EventMachine

scalable non-blocking i/o in ruby

About Me

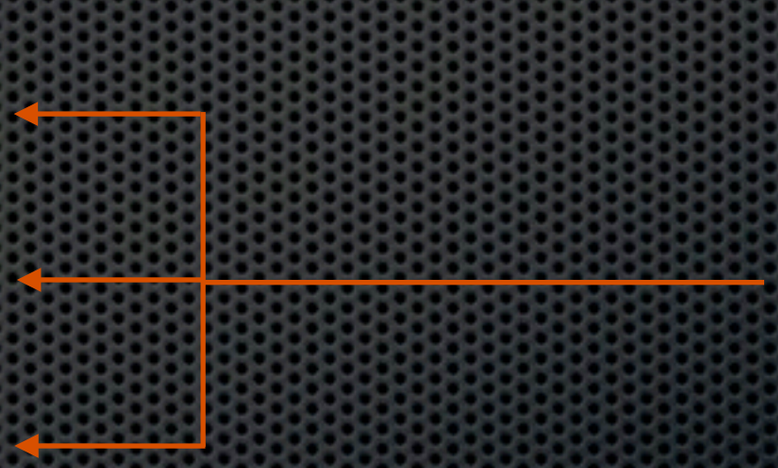
- Aman Gupta
- San Francisco, CA
- Been maintaining EventMachine for 18 months (last 4 releases)
- Ruby Hero 2009
- amqp, REE patches, perftools.rb, gdb.rb, memprof
- github.com/tmm1
- @tmm1

What is EventMachine?

- Implementation of the Reactor pattern
  - similar to Python's Twisted

What is EventMachine?

- Implementation of the Reactor pattern
  - similar to Python's Twisted
- Ruby VM Support
  - Ruby 1.8 (MRI)
  - Ruby 1.9 (YARV)
  - Rubinius
  - JRuby



What is EventMachine?

- Implementation of the Reactor pattern
  - similar to Python's Twisted
- Ruby VM Support
  - Ruby 1.8 (MRI)
  - Ruby 1.9 (YARV)
  - Rubinius
  - JRuby

What is EventMachine?

- Implementation of the Reactor pattern
  - similar to Python's Twisted
- Ruby VM Support
  - Ruby 1.8 (MRI)
  - Ruby 1.9 (YARV)
  - Rubinius
  - JRuby
  - + simple Pure Ruby version

Who uses EventMachine?

Thin

A fast and very simple Ruby web server

heroku

Engine Yard

PostRank™

github SOCIAL CODING

RIGHT SCALE®

Campfire™

and many more...

What is I/O?

- Generally Network I/O
  - mysql query responses
  - http responses
  - memcache results

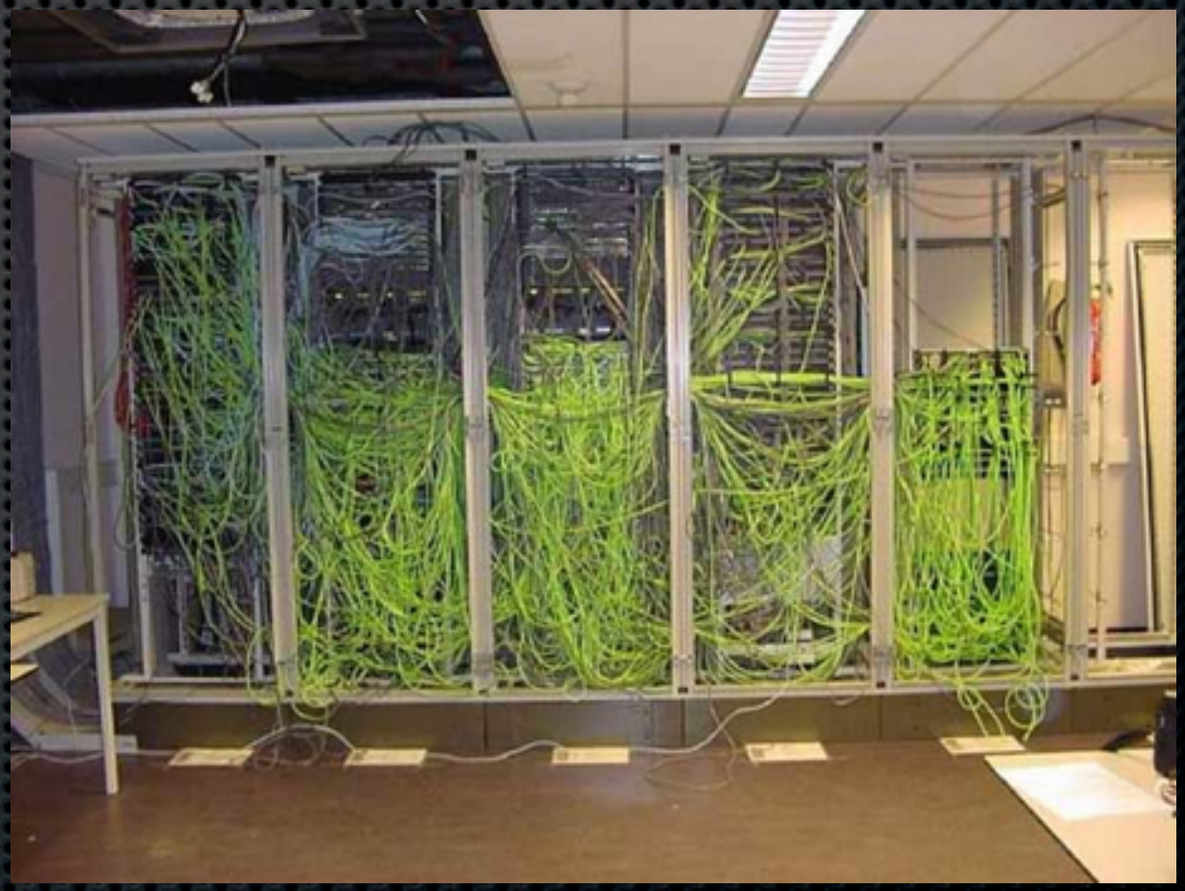
What is I/O?

- Generally Network I/O
  - mysql query responses
  - http responses
  - memcache results

- Most web applications are I/O bound, not CPU bound
- Basic idea behind EM: Instead of waiting on a response from the network, use that time to process other requests

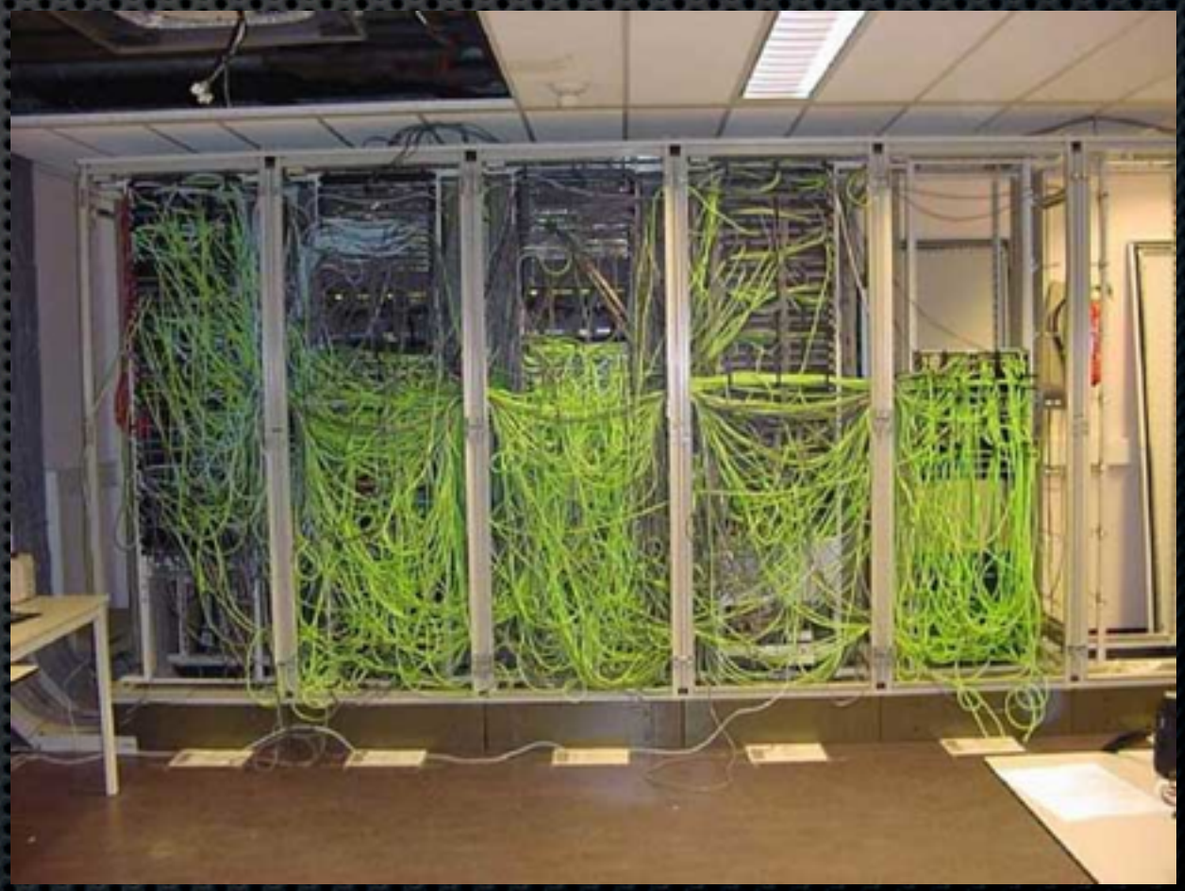
What can I use EM for?

- Scaling I/O heavy applications
  - handle 5-10k concurrent connections with a single ruby process



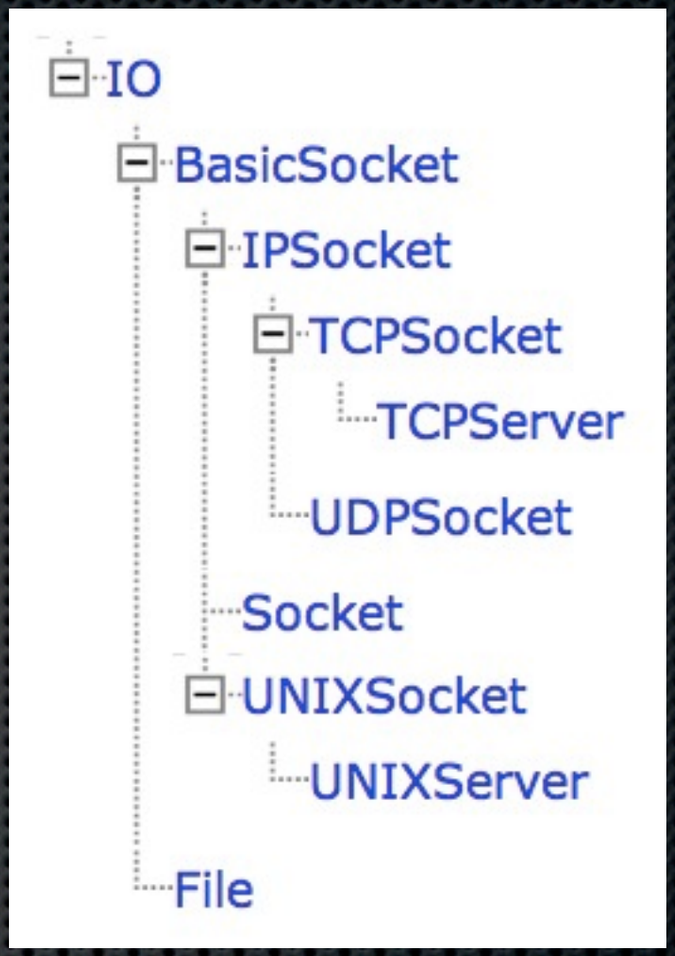
What can I use EM for?

