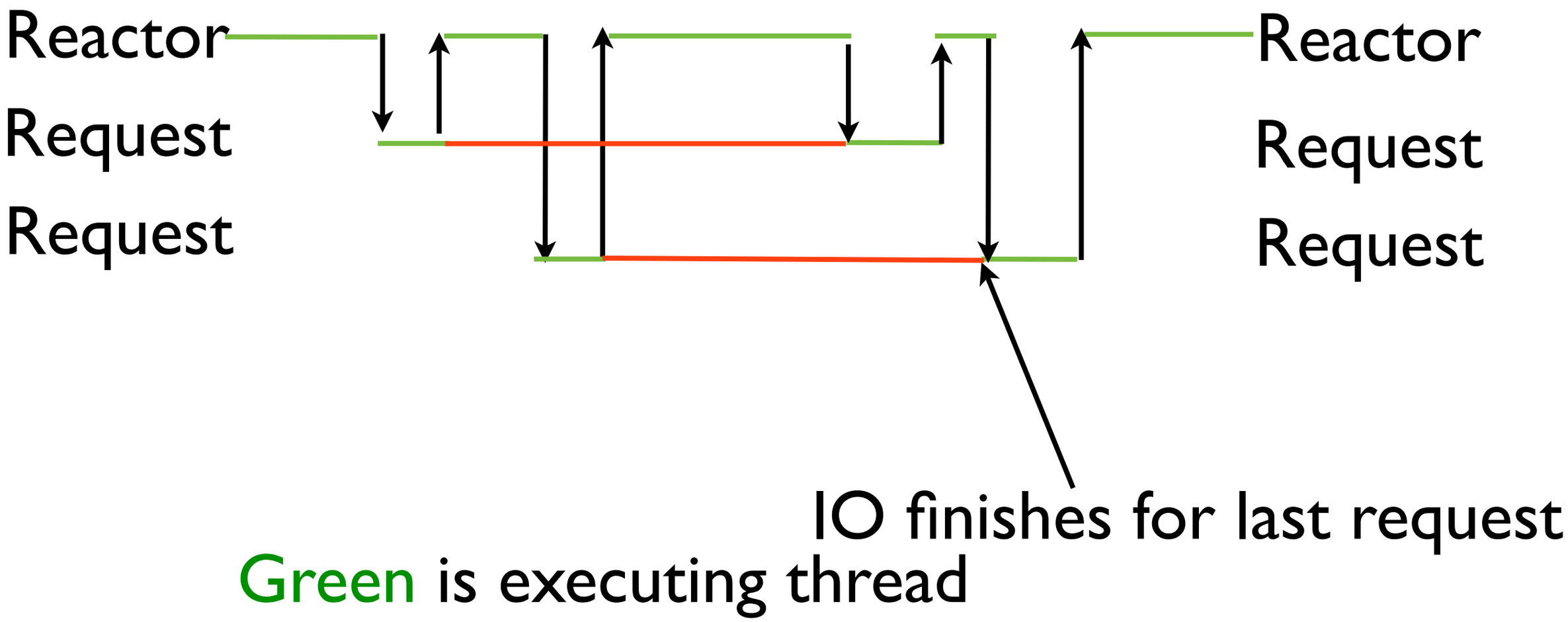
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

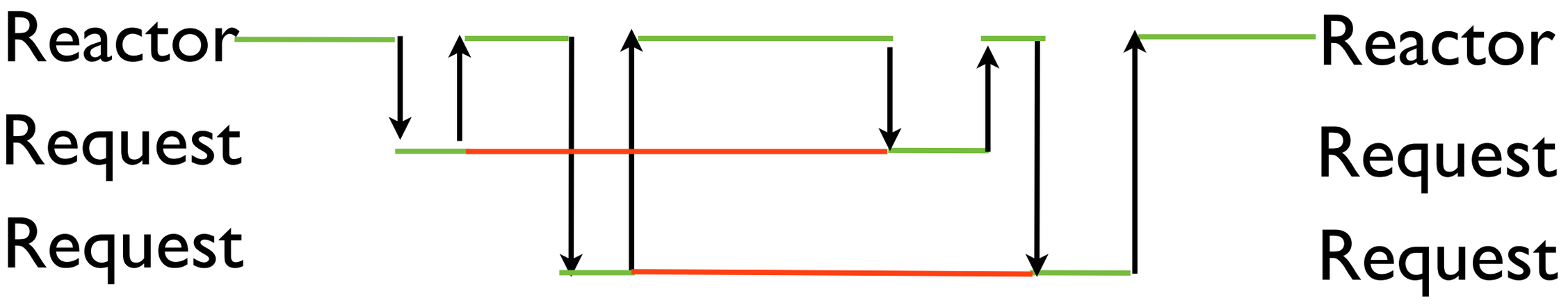
Node Concurrency



What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



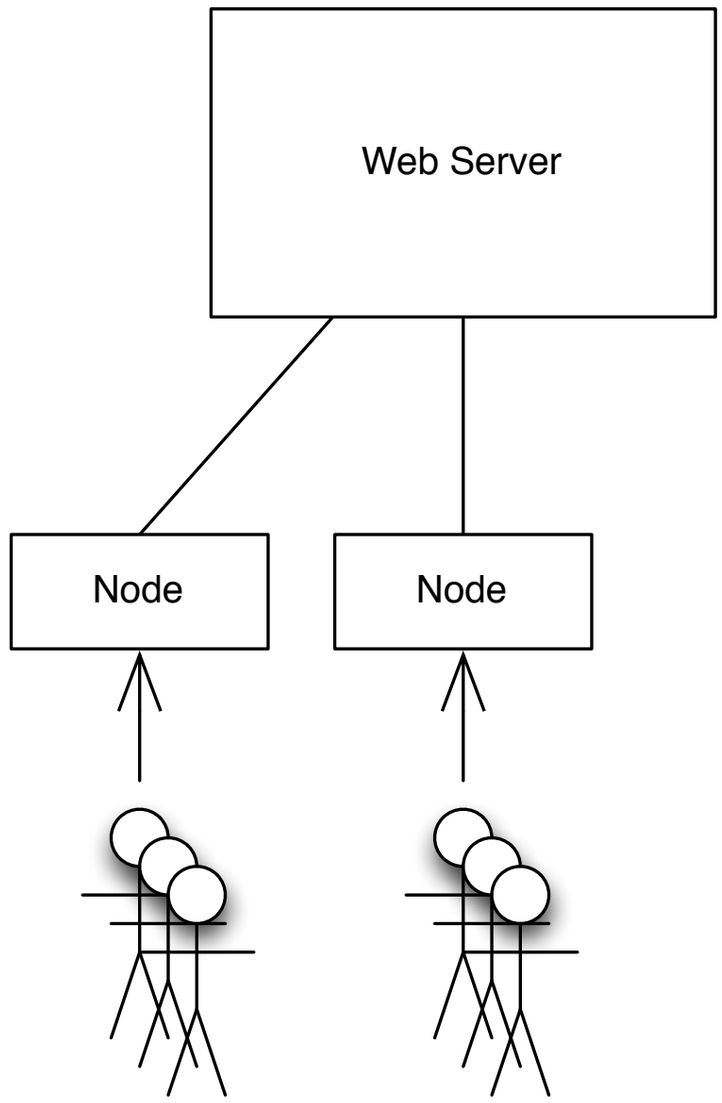
Green is executing thread

Red is waiting on I/O

What this actually looks like

- When we start node, we're already running in a reactor (that's the first green line on top)
- When a request comes in, we start doing some processing, but we return right after we've registered our callbacks. So we're not blocking our reactor. Meanwhile, the blocking I/O in red starts happening.
- Then another request comes in, again we do just enough to **start** the blocking I/O by registering a callback, but then go back to the reactor instead of waiting for the I/O to finish
- Notice that even though we're able to start a new request while blocking I/O is going on, only one thing is running at a time. So these two requests aren't actually really running concurrently at the same time, we're just switching quickly between them when the CPU is not busy.

