Scalable Web Architectures

Common Patterns & Approaches

Cal Henderson

Hello

Flickr

- Large scale kitten-sharing
- Almost 5 years old
  - 4.5 since public launch
- Open source stack
  - An open attitude towards architecture

Flickr

- Large scale
- In a single second:
  - Serve 40,000 photos
  - Handle 100,000 cache operations
  - Process 130,000 database queries

Scalable Web Architectures?

What does scalable mean?

What's an architecture?

An answer in 12 parts

1.

Scaling

Scalability – myths and lies

- What is scalability?

Scalability – myths and lies

- What is scalability ***not*** ?

Scalability – myths and lies

- What is scalability ***not*** ?
  - Raw Speed / Performance
  - HA / BCP
  - Technology X
  - Protocol Y

Scalability – myths and lies

- So what ***is*** scalability?

Scalability – myths and lies

- So what ***is*** scalability?
  - Traffic growth
  - Dataset growth
  - Maintainability

Today

- Two goals of application architecture:

Scale

HA

Today

- Three goals of application architecture:

Scale

HA

Performance

Scalability

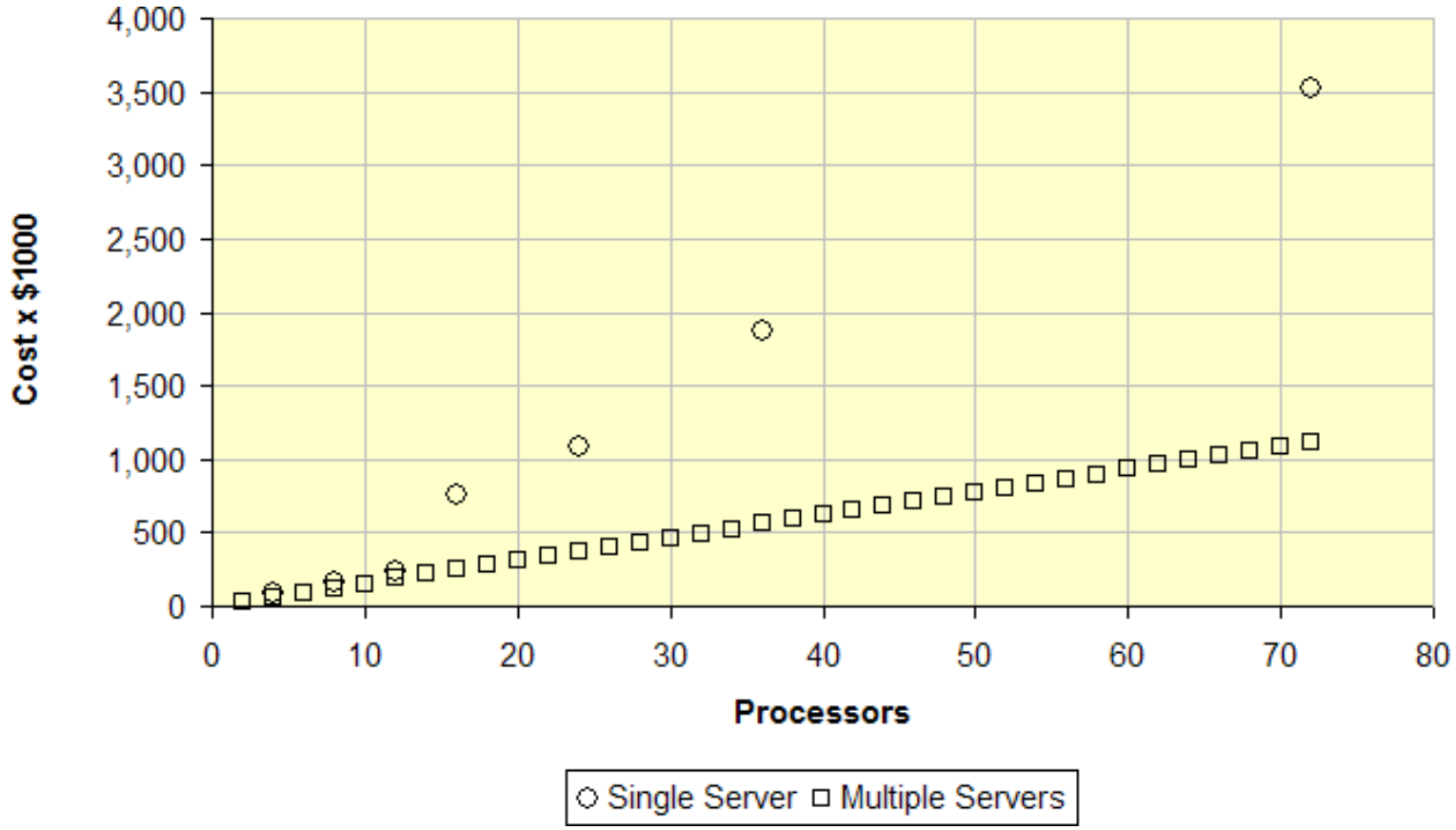
- Two kinds:
  - Vertical (get bigger)
  - Horizontal (get more)

Big Irons

Sunfire E20k
36x 1.8GHz processors
$450,000 - $2,500,000

PowerEdge SC1435
Dualcore 1.8 GHz processor
Around $1,500

Cost vs Cost


4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
Cost x $1000
0
10
20
30
40
50
60
70
80
Processors
Single Server
Multiple Servers

That's OK

- Sometimes vertical scaling is right
- Buying a bigger box is quick (ish)
- Redesigning software is not
- Running out of MySQL performance?
  - Spend months on data federation
  - Or, Just buy a ton more RAM

The H & the V

- But we'll mostly talk horizontal
  - Else this is going to be boring

2.

Architecture

Architectures then?

- The way the bits fit together
- What grows where
- The trade-offs between good/fast/cheap

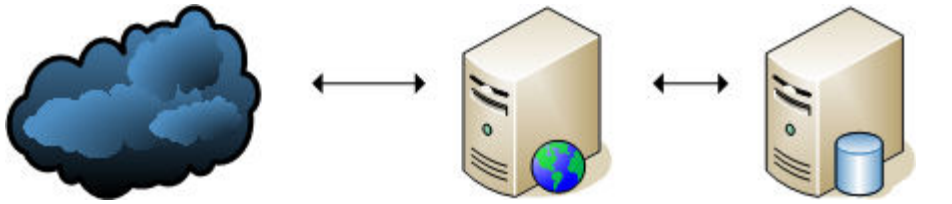
LAMP

- We’re mostly talking about LAMP
  - Linux
  - Apache (or LightHTTPd)
  - MySQL (or Postgres)
  - PHP (or Perl, Python, Ruby)

- All open source
- All well supported
- All used in large operations
- (Same rules apply elsewhere)

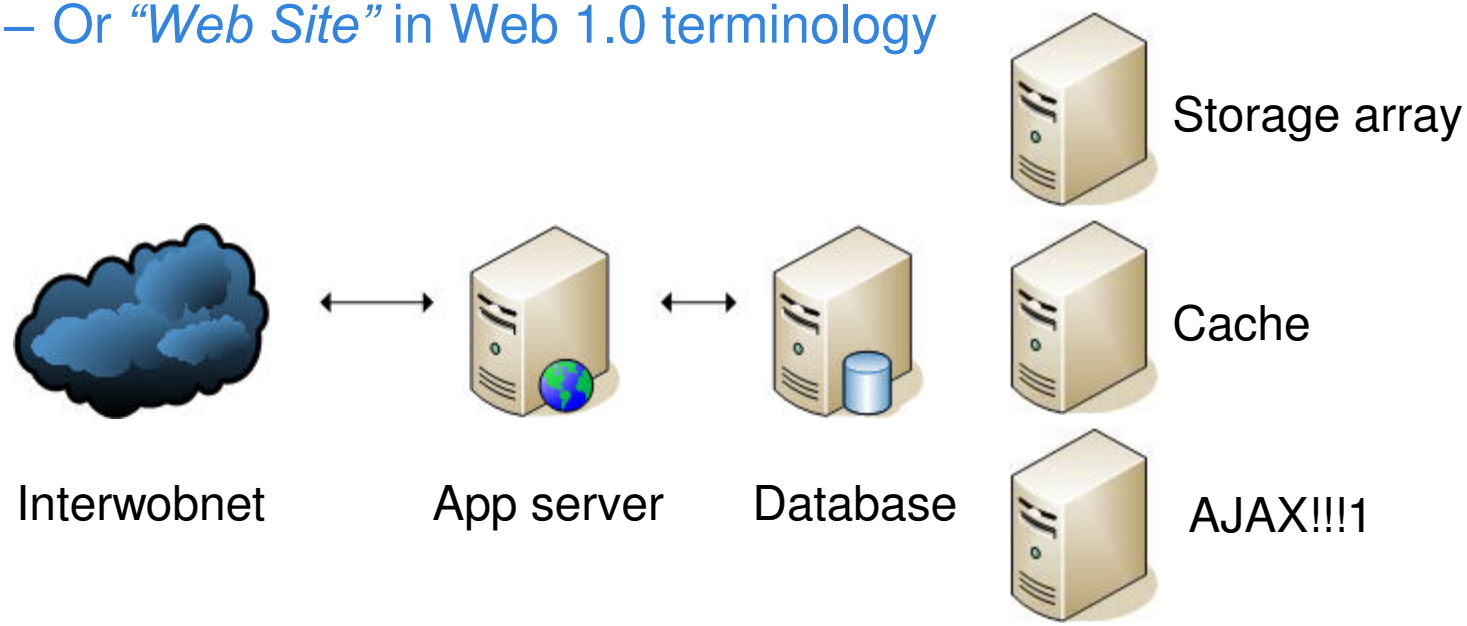
Simple web apps

- A Web Application
  - Or *"Web Site"* in Web 1.0 terminology



Simple web apps

- A Web Application
  - Or *"Web Site"* in Web 1.0 terminology



App servers

- App servers scale in two ways:

App servers

- App servers scale in two ways:
  - Really well

App servers

- App servers scale in two ways:
  - Really well
  - Quite badly

App servers

- Sessions!
  - (State)
  - Local sessions == bad
    - When they move == quite bad
  - Centralized sessions == good
  - No sessions at all == awesome!

Local sessions

- Stored on disk
  - PHP sessions
- Stored in memory
  - Shared memory block (APC)
- Bad!
  - Can’t move users
  - Can’t avoid hotspots
  - Not fault tolerant

Mobile local sessions

- Custom built
  - Store last session location in cookie
  - If we hit a different server, pull our session information across
- If your load balancer has sticky sessions, you can still get hotspots
  - Depends on volume – fewer heavier users hurt more

Remote centralized sessions

- Store in a central database
  - Or an in-memory cache
- No porting around of session data
- No need for sticky sessions
- No hot spots
- Need to be able to scale the data store
  - But we've pushed the issue down the stack

No sessions

- Stash it all in a cookie!

- Sign it for safety
  - $data = $user_id . '-' . $user_name;
  - $time = time();
  - $sig = sha1($secret . $time . $data);
  - $cookie = base64("$sig-$time-$data");

  - Timestamp means it's simple to expire it

Super slim sessions

- If you need more than the cookie (login status, user id, username), then pull their account row from the DB
  - Or from the account cache
- None of the drawbacks of sessions
- Avoids the overhead of a query per page
  - Great for high-volume pages which need little personalization
  - Turns out you can stick quite a lot in a cookie too
  - Pack with base64 and it's easy to delimit fields

App servers

- The Rasmus way
  - App server has ‘shared nothing’
  - Responsibility pushed down the stack
    - Ooh, the stack

Trifle

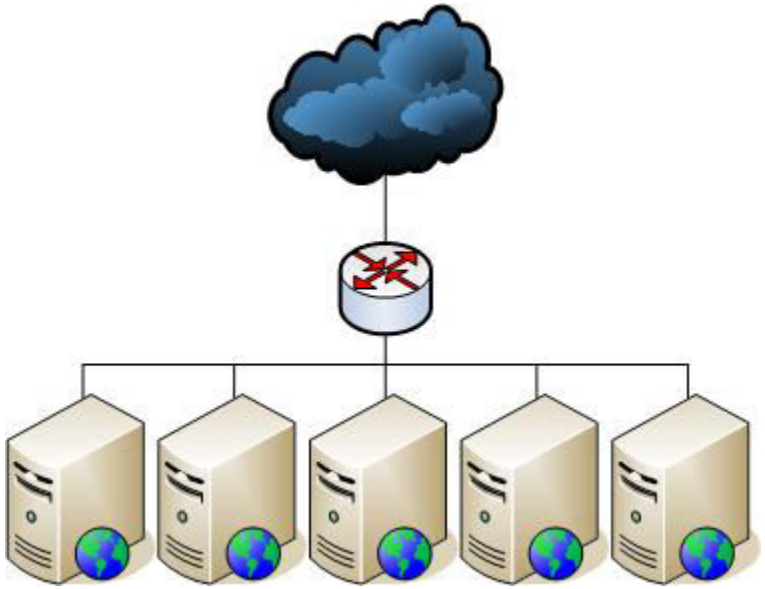
Trifle


Fruit / Presentation
Cream / Markup
Custard / Page Logic
Jelly / Business Logic
Sponge / Database

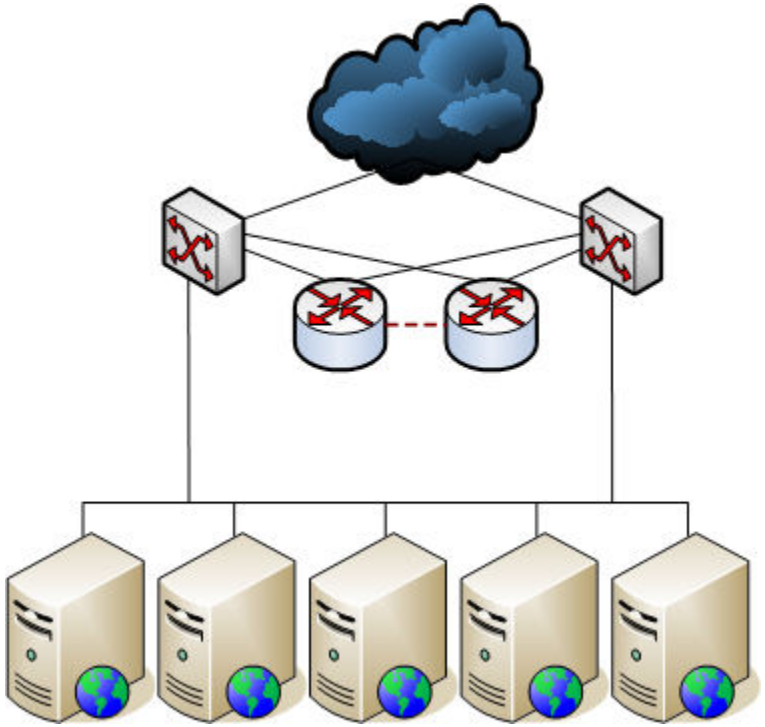
Trifle


Fruit / Presentation
Cream / Markup
Custard / Page Logic
Jelly / Business Logic
Sponge / Database

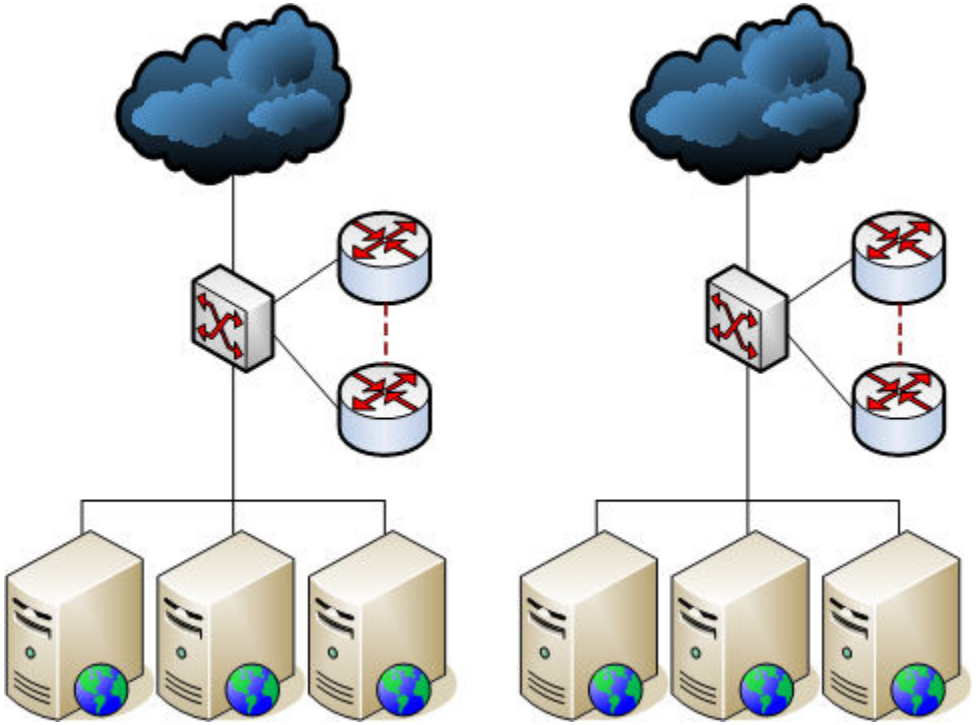
App servers



App servers



App servers



Well, that was easy

- Scaling the web app server part is ***easy***
- The rest is the trickier part
  - Database
  - Serving static content
  - Storing static content

The others

- Other services scale similarly to web apps
  - That is, horizontally
- The canonical examples:
  - Image conversion
  - Audio transcoding
  - Video transcoding
  - Web crawling
  - Compute!