- Scaling I/O heavy applications
  - handle 5-10k concurrent connections with a single ruby process
- Any type of network I/O
  - http requests
  - sending emails
  - custom tcp proxies
  - data access
    - redis
    - couchdb
    - mysql
    - postgres
    - casandra
    - memcached



I/O in Ruby

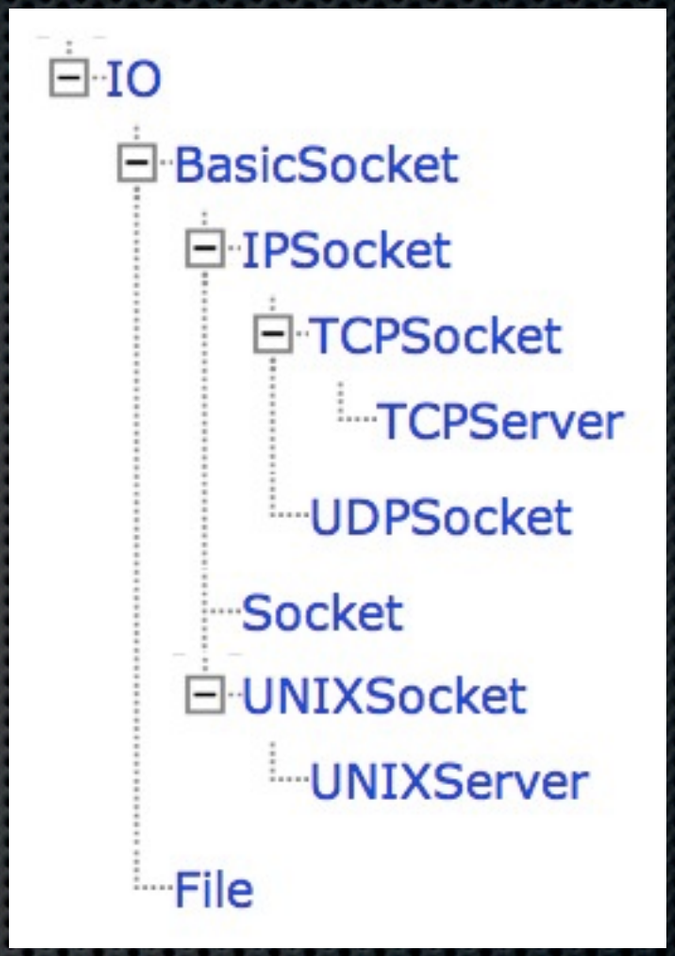
- TCPSocket
- TCPServer
- Socket
  - for raw access to BSD socket API
  - **is not** TCPSocket's superclass



http://memprof.com/demo

I/O in Ruby

- TCPSocket
- TCPServer
- Socket
  - for raw access to BSD socket API
  - **is not** TCPSocket's superclass



http://memprof.com/demo

```
require 'socket'
include Socket::Constants
socket = Socket.new(AF_INET, SOCK_STREAM, 0)
sockaddr = Socket.pack_sockaddr_in(2200, 'localhost')
socket.connect(sockaddr)
socket.puts "Hello from script 2."
puts "The server said, '#{socket.readline.chomp}'"
socket.close
```

Simple TCP Server

- TCPServer#accept to accept connections from new clients
- TCPSocket#read* blocks, so you can only handle one client at a time

```
require 'socket'
server = TCPServer.new(2202)
while client = server.accept
  msg = client.readline
  client.write "You said: #{msg}"
  client.close
end
```

Simple TCP Server

- TCPServer#accept to accept connections from new clients
- TCPSocket#read* blocks, so you can only handle one client at a time
- Common Solution: use a thread per client

```
require 'socket'
server = TCPServer.new(2202)
while client = server.accept
  msg = client.readline
  client.write "You said: #{msg}"
  client.close
end
```

```
require 'socket'
server = TCPServer.new(2202)
while true
  Thread.new(server.accept){ |client|
    msg = client.readline
    client.write "You said: #{msg}"
    client.close
  }
end
```

Non-Blocking I/O

- Alternative to threads: simply don't block

```
require 'socket'
server = TCPServer.new(2202)
clients = []
buffers = {}

while true
  sockets = [server] + clients
  readable, writable = IO.select(sockets)

  readable.each do |sock|
    begin
      if sock == server
        clients << server.accept_nonblock
      else
        client, buf = sock, buffers[sock] ||= ''

        buf << client.read_nonblock(1024)
        if buf =~ /^.+?\r?\n/
          client.write "You said: #{buf}"
          client.close

          buffers.delete(client)
          clients.delete(client)
        end
      end
    rescue Errno::EAGAIN, Errno::EWOULDBLOCK
      # socket would block, try again later
    end
  end
end
```

list of clients

```
require 'socket'
server = TCPServer.new(2202)
clients = []
buffers = {}

while true
  sockets = [server] + clients
  readable, writable = IO.select(sockets)

  readable.each do |sock|
    begin
      if sock == server
        clients << server.accept_nonblock
      else
        client, buf = sock, buffers[sock] ||= ''

        buf << client.read_nonblock(1024)
        if buf =~ /^.+?\r?\n/
          client.write "You said: #{buf}"
          client.close

          buffers.delete(client)
          clients.delete(client)
        end
      end
    rescue Errno::EAGAIN, Errno::EWOULDBLOCK
      # socket would block, try again later
    end
  end
end
```

Non-Blocking I/O

- Alternative to threads: simply don't block

inbound buffer per client

list of clients

```
require 'socket'
server = TCPServer.new(2202)
clients = []
buffers = {}

while true
  sockets = [server] + clients
  readable, writable = IO.select(sockets)

  readable.each do |sock|
    begin
      if sock == server
        clients << server.accept_nonblock
      else
        client, buf = sock, buffers[sock] ||= ''

        buf << client.read_nonblock(1024)
        if buf =~ /^.+?\r?\n/
          client.write "You said: #{buf}"
          client.close

          buffers.delete(client)
          clients.delete(client)
        end
      end
    rescue Errno::EAGAIN, Errno::EWOULDBLOCK
      # socket would block, try again later
    end
  end
end
```

Non-Blocking I/O

- Alternative to threads: simply don't block

Non-Blocking I/O

- Alternative to threads: simply don't block

inbound buffer per client

list of clients

IO.select returns readable/writable sockets

```
require 'socket'
server = TCPServer.new(2202)
clients = []
buffers = {}

while true
  sockets = [server] + clients
  readable, writable = IO.select(sockets)

  readable.each do |sock|
    begin
      if sock == server
        clients << server.accept_nonblock
      else
        client, buf = sock, buffers[sock] ||= ''

        buf << client.read_nonblock(1024)
        if buf =~ /^.+?\r?\n/
          client.write "You said: #{buf}"
          client.close

          buffers.delete(client)
          clients.delete(client)
        end
      end
    rescue Errno::EAGAIN, Errno::EWOULDBLOCK
      # socket would block, try again later
    end
  end
end
```

inbound buffer per client

list of clients

IO.select returns readable/writable sockets

read in available data, process if full line received

Non-Blocking I/O

- Alternative to threads: simply don't block

```
require 'socket'
server = TCPServer.new(2202)
clients = []
buffers = {}

while true
  sockets = [server] + clients
  readable, writable = IO.select(sockets)

  readable.each do |sock|
    begin
      if sock == server
        clients << server.accept_nonblock
      else
        client, buf = sock, buffers[sock] ||= ''

        buf << client.read_nonblock(1024)
        if buf =~ /^.+?\r?\n/
          client.write "You said: #{buf}"
          client.close

          buffers.delete(client)
          clients.delete(client)
        end
      end
    rescue Errno::EAGAIN, Errno::EWOULDBLOCK
      # socket would block, try again later
    end
  end
end
```

EM does Non-Blocking I/O

- handles low level sockets for you
- inbound/ outbound buffers for maximal throughput
- efficient i/o with writev/readv
- epoll & kqueue support

```
module EchoServer
  def post_init
    @buf = ''
  end
  def receive_data(data)
    @buf << data
    if @buf =~ /^.+?\r?\n/
      send_data "You said: #{@buf}"
      close_connection_after_writing
    end
  end
end

require 'eventmachine'
EM.run do
  EM.start_server '0.0.0.0', 2202, EchoServer
end
```

So, what's a Reactor?

```
while reactor_running?
  expired_timers.each{ |timer| timer.process }
  new_network_io.each{ |io| io.process }
end
```

- reactor is simply a **single threaded while loop**, called the "reactor loop"
- your code "reacts" to incoming events
- if your event handler takes too long, other events cannot fire

So, what's a Reactor?

```
while reactor_running?
  expired_timers.each{ |timer| timer.process }
  new_network_io.each{ |io| io.process }
end
```

- reactor is simply a **single threaded while loop**, called the "reactor loop"
- your code "reacts" to incoming events
- if your event handler takes too long, other events cannot fire
- lesson: never block the reactor
  - no sleep(1)
  - no long loops (100_000.times)
  - no blocking I/O (mysql queries)
  - no polling (while !condition)

```
while reactor_running?
  while true
  end
  # reactor is blocked!
end
```

Writing Asynchronous Code

- synchronous ruby code uses return values

```
ret = operation()
do_something_with(ret)
```

Writing Asynchronous Code

- synchronous ruby code uses return values

```
ret = operation()
do_something_with(ret)
```

- evented async code uses blocks instead

```
operation{ |ret| do_something_with(ret) }
```

Writing Asynchronous Code

- synchronous ruby code uses return values

```
ret = operation()
do_something_with(ret)
```

- evented async code uses blocks instead

```
operation{ |ret| do_something_with(ret) }
```

- different from how you usually use ruby blocks. the block is stored and invoked later (it's asynchronous)

```
puts(1)
1.times{ puts(2) }
puts(3)
```

```
puts(1)
operation{ puts(3) }
puts(2)
```

Events are simple

- Instead of **waiting** for something to happen before executing code,
- Put that code in a proc,
- Invoke the proc whenever something happens

Events are simple

- Instead of **waiting** for something to happen before executing code,
- Put that code in a proc,
- Invoke the proc whenever something happens

```
sleep 0.1 until queue.size > 0
use(queue.pop)
```

vs

```
queue.on_push = proc{ use(queue.pop) }
```

But: evented code is hard

```
url = db.find_url
response = http.get(url)
email.send(response)
puts 'email sent'
```

vs

```
db.find_url { |url|
  http.get(url) { |response|
    email.send(response) {
      puts 'email sent'
    }
  }
}
```

- nested blocks are hard to parse
- can't use exceptions- error handling requires more work
- tradeoffs either way- you choose:
  - don't need to scale
  - scale existing code using multiple processes
  - rewrite your code to be async

Hybrid Solution: EM + Threads

- Best of both worlds: run legacy blocking code inside threads, everything else inside the reactor
- EM+Thread has been flaky in the past, but all the major problems have been fixed
- You can either:
  - run the reactor in its own external thread, or
  - run specific parts of your code in external threads

Hybrid Solution: EM + Threads

- Best of both worlds: run legacy blocking code inside threads, everything else inside the reactor
- EM+Thread has been flaky in the past, but all the major problems have been fixed
- You can either:
  - run the reactor in its own external thread, or
  - run specific parts of your code in external threads
- But (ruby) threads are still **=(**
  - unnecessary amount of overhead per client

OK, so how do I use EM?

