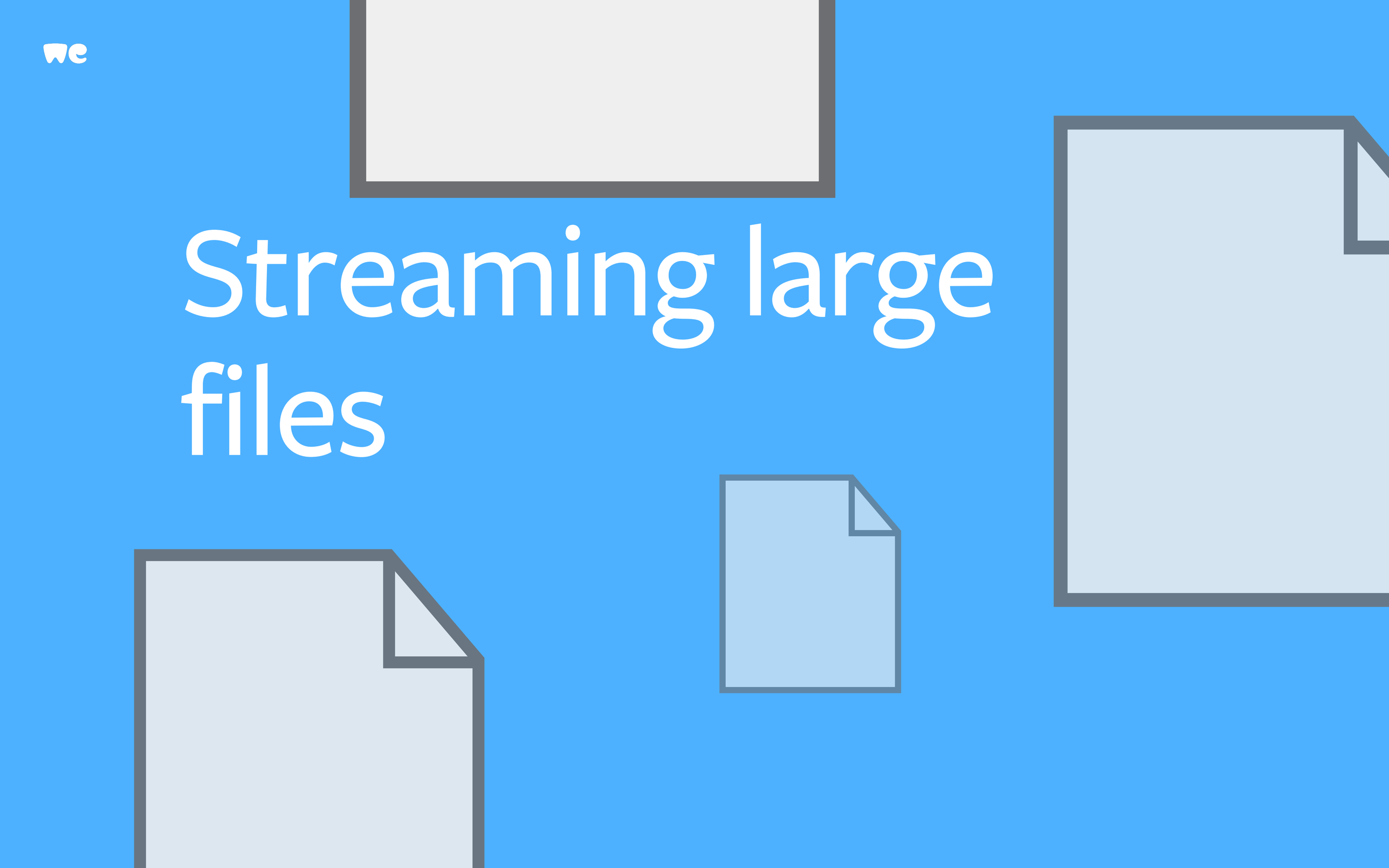
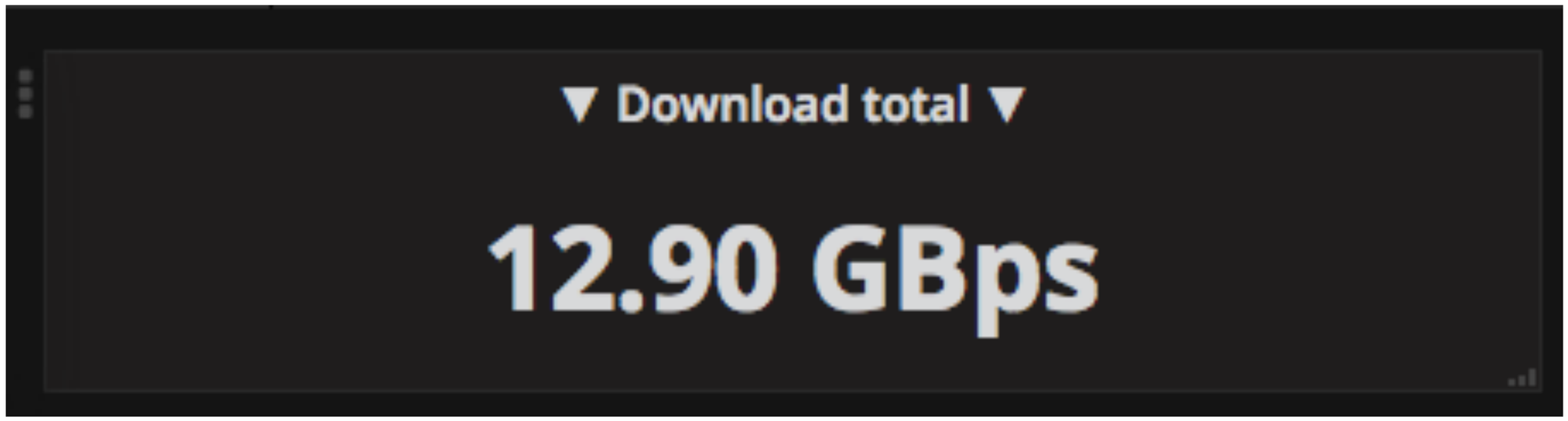
# Streaming large files



# We got a lot of them.

About 15 gigabytes per second outgoing at peak times.


▼ Download total ▼
12.90 GBps

# Let’s just redirect to S3!

..right?

☆ **WeTransfer**

Download confirmation from george.ponomarev@gmail.com via WeTransfer

**To:** Julik Tarkhanov

**Reply-To:** Жорж Пономарев

---

george.ponomarev@gmail.com
downloaded your files

'Fishbowl - my end. это записано как double ender, аудио от другого чувака последует. Все что мне нужно это чтобы наши голоса были внятно сведены по стерео (оба в обоих ухах) и похожи по уровням. и слушались прилично. Это все пойдет в ютуб. Монтировать и резать ничего не надо.'

