EventMachine

Nathan Witmer

boulder.rb · August 2011

?

Wednesday, August 17, 11

What is it?

node.js

Wednesday, August 17, 11

Compare to: node.js.

Not as cool

It’s not as cool

Ruby!

Wednesday, August 17, 11

But it’s ruby!

Older

Wednesday, August 17, 11

Since 2008! And the idea’s been around for a lot longer.

Single-threaded

Wednesday, August 17, 11

And, to be more specific, it's a single-threaded

Asynchronous

asynchronous (doesn’t block)

Event-driven

Wednesday, August 17, 11

event-driven...

IO Library

Wednesday, August 17, 11

library. and it’s best for any sort of IO.

(I'll explain)

I'll explain all that.

Wednesday, August 17, 11

I’m going to talk about a couple things. A little about sockets, a little about threads, a bit about non-blocking IO, and then I’ll show you EventMachine. Also, there are no more bulleted lists in this presentation.

- Sockets

I’m going to talk about a couple things. A little about sockets, a little about threads, a bit about non-blocking IO, and then I’ll show you EventMachine. Also, there are no more bulleted lists in this presentation.

- Sockets
- Threads

Wednesday, August 17, 11

I’m going to talk about a couple things. A little about sockets, a little about threads, a bit about non-blocking IO, and then I’ll show you EventMachine. Also, there are no more bulleted lists in this presentation.

- Sockets
- Threads
- Non-blocking IO

I’m going to talk about a couple things. A little about sockets, a little about threads, a bit about non-blocking IO, and then I’ll show you EventMachine. Also, there are no more bulleted lists in this presentation.

- Sockets
- Threads
- Non-blocking IO
- EventMachine

I’m going to talk about a couple things. A little about sockets, a little about threads, a bit about non-blocking IO, and then I’ll show you EventMachine. Also, there are no more bulleted lists in this presentation.

- Sockets
- Threads
- Non-blocking IO
- EventMachine
- No more lists

I'm going to talk about a couple things. A little about sockets, a little about threads, a bit about non-blocking IO, and then I'll show you EventMachine. Also, there are no more bulleted lists in this presentation.

Echo!

Diving right into the network code. Let’s start with a simple network server.

```ruby
require "socket"
server = TCPServer.open "127.0.0.1", 12345

client = server.accept
```

We’ll start with sockets. A basic echo server.
First, open up a socket, then listen for a connection.
Then, repeat anything back.

```ruby
require "socket"
server = TCPServer.open "127.0.0.1", 12345

client = server.accept

begin
  while data = client.readline
    client.puts data
  end
rescue EOFError
ensure
  client.close
end
```

We’ll start with sockets. A basic echo server.
First, open up a socket, then listen for a connection.
Then, repeat anything back.

```ruby
require "socket"
server = TCPServer.open "127.0.0.1", 12345

while client = server.accept
  begin
    while data = client.readline
      client.puts data
    end
  rescue EOFError
  ensure
    client.close
  end
end
```

Wednesday, August 17, 11

Ok, let’s wrap that in a loop, so it can handle more than one client.
Ok, that’s... well, that’s not gonna work either. Clients will just stack up.

```
require "socket"
server = TCPServer.open "127.0.0.1", 12345

loop do
  Thread.new(server.accept) do |client|
    begin
      while data = client.readline
        client.puts data
      end
    rescue EOFError
    ensure
      client.close
    end
  end
end
```

So, let’s wrap each client in its own thread. Cool, that’s going to work great, right? Right! Well, what’s the problem with this?

Threads Suck

The problem is, threads suck!

Threads Suck*

*For Some Values of Suck

Ok, some of the time. They can be complicated to program.

Threads Suck*

*Green threads, anyway

Wednesday, August 17, 11

At least, green threads. What’s wrong with green threads?

Threads Suck*

*Not for JRuby

And of course not for JRuby, because there, threads are native threads.

Threads Suck*

*For the sake of argument

Wednesday, August 17, 11

Alright, at least for the sake of argument. Too hard to use, or something?

Scheduling

Wednesday, August 17, 11

But the problem with green threads is scheduling. Multiple threads get equal CPU time. Even if they’re just sitting around and waiting.