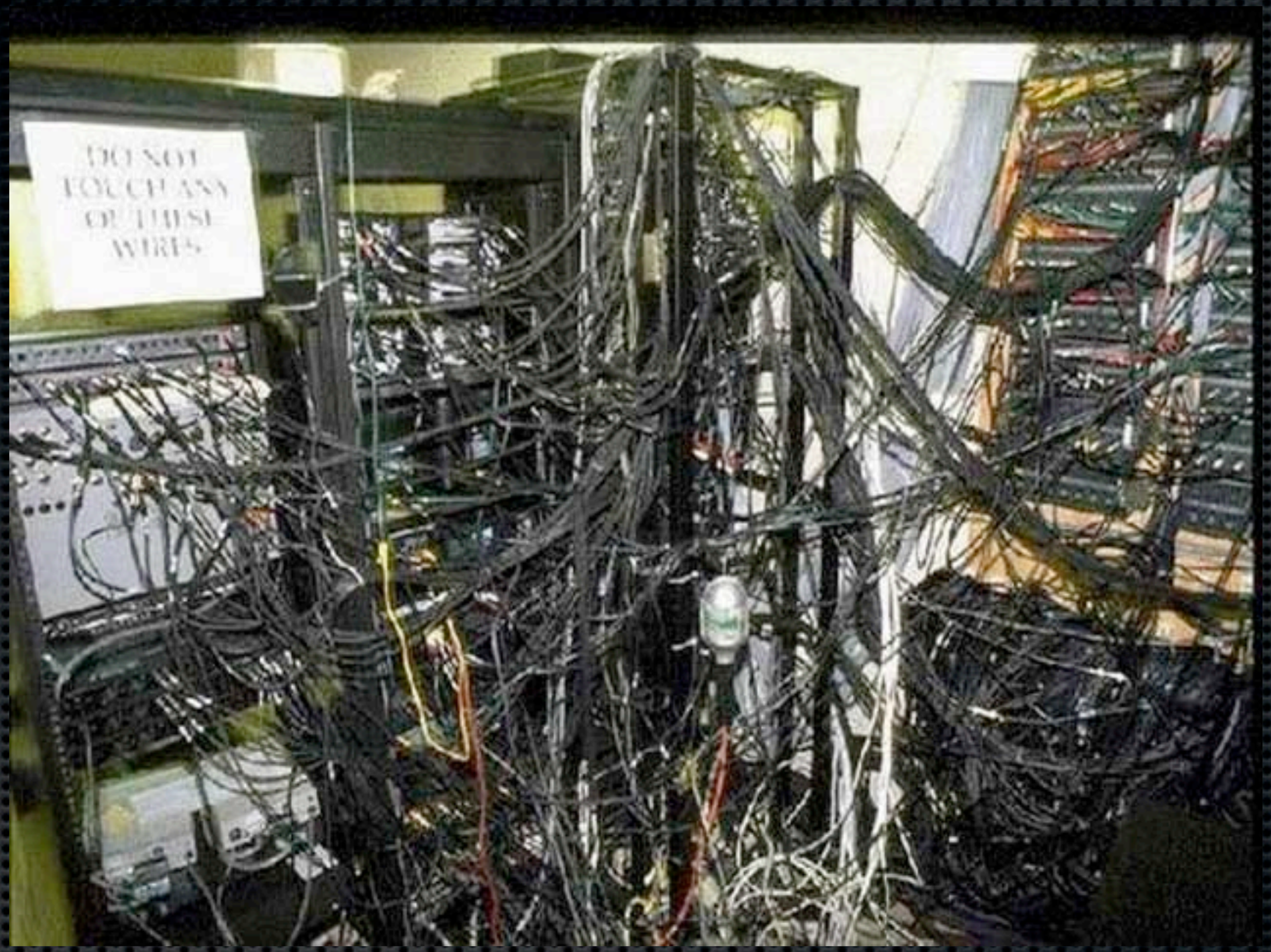
- gem install eventmachine
- require ‘eventmachine’

OK, so how do I use EM?

- gem install eventmachine
- require ‘eventmachine’

Let’s talk about the API

EM.run	EM::Deferrable	EM.popen
EM.reactor_running?	EM::Callback	EM.start_server
EM.stop	EM::Timer	EM.stop_server
EM.next_tick	EM::PeriodicTimer	EM.connect
EM::TickLoop	EM::Queue	EM::Connection
EM.schedule	EM::Channel	EM.watch
EM.threadpool_size	EM::Iterator	EM.watch_file
EM.defer	EM.system	EM::Protocols



C10K with epoll and kqueue

EM.epoll / EM.kqueue

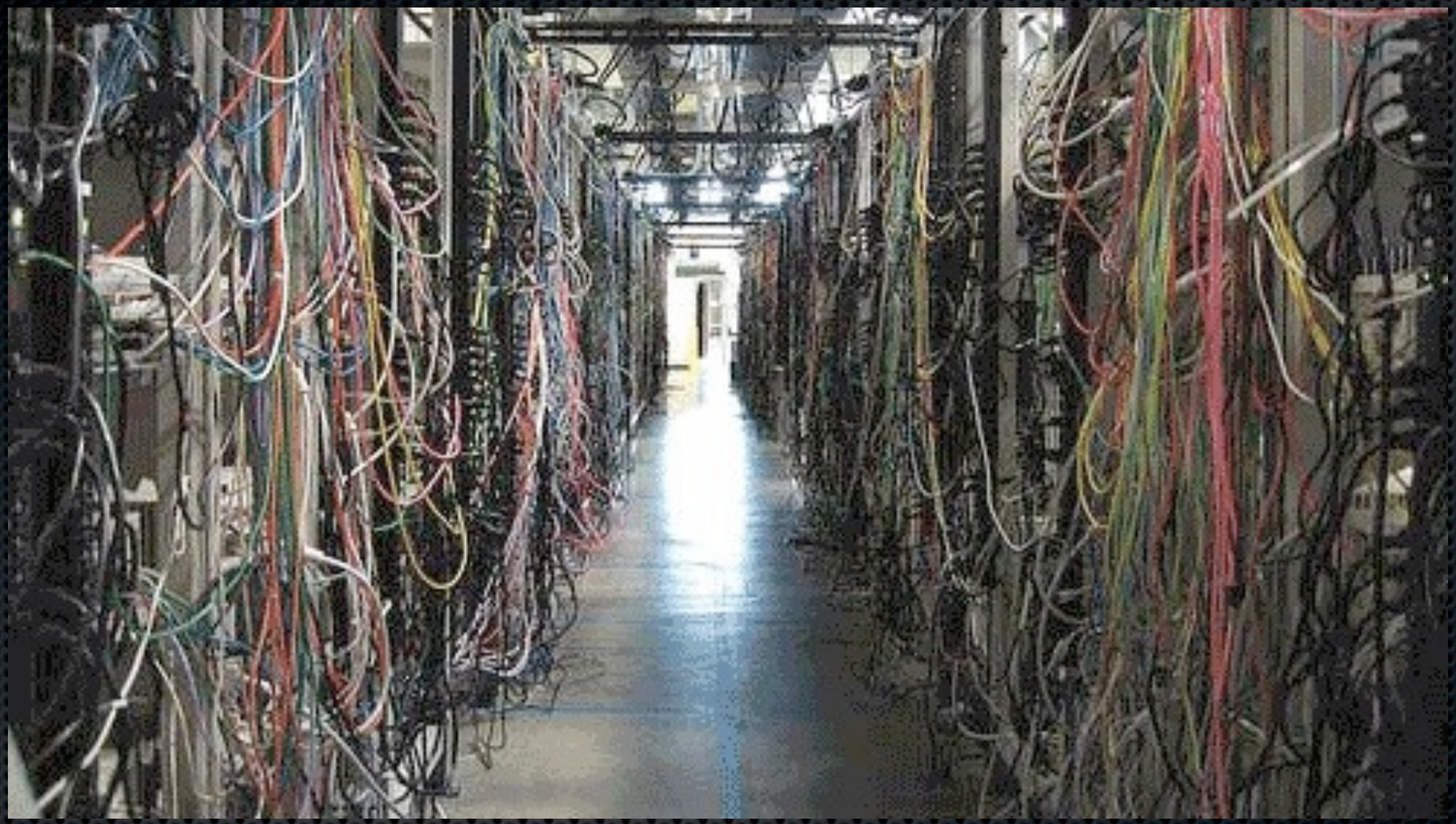
- EM use select() by default
  - portable, works well, but:
  - limited to 1024 file descriptors
  - slow for many connections (fds are copied in and out of the kernel on each call)

EM.epoll / EM.kqueue

- EM use select() by default
  - portable, works well, but:
  - limited to 1024 file descriptors
  - slow for many connections (fds are copied in and out of the kernel on each call)
- Linux has epoll() and OSX/BSD have kqueue()
  - register file descriptors with the kernel once
  - get events when I/O happens
  - scales to many 10s of thousands of concurrent connections

EM.epoll / EM.kqueue

- EM use select() by default
  - portable, works well, but:
  - limited to 1024 file descriptors
  - slow for many connections (fds are copied in and out of the kernel on each call)
- Linux has epoll() and OSX/BSD have kqueue()
  - register file descriptors with the kernel once
  - get events when I/O happens
  - scales to many 10s of thousands of concurrent connections
- Just call EM.epoll or EM.kqueue before doing anything else



Running the EM reactor

EM.run

```
EM.run{
  puts "reactor started!"
}

puts "this won't happen"
```

- starts the reactor loop
- takes a block to execute once the reactor has started
- EM.run is **blocking**!
  - the loop takes over the ruby process, so code after EM.run will not run

EM.reactor_running?

- check if the reactor loop is running

EM.reactor_running?

