How happy they became with H2O/mruby and the future of HTTP

@i110, @kazuho

About Me

- Ichito Nagata
- Software Engineer at Fastly
- Full-time H2O committer
- Ruby, C, Perl, Python, JavaScript
- Splatoon2 (S+0~3)

Introduction

1 year ago..

- "How happy we can become with H2O/mruby"
  - YAPC::Kansai
  - https://www.slideshare.net/ichitonagata/h2o-x-mruby-72949986

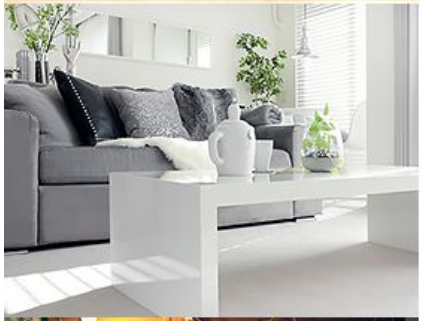
- Talked about the potential of mruby in web servers
- I was searching the opportunity to introduce H2O/mruby
  - in large scale service..

Fish on!

Trans: “Sounds great, wanna say goodbye to our demoniac nginx config..”

RoomClip

- Room photo sharing service largest in Japan
  - CEO and CTO are my college classmates


RoomClip

日本最大のインテリア専門SNS



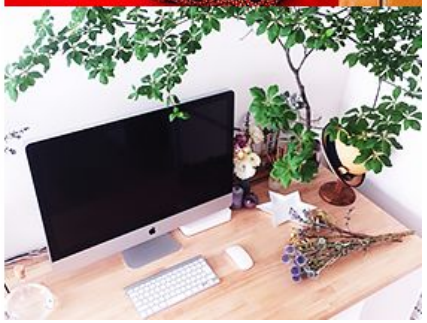
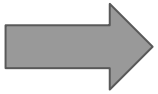
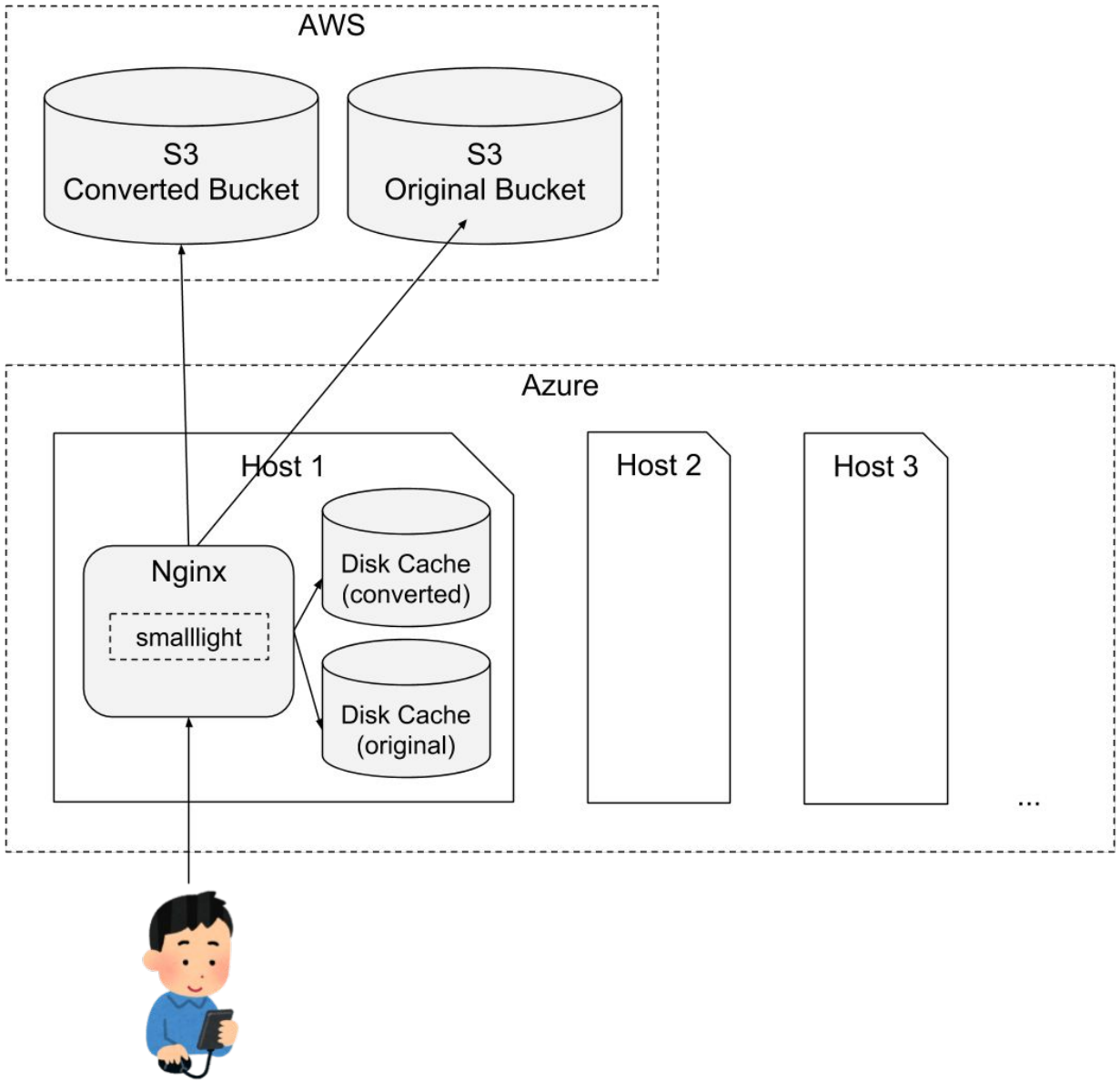


Image Resize and Mask

- Downsize to arbitrary (!) size keeping aspect ratio
- Apply unsharp mask for aesthetic appearance