Scheduling

Thread 1

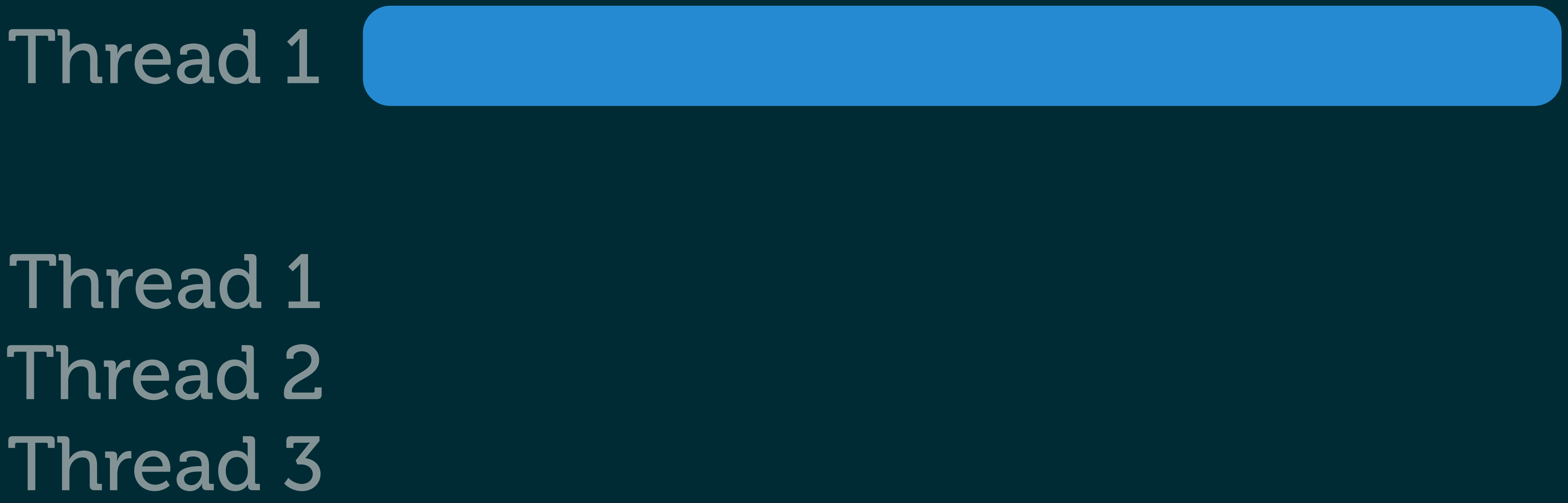
But the problem with green threads is scheduling. Multiple threads get equal CPU time. Even if they’re just sitting around and waiting.

Scheduling

Wednesday, August 17, 11

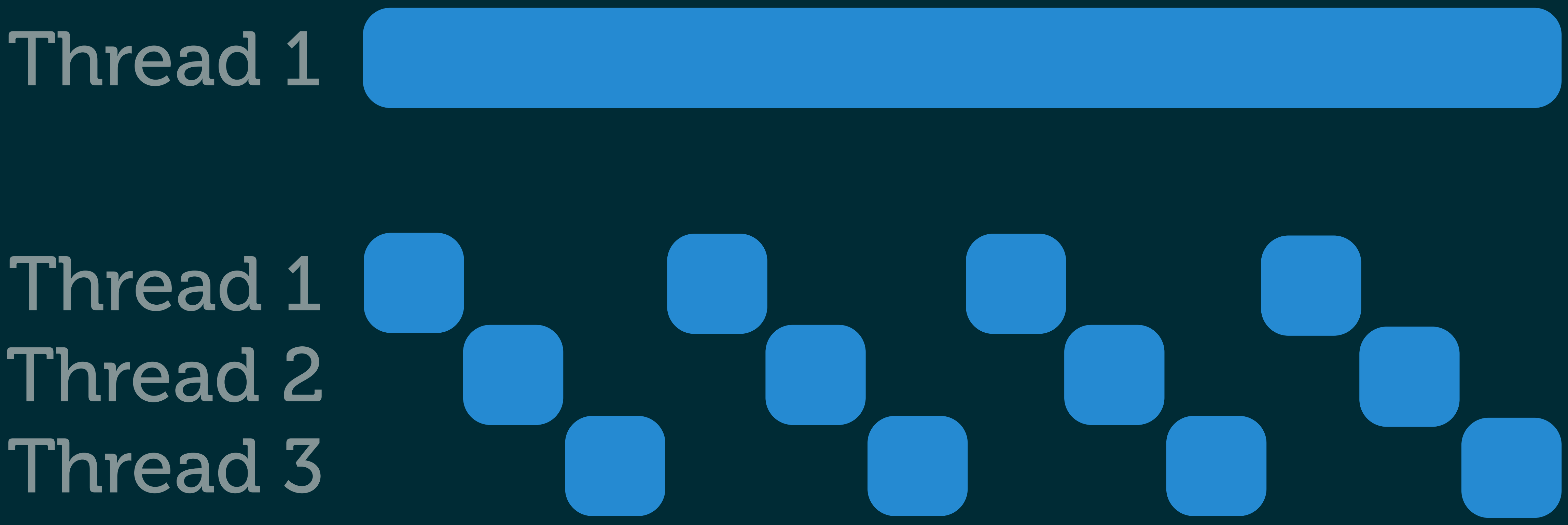
But the problem with green threads is scheduling. Multiple threads get equal CPU time. Even if they're just sitting around and waiting.

Scheduling



But the problem with green threads is scheduling. Multiple threads get equal CPU time. Even if they’re just sitting around and waiting.

Scheduling



Wednesday, August 17, 11

But the problem with green threads is scheduling. Multiple threads get equal CPU time. Even if they're just sitting around and waiting.

Why bother?

Why bother with all of this thrashing around?

