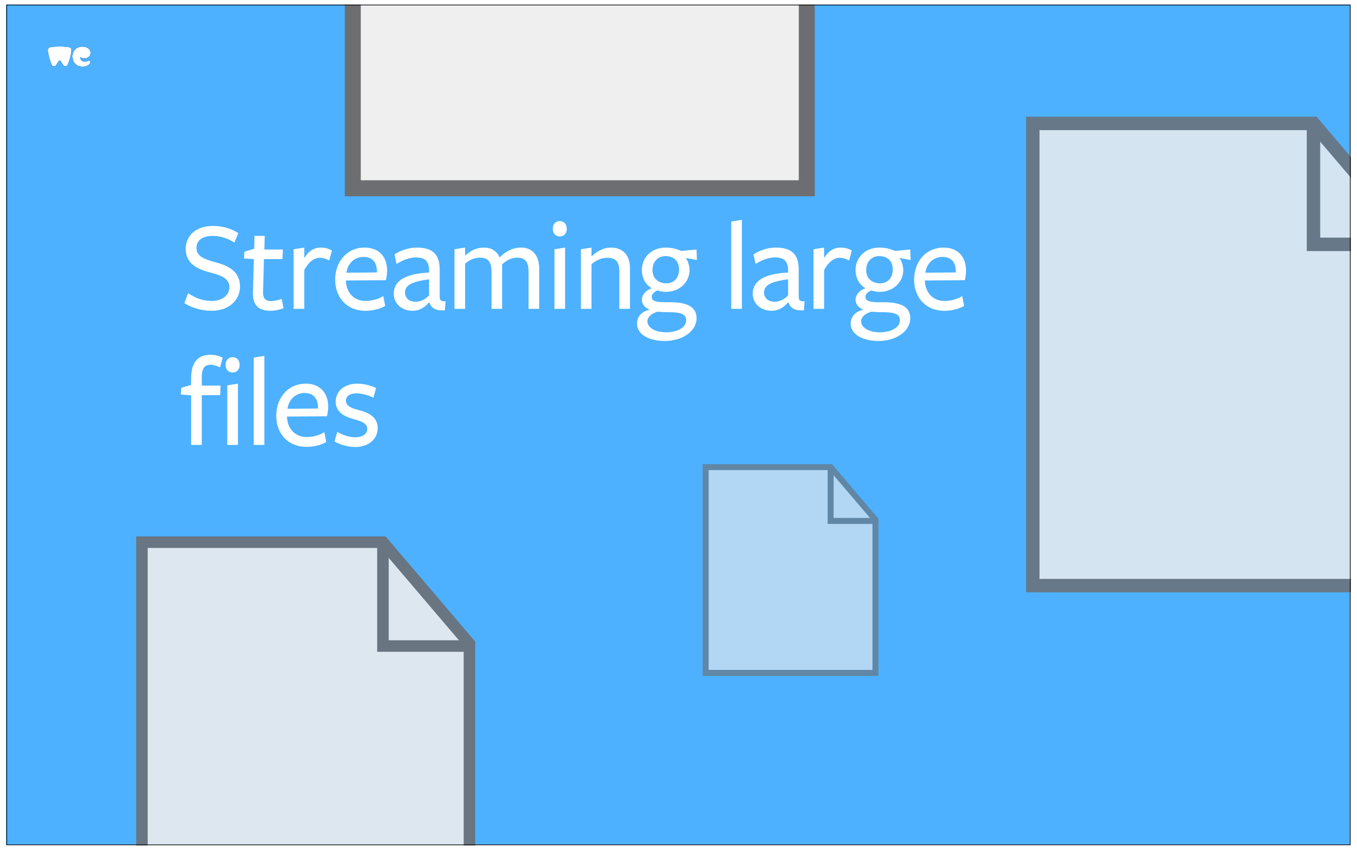


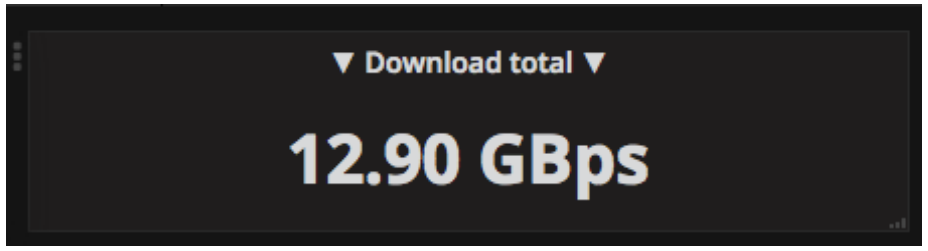
Streaming large files

Hi, I'm Julik and I am the lead engineer at WeTransfer. My job is watching after our backend code and help our team deliver a world class product.

We used to have a problem – streaming large HTTP responses, and I will tell you all about it.

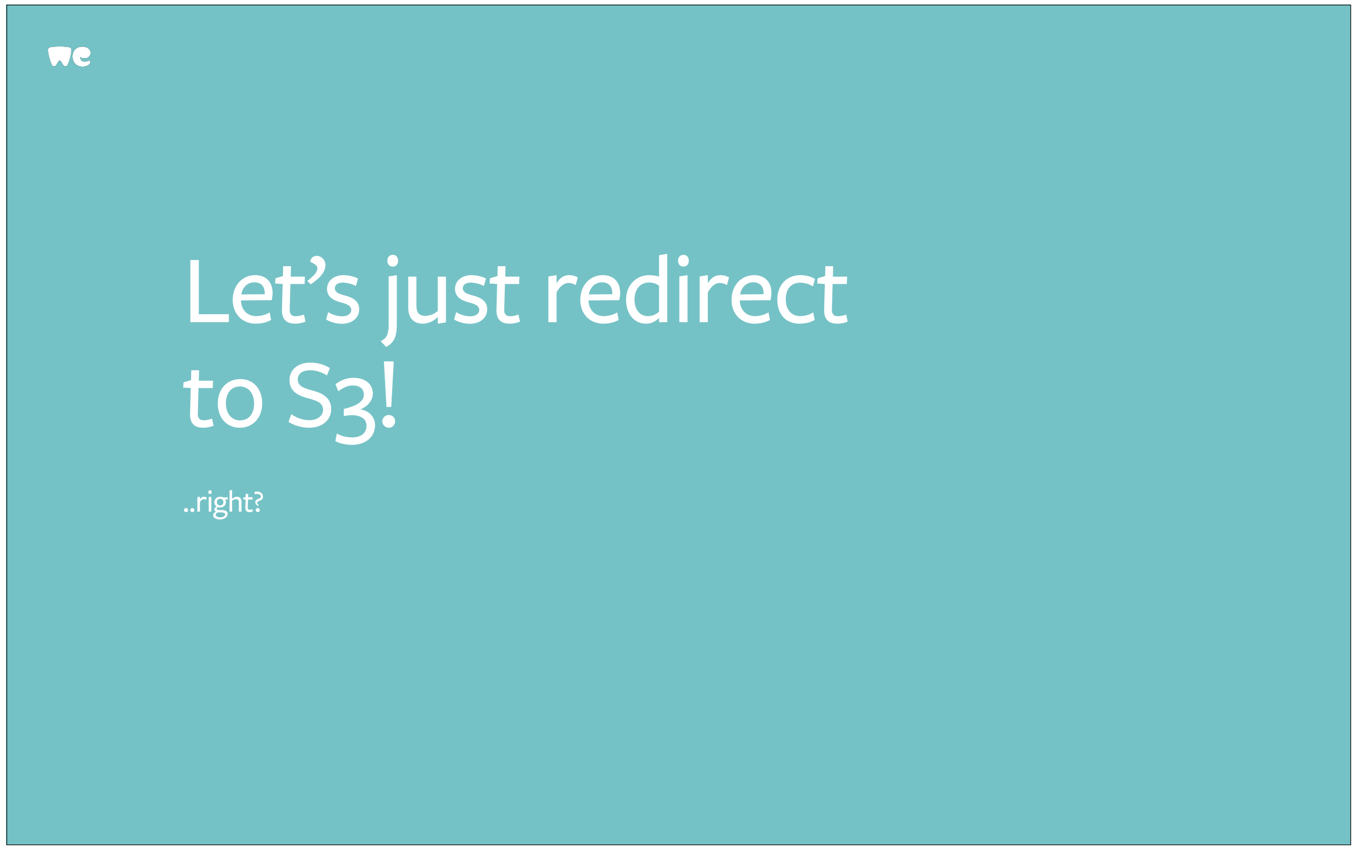
We got a lot of them.

About 15 gigabytes per second outgoing at peak times.


▼ Download total ▼

12.90 GBps

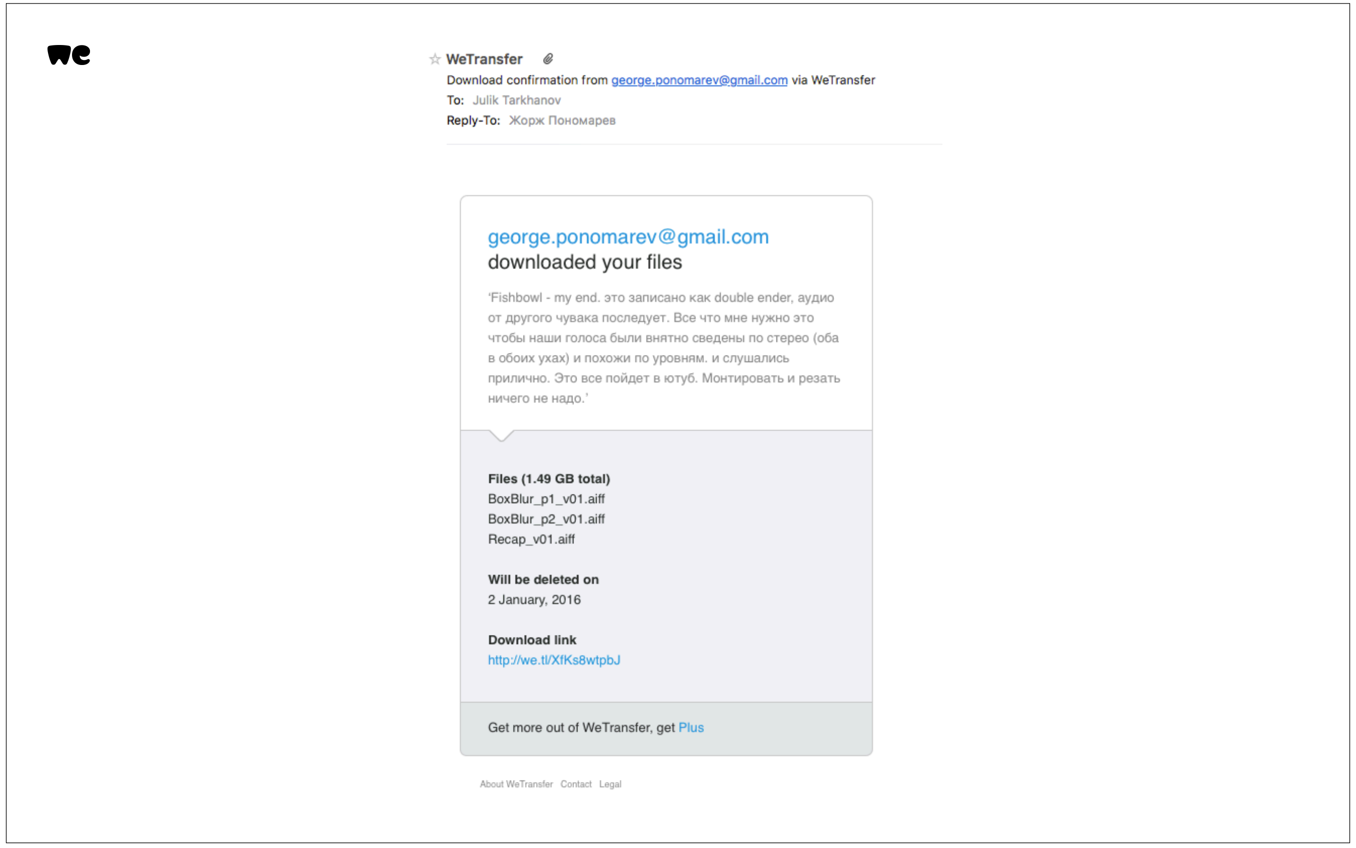
That much (this is today around lunch time)



Let's just redirect to S3!

..right?

We store everything on S3, so we could just redirect the downloads to S3 and be done because….



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

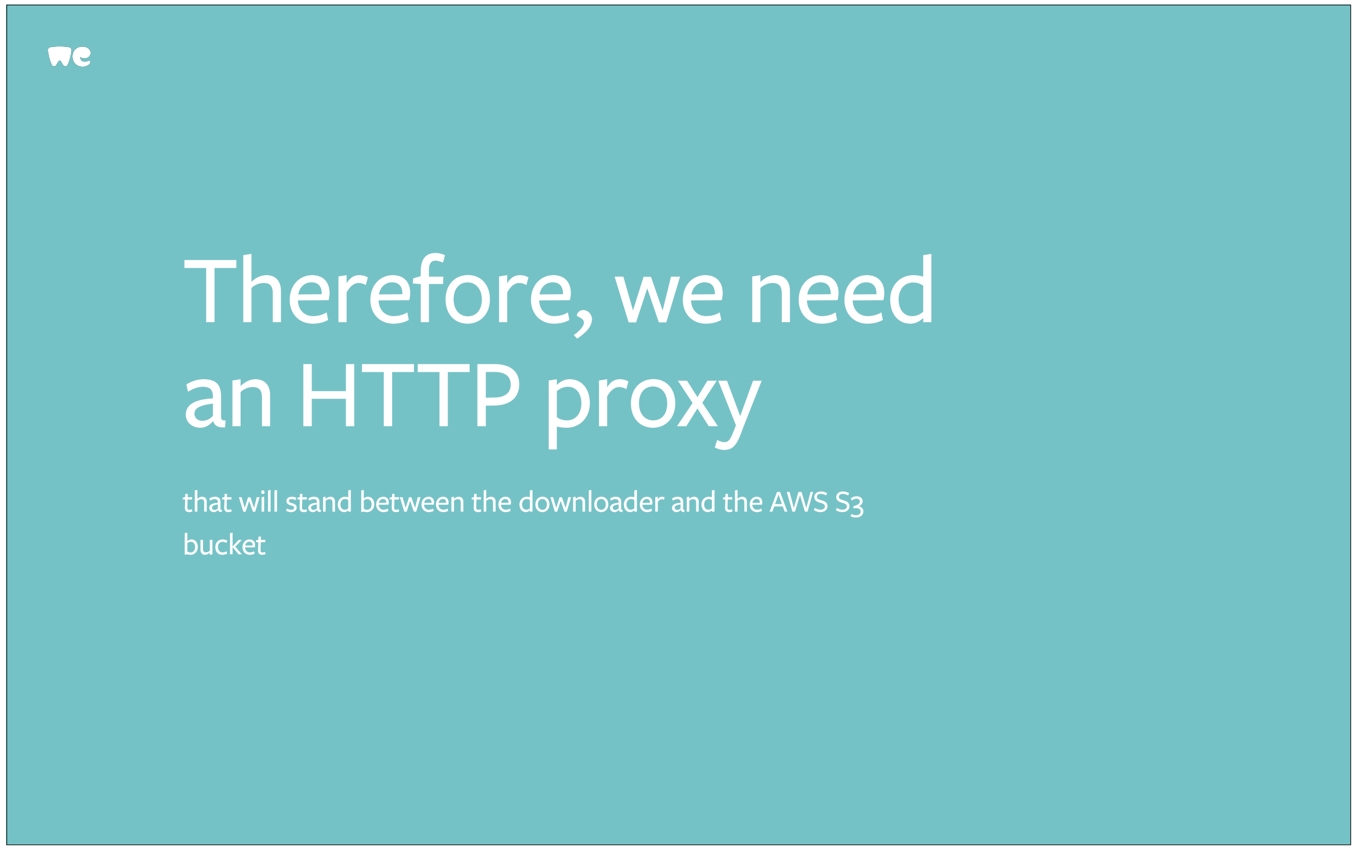
Will be deleted on
2 January, 2016

Download link
http://we.tl/XfKs8wtpbJ

Get more out of WeTransfer, get Plus

About WeTransfer Contact Legal

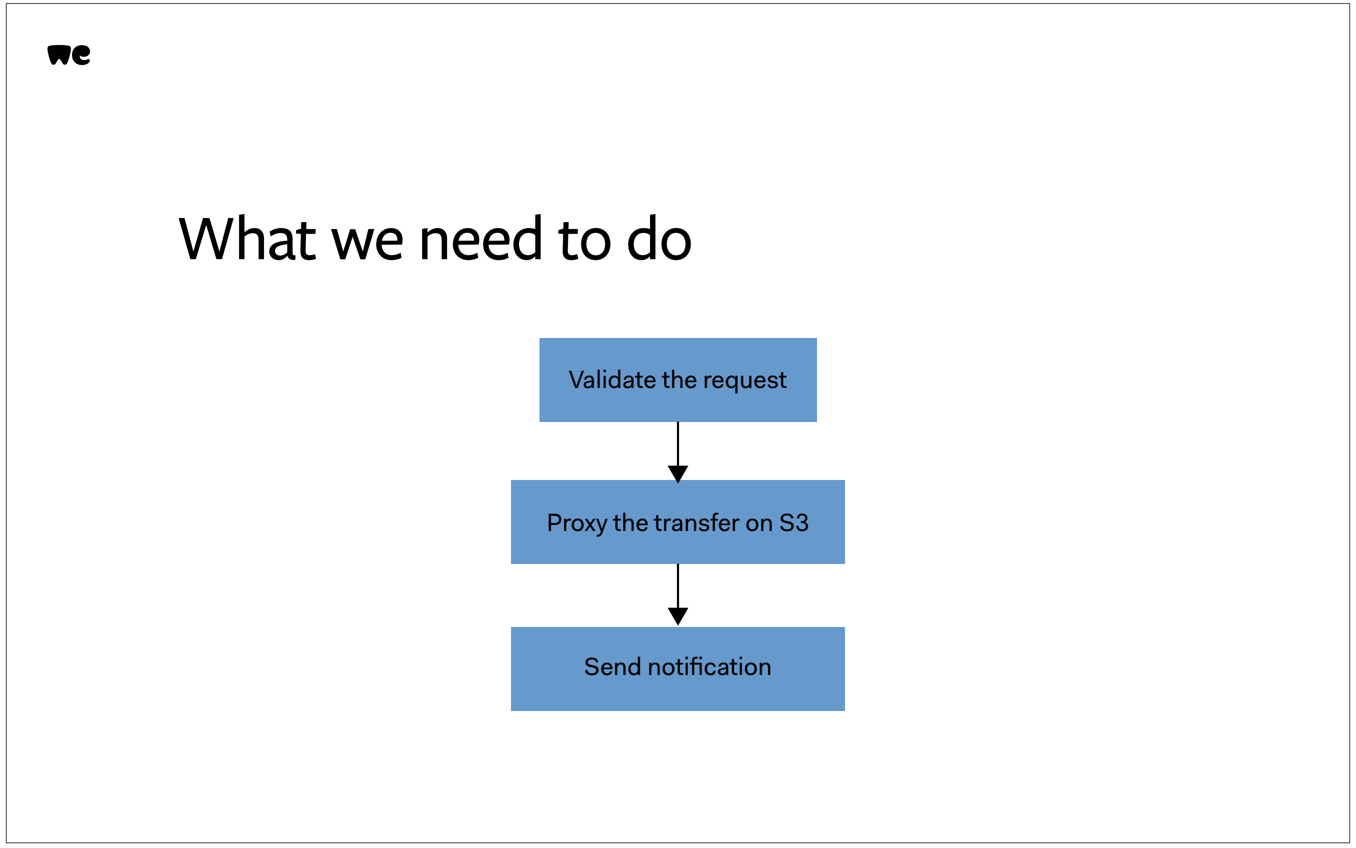
When people download a transfer, we send this email. And this is a feature people actually pay us for. For us "downloaded" doesn't mean "clicked the download button", it means "downloaded the entire thing". And this email can only be sent once we are positively sure the very last byte of the download has been sent to the recipient. Since we have this requirement, we can't simply redirect to
S3 and let people download from there.



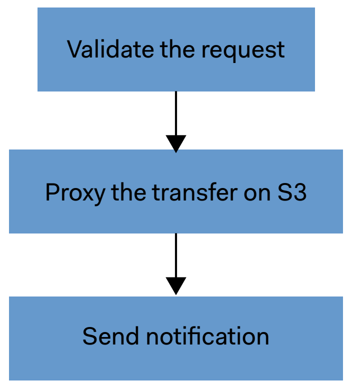
Therefore, we need an HTTP proxy

that will stand between the downloader and the AWS S3 bucket

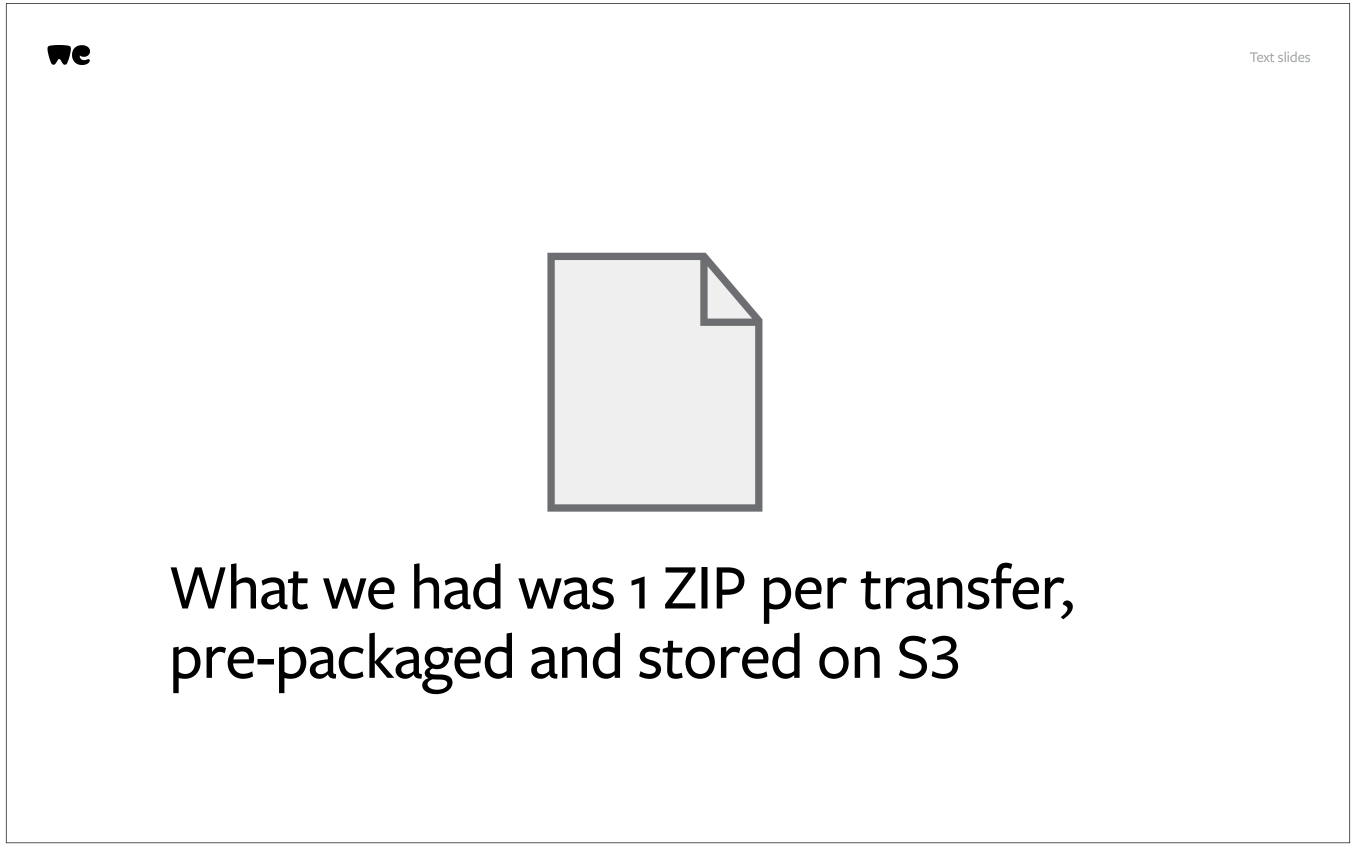
So it means we need a proxy between S3 and the recipient - something we call a "download server". The download server performs two tasks:



What we need to do

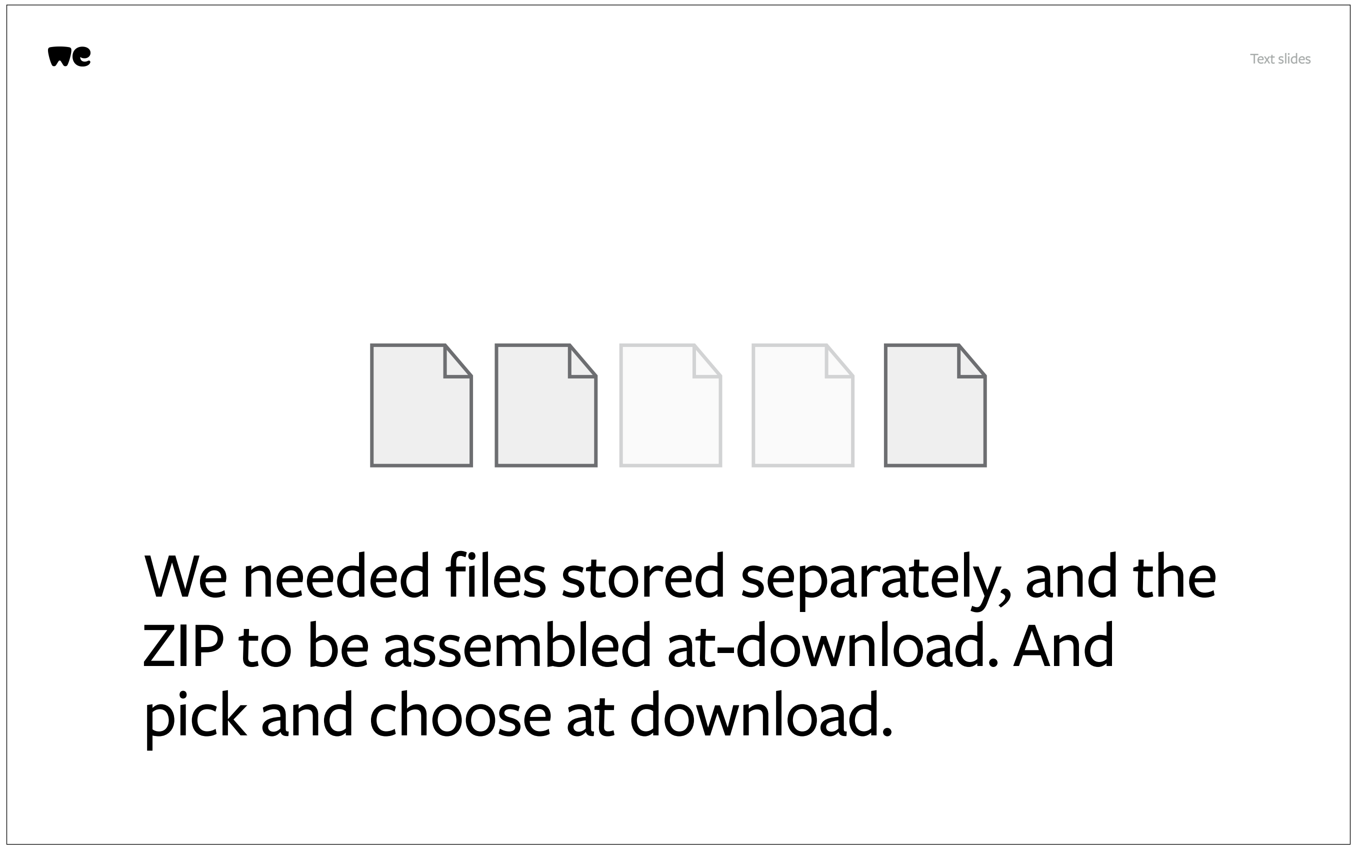


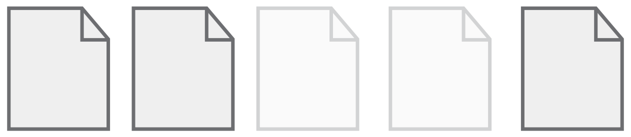
Basically we have to write an HTTP proxy server with edge includes. Now you would say doing it in Ruby is a very bad idea.