Node Concurrency



- Reactor switches between requests
- Fewer processes needed => Less memory

With node, because the reactor can switch between requests, each reactor process we start will able to handle more requests. And by starting fewer processes, we'll need less memory, which translates to a smaller EC2 instance and a cheaper hosting bill.

Latency

Ruby

SSSFSSSSSSSSSFSSSS

JS

SSSFSSSSSSSSSFSSSS
SSSFSSSSSSSSSFSSSS
SSSFSSSSSSSSSFSSSS
SSSFSSSSSSSSSFSSSS

Single request takes the same time

- Blocking I/O is **not** sped up
- Optimize response latency first

An important thing to note is the distinction between latency and concurrency.

Even though non-evented ruby code can only do 1 request at a time, evented JS doesn't speed up the latency of your request.

Code Smell

```
tweet = new Tweet();                        // 1.
tweet.shortenLinks(function(tweet) {        // 2. callback
  tweet.save(function(tweet) {              // 3. callback

  }
})
```

- App code aware of blocking I/O
- Ugly syntax, nested contexts

The node version does handle more concurrency than the non-evented ruby version, but there's a tradeoff for this.

- unreasonable that application code is aware of a low-level detail like blocking IO, our OS already solves that problem for us. - complexity for optimization

- As a side effect, we get this callback spaghetti completely ruins the readability and intention of the code.

Evented Ruby

- ruby is capable of evented programming
- multi-paradigm: procedural, evented, parallel
- mix and match paradigms

So we saw that node gives us better concurrency in the backend than Rails, but with the caveat that the application code is aware of blocking IO.

The question we want answered now is whether we can get the same concurrency benefits with Ruby, and how the drawbacks compare with node.

Ruby is capable of evented programming, and works a general purpose and general paradigm language. For the most part, we think of ruby as a procedural language, but it can also be used within a reactor for evented programming, and for paralllel programming with threads.

One of the main benefit, but also the main drawback is the ability to switch between these paradigms

Ruby Reactors

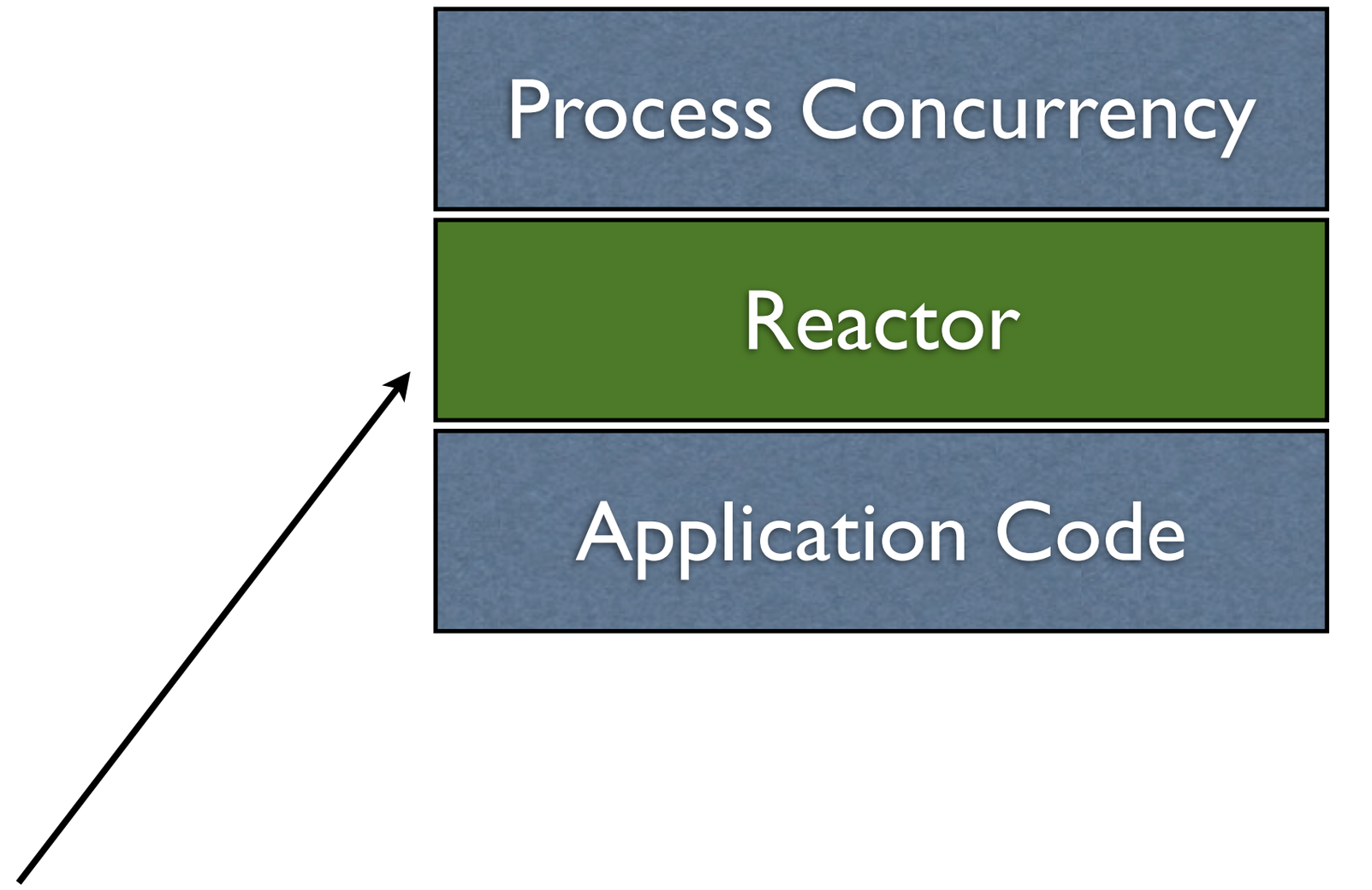
```
# Starting an EventMachine reactor
EM.run {
  # reactor aware code here...
}
```

- reactor is just a gem
- eventmachine, cool.io, others

In order for us to register for callbacks for events, we first need a reactor to deliver events for us. In node, whenever you're running code, you're running inside a reactor. But since ruby lets your write evented code next to non-evented code, you have to explicitly start a reactor.

In ruby, a reactor is just a gem, and there are multiple choices available.

Add a Reactor



Thin, Rainbows!, Passenger (w/ patch)

You can manually start a reactor with EM.start, but there are already app servers out there that run application code within an eventmachine reactor.

Add Events

```
http = EM::HttpRequest.new('http://railsconf2012.com/').get
http.callback {
  # request finished
}
```


Now we have our application code running inside a reactor. But the reactor is only half of the equation. In order for our reactor to have something to do, we need to register events and event callbacks.

This code is an example how to get a web page with a reactor aware http library. First you build a request object, then you register a callback for when the request is finished.

In the eventmachine reactor loop, even though the HTTP call would take a long time to return, eventmachine will be able to start other tasks and dispatch events because all of the blocking code has been encapsulated into callback blocks.

Code Smell

```ruby
http = EM::HttpRequest.new('http://railsconf2012.com/').get
http.callback {
  # request finished
}
```

- app code aware of blocking I/O
- code doesn't look like ruby

But this ruby version has the same code smells as the javascript one. Suddenly our application code is aware of blocking I/O.

The code smells I'm talking about can happen with any evented code. It doesn't matter if it's ruby, java, python or javascript. The real break in abstraction is having app code aware of blocking I/O.

- What's even worse about this code is that it looks nothing like ruby
- The syntax is valid ruby

- in procedural ruby, blocks usually yield a value
- in evented ruby, blocks are used to register future callbacks

Procedural Interface, Evented Execution

```
response = Faraday.get 'http://railsconf2012.com/'
```

Thursday, April 26, 12

- Get back to ruby-esque syntax. What would that look like?

This is what we'd ideally want to work with. This looks like ruby, and if I showed this to you, chances are good you'd know what to do with it even if you weren't familiar with the Faraday library.