Architecture



Their Nginx Config: The Hellfire

- Too hard to decipher..
- Also can be a hotbed of many bugs
- And what is worse: Security Issues!

```
    location ~ ^/@sl/v1/(.+?)/(.+)/(.+/?)+$ {
      set $size $1;
      set $bucket $2;
      set $file $3;
      set $engine "imagemagick";

      # $size must be int
      if ($size !~ ^[0-9]+$) {
        rewrite ^ /404.html;
      }

      # NEED $size < 2000
      if ($size ~ "^[2-9][0-9]{2,}[0-9]$") {
        rewrite ^ /404.html;
      }

      # NEED $size < 1800
      if ($size ~ "^1[8-9][0-9][0-9]$") {
        rewrite ^ /404.html;
      }

      # $size < 640 then base load is img_640
      set $using_short_size NG;
      set $short_bucket roomclip-bucket;
      if ($size ~ "^[0-6]?[0-4][0-9]$") {
        set $using_short_size O;
      }
      if ($bucket ~
"^(roomclip-bucket(.*?)\/)img_[0-9].*$"){
        set $using_short_size
"${using_short_size}K";
        set $short_bucket "$1";
      }
```

Example 1: Bug of Size Restriction

- Converting the size of images with the URL like:
  - https://img.roomclip.jp/v1/{{dst}}/img_{{src}}/deadbeaf.jpg

- But They wanted to set an upper bound of {{dst}}

Their Answer

```
# NEED $dst_size < 2000
if ($dst_size ~ "^[2-9][0-9]{2,}[0-9]$") {
  rewrite ^ /404.html;
}
```

Try 10000000 □

Example 2: Wasting Use of Original Images

- There may be 7 pre-downsized original images
  - 90, 180, 320, 640, 750, 1242, 1536
  - in local cache and S3

- We should select the minimum original image
  - If 100 is requested, 180 should be elected

Their Answer

```
# $size < 640 then base load is img_640
set $using_short_size NG;
set $short_bucket roomclip-bucket;
if ($size ~ "^[0-6]?[0-4][0-9]$") {
  set $using_short_size O;
}
if ($bucket ~ "^(roomclip-bucket(.*?)\/)img_[0-9].*$")
{
  set $using_short_size "${using_short_size}K";
  set $short_bucket "$1";
}
```

640 or 1536. Have your choice

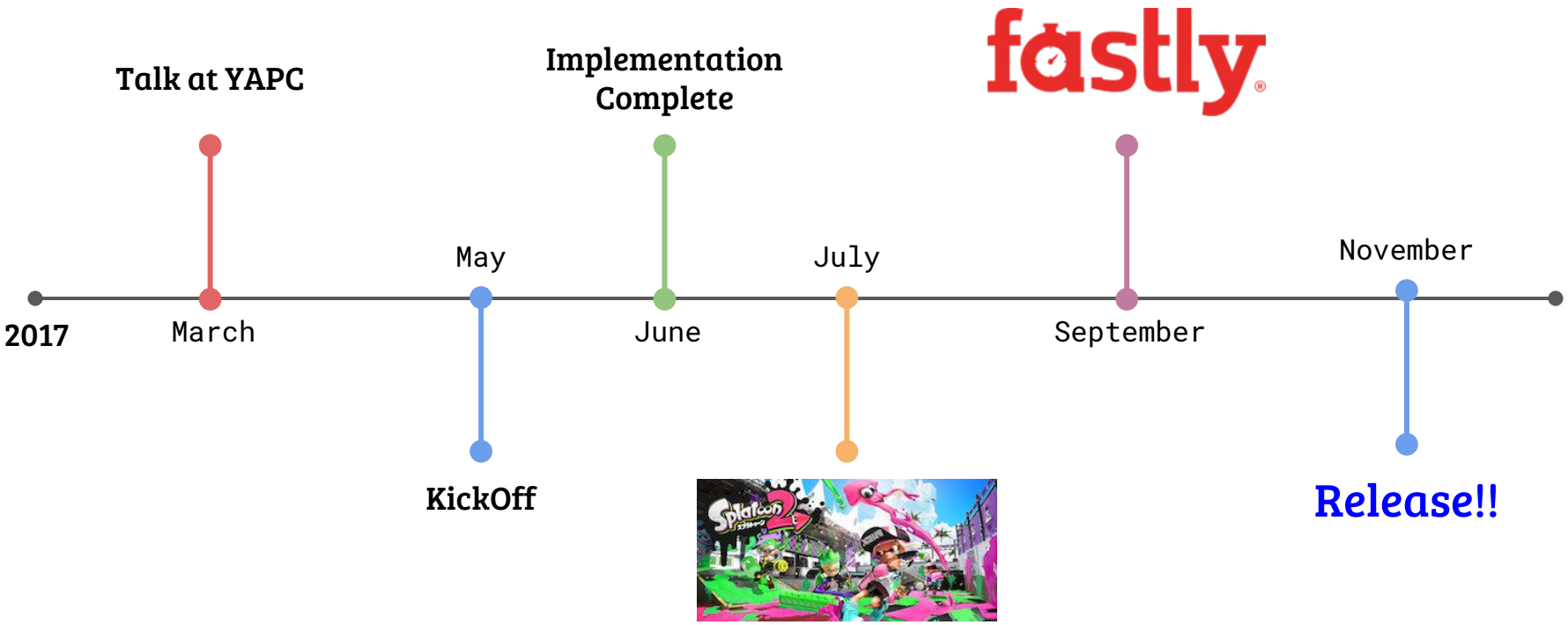
Sources of Problems

1. Nginx configuration is not a "Programming Language"
    a. Hard to do complex things
2. Lack of Debuggability and Testability
    a. All we can do is send actual requests