**Files (1.49 GB total)**
BoxBlur_p1_v01.aiff
BoxBlur_p2_v01.aiff
Recap_v01.aiff

**Will be deleted on**
2 January, 2016

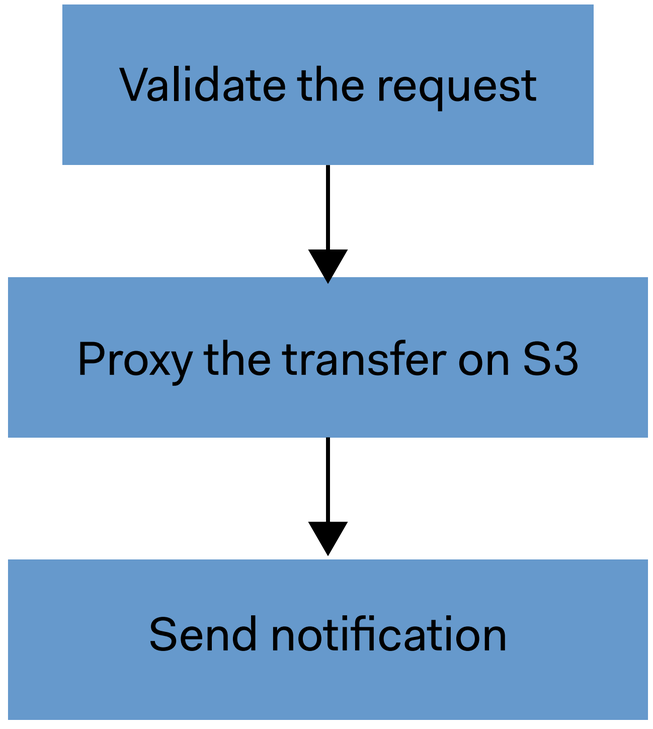
**Download link**
http://we.tl/XfKs8wtpbJ

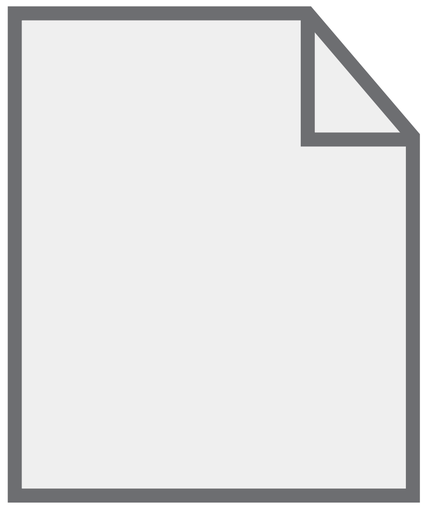
Get more out of WeTransfer, get Plus

# Therefore, we need an HTTP proxy

that will stand between the downloader and the AWS S3 bucket

# What we need to do

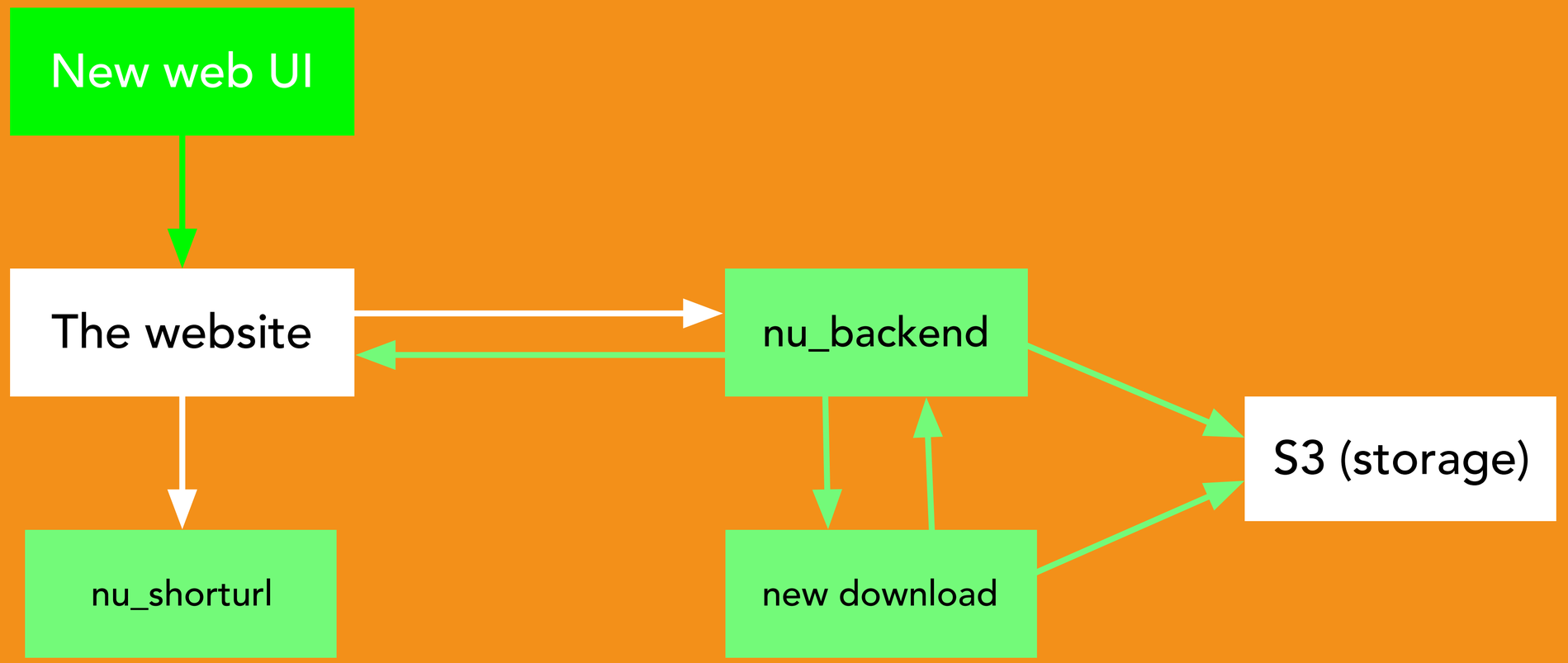




What we had was 1 ZIP per transfer,
pre-packaged and stored on S3

We needed files stored separately, and the ZIP to be assembled at-download. And pick and choose at download.

# Small detour: our service layout



Rewrite?

# Qualities of rotting software

- Rigidity
- Viscosity
- Immobility
- Fragility

# Streaming from Rack: a refresher

```
[200, {'Content-Type' => 'binary/octet-stream'}, [“hello”]]
```

```
[…, [“hello”, “goodbye”]]
```

```
[…, ‘a’..’z’]
```

```
[…, my_response_responding_to_each]
```

# Take 1

```
class ProxyResponseBody < Struct.new(:uri)
  def each
    Net::HTTP.start(uri.host, uri.port) do |http|
      request = Net::HTTP::Get.new(uri)
      http.request request do |response|
        response.read_body {|chunk| yield(chunk) }
      end
    end
  end
end

[200, {'Content-Type' => ‘binary/octet-stream'},
       ProxyResponseBody.new('https://s3.aws.com/...')]
```

# Retries

```
FancyHTTP.get(s3_url, {'Range' => ('bytes/%d-' % bytes_sent_so_far)})
```

# Download-send loop in segments

```
segment = receive(s3_uri, {'Range' => '0-5000'})
send(segment)
segment = receive(s3_uri, {'Range' => '5000-10000'})
send(segment)
```

# String churn will kill you

Even in tight loops Ruby will allocate strings. Lots of strings.

Currently there is no ByteBuffer data structure that allows you to hide them from the heap or from the Ruby GC, and severe memory inflation will result. And managing that memory takes CPU.

# PHP has a few tricks up it’s sleeve

```php
$fd = fopen("php://output", "wb");
curl_setopt($curl, CURLOPT_WRITEDATA, $fd);
```