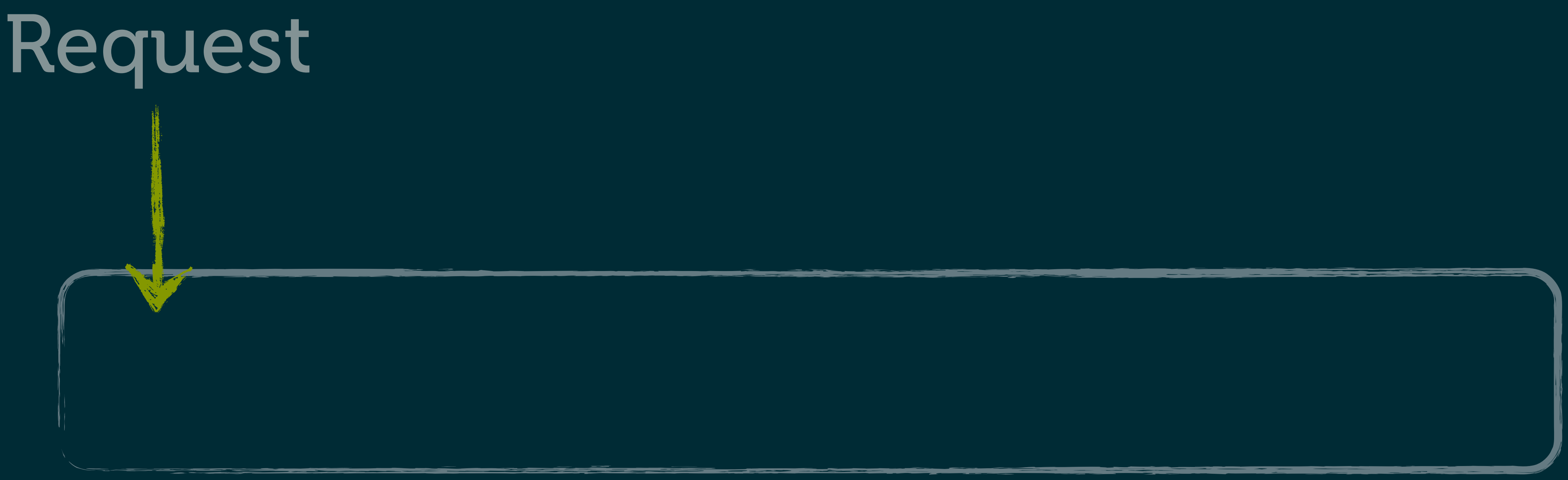
Waiting

Wednesday, August 17, 11

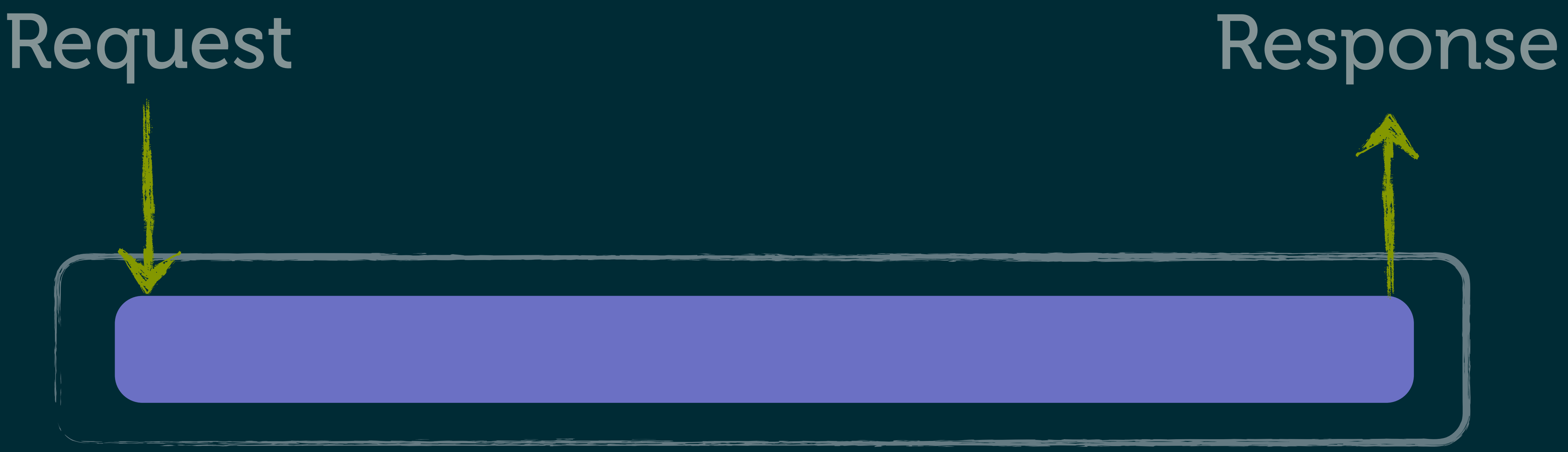
Everyone ends up just standing around waiting all the time. Like the DMV.
Ruby's not going to eat CPU doing that, it'll just bounce around between all the threads, but it's a waste of energy, because everyone's waiting for IO. This happens a lot.

Wednesday, August 17, 11

Even web apps. A request comes in, stuff happens, the response goes out. Easy. But what’s going on? There’s some db queries, a memcache request... and all that time, the server’s doing nothing but waiting around.