Introducing H2O into RoomClip

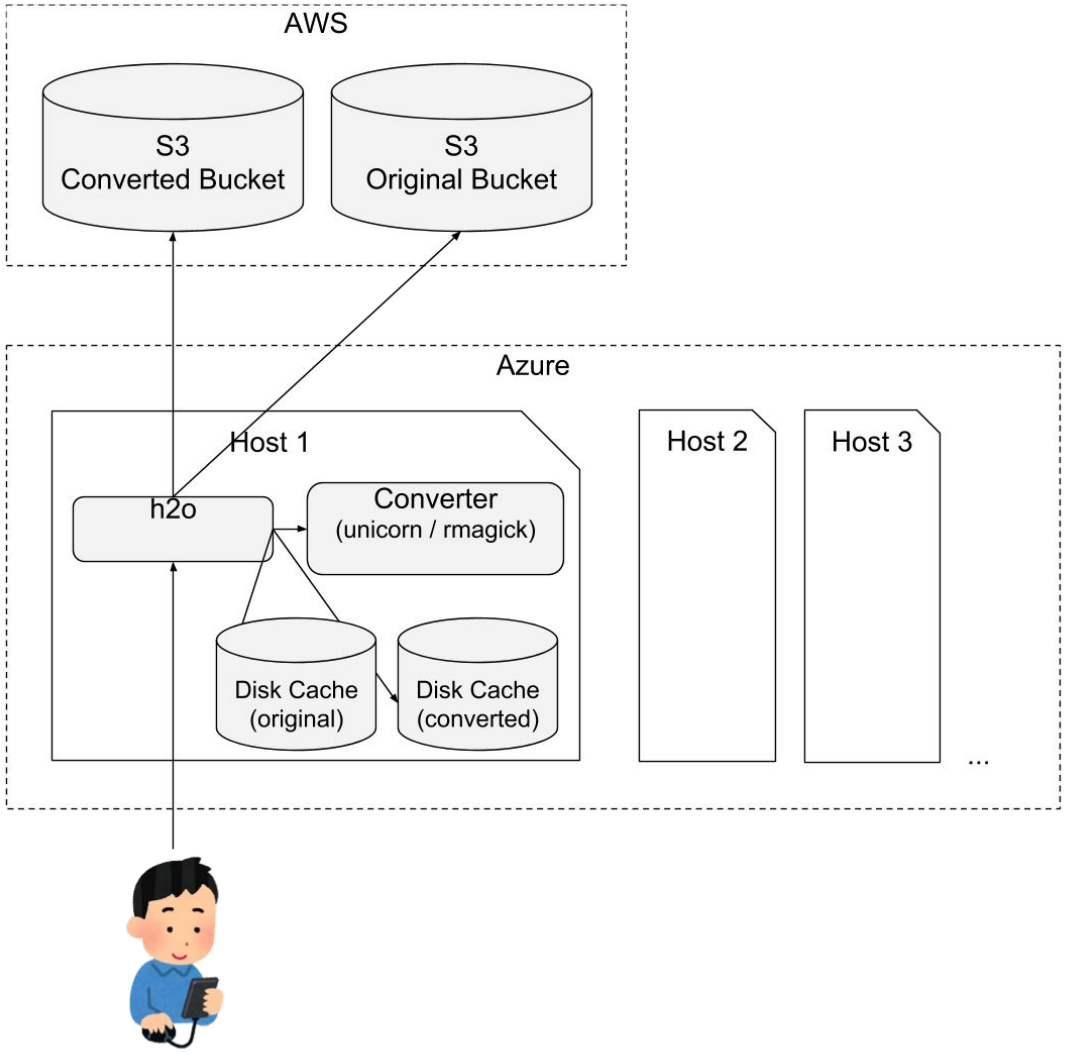
Timeline

- Actually it took 1~2 months to finish



New Architecture

- The only change was detaching image converting process from web server



H2O Configuration

- Almost all requests are handled by mruby (app.rb)

```
<<: !file {{ h2o_etc_dir }}/common.conf
error-doc:
  - status: [403, 404]
    url: /notfound.jpg
  - status: [500, 502, 503, 504]
    url: /notfound.jpg
hosts:
  img.roomclip.jp:
    paths:
      /notfound.jpg:
        file.file: "{{ h2o_docroot_dir}}/notfound.jpg"
      /favicon.ico:
        file.file: "{{ h2o_docroot_dir}}/favicon.ico"
      /robots.txt:
        file.file: "{{ h2o_docroot_dir}}/robots.txt"
      /:
        mruby.handler-file: {{ h2o_app_dir }}/app.rb
```

H2O Configuration - app.rb

- Simply build a Rackapp

```
$LOAD_PATH << File.expand_path('../lib', __FILE__)

require 'roomclip'
RoomClip.build_app({
  :s3_origin      => '{{ s3_origin }}',
  :convert_origin => '{{ convert_origin }}',
  :cache_dir      => '{{ h2o_cache_dir }}',
})
```

H2O Configuration - roomclip.rb (1)

```
def self.build_app(conf)
  Rack::Builder.new {
    use Rack::Reprocessable
    map '/v1' do
      use RoomClip::SourceSizeSelector
      run Rack::Cascade.new([
        Rack::Reprocess.new {|env| "/s3-converted#{env[PATH_INFO]}" },
        Rack::Builder.new {
          use Rack::HTTPCache, storage, cache_option
          use RoomClip::Converter, conf[:convert_origin]
          run Rack::Reprocess.new {|env| "/s3-original#{env[PATH_INFO]}" },
        }.to_app,
      ], [403, 404])
    end
    ['/s3-original', '/s3-converted'].each {|path|
      map path do
        use Rack::HTTPCache, storage, cache_option
        run RoomClip::S3Proxy.new(conf[:s3_origin])
      end
    }
  }.to_app
end
```

H2O Configuration - roomclip.rb (2)

Module	**Description**
`RoomClip::SourceSizeSelector`	**Detect the best size of original image and rewrite** `PATH_INFO`
`RoomClip::Converter`	**Send POST request to Converter process**
`RoomClip::S3Proxy`	**Send GET requests to S3**