Amazon

- Let's talk about Amazon
  - S3 - Storage
  - EC2 – Compute! (XEN based)
  - SQS – Queueing
- All horizontal
- Cheap when small
  - Not cheap at scale

3.

Load Balancing

Load balancing

- If we have multiple nodes in a class, we need to balance between them

- Hardware or software
- Layer 4 or 7

Hardware LB

- A hardware appliance
  - Often a pair with heartbeats for HA
- Expensive!
  - But offers high performance
  - Easy to do > 1Gbps
- Many brands
  - Alteon, Cisco, Netscalar, Foundry, etc
  - L7 - web switches, content switches, etc

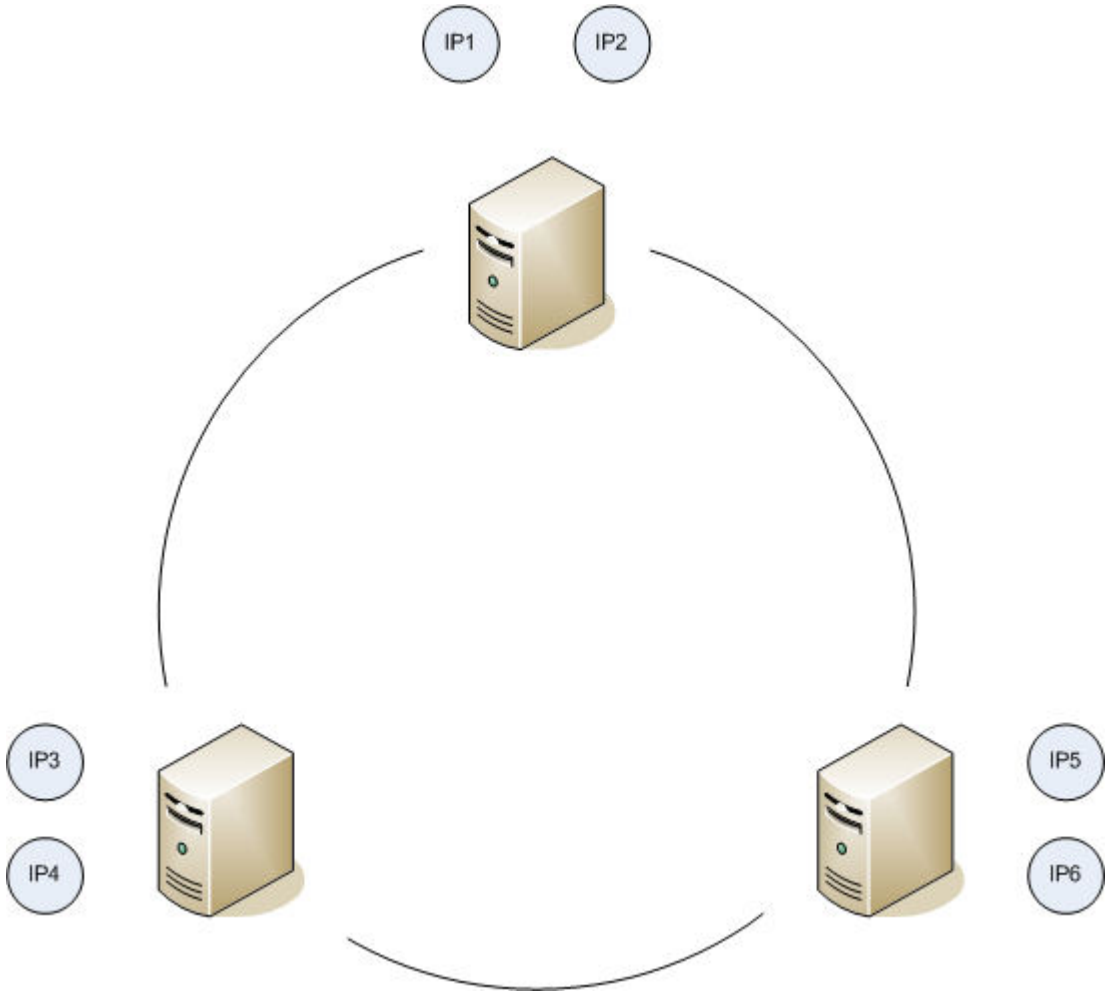
Software LB

- Just some software
  - Still needs hardware to run on
  - But can run on existing servers
- Harder to have HA
  - Often people stick hardware LB's in front
  - But Wackamole helps here

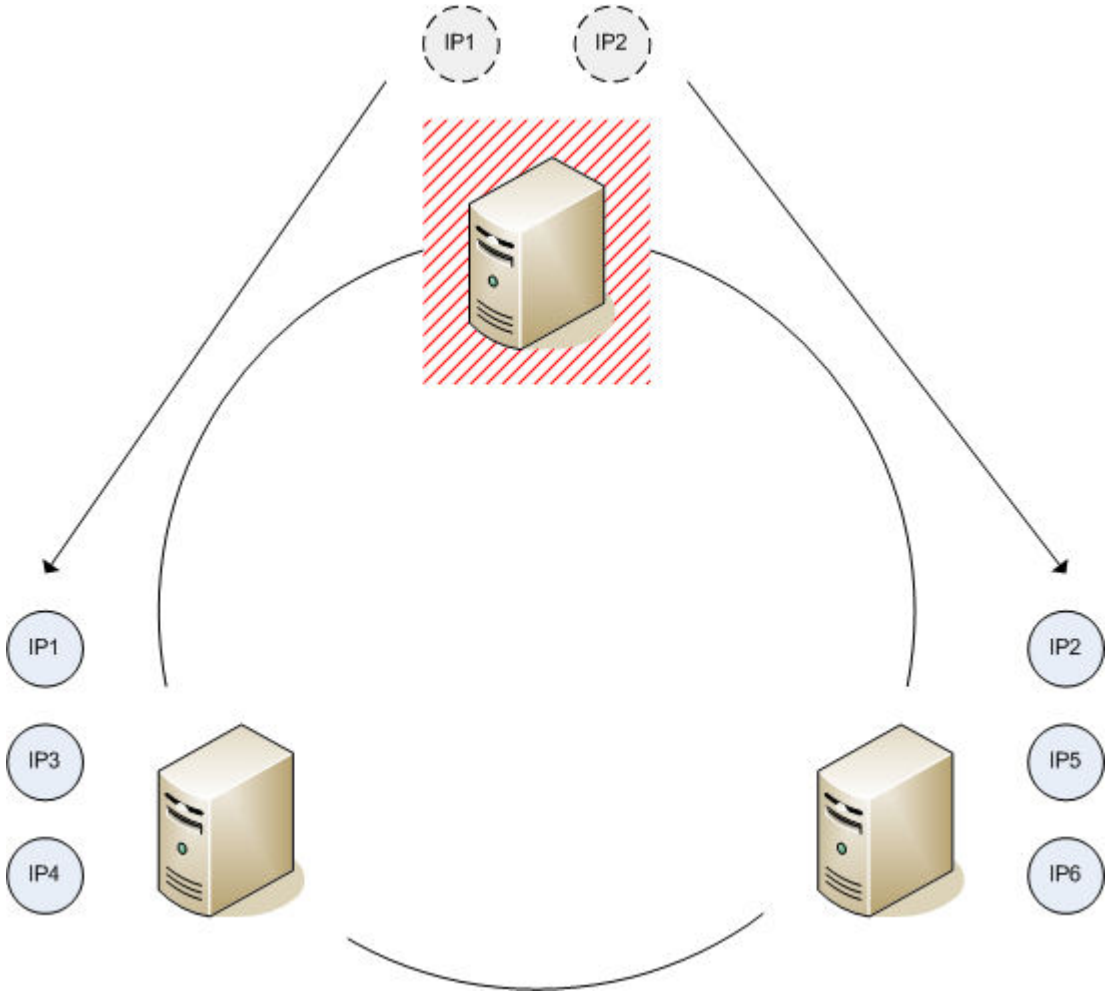
Software LB

- Lots of options
  - Pound
  - Perlbal
  - Apache with mod_proxy
- Wackamole with mod_backhand
  - http://backhand.org/wackamole/
  - http://backhand.org/mod_backhand/

Wackamole



Wackamole



The layers

- Layer 4
  - A ‘dumb’ balance
- Layer 7
  - A ‘smart’ balance
- OSI stack, routers, etc

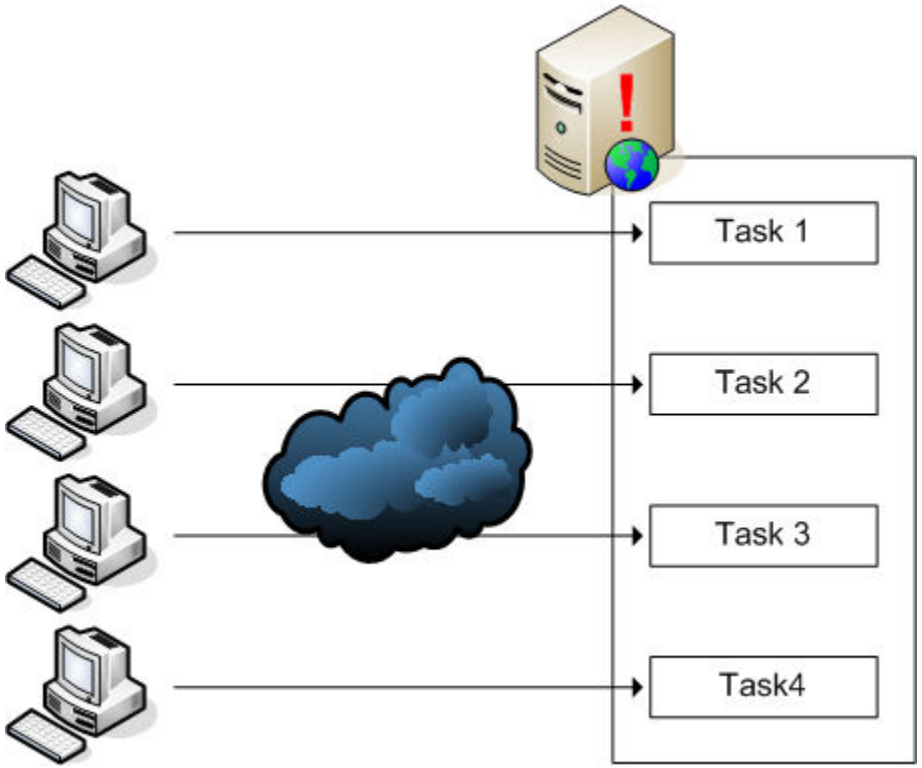
4.

Queuing

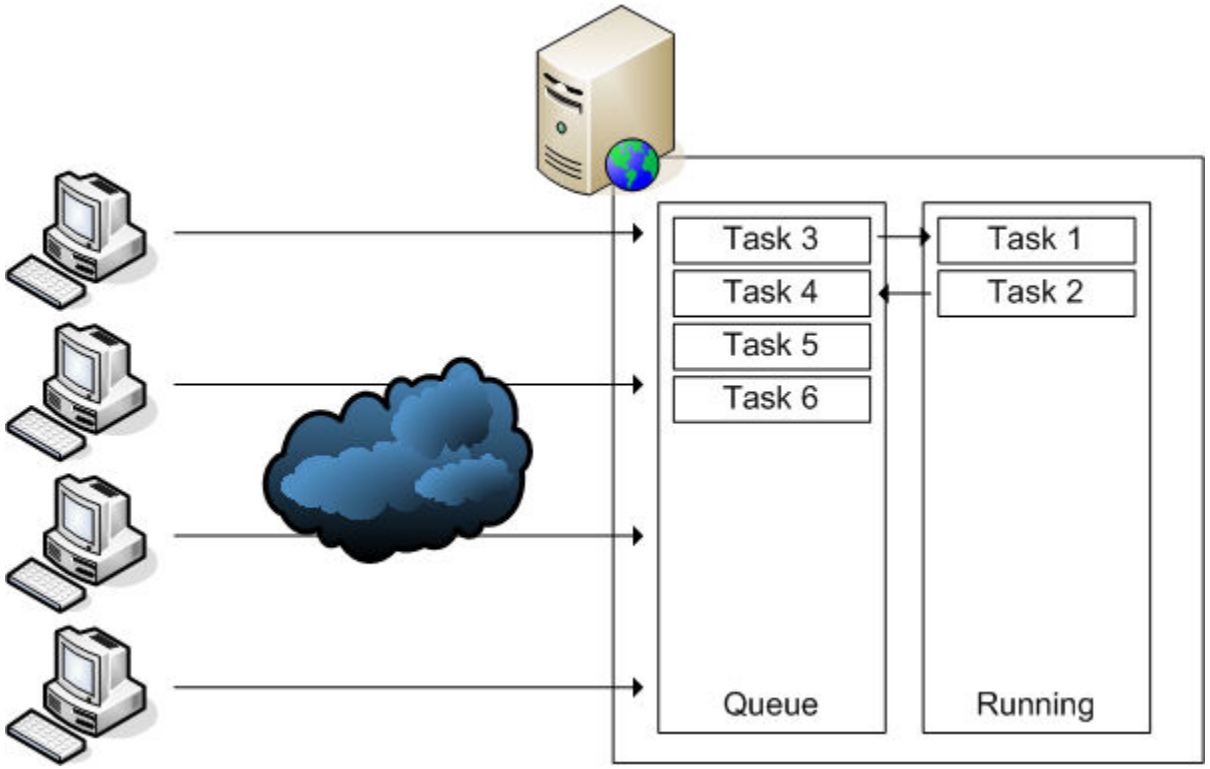
Parallelizable == easy!