# Sending quickly

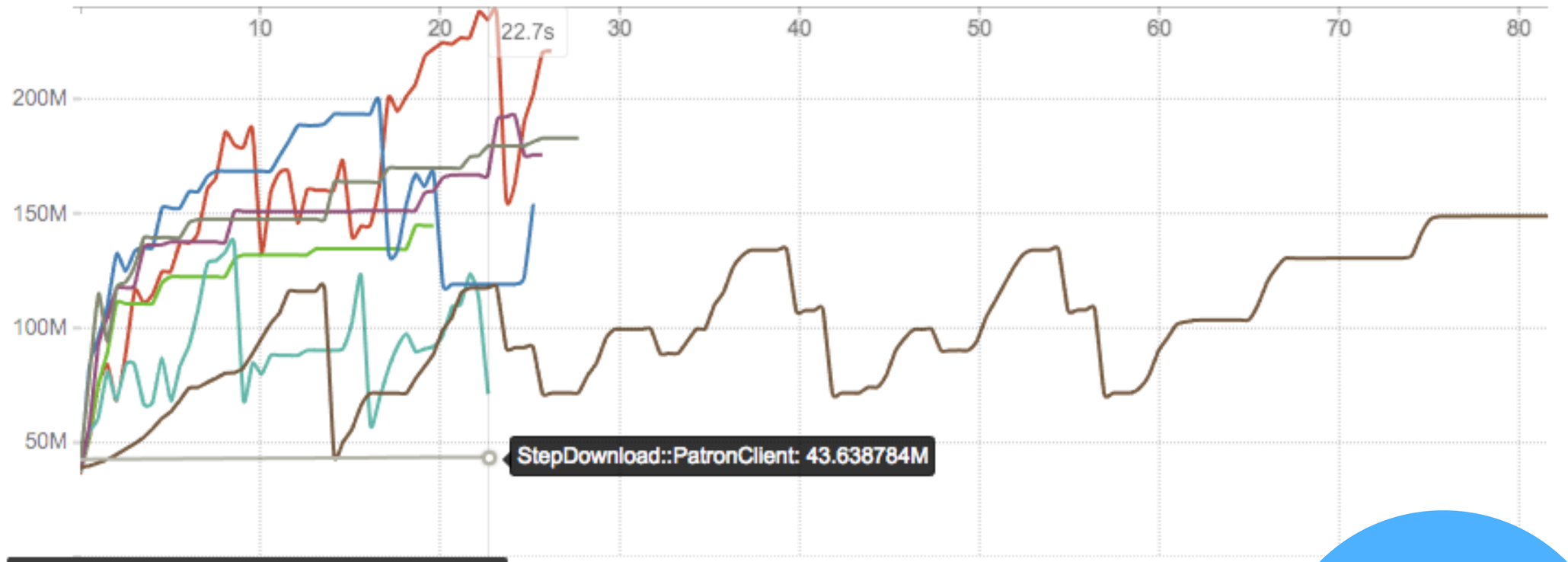
```
NAME
     sendfile -- send a file to a socket

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     #include <sys/uio.h>

     int
     sendfile(int fd, int s, off_t offset, off_t *len, struct sf_hdtr *hdtr,
         int flags);
```

# Sending quickly

```
gem “sendfile”

response_headers[‘rack.hijack’] = ->(socket) {
  socket.sendfile(file)
}
```

# One does not simply choose an HTTP client

# RSS_mri2.2.4_x86_64-linux-gnu_1.0GB




Not all HTTP clients are created equal

# The trick

```
if (RTEST(download_file)) {
  // returns a FILE*
  state->download_file = open_file(download_file, "wb");
  curl_easy_setopt(curl, CURLOPT_WRITEDATA, state->download_file);
} ...
```

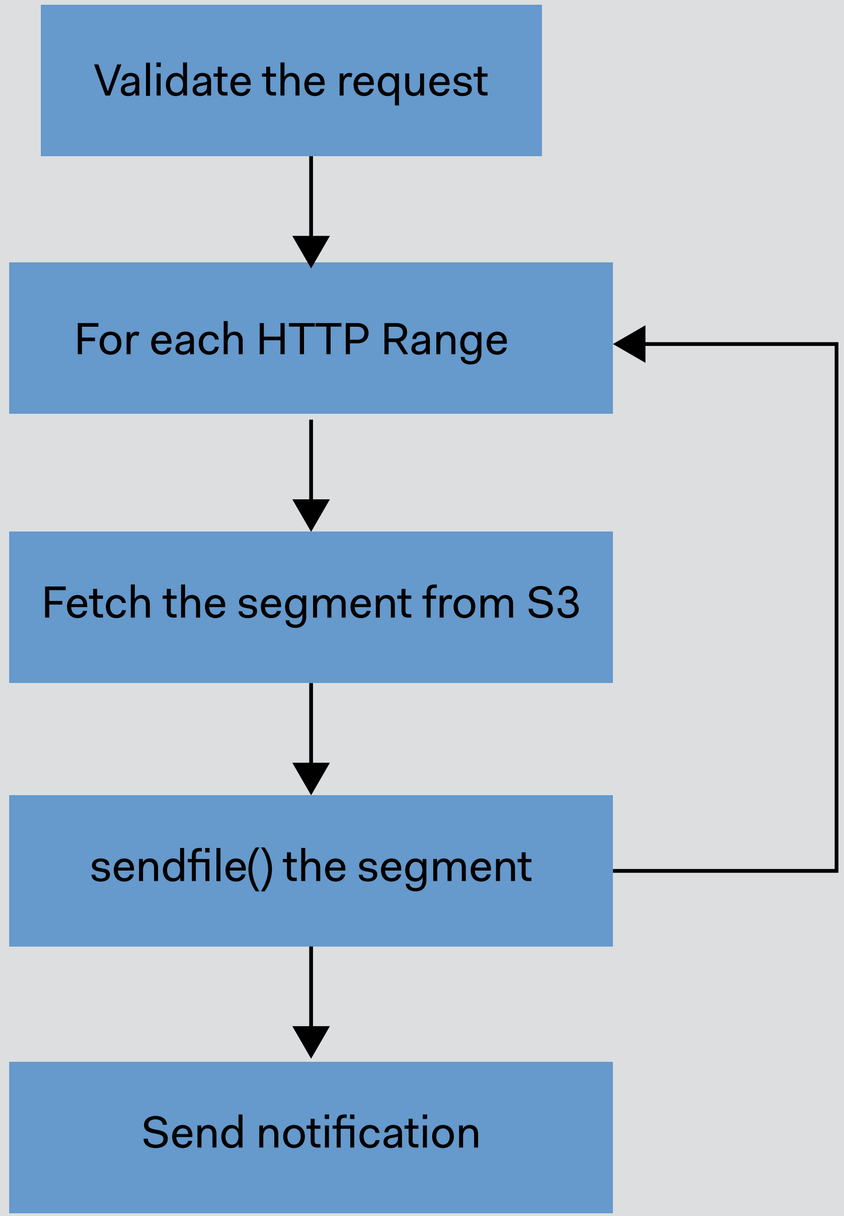
# Patron wins

because it can fetch direct-to-file (a FILE* pointer) in userspace (in raw C, while the GIL is unlocked). This creates optimum threading with a few tiny downsides - threads that are out of the GIL do not answer to signals on time, for instance.

But it threads like crazy!

# Download/send loop

```
content_ranges = RangeUtils.http_ranges_for_size(remote_file_size, 5 * 1024 * 1024)
content_ranges.each do |range|
  tf = Tempfile.new 'buf'
  res = patron.get_file(uri, tf.path,
     {'Range' => ('%d-%d' % [range.begin, range.end])})
  raise "Unexpected status for segment" if res.status != 206
  socket.sendfile(tf)
  tf.close; tf.unlink
end
```


Validate the request
For each HTTP Range
Fetch the segment from S3
sendfile() the segment
Send notification

# Concurrency model

# Puma + Threads

- No callback soup, promises, reactors, awaits, fibers, EventMachines. And no strings.
- Linear imperative code
- Extremely easy to manage
- Actual syscalls, actual sockets, actual exceptions

# The only threading bug we had

```
map '/download' do
  run DownloadServer.new
end


class DownloadServer
  def call(env)
    @request_id = …
…
```

# fast_send

```
class BigResponse
  def each_file
    File.open('/large_file1.bin', 'rb'){|fh| yield(fh) }
    File.open('/large_file2.bin', 'rb'){|fh| yield(fh) }
  end
end

progress = ->(sent_this_time, sent_total) {
  # record this in your stats...
}

[200, {'fast_send.bytes_sent' => progress, ‘Content-Length’ => big_size},
   BigResponse.new]
```

# fast_send

https://github.com/WeTransfer/fast_send

# Yes, but my $fastlang_du_jour can fit 10K people on one server, concurrently

# 10K people per box is nice...

But would you like to be one of them?

Median download speed is 1.2 MBps

You don’t know how fast a specific downloading client is going to go

AWS does not give you cheap instances with lots of bandwidth and low RAM/CPU

How many people would you fit without having them starve each other’s downloads?