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

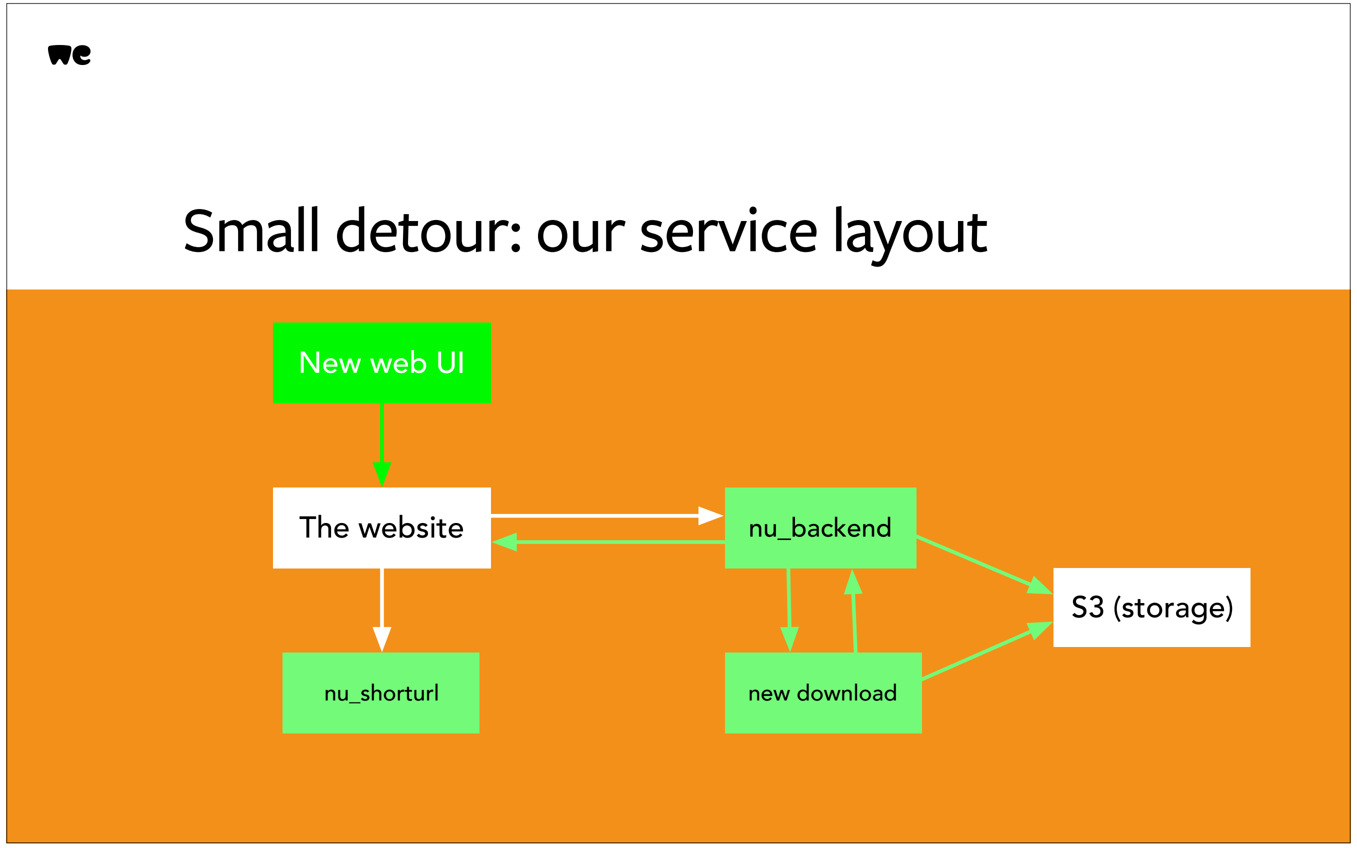
The existing PHP solution was not enough, it only did 1 file (one transfer pre-packaged as a ZIP).





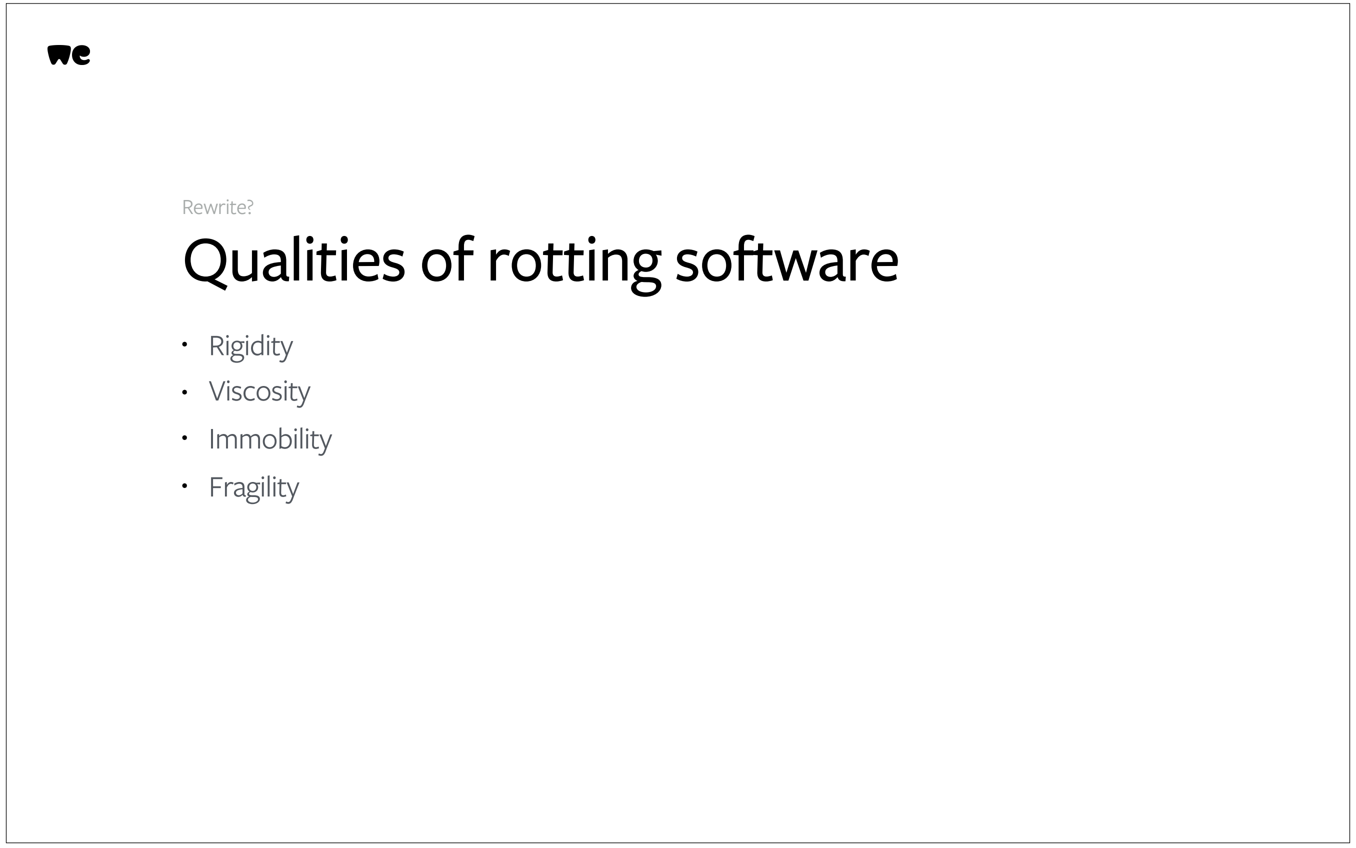
We needed files stored separately, and the ZIP to be assembled at-download. And pick and choose at download.

The existing PHP solution was not enough, it only did 1 file (one transfer pre-packaged as a ZIP). And we needed the user to be able to pick and choose which files she wants to download.



Small detour: our service layout

Our rewrites – the green things in the middle is what has been rewritten from PHP to Ruby in the last 18 months

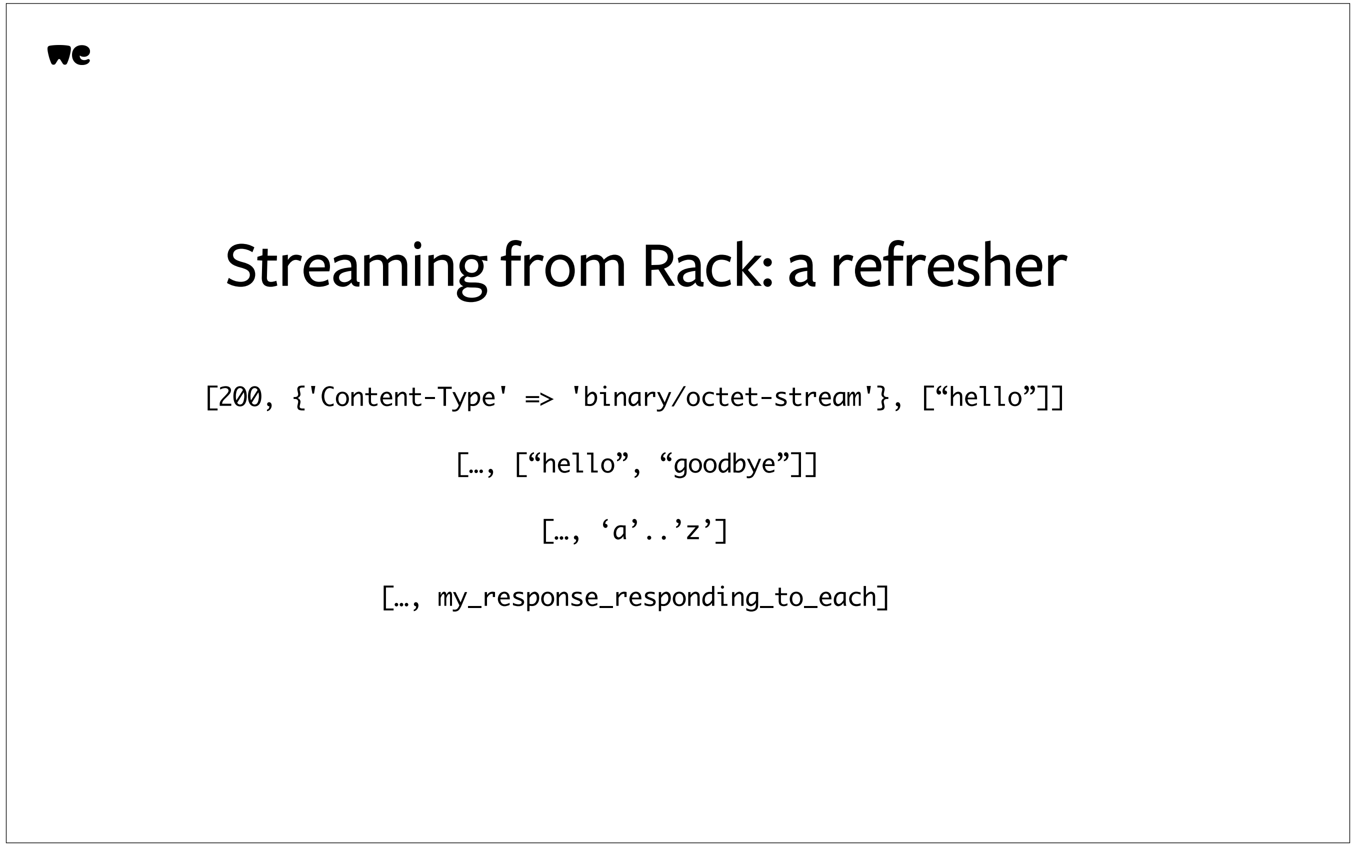


Rewrite?

Qualities of rotting software

- Rigidity
- Viscosity
- Immobility
- Fragility

The existing download solution worked, but the thing was not changeable. 0 unit tests, no ZIP library we could lean on, a stack of dependencies we wanted to step away from (PHP-FPM and FCGI, old AWS SDK)

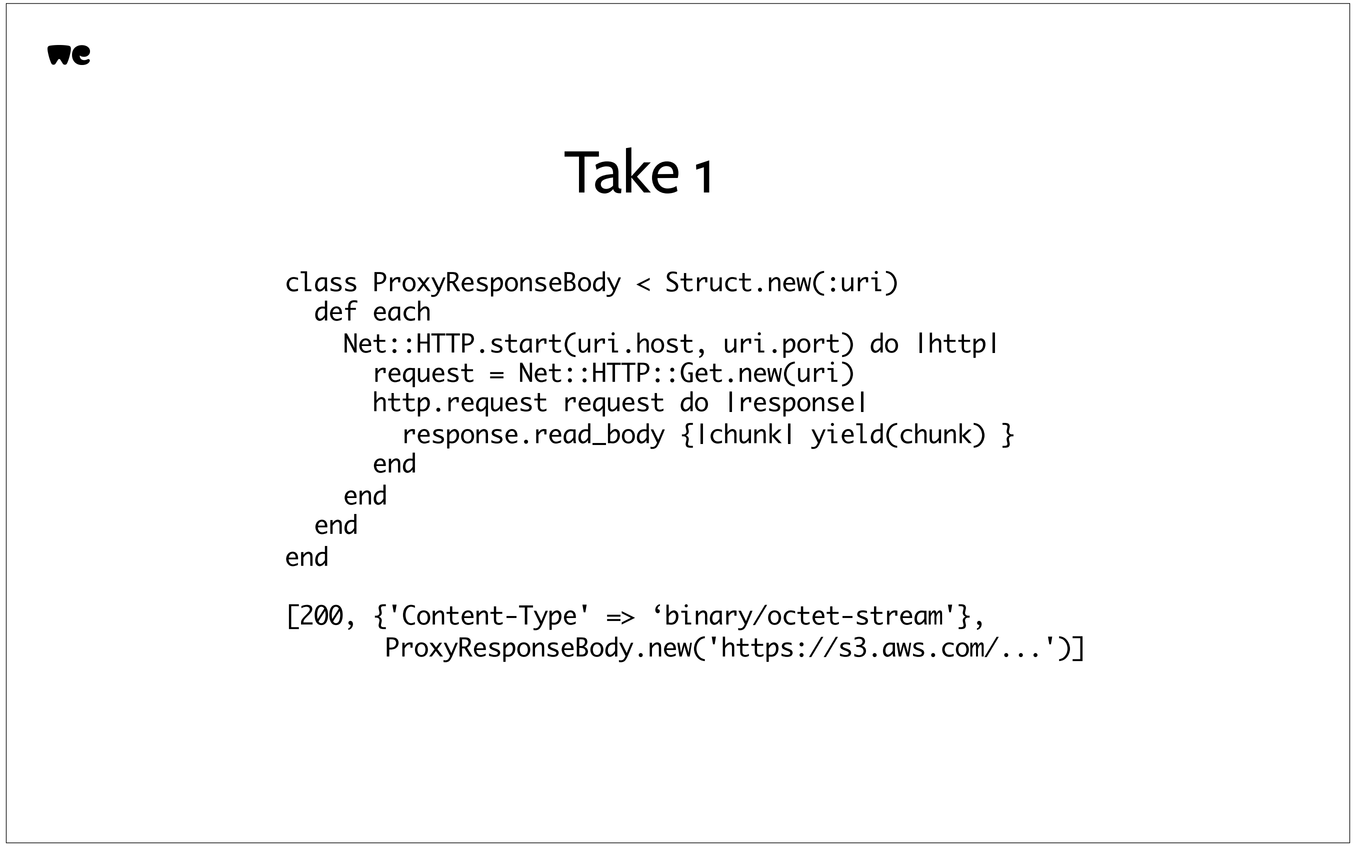


Streaming from Rack: a refresher

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

[…, [“hello”, “goodbye”]]

[…, ‘a’..’z’]

[…, my_response_responding_to_each]
```

So - it is a low level thing, we will build it in raw Rack.

A refresher on raw Rack does HTTP responses (because we do not need Rails to build a server like this one):

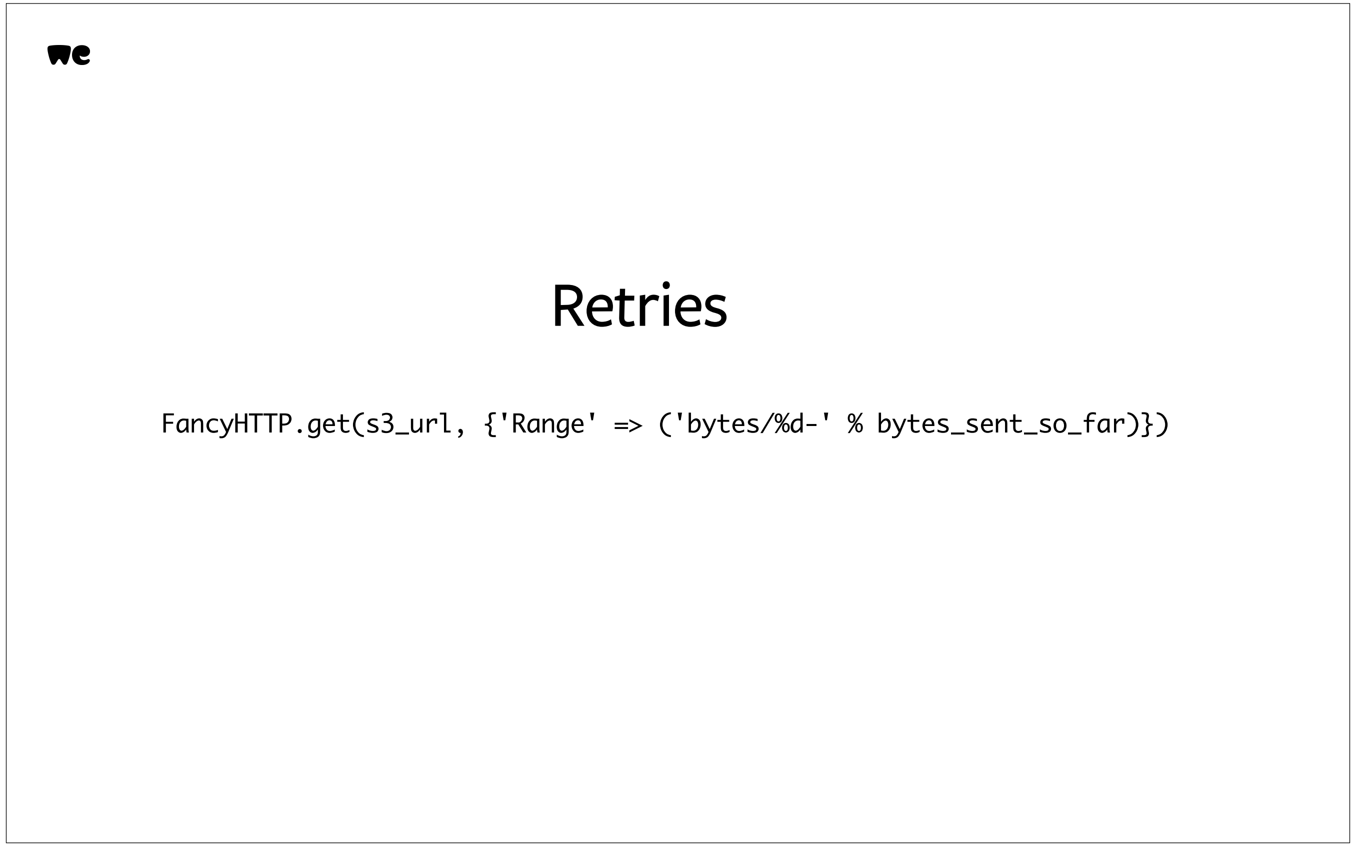


Take 1

```
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/...')]
```

So a natural idea would be to use Net::HTTP to open a connection to S3, and to stream the chunks as they get received.

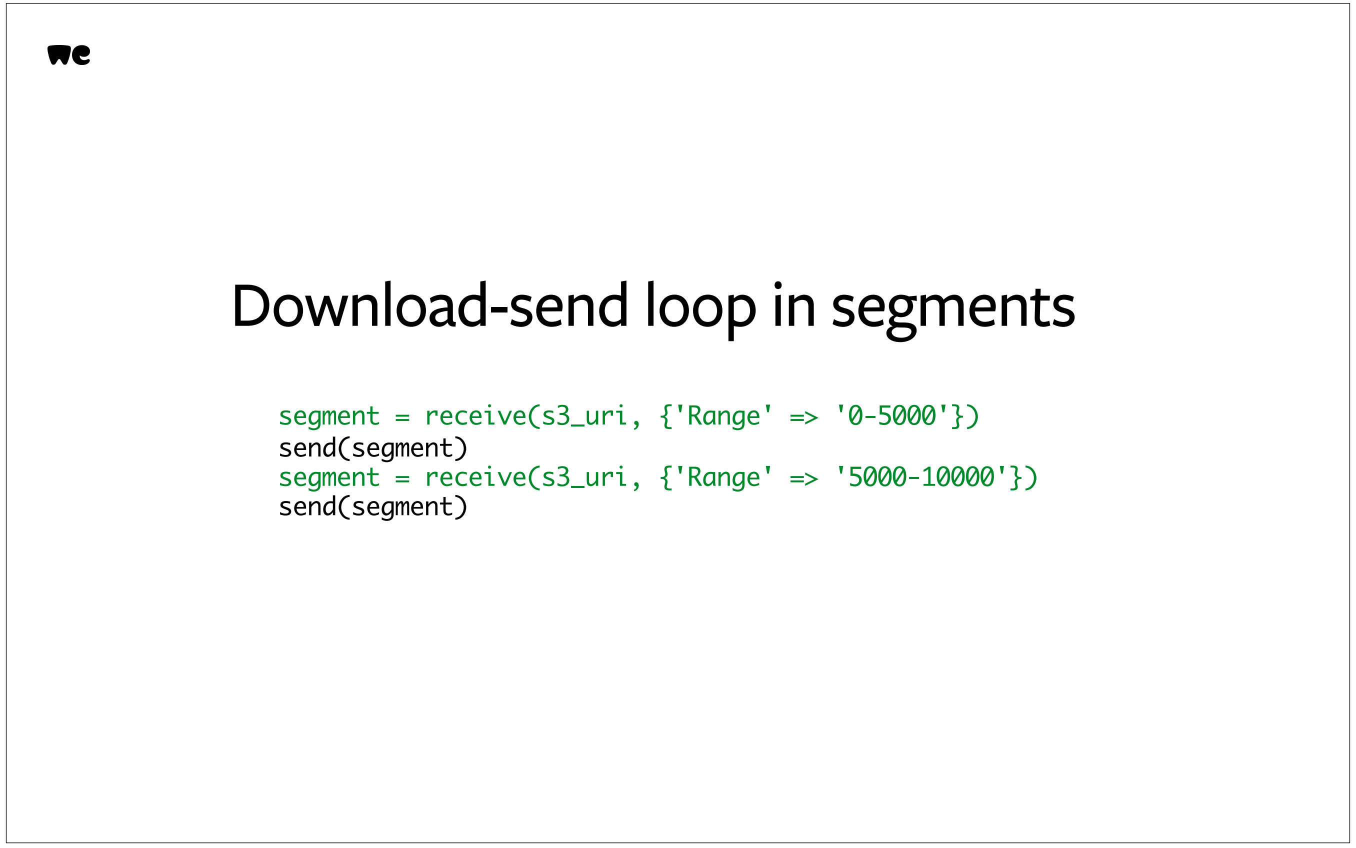
And don't get me wrong, that _does_ work, to a degree. The problem is you will end up using a TON of CPU, and you will end up using a TON of memory. See, it is the string churn



Retries

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

Also, about 0.01% of the S3 requests fail with an error. 0.01% of a few million requests per day is a lot of requests. We need to have retries in there too. We do them using the HTTP Range: headers – we request a part of the object on S3, then the next part, then the next

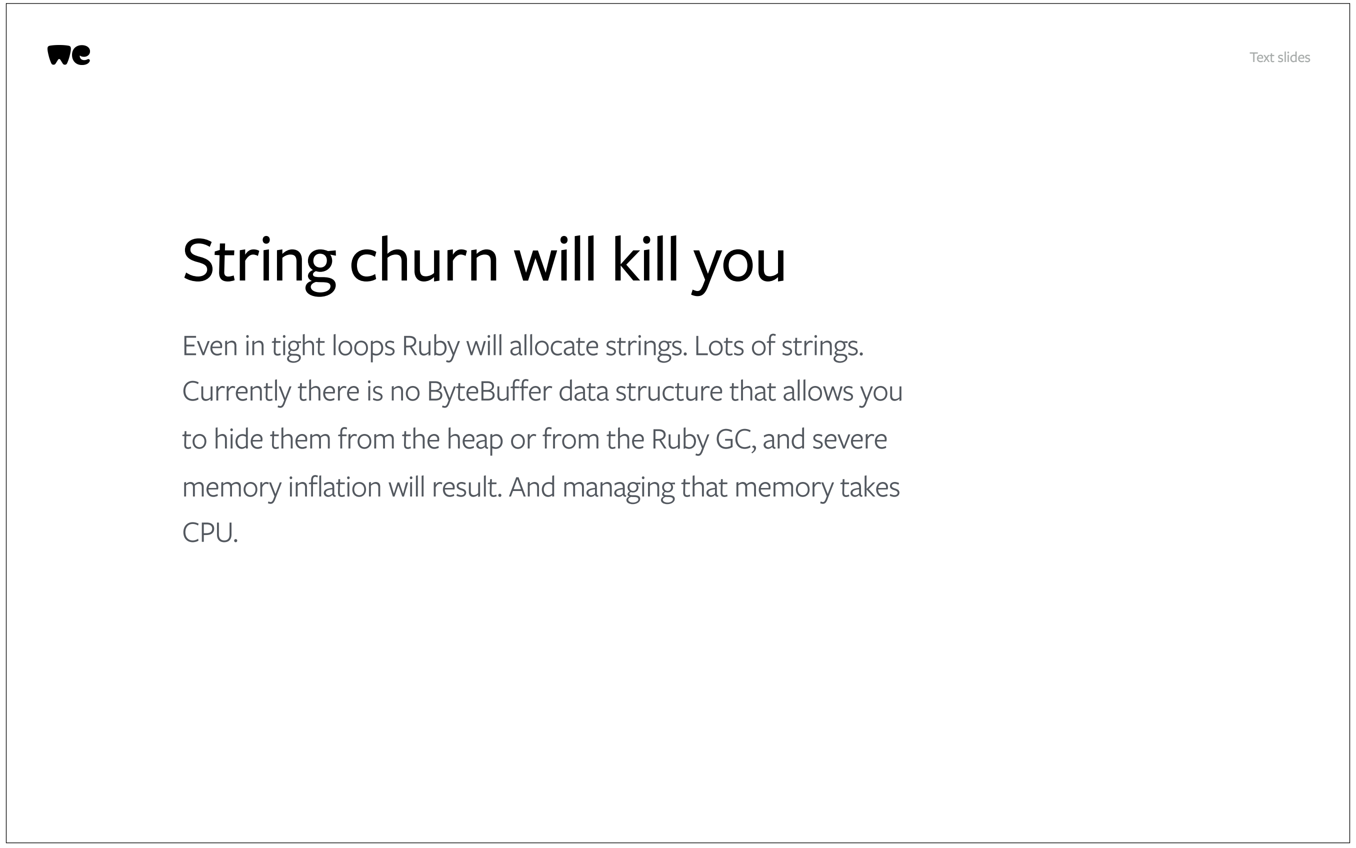


Download-send loop in segments

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

Our upstream connection to S3 is very fast – the server runs in the same AWS region as the one containing the S3 bucket. This means that we are guaranteed to be limited by the speed of the client receiving the file, as opposed to the speed of our server pulling upstream data. Therefore, we can do fetching in ranges.

If we couple the socket open to S3 for the pull request to the socket the client has opened with us for the actual download, we are effectively slowing down the S3 connection to the speed of the client. Instead, we can execute a salvo of Range requests to S3, send the received segment to the downloader, and repeat until the file has been exhausted. If we do, we also get an option of
doing retries _per segment_ as opposed to for the entire response:

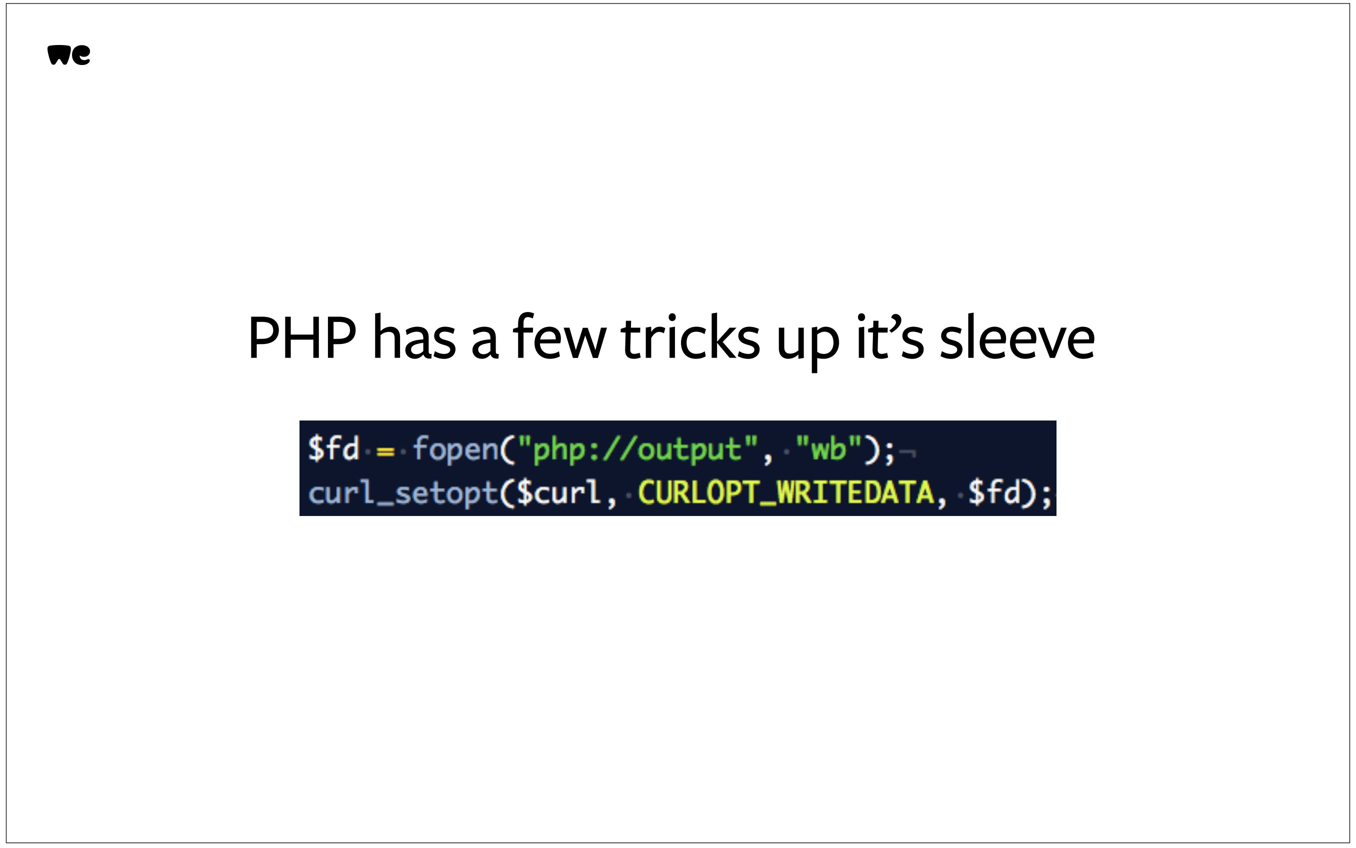


String churn will kill you

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.

Each time there is a `chunk` yielded, it allocates a String on the Ruby heap. With the actual string contents to boot.

If the connections are fast, this heap saturation will happen VERY quickly. Your Ruby process will consume a _ton_ of memory, and at some point the GC will have a really hard time keeping up. So we needed a solution.



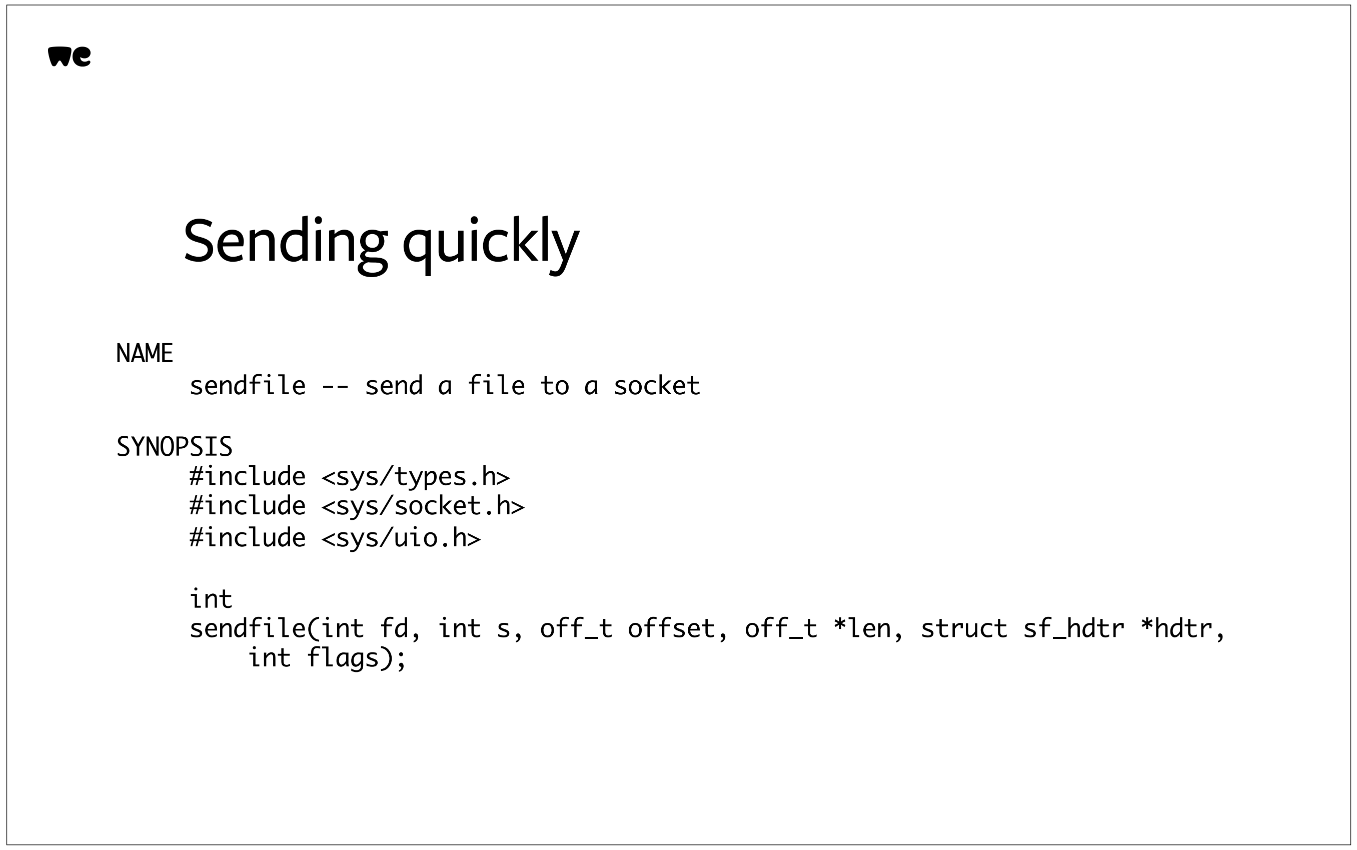
PHP has a few tricks up it’s sleeve

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

PHP had a solution. The previous version of the download server was written in PHP, and it was using a very neat idiom for doing those large-volume sends. And this idiom is _not_ possible to execute in Ruby. See if you can spot what is so special about this PHP snippet:

So, this creates a direct link between the libCURL download and the PHP output socket.

Everything that flows over this link is not going to enter the PHP virtual machine, and will not have strings allocated for it. It is effectively a socket splice (something
that Linux can do natively in kernel space now), but implemented in userspace. Strings flowing from Curl will not enter the VM.



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);
```