Fortunately, with a bit of extra work, ruby lets us hide blocking I/O related callbacks into libraries. Faraday is an http client that lets you switch between different adapters. The default adapter is net-http, but there's also an adapter that's reactor aware.

<build in>

Just by switching to the reactor aware adapter, we're able to keep a procedural interface to our code, but use evented IO behind the scenes.

This keep our app code looking clean and like ruby.

Procedural Interface, Evented Execution

```
Faraday.default_adapter = :em_synchrony
response = Faraday.get 'http://railsconf2012.com/'
```

Thursday, April 26, 12

- Get back to ruby-esque syntax. What would that look like?

This is what we'd ideally want to work with. This looks like ruby, and if I showed this to you, chances are good you'd know what to do with it even if you weren't familiar with the Faraday library.

Fortunately, with a bit of extra work, ruby lets us hide blocking I/O related callbacks into libraries. Faraday is an http client that lets you switch between different adapters. The default adapter is net-http, but there's also an adapter that's reactor aware.

<build in>

Just by switching to the reactor aware adapter, we're able to keep a procedural interface to our code, but use evented IO behind the scenes.

This keep our app code looking clean and like ruby.

Procedural Interface, Evented Execution

```
Faraday.default_adapter = :em_synchrony
response = Faraday.get 'http://railsconf2012.com/'
```

- hides system event callbacks in libraries
- keeps app code clean

- Get back to ruby-esque syntax. What would that look like?

This is what we'd ideally want to work with. This looks like ruby, and if I showed this to you, chances are good you'd know what to do with it even if you weren't familiar with the Faraday library.

Fortunately, with a bit of extra work, ruby lets us hide blocking I/O related callbacks into libraries. Faraday is an http client that lets you switch between different adapters. The default adapter is net-http, but there's also an adapter that's reactor aware.

<build in>

Just by switching to the reactor aware adapter, we're able to keep a procedural interface to our code, but use evented IO behind the scenes.

This keep our app code looking clean and like ruby.

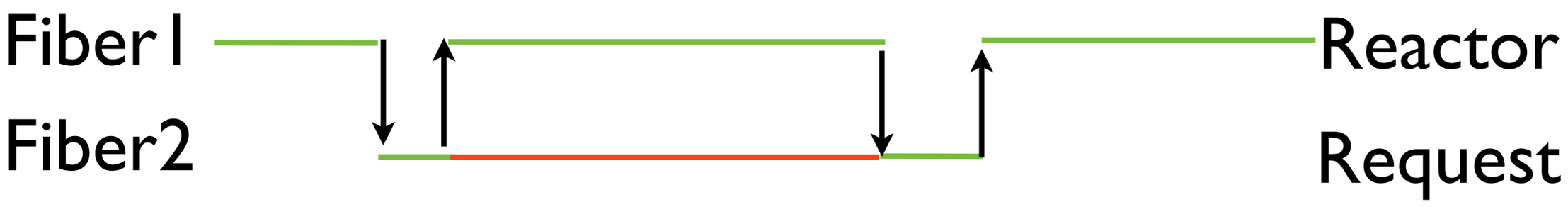
Procedural Interface, Evented Execution

Fibers are primitives for implementing light weight **cooperative concurrency** in Ruby. Basically they are a means of creating code blocks that can be paused and resumed, much like threads. The main difference is that they are never preempted and that the **scheduling must be done by the programmer** and not the VM.

The way we actually hide evented I/O is by using ruby's Fiber object.

http://www.ruby-doc.org/core-1.9.3/Fiber.html

Procedural Interface, Evented Execution



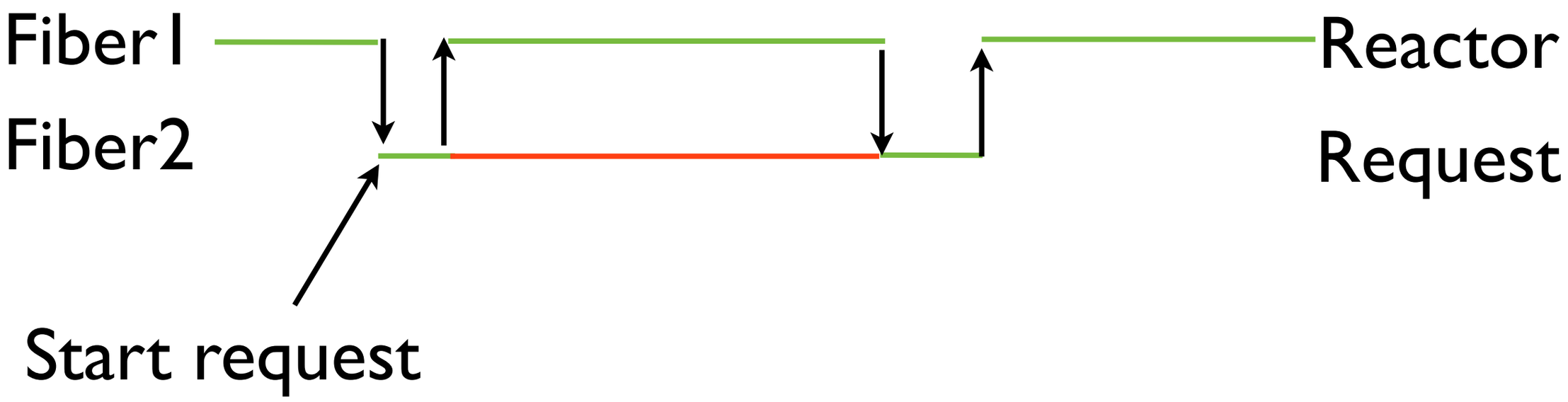
- Fiber is a coroutine, which means they we have to manually control which fiber is running. There's also ever only one running fiber.

- We run the reactor in one fiber, and when we're about to start our GET request, we **create** a new fiber, and start running it after we **pause** the reactor fiber.

- Fibers are **not** preempted, which means that we have to explicitly yield control when we know something will block. When we yield this request fiber, we have to yield control back to the reactor. It's this ability to yield control back to the reactor that lets the reactor dispatch other events to start other tasks.

- When the reactor says our data is ready, we resume our request fiber, and finish processing.

