highscalability.com

How Twitter Handles 3,000 Images Per Second - High Scalability -

11-14 minutes

Today Twitter is creating and persisting 3,000 (200 GB) images per second. Even better, in 2015 Twitter was able to save $6 million due to improved media storage policies.

It was not always so. Twitter in 2012 was primarily text based. A Hogwarts without all the cool moving pictures hanging on the wall. It's now 2016 and Twitter has moved into to a media rich future. Twitter has made the transition through the development of a new *Media Platform* capable of supporting photos with previews, multi-photos, gifs, vines, and inline video.

Henna Kermani, a Software Development Engineer at Twitter, tells the story of the Media Platform in an interesting talk she gave at Mobile @Scale London: 3,000 images per second. The talk focuses primarily on the image pipeline, but she says most of the details also apply to the other forms of media as well.

Some of the most interesting lessons from the talk:

- **Doing the simplest thing that can possibly work can really screw you**. The simple method of uploading a tweet with an image as an all or nothing operation was a form of lock-in. It didn't scale well, especially on poor networks, which made it difficult for Twitter to add new features.
- **Decouple**. By decoupling media upload from tweeting Twitter was able independently optimize each pathway and gain a lot of operational flexibility.
- **Move handles not blobs**. Don't move big chunks of data through your system. It eats bandwidth and causes performance problems for every service that has to touch the data. Instead, store the data and refer to it with a handle.
- Moving to **segmented resumable uploads** resulted in

big decreases in media upload failure rates.

- **Experiment and research**. Twitter found through research that a 20 day TTL (time to live) on image variants (thumbnails, small, large, etc) was a sweet spot, a good balance between storage and computation. Images had a low probability of being accessed after 20 days so they could be deleted, which **saves nearly 4TB of data storage per day**, **almost halves the number of compute servers needed**, and saves millions of dollars a year.
- **On demand**. Old image variants could be deleted because they could be recreated on the fly rather than precomputed. Performing services on demand increases flexibility, it lets you be lot smarter about how tasks are performed, and gives a central point of control.
- **Progressive JPEG** is a real winner as a standard image format. It has great frontend and backend support and performs very well on slower networks.

Lots of good things happened on Twitter's journey to a media rich future, let's learn how they did it...

The Old Way - Twitter in 2012

The Write Path

- A user composes a tweet in an app and possibly attaches an image to it.
- The client posts the tweet to a monolithic endpoint. The image is uploaded as a bundle with all the other tweet metadata and passed around to every single service involved in the process.
- This endpoint was the source of a lot of problems with the old design.
- **Problem #1**: A Lot of Wasted Network Bandwidth
- Creation of the tweet and media upload were tightly coupled into one operation.
- Uploads were on shot, either the upload completely succeeded or completely failed. A failure for any reason, network hiccup, transient error, etc., required the whole upload process to restart from the beginning, including the media upload. The upload could get to 95% complete and if there was a failure it all had to be uploaded again.
- **Problem #2**: Doesn’t’ Scale Well for New Larger Media Sizes
- This approach would not scale to large media sizes like

video. Larger sizes increase the probability of failure, especially in emerging markets like Brazil, India, Indonesia, places with a slow and unreliable network, where they really want to increase the tweet upload success rate.

- **Problem #3**: Inefficient Use of Internal Bandwidth
- The endpoint connected to a TFE, Twitter Front End, that handles user authentication and routing. The user was routed to an Image Service.
- Image Service talks to the Variant Generator that generates instances of the image at different sizes (say small, medium, large, thumbnail). The variants are stored in the BlobStore, which is a key-value store optimized for large payloads like image and video. The images live there foreverish.
- There are number of other services involved in the process of creating and persisting a tweet. Because the endpoint was monolithic, combining media with the tweet metadata, this bundle flowed through all the services as well. This large payload was passed around to services that weren’t directly responsible for handling the image, they weren’t part of the media pipeline, but they were still forced to optimize for

handling large payload. This approach is very inefficient with internal bandwidth.

- **Problem #4**: Bloated Storage Footprint
- Images from tweets that were months and years old, that are no longer requested, would live in BlobStore forever, taking up space. Even sometimes when tweets were deleted the images would stay in BlobStore. There was no garbage collection.

The Read Path

- A user sees a tweet and the image associated with it. Where does the image come from?
- A client requests a variant of an image from a CDN. The CDN may need to ask the origin, TFE, for the image. This will eventually result in a direct lookup in BlobStore for an image for a URL at a particular size.
- **Problem #5**: Impossible to Introduce New Variants
- The design is not very flexible. Adding new variants, that is images of different sizes, would require backfilling the new image size for every image in the BlobStore. There was no variant on demand facility.
- The lack of flexibility made it difficult for Twitter to add new features on the client.

The New Way - Twitter in 2016

The Write Path

Decoupling media upload from tweeting.