H2O Configuration - roomclip.rb (3)

```ruby
class SourceSizeSelector
  SOURCE_SIZES = [90, 180, 320, 640, 750, 1242, 1536]

  def initialize(app)
    @app = app
  end

  def call(env)
    m = env['PATH_INFO'].match(%r{^/v1/(\d+)/img_\d+/(.+)$})
    size = m[1].to_i

    best = SOURCE_SIZES.find {|ss| size <= ss } || SOURCE_SIZES.last
    size = [size, best].min
    env['PATH_INFO'] = "/#{size}/img_#{best}/#{m[2]}"

    @app.call(env)
  end
end
```

- Select best {{src}} (original image size) using {{dst}}
  - https://img.roomclip.jp/v1/{{dst}}/img_{{src}}/deadbeaf.jpg
- Rewrite {{src}} in PATH_INFO

H2O Configuration - roomclip.rb (4)

- Dispatch (@app.call)
- Send a request to Converter processes using POST with body payload
- Some header tweaks

```
class Converter

  def convert(body, dst_size)
    url = "#{@origin}/?sizew=#{dst_size}"
    http_request(url, { :body => body }).join
  end

  def call(env)
    status, headers, body = @app.call(env)
    return [status, headers, body] unless status == 200

    c_status, _, c_body = convert(body, get_dest_size(env))
    unless c_status == 200
      env['rack.errors'].puts "failed to convert image"
      return [500, {}, []]
    end

    headers = Rack::Utils::HeaderHash.new(headers)
    headers['Content-Length'] = c_headers['Content-Length']
    ['Date', 'ETag', 'Last-Modified', 'Age'].
      each {|name| headers.delete(name) }

    [status, headers, c_body]
  end
end
```

H2O Configuration - roomclip.rb (5)

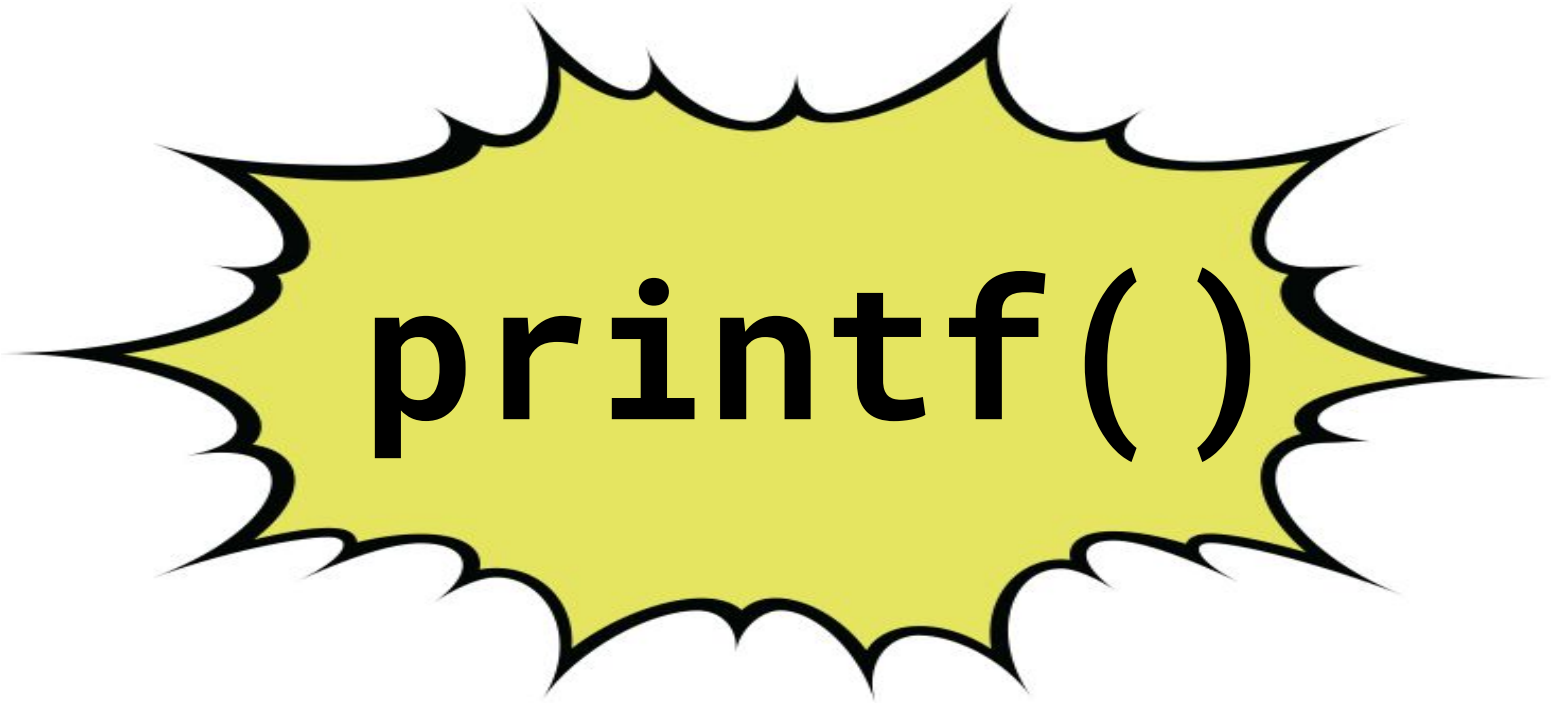
```
class S3Proxy

  def call(env)
    url = "#{@origin}#{env['PATH_INFO']}"
    http_request(url, { :headers => env_to_headers(env) }).join
  end

end
```

- **Proxy to S3 using `http_request` (built-in h2o method)**

Debuggability

Debuggability

- Basically web servers are Black Box
  - What's happening in the processes?
  - What's the value of this variable?
- In Programming Languages:



Debuggability - p as you like :)

```
paths:
  /:
    mruby.handler: |
      proc {|env|
        p env
        [200, {}, ["hello"]]
      }
```

- In H2O error log:

{"REQUEST_METHOD"=>"GET", "SCRIPT_NAME"=>"", "PATH_INFO"=>"/", "QUERY_STRING"=>"", "SERVER_NAME"=>"127.0.0.1", "SERVER_PROTOCOL"=>"HTTP/1.1", "SERVER_ADDR"=>"127.0.0.1", "SERVER_PORT"=>"8080", "HTTP_HOST"=>"127.0.0.1:8080", "REMOTE_ADDR"=>"127.0.0.1", "REMOTE_PORT"=>"49412", "HTTP_ACCEPT"=>"*/*", "HTTP_USER_AGENT"=>"curl/7.59.0", "rack.url_scheme"=>"http", "rack.multithread"=>false, "rack.multiprocess"=>true, "rack.run_once"=>false, "rack.hijack?"=>false, "rack.errors"=>#<H2O::ErrorStream:0x103049ee8>, "SERVER_SOFTWARE"=>"h2o/2.3.0-DEV", "h2o.remaining_delegations"=>5, "h2o.remaining_reprocesses"=>5}

Debuggability - takes your own profiles ;)

```
paths:
  /:
    mruby.handler: |
      proc {|env|
        start_at = Time.now
        resp = http_request("http://example.com").join
        puts "elapsed: #{Time.now - start_at}"
        resp
      }
```

- In H2O error log:

  elapsed: 0.38455

- (of cause there is the way to write it to access log)

Testability

Testability

- How are you testing your web server?
  - Especially when the configuration is changed
- Are you sending actual requests?

Testability - write unit tests in mruby :D

- You can write unit tests of handlers
  - https://github.com/iij/mruby-mtest
  - That's it
- Pitfalls
  - Make sure to use the same version of mruby
    - ${H2O_BUILD_DIR}/mruby/host/bin/mruby
  - Some built-in methods and classes of H2O cannnot be used :(
    - Create your own test doubles
    - Currently considering h2o mruby test mode
      - Something like `h2o --mode mrubytest handler.rb`

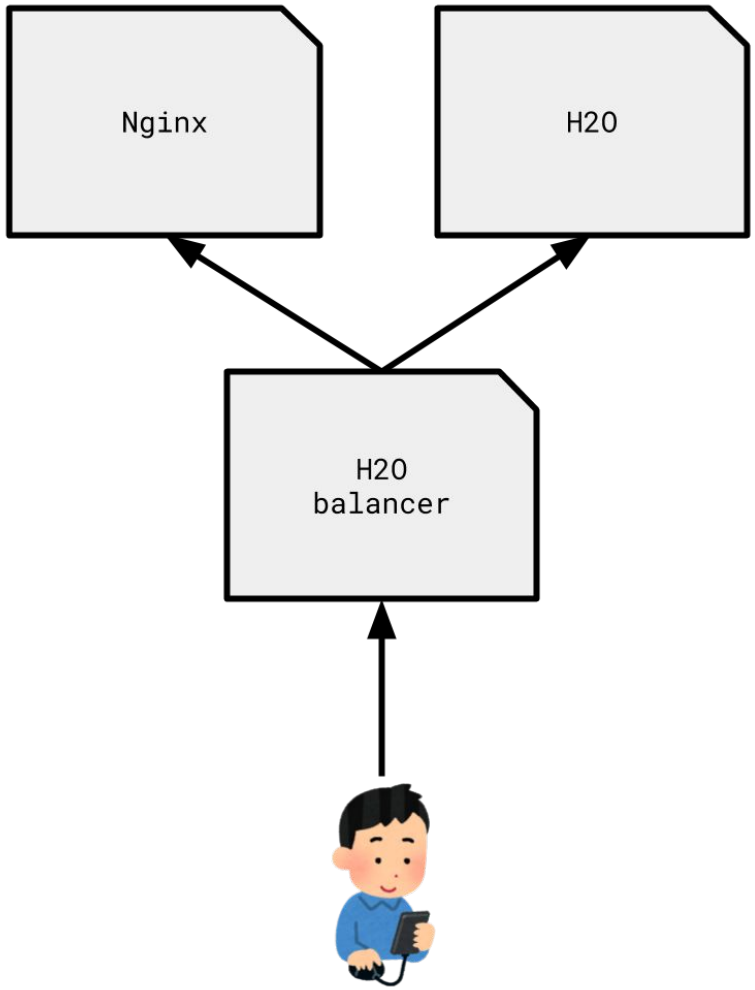
Deploy to Production

Deploy to Production (1)

- Wanted H2O to handle requests eventually
- Balancing the requests between Nginx and H2O

Deploy to Production (2)

- Use H2O as a load balancer



Deploy to Production (3)

- Eventually dispatch the requests to H2O

```
mruby.handler: |
  STARTED_AT = Time.now
  DURATION = {{ balance_duration }} # seconds
  H2O_HOST = '{{ balance_h2o_host }}'
  NGINX_HOST = '{{ balance_nginx_host }}'

  proc {|env|
    Elapsed = Time.now - STARTED_AT
    p = elapsed / DURATION
    host = rand() < p ? H2O_HOST : NGINX_HOST

    url = "#{ env['rack.url_scheme'] }://#{ host }#{ env['PATH_INFO'] }"
    req_headers = env_to_headers(env) # convert HTTP_FOO_BAR into foo-bar

    status, headers, body = http_request(url,
      { :method => env['REQUEST_METHOD'], :headers => req_headers }).join

    [status, headers, body]
  }
```