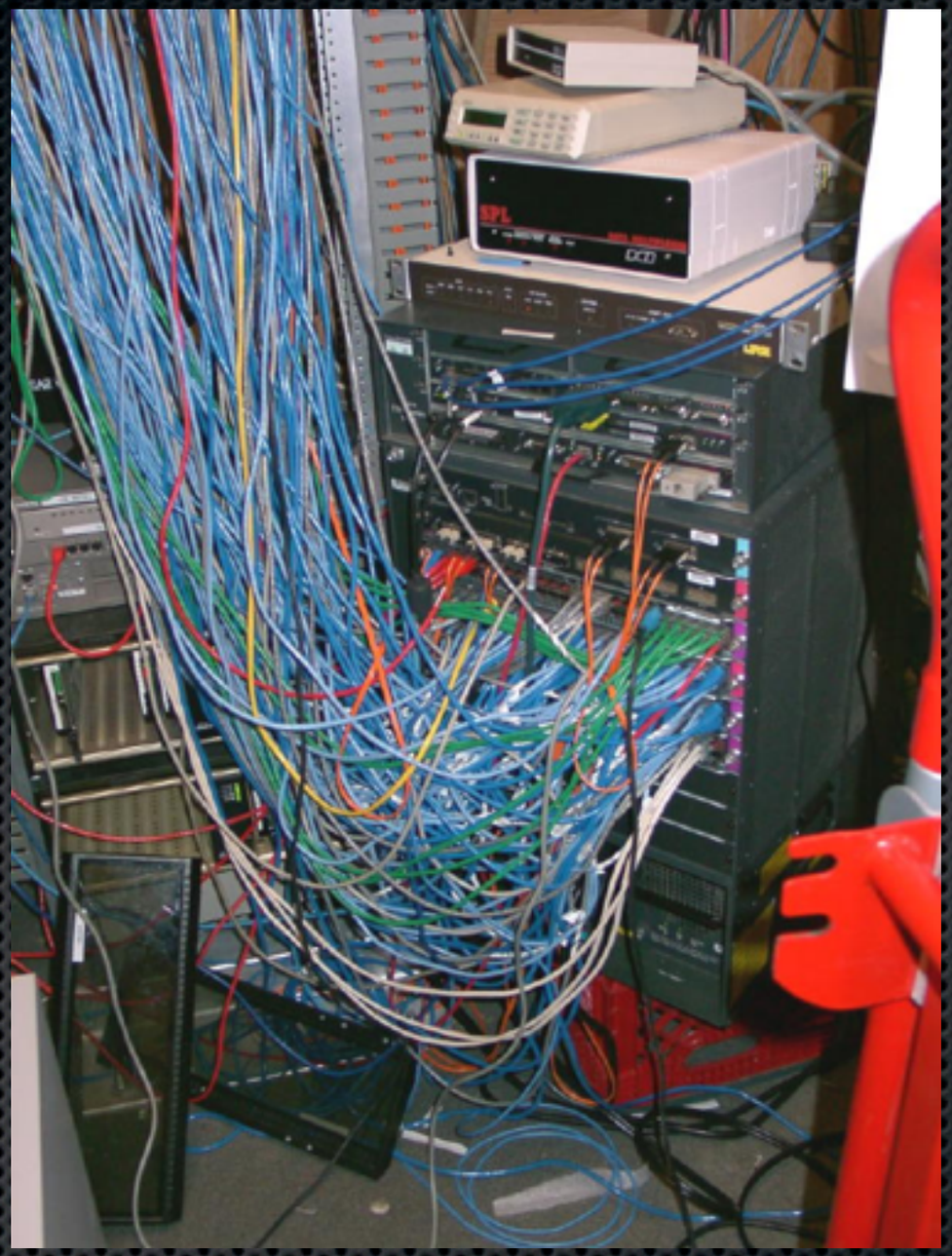
- check if the reactor loop is running

EM.stop

- stop the reactor and continue execution of the ruby script

```ruby
EM.run{
  puts "reactor started!"
  EM.stop
}

puts "this will happen"
```



Dealing with reactor iterations

EM.next_tick

- queue a proc to be executed on the **next iteration** of the reactor loop
- basic rule: do not block the reactor
  - split up work across multiple iterations of the reactor

EM.next_tick

- queue a proc to be executed on the **next iteration** of the reactor loop
- basic rule: do not block the reactor
  - split up work across multiple iterations of the reactor
  - common pattern: “recursive” calls to EM.next_tick

```
n=0
while n < 1000
  do_something
  n += 1
end
```

```
n=0
do_work = proc{
  if n < 1000
    do_something
    n += 1
    EM.next_tick(&do_work)
  end
}
EM.next_tick(&do_work)
```

EM::TickLoop

- simple wrapper around “recursive” EM.next_tick
- use carefully, TickLoop will peg your CPU
  - reactor loop iterations happen *very often* (30k+ iterations per second on my laptop)
- useful for integrating with other reactors like GTK & TK

```
n = 0
tickloop = EM.tick_loop do
  if n < 1000
    do_something
    n += 1
  else
    :stop
  end
end
tickloop.on_stop{ puts 'all done' }
```

EM::Iterator

- easier way to iterate over multiple ticks
- fine grained control over concurrency
- provides async map/inject

EM::Iterator

- easier way to iterate over multiple ticks
- fine grained control over concurrency
- provides async map/inject

```
(0..10).each{ |num| }
```

← end of block signals next iteration

EM::Iterator

- easier way to iterate over multiple ticks
- fine grained control over concurrency
- provides async map/inject

```
(0..10).each{ |num| }
```

← end of block signals next iteration

```
EM::Iterator.new(0..10).each{ |num,iter| iter.next }
```

explicitly signal next iteration →

EM::Iterator

- easier way to iterate over multiple ticks
- fine grained control over concurrency
- provides async map/inject

```
(0..10).each{ |num| }
```

← end of block signals next iteration

```
EM::Iterator.new(0..10).each{ |num,iter| iter.next }
```

explicitly signal next iteration →

```
EM::Iterator.new(0..10, 2).each{ |num,iter|
  EM.add_timer(1){ iter.next }
}
```

← wait 1 second before next iteration

EM::Iterator

- easier way to iterate over multiple ticks
- fine grained control over concurrency
- provides async map/inject

```
(0..10).each{ |num| }
```

← end of block signals next iteration

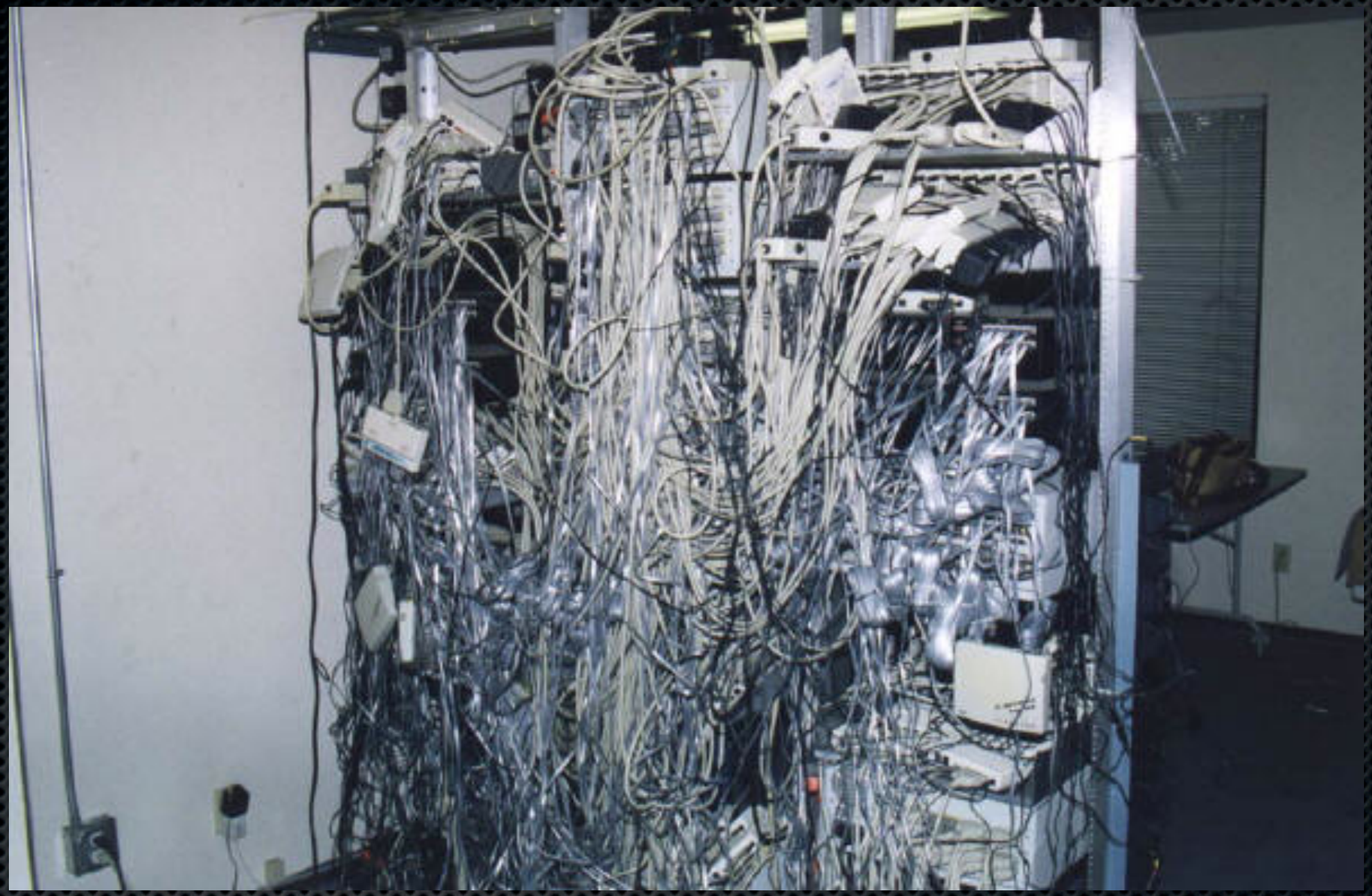
```
EM::Iterator.new(0..10).each{ |num,iter| iter.next }
```

explicitly signal next iteration →

← iterate two at a time

```
EM::Iterator.new(0..10, 2).each{ |num,iter|
  EM.add_timer(1){ iter.next }
}
```

← wait 1 second before next iteration



Threading with the reactor

Threaded EM.run

```
thread = Thread.current
Thread.new{
  EM.run{ thread.wakeup }
}

# pause until reactor starts
Thread.stop
```

- run the reactor loop in an external thread
- uses Thread.stop to pause current thread until the reactor is started
- for thread-safety, you **must** use EM.schedule to call into the EM APIs from other threads

EM.schedule

- simple wrapper for EM.next_tick
- if Thread.current != reactor_thread, schedule the proc to be called inside the reactor thread on the next iteration

EM.schedule

- simple wrapper for EM.next_tick
- if Thread.current != reactor_thread, schedule the proc to be called inside the reactor thread on the next iteration