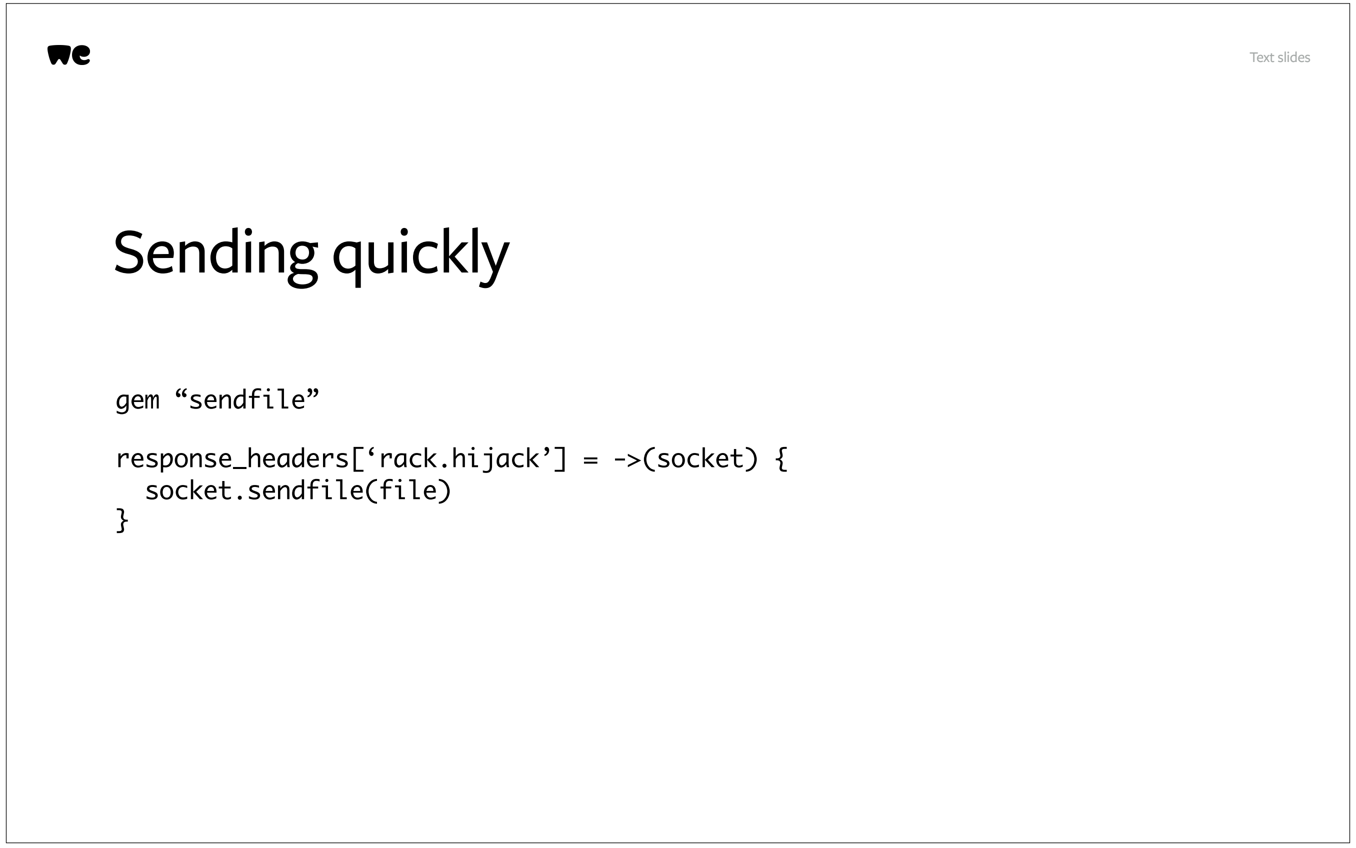
How do we store intermediate data we do nothing with? when we want it off the heap of the VM?

There is only one solution – _files._ And there is a thing to tell your OS to “send this file through that TCP socket, as fast as you can”.

It is **superfast**. It uses next to no memory. It happens in the kernel. It has a non-blocking version.

It is _not_ available in EventMachine (and other Patented Spaghetti Server Solutions Using libuv).

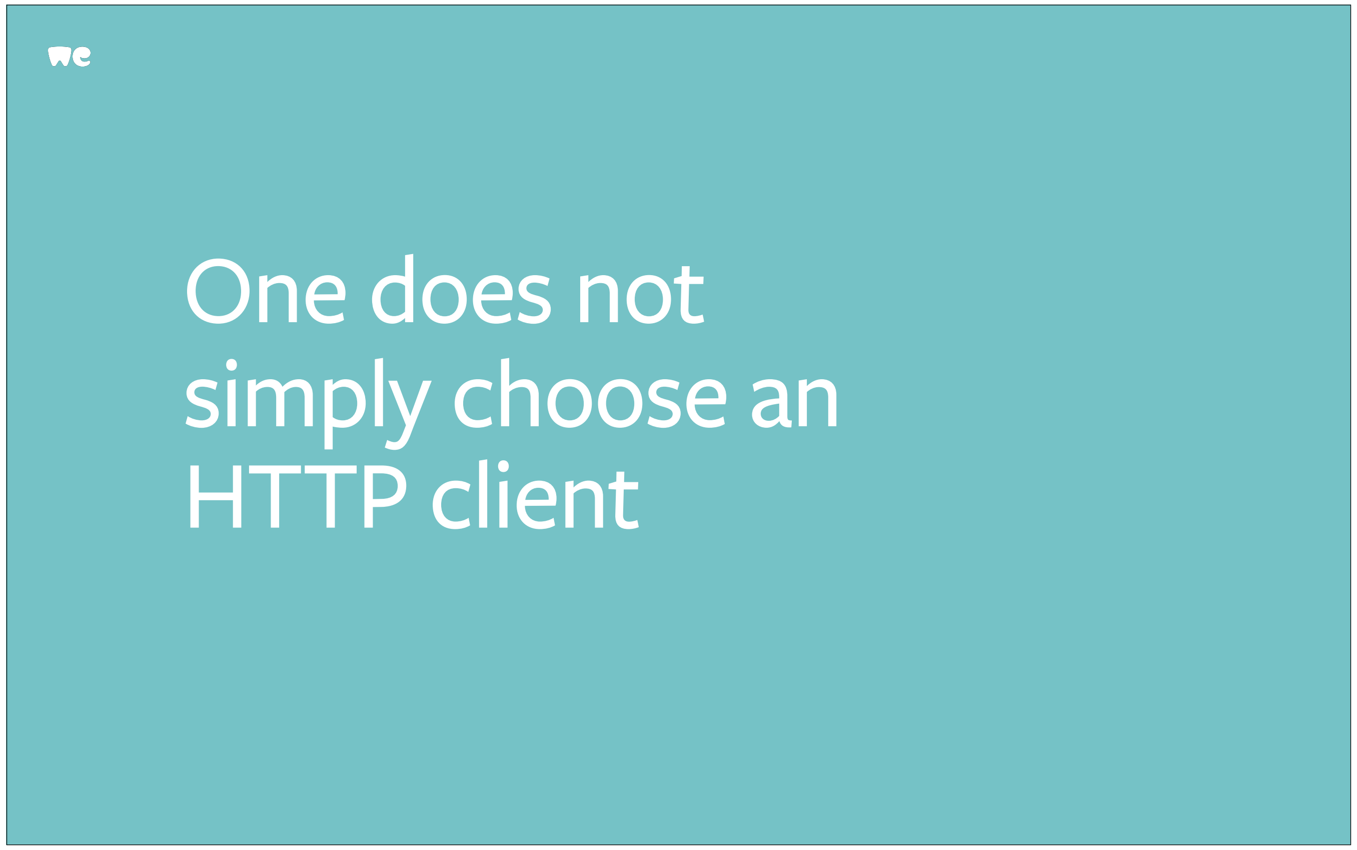
And Netflix uses exactly that – they do it from Java, but the principle is the same. If it works for Netflix, it should work for us?..



Sending quickly

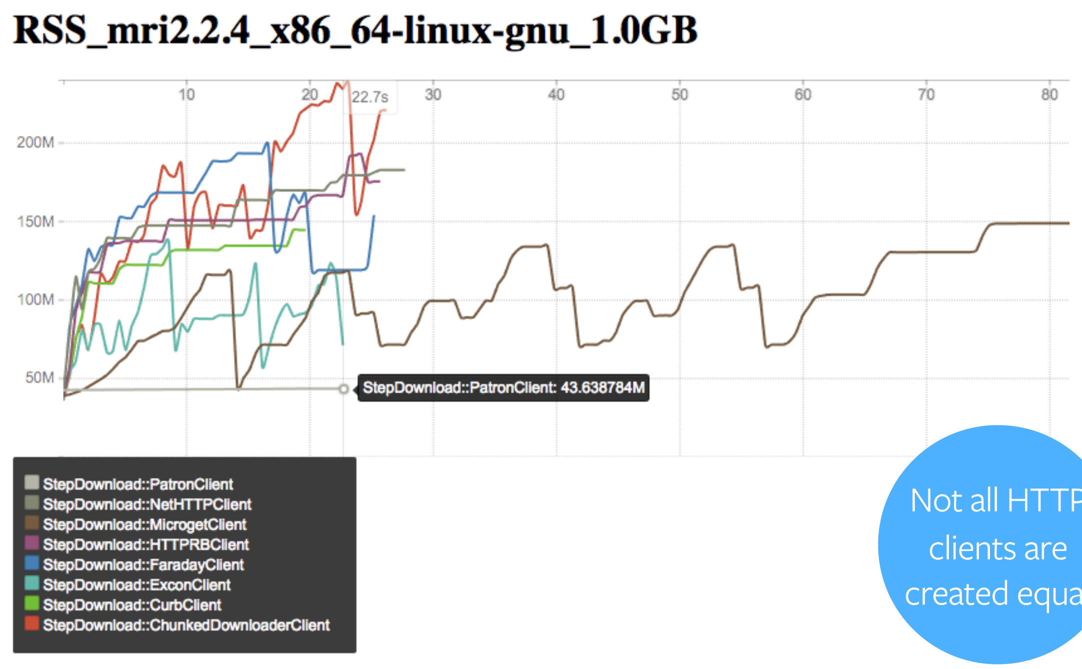
```
gem “sendfile”

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

There is a great Ruby gem from Eric Wong, the author of Unicorn, that does just that. We use it from Rack hijack – a way to work with the client TCP socket directly.



One does not simply choose an HTTP client

Now, this is the send() part. The `receive_part` is where we grab that fragment from S3.

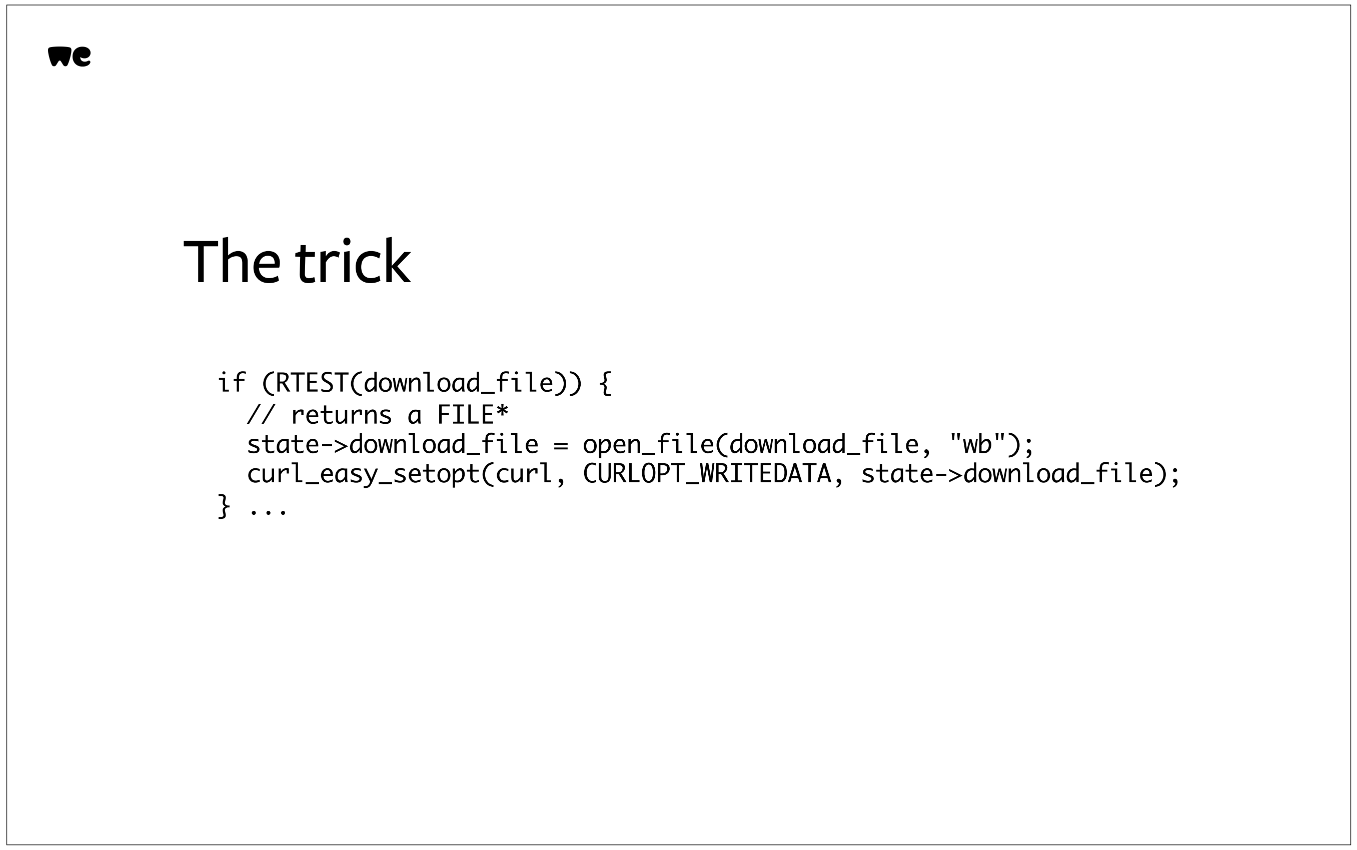


Not all HTTP clients are created equal

Here is a graph. The graph represents memory use (RSS) of a Ruby process performing this stepped download procedure (fetch a large file over HTTP in ranges). We download 1GB of data in chunks of 5MB. The data downloaded gets written to files:

On this graph we have Faraday (backend by Net::HTTP), we have Curb... and as you can see all of these clients saturate the MRI heap very quickly, because they pump those strings we do nothing with. The strings get dumped to files on disk (here we write them to `/dev/null`), but still – their contents has to travel through the Ruby heap along the way – and this is what was bringing us _down_.

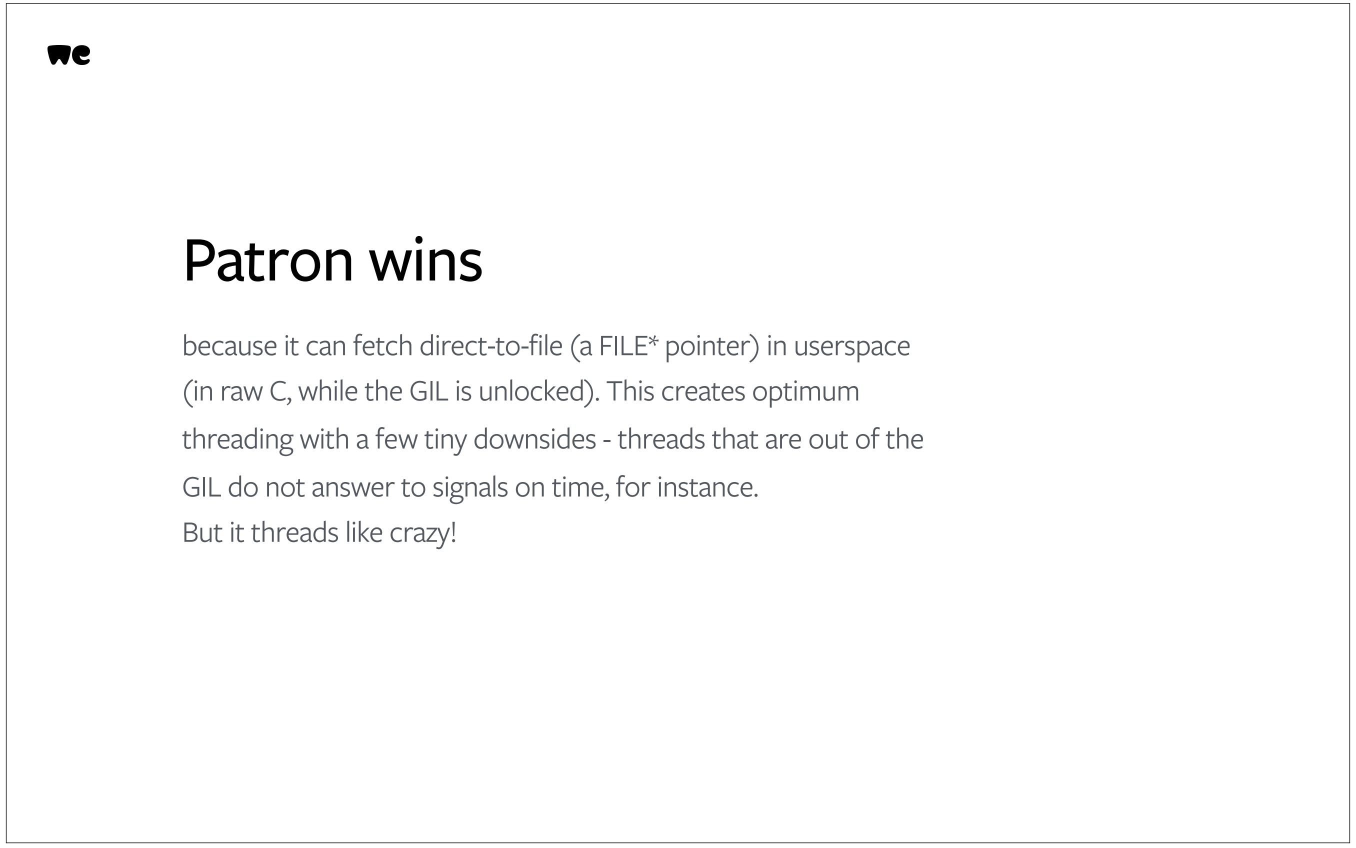
Now look at this graph right at the bottom. This graph here. This is Patron. Patron, for those who are not familiar, is the most _to the metal_ binding to libCURL for Ruby. It does not have as many features as Curb, and it does not use FFI like Excon does – it's a very down-to-earth Ruby extension with a `.c` file in it. And it has a shortcut:



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);
} ...
```

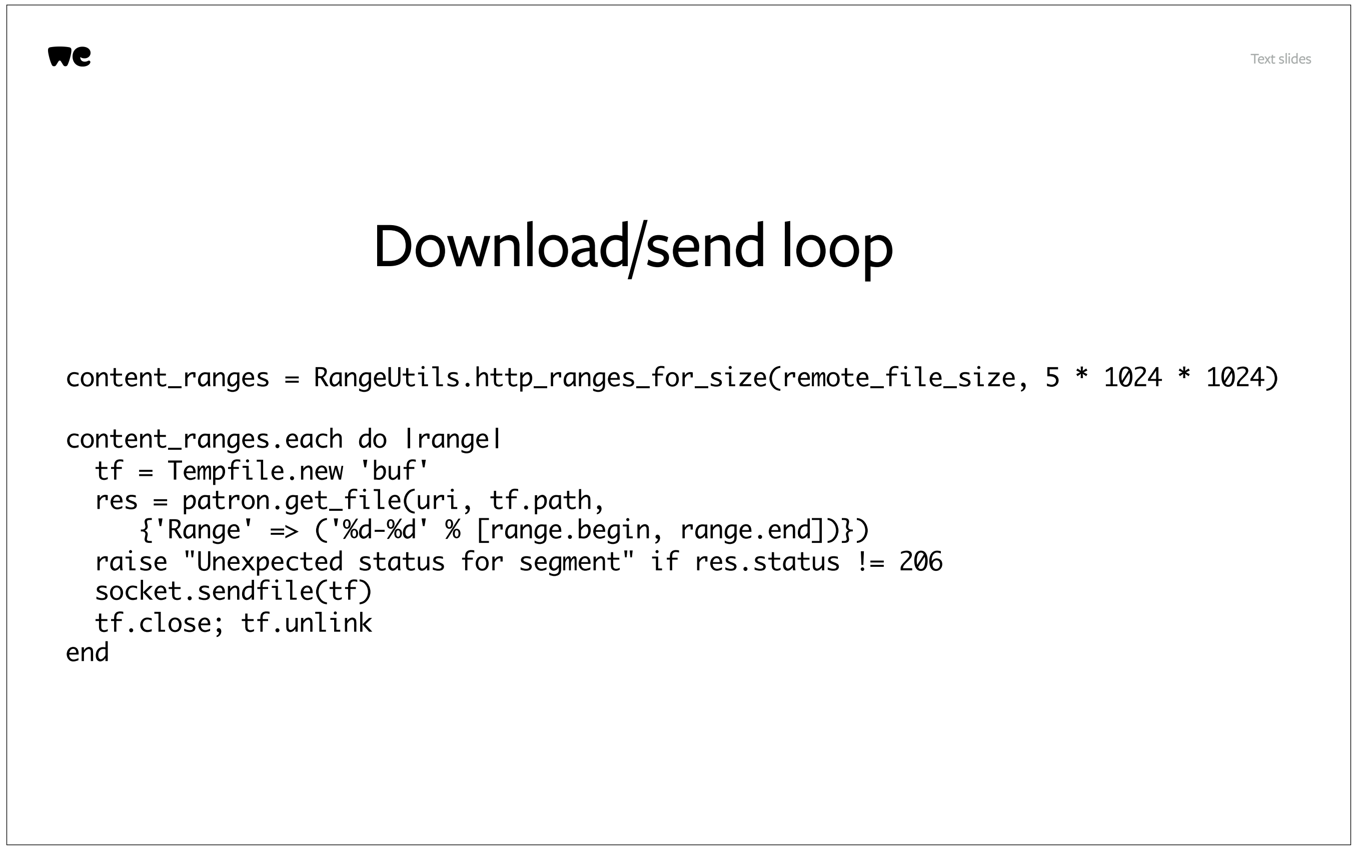
The data gets written to a file directly in C, and the writes are managed by libCURL. The Ruby VM never even sees that data going through. You just tell Patron that you will be downloading into a file, and it takes care of the rest.



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!

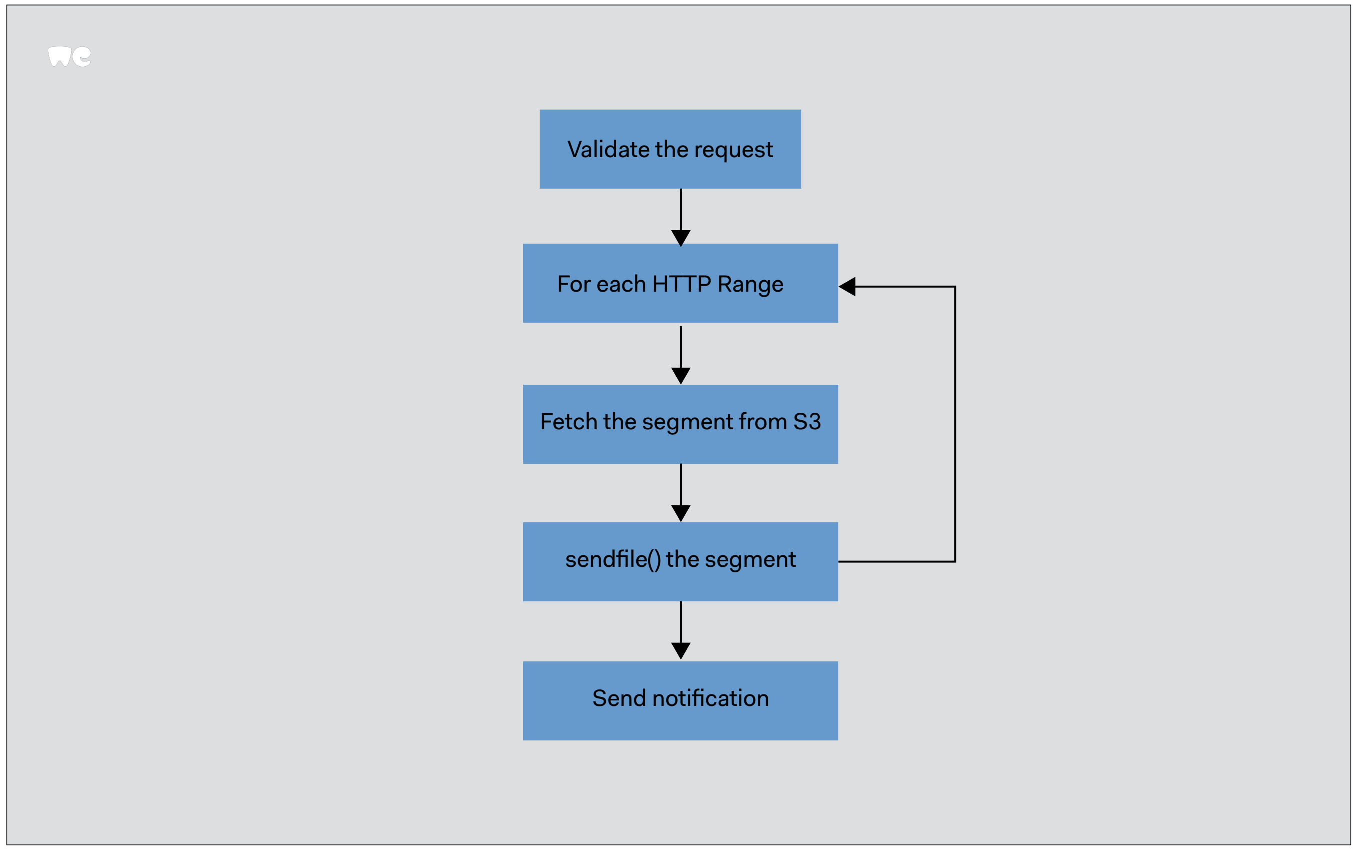
What this `get_file` method does, is it sets a FILE pointer in userland C as the destination for the CURL writes during download. Conversely, it means that all the data buffers (C strings) flowing through CURL will _never_ end up on the Ruby heap - they will be written directly to the given file on the filesystem. This is _precisely_ why this graph stays at the ridiculous near-zero growth all along this plot - MRI does not even _arrive_ at performing a GC cycle since the memory usage stays so low. Before you ask - no, _none_ of the HTTP clients we tested have this feature in Ruby-land. And none have a feature of writing direct to socket - what we used to use in PHP (coming back full circle). So not all things in PHP-land are bad things - it has some exceptional tricks up it's sleeve to work some hard problems very efficiently.