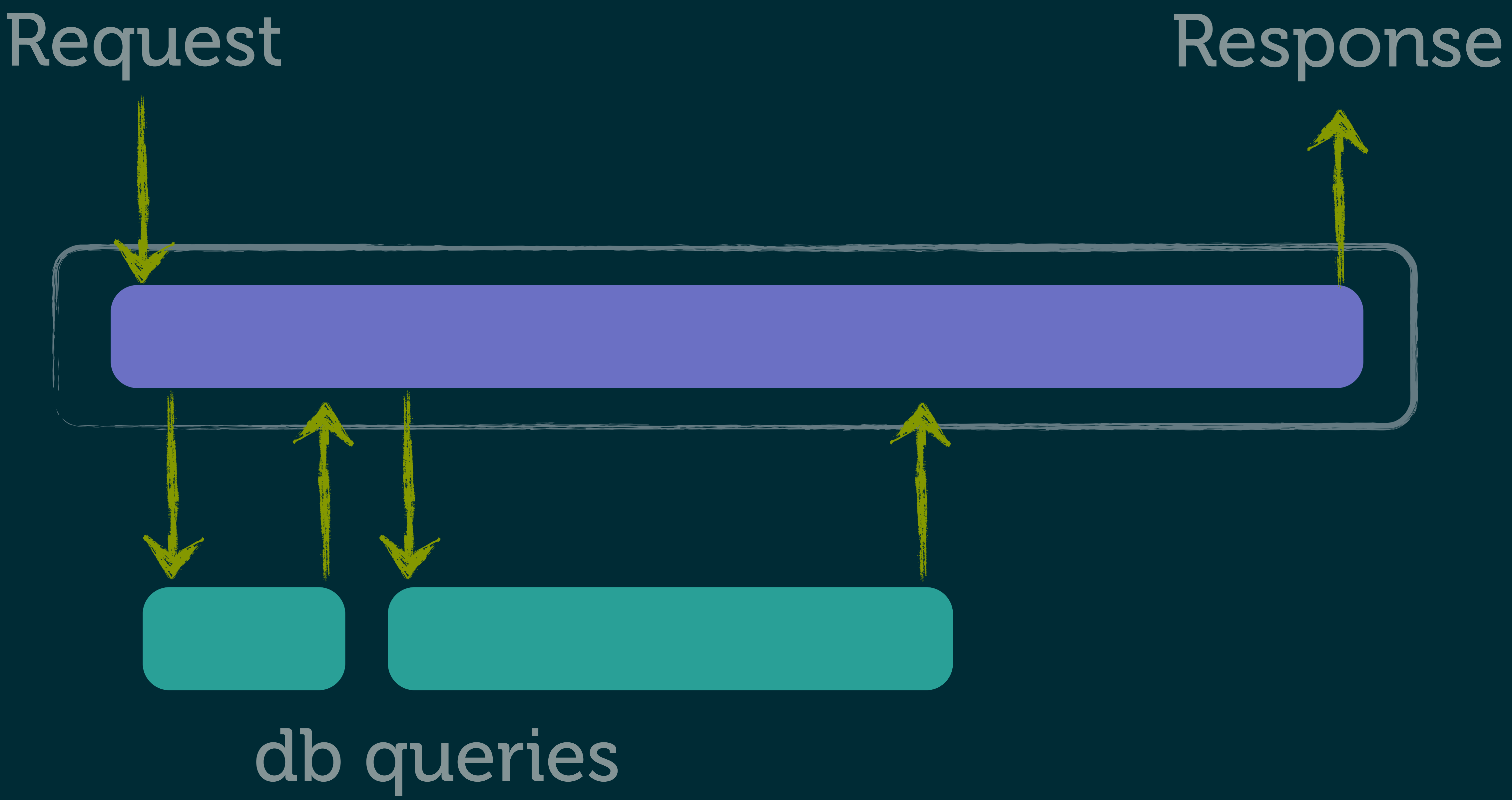


Even web apps. A request comes in, stuff happens, the response goes out. Easy. But what's going on? There's some db queries, a memcache request... and all that time, the server's doing nothing but waiting around.