Procedural Interface, Evented Execution



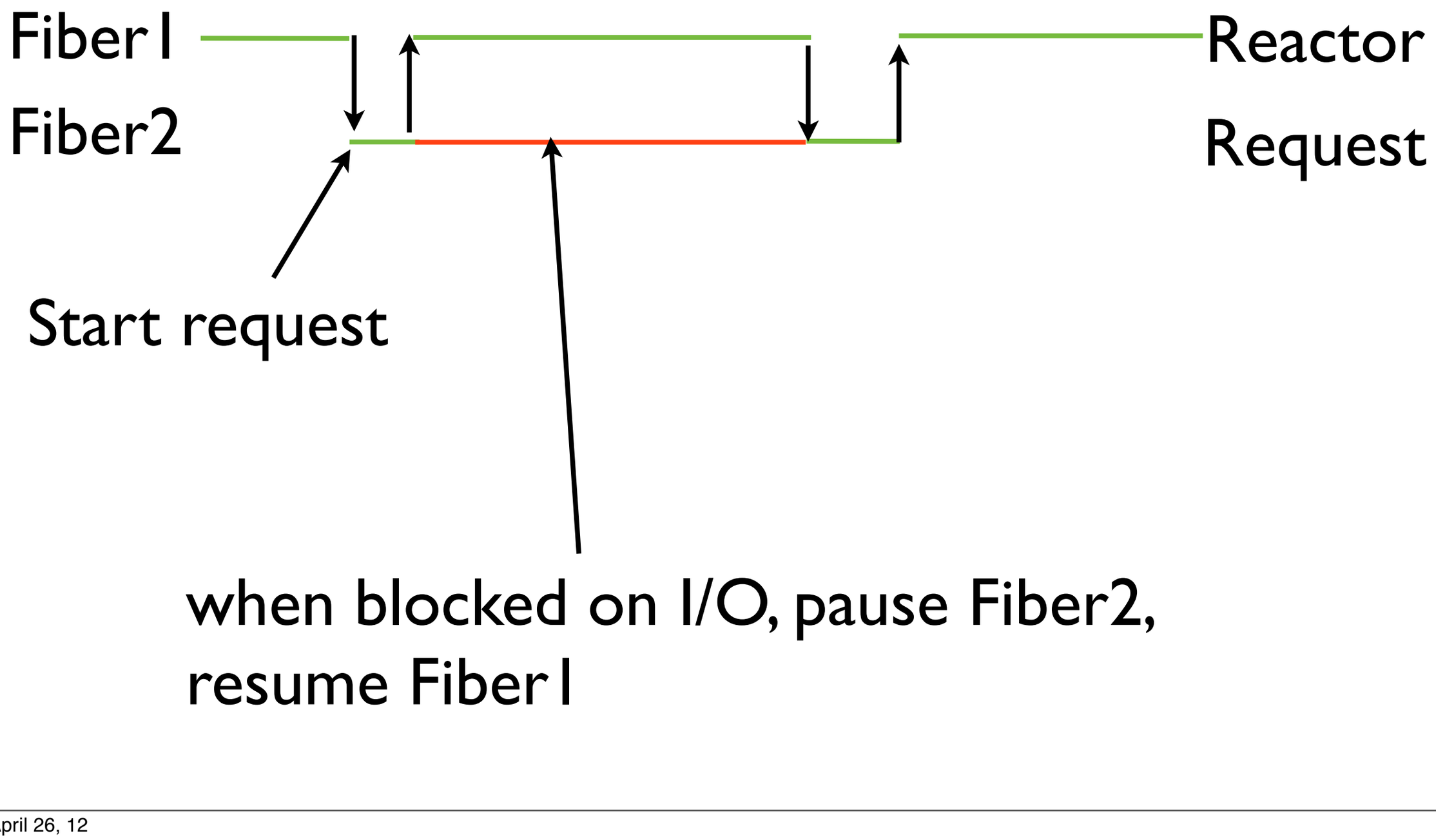
- Fiber is a coroutine, which means they we have to manually control which fiber is running. There's also ever only one running fiber.

- We run the reactor in one fiber, and when we're about to start our GET request, we **create** a new fiber, and start running it after we **pause** the reactor fiber.

- Fibers are **not** preempted, which means that we have to explicitly yield control when we know something will block. When we yield this request fiber, we have to yield control back to the reactor. It's this ability to yield control back to the reactor that lets the reactor dispatch other events to start other tasks.

- When the reactor says our data is ready, we resume our request fiber, and finish processing.

Procedural Interface, Evented Execution



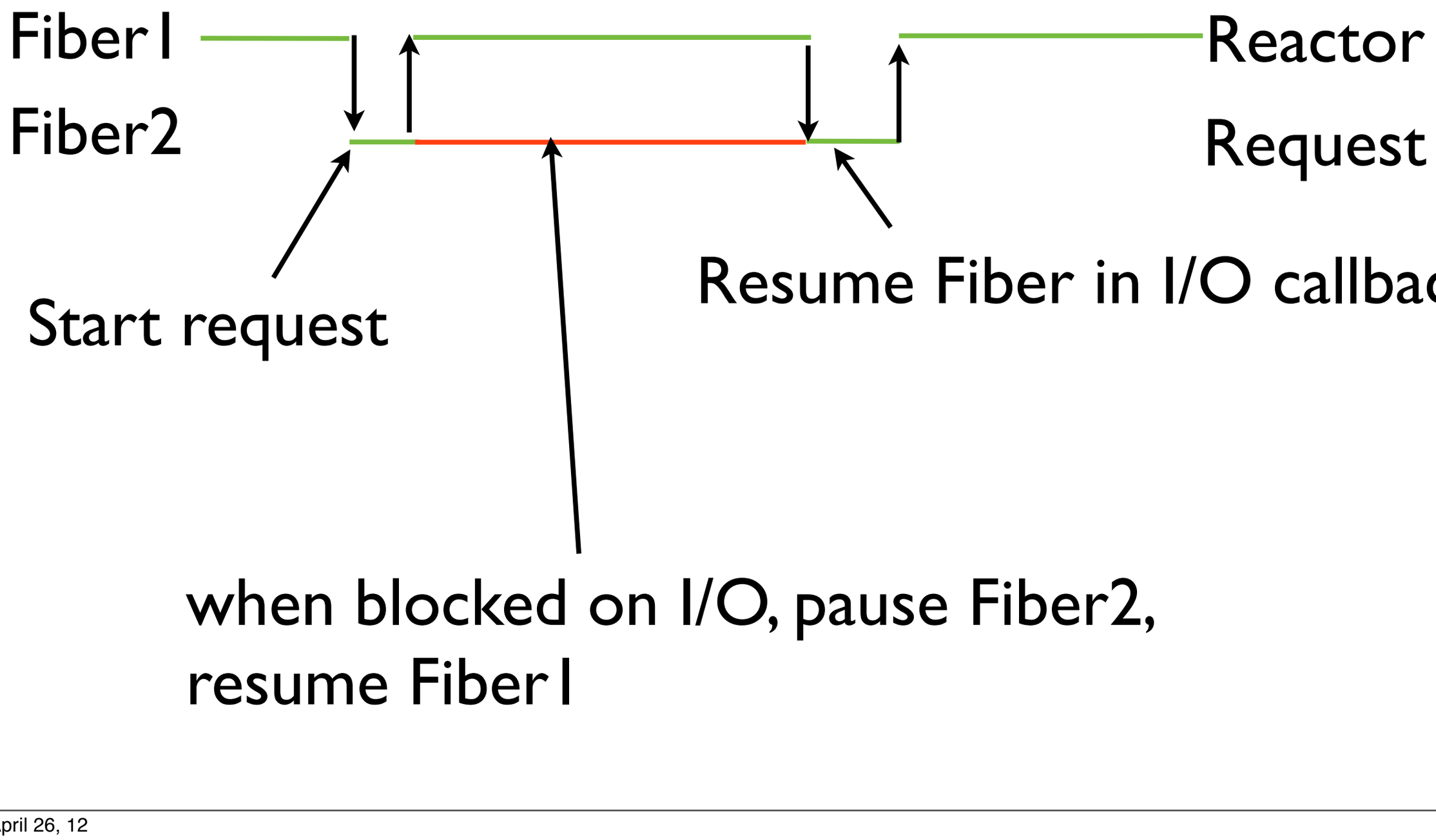
- Fiber is a coroutine, which means they we have to manually control which fiber is running. There's also ever only one running fiber.

- We run the reactor in one fiber, and when we're about to start our GET request, we **create** a new fiber, and start running it after we **pause** the reactor fiber.

- Fibers are **not** preempted, which means that we have to explicitly yield control when we know something will block. When we yield this request fiber, we have to yield control back to the reactor. It's this ability to yield control back to the reactor that lets the reactor dispatch other events to start other tasks.

- When the reactor says our data is ready, we resume our request fiber, and finish processing.

Procedural Interface, Evented Execution



- Fiber is a coroutine, which means they we have to manually control which fiber is running. There's also ever only one running fiber.

- We run the reactor in one fiber, and when we're about to start our GET request, we **create** a new fiber, and start running it after we **pause** the reactor fiber.

- Fibers are **not** preempted, which means that we have to explicitly yield control when we know something will block. When we yield this request fiber, we have to yield control back to the reactor. It's this ability to yield control back to the reactor that lets the reactor dispatch other events to start other tasks.

- When the reactor says our data is ready, we resume our request fiber, and finish processing.

Code Smell?

...The main difference is that they are never preempted and that the **scheduling must be done by the programmer** and not the VM.

But wait a second, isn't this still the same code smell? Aren't we still saying that we're manually taking control of scheduling I/O?