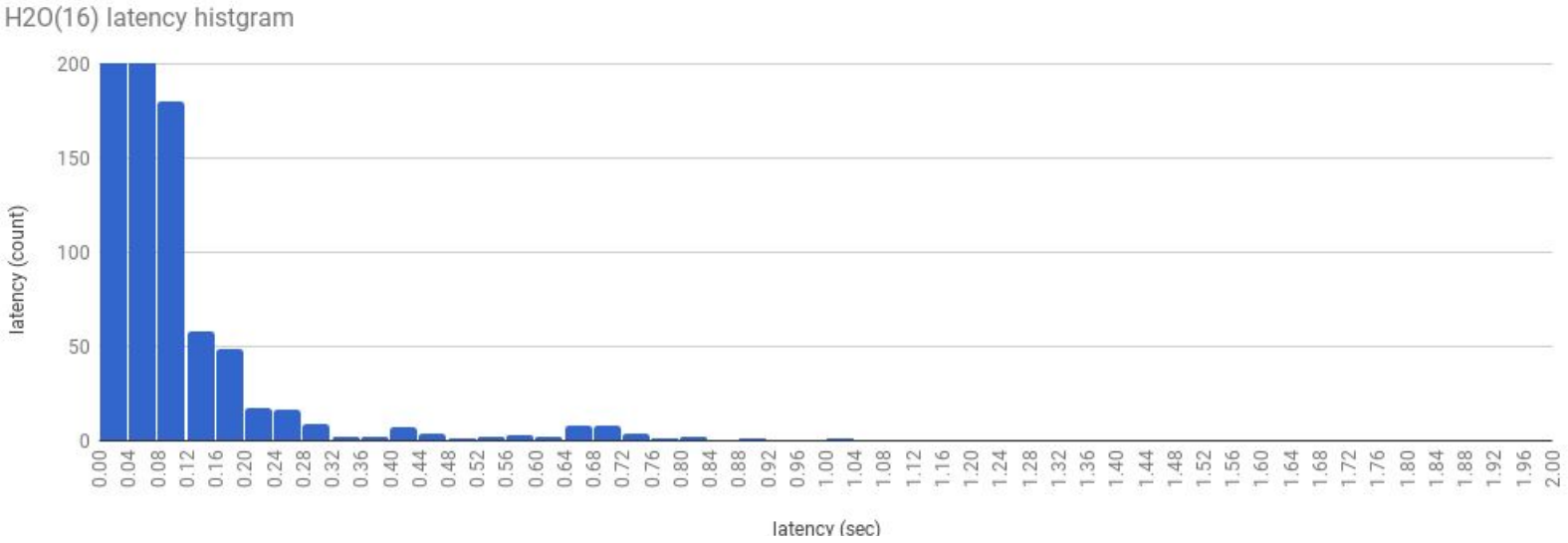
Benchmarks

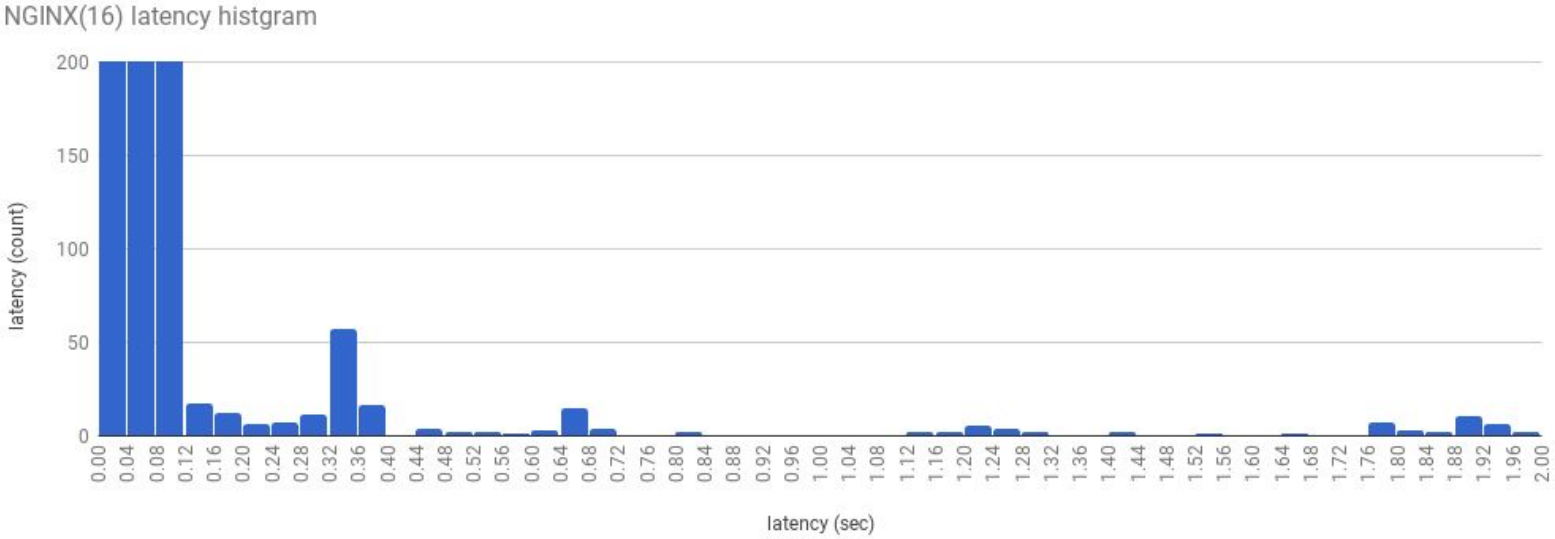
Benchmarks

- Performed some load tests, compared to Nginx + smalllight

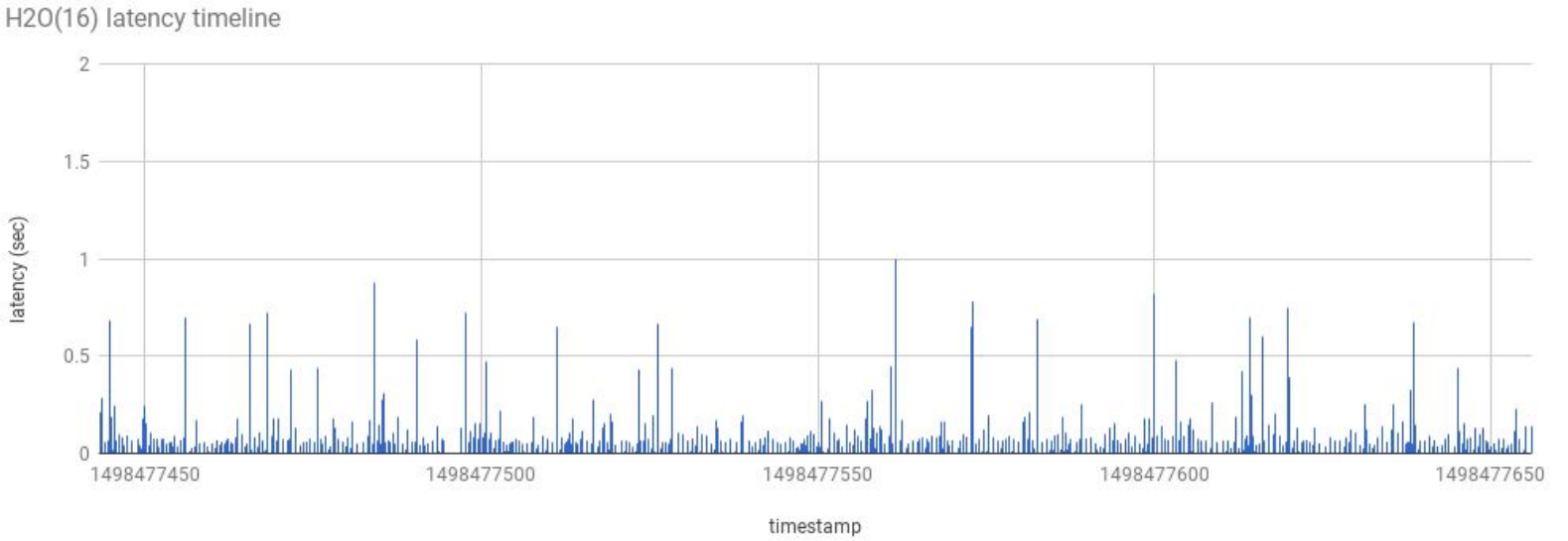
- Used real access logs with goreplay
  - https://github.com/buger/goreplay

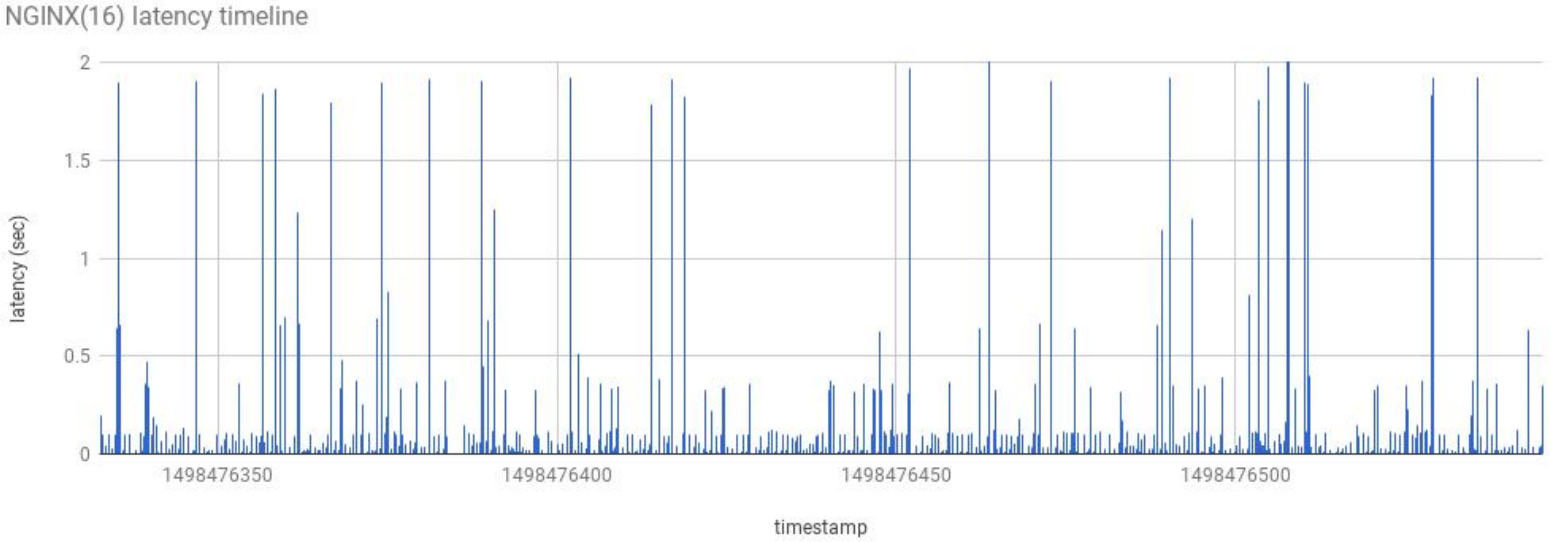
Benchmarks - Latency Histogram





Benchmarks - Latency Timeline





Benchmarks - Percentile

	Nginx	H2O	Ratio
average latency (ms)	32.49	30.56	94.1%
worst latency (ms)	2133	1000	46.9%
80 percentile latency(ms)	19.00	37.56	197.7%
95 percentile latency (ms)	94.06	74.53	79.2%
disk usage(KB)	891320	717944	80.5%
s3 transfer(B)	863139133	689163913	79.8%

Reactions from RoomClip

Reactions from RoomClip

- We can read!!!
- We ♥ printf debug!!
- OTOH we need more document..

Byproducts

mruby-rack

- https://github.com/i110/mruby-rack
- mruby port of Rack. Included in 2.3.0b1 released today!!
- Run original Rack specs

- It was harder than I expected..
  - Subtle syntax differences between mruby and CRuby..
  - lack of many core modules..

mruby-rack-httpcache

- https://github.com/i110/mruby-rack-httpcache
- Caching middleware
  - Inspired by Rack::Cache
  - 100% mruby compatible
  - More comformant to RFC7234

- Heavily tested in RoomClip
- But please treat it as Experimental..

