CS 695: Virtualization and Cloud Computing

Lecture 15: Application-specific cloud storage: Haystack

Mythili Vutukuru

IIT Bombay

Spring 2021

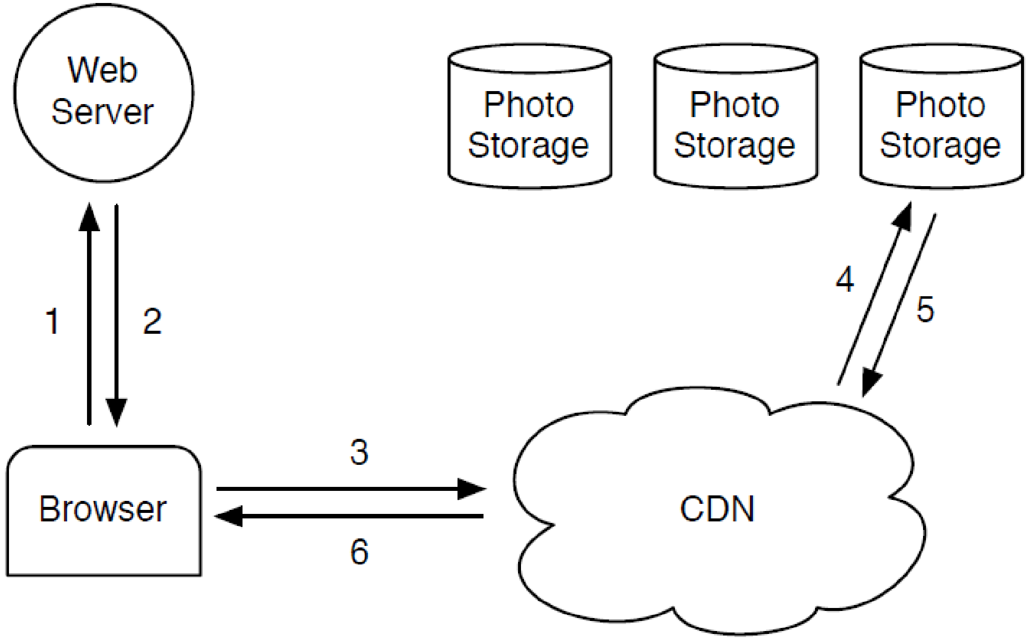
Facebook's photo storage: Haystack

- Some cloud storage systems are optimized for specific applications
  - Facebook's Haystack is optimized for photo storage
- Why not store photos as regular files on a POSIX-compliant filesystem?
  - Many attributes like permissions are meaningless
  - Lot of metadata accesses (inodes) before actual photo access
  - App specific knowledge: photos are written once, read often, rarely modified or deleted
  - High throughput, low latency, fault tolerance, with cost-effectiveness

Finding a needle in Haystack: Facebook's photo storage
Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel

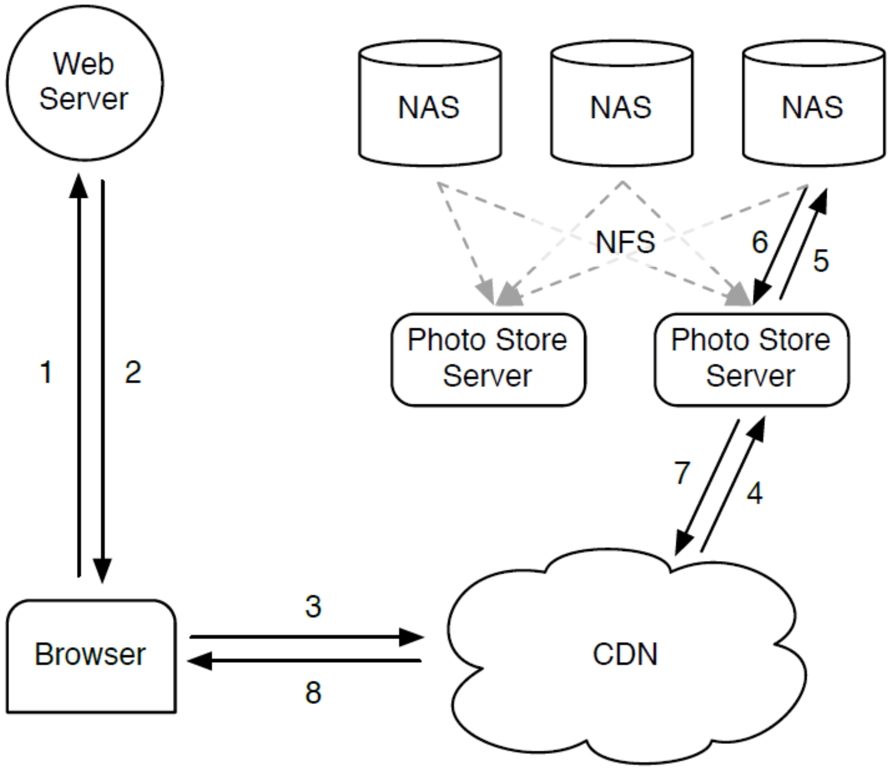
Typical design of photo/object storage

- Photos and other read-only objects are served from Content Delivery Networks (CDNs), e.g., Akamai
- DNS redirects to geographically closest CDN servers
- If object cached in CDN, served directly from CDN
  - Else, fetch object from original storage and serve, cache
- CDNs improve performance only for the hottest objects found in cache
  - Photos have a “long tail”: unpopular photos form significant part of traffic
- Need to optimize photo storage even if using CDN cache



NFS-based photo storage

- Each photo stored as a separate file on a commercial NAS (Network Attached Storage) box, served by NFS
- At least 3 disk accesses to read a photo from filesystem
  - Get inode number of file by reading parent directory blocks), read inode block, then fetch actual file
  - Large directories spread over multiple blocks incur even higher overhead
- Can cache inodes in memory to save disk accesses, but too much memory consumed to store all inodes of all files



Motivation and key idea

- Ideally, access photo directly on disk, without multiple disk accesses
  - Metadata (inode) to locate photo on disk should be in memory
  - However, caching all inodes for even unpopular photos is not possible
- Existing systems do not have the right “RAM-to-disk” ratio
  - Each photo as a separate file, each inode occupied ~100 bytes in memory
  - Too much memory for metadata in general purpose filesystems
- Goal: reduce metadata per photo, so that all metadata in memory, only one disk access even for unpopular photos
- Key idea: new filesystem, store multiple photos in large files, minimal metadata per photo
  - Redesigning filesystem is better than buying more NAS appliances / web servers / CDN storage

Haystack architecture

- 3 components: Store, Cache, Directory
- Store has the actual photos
  - Each server has many physical volumes (disks) which are organized into logical volumes
- Cache caches popular content that is not already cached in CDNs
- Directory maintains location mapping (which CDN/cache/store/logical volume may have photo)
  - When user requests photo from Facebook's webserver, it looks up directory
  - Directory returns a URL which encodes the location of the photo: CDN/Cache/Store/logical volume info

http://⟨CDN⟩/⟨Cache⟩/⟨Machine id⟩/⟨Logical volume, Photo⟩

  - Can check in CDN first, or directly go to cache
  - Balances load across store machines

