Building Scalable Web Sites:
Tidbits from the sites that made it work

Gabe Rudy
What Is This About

- “Scalable” is hot
- Web startups tend to die or grow... really big
    - 03/2006 – 30 million videos a day
    - 07/2006 – 100 million videos a day
- Start with small budget
- LAMP
- Scale, but try to use commodity hardware
- Can’t we just throw some more servers at the problem?
Cred?

- I have no cred
  - Desktop apps
  - C++
  - Python for kicks
- But these guys do:
  - Chung Do (YouTube)
  - Cal Henderson (Flickr)
- So let's talk about
  - Software
  - Hardware
  - Databases
What It Breaks Down Into

- **Web Application**
  - Apache
  - + module friends
  - Python/PHP (no perl thank god)

- **Database Server**
  - MySQL
  - Memcache (more later)

- **Content**
  - Images
  - Videos

- **High load?**
  - Punt. Dual-quad Opteron with 16GB of RAM will go a long, long way.
Really? No hand crafted C?

- Wait? Aren’t Python/PHP slow?
- Doesn’t matter: They are fast enough
- What does matter: Developer time
- All the time gets spent:
  - On the wire (serving files, RPC, server communication)
  - Database
- But there are optimizations at every level
- Caching at every level
The Web App

- Build dynamic HTML
- Youtube used:
  - Apache with mod_fastcgi and Python
  - psyco (dynamic python -> C compiler)
  - A few C extensions for encryption
- Flickr used:
  - Apache with ? and PHP
  - Custom PHP framework
  - Async web design – requests get quick response, poll on status of server intensive tasks
- Cache?
  - Pre-generated, entire HTML for high traffic pages
Scaling Web Servers

- **Load balancing**
- **Hardware (Layer 4 or Layer 7)**
  - Bigger, faster, stronger
  - More expensive
  - Alteon, Cisco, Netscaler (YouTube), Foundry, etc
- **Software (still need hardware)**
  - mod_backhand
  - whackamole
- **End up integrating with your reverse proxy/CDN (more later)**
Scaling the DB

- Read/Write ratio generally 80/20 or 90/10
- Scale read capacity: MySQL replication
- Master (for writes) <-> Many slaves (for reads)
  - Writes go through master, propagates to all slaves
  - Does not scale
  - Soon slaves are spending 80% time syncing writes (unhealthy)
  - Propagation delay increases (read old values)
  - YouTube hacked MySQL master
    - Cache primer thread that pre-fetches reads of queries from disk to prevent stalling while handing write queue. Bought them time

- Master <-> Master pair
  - Provides High Availability
  - Reads faster than single master
  - Limits at most doubled
  - Still have row table limits etc

- Pragmatic solution...
Partitioning, Sharding, Federated Data

- **Vertical Partitioning**
  - Create partition of tables that will never need to be joined
  - Logical limits depending on application

- **Horizontal Partitioning – Sharding**
  - Same schema, partition by some primary field (like user)
  - Place each shard on a cluster (master-master pair?)
  - Spreads reads and writes
  - Better cache locality
  - Must avoid shard walking
  - Don’t assign to shard algorithmically
  - Requires central lookup cluster (hash table) to map user to shard
Shards Cont

- **Advantages**
  - Need more capacity? Just add more shards
  - Heterogeneous hardware is fine, just assign less/more objects per shard

- **Disadvantages**
  - App logic gets more complicated
  - More clusters to manage
  - Constrains lookups

- **Denormalization – Performance trick (not sharding)**
  - ‘Copied’ field from main table to linking table to make queries faster
  - Cached fields: Say in a parent/child relationship, cache count
Database Caching

- Write-through cache
- Write-back cache
- Sideline cache
  - Takes application logic (manually invalidate)
  - Usually best
  - Memcached
Content Caching

- **Reverse proxy/Caching proxy** – can be geographically dispersed
  - Scales well
  - Fast – usually serve out of memory
  - Dealing with invalidation is tricky
  - Direct prodding to invalidate scales badly
  - Instead, change URLs of modified resources
  - Old ones will drop out of cache naturally