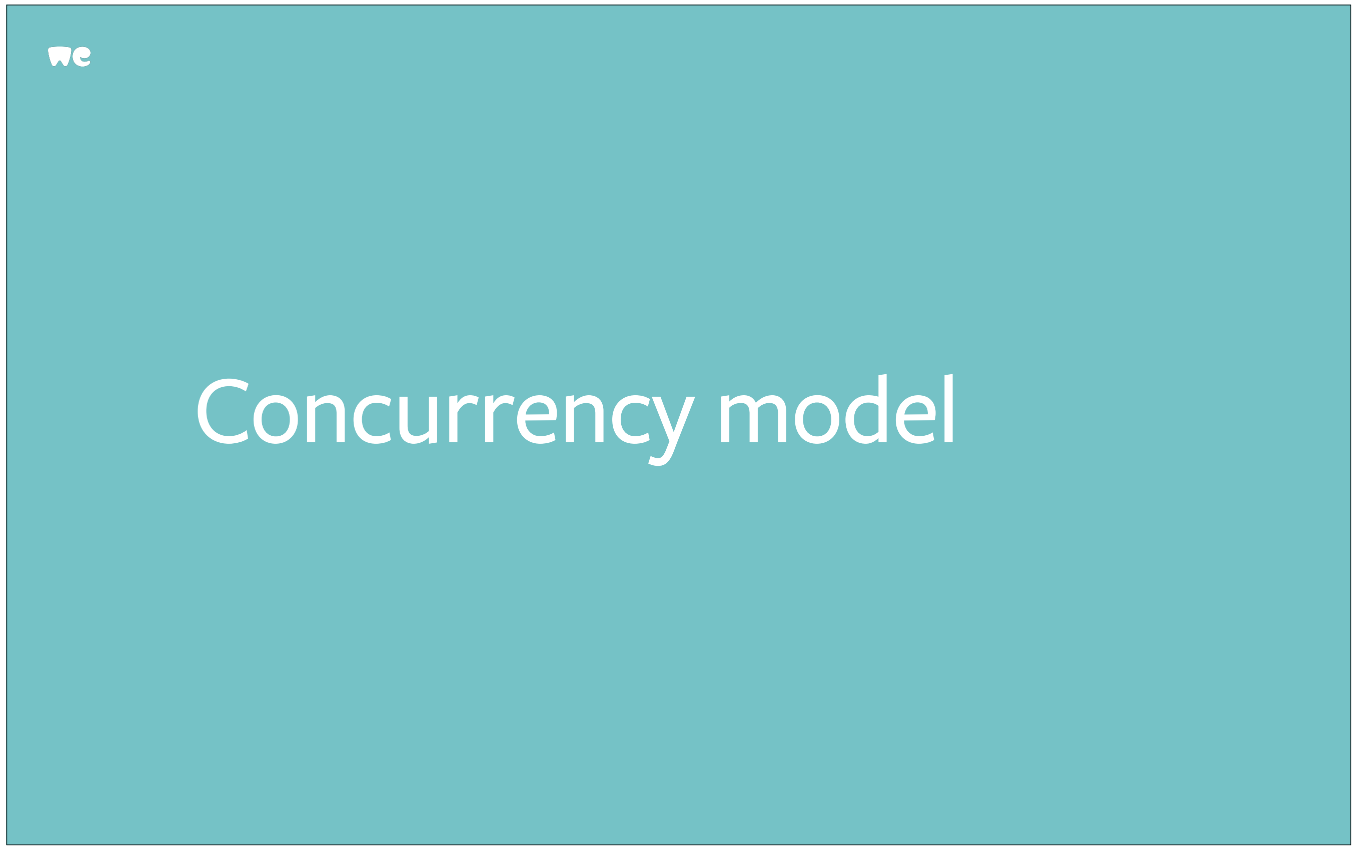


Download/send loop

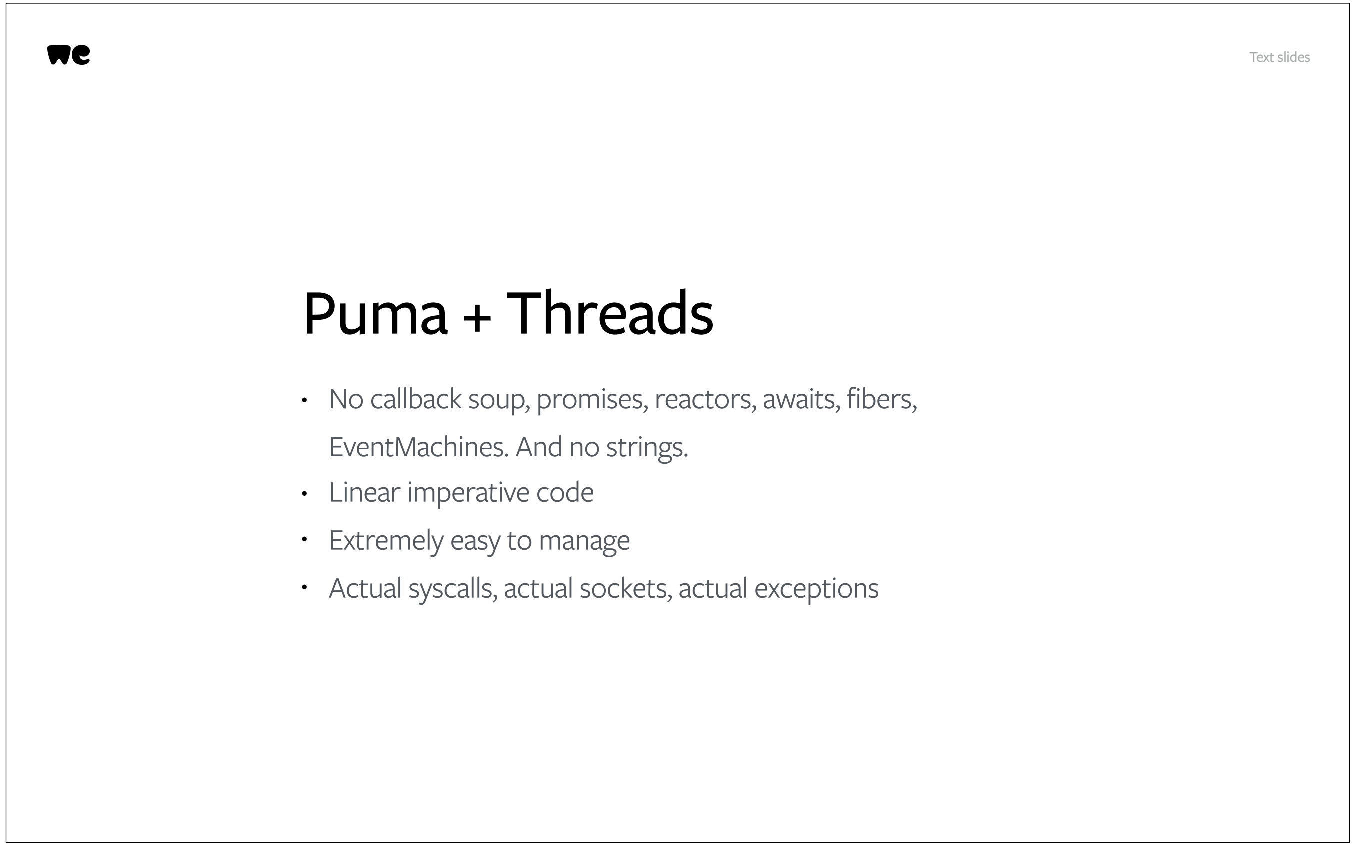
```
content_ranges = RangeUtils.http_ranges_for_size(remote_file_size, 5 * 1024 * 1024)

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
```

So, Patron it was going to be. And we were going to use `sendfile()` to send the data over the socket once Patron has pulled it in. Our send then looks like this:


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



Concurrency model


Text slides

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

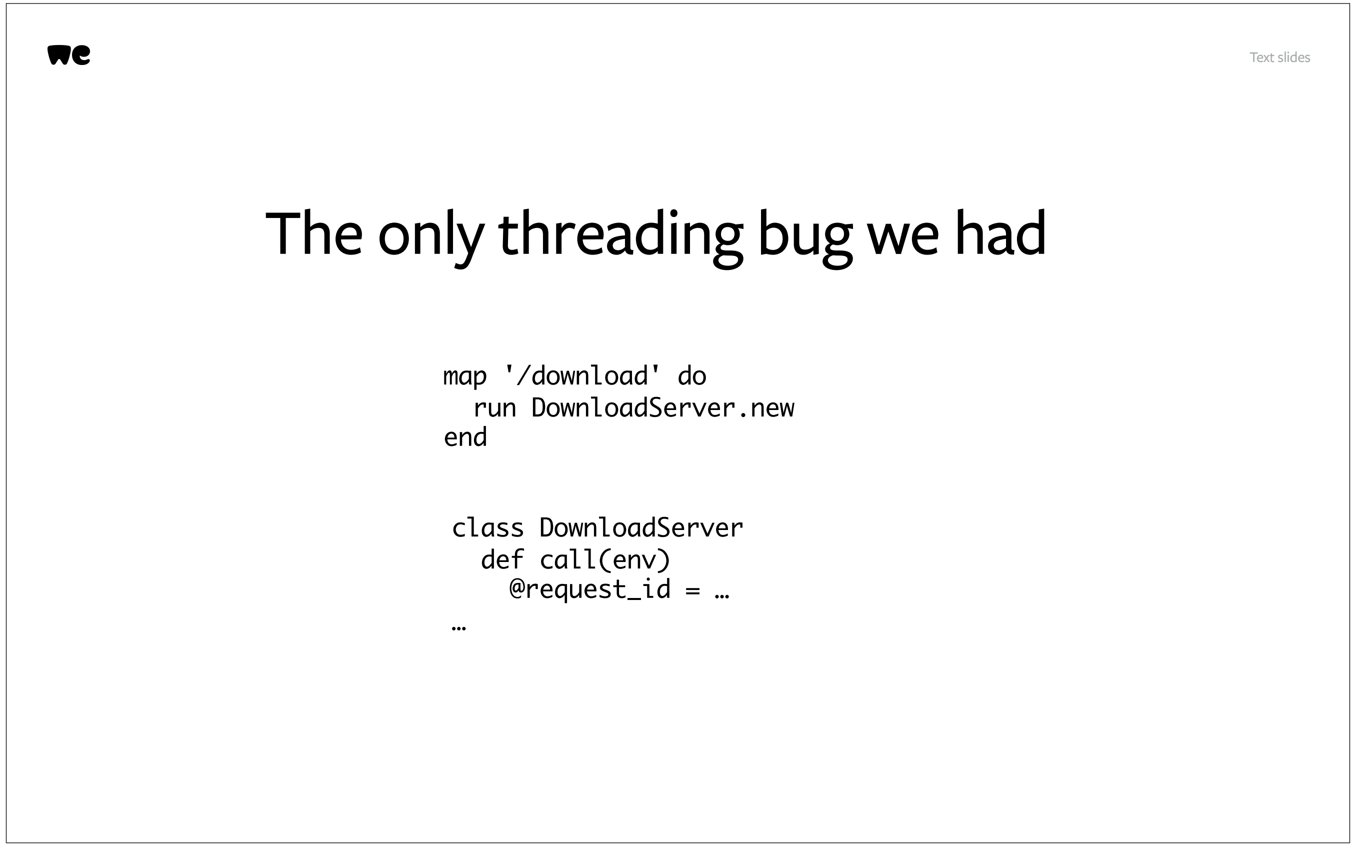
Now, this will _not_ work with evented servers. EventMachine has _NO_ way to give you the actual damn socket, and to recover that socket once you are done. It will _buffer_ your strings during the event loop, and it might create congestion, and has all the same problems the previous implementations had - PLUS you have to program with callbacks. Welcome and hail The Spaghetti Monster, accompanied by Zalgo and a lot of failed gem compiles.
So for us the path was clear:

* We were going to use threads (one thread per client, clear flow control etc.)
* We were going to use Rack hijack
* We were going to use `sendfile()`
* We were going to use Patron

Threads are not all roses because they allocate a huge Ruby stack. One thread (one person) consumes
8 megs of RSS memory at the very minimum. So servicing a thousand people will require, at the very least, having 8GB of RAM just to satisfy this constraint. This is our _only_ gripe with threads. What we get in return, however, is much more precious:

  * Normal exception handling
  * All the standard non-blocking UNIX IO (and all libraries!)
  * No spaghetti code
  * No callbacks, futures, promises, awaits and event loop congestions (well ok, we do callbacks - more later)

* No EventMachine
* No Ruby strings



The only threading bug we had

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


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

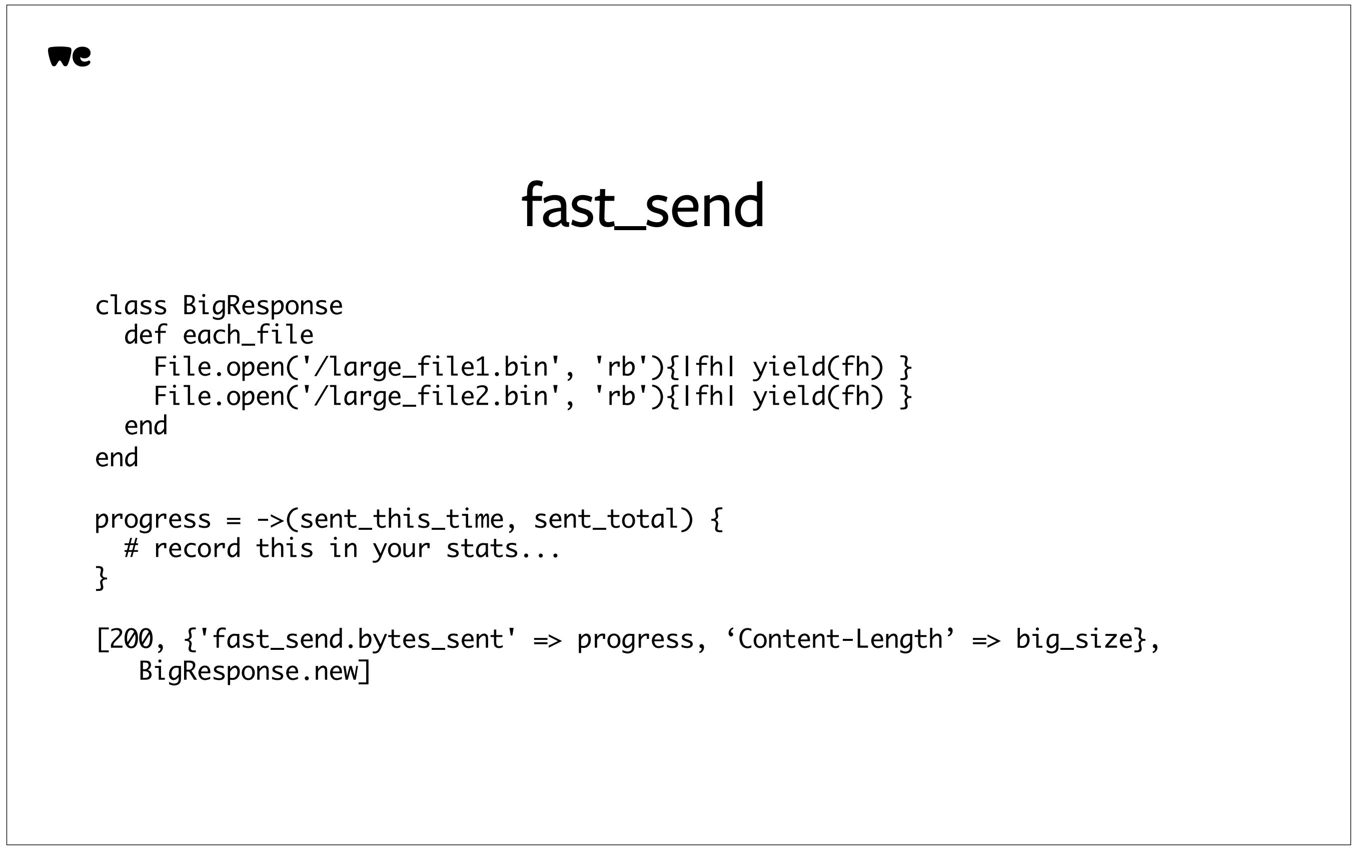
# Downsides

Threads. Not threads as a programming model. Threads as a programming model are absolutely fine as long as you are careful sharing state between them. We had exactly _one_ thread safety bug during our dev cycle, and here is how it looked:

```
# config.ru
map '/download' do
  run DownloadServer.new
end
```

This object that gets created in `DownloadServer.new` will be _shared_ between all the threads Puma spins up. This means that if you use instance variables in this application you are in for _very_ unpleasant surprises. Just don't share the application between threads. Use a Proc that instantiates a new app for every request:

```
  map '/download' do
    run ->(env) { DownloadServer.new.call(env) }
  end
```



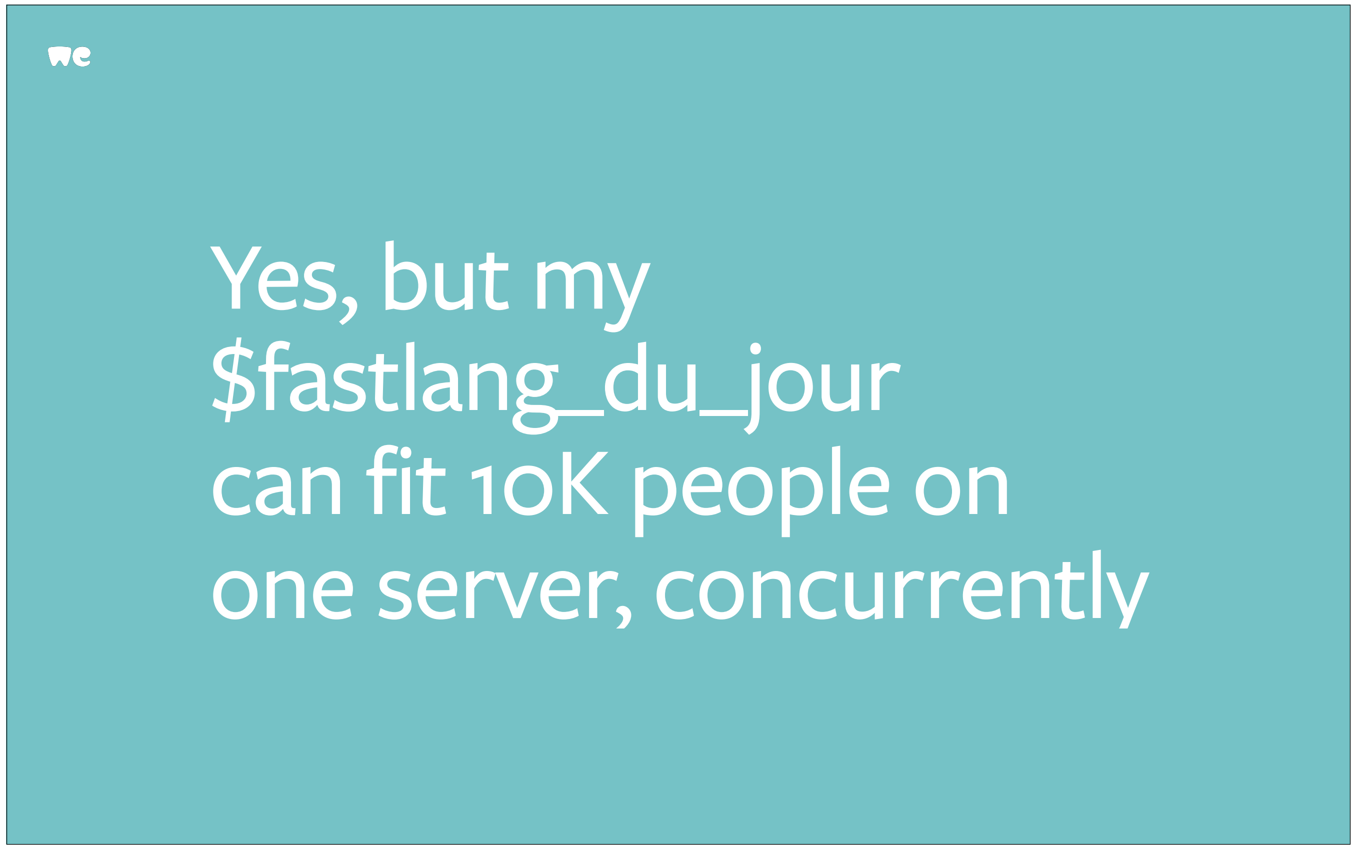
fast_send

```
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]
```

So we productized and gemified the “Send via files” thing, and you can use it too now.

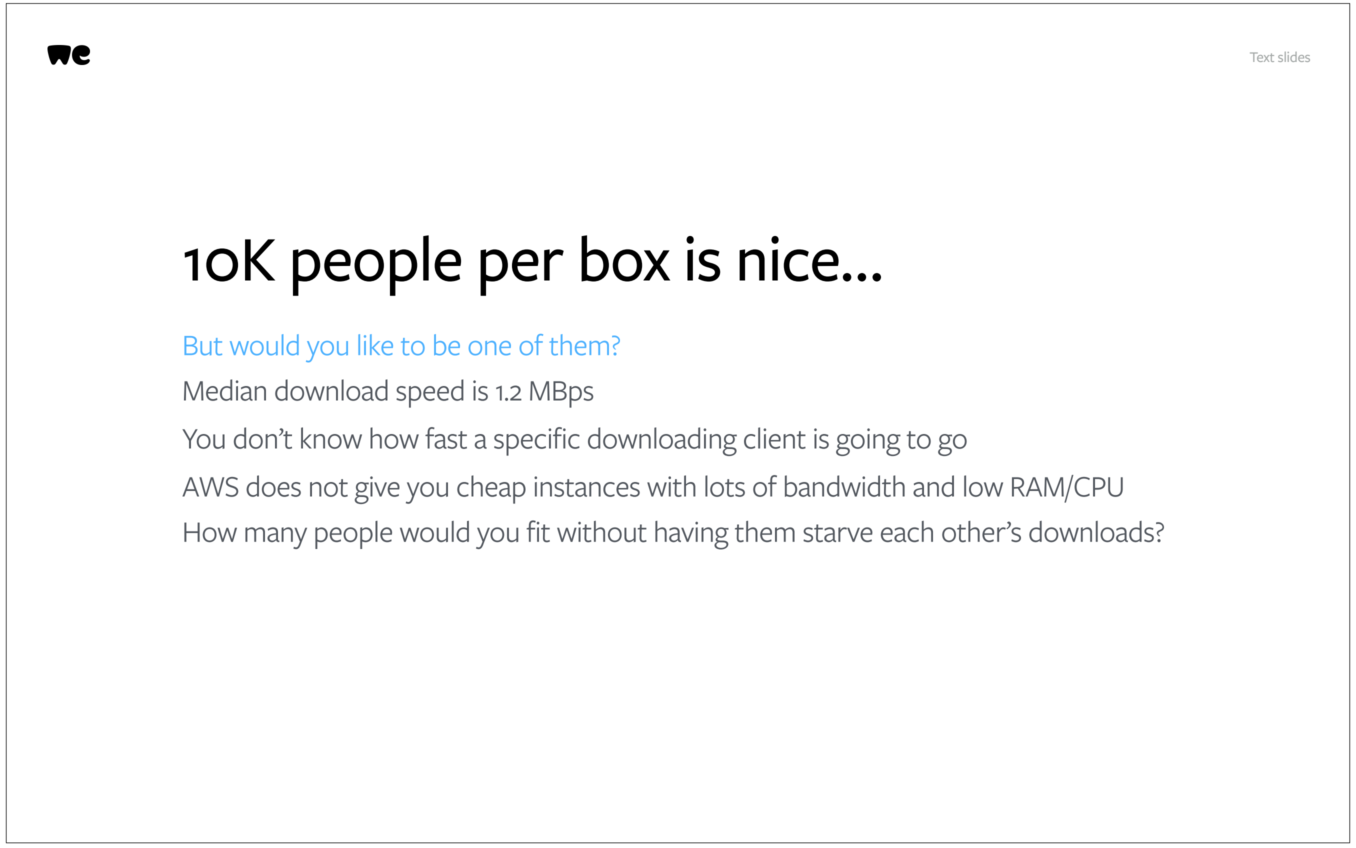
fast_send

https://github.com/WeTransfer/fast_send



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

Now there will be people saying that “Scala|Rust|Go|Java|Clojure|Elixir|Erlang can…”


Text slides

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?

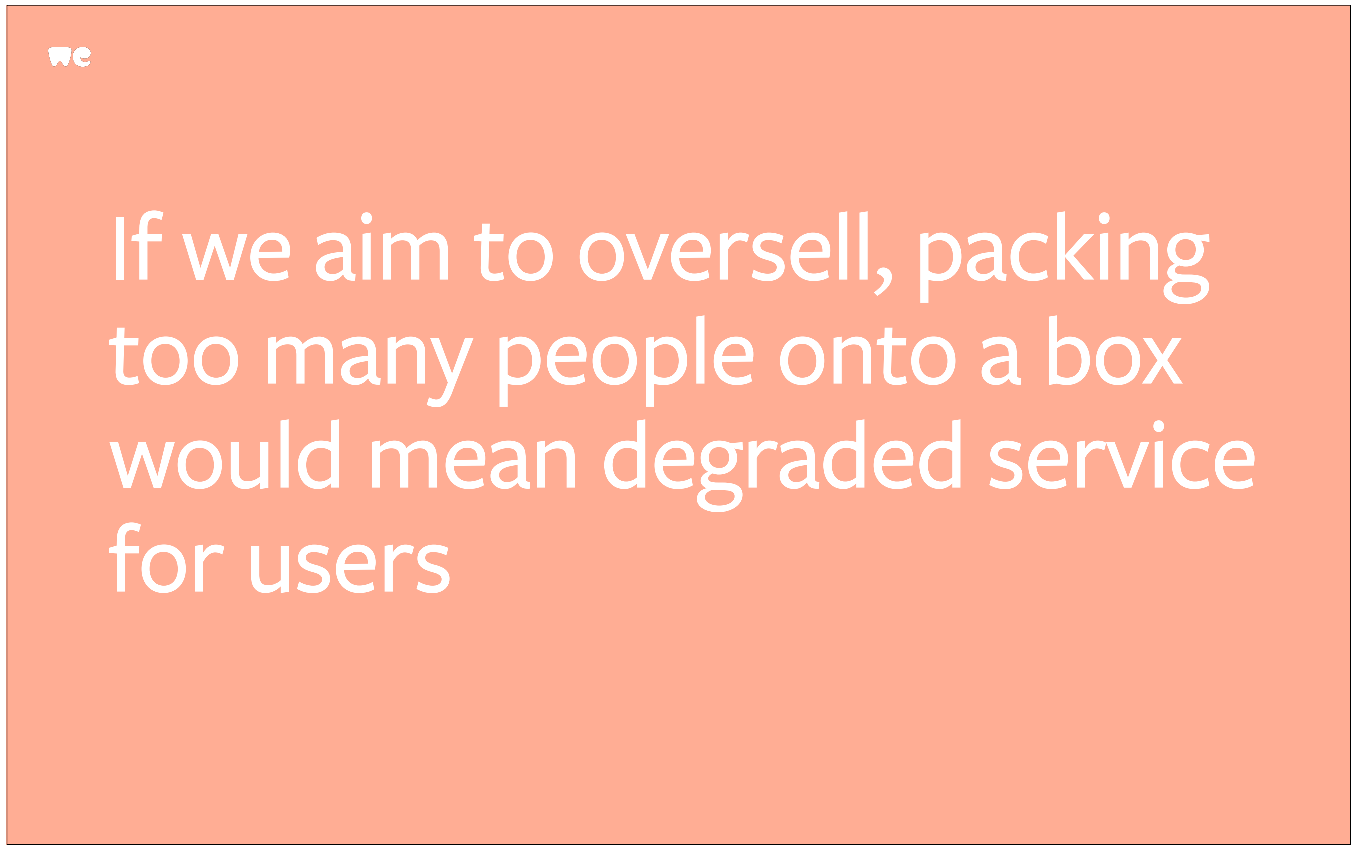
Instead we could have had…

  * Spaghetti stack and a horrible language and callbacks (node)
  * Spaghetti stack and a pedestrian language and coroutines (Go)
  * One process that eats up the entire machine and _maybe_ works better (Java)
  * Two new languages we would have to become familiar with intimately (Elixir) or one in case of bare Erlang
  * All of that to wave a flag and say "we pack 10 thousand miserable people into one machine to spare money"

In effect, what we did - since we had this solution with socket sends - and we could _measure_ the actual download speeds people get, we were able to do two things:

  * Determine how many people saturate the MRI/Pumas to the brim - turned out we can service about 300 people
  * Determine how much bandwidth the _current_ clients actually consume, and how much we have available.
  * Refuse or accept new connections based on these two criteria (our own load balancing).

