Review of Haystack: Facebook's Photo Storage
D. Beaver, S. Kumar, H.C. Li, J. Sobel, and P. Vajgel
Jonathan Ledlie, Feb 17, 2011 (6.897 Spring 2011)

1 Problem

Since time immemorial, or at least for the past two decades, web sites have served up images using a standard setup. This setup assumes that the working set of images can fit into the memory of a reasonable set of servers. These servers mount (typically by NFS) read-only copies of the images and we're done: with enough memory (or CDN memory), the problem is solved.

The problem is when the request distribution is much flatter (much less concentrated) than the typical Zipfian one seen on the web. And this is the case with Facebook's photos (although they never actually show how flat it is). This flat request stream, coupled with a large absolute number of photos, means that no strategy will be able to keep the working set of images in memory: it is simply too big.

They note that their workload has the following characteristics: each user-provided image is read 1000x (i.e., reads dominate writes); reads occur soon after writes; photos are seldom modified (happens infrequently enough so that just making a new copy makes sense); somewhat rarely deleted (25% annually); low latency is essential.

2 Their Solution

Given these characteristics, Beaver et al. come up with a nice, sensible, application-level solution (no kernel mucking and/or new file system). They essentially have two main parts: (1) a metadata server, called Directory and (2) an http-accessible file server, called Store. In front of the Store, they put a Cache; and a CDN can (does) optionally sit in front of the Cache.

They spend much of the detail in the paper talking about the Store. Each Store is composed of 100GB physical volumes, called Haystacks. Three physical volumes from three distinct machines make up a logical volume. In order to find a photo, the Directory returns a URL which contains the physical machine, the logical volume and the photo's ID (and a cookie). When this URL is received by the Store (as denoted by the physical machine part of the URL), it is able to resolve the offset within the physical volume through an in-memory map. I.e., there is a per-logical-volume map from ¡photo id¿-¿¡offset1,size1¿,.. (there are four sizes of photos stored per photo uploaded). Because this resolution happens in memory, only one disk access is needed to read the photo from the disk (unless the photo crosses a RAID boundary).

They have one particularly nice optimization based on the observation that photos are often read soon after they are written. XFS, the underlying fs on the Stores, is bad at mixed read/write workloads. So it is better to limit Stores to either doing reads *or* writes, but not both, at a given time. They attempt to achieve this by only putting entries in the Cache if they are coming from write-enabled machines; otherwise they pass through the Cache. In effect, this shields write-enabled machines from reads.

3 Experimental Evaluation

I found the evaluation lacking. First, they do not actually show the concentration of the request distribution – how much memory would make the traditional solution work? Figure 8, which shows that nine machines running at the same time report the same workload due to hash-based load balancing, is a waste of space. Also, Table 3, on the volume of traffic, is poorly explained. Ideally, Table 4, which shows the results from different workloads, would compare to an alternative: how much of an improvement is it as compared to NFS?

One obvious, perhaps fixable, problem with the design is what to do with the traffic sitting on read-only machines that is still hot. As it stands, the only thing they can do is explicitly notice this at the Directory, and send these requests through the CDN. But they pay for the CDN and are trying to do away with its use. They show that 20% of traffic is not cached; i.e., is coming from read-only machines (Hmm, not necessarily - some could have been kicked out of the Cache). Also, according to Fig. 10, the read-only machines are doing 2-3× fewer ops/sec than the write-enabled ones, suggesting that the work could be shifted around somewhat more optimally.

4 Questions for Discussion

1. How would we introduce heterogeneous hardware into this design?
2. What are some design alternatives for this same workload?
3. Is this type of workload general enough for hosting it to be offered as a cloud service (by someone else)?
4. Are there other not-completely-general-but-common-enough workloads that could be offered as a service?