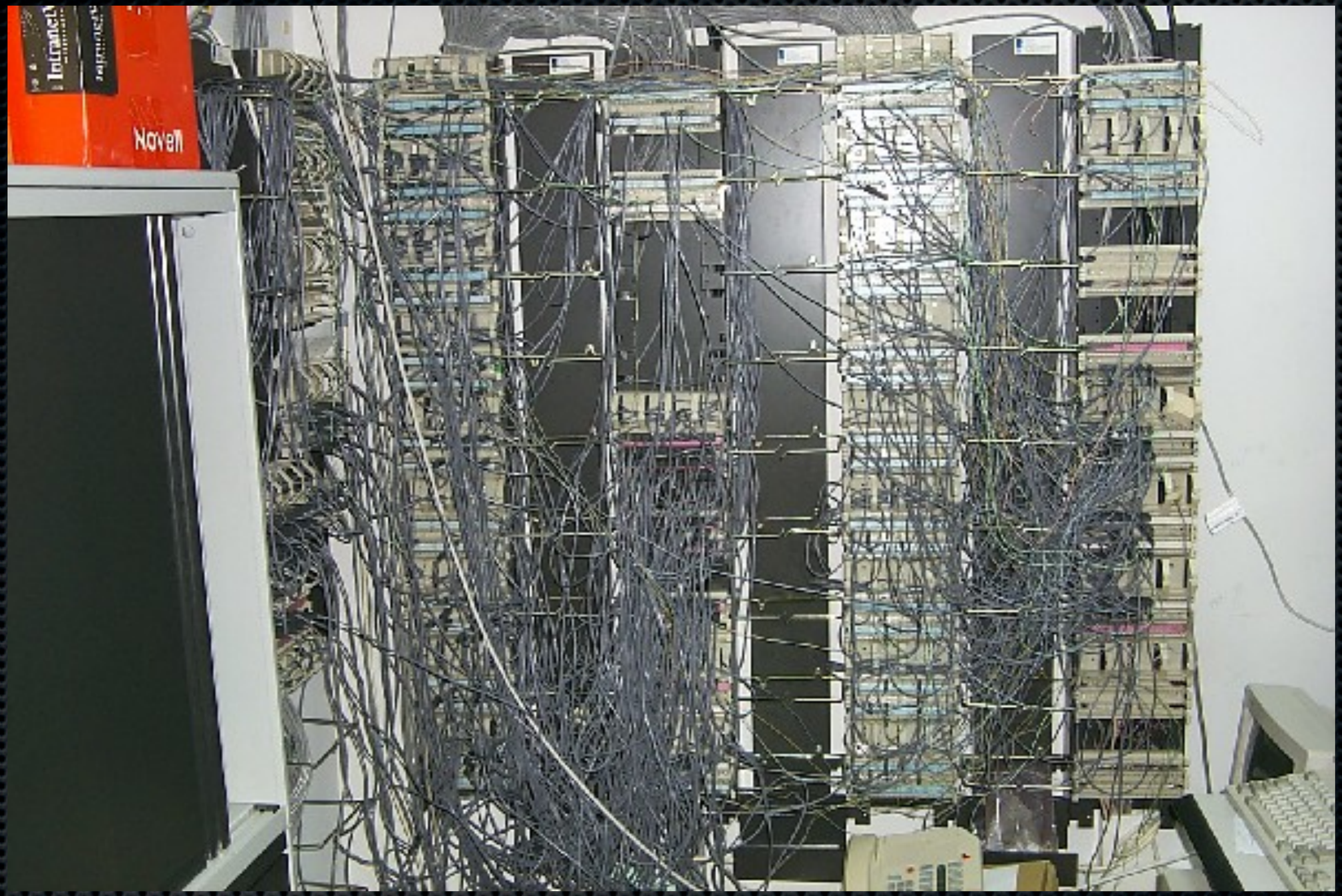
EM.threadpool_size

- instead of running the reactor in a thread, you can run specific pieces of code inside a thread pool
- useful for legacy blocking code, like database access
- threadpool_size defaults to 20
- EM **does not** use any threads by default, only when you call EM.defer

EM.defer

- on first invocation, spawns up EM.threadpool_size threads in a pool
- all the threads read procs from a queue and execute them in parallel to the reactor thread
- optional second proc is invoked with the result of the first proc, but back inside the reactor thread

```
EM.defer(proc{
  # running in a worker thread
  result = long_blocking_call
  result = process(result)
  result
}, proc{ |result|
  # back in the reactor thread
  use(result)
})
```



Timing inside the reactor

EM::Timer

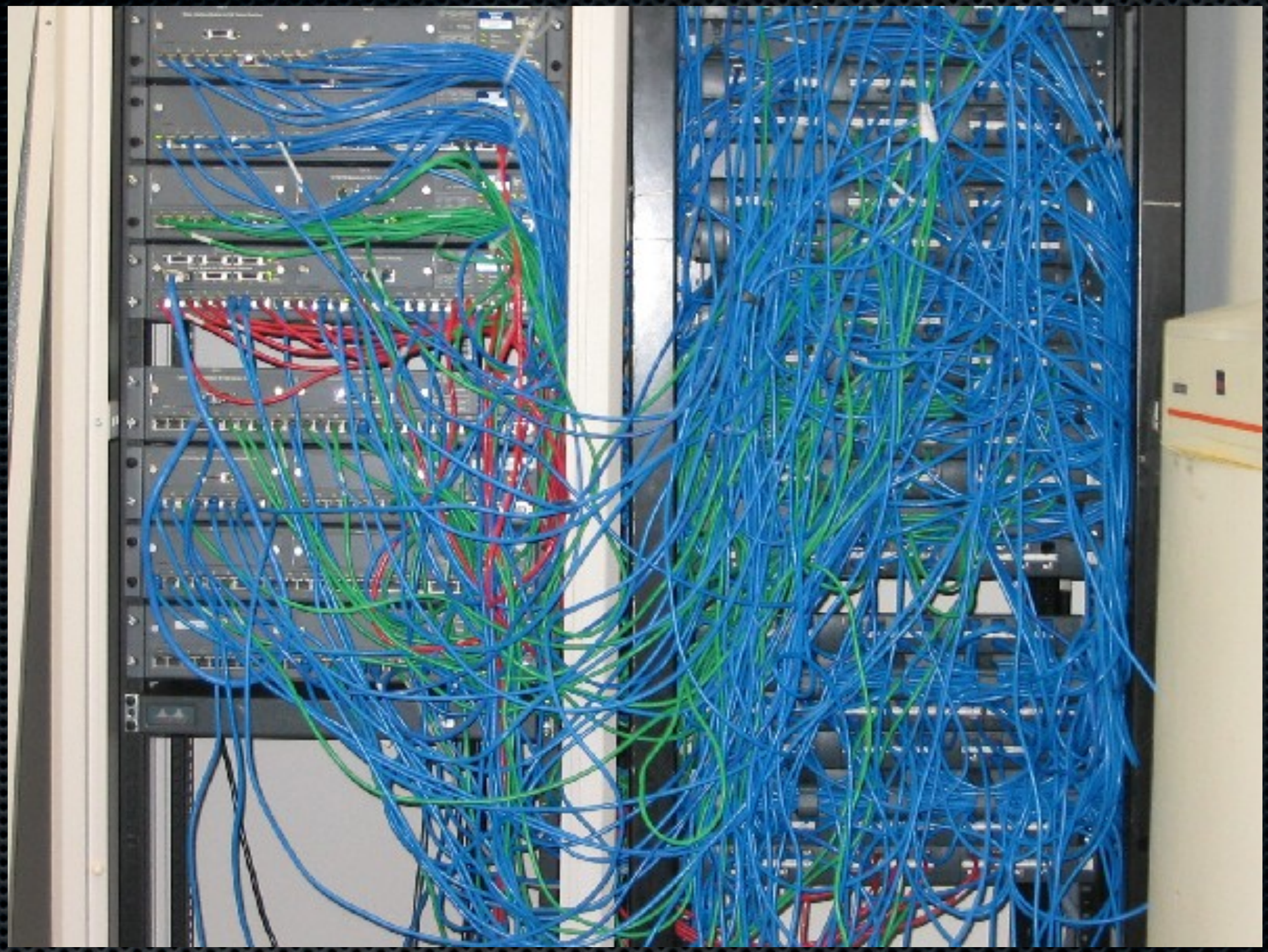
- basic rule: don't block the reactor
  - sleep(1) will block the reactor
  - use timers instead
- EM.add_timer or EM::Timer.new
- EM.add_periodic_timer or EM::PeriodicTimer.new

```
EM::PeriodicTimer.new(0.25) do
  sleep 1
end
```

timer won't fire every 0.25s

cancel timer so it never fires

```
timer = EM::Timer.new(5){
  puts 'wont happen'
}
timer.cancel
```



Managing events in the reactor

EM::Callback

- many ways to specify event handlers
  - operation{ do_something }
  - operation(proc{ do_something })
  - operation(&method(:do_something))
  - operation(obj.method(:do_something))

EM::Callback

- many ways to specify event handlers
  - operation{ do_something }
  - operation(proc{ do_something })
  - operation(&method(:do_something))
  - operation(obj.method(:do_something))

- EM::Callback supports a standardized interface
  - use it in your methods to support all of the above

```
def operation(*args, &blk)
  handler = EM::Callback(*args, &blk)
  handler.call
end
```

EM::Deferrable (!= EM.defer)

- represents an event as an object
- anyone can add procs to be invoked when the event succeeds or fails

EM::Deferrable (!= EM.defer)

- represents an event as an object
- anyone can add procs to be invoked when the event succeeds or fails
- you don’t have to worry about when or how the event gets triggered
- event might already have triggered: if you add a callback it will just get invoked right away

EM::Deferrable (!= EM.defer)

- represents an event as an object
- anyone can add procs to be invoked when the event succeeds or fails
- you don't have to worry about when or how the event gets triggered
  - event might already have triggered: if you add a callback it will just get invoked right away
- EM::Deferrable is a module, include in your own class

```
dfr = EM::DefaultDeferrable.new
dfr.callback{ puts 'success!' }
dfr.errback { puts 'failure!' }
dfr.succeed
```

```
dfr = http.get('/')
dfr.callback(&log_request)
dfr.callback(&save_to_file)
dfr.callback(&make_next_request)
```

EM::Queue

- async queue
- #pop does not return a value, takes a block instead
  - if the queue isn't empty, block is invoked right away
  - otherwise, it is invoked when someone calls #push

```
q = EM::Queue.new
q.pop{ |item| use(item) }
```

```
q = EM::Queue.new
processor = proc{ |item|
  use(item){
    q.pop(&processor)
  }
}
q.pop(&processor)
```

EM::Queue

- async queue
- #pop does not return a value, takes a block instead
  - if the queue isn't empty, block is invoked right away
  - otherwise, it is invoked when someone calls #push

```
q = EM::Queue.new
q.pop{ |item| use(item) }
```

"recursive" proc →

```
q = EM::Queue.new
processor = proc{ |item|
  use(item){
    q.pop(&processor)
  }
}
q.pop(&processor)
```

EM::Queue

- async queue
- #pop does not return a value, takes a block instead
  - if the queue isn't empty, block is invoked right away
  - otherwise, it is invoked when someone calls #push

```
q = EM::Queue.new
q.pop{ |item| use(item) }
```

"recursive" proc →

```
q = EM::Queue.new
processor = proc{ |item|
  use(item){
    q.pop(&processor)
  }
}
q.pop(&processor)
```

pop first item →

EM::Queue

- async queue
- #pop does not return a value, takes a block instead
  - if the queue isn't empty, block is invoked right away
  - otherwise, it is invoked when someone calls #push

```
q = EM::Queue.new
q.pop{ |item| use(item) }
```