If we aim to oversell, packing too many people onto a box would mean degraded service for users

# On-the-fly ZIPs

# Transfer structure

```
s3://bucket/<transfer_id>/file1
s3://bucket/<transfer_id>/file2
s3://bucket/<transfer_id>/file3
s3://bucket/<transfer_id>/manifest.json
```

# mod_zip

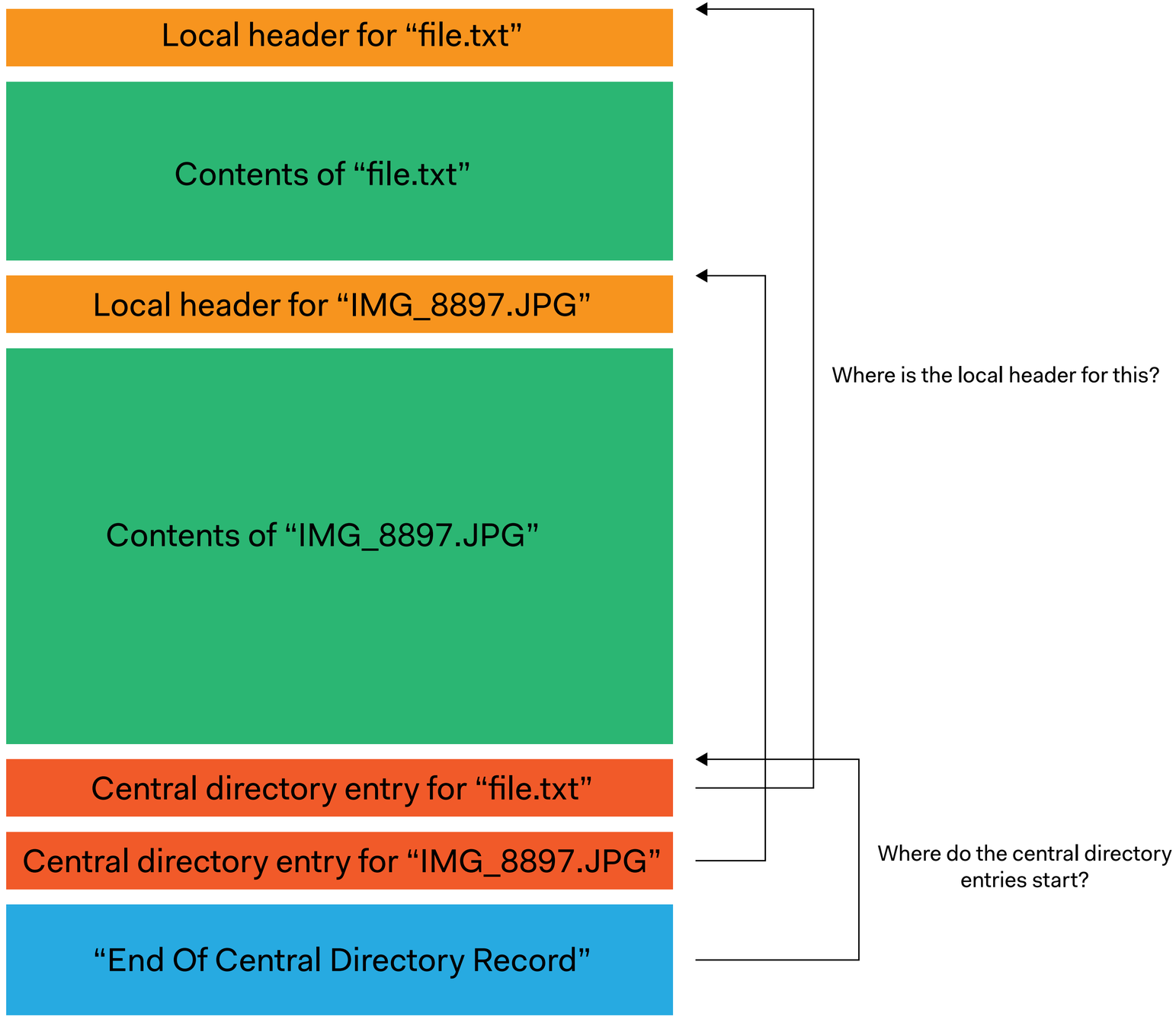
Our app

List of filenames, URLs and sizes

```
s3://bucket/<transfer_id>/file1
s3://bucket/<transfer_id>/file2
s3://bucket/<transfer_id>/file3
s3://bucket/<transfer_id>/manifest.json
```

Statically linked plugin in nginx written in evented C

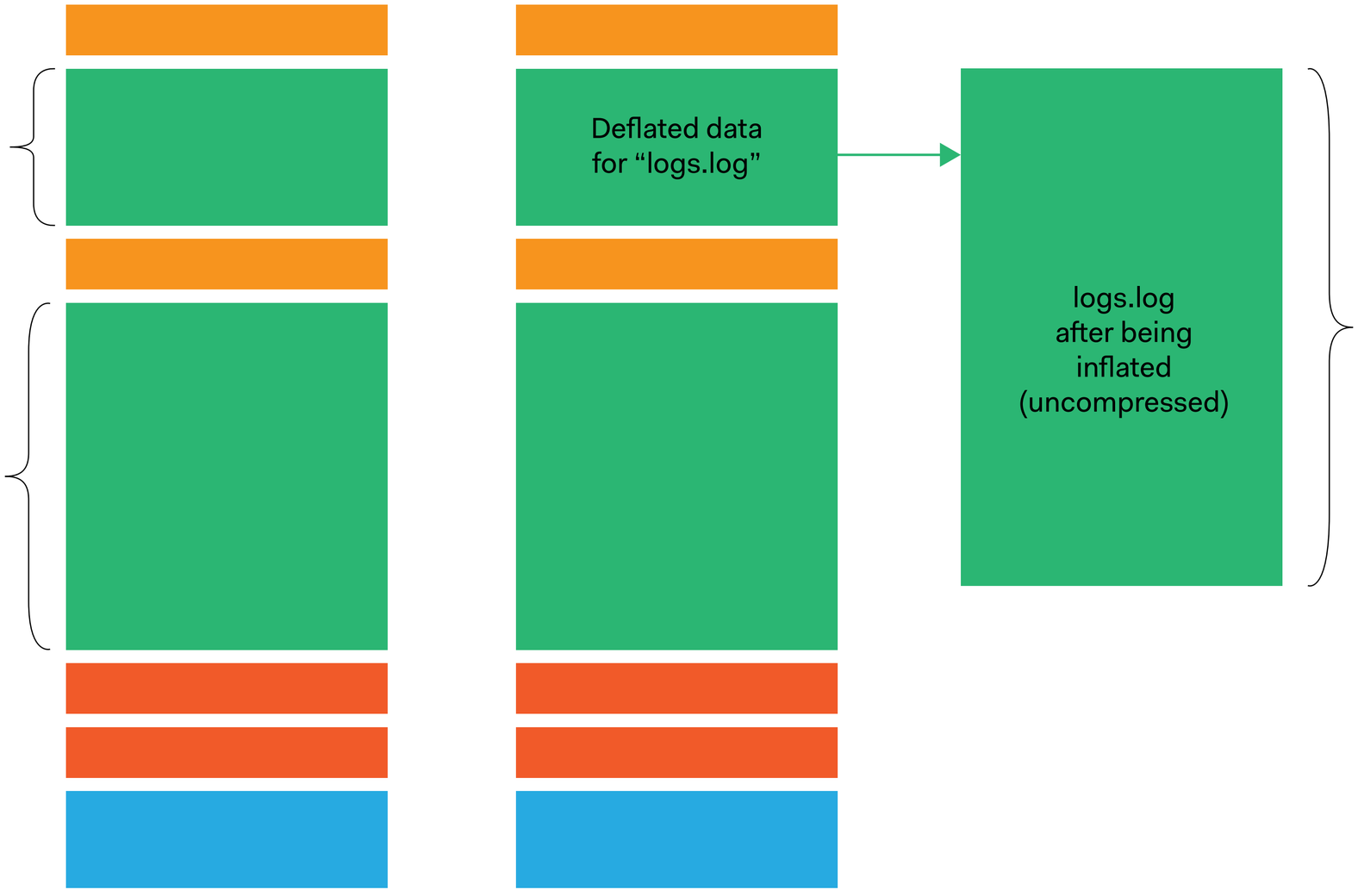
Magic!