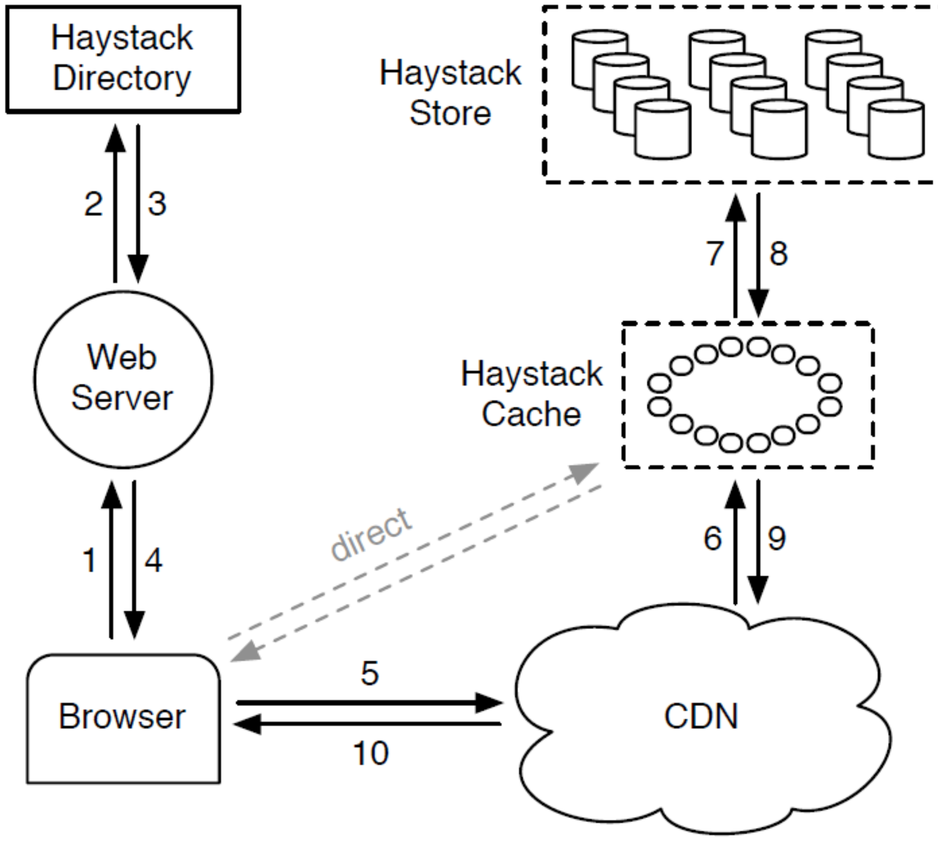


Figure 3: Serving a photo

Photo uploads

- Upload path:
  - Photo goes to webserver, which looks up directory
  - Directory returns the location of the Store server and logical volume where photo to be stored
  - A logical volume is replicated at multiple physical volumes for resiliency
  - Web server uploads photos at the multiple locations of a logical volume