"recursive" proc →

pop next item →

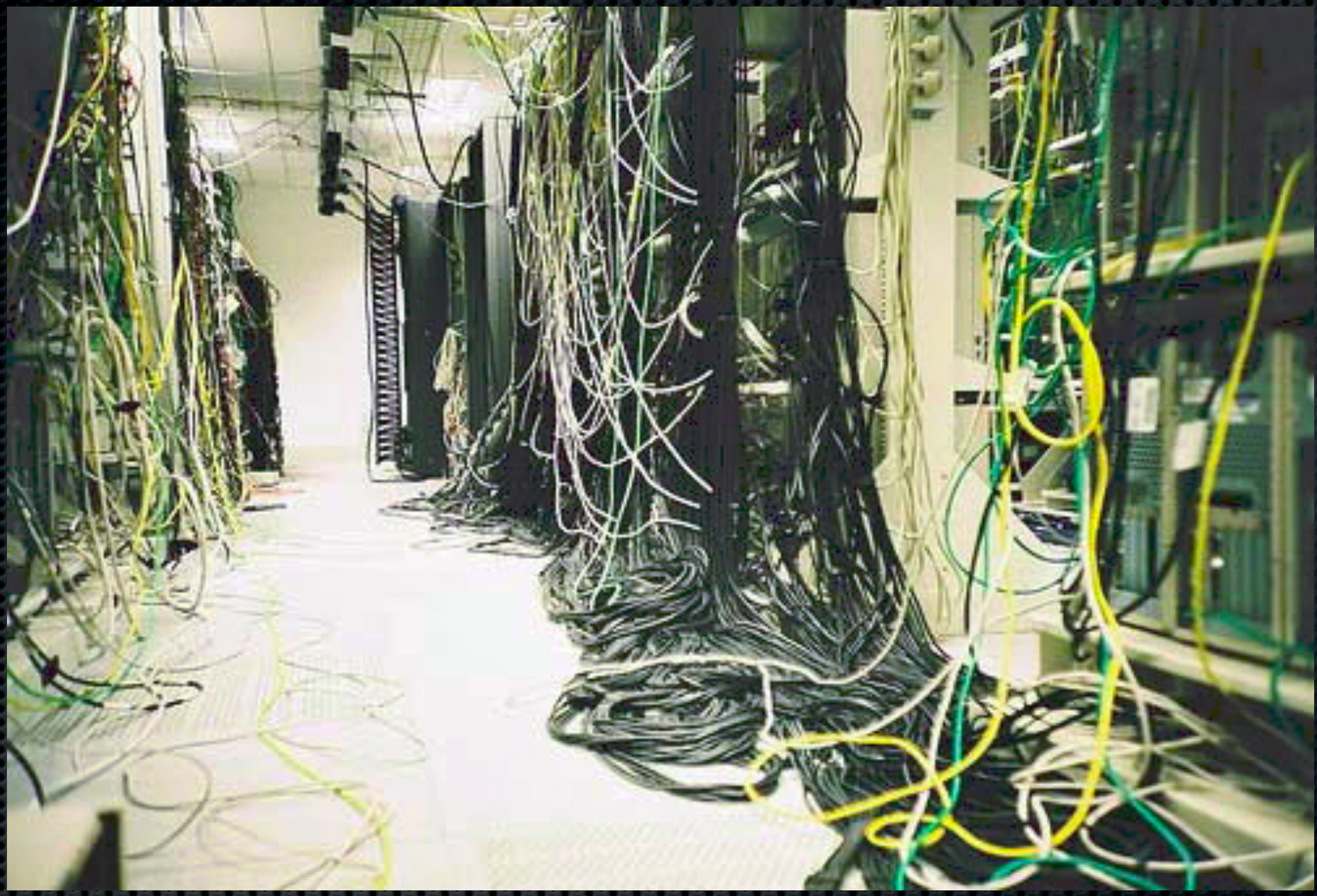
pop first item →

```
q = EM::Queue.new
processor = proc{ |item|
  use(item){
    q.pop(&processor)
  }
}
q.pop(&processor)
```

EM::Channel

- subscribers get a copy of each message published to the channel
- subscribe and unsubscribe at will

```
channel = EM::Channel.new
sid = channel.subscribe{ |msg|
  p [:got, msg]
}
channel.push('hello world')
channel.unsubscribe(sid)
```



Reacting with subprocesses

EM.system

- run external commands without blocking
- block receives stdout and exitstatus

```
EM.system('ls'){ |output,status|
  puts output if status.exitstatus == 0
}
```

EM.system

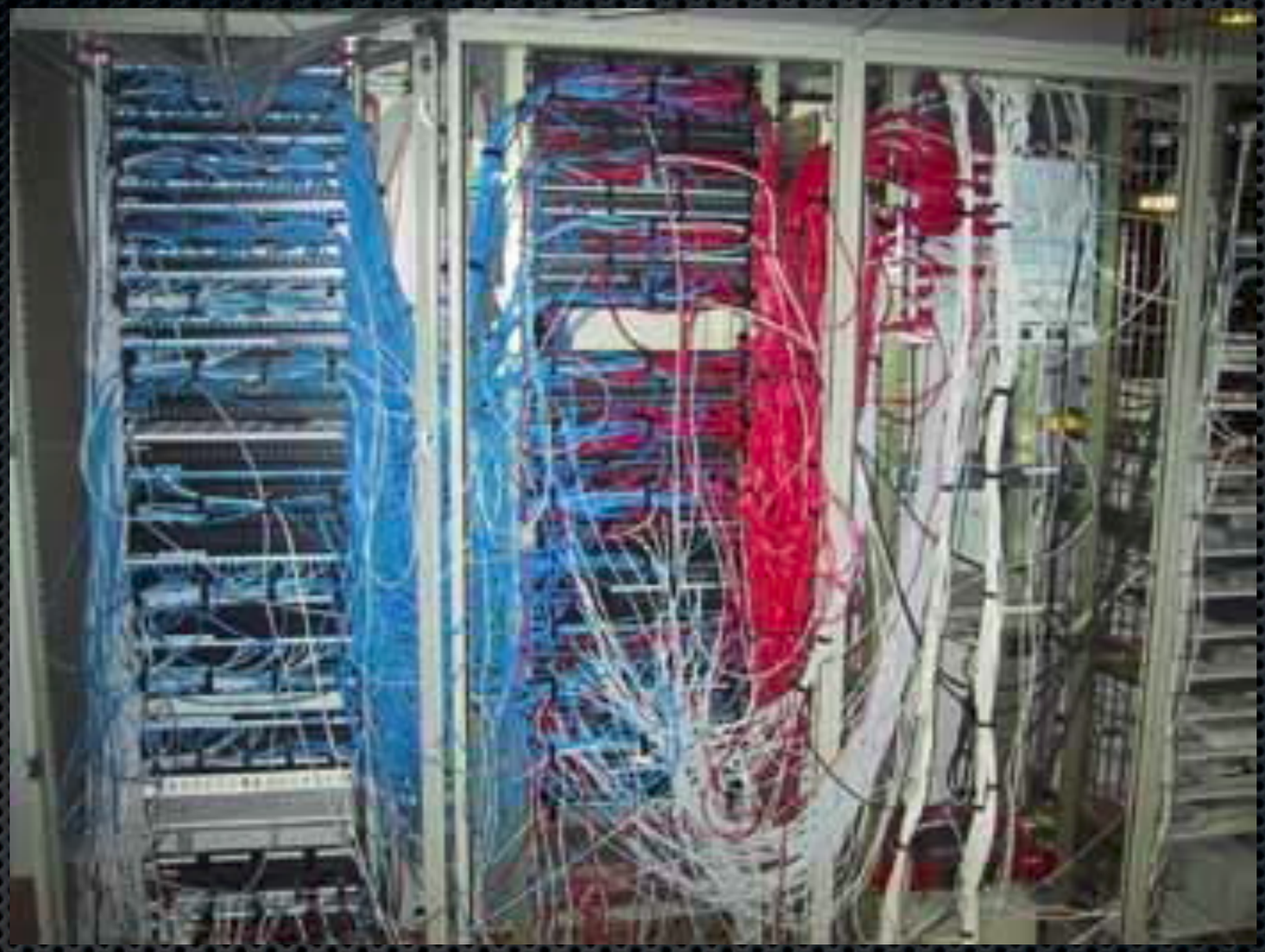
- run external commands without blocking
- block receives stdout and exitstatus

```
EM.system('ls'){ |output,status|
  puts output if status.exitstatus == 0
}
```

EM.popen

- lower level API used by EM.system
- takes a handler and streams stdout to your handler

```
EM.popen 'ls', LsHandler
```



Reacting with event handlers

EM Handlers

EM Handlers

- handler is a module or a class
- methods for each event (instead of creating and passing around procs)
- handler module/class is instantiated by the reactor
- use instance variables to keep state
- preferred way to write and organize EM code

EM Handlers

- handler is a module or a class
- methods for each event (instead of creating and passing around procs)
- handler module/class is instantiated by the reactor
- use instance variables to keep state
- preferred way to write and organize EM code

```
db.find_url { |url|
  http.get(url) { |response|
    email.send(response) {
      puts 'email sent'
    }
  }
}
```

EM Handlers

- handler is a module or a class
- methods for each event (instead of creating and passing around procs)

```
db.find_url { |url|
  http.get(url) { |response|
    email.send(response) {
      puts 'email sent'
    }
  }
}
```

- handler module/class is instantiated by the reactor
- use instance variables to keep state
- preferred way to write and organize EM code

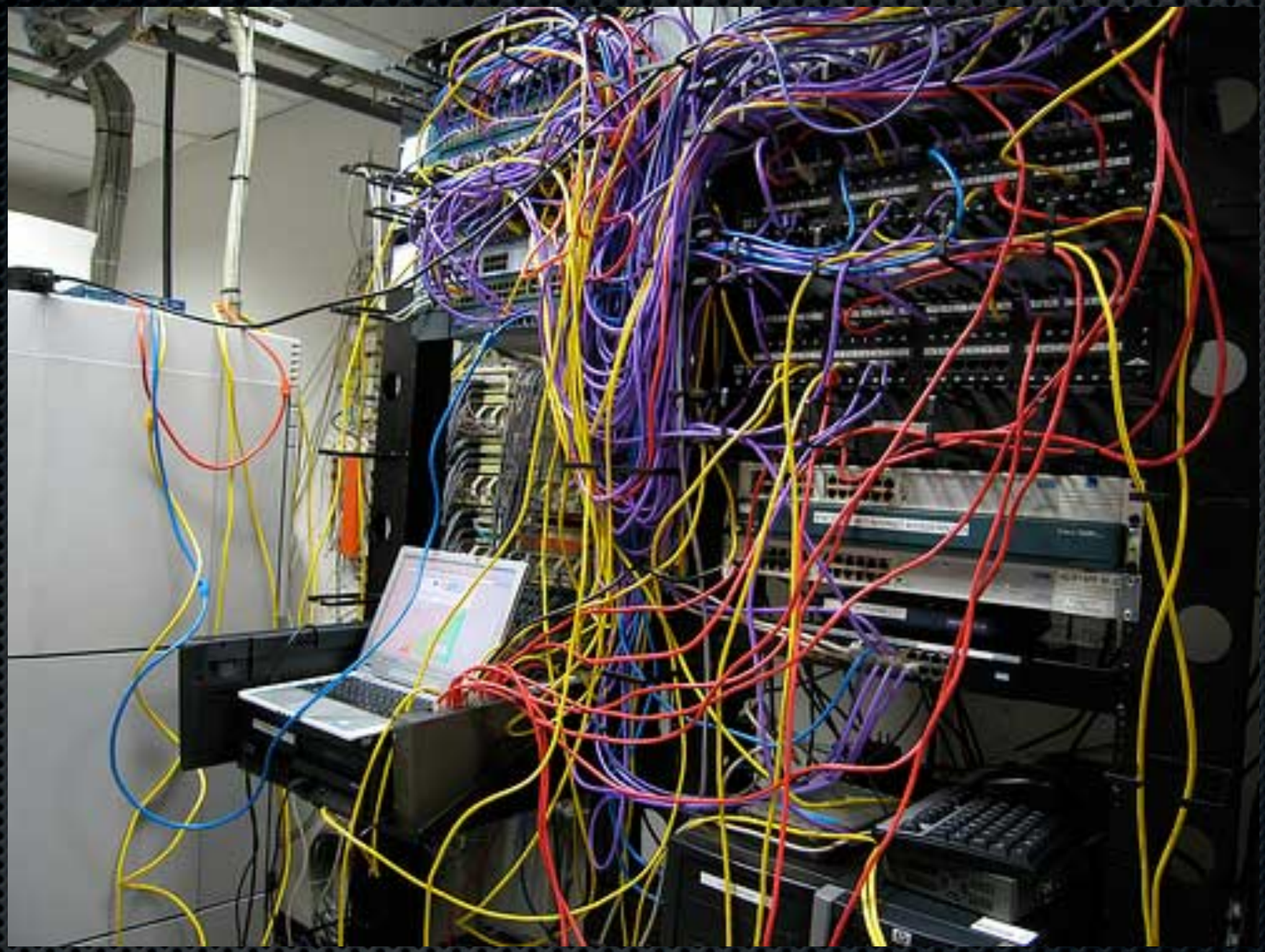
EM Handlers

- handler is a module or a class
- methods for each event (instead of creating and passing around procs)
- handler module/class is instantiated by the reactor
- use instance variables to keep state
- preferred way to write and organize EM code

```
db.find_url { |url|
  http.get(url) { |response|
    email.send(response) {
      puts 'email sent'
    }
  }
}
```



```
module Handler
  def db_find_url
    db.find_url(method(:http_get))
  end
  def http_get(url)
    http.get(url,
             method(:email_send))
  end
  def email_send(response)
    email.send(response,
               method(:email_sent))
  end
  def email_sent
    puts 'email sent'
  end
end
```