- If we can transcode/crawl in parallel, it's easy
  - But think about queuing
  - And asynchronous systems
  - The web ain't built for slow things
  - But still, a simple problem

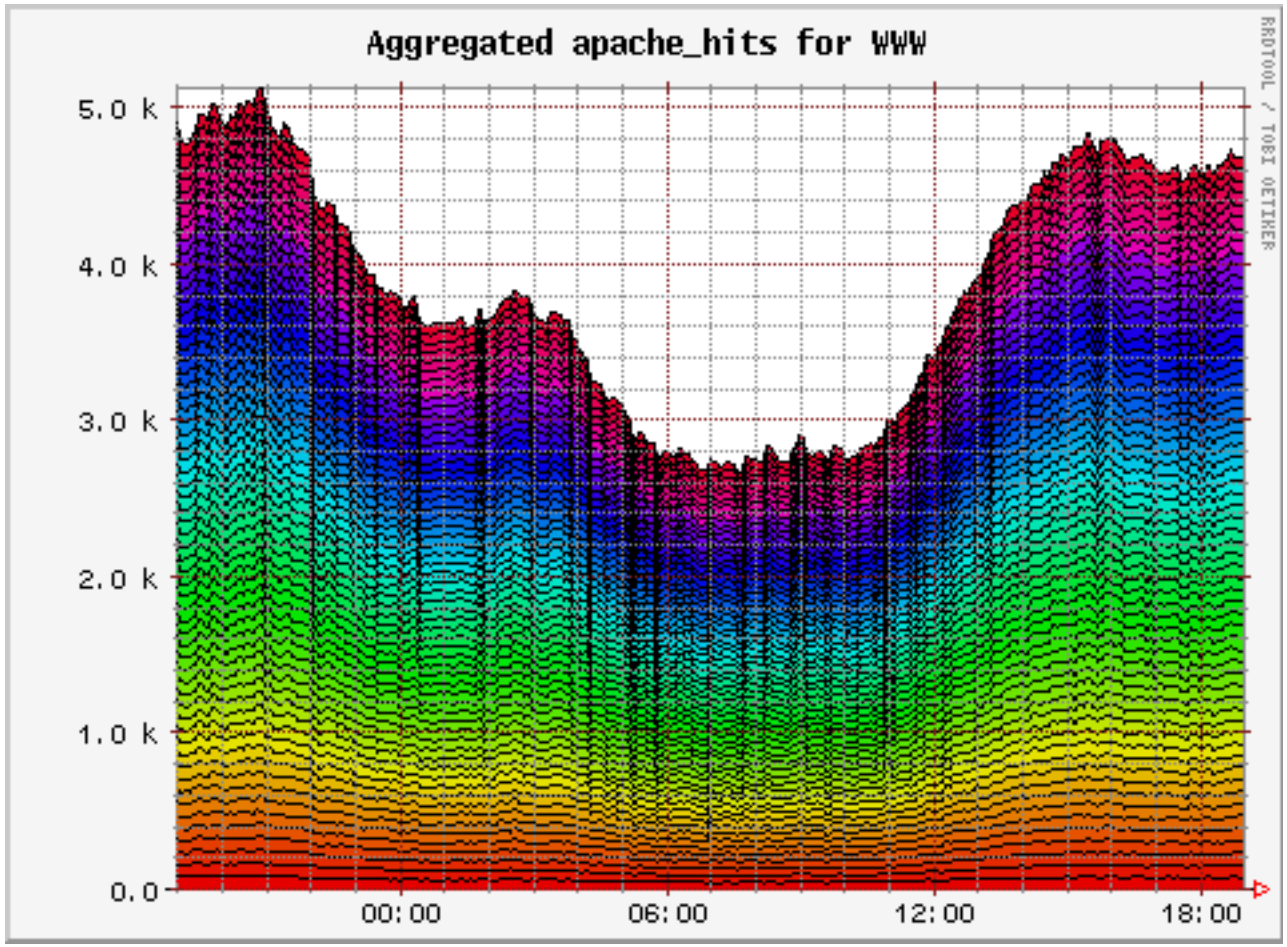
Synchronous systems



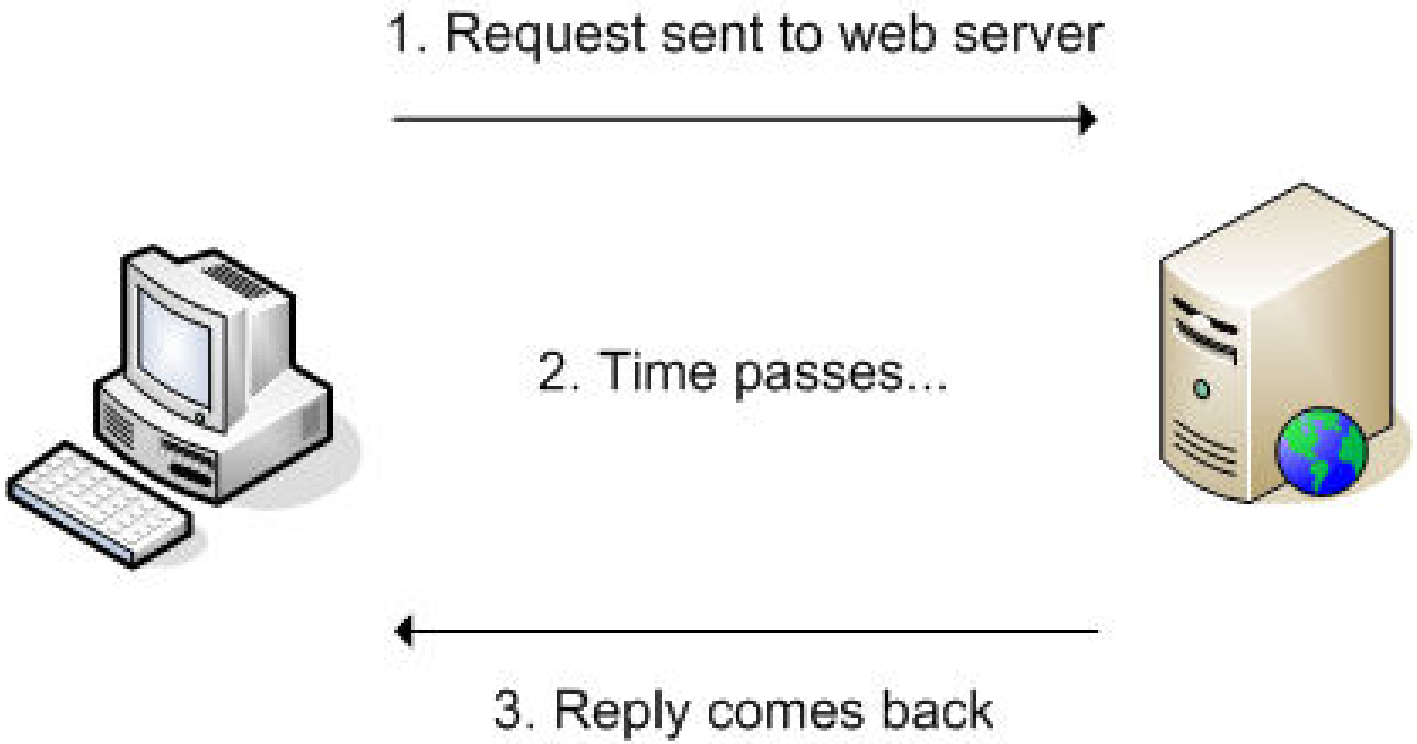
Asynchronous systems



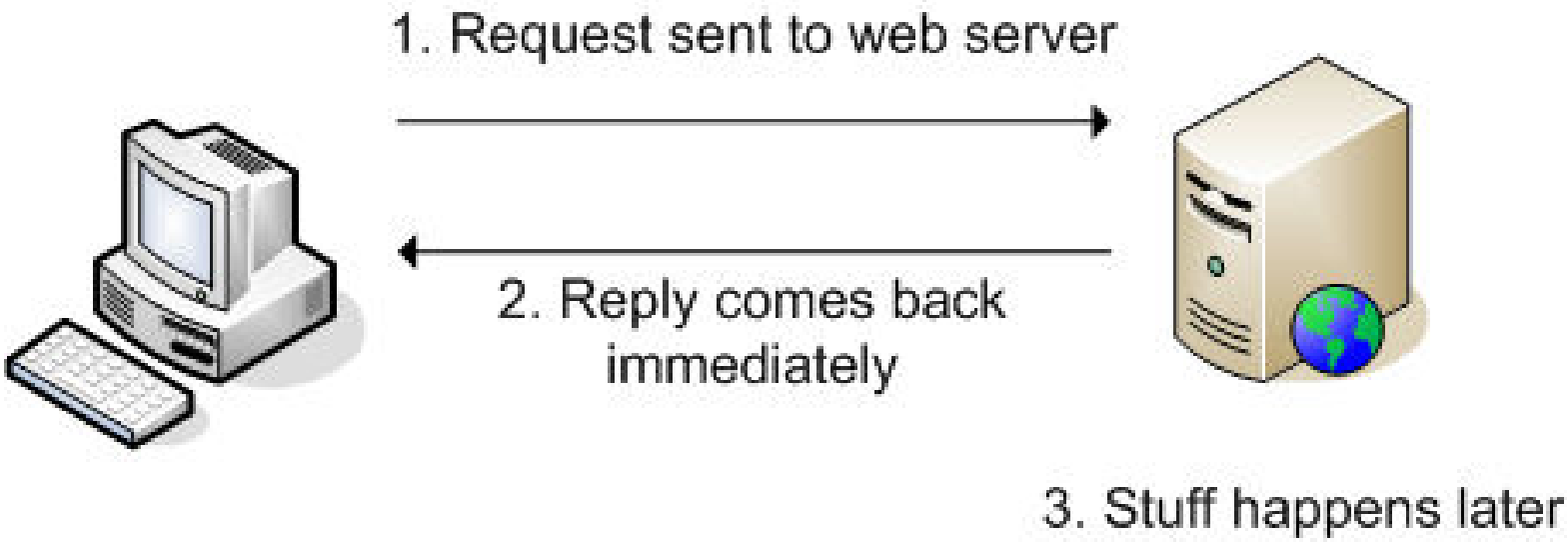
Helps with peak periods



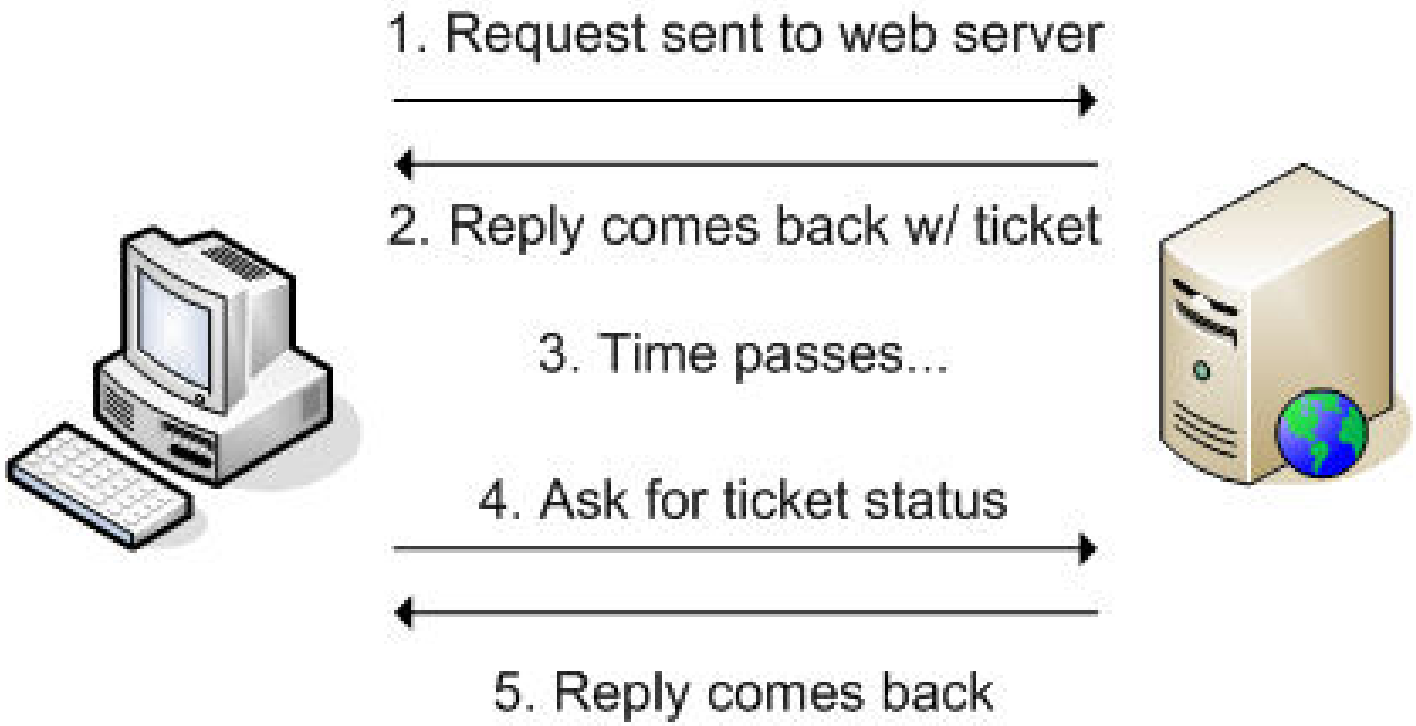
Synchronous systems



Asynchronous systems



Asynchronous systems



5.

Relational Data

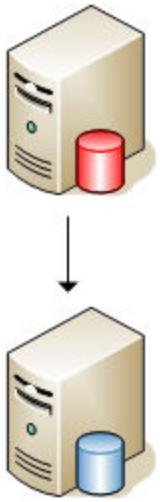
Databases

- Unless we're doing a lot of file serving, the database is the toughest part to scale
- If we can, best to avoid the issue altogether and just buy bigger hardware
- Dual-Quad Opteron/Intel64 systems with 16+GB of RAM can get you a long way

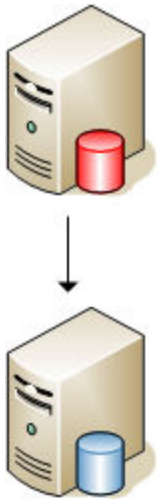
More read power

- Web apps typically have a read/write ratio of somewhere between 80/20 and 90/10
- If we can scale read capacity, we can solve a lot of situations
- MySQL replication!