Local header for “file.txt”
Contents of “file.txt”
Local header for “IMG_8897.JPG”
Contents of “IMG_8897.JPG”
Central directory entry for “file.txt”
Central directory entry for “IMG_8897.JPG”
“End Of Central Directory Record”
Where is the local header for this?
Where do the central directory entries start?

# To write out this data

bytesize compressed (how many bytes to read from the file)
bytesize once extracted
CRC32 of the uncompressed file's contents (has to be known upfront)
bytesize("file.txt")
file.txt
Size of extra fields (zero in this case, but still gets written out)

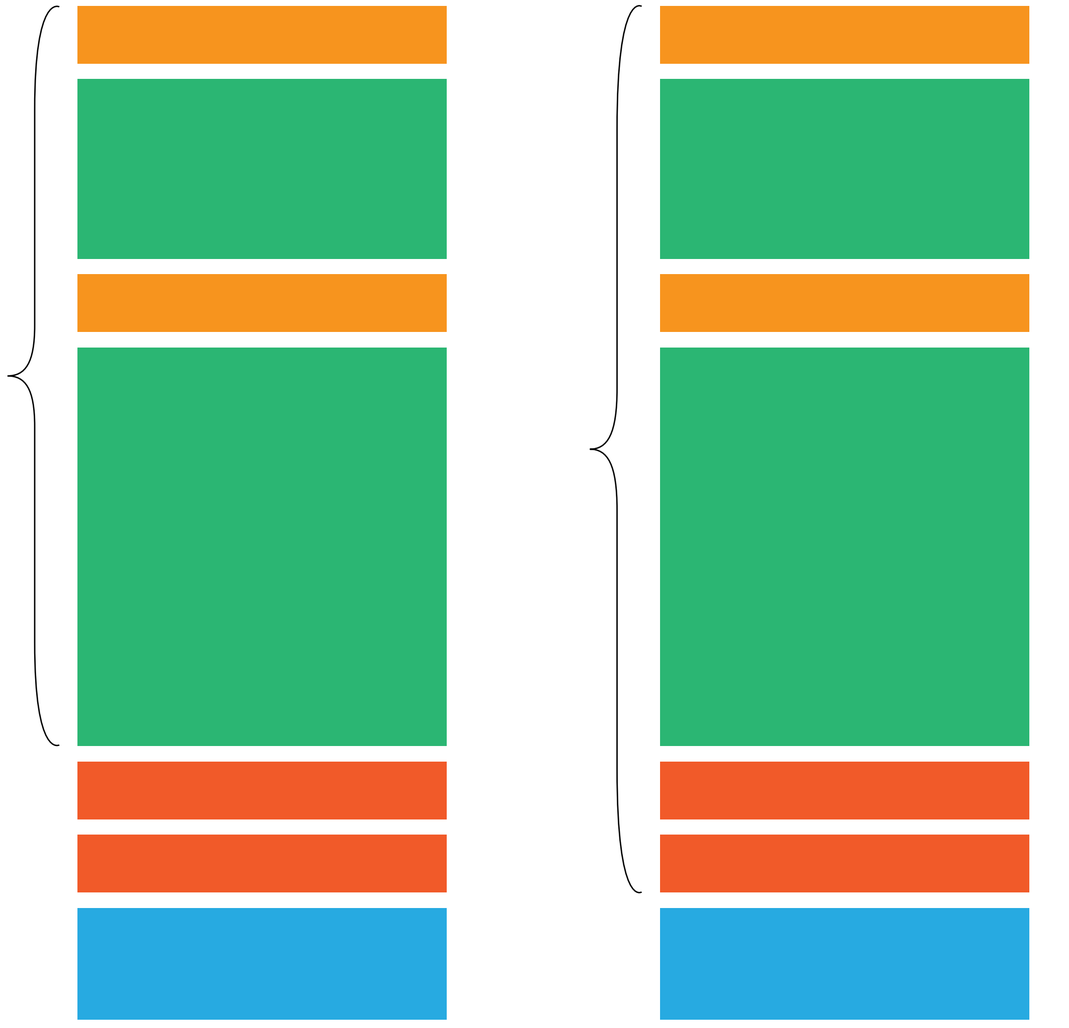
# When do you need Zip64




If any of the compressed
data blocks is larger than 4GB

If just one file once extracted
is larger than 4GB

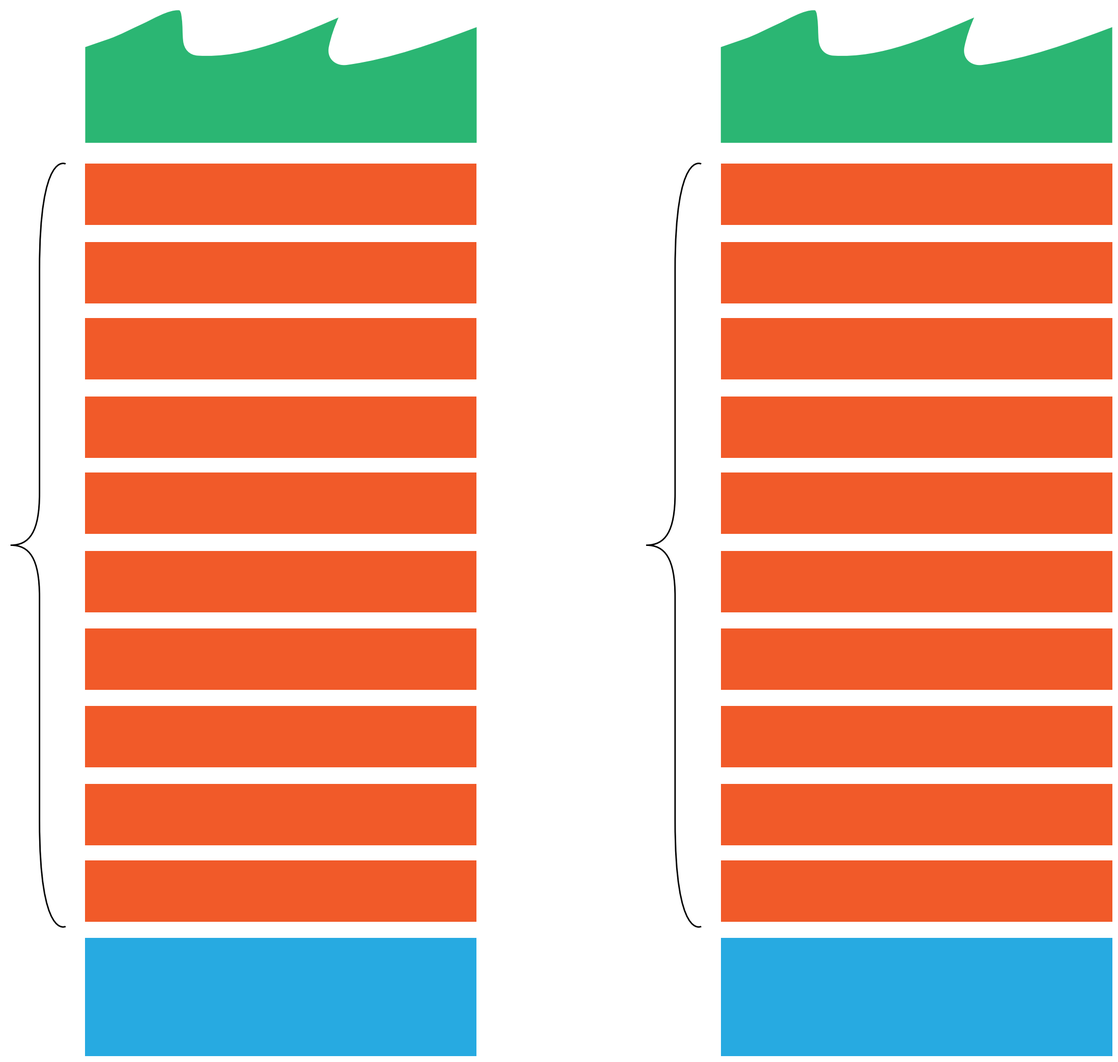
# When do you need Zip64



If the central directory
starts beyound the 4GB offset

If the end of central directory
record is beyound 4GB offset

# When do you need Zip64

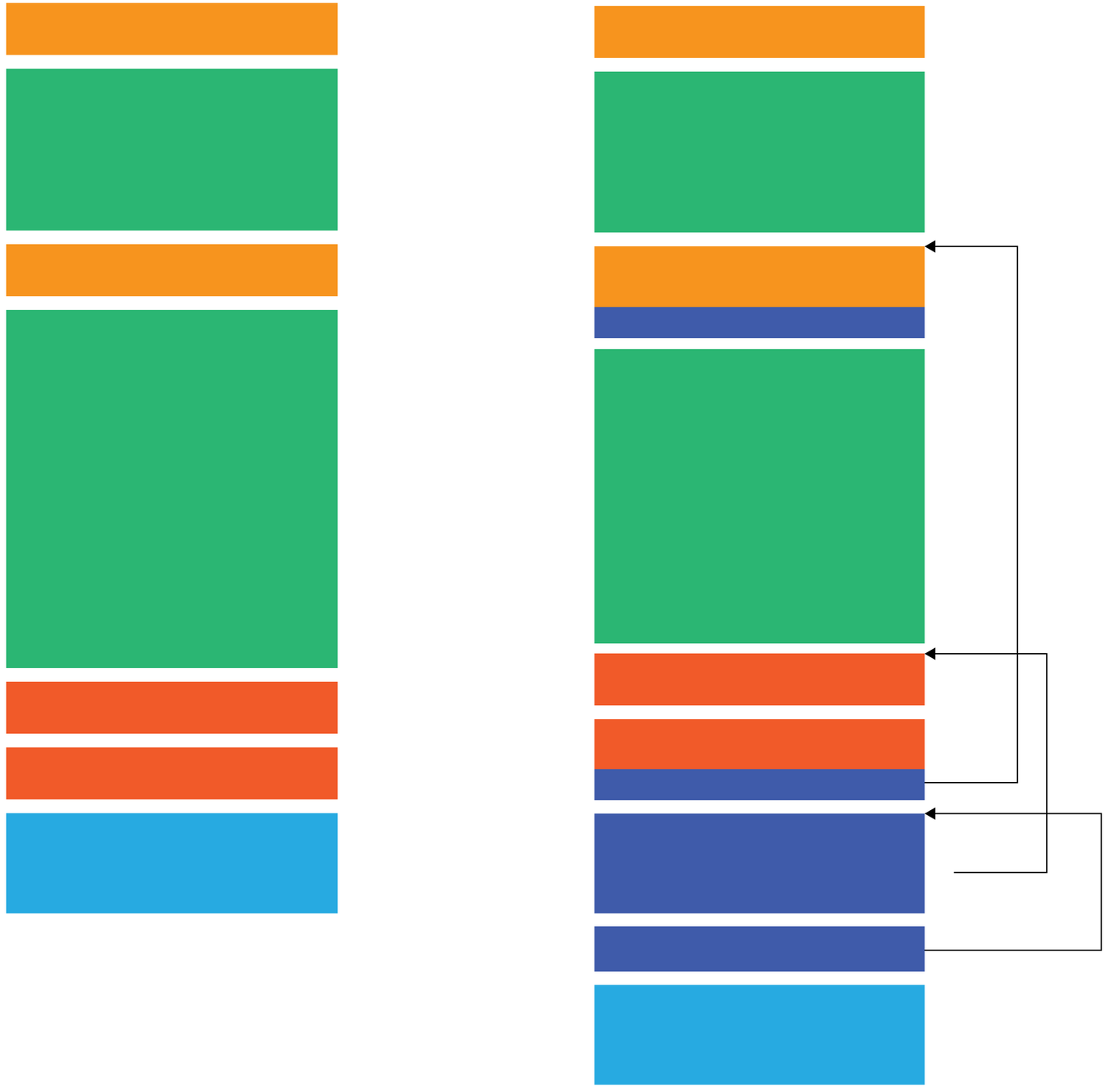


If you have more
than 65535 files in the ZIP

If the central directory
is larger than 4GB

And if you store your ZIP across more than 1024 disks.