Networking with the reactor

Network Servers and Clients

- EM.start_server (and EM.stop_server) — TCP Server
- EM.connect (and EM.bind_connect) — TCP Client
- EM.open_datagram_socket — UDP Socket

Network Servers and Clients

- EM.start_server (and EM.stop_server) — TCP Server
- EM.connect (and EM.bind_connect) — TCP Client
- EM.open_datagram_socket — UDP Socket

```
server = EM.start_server '127.0.0.1', 8080, ClientHandler
EM.stop_server(server)

EM.connect '/tmp/mysql.sock', MysqlHandler
```

use host/port or unix domain sockets

Network Servers and Clients

- EM.start_server (and EM.stop_server) — TCP Server
- EM.connect (and EM.bind_connect) — TCP Client
- EM.open_datagram_socket — UDP Socket

one instance of ClientHandler per client that connects

```
server = EM.start_server '127.0.0.1', 8080, ClientHandler
EM.stop_server(server)

EM.connect '/tmp/mysql.sock', MysqlHandler
```

use host/port or unix domain sockets

EM::Connection

- events
  - post_init
  - connection_completed
  - receive_data
  - unbind
  - ssl_handshake_completed
- methods
  - start_tls
  - get_peername
  - send_data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data
  - unbind
  - ssl_handshake_completed
- methods
  - start_tls
  - get_peername
  - send_data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data ⟵ incoming data
  - unbind
  - ssl_handshake_completed
- methods
  - start_tls
  - get_peername
  - send_data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data ⟵ incoming data
  - unbind ⟵ connection closed
  - ssl_handshake_completed
- methods
  - start_tls
  - get_peername
  - send_data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data ⟵ incoming data
  - unbind ⟵ connection closed
  - ssl_handshake_completed
- methods
  - start_tls ⟵ built in ssl support
  - get_peername
  - send_data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data ⟵ incoming data
  - unbind ⟵ connection closed
  - ssl_handshake_completed
- methods
  - start_tls ⟵ built in ssl support
  - get_peername ⟵ get remote ip/port
  - send_data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data ⟵ incoming data
  - unbind ⟵ connection closed
  - ssl_handshake_completed
- methods
  - start_tls ⟵ built in ssl support
  - get_peername ⟵ get remote ip/port
  - send_data ⟵ buffer outgoing data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to
  - pause/resume

EM::Connection

- events
  - post_init
  - connection_completed ⟵ only for EM.connect
  - receive_data ⟵ incoming data
  - unbind ⟵ connection closed
  - ssl_handshake_completed
- methods
  - start_tls ⟵ built in ssl support
  - get_peername ⟵ get remote ip/port
  - send_data ⟵ buffer outgoing data
  - close_connection
  - close_connection_after_writing
  - proxy_incoming_to ⟵ lot more fancy features
  - pause/resume

- client.readline blocks until it receives a newline

```
require 'socket'
server = TCPServer.new(2202)
while client = server.accept
  msg = client.readline
  client.write "You said: #{msg}"
  client.close
end
```

```
module EchoHandler
  def post_init
    puts "-- someone connected"
    @buf = ''
  end
  def receive_data(data)
    @buf << data
    while line = @buf.slice!(/(.+)\r?\n/)
      send_data "You said: #{line}\n"
    end
  end
  def unbind
    puts "-- someone disconnected"
  end
end

EM.run{
  EM.start_server(
    '0.0.0.0', 2202, EchoHander
  )
}
```

- client.readline blocks until it receives a newline
- with non-blocking i/o, you get whatever data is available

```
require 'socket'
server = TCPServer.new(2202)
while client = server.accept
  msg = client.readline
  client.write "You said: #{msg}"
  client.close
end
```

```
module EchoHandler
  def post_init
    puts "-- someone connected"
    @buf = ''
  end
  def receive_data(data)
    @buf << data
    while line = @buf.slice!(/(.+)\r?\n/)
      send_data "You said: #{line}\n"
    end
  end
  def unbind
    puts "-- someone disconnected"
  end
end

EM.run{
  EM.start_server(
    '0.0.0.0', 2202, EchoHander
  )
}
```

- client.readline blocks until it receives a newline
- with non-blocking i/o, you get whatever data is available
- **must** verify that a full line was received

```
require 'socket'
server = TCPServer.new(2202)
while client = server.accept
  msg = client.readline
  client.write "You said: #{msg}"
  client.close
end
```

```
module EchoHandler
  def post_init
    puts "-- someone connected"
    @buf = ''
  end
  def receive_data(data)
    @buf << data
    while line = @buf.slice!(/(.+)\r?\n/)
      send_data "You said: #{line}\n"
    end
  end
  def unbind
    puts "-- someone disconnected"
  end
end

EM.run{
  EM.start_server(
    '0.0.0.0', 2202, EchoHander
  )
}
```

- client.readline blocks until it receives a newline
- with non-blocking i/o, you get whatever data is available
- **must** verify that a full line was received
- TCP is a **stream**

```
require 'socket'
server = TCPServer.new(2202)
while client = server.accept
  msg = client.readline
  client.write "You said: #{msg}"
  client.close
end
```

TCP is a Stream

TCP is a Stream

send_data(“hello”)
send_data(“world”)

TCP is a Stream

send_data(“hello”)
send_data(“world”)

TCP is a Stream

send_data(“hello”)
send_data(“world”)

receive_data(“he”)
receive_data(“llowo”)
receive_data(“rld”)

TCP is a Stream

```
def receive_data(data)
  send_data "You said: #{data}"
end
```

send_data(“hello”)
send_data(“world”)

receive_data(“he”)
receive_data(“llowo”)
receive_data(“rld”)

TCP is a Stream

```ruby
def receive_data(data)
  send_data "You said: #{data}"
end
```

send_data(“hello”)
send_data(“world”)

network

receive_data(“he”)
receive_data(“llowo”)
receive_data(“rld”)

```ruby
def receive_data(data)
  (@buf ||= '') << data
  if line = @buf.slice!(/(.+)\r?\n/)
    send_data "You said: #{line}"
  end
end
```

TCP is a Stream

```
def receive_data(data)
  send_data "You said: #{data}"
end
```

send_data(“hello”)
send_data(“world”)

network

receive_data(“he”)
receive_data(“llowo”)
receive_data(“rld”)

```
def receive_data(data)
  (@buf ||= '') << data
  if line = @buf.slice!(/(.+)\r?\n/)
    send_data "You said: #{line}"
  end
end
```

append to buffer

TCP is a Stream

```
def receive_data(data)
  send_data "You said: #{data}"
end
```

send_data(“hello”)
send_data(“world”)

network

receive_data(“he”)
receive_data(“llowo”)
receive_data(“rld”)

```
def receive_data(data)
  (@buf ||= '') << data
  if line = @buf.slice!(/(.+)\r?\n/)
    send_data "You said: #{line}"
  end
end
```

← append to buffer
← parse out lines

TCP is a Stream

```
def receive_data(data)
  send_data "You said: #{data}"
end
```

send_data(“hello”)
send_data(“world”)

network

receive_data(“he”)
receive_data(“llowo”)
receive_data(“rld”)

```
def receive_data(data)
  (@buf ||= '') << data
  if line = @buf.slice!(/(.+)\r?\n/)
    send_data "You said: #{line}"
  end
end
```

← append to buffer
← parse out lines

```
def receive_data(data)
  @buf ||= BufferedTokenizer.new("\n")
  @buf.extract(data).each do |line|
    send_data "You said: #{line}"
  end
end
```

← included in EM

EM::Protocols

```
module RubyServer
  include EM::P::ObjectProtocol

  def receive_object(obj)
   send_object({'you said' => obj})
  end
end
```

EM::Protocols

```
module RubyServer
  include EM::P::ObjectProtocol

  def receive_object(obj)
   send_object({'you said' => obj})
  end
end
```

ObjectProtocol
defines receive_data

EM::Protocols

```ruby
module RubyServer
  include EM::P::ObjectProtocol

  def receive_object(obj)
   send_object({'you said' => obj})
  end
end
```

ObjectProtocol defines receive_data

```ruby
def receive_data data
  (@buf ||= '') << data

  while @buf.size >= 4
    if @buf.size >= 4+(size=@buf.unpack('N').first)
      @buf.slice!(0,4)
      receive_object Marshal.load(@buf.slice!(0,size))
```

EM::Protocols

```ruby
module RubyServer
  include EM::P::ObjectProtocol

  def receive_object(obj)
   send_object({'you said' => obj})
  end
end
```

ObjectProtocol defines receive_data

```ruby
def receive_data data
  (@buf ||= '') << data

  while @buf.size >= 4
    if @buf.size >= 4+(size=@buf.unpack('N').first)
      @buf.slice!(0,4)
      receive_object Marshal.load(@buf.slice!(0,size))
```

EM::Protocols

```ruby
module RubyServer
  include EM::P::ObjectProtocol

  def receive_object(obj)
   send_object({'you said' => obj})
  end
end
```

ObjectProtocol defines receive_data

```ruby
def receive_data data
  (@buf ||= '') << data

  while @buf.size >= 4
    if @buf.size >= 4+(size=@buf.unpack('N').first)
      @buf.slice!(0,4)
      receive_object Marshal.load(@buf.slice!(0,size))
```