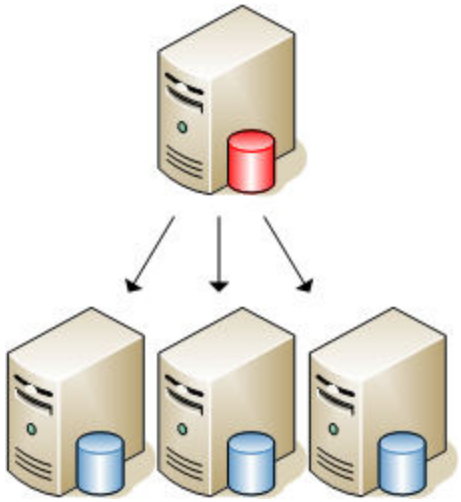
Master-Slave Replication



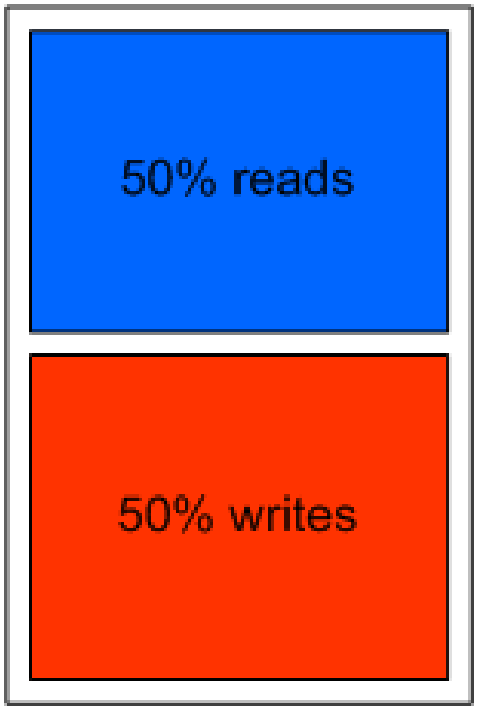
Master-Slave Replication



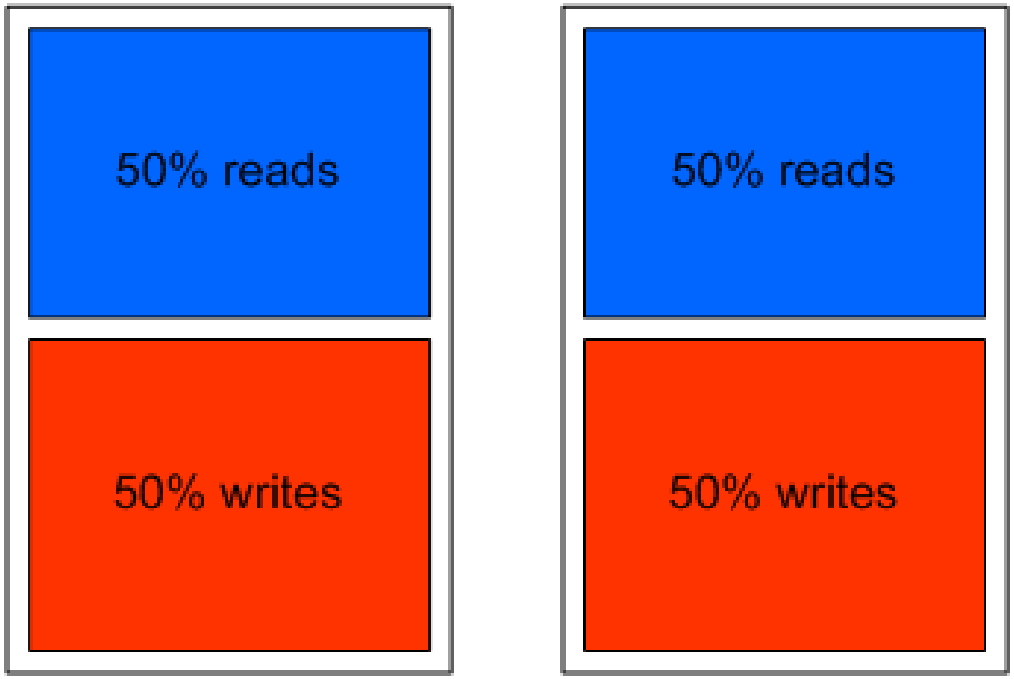
Master-Slave Replication



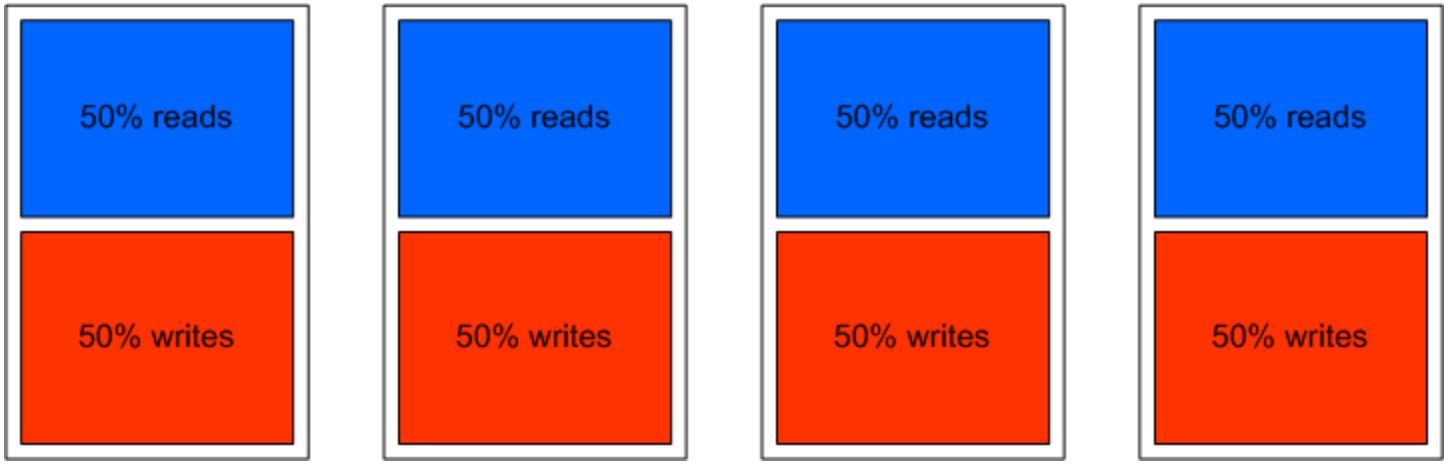
Master-Slave Replication



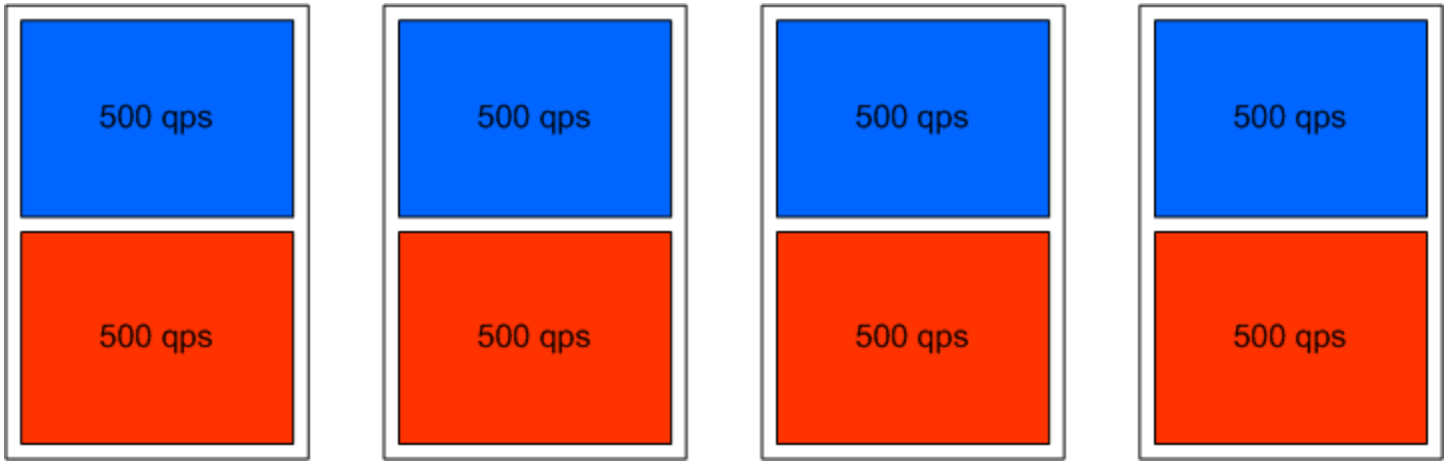
Master-Slave Replication



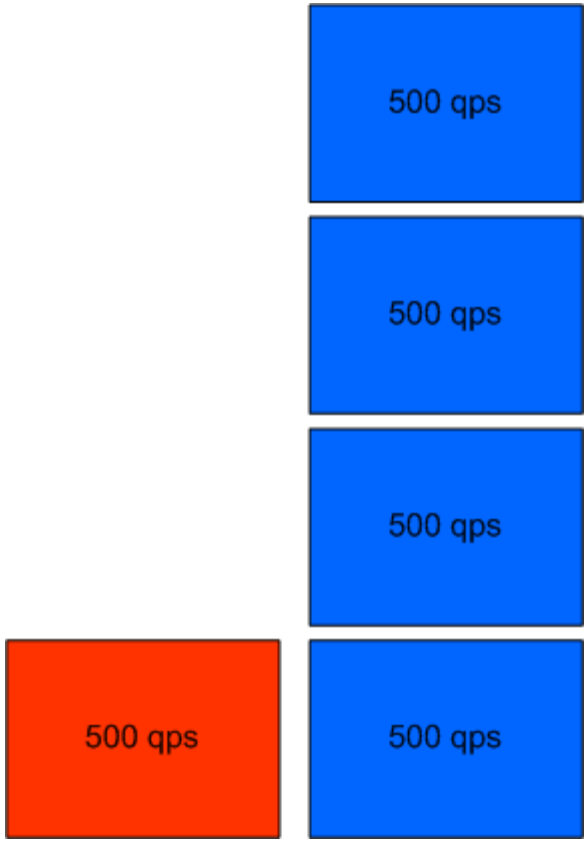
Master-Slave Replication



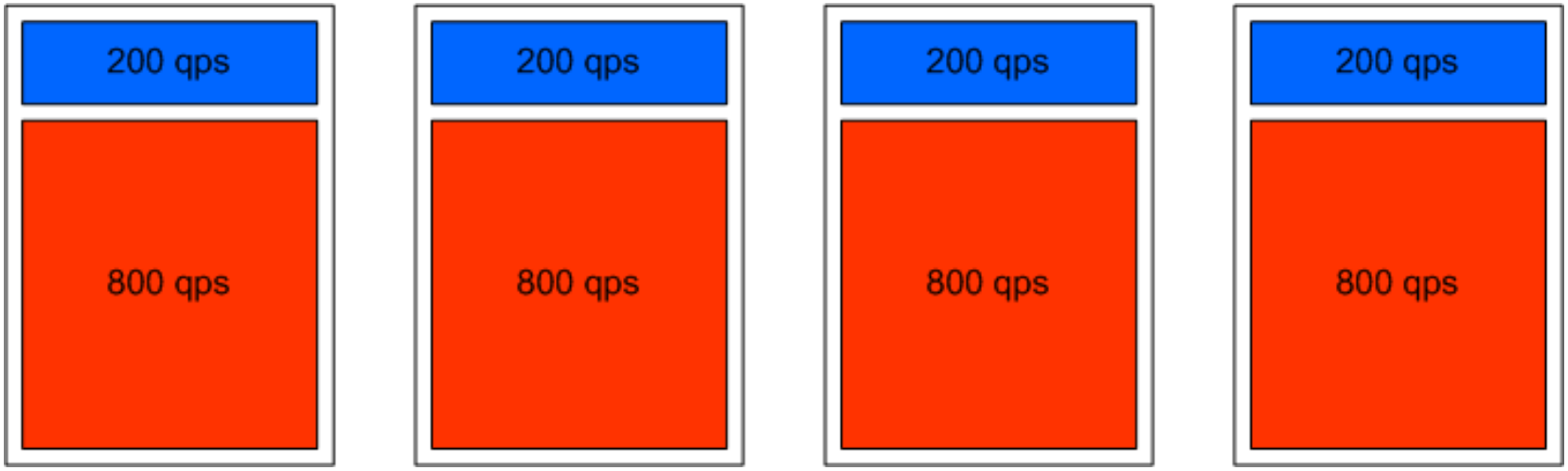
Master-Slave Replication



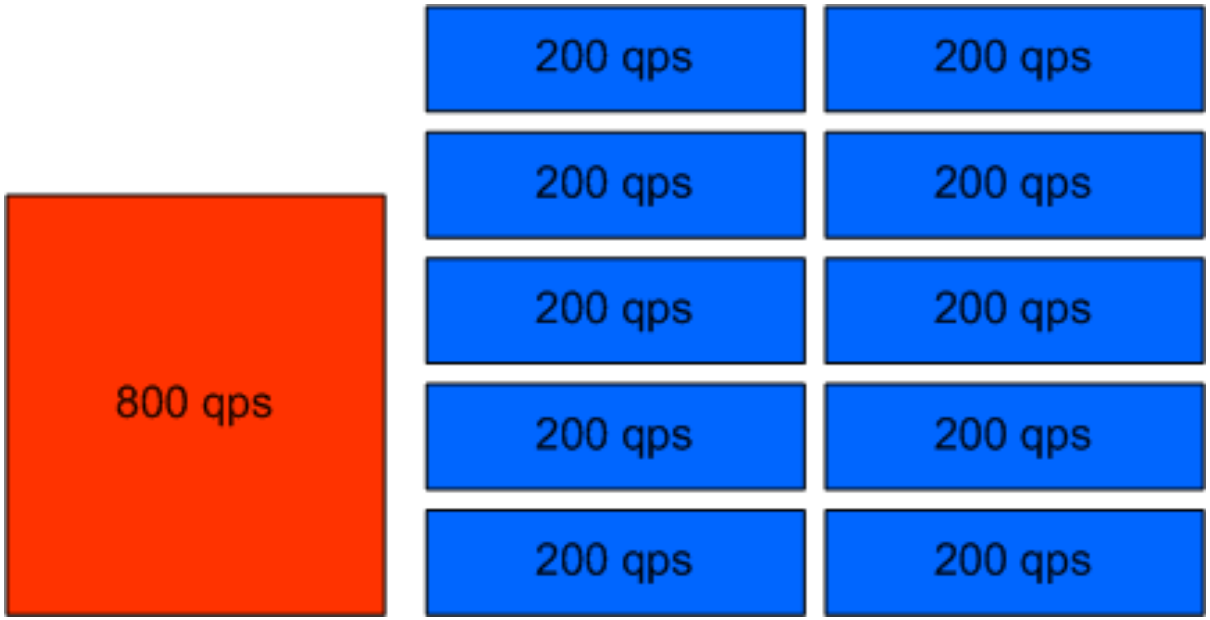
Master-Slave Replication



Master-Slave Replication



Master-Slave Replication



6.

Caching

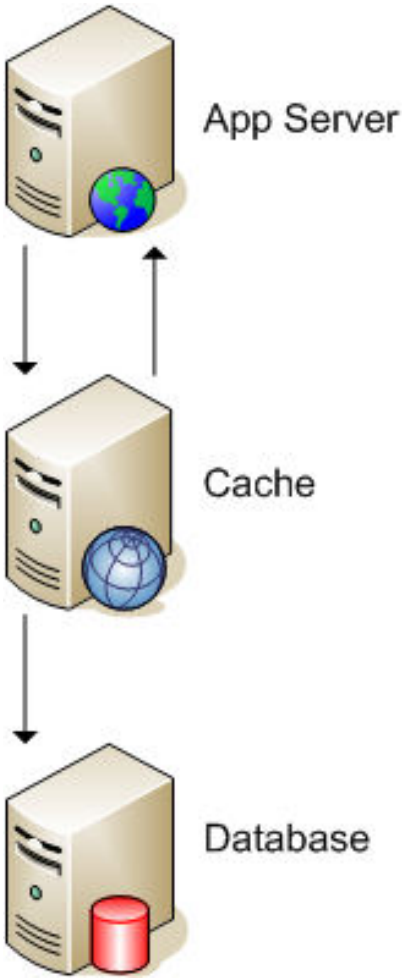
Caching

- Caching avoids needing to scale!
  - Or makes it cheaper
- Simple stuff
  - mod_perl / shared memory
    - Invalidation is hard
  - MySQL query cache
    - Bad performance (in most cases)

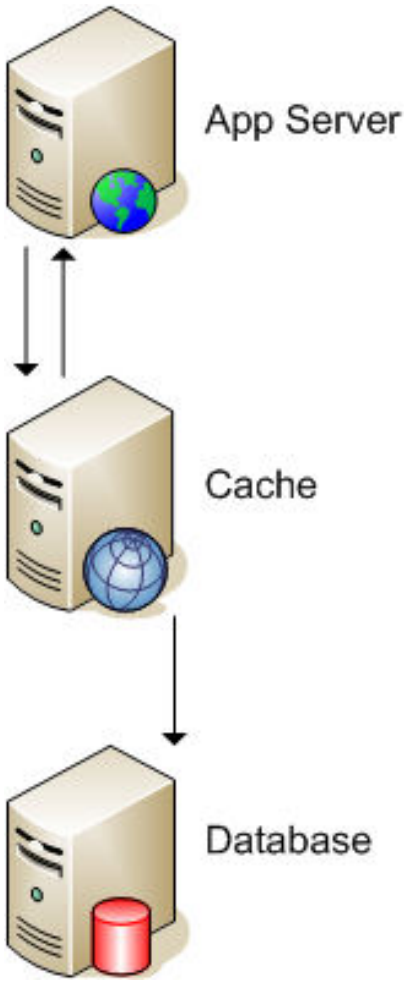
Caching

- Getting more complicated…
  - Write-through cache
  - Write-back cache
  - Sideline cache

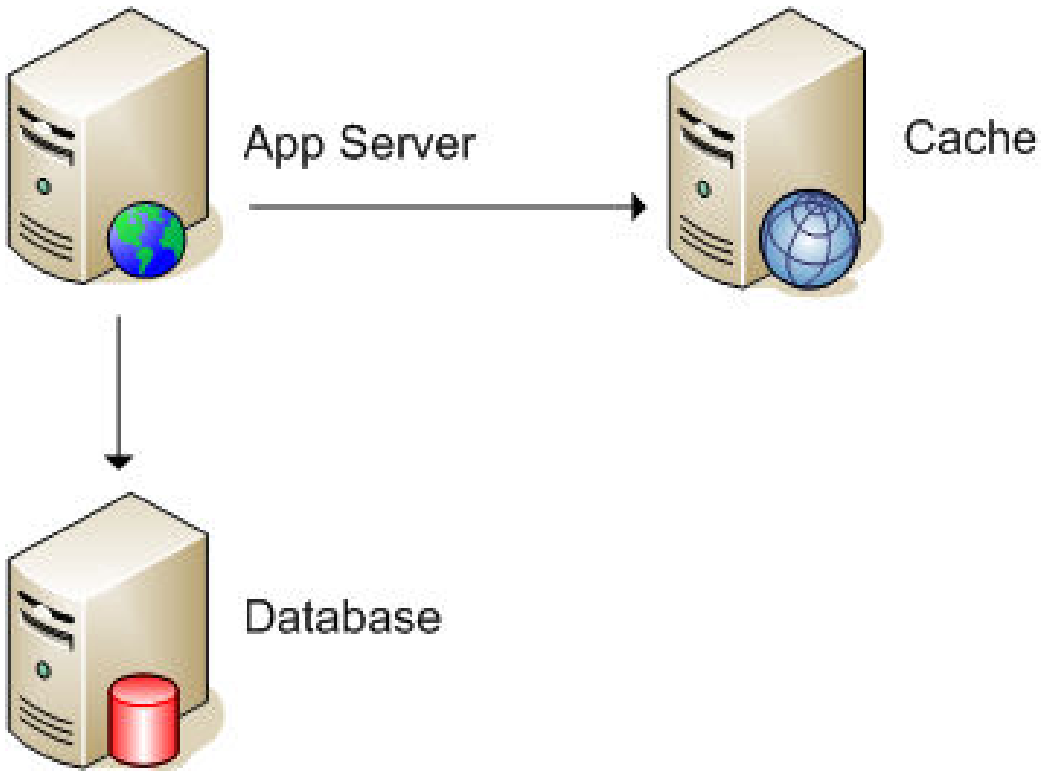
Write-through cache



Write-back cache



Sideline cache



Sideline cache

- Easy to implement
  - Just add app logic
- Need to manually invalidate cache
  - Well designed code makes it easy
- Memcached
  - From Danga (LiveJournal)
  - http://www.danga.com/memcached/

Memcache schemes

- Layer 4
  - Good: Cache can be local on a machine
  - Bad: Invalidation gets more expensive with node count
  - Bad: Cache space wasted by duplicate objects

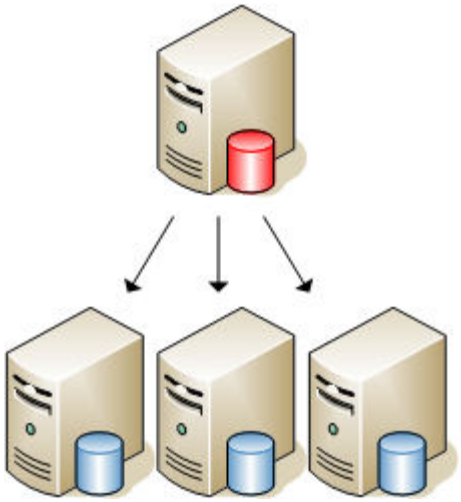
Memcache schemes

- Layer 7
  - Good: No wasted space
  - Good: linearly scaling invalidation
  - Bad: Multiple, remote connections
    - Can be avoided with a proxy layer
      - Gets more complicated
        - » Last indentation level!