Yes, it is still introducing more complexity, but we have more choice in how we structure our application code and where we add the complexity. In our Tweet example, our domain model is Tweets. Tweets shouldn't know anything about I/O, but the database adapter that we use does deal with I/O. So by making our database adapter reactor-aware, but still use a procedural interface, our domain models can stay clean.

Evented Interface

```
redis.subscribe('channel') do |on|
  on.message do |channel, msg|
    # happens in the future
  end
end
```

- only use blocks for semantic events
- hide system events in library code

When we're talking about blocking IO, I think it's best that we hide the event behind a synchronous interface so we can keep our domain abstraction clean. But just because we can use fibers to hide our evented plumbing, it doesn't make sense for all events to be hidden behind a synchronous interface.

So for example, even though this code snippet is using blocks as a way to register for publish events in redis, we generally don't want to try and hide this because the event is the subject of what we're talking about.

One Request per Fiber

- wrap each request in it's own fiber
- web requests are independent from one another
- switch between requests instead of processes

We have a reactor in one fiber, but we still need to have requests in their own fibers so we can pause and resume different fibers.

The web makes this extra easy because web requests are naturally independent of one another.

Once we have one request per fiber, the our reactor app server can switch between fibers when there's blocking I/O, rather than switching to a different process

Rack::FiberPool

```ruby
# Rails
config.middleware.prepend Rack::FiberPool


# Generic rack apps: sinatra, grape, etc
use Rack::FiberPool
```


Because Rails is rack, it's very easy to wrap each request with a fiber. There's a gem called Rack::FiberPool that does it for you. We basically add it to the top of Rail's middleware stack, and all incoming requests will be wrapped in it's own fiber.

And since it's rack, adding it for any rack application is also easy. So you can also add this for sinatra, or other ruby web frameworks

Recap

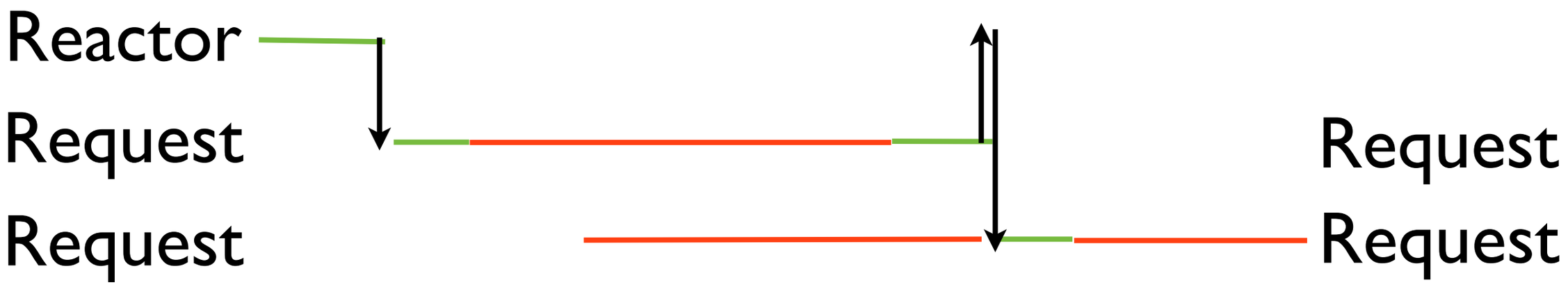
- App server is reactor-aware
- One fiber per request
- App code is unchanged

The only infrastructure change we've made is change an app server. But there's good odds that you're already using an app server that's reactor-aware

The only code change we've done is to configure a middleware that wraps each request in it's own fiber. We haven't touched any of our models, controllers, or views. All of that continues to work like it used to.

But if you try and benchmark your application at this point, your app isn't going to feel any faster.