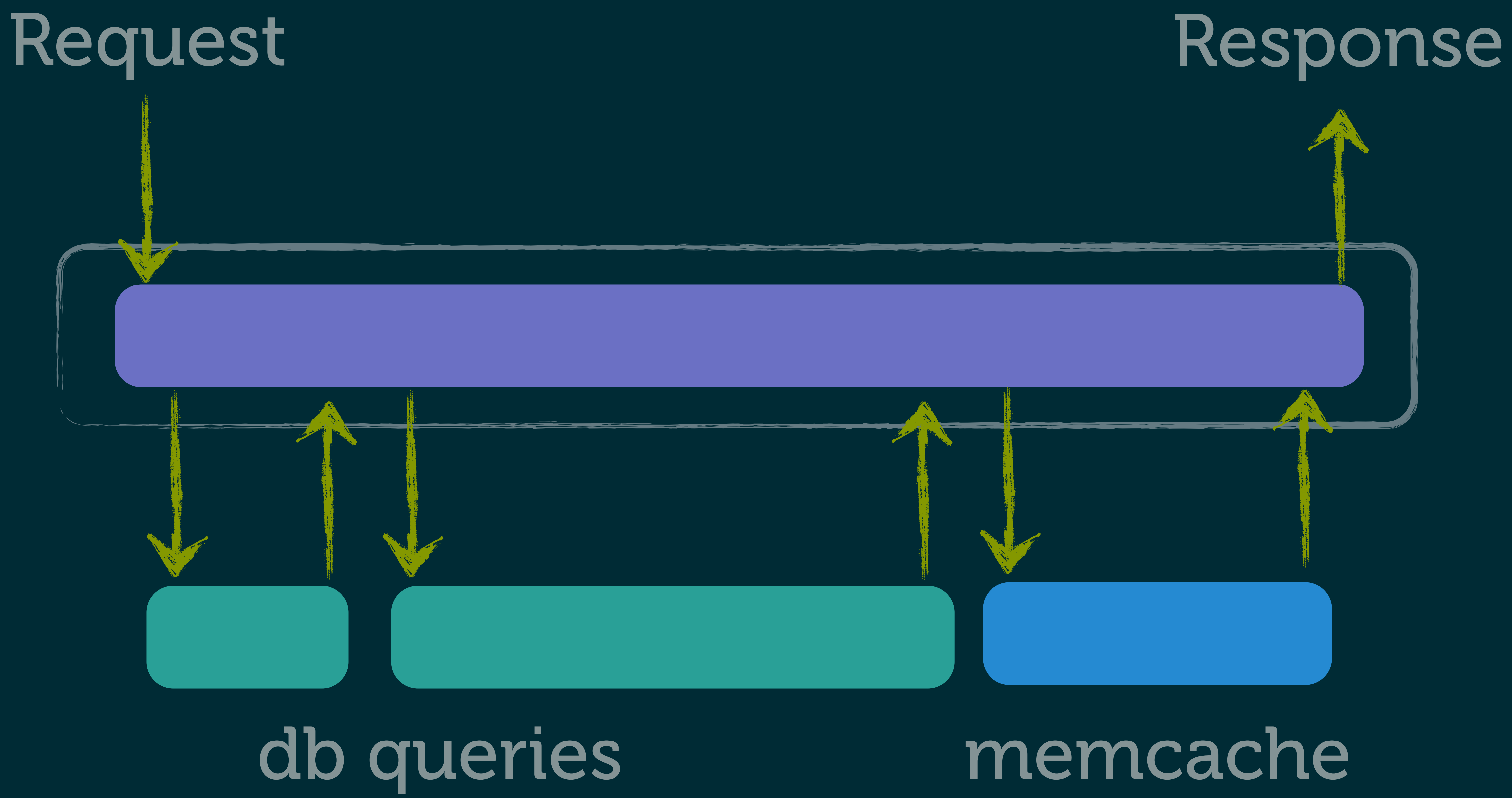
Wednesday, August 17, 11

Even web apps. A request comes in, stuff happens, the response goes out. Easy. But what’s going on? There’s some db queries, a memcache request... and all that time, the server’s doing nothing but waiting around.