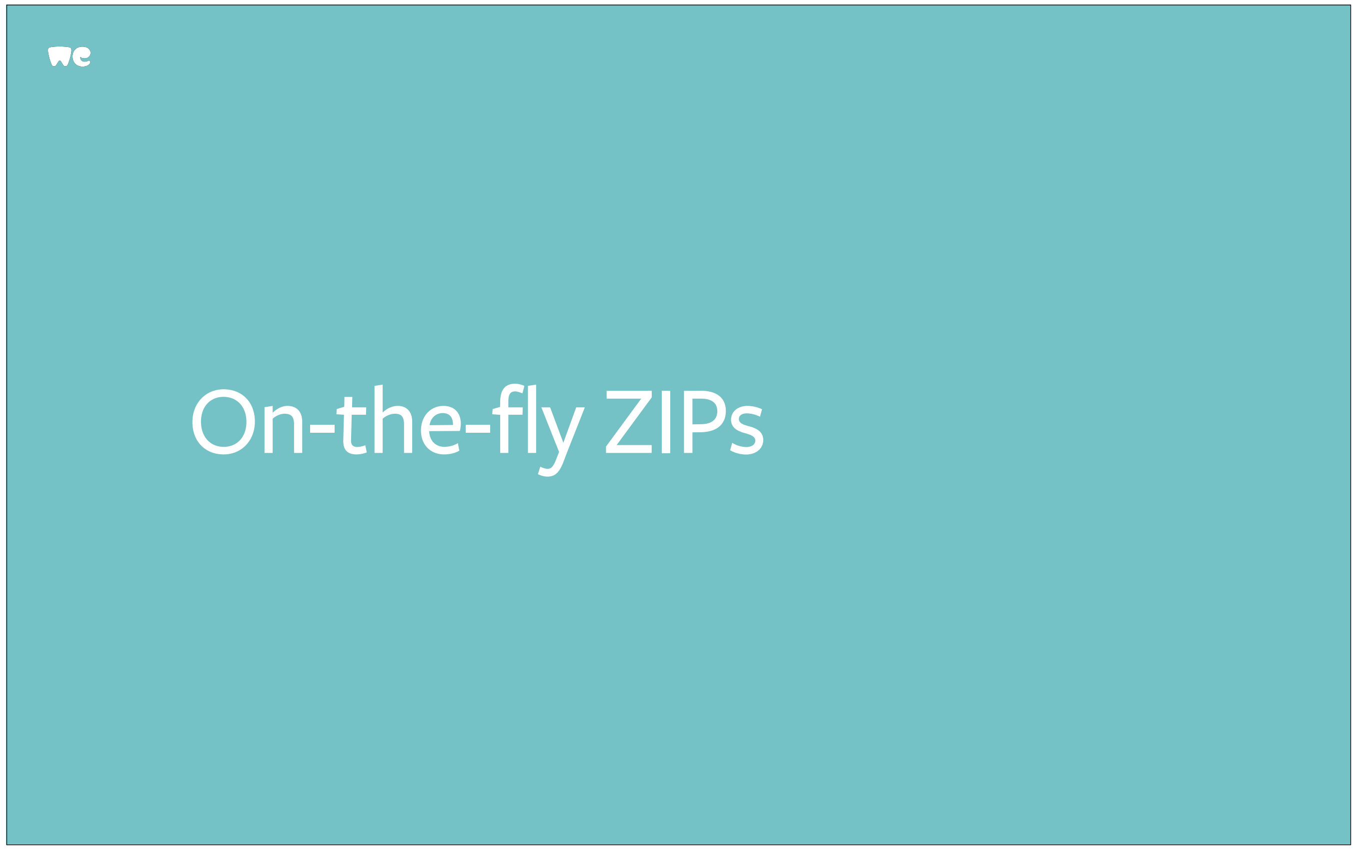
So: yes we use threads. Yes they use a lot of RAM as they are. Yes we could use a dramatically different approach to use something evented or to use a concurrent VM with a different language. But we do not win anything by packing too many people on one machine - it works _great_ for WhatsApp and Slack and whatnot, but it would not work well for us - and probably would not work that well for Netflix and Dropbox and the like.



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

In addition, we are not doing a chat application here. Most examples for highly concurrent web apps
hover around a situation where you have a tremendous number of clients with open connections, but those clients are _idle_.

When a chat message comes in, it gts multiplexed to the clients. However, this does
not cover the case when all of those clients want to _pull_ at maximum possible speed. When you do consider it, having 10 thousand people on one box becomes much less appealing. Why? Because Youth These Days Have Good Internet. A virtual machine on a cloud service has (in theory) one 1 gigabit connection you can saturate, outboind. This is about 100-200 megabytes per second. Most of our users can these days pull at around 500KB-1MB per second. This means, that we can only put about 100 people on one download server before each and every one of them starts seeing his download slow down because of this multitenancy. Yes, we would save ourselves money by doing this, but we would make our users unhappy - their download speed would degrade

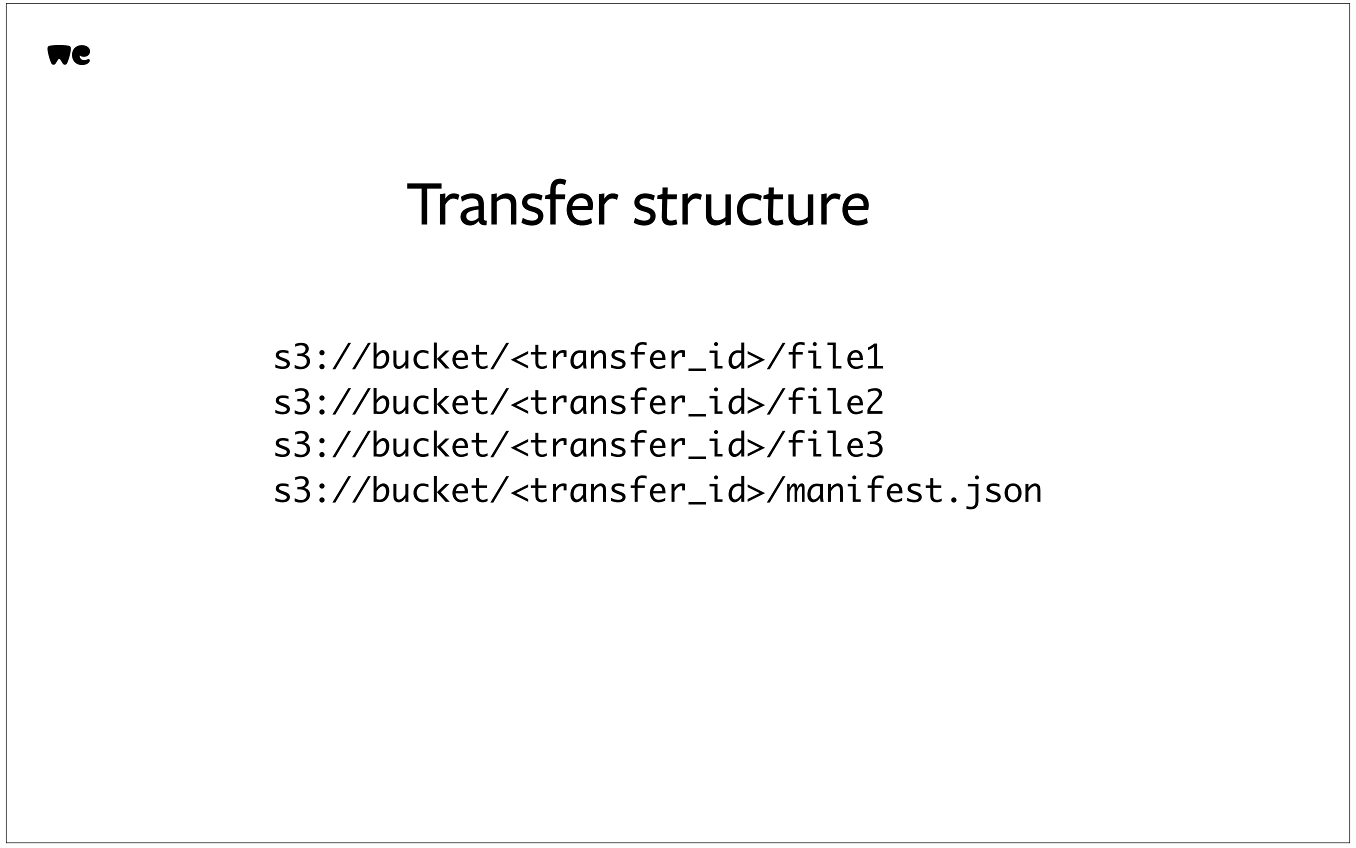


On-the-fly ZIPs

And now it gets intense. This was the goal of our rewrite - so that we could provide this.
At the outset we only supported downloading one file. That was totally doable, and we got it
to work OK. The next challenge came when we decided to get rid of the "one transfer = one file"
concept. Now we would do this for every transfer:

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

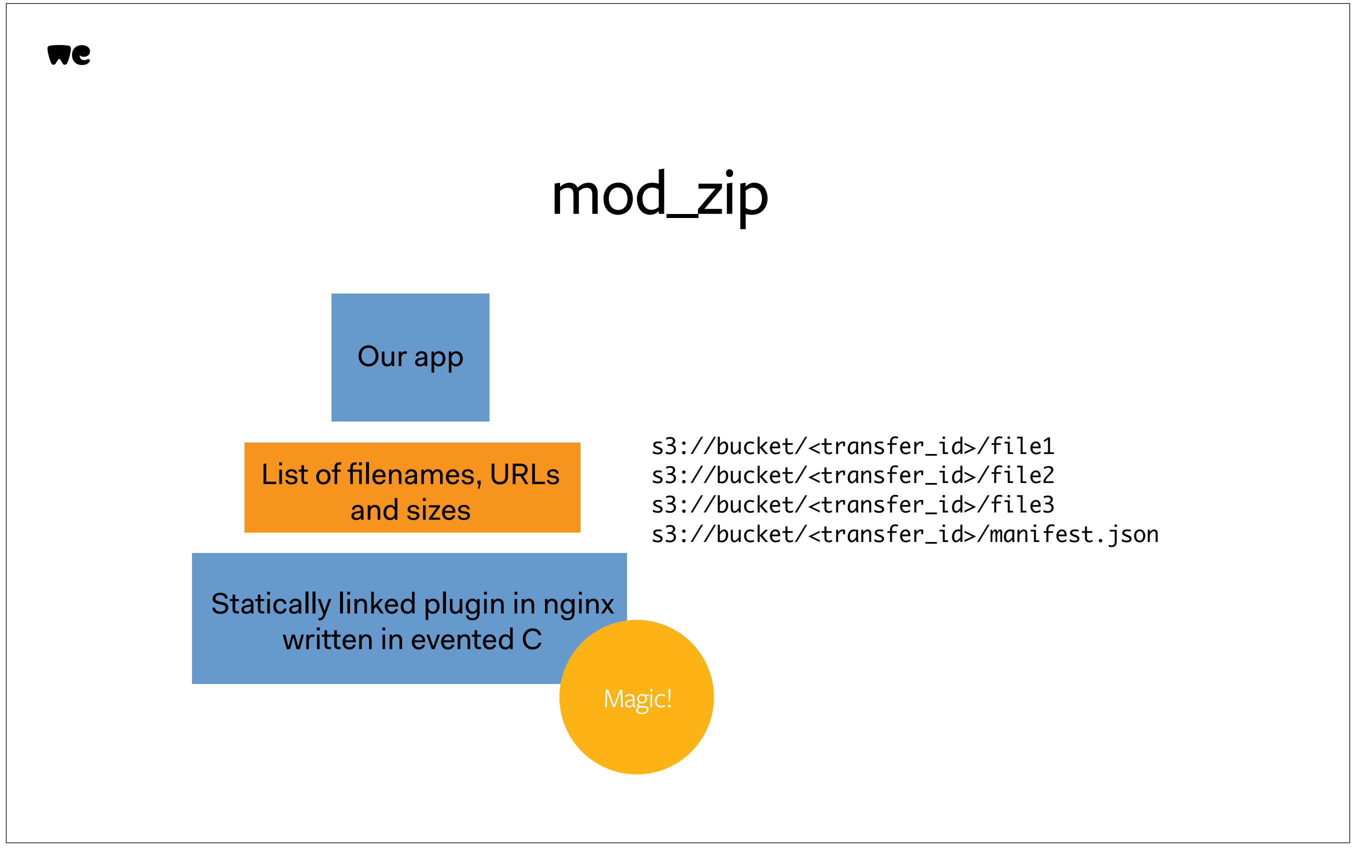
and we would generate a ZIP file with these files _on_demand_ Pre-processing these ZIPs
was literally killing us on the backend, and additionally we could not implement a feature that
would allow you to download one specific file in a transfer - something we absolutely had to have.



Transfer structure

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

This is how a “transfer” looks internally BTW



mod_zip

Our app
List of filenames, URLs and sizes
Statically linked plugin in nginx written in evented C
Magic!

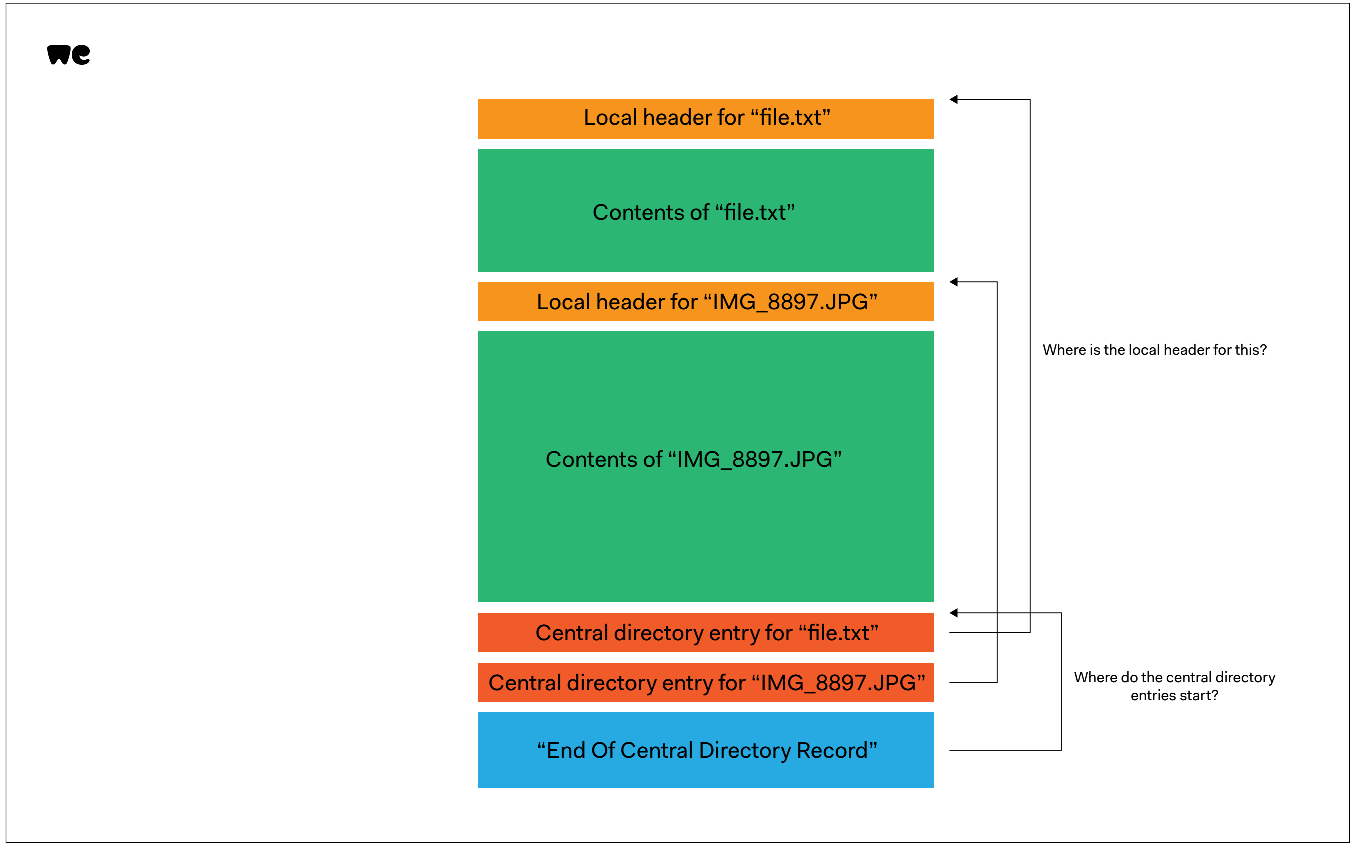
```
s3://bucket/<transfer_id>/file1
s3://bucket/<transfer_id>/file2
s3://bucket/<transfer_id>/file3
s3://bucket/<transfer_id>/manifest.json
```

Today there is exactly _one_ option on the market for doing this “out of the box” – it's the `mod_zip` extension for Nginx from Evan Miller. It works by proxy – slightly similar to how the Gitlab workhorse works. You return a special text file in your response instead of actual data, and Nginx will do all the fetching and proxying and ZIP computation and whatnot.

However, it has a few special qualities:

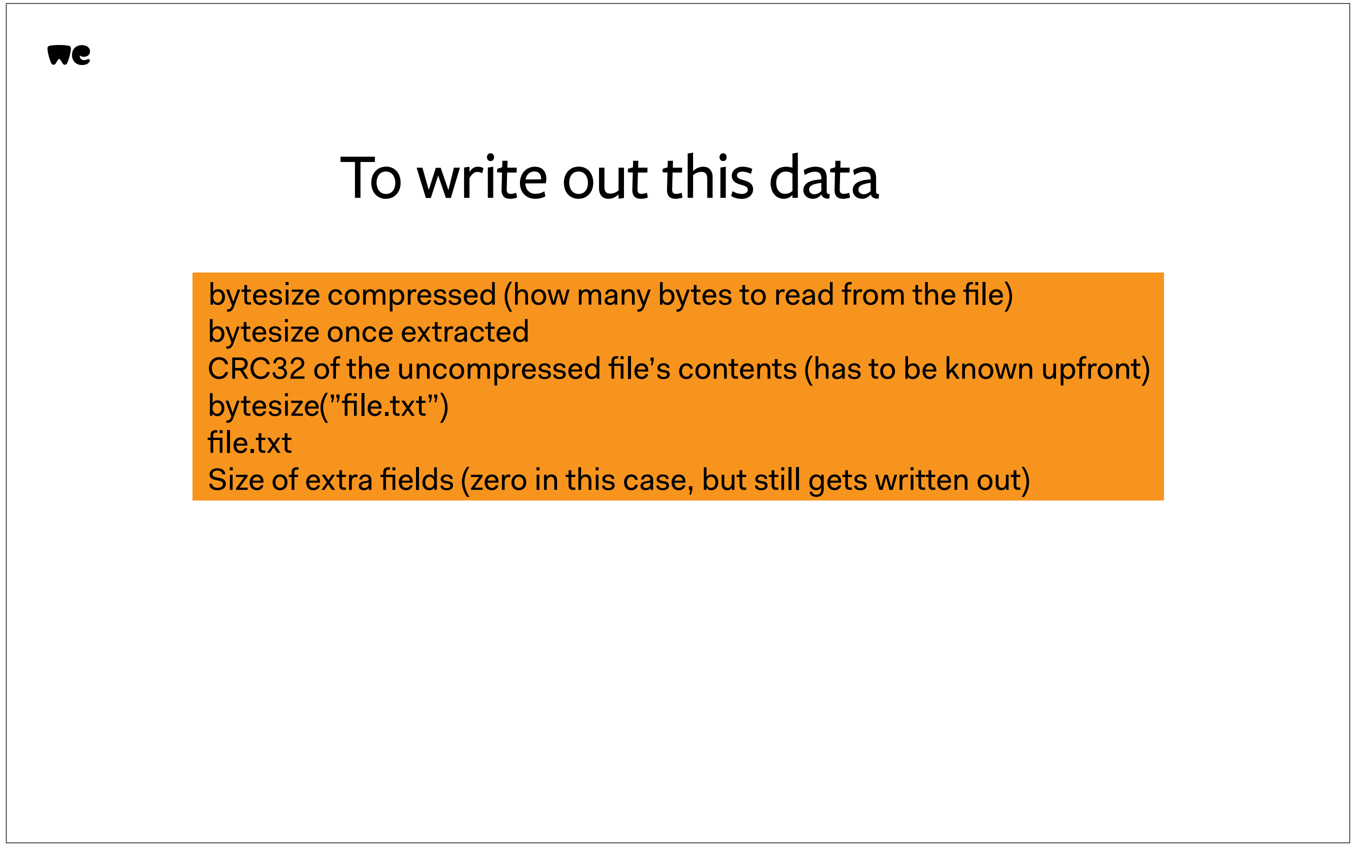
* It does not have callbacks to generate our download emails
* It is evented C heavily integrated with nginx' evented C
* It uses a homegrown format for the file list. Homegrown format + user-supplied UTF-8 filenames?..
* It probably does retries, for those S3 500s. Somewhere in nginx event loop. In C. Probably. It also has timeout control. Probably.

Basically the defining quality if we wanted to fix it or change it in any way was this: either we know the nginx dialect of evented C, _OR_ we have to stack another nginx plugin on top (more evented C or complicated nginx config), _OR_ we just write the darn thing ourselves, in not-evented-something that we can actually read.


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?

So we needed a library to write ZIPs. And of course we have RubyZip. But before we get to the libraries, a little refresher on the ZIP file structure:

```
[Local file header for "file1"] {size, crc32, filename_size, filename_bytes}
[File contents for "file1"] {bytes}
[Local file header for "file2"] {size, crc32, filename_size, filename_bytes}
[File contents for "file2"] {bytes}
...
[Central directory header for "file1"] {size, crc32, filename_size, filename_bytes}
[Central directory header for "file2"] {size, crc32, filename_size, filename_bytes, offset_of__local_file_header}
...
[End of central directory record] {size of the ZIP so far, how many files stored etc.}
```



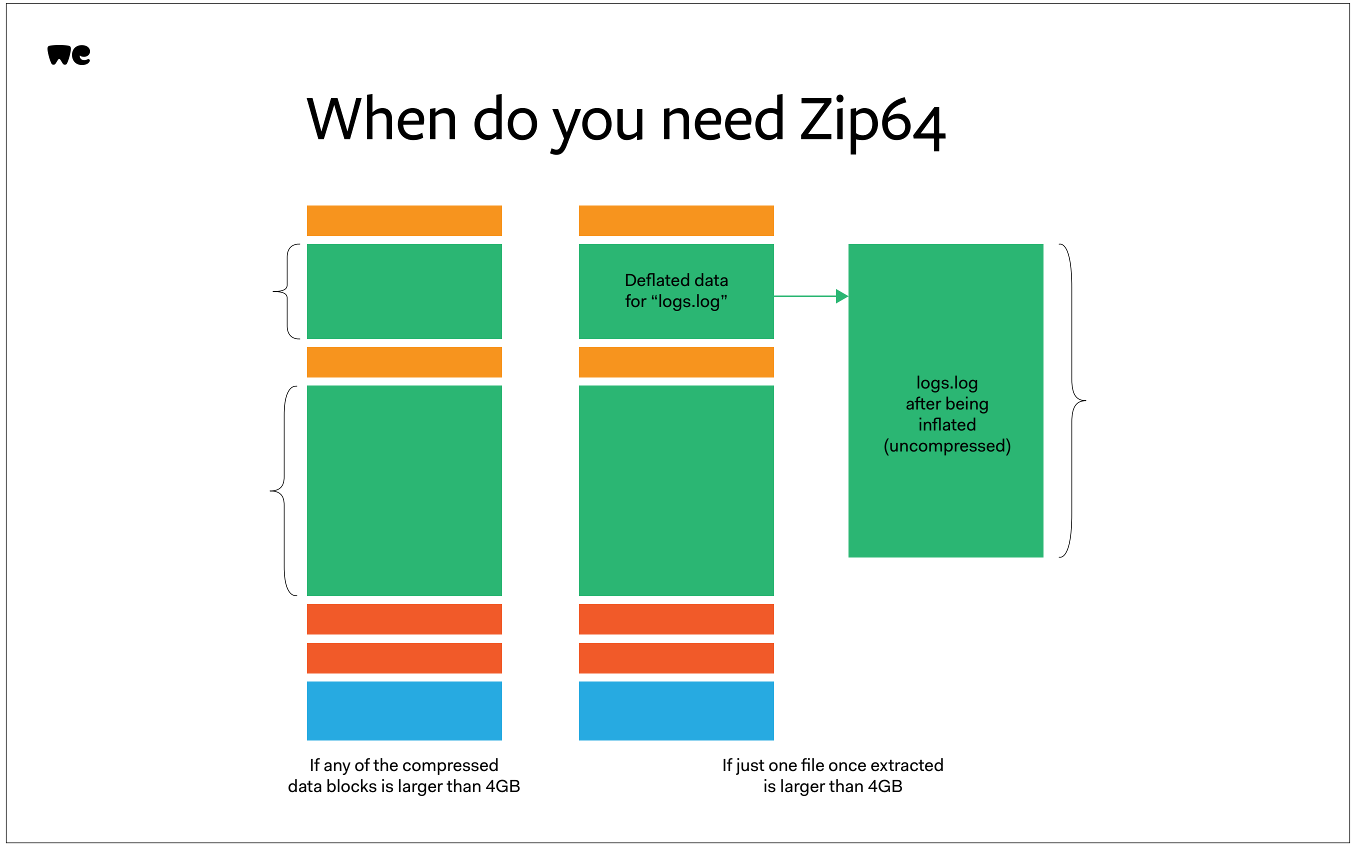
To write out this data

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)

The sizes recorded have fixed offsets, and the offsets are limited to a 4 byte unsigned integer. This means that
if you store files in the ZIP which are larger than 4GB in size, OR your file headers and their contents _end_ at an offset larger than 4GBs, you have to do some extra things to properly record these sizes:

```
[Local file header for "file1"] {fake_size_, crc32, filename_size, filename_bytes, Zip64 extra data}
```

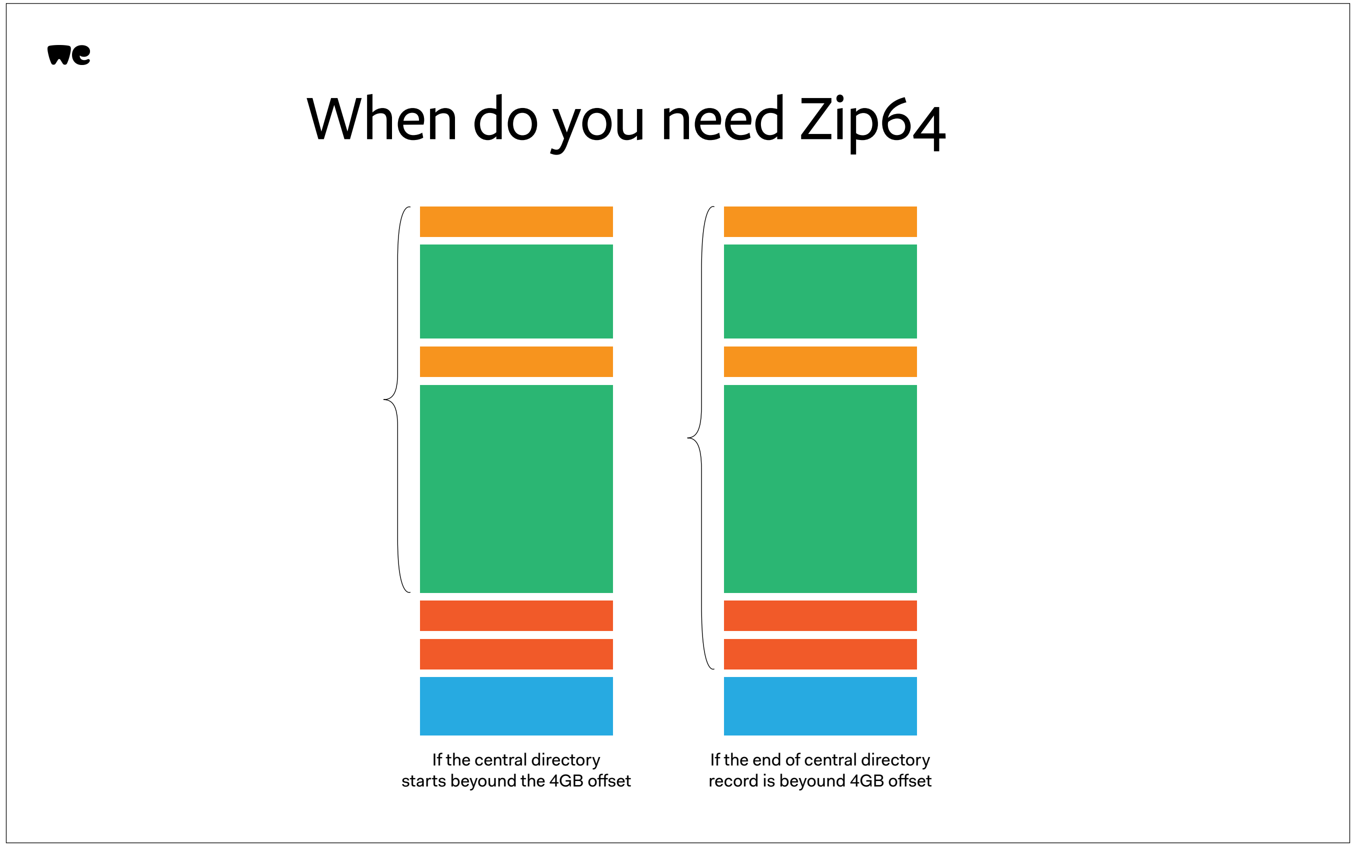
So what do we have to deal with Zip files in Ruby? Of course, we have Rubyzip!But we also have large files, so we will need Zip64.



