#ASLI ENGINEERING

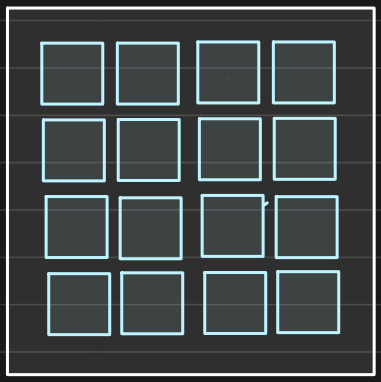
How Dropbox efficiently serves large number of thumbnails

SWIPE

BY

ARPIT BHAYANI

How Dropbox efficiently serves a large number of Thumbnails



We can upload photos, videos and other media objects to Dropbox. When we open the corresponding folder we can see all photos arranged in a grid.

Instead of serving the actual photo, Dropbox serves thumbnails of each

↓

smaller resolution photo → To save data transfer

Challenge: when user is quickly scrolling through the photos how can we serve large number of thumbnails quickly and efficiently

Note: We assume communication is happening over HTTP 1.1

Note: HTTP 2 has different approach to solve this problem.

Other related systems: Google Photos, Instagram, Flickr, and many more

ARPIT BHAYANI

Why is this even a problem ?

The answer is simple, Request Queueing

Browser / Client can creak at max 6 or 8 concurrent request to a domain

eg: say we have 60 photos of equal size
ie time to fetch any one photo is same

When your browser fires 60 requests to load 60 photos, the request will be made 6 at a time

Photos 1 to 6
Photos 7 to 12
⋮
Photos 55 to 60

while the other requests would be queued.

So, how did Dropbox solve this ?

Note: This is HTTP 1.1 based solution (excellent hack)

if you are on HTTP 2, a lot of other approaches can be taken.

Batching Request

Expose an endpoint (GET) that accepts multiple image paths

GET https://photos.dropbox.com/tbatch?
paths = /path/timg1.png, /path/timg2.png,
/path/timg3.png, /path/timg4.png

The server fetches these images in the backend, encodes it in base64 and responds

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-encoding: chunked
0:  ————————
2:  ————————
```

* → Transfer-encoding

image index → 0, 2

base 64 encoded image

1. server fetch images in parallel
2. it does not wait for all images before sending response

it sends chunked response

ARPIT BHAYANI

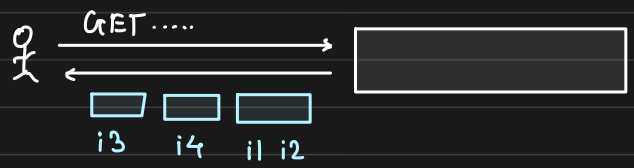
Chunked Transfer Encoding

Response header 'Transfer-Encoding: Chunked'

We use this when we do not know the length of the complete response.

Core idea: server keeps on sending HTTP response and once entire request is complete then it sends the termination chunk (NULL response)

Server fetchs all images in parallel. Whenever it has a few handy,



it creates a 'chunk' response and sends back to client

upon receiving images (encoded) the client plugs them in the <img src="—" > tag using JavaScript.

* we are kind of streaming the response from server a few image at a time, hence no Head of line block.