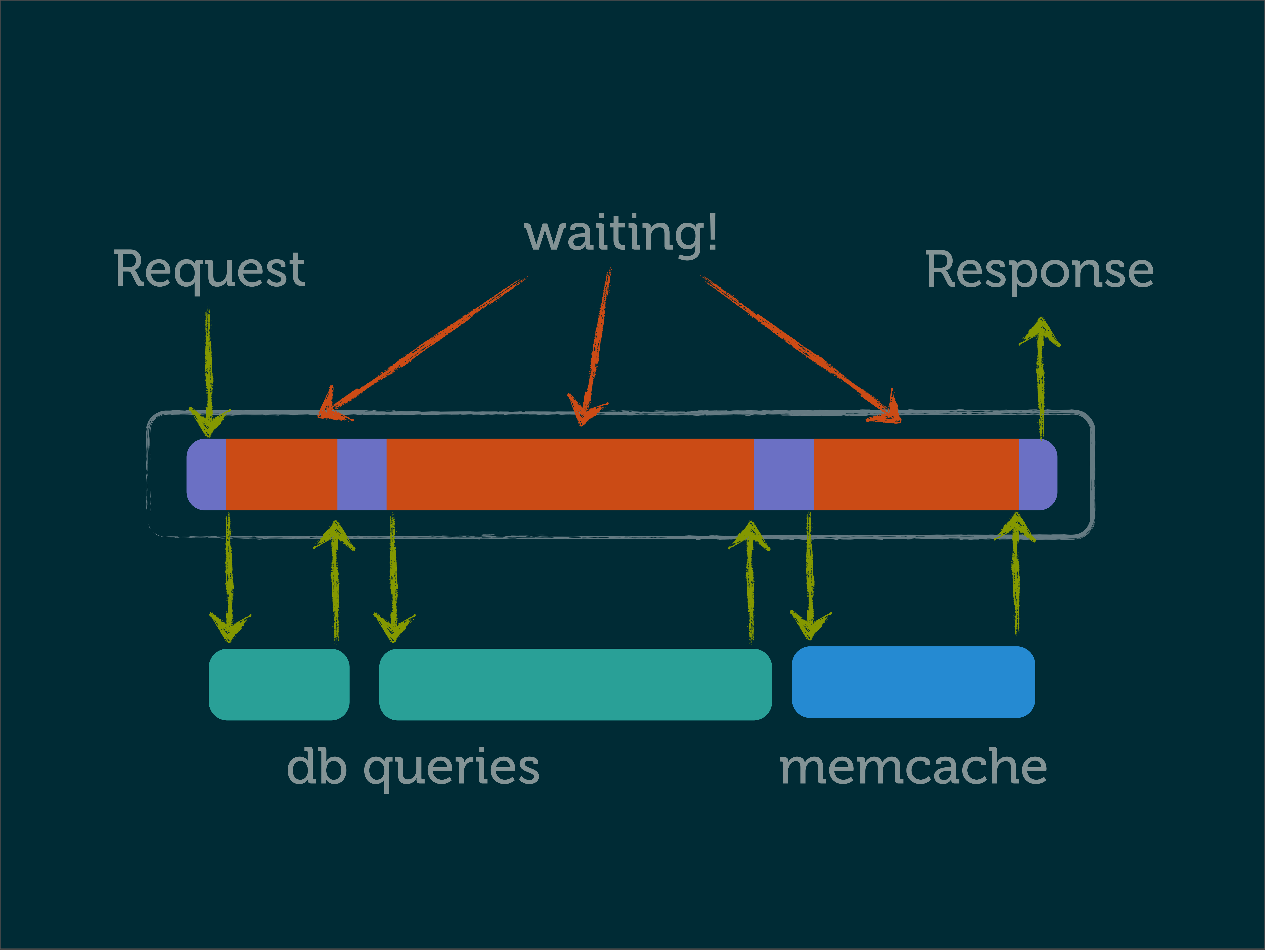
Wednesday, August 17, 11

Even web apps. A request comes in, stuff happens, the response goes out. Easy. But what’s going on? There’s some db queries, a memcache request... and all that time, the server’s doing nothing but waiting around.



Wednesday, August 17, 11

Even web apps. A request comes in, stuff happens, the response goes out. Easy. But what's going on? There's some db queries, a memcache request... and all that time, the server's doing nothing but waiting around.



Wednesday, August 17, 11

Even web apps. A request comes in, stuff happens, the response goes out. Easy. But what's going on? There's some db queries, a memcache request... and all that time, the server's doing nothing but waiting around.

Ding! Fries are done!

Wouldn’t it be nice if the software told *me* when it was ready?

`Kernel.select`

Wednesday, August 17, 11

Select. It’s a low-level kernel API call.

```
Kernel.select read_array
```

Takes a few arguments, only the first is required. An array of IO objects or file descriptors.

```
Kernel.select read_array

Kernel.select read, write
```

Wednesday, August 17, 11

Takes a few arguments, only the first is required. An array of IO objects or file descriptors.

```ruby
Kernel.select read_array

Kernel.select read, write

Kernel.select read, write, error
```

Takes a few arguments, only the first is required. An array of IO objects or file descriptors.

```
Kernel.select read_array

Kernel.select read, write

Kernel.select read, write, error

Kernel.select read, write, error, timeout
```

Takes a few arguments, only the first is required. An array of IO objects or file descriptors.

```
readable, writeable = Kernel.select(...)
```

Wednesday, August 17, 11

Select returns a list of each object that is readable, writeable, or has an error.
If no timeout is given, it will wait forever.

```
server = TCPServer.open ...

loop do
  read, write = IO.select([server])
  # ...
end
```

So let’s use this. Maybe we can service a bunch of clients all at once. We’ll start with a loop, and then call select on our server.

```
server = TCPServer.open ...
clients = []

loop do
  read, write = IO.select([server] + clients)
  read.each do |io|
    if io == server
      clients << server.accept
    else
      io.write io.read
    end
  end
end
```

Alright, so the server will tell us when it’s ready, i.e. someone just connected. First, let’s keep track of the connected clients. Then, if a client’s got data, we’ll write it right back.

Hooray!

Wednesday, August 17, 11

Well, that’s that! It works!

But...

But... that’s a lot of work to do just a little bit of code.

API calls?

What if we need to talk to a bunch of services, like HTTP servers?

Databases?

What about databases?

Timers?

Wednesday, August 17, 11

And what happens when we need to do things at intervals?

```
sleep 10
```

```ruby
readable, writeable = IO.select clients
readable.each do |io|
  data = io.read

  sleep 10

  io.write data
end
```

Uh oh!

Wednesday, August 17, 11

Kernel.select can’t do a thing about this! Shoot. Now what are the options? Go back into threads again? Bleh.

Other problems?

Wednesday, August 17, 11

What other problems are there with this approach?

c10k

Wednesday, August 17, 11

The c10k problem: how do we write a program that can handle 10,000 simultaneous connections?

`select()` limit: 1024

Wednesday, August 17, 11

The select call can only handle up to 1024 descriptors, so that’s only 1024 connections.

Threads Suck*

*still

Threads, again. We could throw a ton of threads at the problem. Ruby’s not going to do very well with thousands of threads.

epoll / kqueue

Wednesday, August 17, 11

epoll on linux, and kqueue on bsd (including mac): better than select, can handle huge numbers. Hooray!

Too low-level

But the real problem is, this all ends up being too low-level. It took a lot of work to make a simple network server. Now imagine setting up something more complex. Parsing HTTP requests, making api calls, talking to a database, etc.

Patterns

Let’s step back, generalize a bit, and talk about what we’re doing in terms of patterns.

```
loop do
  wait_for_data
  do_something_with_it
end
```

This is the general pattern that we were just using. In a loop, wait for data, then do something with it. Call it an event loop?

```
data = client.get_data

do_stuff_with(data)
```

The core of what's going on is: we ask the client for data, then we do something with it. We're the ones controlling the interaction. What if, instead, the client told *us* when data was ready?

```
client.receive do |data|
  do_stuff_with(data)
end
```

Now, the client is telling us when the data is ready. We give the client a block of code to run at its leisure, whenever the data's ready.