When do you need Zip64

Deflated data for “logs.log”
logs.log after being inflated (uncompressed)
If any of the compressed data blocks is larger than 4GB
If just one file once extracted is larger than 4GB

Which you need if you have this

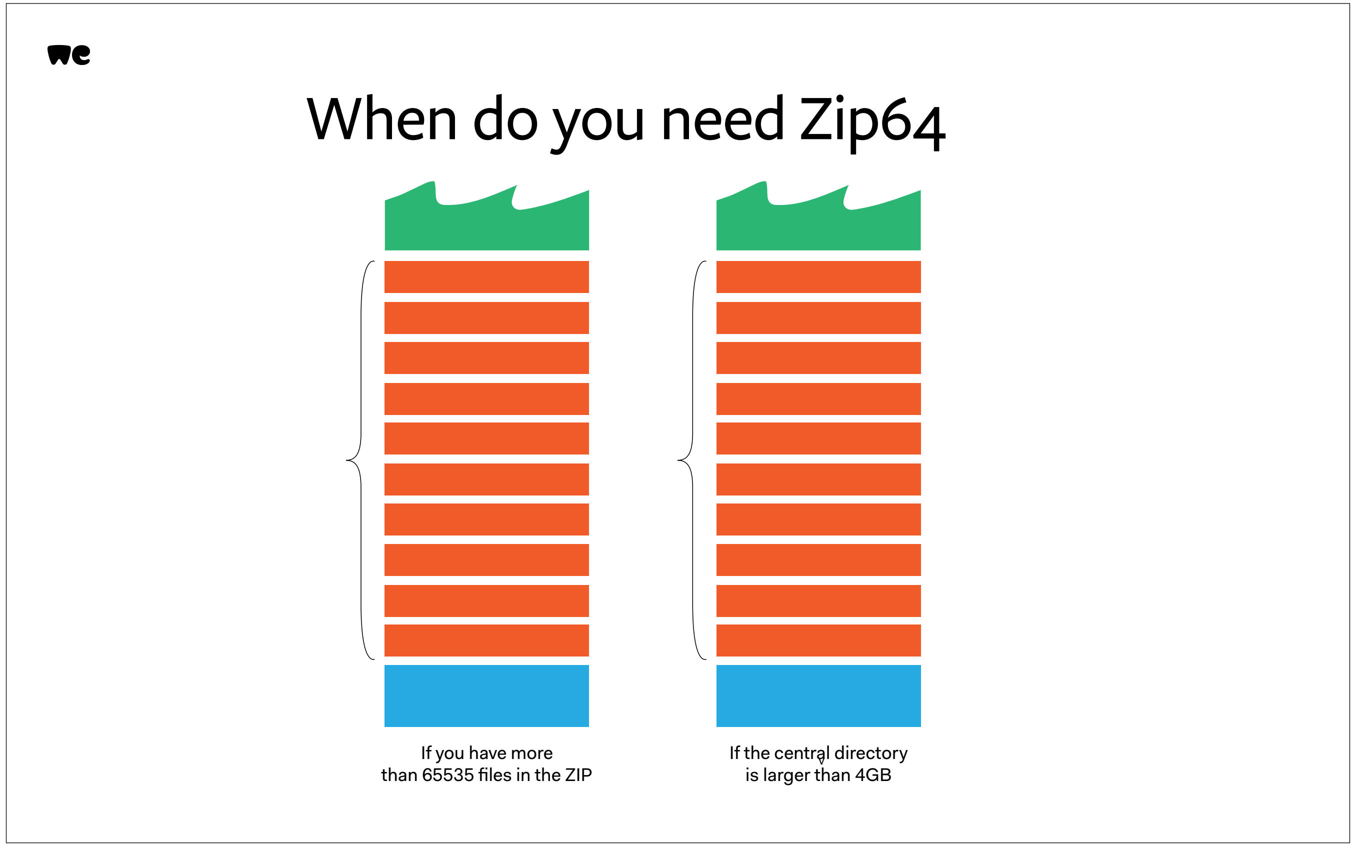


When do you need Zip64

If the central directory starts beyound the 4GB offset

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

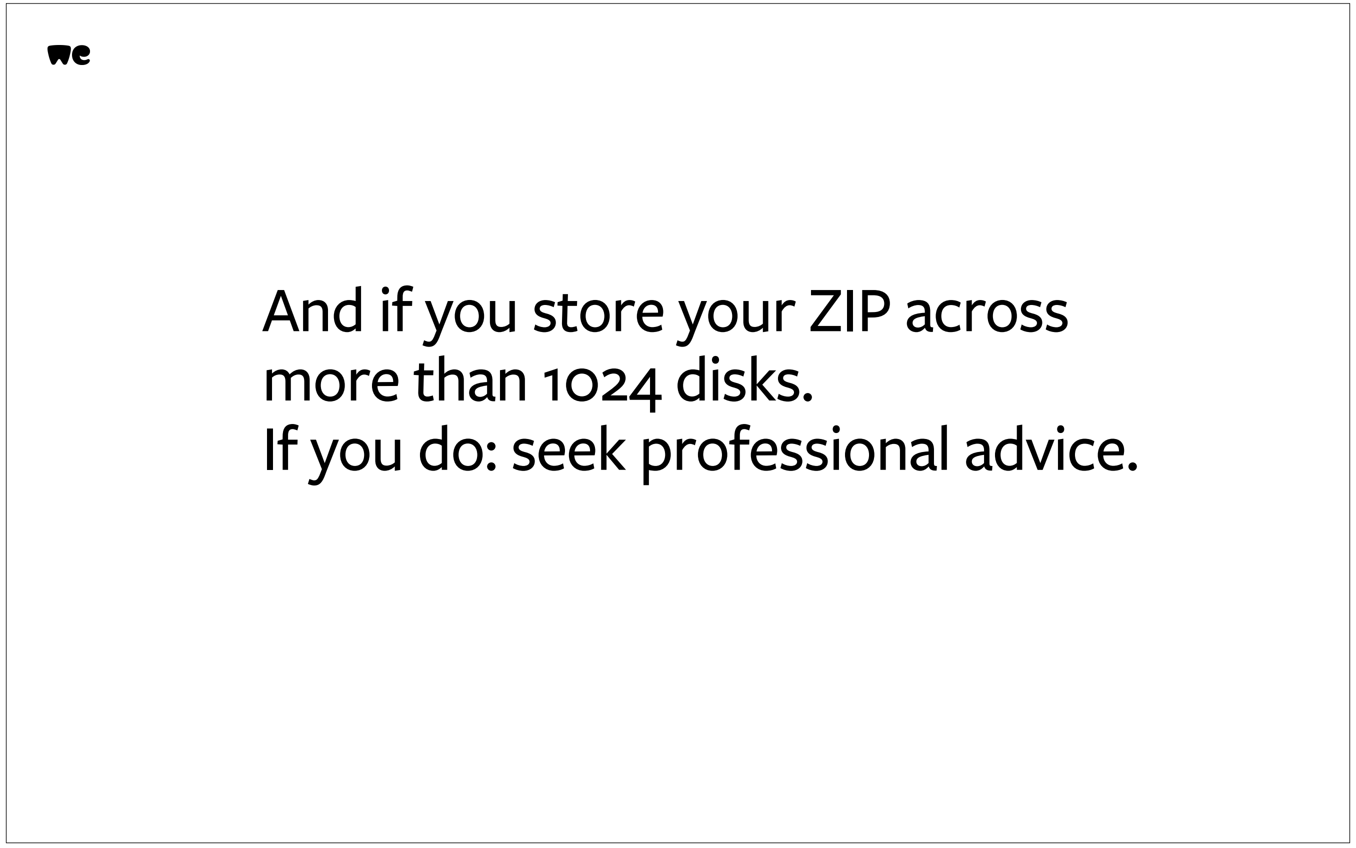
or this



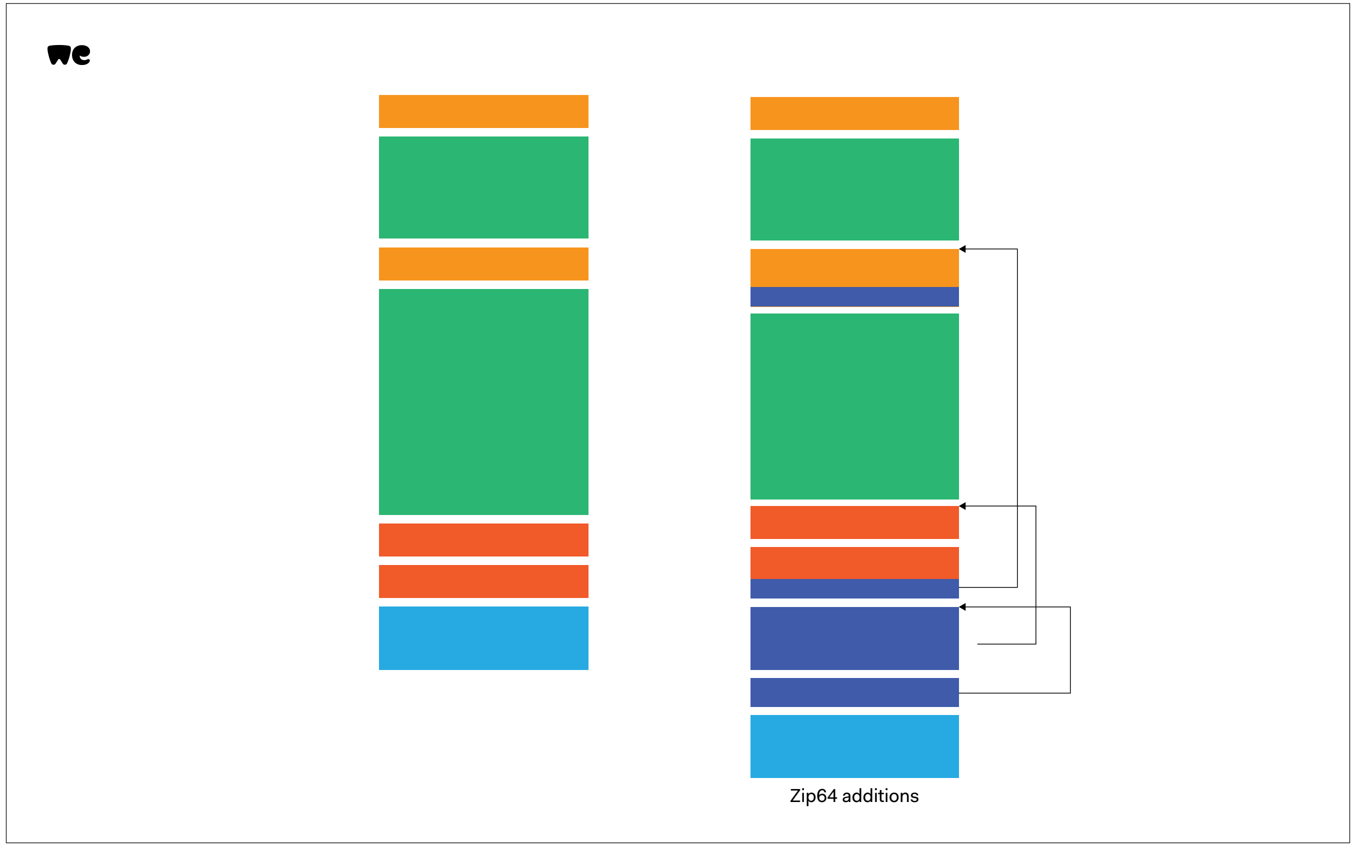
When do you need Zip64

If you have more than 65535 files in the ZIP
If the central directory is larger than 4GB

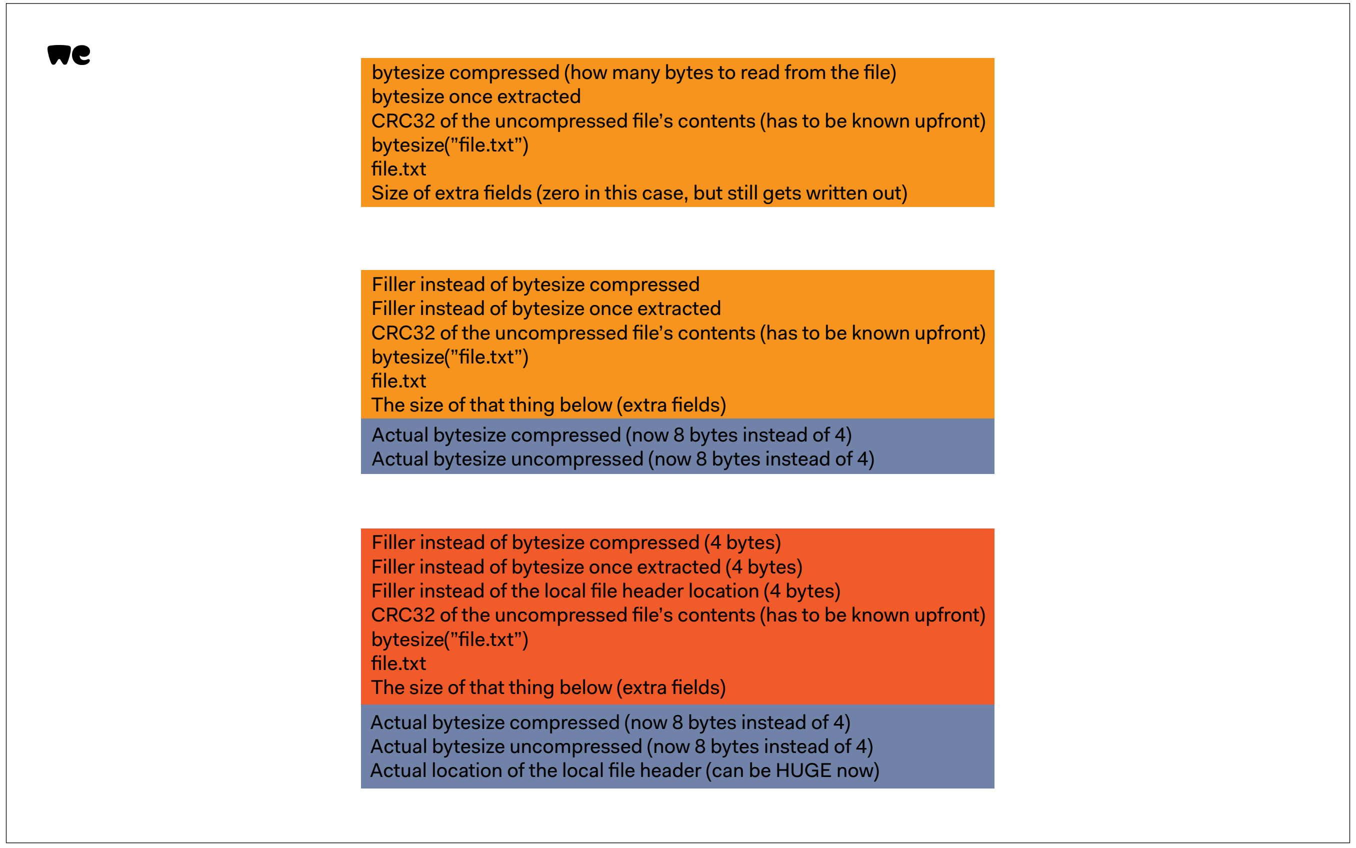
or this


And if you store your ZIP across
more than 1024 disks.
If you do: seek professional advice.

Or you work for The Internet Archive



So when you do need Zip64 (and we were going to need it often) you just write some extra data into the file that can fit the offsets that go beyound what the original spec could permit

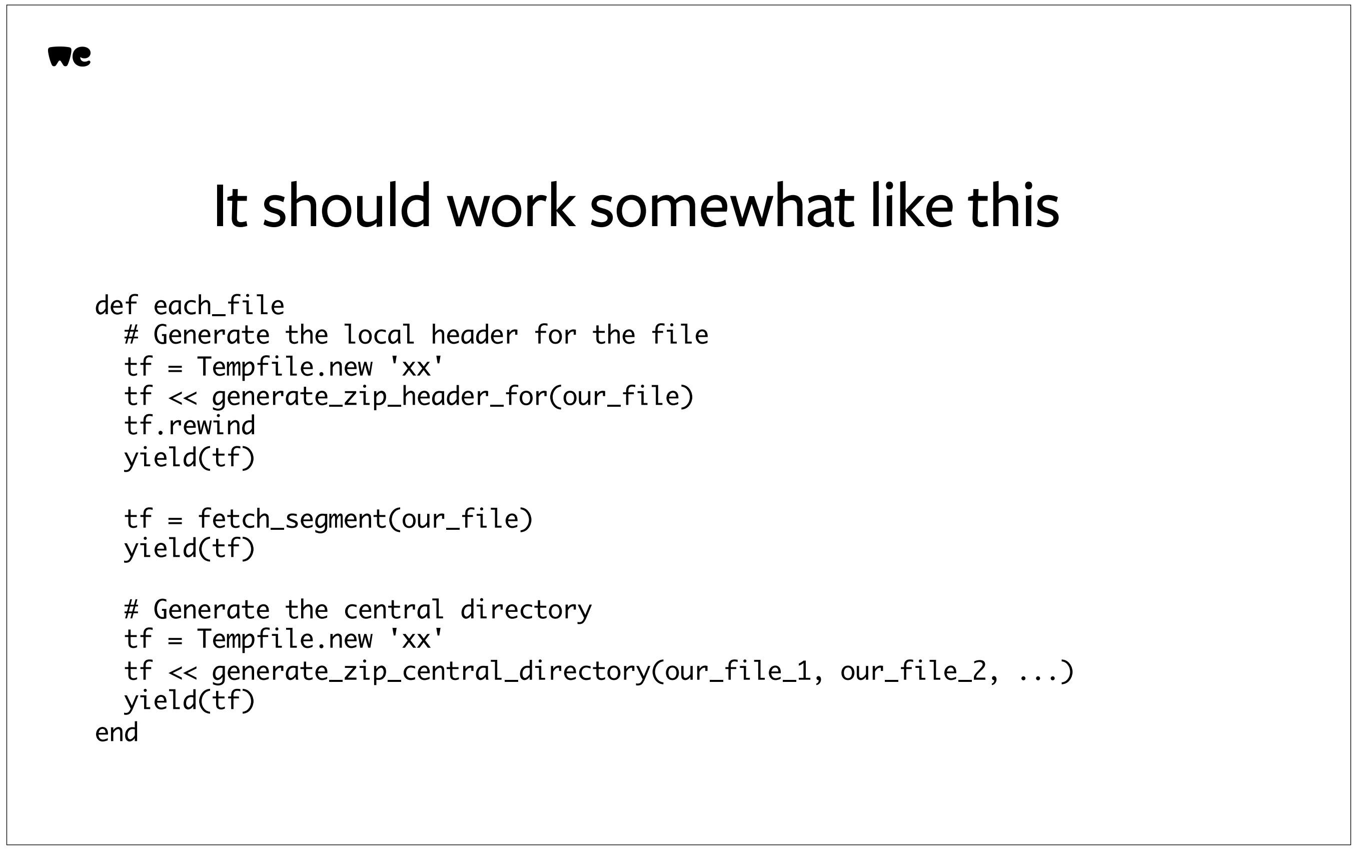


bytesize compressed (how many bytes to read from the file)
bytesize once extracted
CRC32 of the uncompressed file's contents (has to be known upfront)
bytesize("file.txt")
file.txt
Size of extra fields (zero in this case, but still gets written out)

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)

Those headers are fixed size.

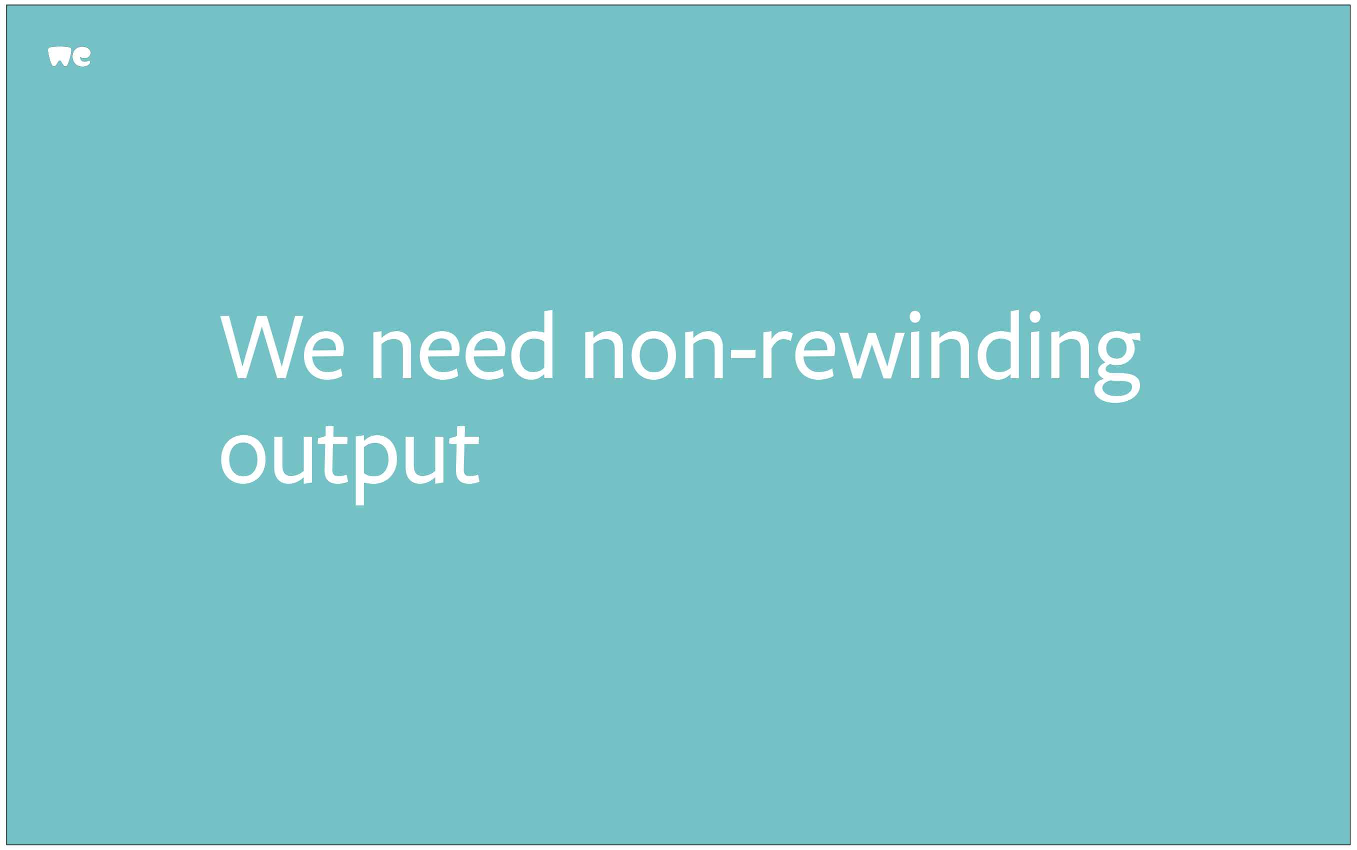


we

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
```

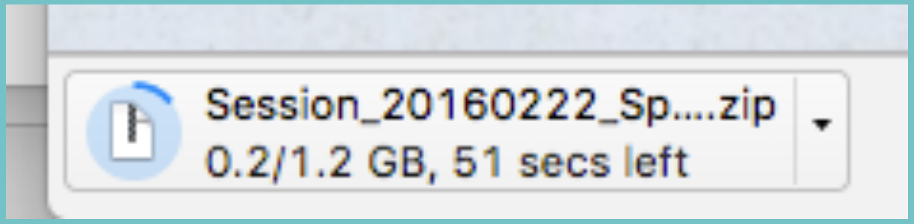
And here is how the API should look



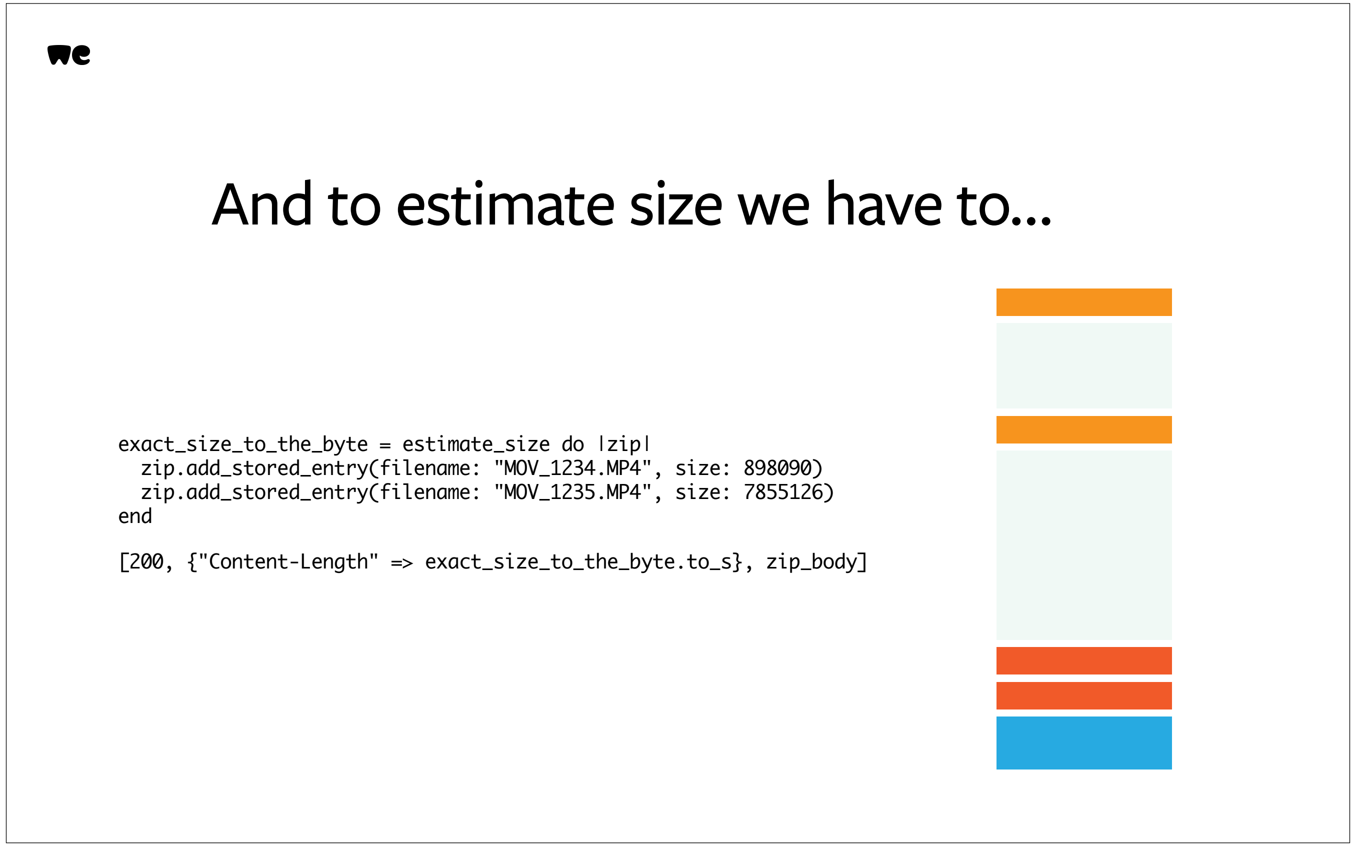
We need non-rewinding output

So what do we have to deal with Zip files in Ruby? Of course, we have Rubyzip! However, the ZIP spec has this interesting quality that you have to know how large your output compressed file _within_ the ZIP is going to be _before_ you start writing it. RubyZip – and other Ruby zip libraries – bypass it by rewinding.

And size estimation!


Session_20160222_Sp....zip
0.2/1.2 GB, 51 secs left

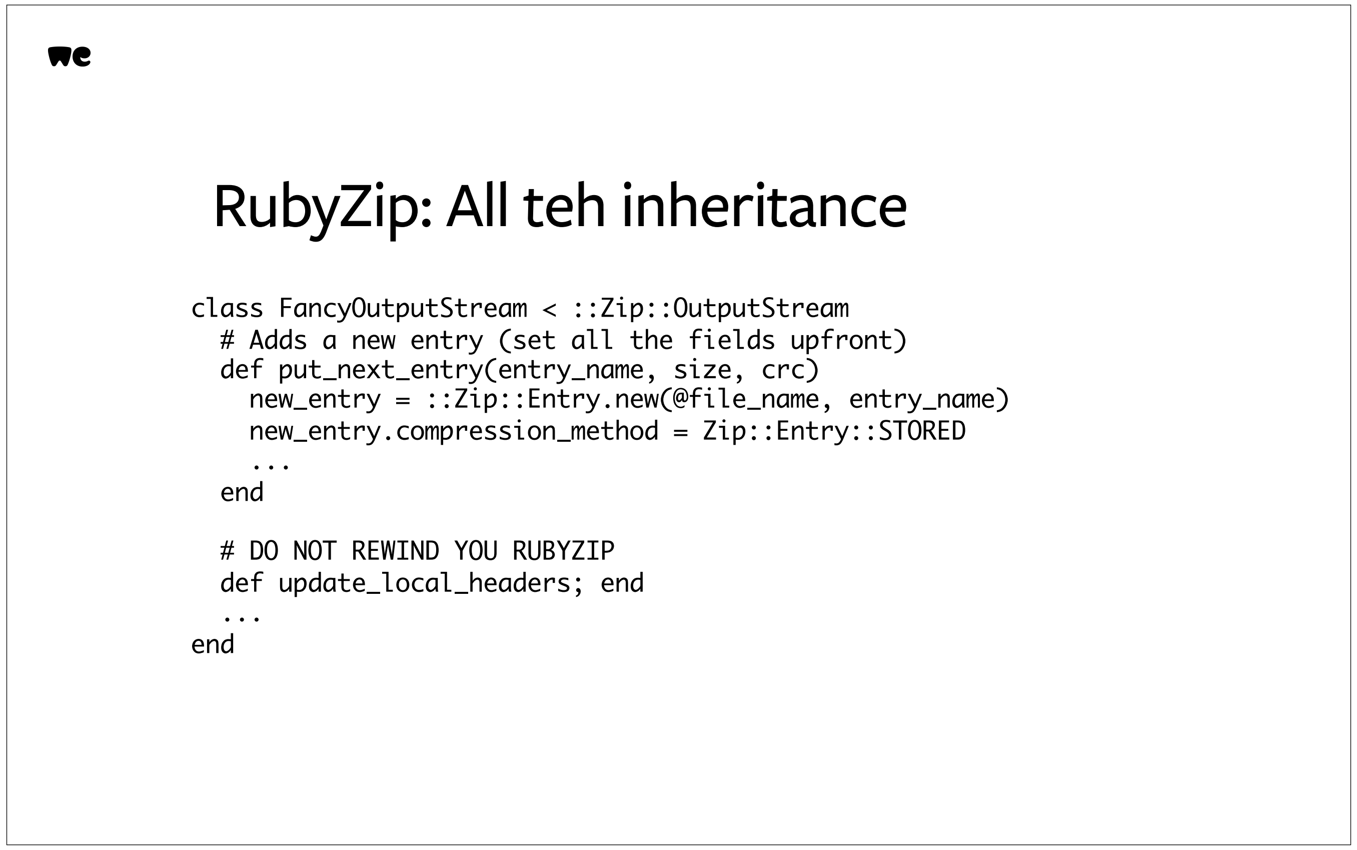
We also need our downloads to show this progress bar. And you can only show it if you know how large exactly your ZIP is going to be in the end



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]
```

And to do that you need to do a "fake ZIP" creation where you do _not_ fetch the entire transfer of 4GB from the server, and yet make the ZIP library go through the same motions to compute the size upfront.