Mixing Paradigms



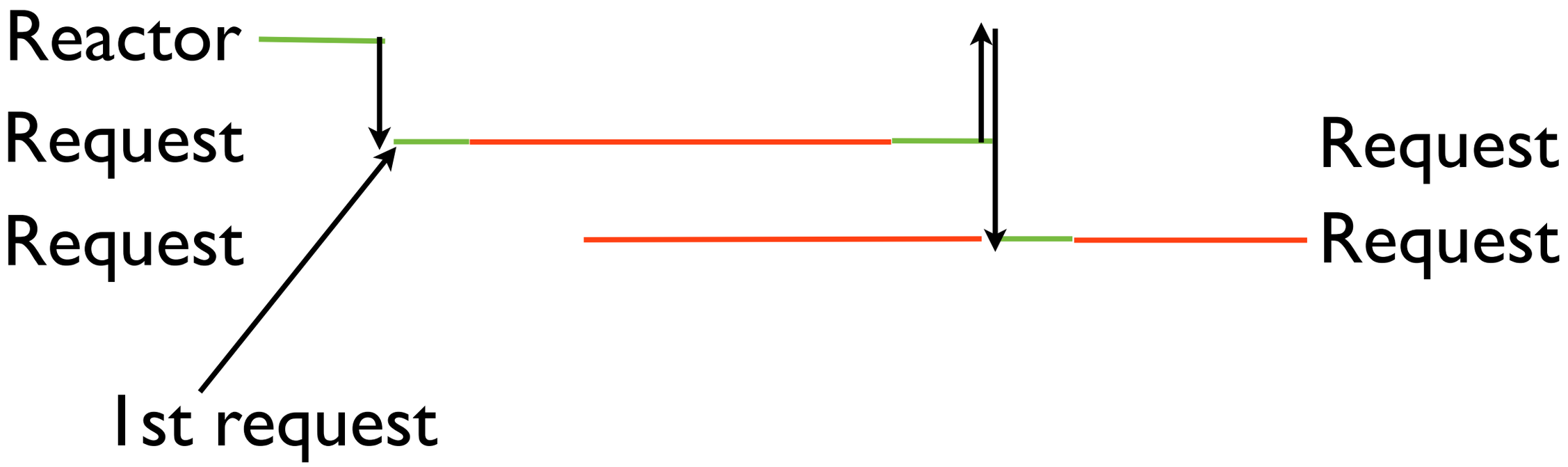
- libraries may block reactor

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



- libraries may block reactor

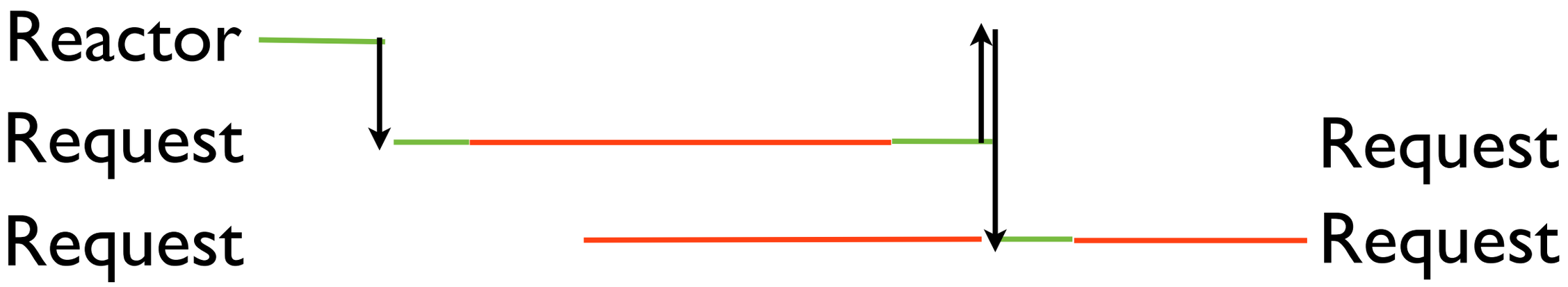
Thursday, April 26, 12

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



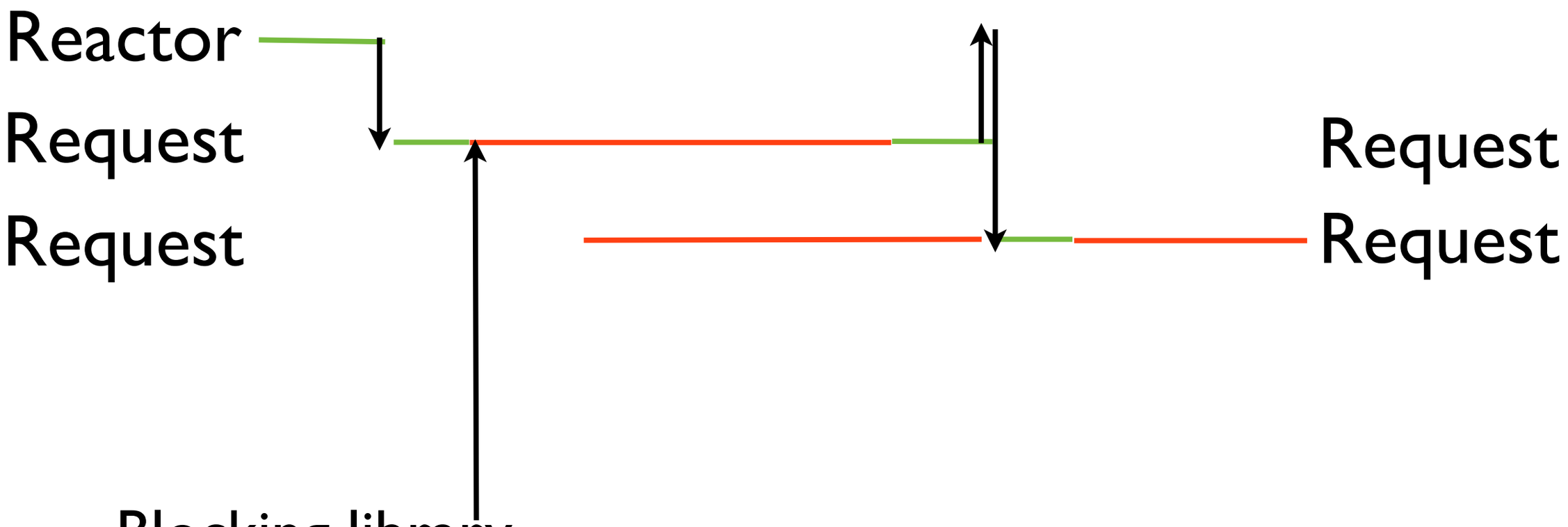
- libraries may block reactor

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



- libraries may block reactor

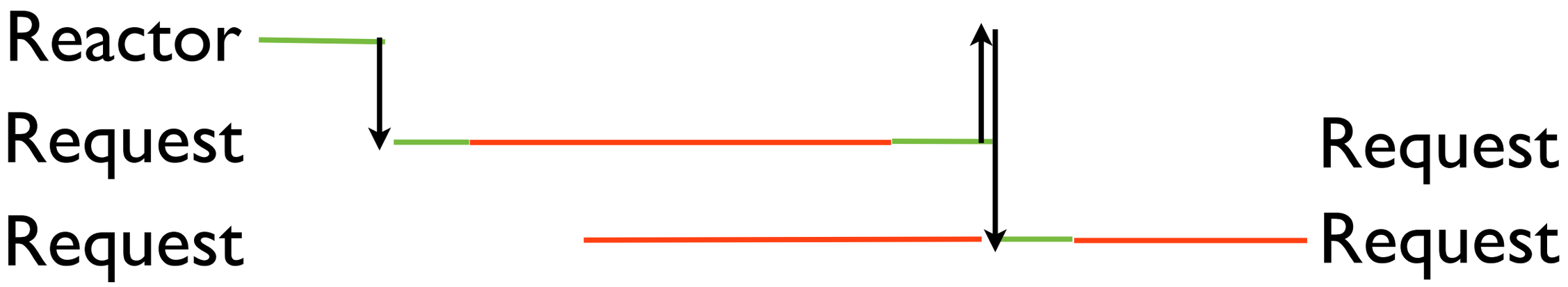
Thursday, April 26, 12

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



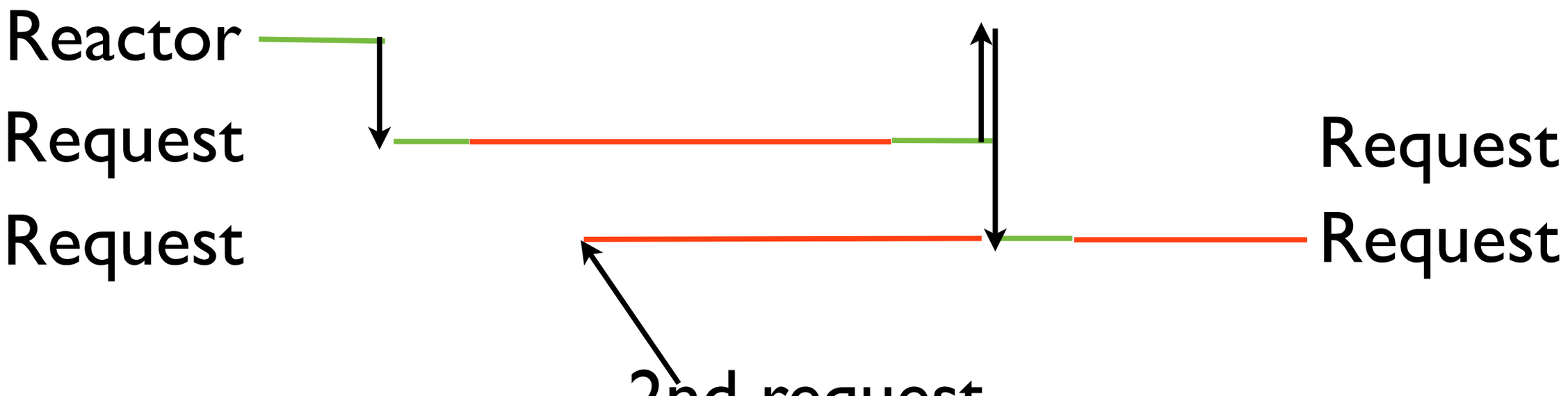
- libraries may block reactor

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



- libraries may block reactor

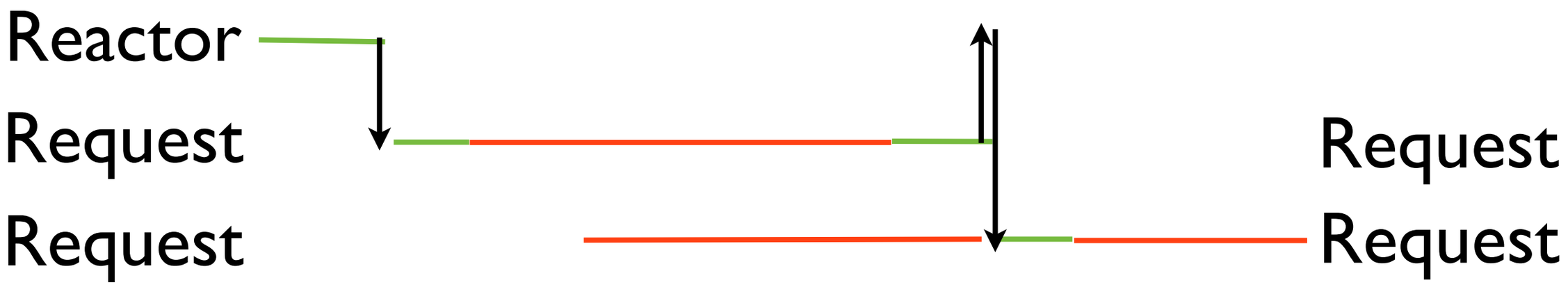
Thursday, April 26, 12