If you do: seek professional advice.



Zip64 additions

bytesize compressed (how many bytes to read from the file)
bytesize once extracted
CRC32 of the uncompressed file's contents (has to be known upfront)
bytesize("file.txt")
file.txt
Size of extra fields (zero in this case, but still gets written out)

Filler instead of bytesize compressed
Filler instead of bytesize once extracted
CRC32 of the uncompressed file's contents (has to be known upfront)
bytesize("file.txt")
file.txt
The size of that thing below (extra fields)

Actual bytesize compressed (now 8 bytes instead of 4)
Actual bytesize uncompressed (now 8 bytes instead of 4)

Filler instead of bytesize compressed (4 bytes)
Filler instead of bytesize once extracted (4 bytes)
Filler instead of the local file header location (4 bytes)
CRC32 of the uncompressed file's contents (has to be known upfront)
bytesize("file.txt")
file.txt
The size of that thing below (extra fields)

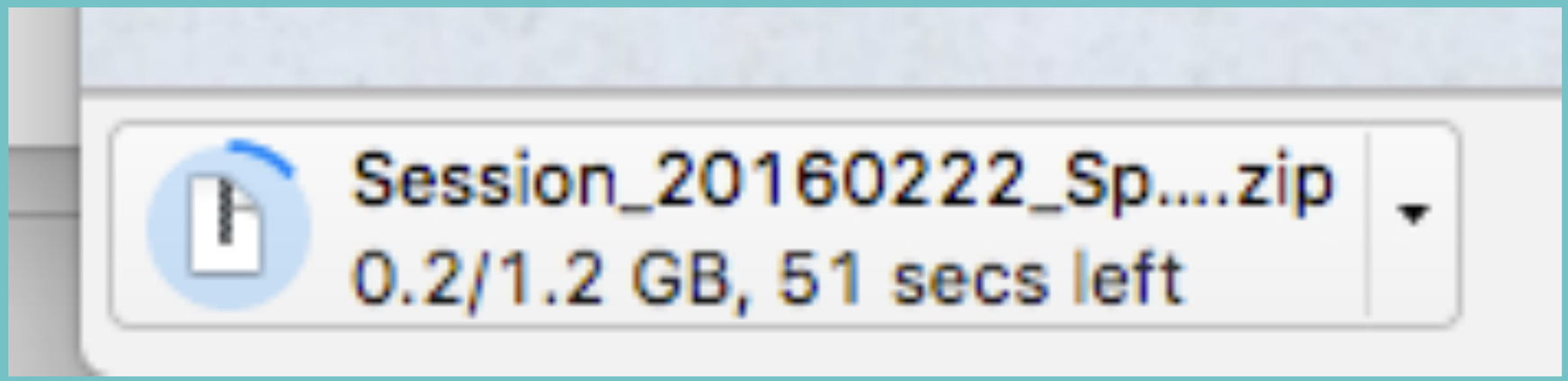
Actual bytesize compressed (now 8 bytes instead of 4)
Actual bytesize uncompressed (now 8 bytes instead of 4)
Actual location of the local file header (can be HUGE now)