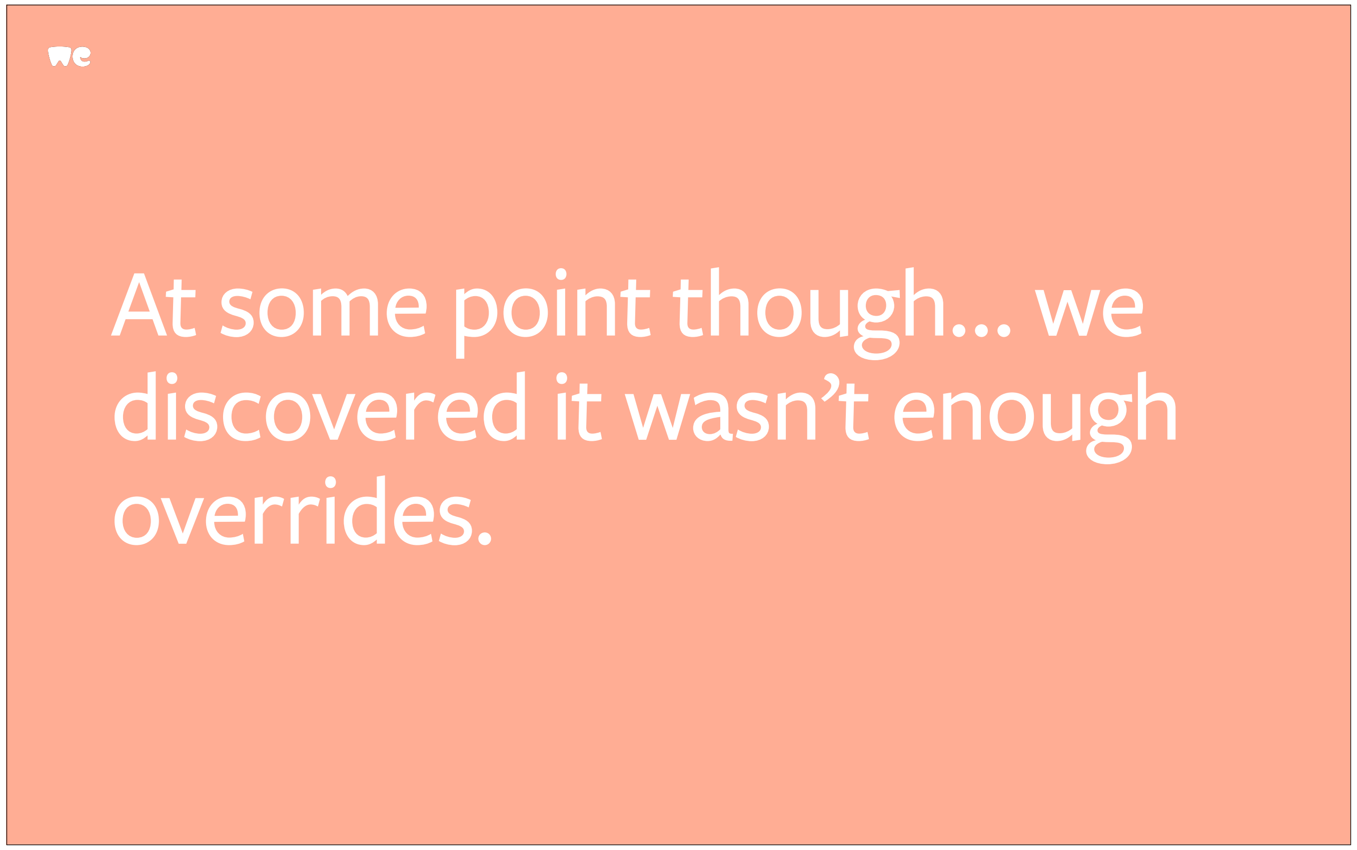


RubyZip: All teh inheritance

```
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
```

And so we went in and manhandled RubyZip not to rewind when it writes.

We also implemented a workaroundy-thing that used this same class to precompute the size of the ZIP without having to pull in the actual files.


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

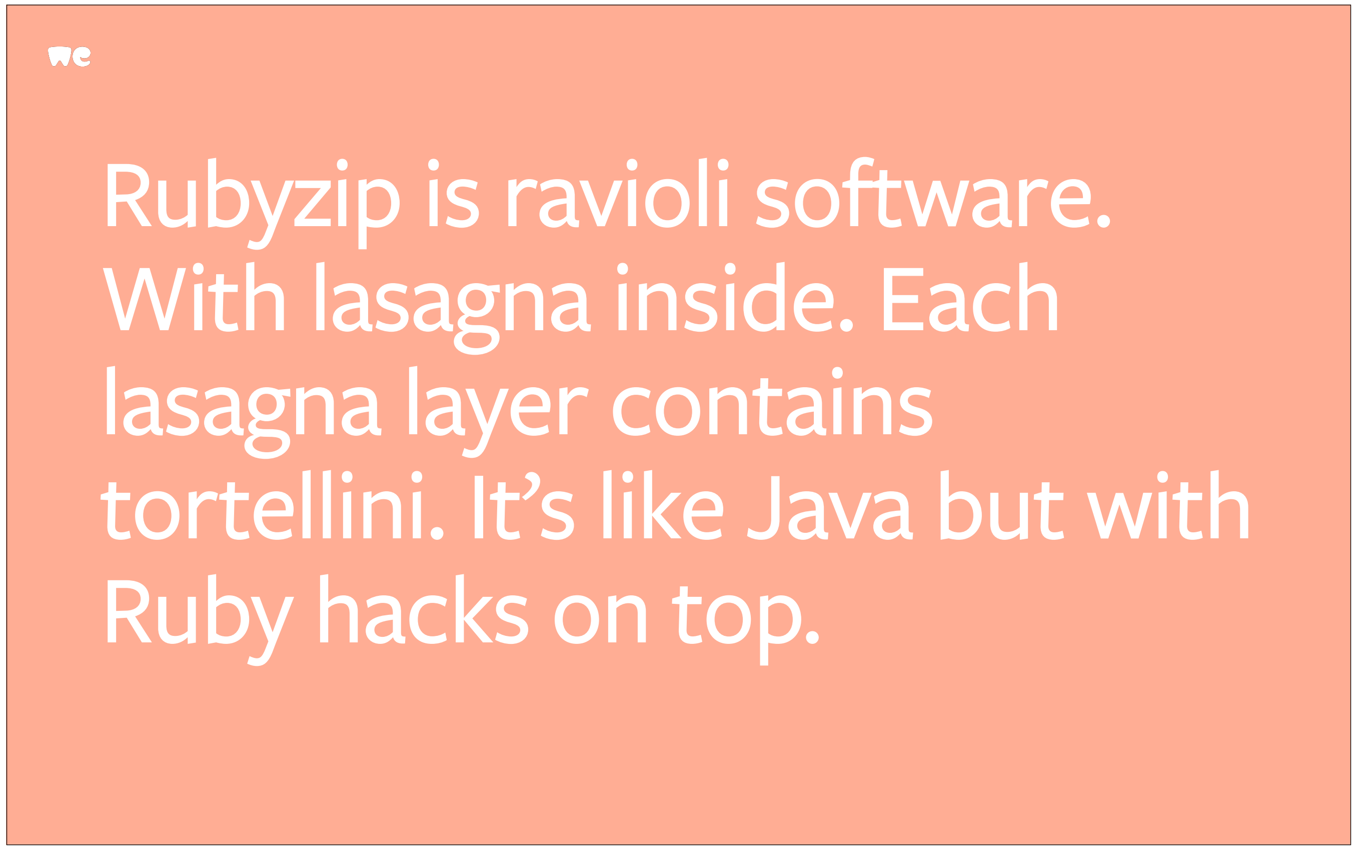
And that worked, until one day...

> Your ZIP files do not open for me in Windows Explorer

And so we went digging. And we found the following.

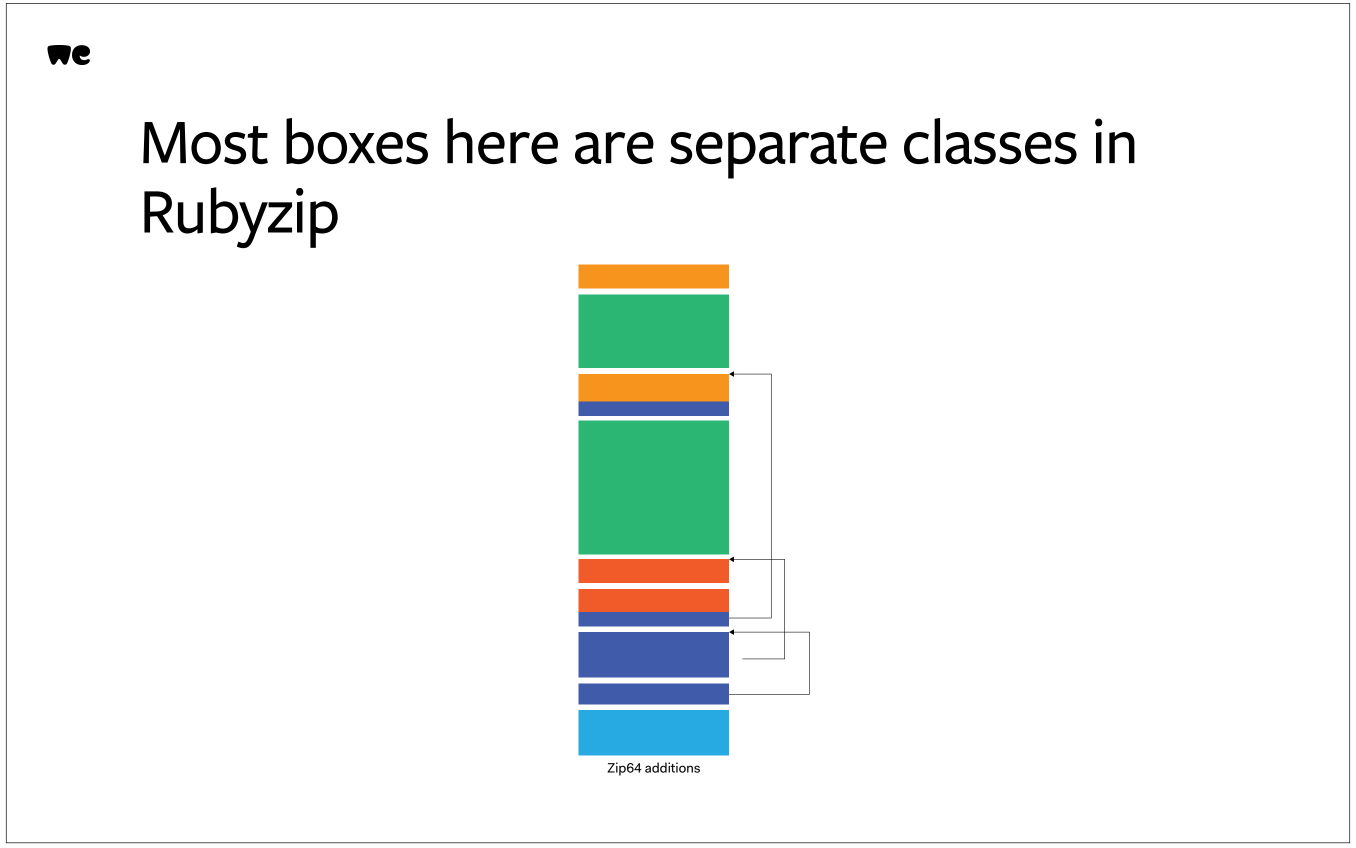
When we write out those entries, RubyZip allocates
padding – an extra field, for the time when it might have to write out the Zip64 extra fields. It has to, because it cannot "come in" afterwards and "insert" this Zip64 extra field into the file after the fact, it can only overwrite the padding.

And it turned out, if Windows Explorer sees an extra field that it cannotdecode, and this is the first (and in our case only) extra field for the entry, it will refuse to unpack the
entry entirely. In fact it will refuse to even show you the list of files in the archive. We had to override MORE of Rubyzip to make it do the thing.

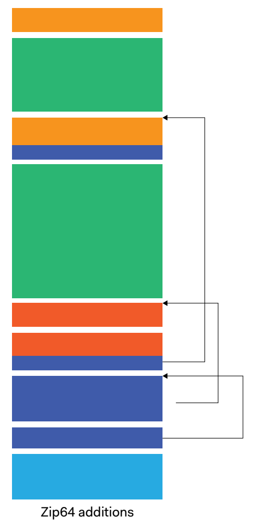


Rubyzip is ravioli software. With lasagna inside. Each lasagna layer contains tortellini. It’s like Java but with Ruby hacks on top.

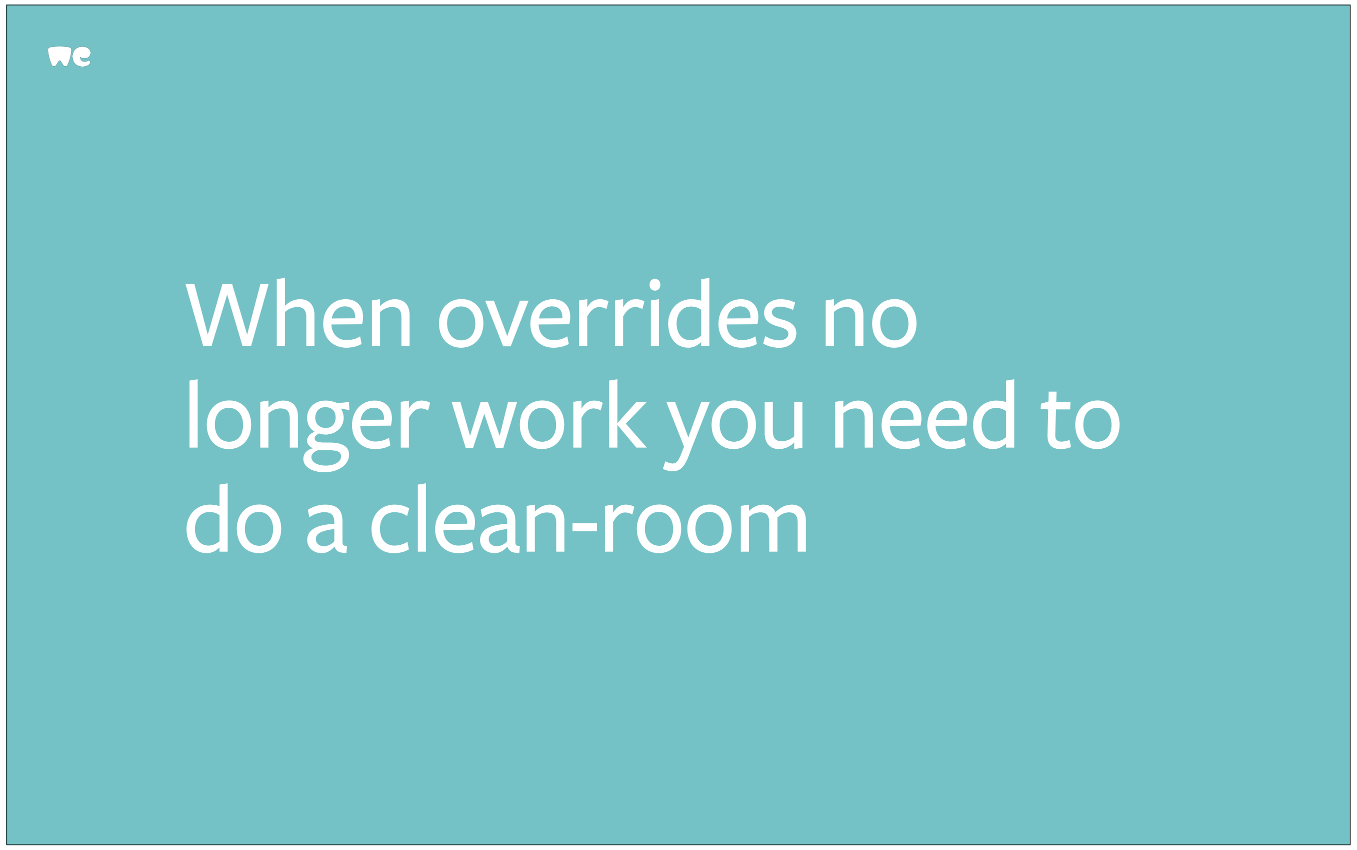
And so we went looking again how we can avoid writing out this extra padding. And then the horror of our situation became apparent. Rubyzip is a Java-style ravioli library. And inside the ravioli there is the lasagna, and within the lasagna you have little tortellinis too. To figure out where the writes happen for the entries, you need to dive down about 4 levels of abstraction, if not more. Removing the rewinding feature is _hard._ As a bonus, the writing of the (unnecessary, as it turned out!) Zip64 extra padding fields was a _feature_. It was a pull request someone submitted and had accepted, with code reviews and stuff! So now we would have to go back and say “this PR was nonsense, and we want it reverted, because it does not permit us to do our thing AND the extra padding you write out is completely unnecessary”.



Most boxes here are separate classes in Rubyzip

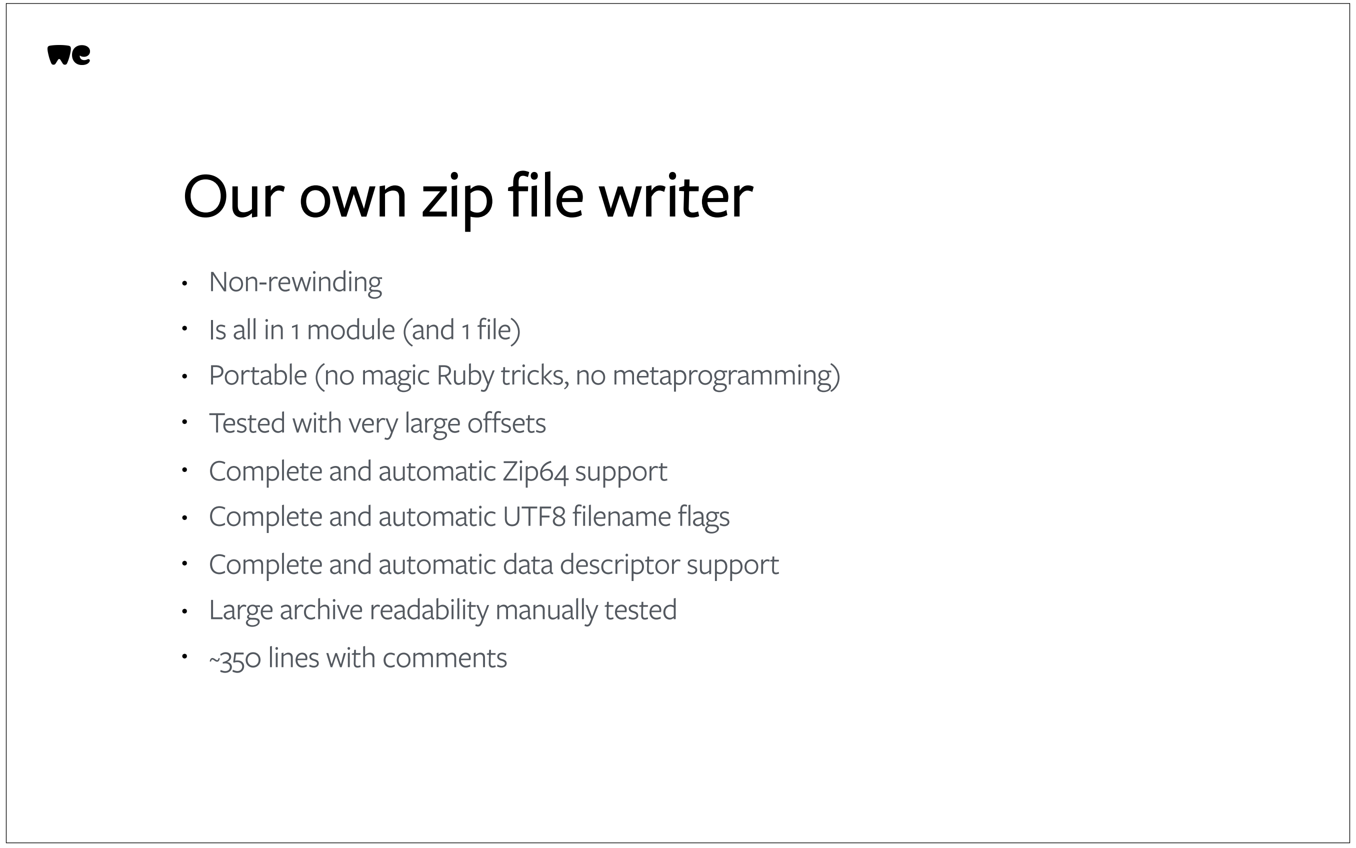


So we played and played and played with it, and in the end we came to the conclusion that instead of having this Java-library-in-disguise-that-pretends-to-be-Ruby we just can write our own using the ZIP file format specification. We _already_ had to know how the format was put together to sort of work around all of the issues in RubyZip itself. We already had to override pretty much all of the use cases for RubyZip in our code. The only part we used RubyZip for is actually encoding this binary stuff and writing it out. And if we could replace that single part of RubyZip with something we could change and play with we could solve this problem easily.



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

That.

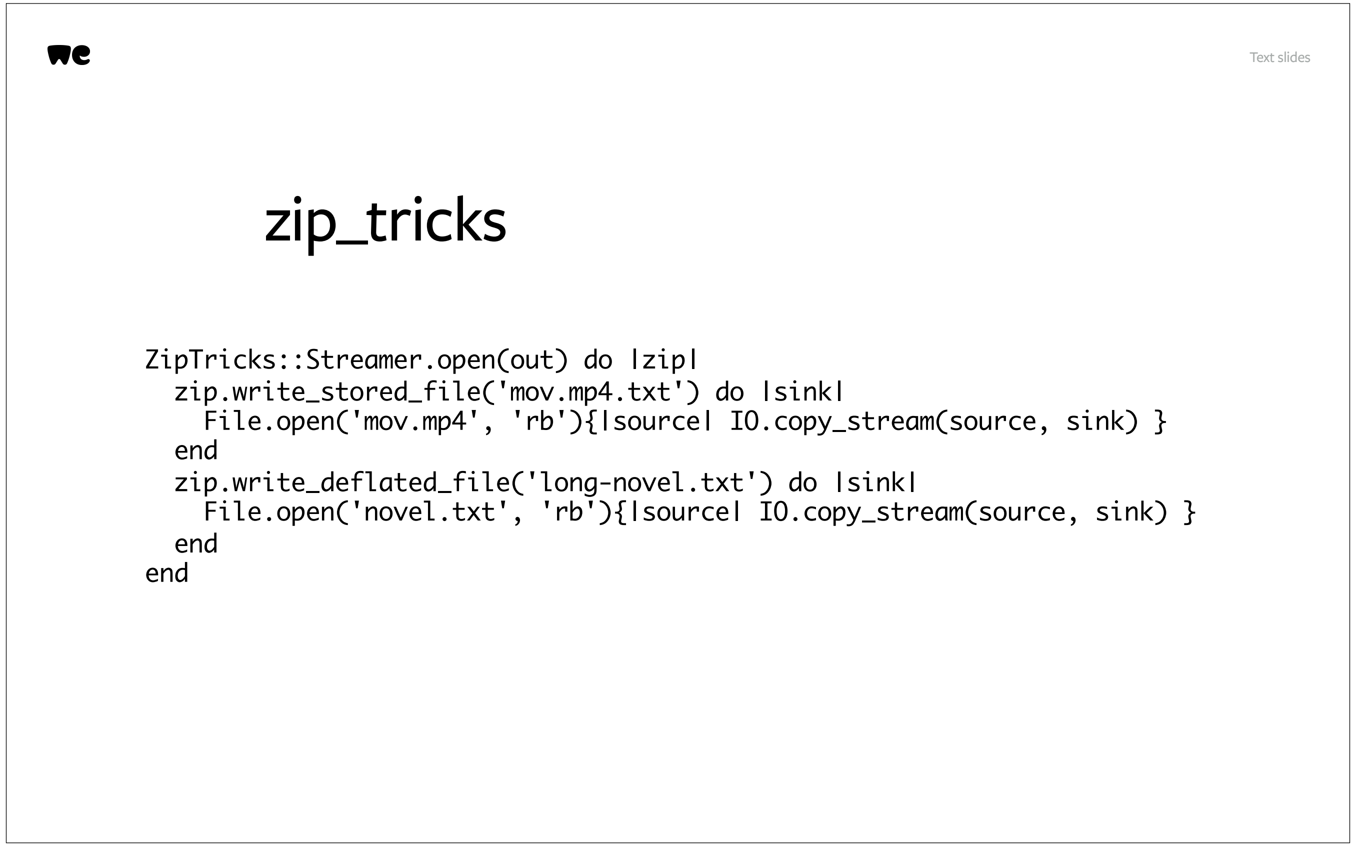


Our own zip file writer

- 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

So a from-scratch writer it was going to be. We could not make RubyZip work for us without rewriting a good half of it (and neutering the other). So this is how hard it was to write our own.

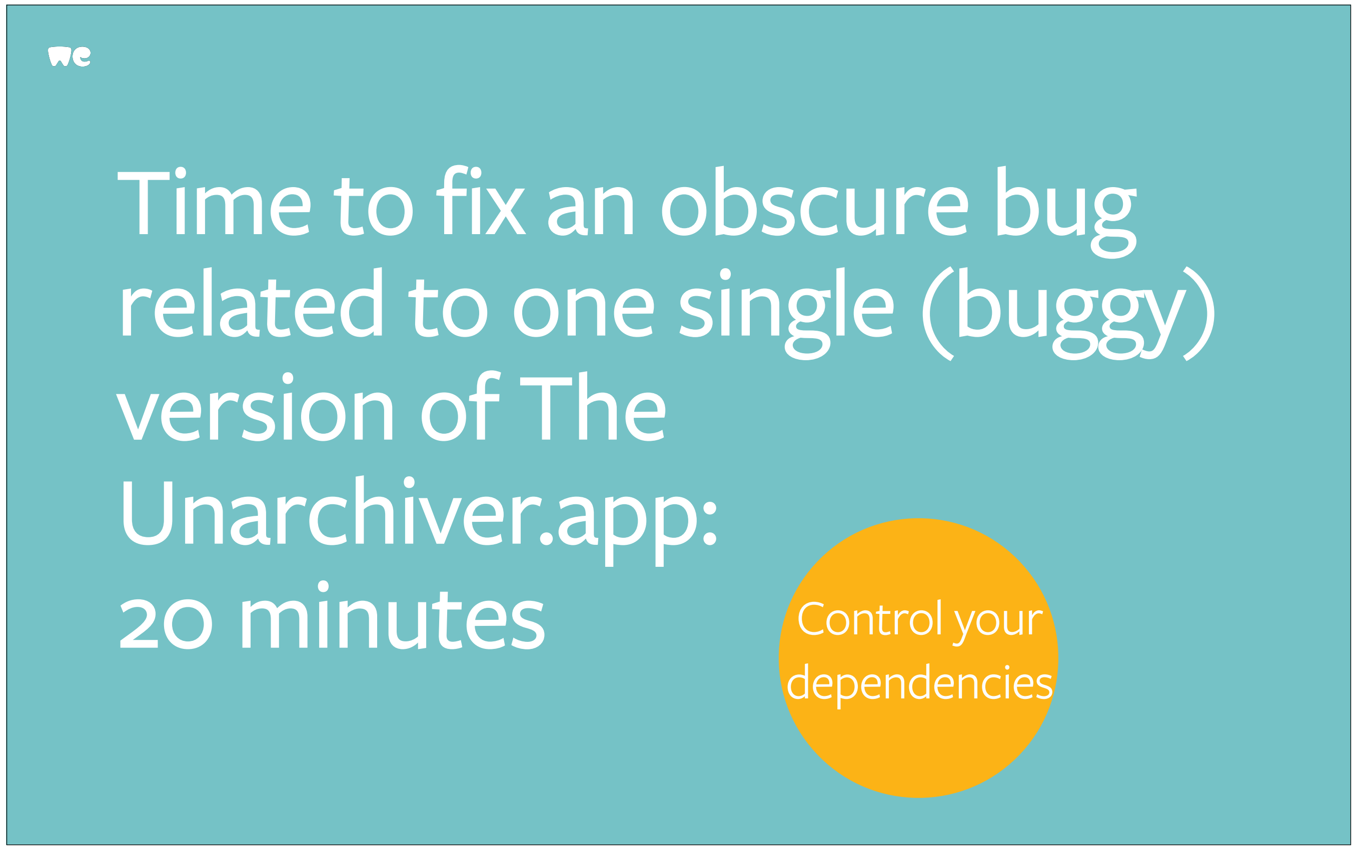
2 days of work, many unit tests, and some old-fashioned nosing around the ZIP format spec. But now we have a robust implementation which consists of exactly one module.