- **CDN – Content Delivery Network**
  - Akamai, Savvis, Mirror Image Internet, Netscaler, etc
  - Operated by 3rd parties. Already in place
  - Gives you GSLB/DNS balancing (Global Server Load Balancing)
  - Once something is cached on CDN, assume that it never changes.
  - SSL can be at proxy point with SSL hardware
  - Sometimes does load balancing as well
CDN

Building Scalable Web Sites
Content Caching Cont

- **Versioning**
  - Rule of thumb: if item is modified, change it’s URL
  - Independent of your file system, can use mod_rewrite etc

- **Tools**
  - Squid (web cache, proxy server), GPL
  - Mod_proxy and mod_cache for Apache
  - Perlbal (mostly load balancer) and memcached
Sessions

- Non-trivial assuming load balanced application servers
- Local == bad:
  - PHP sessions are stored locally on disk
  - Can’t move users, can’t avoid hotspots, no fault tolerance
- Mobile local sessions:
  - Store last session location in cookie
  - If request gets to different server, then pull session info
- Single centralization database (or in-mem cache)
  - No hot spots, porting of session data, good
  - Problems scaling, bad
- Stash the whole damn session in a cookie!
  - “user_id + user_name + time + secret” -> sign -> base64
  - Timestamp can expire it
  - User_name is usually most used user info (“hello user_name”)
  - Fewer queries per page (some pages need little personalization)
Authentication (private and privileged content)

- Perlbal – reverse proxy can handle custom authentication modules
- Bake permissions into URL
  - Can do auth at web app
  - Use magic to translate URL to real resource paths
  - Don’t want paths to be guessable
  - Skips need for Perlbal or re-proxy step
- Downsides:
  - permission for live
  - Unless you bake in an expiring token
  - Ugly URLs?
- Upsides:
  - scales very nicely and works
File Storage

- Eventually outgrow single box’s capacity
- Horizontal scaling
- Remote file protocol?
  - NFS – stateful == sucks
  - SMB/Samba – stateful but degrades gracefully
  - HTTP – Stateless!
- At some point need multiple volumes
- At some point need multiple hosts
- Also want high availability and failure recovery (rebuild)
- So we can ignore RAID
Distributed fault tolerant file systems

- **MogileFS – application level distributed file system**
  - Metadata store (MySQL – can be clustered)
  - Replication automatic and piecemeal
  - I/O goes through tracker nodes that use storage nodes
- **GFS – Google File System (proprietary)**
  - “Master” node holds metadata (shadow master for warm swap)
  - Master node manages I/O (leases)
  - Grid of ‘chunkservers’, files usually dispersed among many servers
  - Designed to read large files fast
  - Automatic replication and self repairing
- **Flickr File System - proprietary**
  - No metadata store
  - App must store metadata – virtual volume number
  - StorageMaster nodes responsible for writing (organization)
  - Virtual nodes store data, are mirrored
  - Reading is done directly from nodes (scales really well)
- **Amazon S3 – big disk in the sky**
  - Files have user-defined keys
  - Data + metadata
  - Cost, linear and gets expensive as you scale
Story – YouTube thumbnails

- Loads of them (a dozen per video)
- Lots of disk seeks
- High # requests/sec (when you search, you get back thumbnails)
- Lots of files in the filesystem
  - Start to blow per-directory limits
  - Move to hierarchial storage
  - Sync between servers really slow

Interim solution
- Move from Apache to lighttpd
  - Then hack lighttpd to have I/O bound worker threads (it’s open source)

Long term solution
- Google Big Table
- Avoid small file problem
- Get fault tolerance
- Distributed access
- Caching
End

- Building Scalable Web Sites: Building, scaling, and optimizing the next generation of web applications -- Cal Henderson
- Scalable Internet Architectures -- Theo Schlossnagle