# It should work somewhat like this

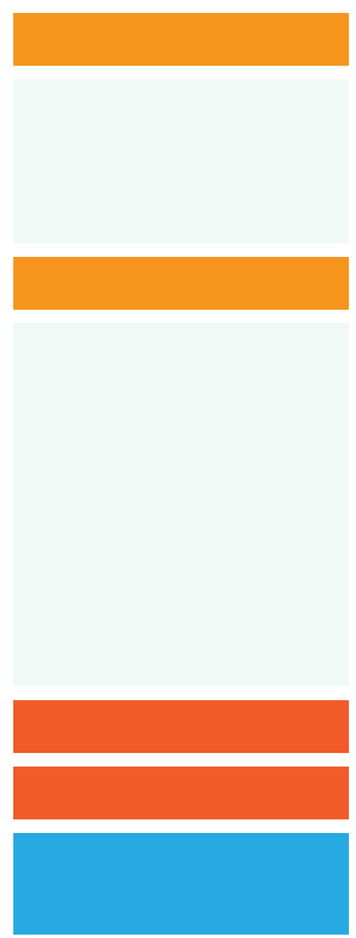
```
def each_file
  # Generate the local header for the file
  tf = Tempfile.new 'xx'
  tf << generate_zip_header_for(our_file)
  tf.rewind
  yield(tf)

  tf = fetch_segment(our_file)
  yield(tf)

  # Generate the central directory
  tf = Tempfile.new 'xx'
  tf << generate_zip_central_directory(our_file_1, our_file_2, ...)
  yield(tf)
end
```

# We need non-rewinding output

# And size estimation!



# And to estimate size we have to...



```
exact_size_to_the_byte = estimate_size do |zip|
  zip.add_stored_entry(filename: "MOV_1234.MP4", size: 898090)
  zip.add_stored_entry(filename: "MOV_1235.MP4", size: 7855126)
end

[200, {"Content-Length" => exact_size_to_the_byte.to_s}, zip_body]
```

# RubyZip: All teh inheritance

```
class FancyOutputStream < ::Zip::OutputStream
  # Adds a new entry (set all the fields upfront)
  def put_next_entry(entry_name, size, crc)
    new_entry = ::Zip::Entry.new(@file_name, entry_name)
    new_entry.compression_method = Zip::Entry::STORED
    ...
  end

  # DO NOT REWIND YOU RUBYZIP
  def update_local_headers; end
  ...
end
```

At some point though… we discovered it wasn't enough overrides.

Rubyzip is ravioli software. With lasagna inside. Each lasagna layer contains tortellini. It’s like Java but with Ruby hacks on top.

# Most boxes here are separate classes in Rubyzip

Zip64 additions

# When overrides no longer work you need to do a clean-room

# Our own zip file writer

- Non-rewinding
- Is all in 1 module (and 1 file)
- Portable (no magic Ruby tricks, no metaprogramming)
- Tested with very large offsets
- Complete and automatic Zip64 support
- Complete and automatic UTF8 filename flags
- Complete and automatic data descriptor support
- Large archive readability manually tested
- ~350 lines with comments

# zip_tricks

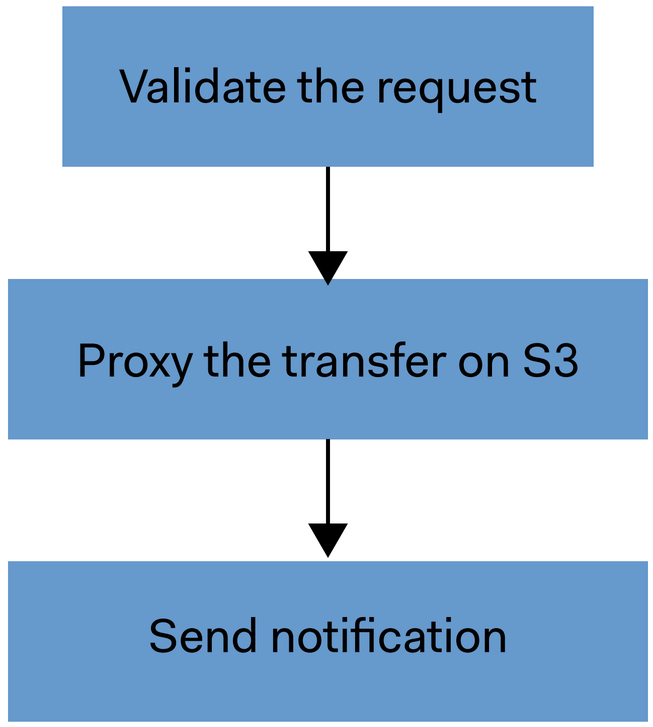
```
ZipTricks::Streamer.open(out) do |zip|
  zip.write_stored_file('mov.mp4.txt') do |sink|
    File.open('mov.mp4', 'rb'){|source| IO.copy_stream(source, sink) }
  end
  zip.write_deflated_file('long-novel.txt') do |sink|
    File.open('novel.txt', 'rb'){|source| IO.copy_stream(source, sink) }
  end
end
```

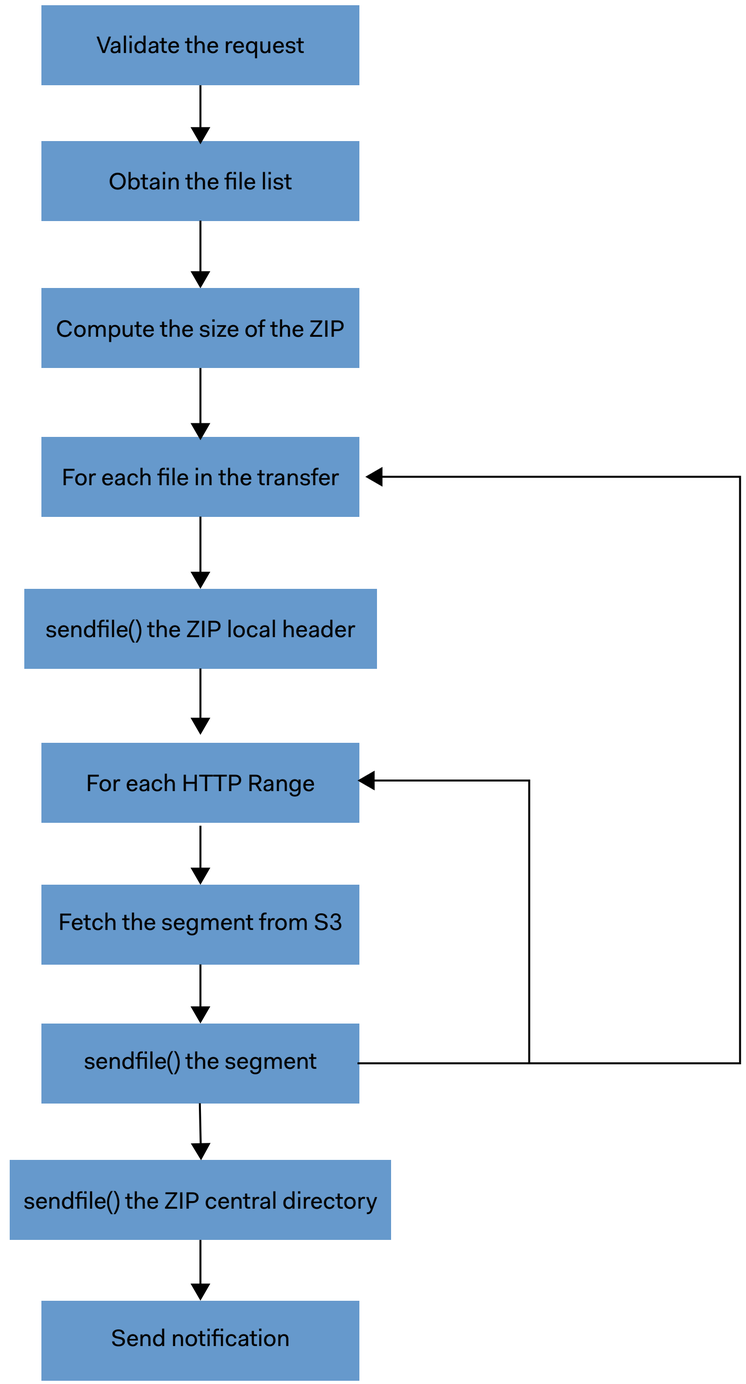
# Time to fix an obscure bug related to one single (buggy) version of The Unarchiver.app: 20 minutes