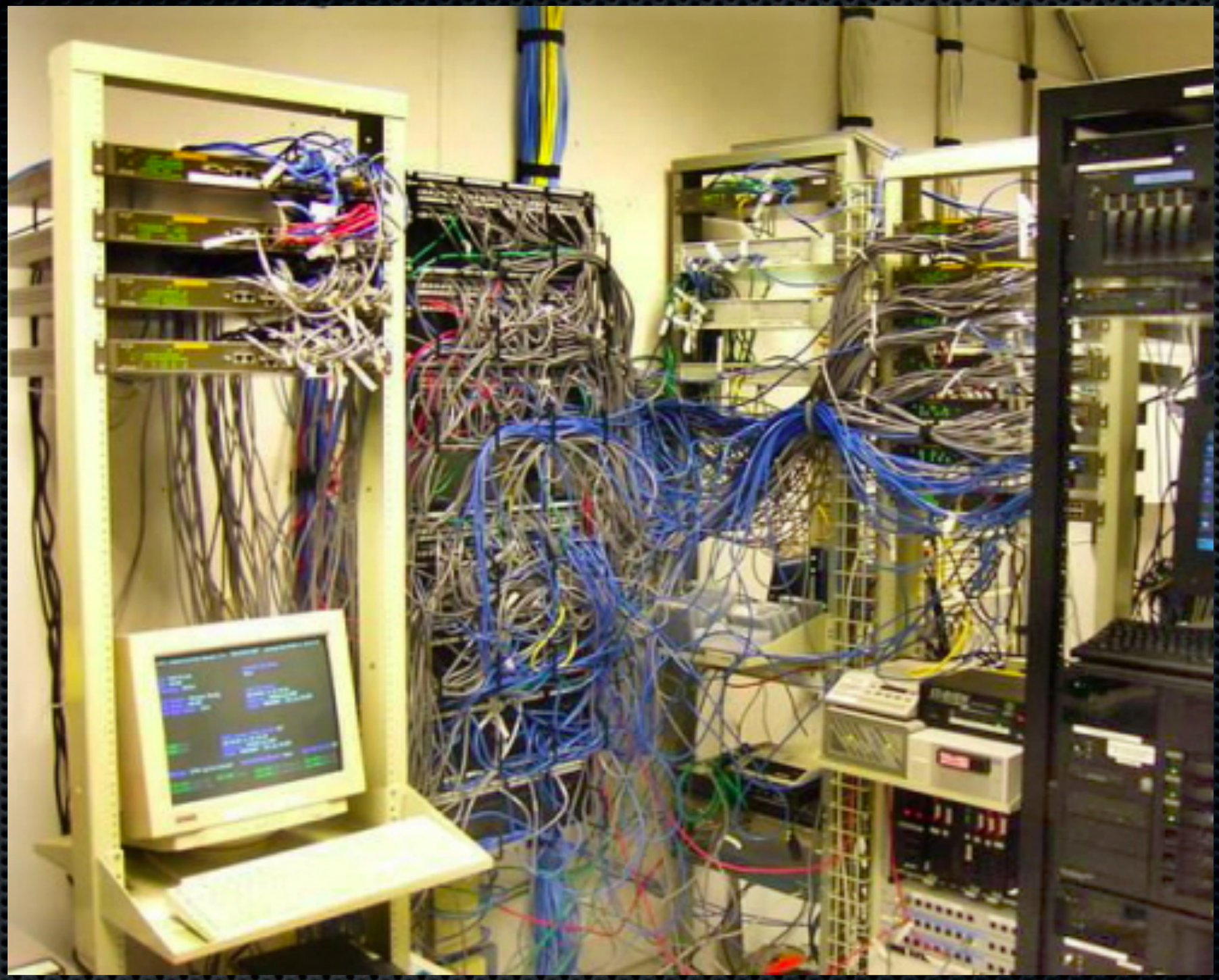
More EM::Protocols

- Built-In
  - Memcache
  - SmtpClient
  - SmtpServer
  - Stomp

http://eventmachine.rubyforge.org/EventMachine/Protocols.html

- External
  - HttpRequest
  - Redis
  - CouchDB
  - WebSocket
  - Cassandra

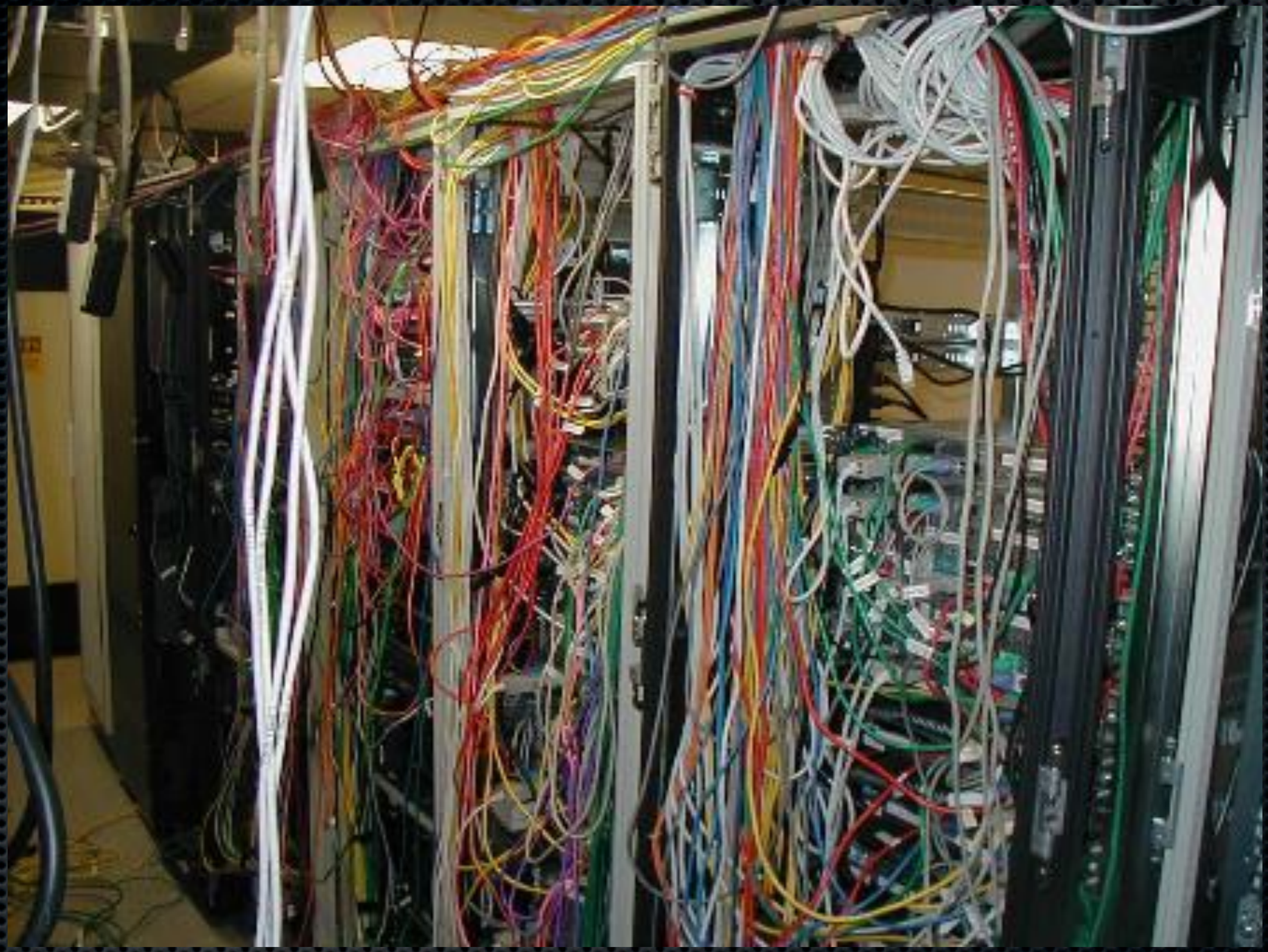
http://wiki.github.com/eventmachine/eventmachine/protocol-implementations



Doing more with the reactor

Other EM Features

- EM.watch and EM.attach
  - use external file descriptors with the reactor
- EM.watch_file
  - watch files and directories for changes
  - events: file_modified, file_deleted, file_moved
- EM.watch_process
  - watch processes
  - events: process_forked, process_exited
- EM.open_keyboard
  - receive_data from stdin



Using the reactor

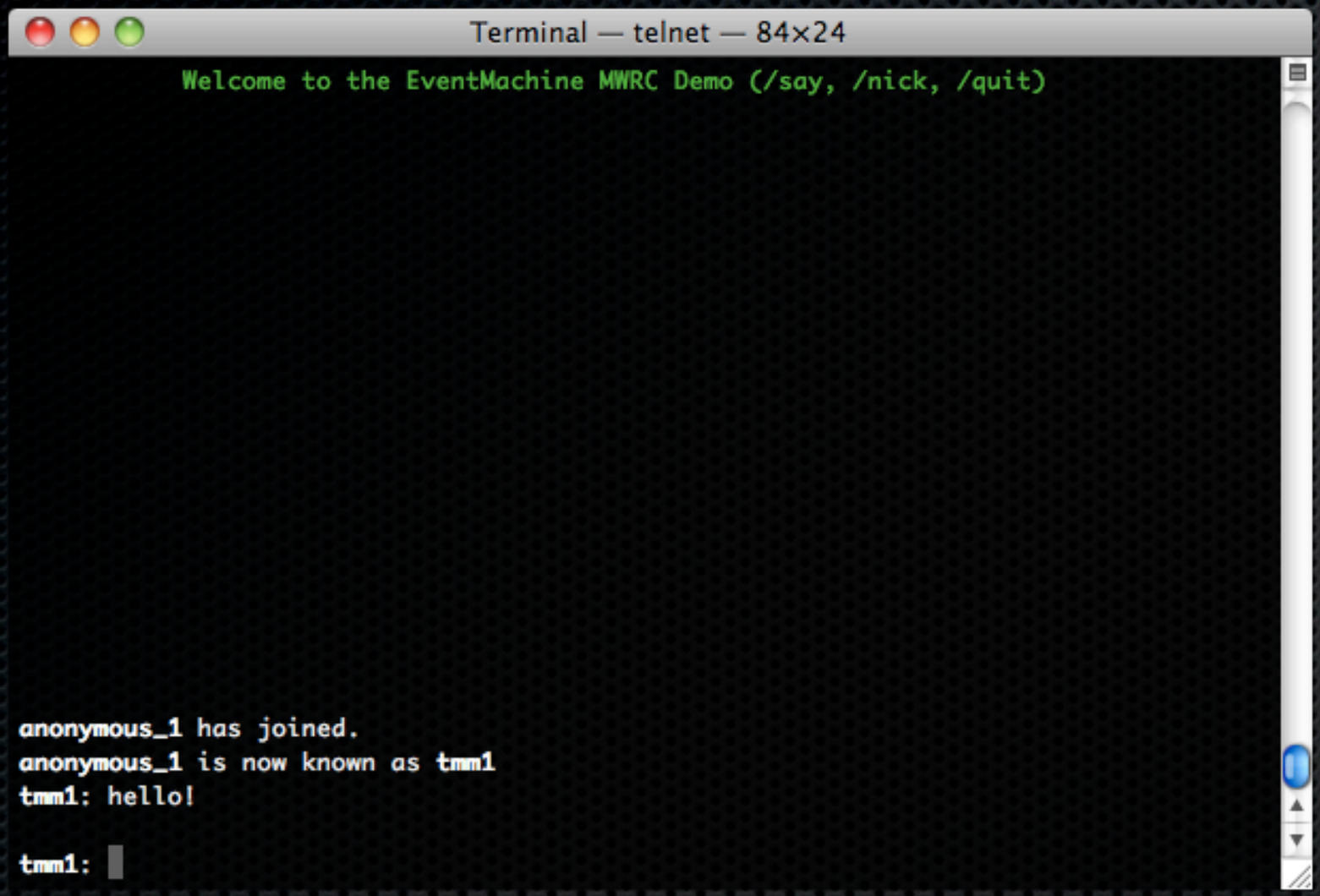
Using EM in your webapp

- Run EM in a thread
- Use an EM enabled webserver
  - Thin or Rainbows!
  - call EM API methods directly from your actions
- Use async_sinatra for streaming and delayed responses
  - http://github.com/raggi/async_sinatra

```
aget '/delay/:n' do |n|
  EM.add_timer(n.to_i){
    body "delayed for #{n}s"
  }
end
```

```
aget '/proxy' do
  url = params[:url]
  http = EM::HttpRequest.new(url).get
  http.callback{ |response|
    body response.body
  }
end
```

DEMO



telnet 10.0.123.89 1337

http://gist.github.com/329682

- simple line based chat server
- EM::Channel to distribute chat messages
- /say uses EM.system and EM::Queue to invoke `say` on the server
- EM::HttpRequest and Yajl to include tweets in the chat

More Information

- Homepage: http://rubyeventmachine.com
- Group: http://groups.google.com/group/eventmachine
- RDoc: http://eventmachine.rubyforge.org
- Github: http://github.com/eventmachine/eventmachine
- IRC: #eventmachine on irc.freenode.net
- These slides: http://timetobleed.com

Questions?

@tmm1

github.com/tmm1