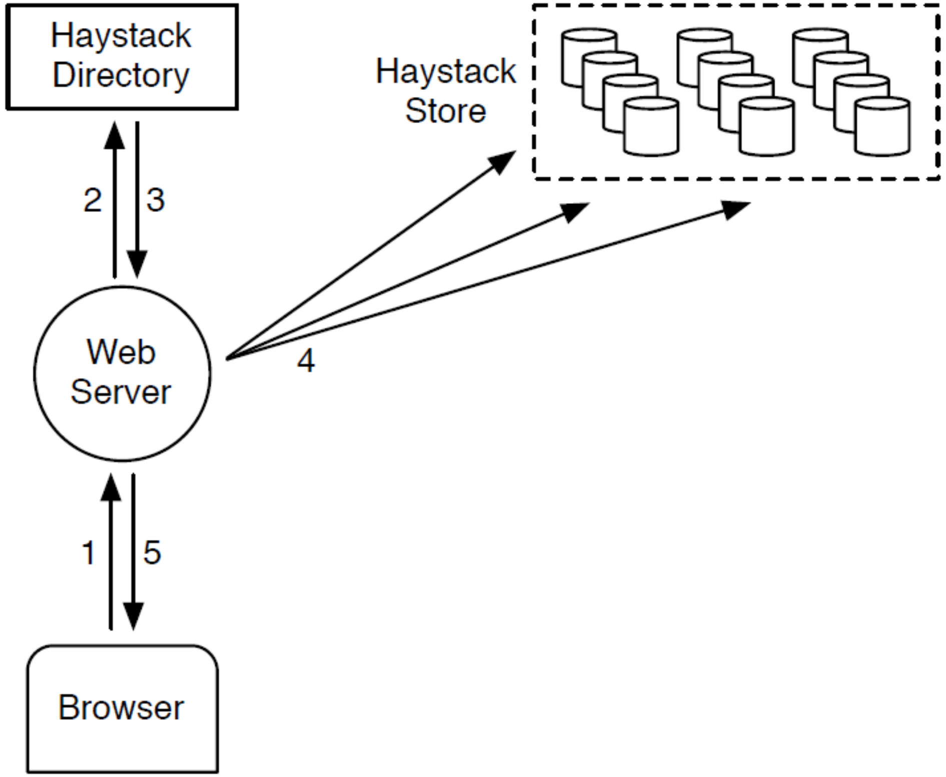


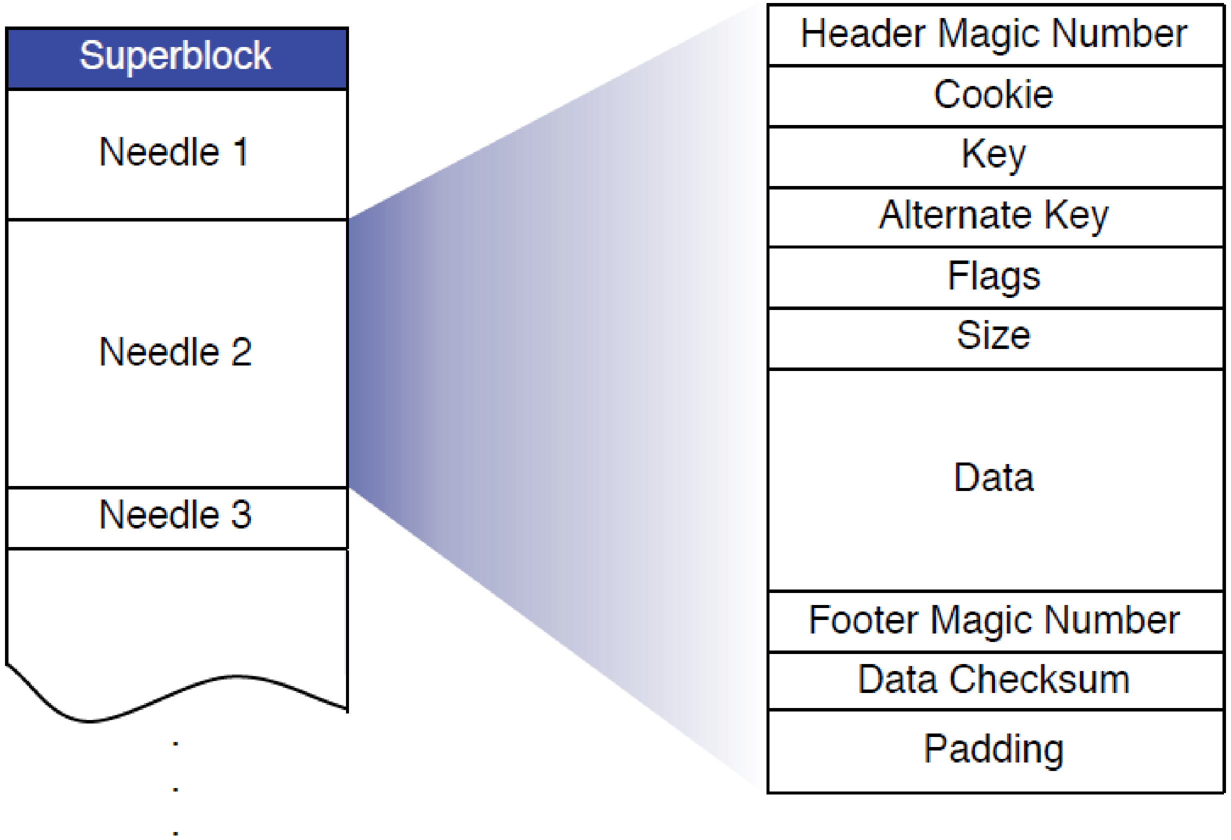
Figure 4: Uploading a photo

Store server architecture

- Each Store server has multiple physical volumes/disks
    - Physical volume is a large file (~100GB) with millions of photos
    - Each physical volume belongs to one of the logical volumes
- Logical volume = collection of different physical volumes on different servers
    - When a photo stored on a logical volume, it is replicated at all physical volumes of the logical volume, for resiliency
    - Directory has all info on physical and logical volumes on all store servers
- Photo identified by Store machine ID, logical volume, photo identifier/key
    - Go to server, find physical volume associated with logical volume, lookup photo
- New machine added to store: write-enabled, accepts uploads
    - Once capacity is full, moves to read-only mode, only serves photos
    - Cache mostly caches data from write-enabled store machines, because the most recently uploaded photos are frequently accessed by users

Store server: Disk Layout

- Each physical volume has a superblock and a set of "needles"
  - A single file with all photos
- Needle = photo + all its metadata (key, alternate key, size, etc.)
  - Alternate key is a way to distinguish multiple versions of a photo (e.g., different resolutions)
- Large file stored on disk using existing filesystems (XFS)



Store server: In-memory data structures

- Store server has open file descriptor for each physical volume
- In-memory index mapping photo key/alternate key to offset within disk
  - Lower overhead than full-fledged inodes
- Read request: lookup photo's key in index, find disk offset, read data from disk
  - Achieved goal of one disk access per photo
- Write request of new photo: appended to disk at end, index updated
- Modification/deletion of existing photo (rare): new copy appended at end, index updated to point to latest version
  - Modifications (e.g., rotations) have same key and alternate key
  - Old data not overwritten on disk for modifications or deletions, instead updated entry (or deletion record) is appended to disk
  - Periodic compactions of disk files to delete stale entries

Updating index file

- Where is index stored? In theory, no need to store index on disk, reconstructed from disk data on booting
  - If two entries on disk for same photo key (e.g., deletion or modification), index points to latest entry
  - However, may take a long time for large disks
- Periodically checkpoint index into a file on disk for quick bootup:
  - Index file written to disk asynchronously after appending actual data to disk
  - If system crashes after updating actual data but before updating index, index file on disk may be stale
  - For example, we can have orphans (photos on disk without entry in index)
  - During bootup, start with index file, see latest entry in index, all disk records after that are read and incorporated into index

Summary

- Application specific knowledge to optimize cloud storage
    - Not optimal to use general purpose filesystem for storing a specific type of files (photos) with specific usage patterns (write once, read multiple times, rarely modified)
    - Efficient disk layout and index design ensures close to one disk access per photo read
    - Good performance: benchmarks in paper show that Haystack achieves 85% of raw disk throughput, and only 17% extra latency (almost close to one disk access per photo)