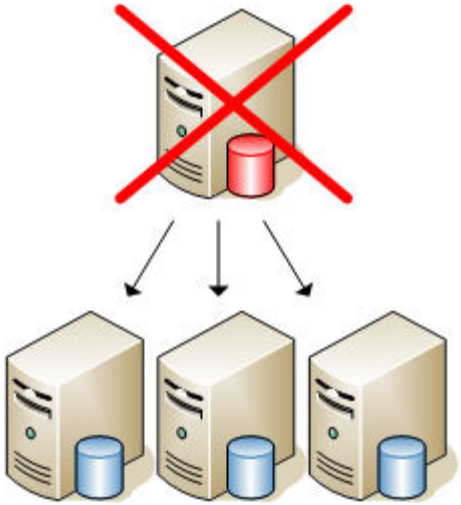
7.

HA Data

But what about HA?



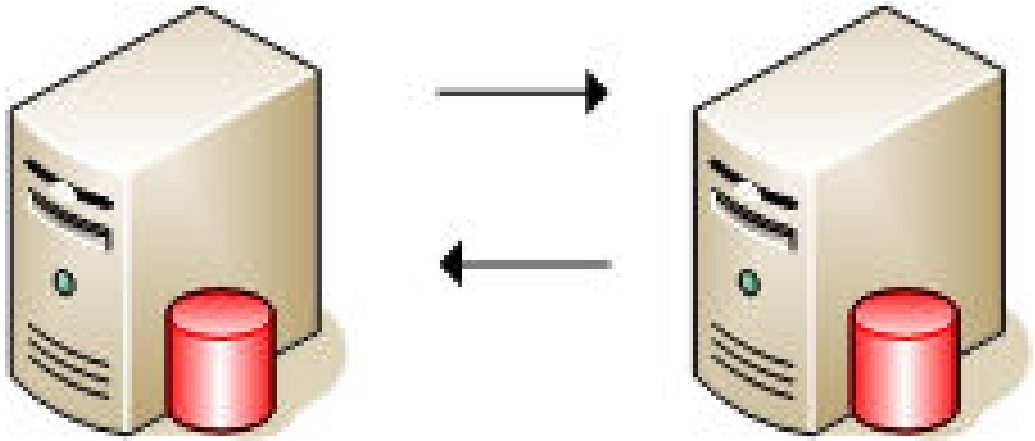
But what about HA?



SPOF!

- The key to HA is avoiding SPOFs
  - Identify
  - Eliminate
- Some stuff is hard to solve
  - Fix it further up the tree
    - Dual DCs solves Router/Switch SPOF

Master-Master



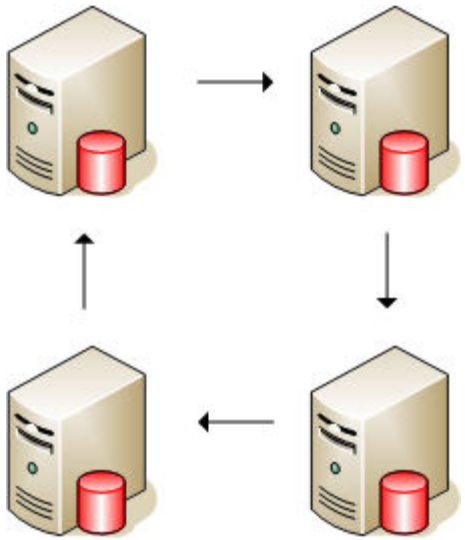
Master-Master

- Either hot/warm or hot/hot
- Writes can go to either
  - But avoid collisions
  - No auto-inc columns for hot/hot
    - Bad for hot/warm too
    - Unless you have MySQL 5
      - But you can't rely on the ordering!
  - Design schema/access to avoid collisions
    - Hashing users to servers

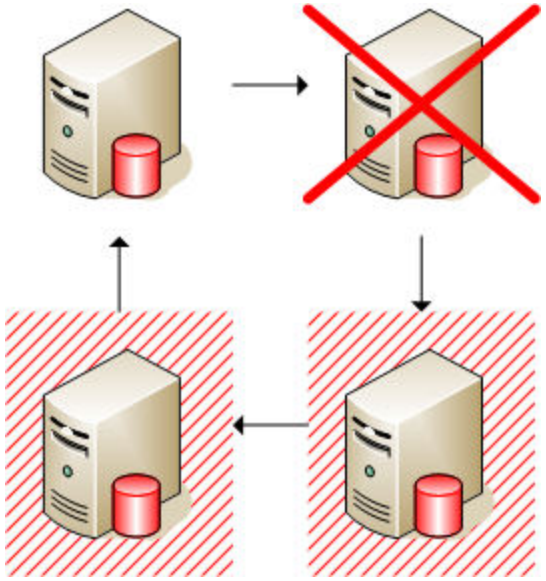
Rings

- Master-master is just a small ring
  - With 2 nodes
- Bigger rings are possible
  - But not a mesh!
  - Each slave may only have a single master
  - Unless you build some kind of manual replication

Rings



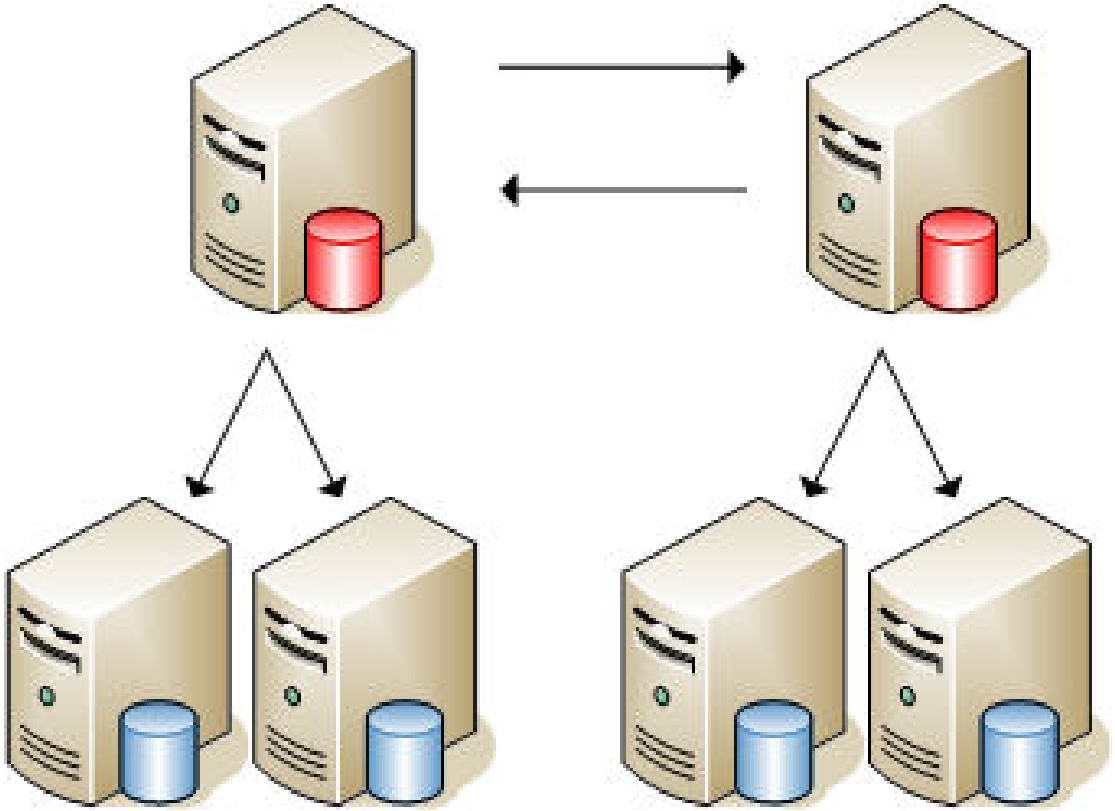
Rings



Dual trees

- Master-master is good for HA
  - But we can't scale out the reads (or writes!)
- We often need to combine the read scaling with HA
- We can simply combine the two models

Dual trees



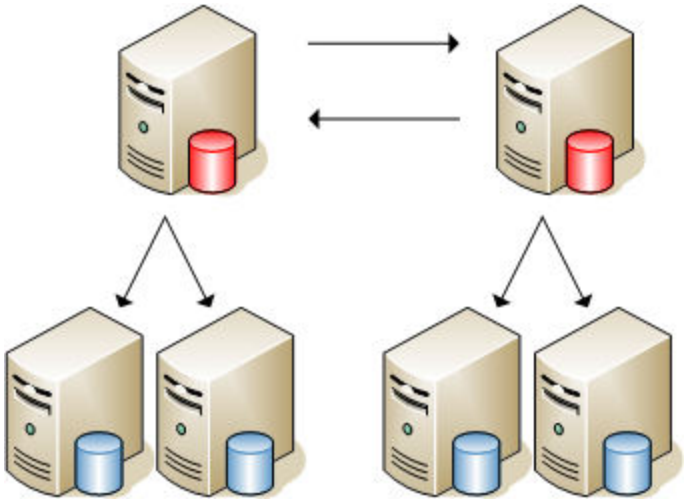
Cost models

- There's a problem here
  - We need to always have 200% capacity to avoid a SPOF
    - 400% for dual sites!
  - This costs too much
- Solution is straight forward
  - Make sure clusters are bigger than 2

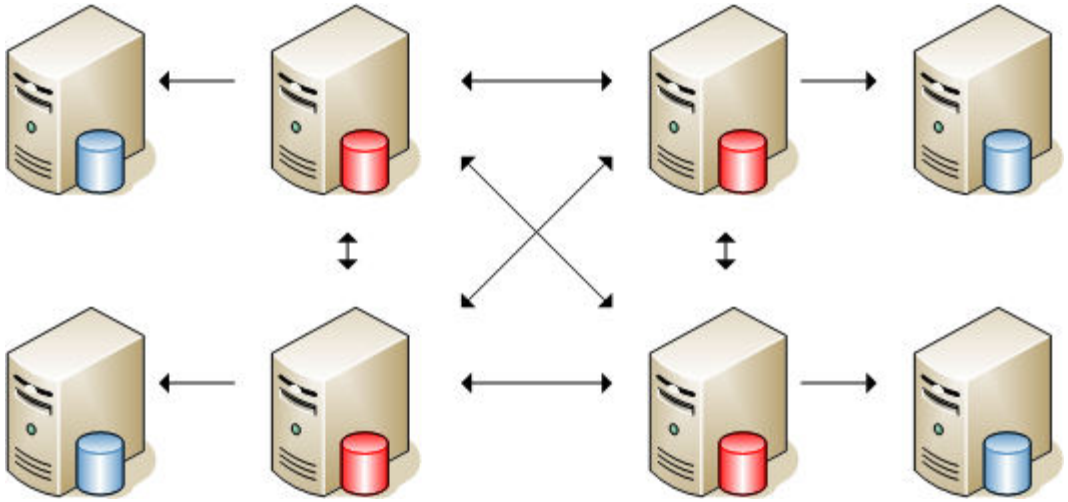
N+M

- N+M
  - N = nodes needed to run the system
  - M = nodes we can afford to lose
- Having M as big as N starts to suck
  - If we could make each node smaller, we can increase N while M stays constant
  - (We assume smaller nodes are cheaper)

1+1 = 200% hardware



3+1 = 133% hardware



Meshed masters

- Not possible with regular MySQL out-of-the-box today
- But there is hope!
  - NBD (MySQL Cluster) allows a mesh
  - Support for replication out to slaves in a coming version
    - RSN!

8.

Federation

Data federation

- At some point, you need more writes
  - This is tough
  - Each cluster of servers has limited write capacity
- Just add more clusters!

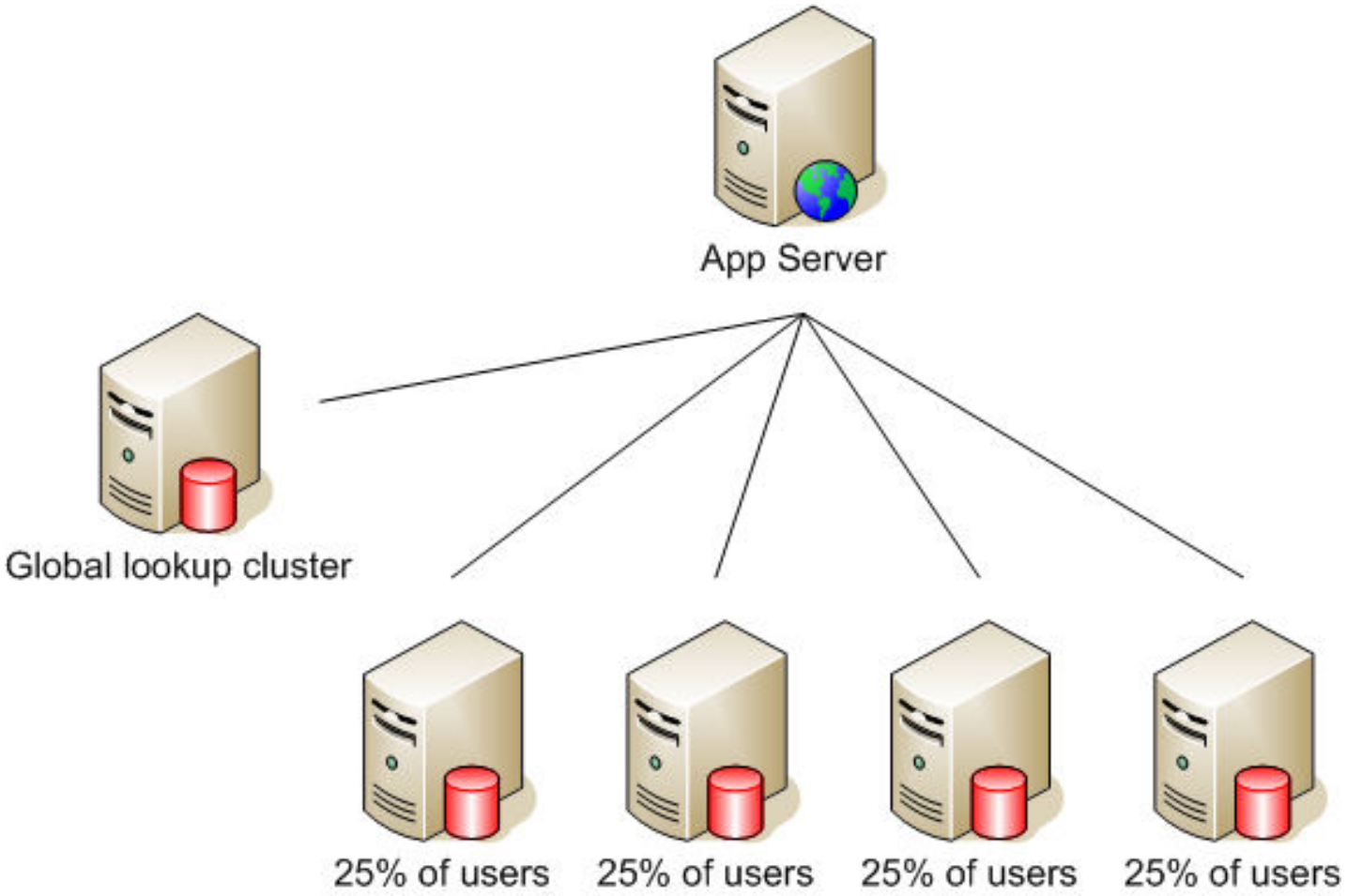
Simple things first

- Vertical partitioning
  - Divide tables into sets that never get joined
  - Split these sets onto different server clusters
  - Voila!
- Logical limits
  - When you run out of non-joining groups
  - When a single table grows too large

Data federation

- Split up large tables, organized by some primary object
  - Usually users
- Put all of a user's data on one 'cluster'
  - Or shard, or cell
- Have one central cluster for lookups

Data federation



Data federation

- Need more capacity?
  - Just add shards!
  - Don't assign to shards based on user_id!
- For resource leveling as time goes on, we want to be able to move objects between shards
  - Maybe – not everyone does this
  - 'Lockable' objects

The wordpress.com approach

- Hash users into one of *n* buckets
  - Where *n* is a power of 2
- Put all the buckets on one server
- When you run out of capacity, split the buckets across two servers
- Then you run out of capacity, split the buckets across four servers
- Etc

Data federation

- Heterogeneous hardware is fine
  - Just give a larger/smaller proportion of objects depending on hardware
- Bigger/faster hardware for paying users
  - A common approach
  - Can also allocate faster app servers via magic cookies at the LB

Downsides

- Need to keep stuff in the right place
- App logic gets more complicated
- More clusters to manage
  - Backups, etc
- More database connections needed per page
  - Proxy can solve this, but complicated
- The dual table issue
  - Avoid walking the shards!