- Uploading was made a first class citizen. An upload endpoint was created, it’s only responsibility is to put the original media in BlobStore
- This gives a lot of flexibility in how upload is handled.
- The client talks to TFE which talks to Image Service which puts the image in BlobStore and adds data into a Metadata store. That’s it. There are no other hidden services involved. No one is handling the media no one is passing it around.
- A mediaId, a unique identifier for the media, is returned from the Image Service. When a client wants to create a tweet, a DM, or update their profile photo, the mediaId will be used as a handle to reference the media rather than supplying the media.
- Let’s say we want to create a tweet with the media that was just uploaded. The flow goes like:
- The client hits the update endpoint, passing the mediaId in the post; it will hit the Twitter Front End; the

TFE will route to the service that's appropriate for the entity that is being created. For tweets it's TweetyPie. There are different services for DMs and Profiles; all the services will talk to the Image Service; The Image Server has post processing queues that handle features like face detection and child pornography detection; when that's finished the Image Service talks to ImageBird for images or VideoBird for videos; ImageBird will generate variants; VideoBird will do some transcoding; whatever media is generated will be put in BlobStore.

- No media is being passed around. A lot of wasted bandwidth has been saved.

Segmented resumable uploads.

- Walk into a subway, come out 10 minutes later, the upload process will be resumed from where it was left off. It's completely seamless for the user.
- A client initializes an upload session using the upload API. The backend will give it a mediaId that is the identifier to use through the entire upload session.
- An image is divided into segments, say three segments. The segments are appended using the API,

each append call gives the segment index, all appends are for the same mediaId. When the upload is completed the upload is finalized and the media is ready to be used.

- This approach is much more resilient to network failures. Each individual segment can be retried. If the network goes down for any reason you can pause and pick up the segment you left off at when the network comes back.
- A simple approach with huge gains. For files > 50KB there was a 33% drop in image upload failure rate in Brazil, 30% in India, and 19% in Indonesia.

The Read Path

Introduced a CDN Origin Server called MinaBird.

- MinaBird can talk to ImageBird and VideoBird so image size variants and video format variants can be generated on the fly if they don’t exist.
- MinaBird is more fluid and more dynamic in how client requests are handled. If there’s a DMCA Takedown, for example, it’s very easy to block access or reenable access to a particular piece of media.

- Being able to generate variants and transcodings on the fly let’s Twitter be a lot smarter about storage.
- On demand variant generation means all the variants do not need to be stored in BlobStore. A huge win.
- The original image is kept until deletion. Variants are only kept for 20 days. The Media Platform team did a lot of research on the best expiration period. About 50% of all requested images are at most 15 (or so) days old. Keeping images around that are older than that yields diminishing returns. Chances are nobody is requesting older media. There’s a very long tail after 15 days.
- With no TTL (time to live), no expiration, media storage results in a daily storage growth of 6TB every day. The lazy method, generating all variants on demand, results in a daily storage growth of 1.5TB. The 20 day TTL doesn’t use much more storage than the lazy method, so it doesn’t cost much in terms of storage, but it’s a huge win in terms of computation. Using the lazy approach of computing all variants on reads would require 150 ImageBird machines per datacenter versus 75 or so with the 20 day TTL. So the 20 day TTL is a sweet spot, a good balance between storage and computation.

- Since saving storage and computation is saving money, in 2015 Twitter saved $6 million by introducing the 20 day TTL.

Client Improvements (Android)

- Performed a 6 month experiment with WebP, a Google created image format.
- Images were on average 25% smaller than corresponding PNG or JPEG images.
- Saw increases in user engagement, especially in emerging markets, where the smaller image size cause less network stress.
- Not supported on iOS.
- Only supported on Android 4.0+.
- The lack of platform support made WebP costly to support.
- Progressive JPEG was another option Twitter tried. It renders in successive scans. The first scan might be blocky, but it will refine itself with successive scans.
- Better performance.
- Easy to support on the backend.

- 60% slower to encode than traditional JPEG. Since encoding happens once and serving happens many times it's not a huge problem.
- No transparency support, so transparent PNGs are kept around, but everything else is converging on progressive JPEG.
- On the client side support is provided by Facebook's Fresco library. Lots of very good things to say about Fresco. The results over a 2G connection were quite impressive. The first scan of PJPEG only requires about 10kb, so it doesn't take long to load. The native pipeline was still waiting to load, showing nothing, while the PJPEG was showing recognizable images.
- Results of an ongoing experiment for loads in the tweet detail view. A 9% decrease in p50 load times. A 27% decrease in p95 load times. A 74% decrease in failure rates. Users with slower connections really see a big win.

Related Articles

- On HackerNews
- The Architecture Twitter Uses To Deal With 150M Active Users, 300K QPS, A 22 MB/S Firehose, And

Send Tweets In Under 5 Seconds

- How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances
- DataSift Architecture: Realtime Datamining At 120,000 Tweets Per Second