Inversion of Control

Wednesday, August 17, 11

This pattern is called “inversion of control”.

Holla Back!

Wednesday, August 17, 11

You’ve probably seen this before. Callbacks!

jQuery?

Wednesday, August 17, 11

Who here's done ajax with jQuery?

```
$.ajax({
  type: 'GET',
  url: '/stuff',
  success: function(data) {
    // do stuff with data
  }
});
```

A jQuery ajax call. Makes the call, asynchronously, and when the data’s ready, *it* tells *us* that it’s ready, and calls our code in the “success” callback.

```
clients = []
loop do
  readable, _ = IO.select([server] + clients)
  readable.each do |io|
    if io == server
      clients << server.accept
    else
      io.write io.read
    end
  end
end
```

Wednesday, August 17, 11

Let’s generalize what we were doing, but this time using callbacks. Here’s where we started. In a loop, do two things: accept new clients, and echo things back to existing clients.

```
server.client_connected do |client|
  client.receive_data do |data|
    client.send_data data
  end
end

loop do
  wait_for_server_or_clients
  notify_server_or_clients
end
```

Wednesday, August 17, 11

Now, reimagine this with callbacks. Two callbacks we care about: new clients, and receiving data from clients.

Reactor Pattern

And there’s a name for this pattern: the “reactor pattern”. Event handling loop, which notifies your code when things happen.

EventMachine

Wednesday, August 17, 11

Alright, let's talk about EventMachine. As you probably guessed, EventMachine uses the reactor pattern and an event loop to do its thing.

Event Loop

EventMachine is based around a simple event loop.

```ruby
require "eventmachine"

EventMachine.run do
  # ...
end
```

And that’s the event loop. EventMachine.run.

```ruby
require "eventmachine"

EM.run do
  # ...
end
```

For shorthand, EventMachine is aliased as EM.

```ruby
EM.run do
  EM.start_server '127.0.0.1', 12345
end
```

Let’s start a server up. Listening on the same port.

```
module Echo
  # callbacks
end

EM.run do
  EM.start_server '127.0.0.1', 12345, Echo
end
```

Of course, there's nothing there to handle clients. What I left out was the final parameter to start_server: a module (or class) that implements the necessary callbacks.

```
module Echo
  def receive_data(data)
    send_data data
  end
end

EM.run do
  EM.start_server '127.0.0.1', 12345, Echo
end
```

And let’s fill out the Echo module. Using the “receive_data” callback, and calling “send_data” within it.

```ruby
module Echo
  def post_init
    puts "connection initialized"
  end

  def connection_completed
    puts "connection established"
  end

  def receive_data(data)
    puts "received #{data}"
  end

  def unbind
    puts "connection closed"
  end
end
```

Here’s the available callbacks on an EM connection.

```
EventMachine.run do
  # get me out of here!
end
```

All this time we’ve working inside the event loop. What if we need to get out?

```
EventMachine.run do

  # ...

  EM.stop
end
```

You can stop the loop at any time by calling EM.stop

```ruby
EM.run do
  trap("INT") do
    # clean up...
    EM.stop
  end
end
```

Wednesday, August 17, 11

Can even handle ctrl+c gracefully.

```
EM.kqueue

EM.epoll
```

And one more thing. EM uses select by default, but supports epoll and kqueue as well. Just call these, and it’ll enable it if it’s available.

```ruby
# running as superuser on linux
EM.epoll

EM.set_descriptor_table_size(60000)

# now, drop our privileges
EM.set_effective_user "nobody"
```

Some other issues include kernel limitations of how many descriptors you're allowed to use. Will probably need superuser, but you can set the descriptor size and then de-escalate privileges once things are set up.

Protocols

Wednesday, August 17, 11

EventMachine can handle a whole bunch of protocols.

```
module EchoLines
  def receive_data(data)
    send_data data
  end
end
```

So let's talk about our echo server again. What data are we receiving here? Who knows! It's whatever gets sent, period. TCP is a *stream* of data. But let's say we want to echo lines, not characters, as we receive them.

TCP is a stream!

```ruby
module EchoLines
  def post_init
    @buffer = ""
  end

  def receive_data(data)
    @buffer << data
    *lines, @buffer = @buffer.split "\n", -1
    lines.each do |line|
      # handle line
    end
  end
end
```

And here’s how we can handle that. Buffer things until we get a line, then process it.

```ruby
module EchoLines
  include EM::Protocol::LineText2

  def receive_line(line)
    # ...
  end
end
```

Or, let’s let EventMachine handle this. There’s a whole bunch of protocols implemented, and one of these handles the line-based buffering. It’s LineText2 because there’s already a LineAndText protocol, and this is an improved version of it.

Built-in!

There’s a whole bunch that are built in!

Socks4 Memcache

Headers & Content PostgreSQL

Basic HTTP client Stomp SASL Auth

delimiter-based protocols SMTP Client

Marshaled Ruby Objects SMTP Server

Some of the other protocols implemented. Basic http client, SMTP client (and server!), marshaled ruby objects, delimiter-based stuff (lines),stomp, postgres, memcached. Neat.