Bottom line

Data federation is how large applications are scaled

Bottom line

- It's hard, but not impossible
- Good software design makes it easier
  - Abstraction!
- Master-master pairs for shards give us HA
- Master-master trees work for central cluster (many reads, few writes)

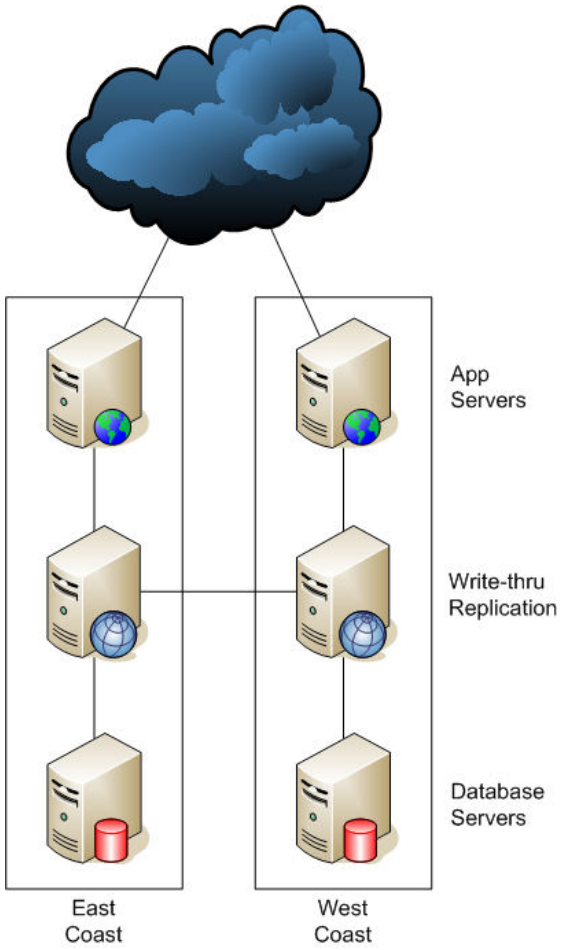
9.

Multi-site HA

Multiple Datacenters

- Having multiple datacenters is hard
  - Not just with MySQL
- Hot/warm with MySQL slaved setup
  - But manual (reconfig on failure)
- Hot/hot with master-master
  - But dangerous (each site has a SPOF)
- Hot/hot with sync/async manual replication
  - But tough (big engineering task)

Multiple Datacenters



GSLB

- Multiple sites need to be balanced
  - Global Server Load Balancing
- Easiest are AkaDNS-like services
  - Performance rotations
  - Balance rotations

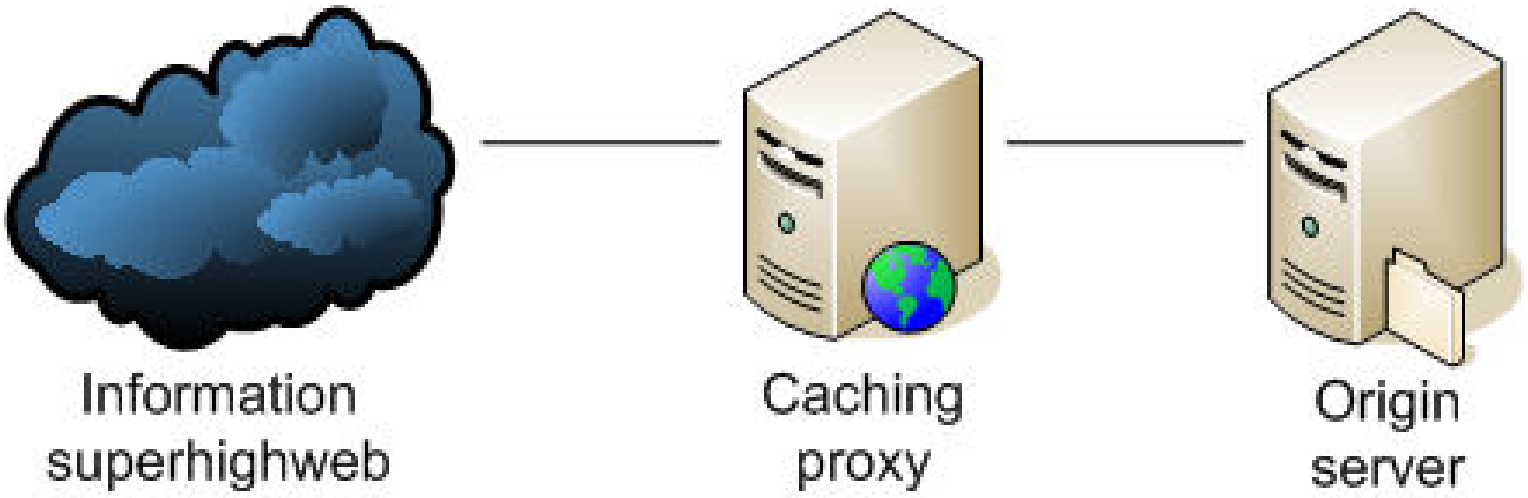
10.

Serving Files

Serving lots of files

- Serving lots of files is not too tough
  - Just buy lots of machines and load balance!
- We're IO bound – need more spindles!
  - But keeping many copies of data in sync is hard
  - And sometimes we have other per-request overhead (like auth)

Reverse proxy



Reverse proxy

- Serving out of memory is fast!
  - And our caching proxies can have disks too
  - Fast or otherwise
    - More spindles is better
- We can parallelize it!
  - 50 cache servers gives us 50 times the serving rate of the origin server
  - Assuming the working set is small enough to fit in memory in the cache cluster

Invalidation

- Dealing with invalidation is tricky
- We can prod the cache servers directly to clear stuff out
  - Scales badly – need to clear asset from every server – doesn't work well for 100 caches

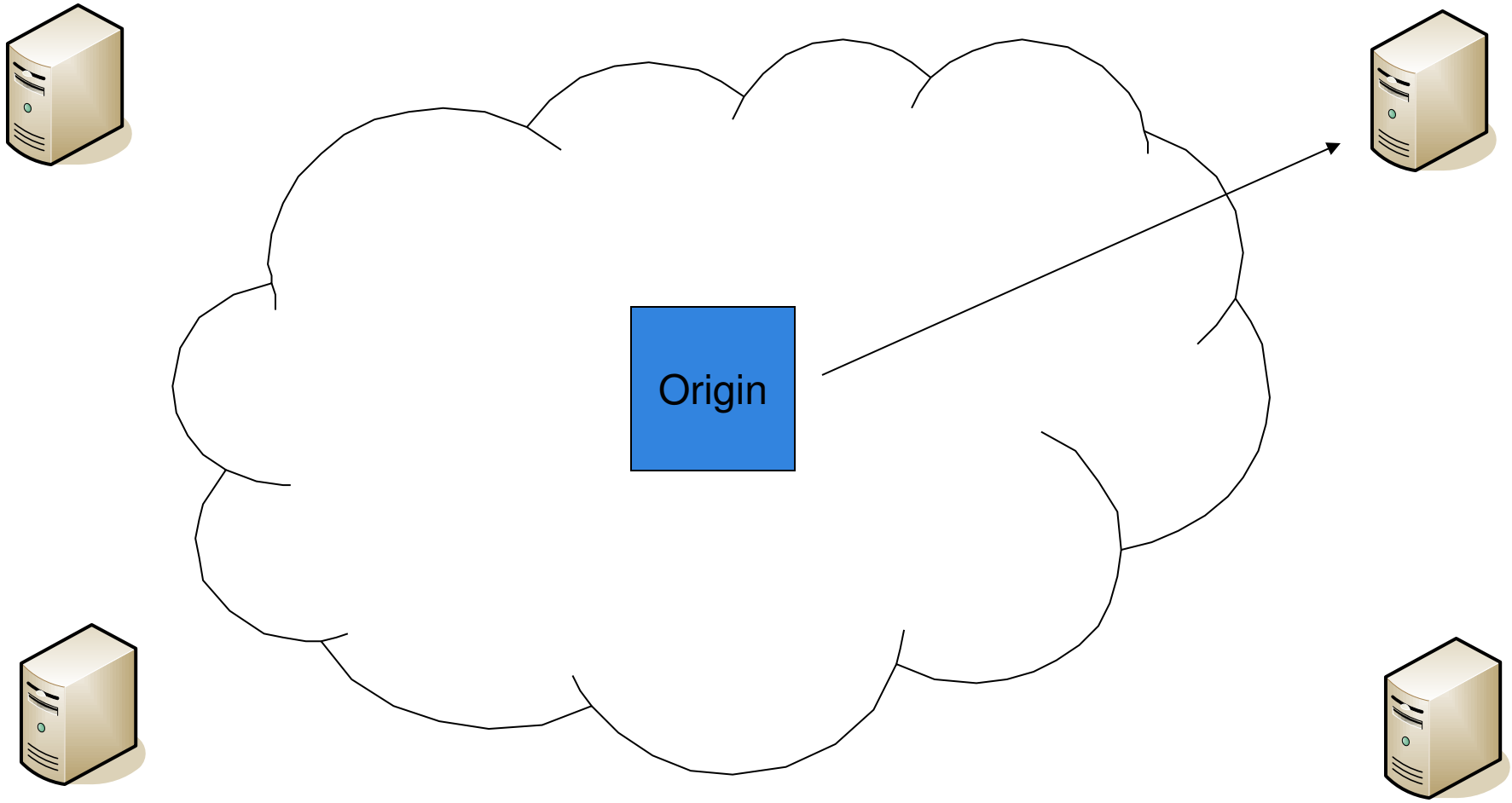
Invalidation

- We can change the URLs of modified resources
  - And let the old ones drop out cache naturally
  - Or poke them out, for sensitive data
- Good approach!
  - Avoids browser cache staleness
  - Hello Akamai (and other CDNs)
  - Read more:
    - http://www.thinkvitamin.com/features/webapps/serving-javascript-fast

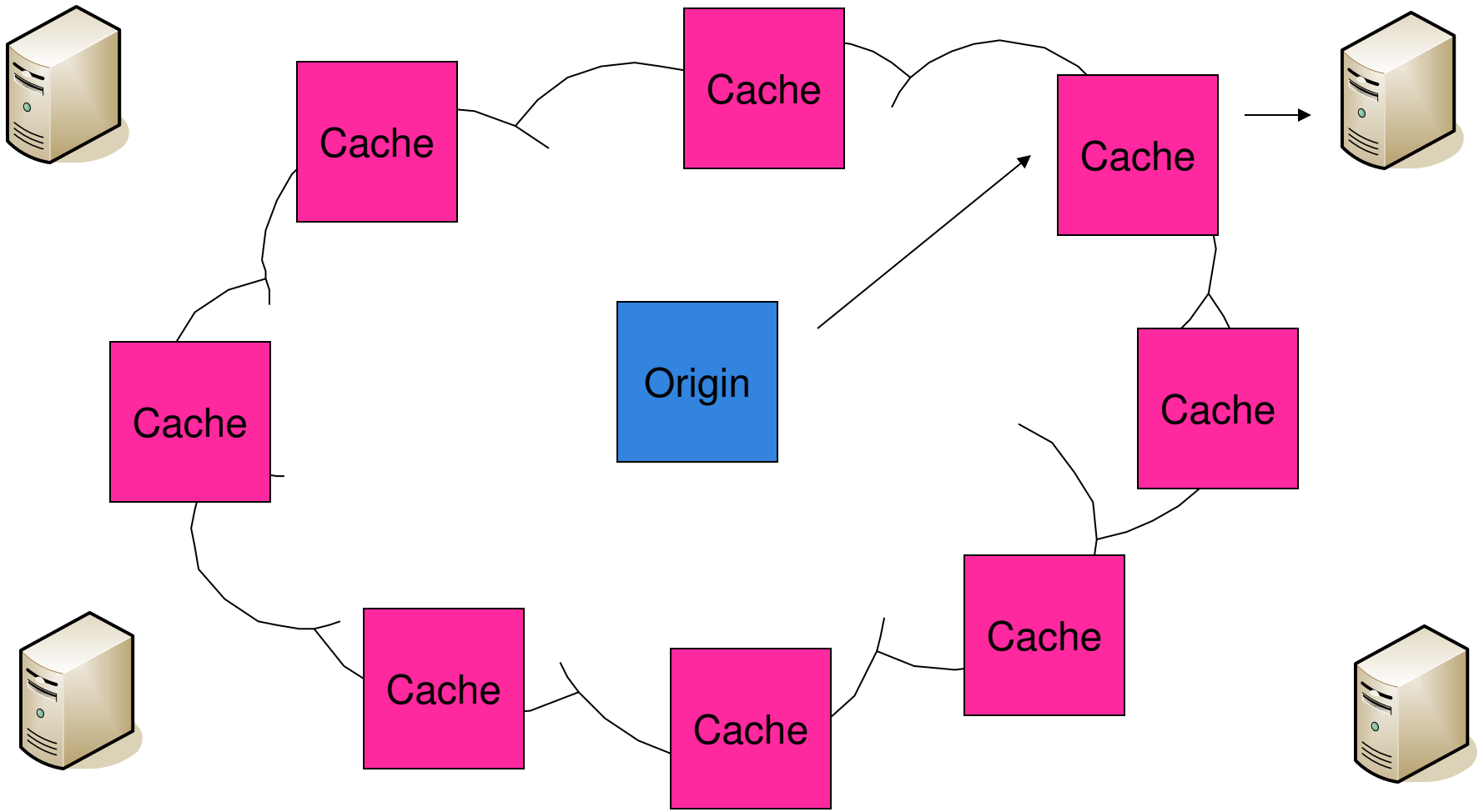
CDN – Content Delivery Network

- Akamai, Savvis, Mirror Image Internet, etc
- Caches operated by other people
  - Already in-place
  - In lots of places
- GSLB/DNS balancing

Edge networks



Edge networks



CDN Models

- Simple model
  - You push content to them, they serve it
- Reverse proxy model
  - You publish content on an origin, they proxy and cache it

CDN Invalidation

- You don't control the caches
  - Just like those awful ISP ones
- Once something is cached by a CDN, assume it can never change
  - Nothing can be deleted
  - Nothing can be modified

Versioning

- When you start to cache things, you need to care about versioning
  - Invalidation & Expiry
  - Naming & Sync

Cache Invalidation

- If you control the caches, invalidation is possible
- But remember ISP and client caches
- Remove deleted content explicitly
  - Avoid users finding old content
  - Save cache space

Cache versioning

- Simple rule of thumb:
  - If an item is modified, change its name (URL)
- This can be independent of the file system!