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



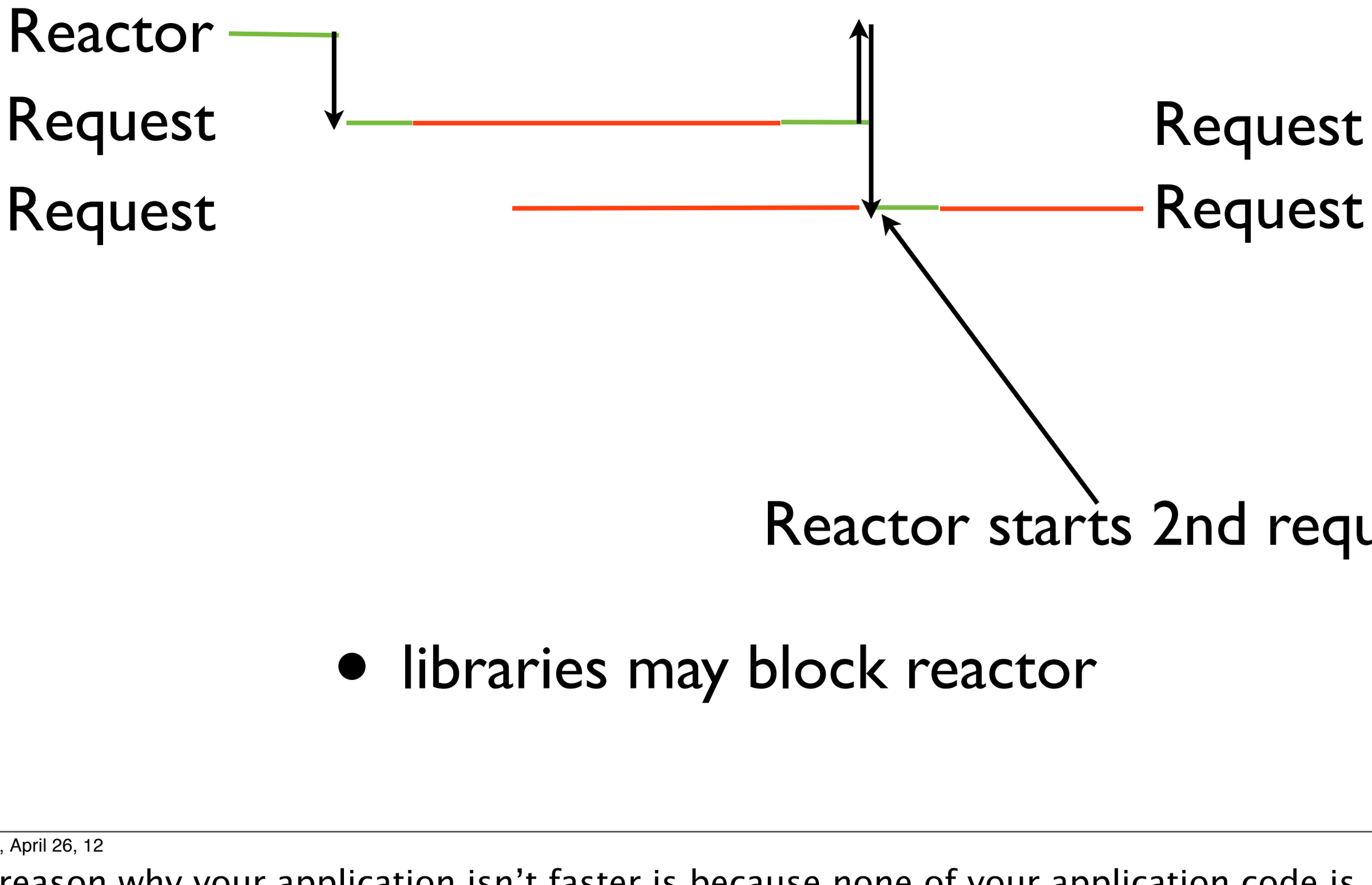
- libraries may block reactor

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

Mixing Paradigms



- libraries may block reactor

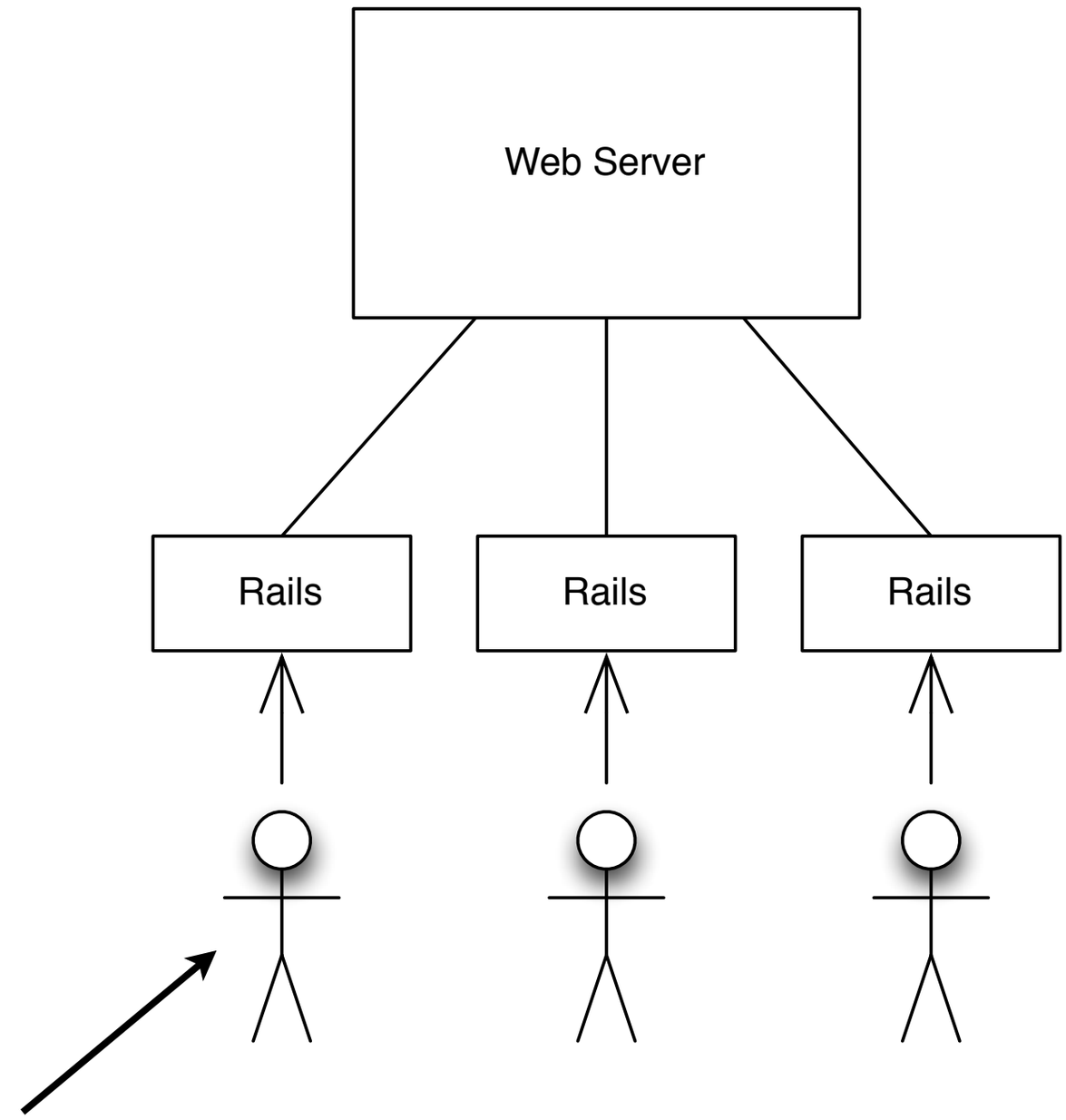
Thursday, April 26, 12

The reason why your application isn't faster is because none of your application code is reactor aware.

Even if each request is running in a fiber of it's own, the request fiber is not yielding control back to the reactor until it finishes running. Remember that fibers are user controlled, so unless something explicitly yields control, it's just going to keep running. And since there's only one fiber running at a time, if we don't give up control when we're handling a request, then the reactor can't do anything.

<build in>

But that's ok...