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
```

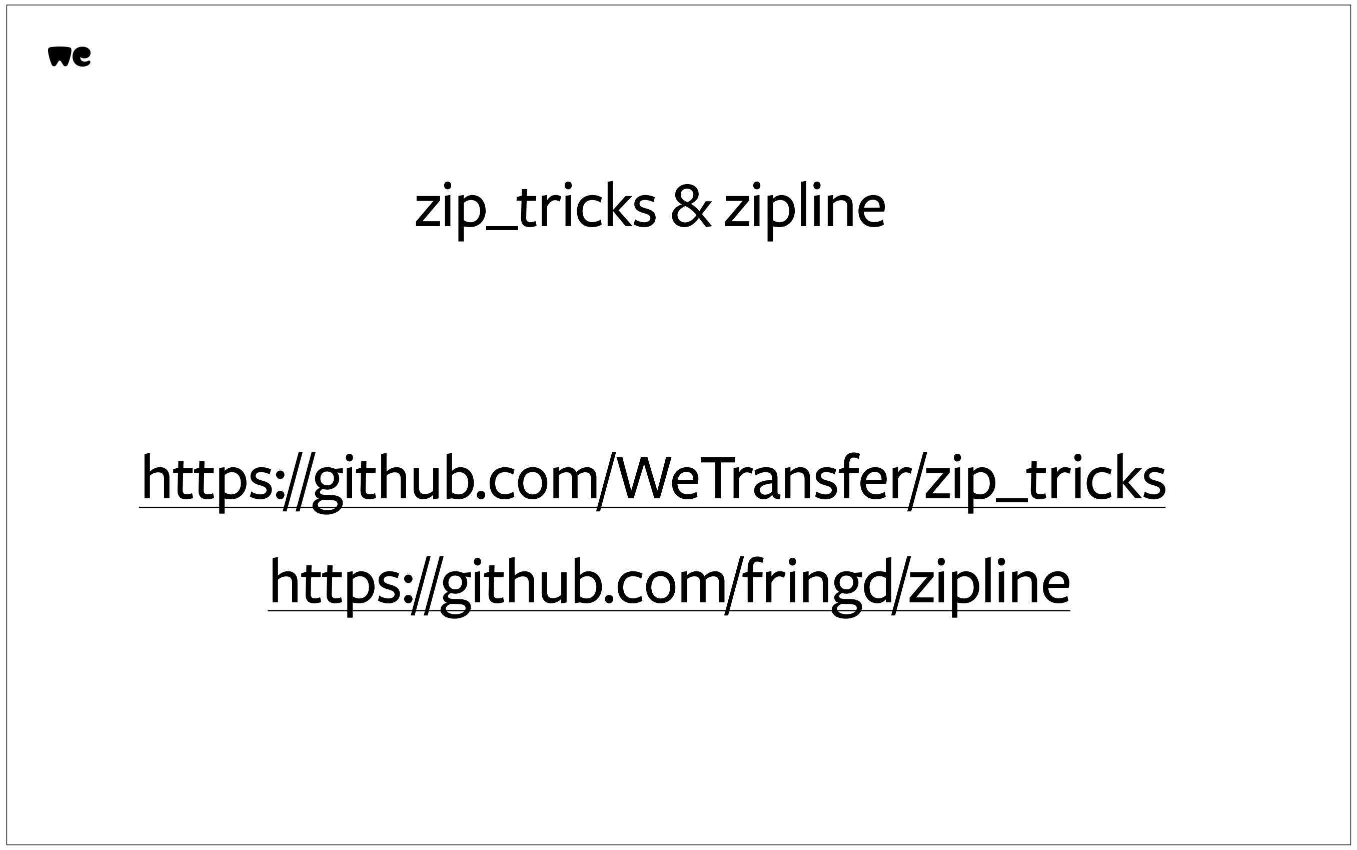
This is the library API in the end (well the most surface thing of it).



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

Control your dependencies

And benefits there were. For example, we could quickly fix an issue with one particular byte field in zip output that triggered one particular bug on one particular version of The Unarchiver on OSX. Very quickly.

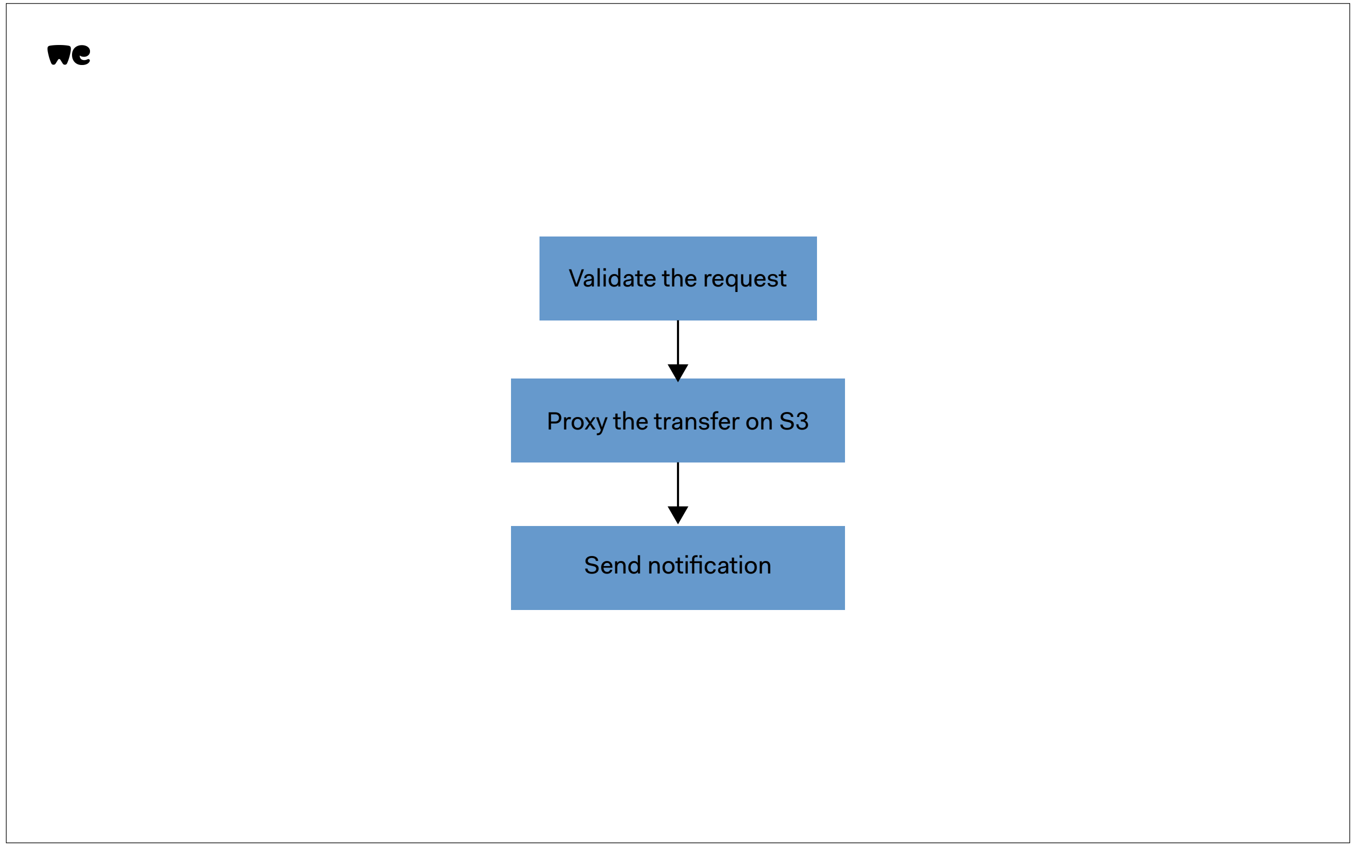


zip_tricks & zipline

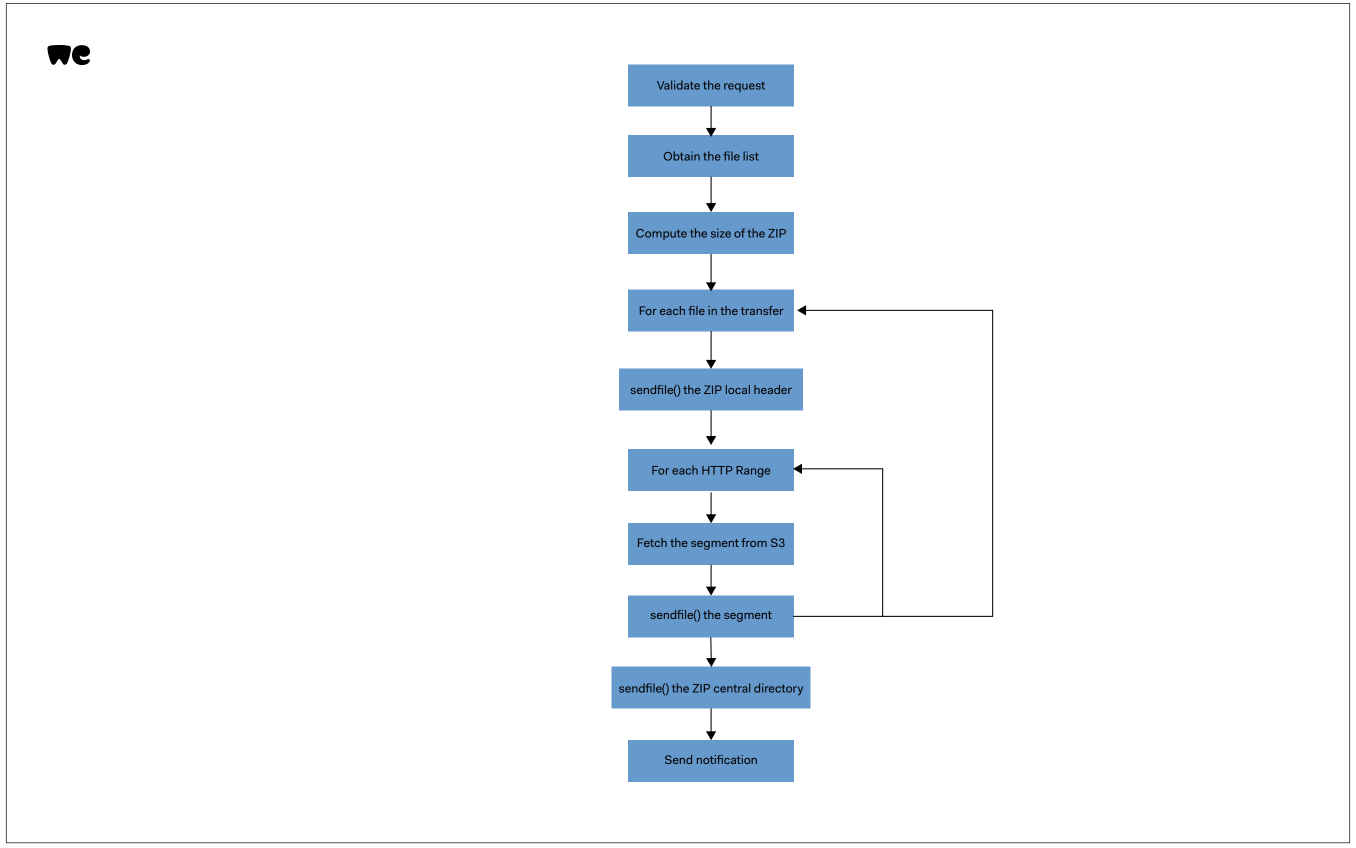
https://github.com/WeTransfer/zip_tricks

https://github.com/fringd/zipline

zipline is the original streaming ZIPs gem for Rails. It used to use RubyZip with different (and also ugly!) hacks, we helped move it to zip_tricks under the hood. If you are using zipline already and you update it, you will be using zip_tricks without even noticing.

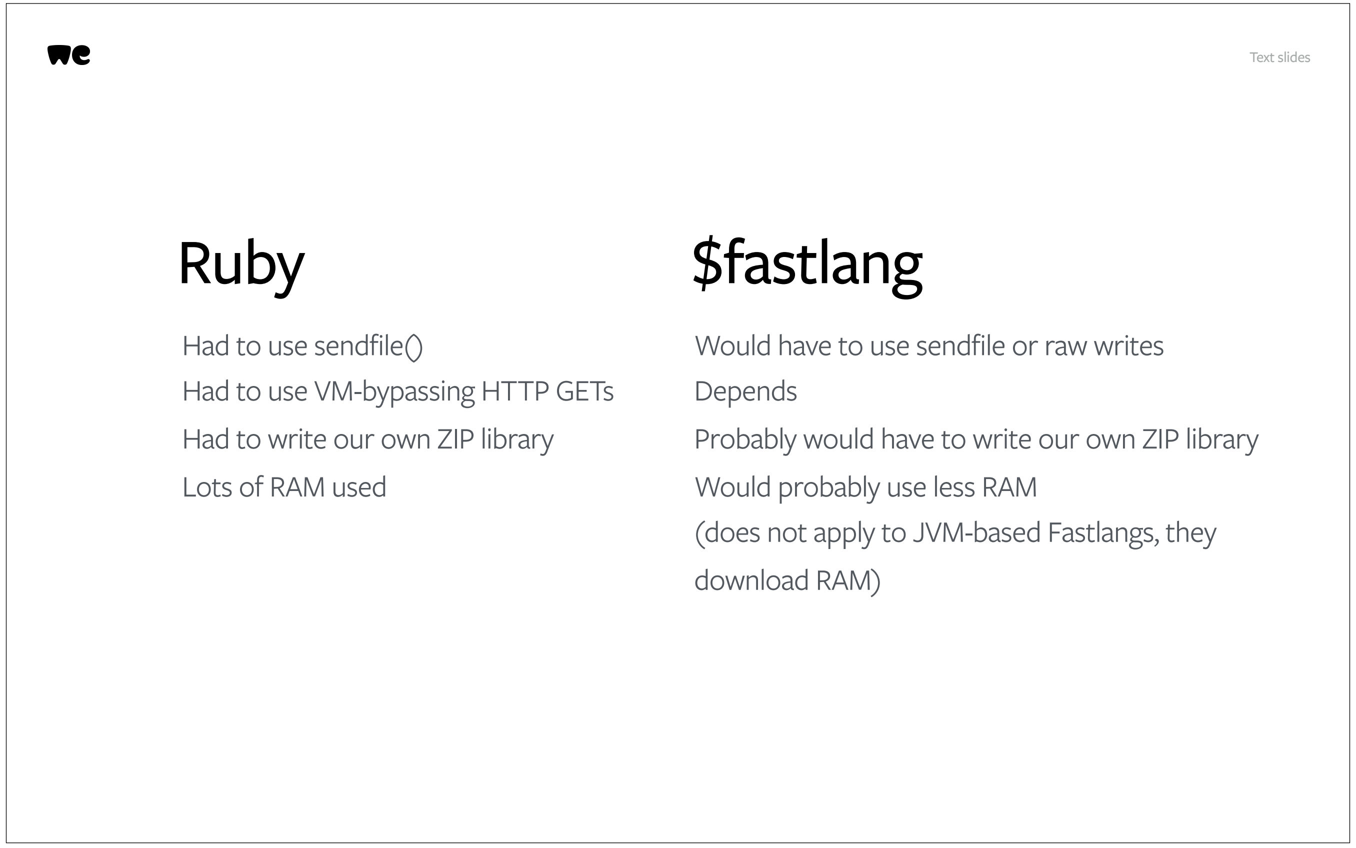


So we had to go from this…


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

To this. It is more complex, and there are more moving parts – but in return we get the feature we wanted. We get a decent language we can write on. We get unit tests for everything.

We even support dynamic `Range` requests on our end, so that download managers can be used – yes, we can exactly predict where the Range will land on a specific byte in the ZIP archive, and send out only that.

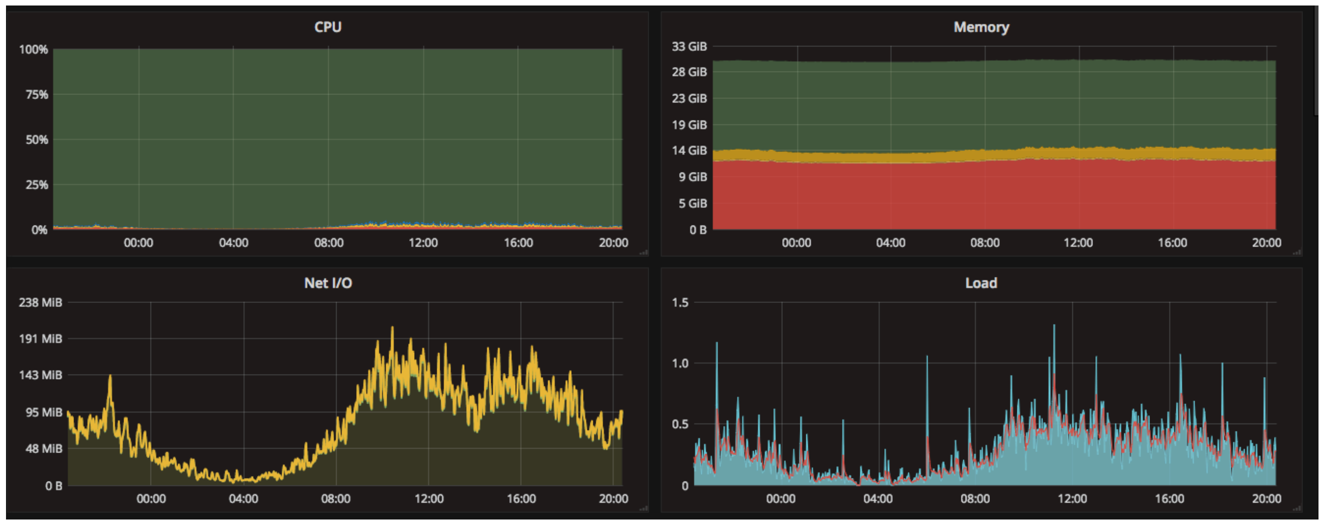


Text slides

Ruby	$fastlang
Had to use sendfile()	Would have to use sendfile or raw writes
Had to use VM-bypassing HTTP GETs	Depends
Had to write our own ZIP library	Probably would have to write our own ZIP library
Lots of RAM used	Would probably use less RAM
	(does not apply to JVM-based Fastlangs, they download RAM)

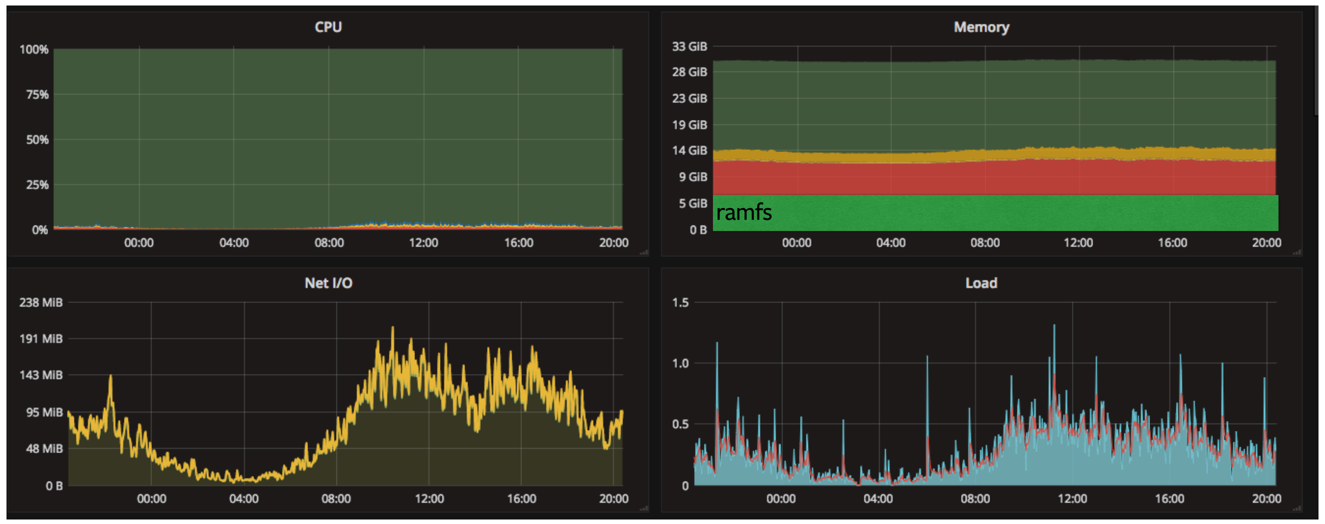
So imagine we went for a fastlang instead…

What is the actual overhead?



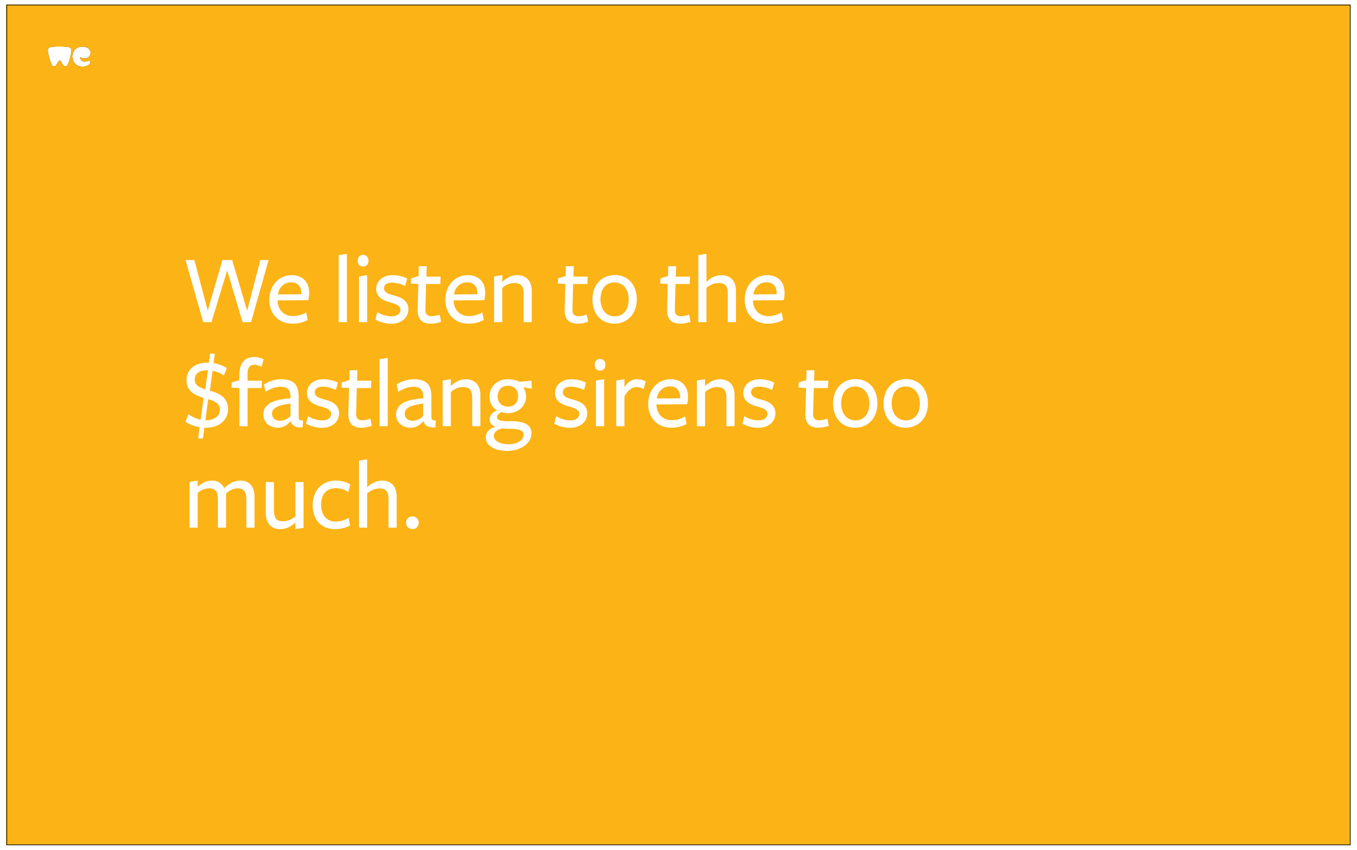
And this is how this whole thing runs and how much resources it consumes.

Wait a minute...



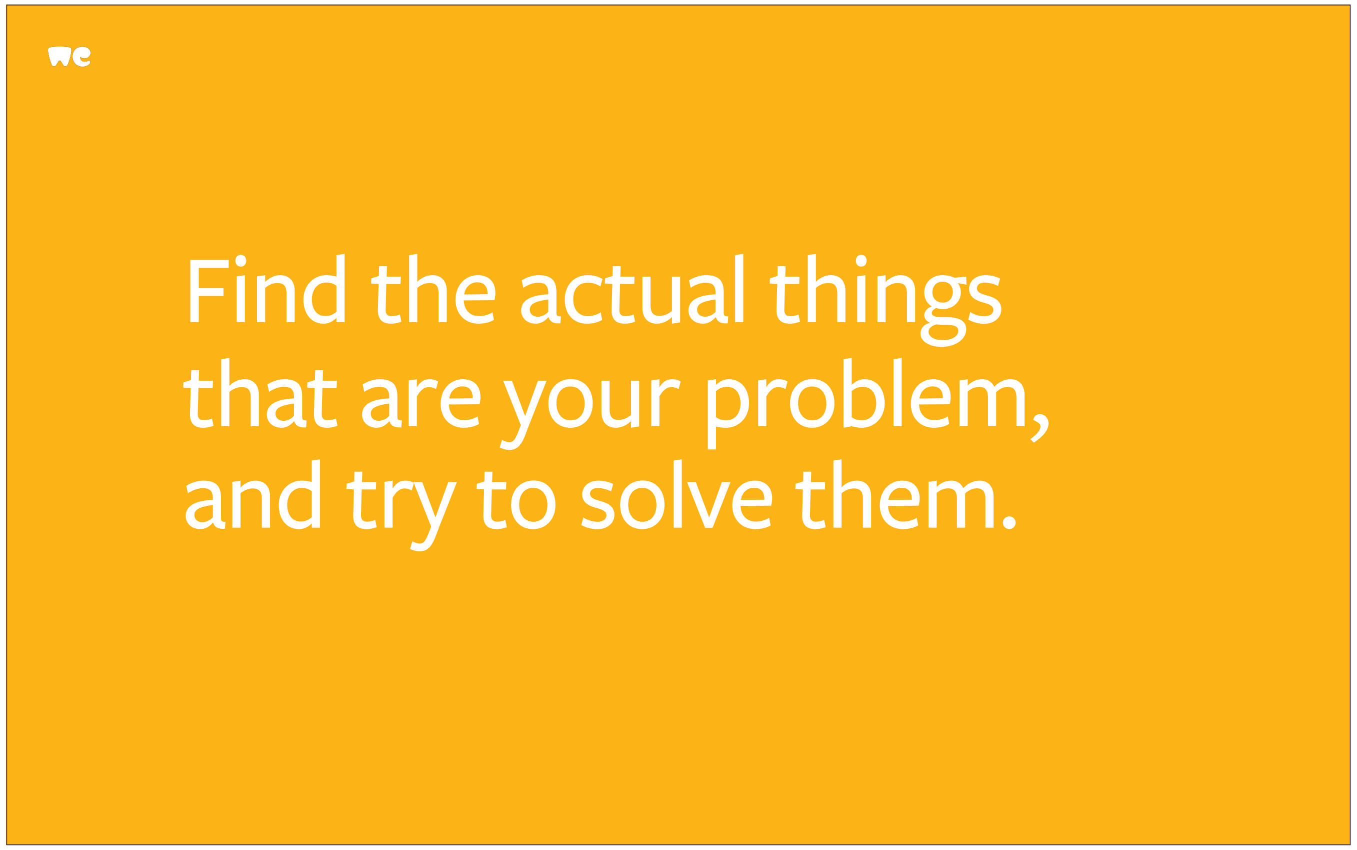
Don't be misled – due to AWS IOPS credit we also use part of that RAM (that we get because we have to rent a big box because we need good network throughput) as a filesystem. IOPS credit means that if we don't and we use tempfiles on the actual virtual machine's file system, after N hours the speed of the virtual hard drive of the EC2 instance degrades to the speed of a floppy drive.

Zero GC tweaks – we could have pushed it more. Near-0 CPU load. Yes, a lot of RAM. But a constant amount. We didn't have a single ENOMEM with this setup once all the bugs were ironed out.

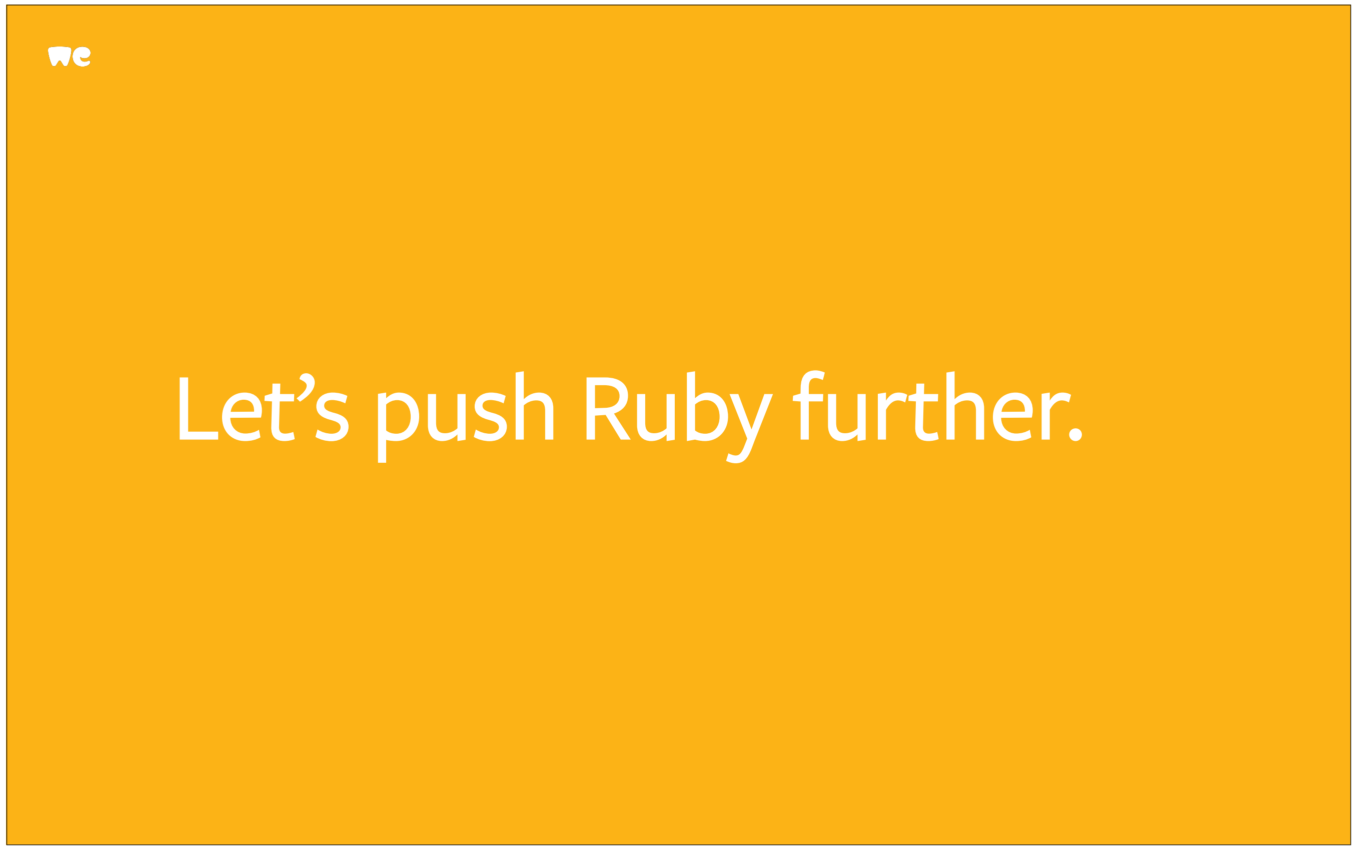


We listen to the $fastlang sirens too much.

In the end we listen to the siren calls of fastlangs too much. “Use that magic strict type-inferring compiled algebraic datatype JVM-based yadda yadda and all your problems will go away”. No they won’t because you probably havent analyzed the actual problems first.


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

For us it was the issue of having to move strings through the VM. The red scare of memory use is real, but in the end it turned out that with our load profile it didn’t matter! The two lines of Curl configuration in the previous implementation were the most important part of the previous PHP implementation pre-rewrite and this is what we had to reproduce, in fact. This is what mattered.



Let’s push Ruby further.

We can do more.

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