Other Recent Improvements on H2O

H2O::Channel and task method

- Implemented by @narittan (Thank you!)
- Example: Combine first 2 responses from upstream
- So Cool!!

```
mruby.handler: |
  proc {|env|
    ch = H2O::Channel.new
    task { ch.push(http_request("http://example.com/foo").join) }
    task { ch.push(http_request("http://example.com/bar").join) }
    task { ch.push(http_request("http://example.com/baz").join) }
    first, second = ch.shift, ch.shift
    [200, {}, [first[2].join, second[2].join]]
}
```

H2O::Channel and task method

- Example: Redis pub/sub using `task`

```
mruby.handler: |
  cached = 'initial'


  task {
    redis = H2O::Redis.new(:host => '127.0.0.1', :port => 6379)
    redis_channel = redis.subscribe('chan1').join
    while reply = redis_channel.shift
      cached = reply[1]
    end
  }


  proc {|env|
    [200, {}, ["current: #{cached}"]]
  }
```

H2O::TCPSocket

- Write your own protocol binding in mruby

```
proc {|env|
  sock = H2O::TCPSocket.open('127.0.0.1', 8080)
  sock.write([
    "GET / HTTP/1.1", "Host: 127.0.0.1", "Connection: close", "\r\n"
  ]).join("\r\n")
  content = sock.read
  [200, {}, [content]]
}
```

Server Timing

- Server performance metrics
  - https://www.w3.org/TR/server-timing/
- `server-timing: ON` config enables this feature

```
HTTP/1.1 200 OK
transfer-encoding: chunked
trailer: server-timing
server-timing: connect; dur=0.153, request-header; dur=0, request-body; dur=0, request-total; dur=0, process; dur=205.724,
        proxy-idle; dur=103.091, proxy-connect; dur=0.232, proxy-request; dur=0, proxy-process; dur=102.401

(body)

server-timing: response; dur=100.816, total; dur=306.54, proxy-response; dur=100.816, proxy-total; dur=203.217
```

103 Early Hints

- https://tools.ietf.org/html/rfc8297
- H2O now supports rack.early_hints

```
paths:

  /with-mruby:
    mruby.handler: |
      proc {|env|
        env['rack.early_hints'].call({'link' => "</style.css>; rel=preload"})
      }
    proxy.reverse.url: https://example.com

  /without-mruby:
    headers.add:
      header:
        - "link: </style.css>; rel=preload"
      when: early
    proxy.reverse.url: https://example.com
```

Early-Data header and 425 Too Early

- TLS 1.3 introduces Early-data
  - application data sent before TLS handshake completes
  - can be replayed by an attacker, since it is sent at 0-RTT
- replay isn't an issue for idempotent requests, but...
  - in reality, only the webapp knows if a request has idempotency
- webapp behind a reverse proxy can't tell if the request was sent using Early-data
- Solution:
  - reverse proxies add Early-Data: 1 to requests received in 0-RTT
  - webapps can refuse the request for later replay using 425

History of 103 Early Hints

Feb 2016

- how should an app. server notify the H2 server to start pushing assets before generating the response?
  - some webapps emit 200 OK then builds their response
  - but that approach does not work well when other status codes are involved (e.g. 302)
- let's use 100-continue + link: rel-preload
  - implemented in H2O, nghttp2
  - should we standardize the feature?

The pushback

- changing the semantics of 100-continue is a hack
- we might want to have a new 1xx status code, but...
  - how does a final response update the headers sent using 1xx?

Nov 2016

- how about considering the headers of 1xx as "hints"?
  - fits nicely with the current generic definition of 1xx status codes
    - i.e. that unknown 1xx can be "ignored"
  - retransmitting same header twice is a non-issue in H2 thanks to HPACK
- response at IETF 97: "let's do that!"
  - becomes a Working Group document (Dec 2017)

What about clients that cannot handle 1xx?

- "we cannot break the web"
- "let buggy clients fail miserably. Protocol development should not be hindered by them"
- "how about negotiating the use?"
  - "header-based negotiation is end-to-end. But proxies in the middle might be buggy"
- conclusion:
  - just warn that use of 103 might have issues with H1 clients
  - no worries about h2 over tls

Existing clients might accept cookies in 1xx

- that would contradict with the concept of 103 being purely "hints"
- the choice: define 103 as something that
  - a) "can be considered as hints"
  - b) "is hints"
- conclusion: b
  - let's dismiss the theoretical existence of such clients
  - having a clear definition will be better in the long run

Jun 2017

- sent from the working group to IESG

Aug 2017

- approved by IESG

How to evaluate multiple 103 responses?

- two interpretations:
  - a) the following hints replace the preceding ones
  - b) each hints is individual
- conclusion: b
  - each source of the hints might have different policy on what to send as hints
    - e.g. a proxy might send hints based on machine learning, while the origin might send hints for js, css based on written rules
    - sending each signal as-is will give client more information to decide

Oct 2017

- "What's the blocker for 103 becoming RFC?"
- "I'm submitting the final draft now"

Dec 2017

- published as RFC 8297

Will 103 and H2 server push coexist?

- they can coexist 103
  - as hints sent to h2 server
  - server push is 1-RTT faster
    - but has issues, especially without Cache-Digests
- some hope 103 to kill push
  - because it's simpler
- API in the webapp will be indifferent
  - e.g. rack.early_hints

Lessons Learned

- ossification is an issue
  - buggy clients
  - new protocols will grease and obfuscate
    - occasionally send dummy extensions in any extension point that exists
    - make information unobservable by middleboxes
      - e.g. packet number encryption in QUIC
- we will see forgotten features of HTTP being used
  - informational response trailers what's next?

Summary

Supporting complex interaction with a simple API

- HTTP is becoming more and more complex
  - rationale: for speed and new use-cases
  - H2 push, early-hints, 425, Server-Timing, QUIC, WebSockets on H2, ...
- provide simple, programmable API for using the features
  - mruby & Rack to the rescue!