# zip_tricks & zipline

https://github.com/WeTransfer/zip_tricks

https://github.com/fringd/zipline


Validate the request
Proxy the transfer on S3
Send notification


Validate the request
Obtain the file list
Compute the size of the ZIP
For each file in the transfer
sendfile() the ZIP local header
For each HTTP Range
Fetch the segment from S3
sendfile() the segment
sendfile() the ZIP central directory
Send notification

# Ruby

Had to use sendfile()

Had to use VM-bypassing HTTP GETs

Had to write our own ZIP library

Lots of RAM used

# $fastlang

Would have to use sendfile or raw writes

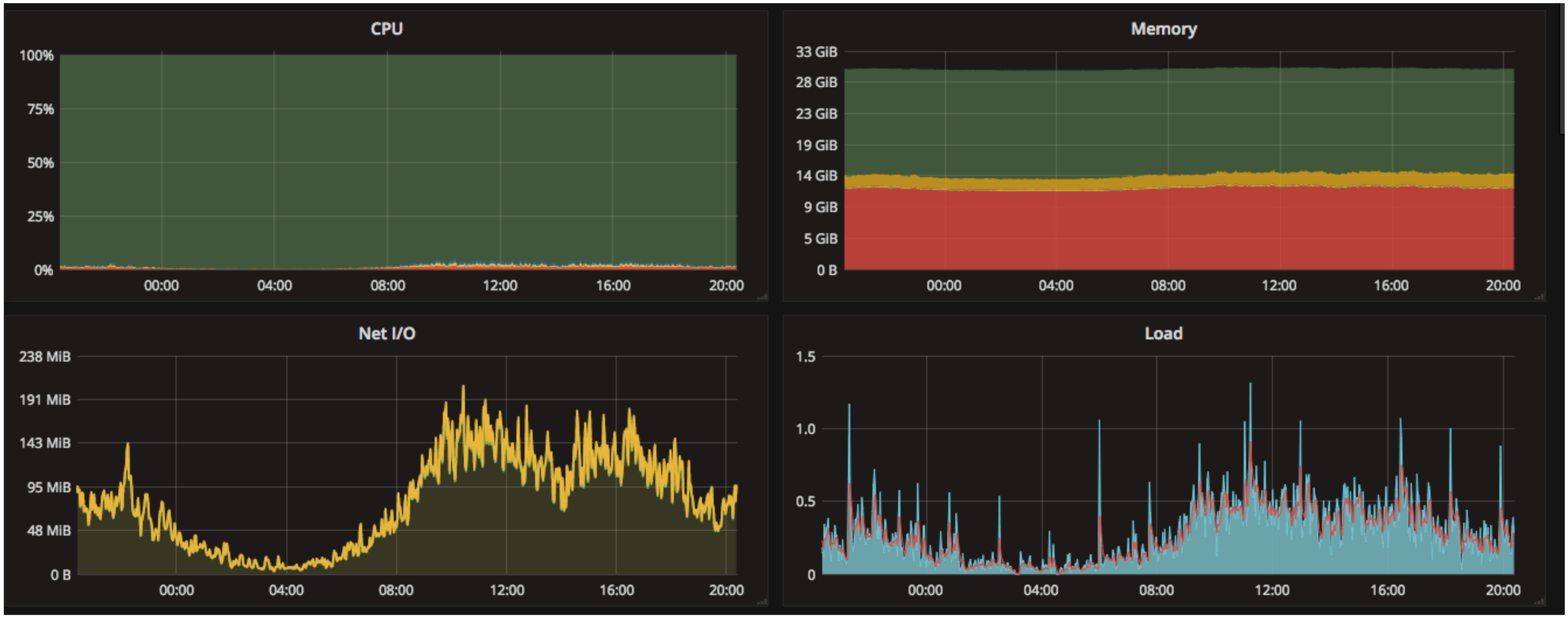
Depends

Probably would have to write our own ZIP library

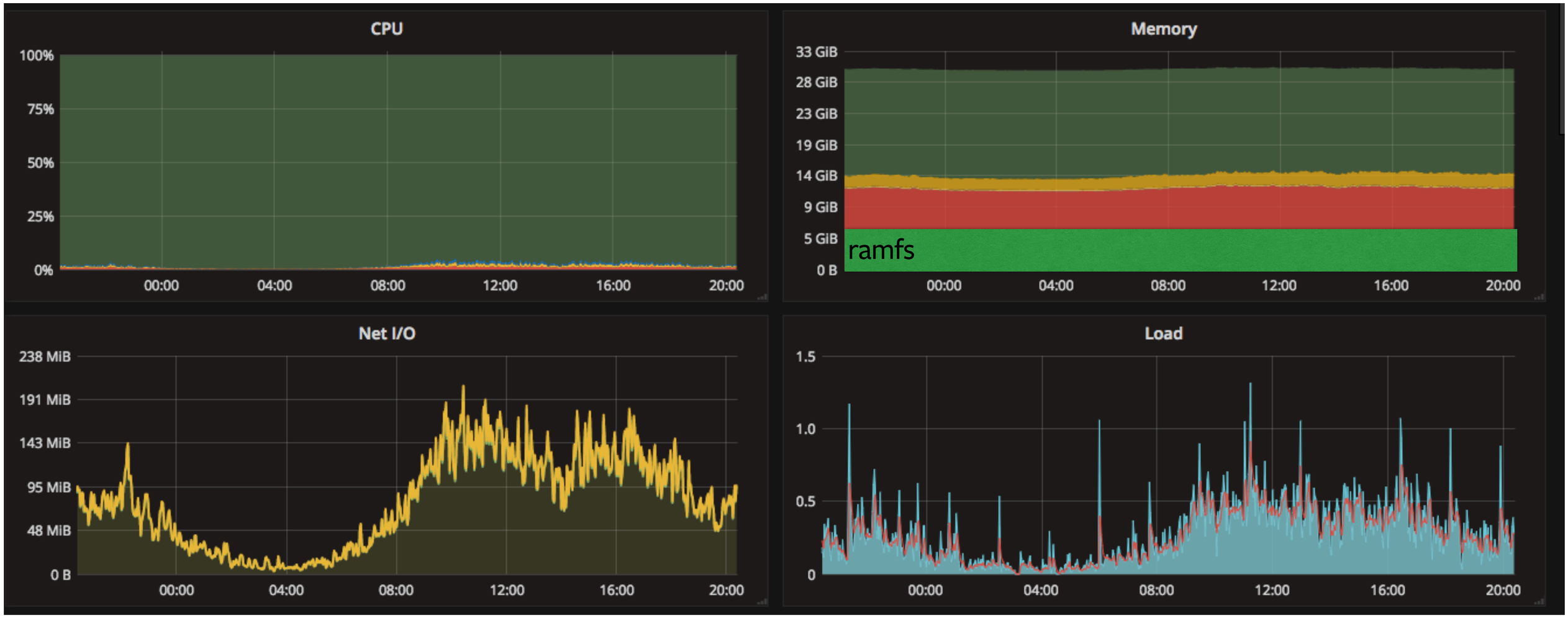
Would probably use less RAM

(does not apply to JVM-based Fastlangs, they download RAM)

# What is the actual overhead?



# Wait a minute...



We listen to the $fastlang sirens too much.

Find the actual things
that are your problem,
and try to solve them.

Let’s push Ruby further.

https://github.com/WeTransfer/fast_send
https://github.com/WeTransfer/zip_tricks
https://github.com/toland/patron