Virtual versioning

- Database indicates version 3 of file
- Web app writes version number into URL
- Request comes through cache and is cached with the versioned URL
- mod_rewrite converts versioned URL to path

Reverse proxy

- Choices
  - L7 load balancer & Squid
    - http://www.squid-cache.org/
  - mod_proxy & mod_cache
    - http://www.apache.org/
  - Perlbal and Memcache?
    - http://www.danga.com/

More reverse proxy

- HA proxy
- Squid & CARP
- Varnish

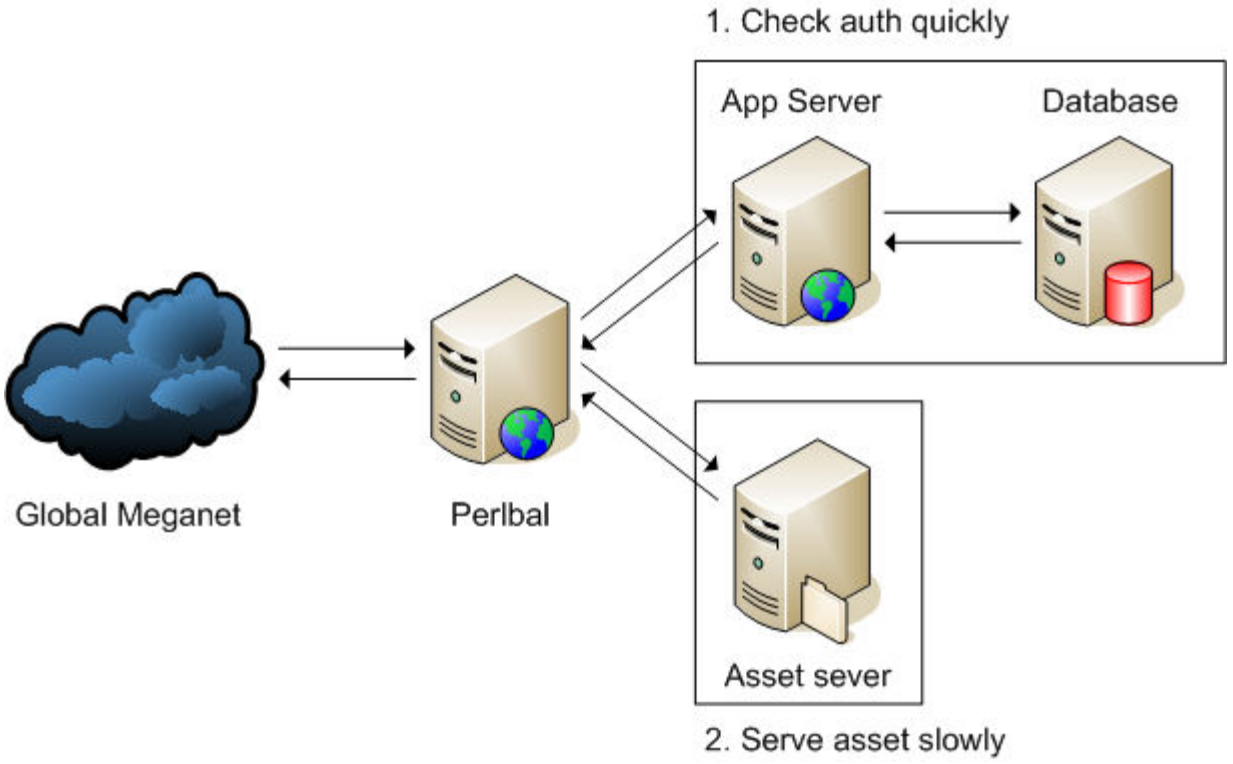
High overhead serving

- What if you need to authenticate your asset serving?
  - Private photos
  - Private data
  - Subscriber-only files
- Two main approaches
  - Proxies w/ tokens
  - Path translation

Perlbal Re-proxying

- Perlbal can do redirection magic
  - Client sends request to Perlbal
  - Perlbal plugin verifies user credentials
    - token, cookies, whatever
    - tokens avoid data-store access
  - Perlbal goes to pick up the file from elsewhere
  - Transparent to user

Perlbal Re-proxying



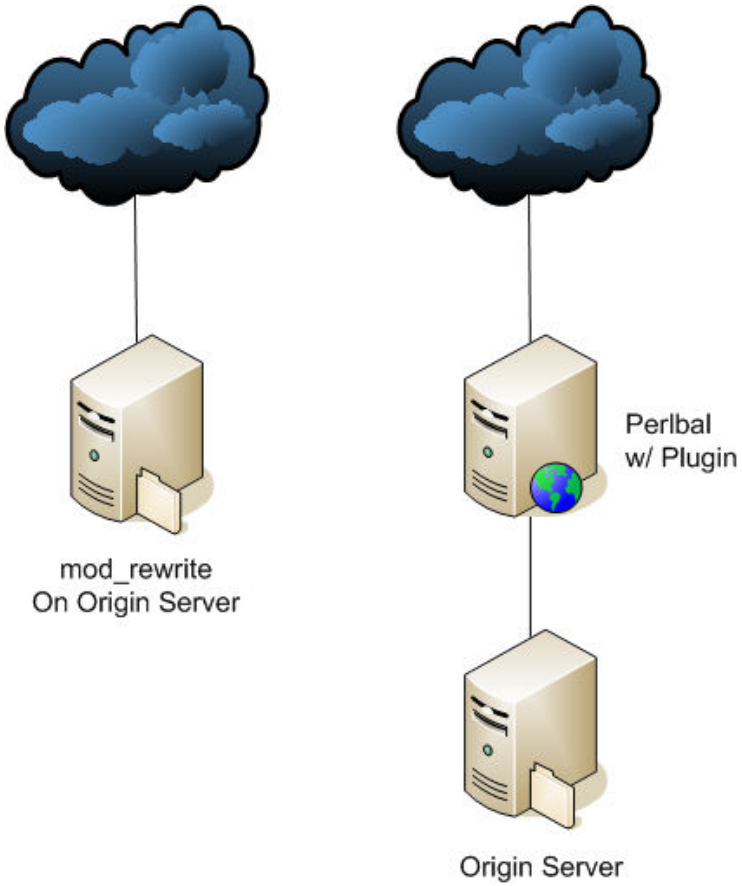
Perlbal Re-proxying

- Doesn’t keep database around while serving
- Doesn’t keep app server around while serving
- User doesn’t find out how to access asset directly

Permission URLs

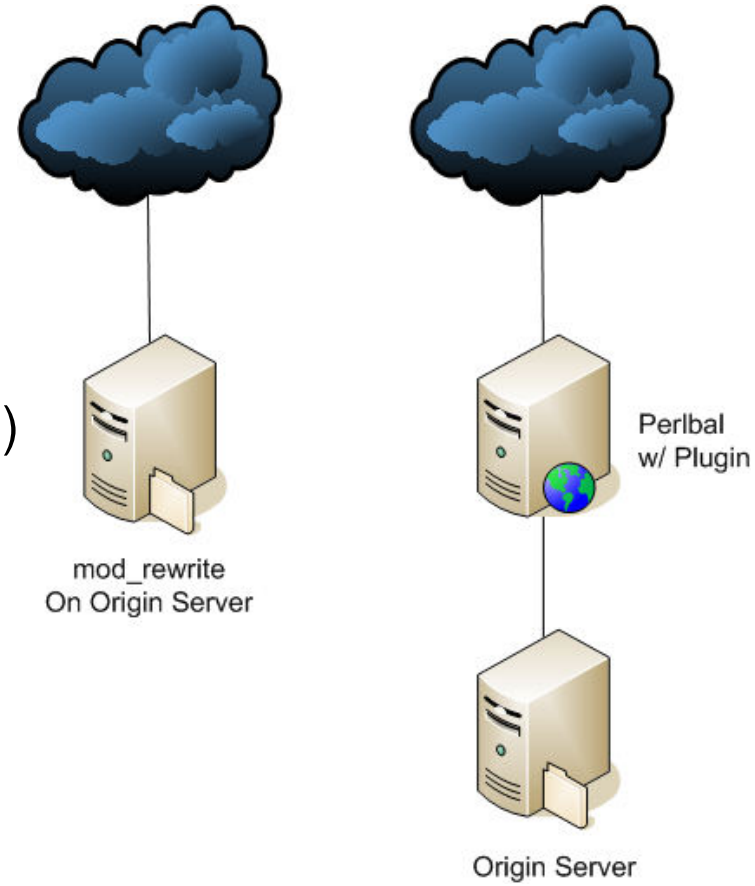
- But why bother!?
- If we bake the auth into the URL then it saves the auth step
- We can do the auth on the web app servers when creating HTML
- Just need some magic to translate to paths
- We don’t want paths to be guessable

Permission URLs



Permission URLs

(or mod_perl)



Permission URLs

- Downsides
  - URL gives permission for life
  - Unless you bake in tokens
    - Tokens tend to be non-expirable
      - We don't want to track every token
        - » Too much overhead
    - But can still expire
      - But you choose at time of issue, not later
- Upsides
  - It works
  - Scales very nicely

11.

Storing Files

Storing lots of files

- Storing files is easy!
  - Get a big disk
  - Get a bigger disk
  - Uh oh!
- Horizontal scaling is the key
  - Again

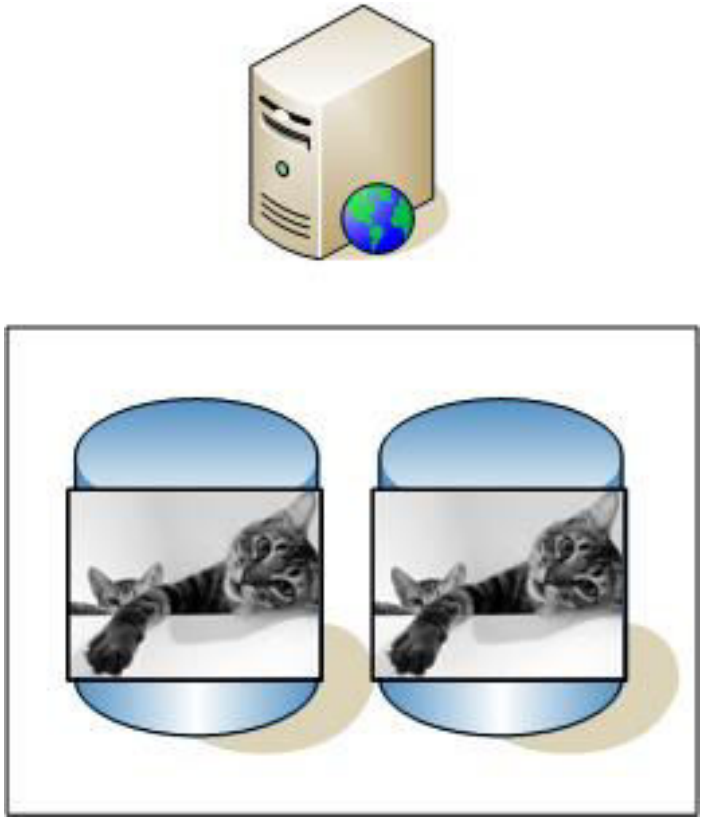
Connecting to storage

- NFS
  - Stateful == Sucks
  - Hard mounts vs Soft mounts, INTR
- SMB / CIFS / Samba
  - Turn off MSRPC & WINS (NetBOIS NS)
  - Stateful but degrades gracefully
- HTTP
  - Stateless == Yay!
  - Just use Apache

Multiple volumes

- Volumes are limited in total size
  - Except (in theory) under ZFS & others
- Sometimes we need multiple volumes for performance reasons
  - When using RAID with single/dual parity
- At some point, we need multiple volumes

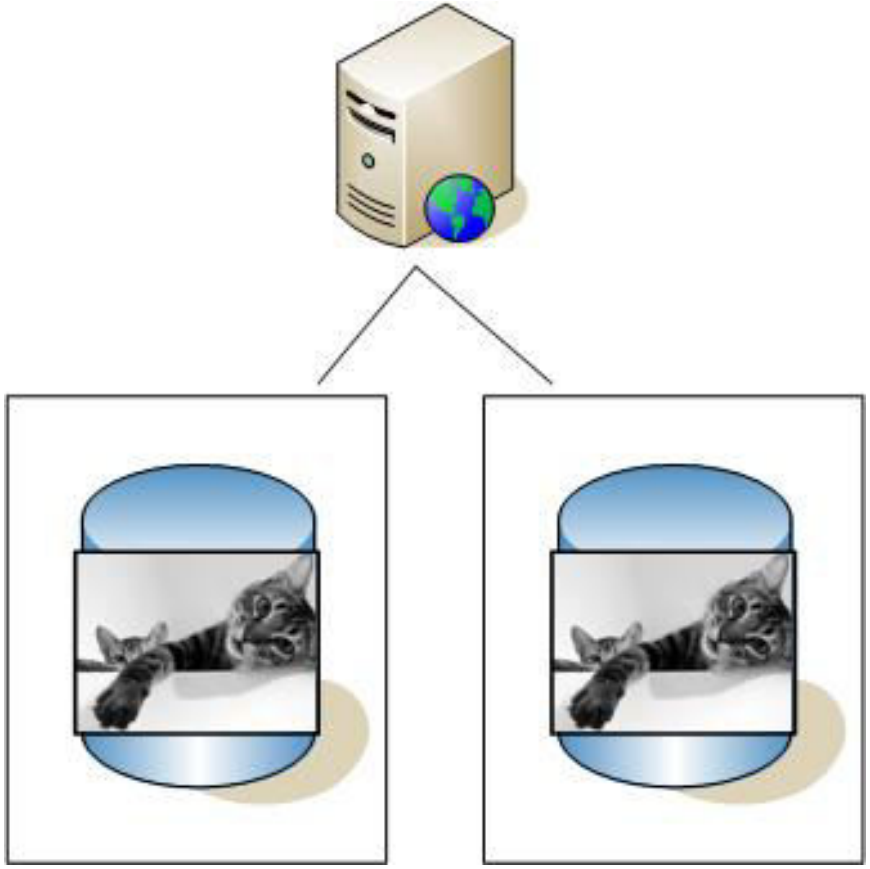
Multiple volumes



Multiple hosts

- Further down the road, a single host will be too small
- Total throughput of machine becomes an issue
- Even physical space can start to matter
- So we need to be able to use multiple hosts

Multiple hosts



HA Storage

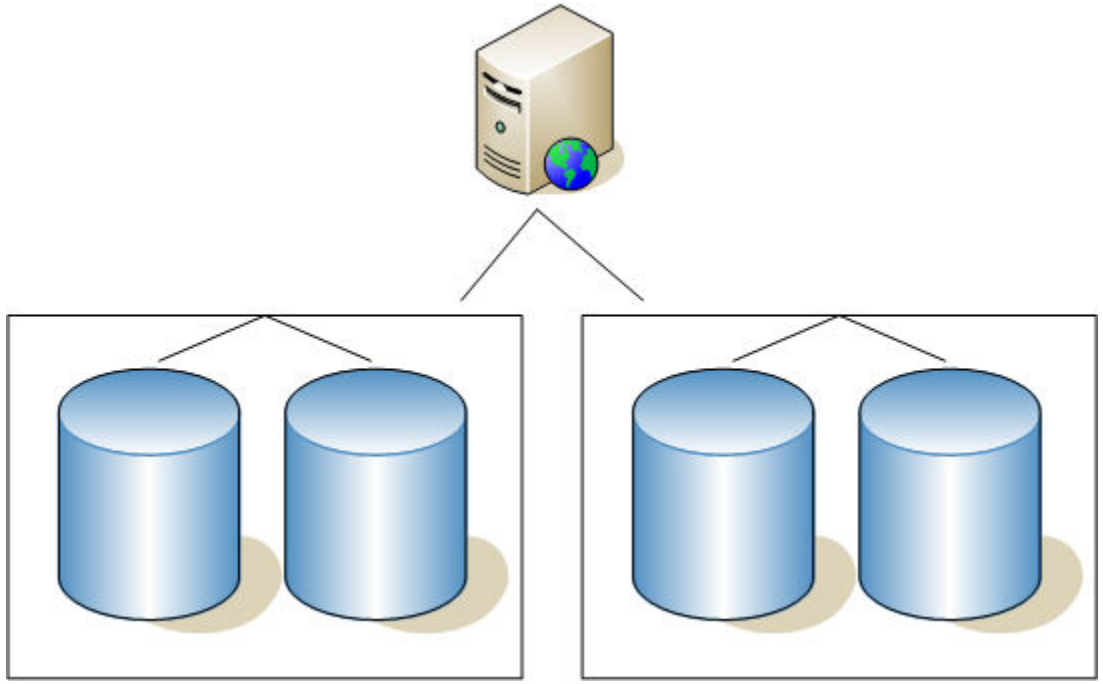
- HA is important for file assets too
  - We can back stuff up
  - But we tend to want hot redundancy
- RAID is good
  - RAID 5 is cheap, RAID 10 is fast

HA Storage

- But whole machines can fail
- So we stick assets on multiple machines

- In this case, we can ignore RAID
  - In failure case, we serve from alternative source
  - But need to weigh up the rebuild time and effort against the risk
  - Store more than 2 copies?