Can't do worse than one request per process

- This goes back to my 2nd slide about lazy optimizations. Since you're at this talk, chances are good that your production app is **not** crashing and burning under load at the moment.

- Even if we block the reactor, process concurrency still works.

- Not the most efficient, but we can only improve from here.

Starting Points

- data stores
- http
- `Kernel.system` calls

So a few broad areas for making libraries reactor aware are data stores, external http, and system calls

Data Stores

- redis
- mysql
- postgresql
- mongo

- some adapters will use EM if it's available. see https://github.com/eventmachine/eventmachine/wiki/Protocol-Implementations

HTTP

- Faraday: use an EM-aware http adapter

https://github.com/technoweenie/faraday

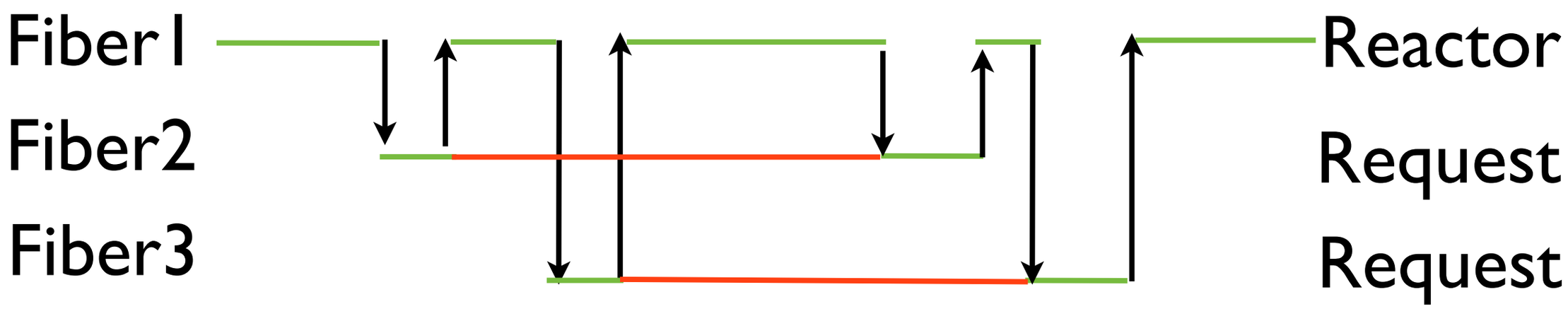
System Calls

- EM.popen - non-blocking version
- EM.defer - run blocking call outside of reactor thread

When you need to read in the output of a command from the shell, eventmachine provides it's own version of Kernel's 'popen' that is nonblocking.

Also, if you need to do a blocking chunk of code, but can't change it to be evented, you can run the chunk in a separate thread from the reactor thread so the reactor doesn't get blocked. Then when the chunk finishes, the result is fed back into the reactor.

Final Result



Green is executing thread

Red is waiting on I/O

After we've tweaked our libraries to be reactor aware, then our execution pattern starts to look more like node's. Each request is in it's own fiber. Fibers know to yield control when they're about to block. And the reactor can start new requests in new fibers.

Why Ruby

- Reuse existing code
- Performance won't be worse
- Keep procedural syntax
- Multi-paradigm

If you do spend some time trying to add evented I/O to your Rails apps, you'll get some very clear benefits.

- You get to reuse your existing code: models, controllers, views.
- You won't have worse concurrency than you started. The performance increase you get will depend on how many of your dependencies you can convert or tweak to be reactor aware.
- You'll get to keep your ruby code looking like ruby code, and have a much more readable codebase.
- Also, evented programming isn't the only way to increase concurrency. With ruby, you can also explore other programming paradigms that might fit your needs even better.

Why Node

- Single paradigm consistency
- Community

- the same multi-paradigm benefit in ruby is also a drawback depending on how you look at it. In node, all libraries are evented, so you have more options, and you don't have to hunt down things that are blocking.
- the ruby evented community is much less active than node's. There'll be more support, and more people to help out.

Wrapping Up

Evented programming is a hard topic. Ideally, we want to use evented programming for problems that fit that paradigm.

But if we do want to use evented IO for getting getting better concurrency, you can't expect to drop in a gem and suddenly see a big improvement. You need to actually figure out the things that are blocking and make them reactor aware. Usually this means finding a configuration for your adapter and turning it on.

Before even going down this route of evented I/O, I recommend improving your response latency first because evented programming won't make your slow responses run any faster. And if your responses are faster, then your app server will be able to serve more requests anyways.

The most important thing is to make sure your domain isn't polluted by I/O callbacks. The only places where I/O callbacks should live are libraries that directly have to deal with I/O. But when your business logic uses these libraries, they shouldn't have to know the internal

Wrapping Up

- Evented programming is hard in any language

Evented programming is a hard topic. Ideally, we want to use evented programming for problems that fit that paradigm.

But if we do want to use evented IO for getting getting better concurrency, you can't expect to drop in a gem and suddenly see a big improvement. You need to actually figure out the things that are blocking and make them reactor aware. Usually this means finding a configuration for your adapter and turning it on.

Before even going down this route of evented I/O, I recommend improving your response latency first because evented programming won't make your slow responses run any faster. And if your responses are faster, then your app server will be able to serve more requests anyways.

The most important thing is to make sure your domain isn't polluted by I/O callbacks. The only places where I/O callbacks should live are libraries that directly have to deal with I/O. But when your business logic uses these libraries, they shouldn't have to know the internal

Wrapping Up

- Evented programming is hard in any language
- Evented programming for evented problems

Evented programming is a hard topic. Ideally, we want to use evented programming for problems that fit that paradigm.

But if we do want to use evented IO for getting getting better concurrency, you can't expect to drop in a gem and suddenly see a big improvement. You need to actually figure out the things that are blocking and make them reactor aware. Usually this means finding a configuration for your adapter and turning it on.

Before even going down this route of evented I/O, I recommend improving your response latency first because evented programming won't make your slow responses run any faster. And if your responses are faster, then your app server will be able to serve more requests anyways.

The most important thing is to make sure your domain isn't polluted by I/O callbacks. The only places where I/O callbacks should live are libraries that directly have to deal with I/O. But when your business logic uses these libraries, they shouldn't have to know the internal

Wrapping Up

- Evented programming is hard in any language
- Evented programming for evented problems
- Evented programming doesn't fix latency

Evented programming is a hard topic. Ideally, we want to use evented programming for problems that fit that paradigm.

But if we do want to use evented IO for getting getting better concurrency, you can't expect to drop in a gem and suddenly see a big improvement. You need to actually figure out the things that are blocking and make them reactor aware. Usually this means finding a configuration for your adapter and turning it on.

Before even going down this route of evented I/O, I recommend improving your response latency first because evented programming won't make your slow responses run any faster. And if your responses are faster, then your app server will be able to serve more requests anyways.

The most important thing is to make sure your domain isn't polluted by I/O callbacks. The only places where I/O callbacks should live are libraries that directly have to deal with I/O. But when your business logic uses these libraries, they shouldn't have to know the internal

Wrapping Up

- Evented programming is hard in any language
- Evented programming for evented problems
- Evented programming doesn't fix latency
- Avoid evented I/O interface in app code

Evented programming is a hard topic. Ideally, we want to use evented programming for problems that fit that paradigm.

But if we do want to use evented IO for getting getting better concurrency, you can't expect to drop in a gem and suddenly see a big improvement. You need to actually figure out the things that are blocking and make them reactor aware. Usually this means finding a configuration for your adapter and turning it on.

Before even going down this route of evented I/O, I recommend improving your response latency first because evented programming won't make your slow responses run any faster. And if your responses are faster, then your app server will be able to serve more requests anyways.

The most important thing is to make sure your domain isn't polluted by I/O callbacks. The only places where I/O callbacks should live are libraries that directly have to deal with I/O. But when your business logic uses these libraries, they shouldn't have to know the internal

Thanks!

http://github.com/jch/railsconf2012

@whatcodecraves