Gems!

Wednesday, August 17, 11

And a whole lot more available as gems

XMPP	MySQL	AMQP
Redis	Beanstalk	DNS
Thrift	Websockets	Oscar (AIM)
Cassandra	CouchDB	0MQ
ICMP	SNMP	MongoDB

What else is available? Not an exhaustive list! I'll show you one of these in just a moment.

```ruby
gem "em-http-request"

require "em-http-request"

EM.run do
  http = EM::HttpRequest.new(ARGV.first).get

  http.callback do
    puts "success: #{http.response_header.status}"
    puts http.response
    EM.stop
  end

  http.errback do
    puts "error: " + http.error
    EM.stop
  end
end
```

So let’s try one of these. An http client. This is not the built-in one, as it’s somewhat limited. This one’s better. And look, there’s even error handling!

HTTP APIs

Wednesday, August 17, 11

And there’s a ton of HTTP api clients for eventmachine.

AWS-S3 Flickr

Solr Twitter Campfire

PubSubHubbub

And of course there are more than this. But this does include streaming APIs, like campfire and twitter, so it’s well-suited for doing campfire bots, etc.

```
module Status
  # more shorthand!
  include EM::P::LineText2

  def receive_line(url)
    http = EM::HttpRequest.new(url).head

    http.callback do
      send_data "#{url} is up!\n"
    end

    http.errback do
      send_data "#{url} unavailable\n"
    end
  end

end

EM.run do
  EM.start_server 'localhost', 12345, Status
end
```

Wednesday, August 17, 11

To tie some of that together, here’s a slightly more complex example.

Timers

```
EM.run do
  sleep 10
end
```

NO! bad! everything stops!

```
EM.run do
  ~~sleep 10~~
end
```

NO! bad! everything stops!

First Rule of EventMachine

The first rule of event machine is...

Do Not Block The Event Loop!

If you block the event loop, *nothing else can happen*.

```
EM.run do
  EM.add_timer(10) { "slept!" }
end
```

calling “add_timer” sets a timer in the internal eventmachine loop, and after it’s expired the callback is run.

```
EM.run do
  EM.add_periodic_timer(5) do
    "every 5 seconds!"
  end
end
```

You can even add periodic timers.

```
EM.run do
  EM.add_periodic_timer(5) do
    "every 5 seconds!"
  end

  # other stuff, yay!
end
```

But the best part is, the timers are asynchronous. Everything else can keep running, and you can do other stuff at the same time.

Single-threaded

Notice I haven't said anything about threads in EventMachine. That's because by default, it doesn't do anything with threads at all. One thread, one CPU, lots of IO.

Heavy Lifting

But what if I want to do heavy lifting? Lots of CPU usage?

```
EM.run do
  fibonacci(1_000_000)
end
```

Like doing a lot of calculations. Bad idea! It blocks the event loop!

```
EM.run do
  ~~fibonacci(1_000_000)~~
end
```

Wednesday, August 17, 11

Like doing a lot of calculations. Bad idea! It blocks the event loop!

First Rule of EventMachine

The first rule of event machine is...

Do Not Block The Event Loop!

If you block the event loop, *nothing else can happen*.

Defer

Wednesday, August 17, 11

So: let’s defer the CPU to elsewhere.

```ruby
EM.run do
  EM.defer do
    fibonacci(1_000_000)
  end
end
```

Thread pool

For EM.defer, EM keeps an internal thread pool around. Only 20 threads by default, to keep performance good. Can’t spend too much time mucking about with threads!

Go Easy

Wednesday, August 17, 11

In short, go easy on eventmachine. Don’t do lots of CPU, or make sure you optimize things as well as you can.

Testing

Wednesday, August 17, 11

Can’t not mention testing.

em-spec

Wednesday, August 17, 11

em-spec, for testing asynchronous code.

```
require "em/spec"

EM.describe EventMachine do

  should "have timers" do
    start = Time.now
    EM.add_timer(0.5){
      (Time.now-start).should.be.close 0.5, 0.1
      done # tell em-spec we're done
    }
  end

end
```

An example (from the README) of testing a piece of EM code.

What’s it good for?

Glue!

API clients and servers

Networking

Streaming

In the wild

Eventmachine in the wild, where you might see it.

Thin

the Thin rack server

Rainbows

Wednesday, August 17, 11

Rainbows, also a rack server

Cramp

Cramp, which is an async web app framework. Does websockets and things really well.

Goliath

Goliath. Async api server.

Alternatives

node.js

Wednesday, August 17, 11

Kidding, sorta. There are a lot of similarities.

cool.io

Wednesday, August 17, 11

But here's another serious alternative. Using libev, rather than a hand-rolled event loop. Actor pattern, rather than reactor, and replaces the underlying ruby IO objects rather than adding its own. Worth checking out.

Questions?

Links

EventMachine – https://github.com/eventmachine/eventmachine

EM Wiki – https://github.com/eventmachine/eventmachine/wiki

em-http-request – https://github.com/igrigorik/em-http-request

em-spec – https://github.com/tmm1/em-spec

c10k problem – http://www.kegel.com/c10k.html

Cramp – http://cramp.in/

Goliath – http://postrank-labs.github.com/goliath/

cool.io – http://coolio.github.com/

Thanks!