HA Storage



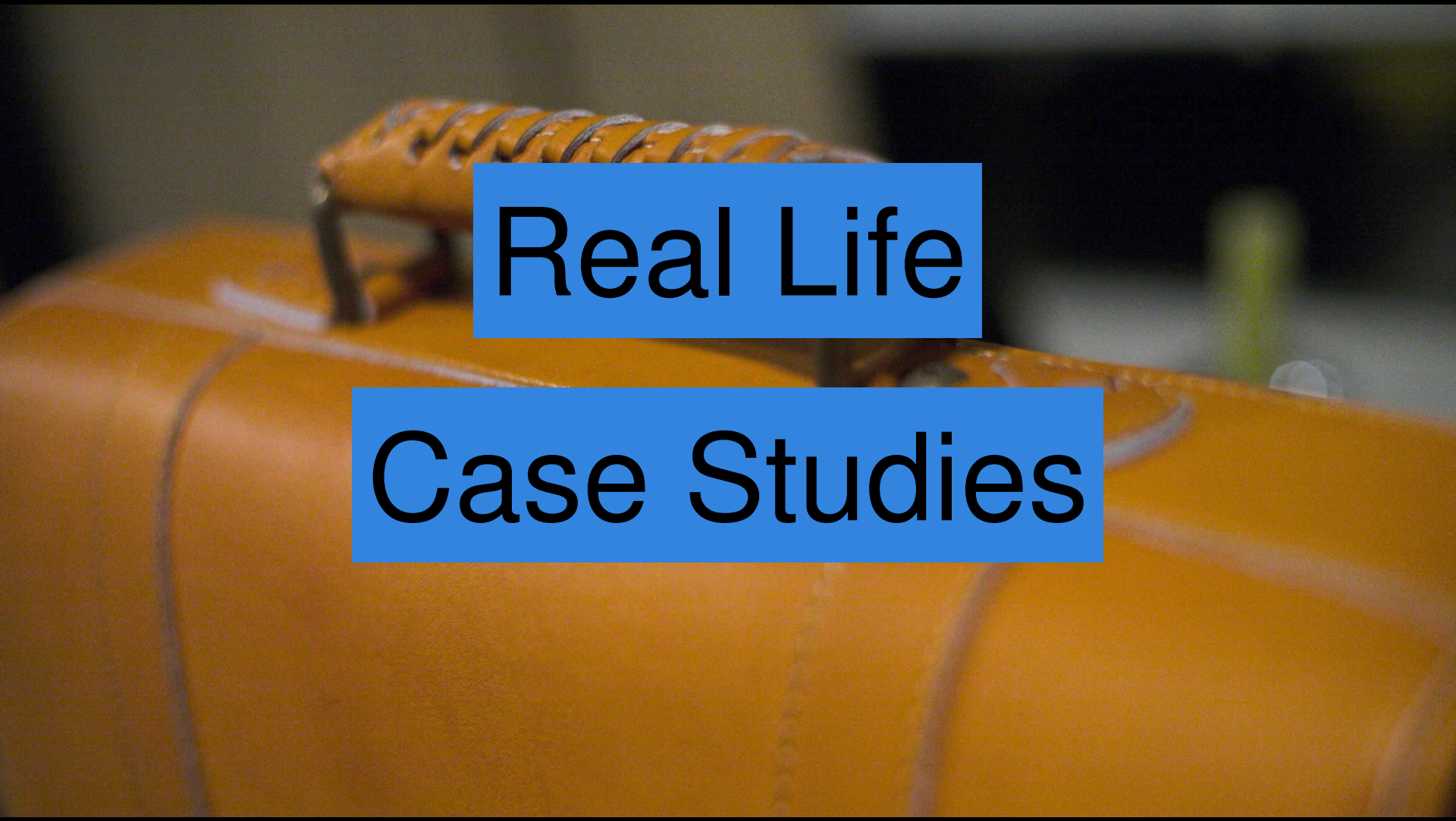
HA Storage

- But whole colos can fail
- So we stick assets in multiple colos

- Again, we can ignore RAID
  - Or even multi-machine per colo
  - But do we want to lose a whole colo because of single disk failure?
  - Redundancy is always a balance

Self repairing systems

- When something fails, repairing can be a pain
  - RAID rebuilds by itself, but machine replication doesn't
- The big appliances self heal
  - NetApp, StorEdge, etc
- So does MogileFS (reaper)


Real Life
Case Studies

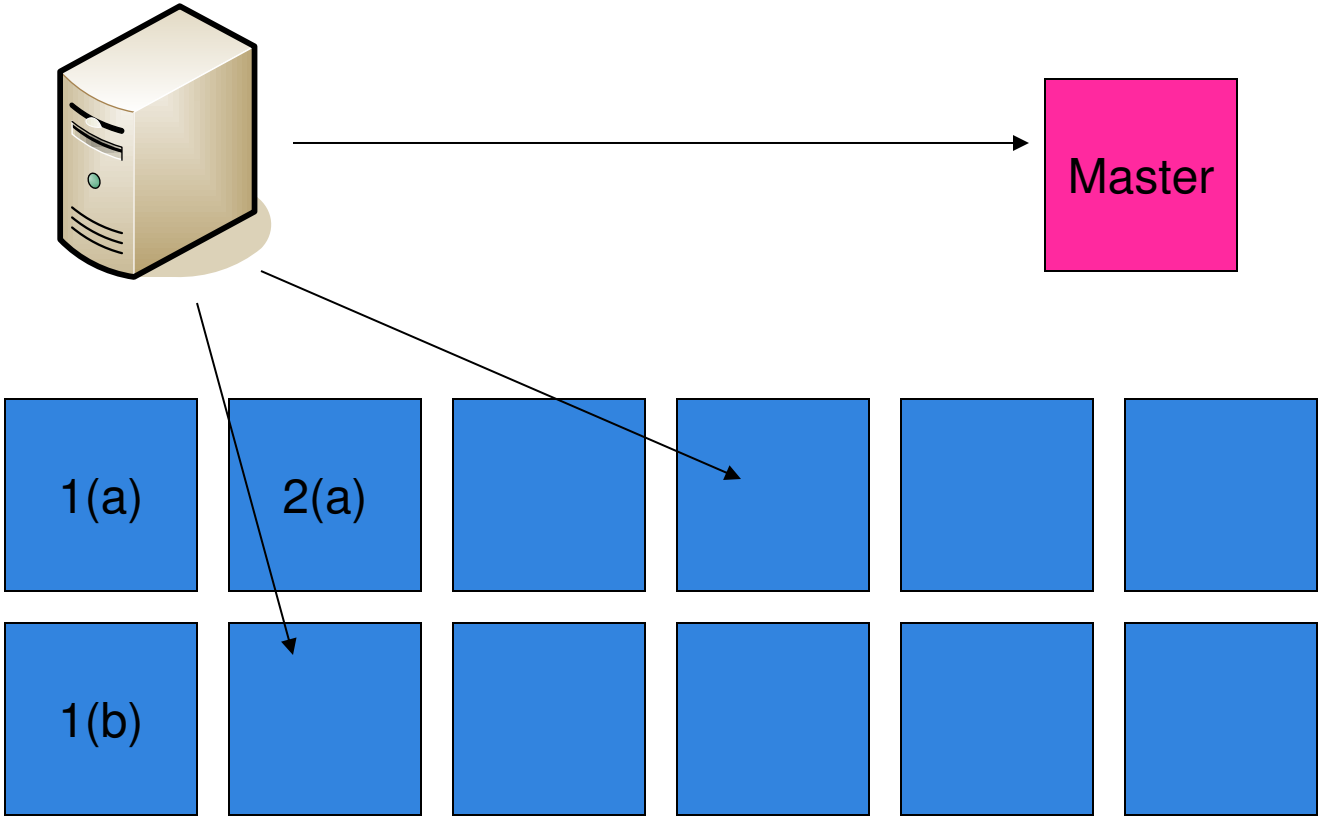
GFS – Google File System

- Developed by … Google
- Proprietary
- Everything we know about it is based on talks they've given
- Designed to store huge files for fast access

GFS – Google File System

- Single ‘Master’ node holds metadata
  - SPF – Shadow master allows warm swap
- Grid of ‘chunkservers’
  - 64bit filenames
  - 64 MB file chunks

GFS – Google File System



GFS – Google File System

- Client reads metadata from master then file parts from multiple chunkservers
- Designed for big files (>100MB)
- Master server allocates access leases
- Replication is automatic and self repairing
  - Synchronously for atomicity

GFS – Google File System

- Reading is fast (parallelizable)
  - But requires a lease
- Master server is required for all reads and writes

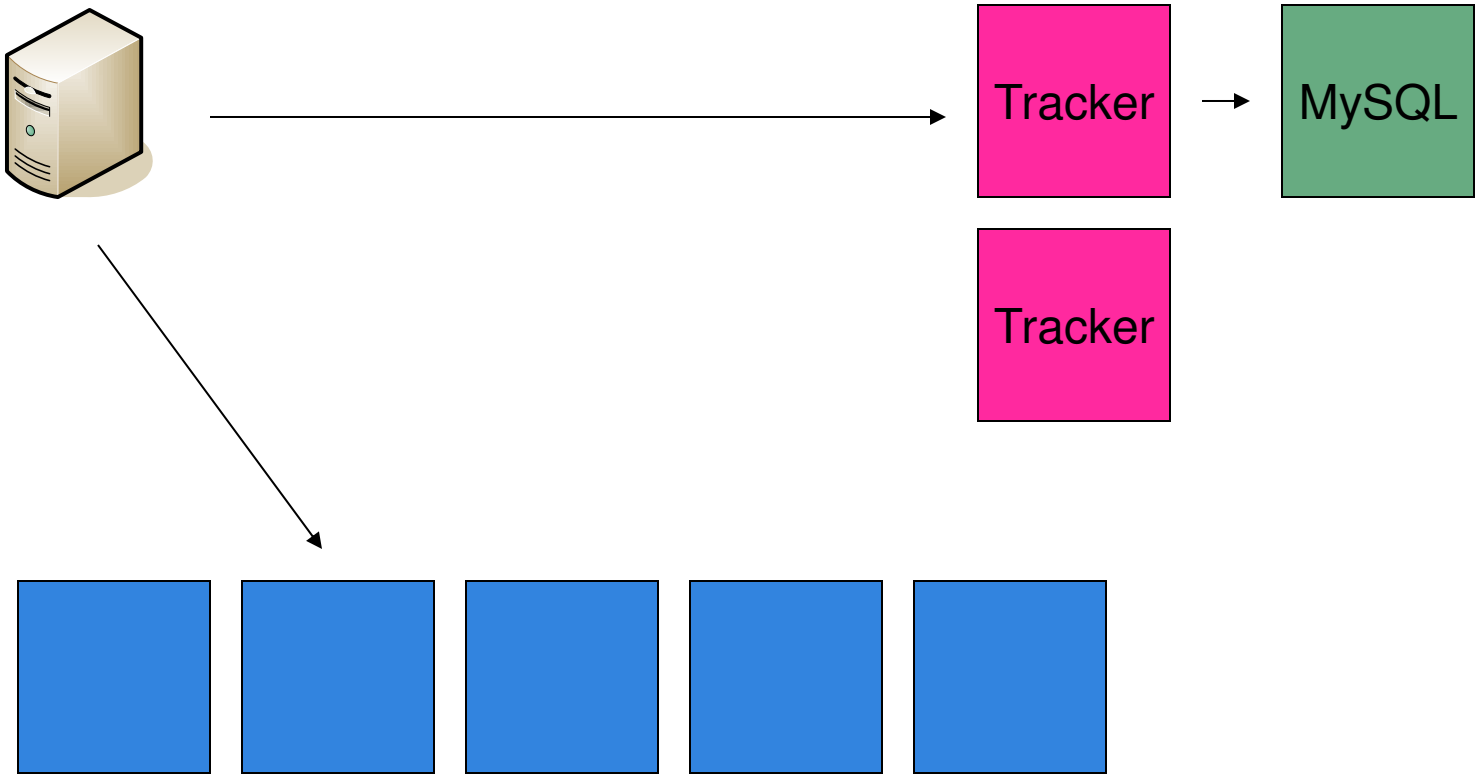
MogileFS – OMG Files

- Developed by Danga / SixApart
- Open source
- Designed for scalable web app storage

MogileFS – OMG Files

- Single metadata store (MySQL)
  - MySQL Cluster avoids SPF
- Multiple ‘tracker’ nodes locate files
- Multiple ‘storage’ nodes store files

MogileFS – OMG Files



MogileFS – OMG Files

- Replication of file ‘classes’ happens transparently
- Storage nodes are not mirrored – replication is piecemeal
- Reading and writing go through trackers, but are performed directly upon storage nodes

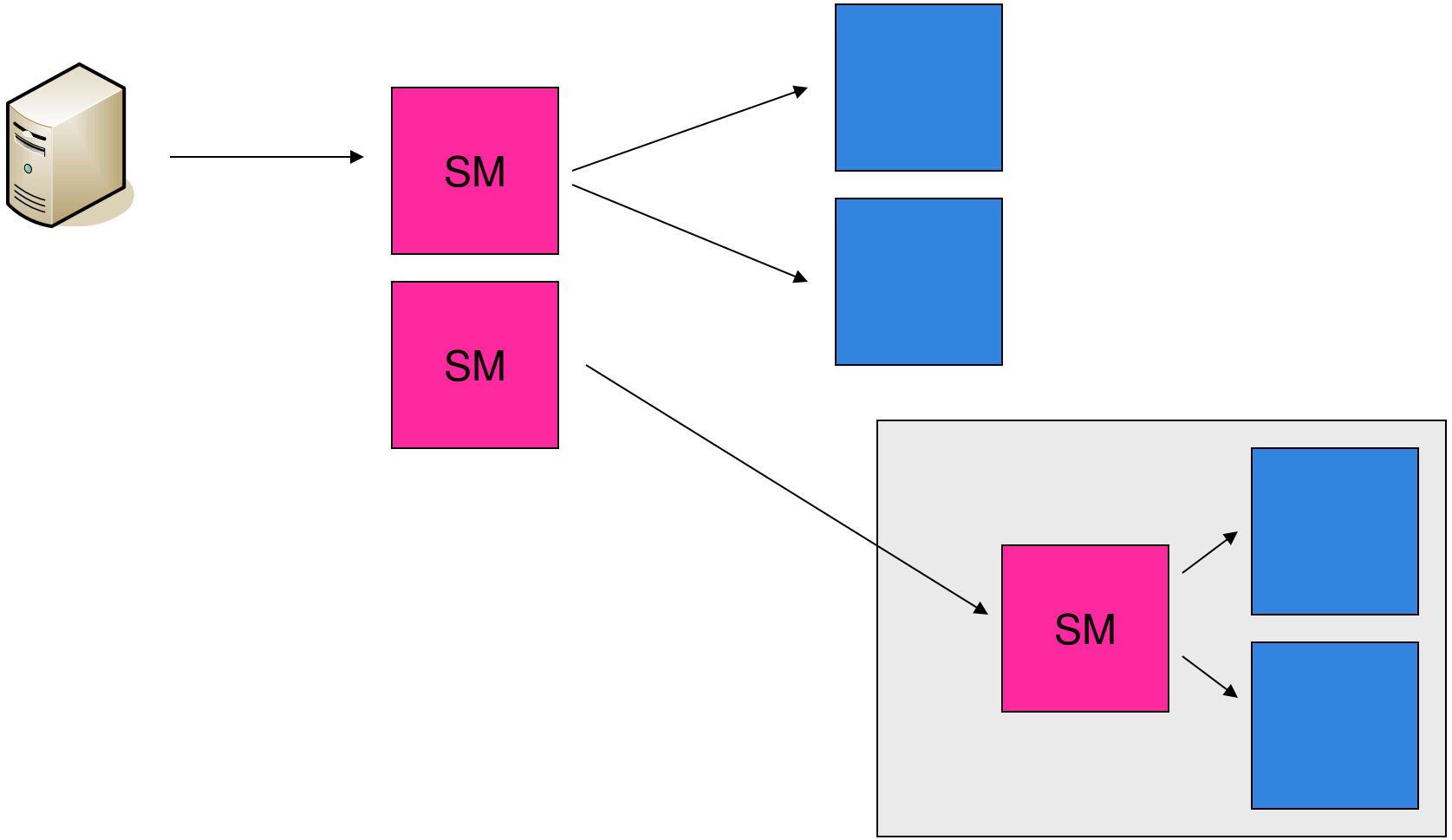
Flickr File System

- Developed by Flickr
- Proprietary
- Designed for very large scalable web app storage

Flickr File System

- No metadata store
  - Deal with it yourself
- Multiple ‘StorageMaster’ nodes
- Multiple storage nodes with virtual volumes

Flickr File System



Flickr File System

- Metadata stored by app
  - Just a virtual volume number
  - App chooses a path
- Virtual nodes are mirrored
  - Locally and remotely
- Reading is done directly from nodes

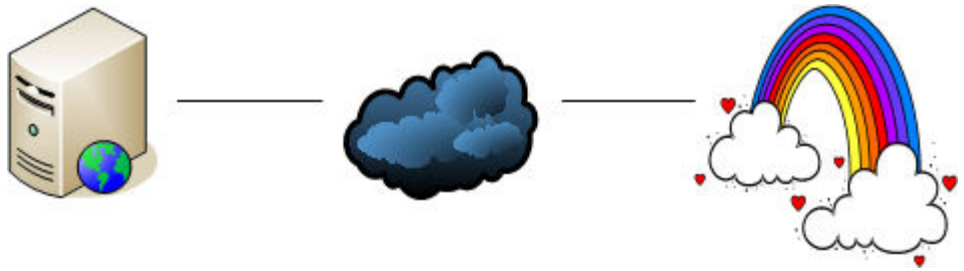
Flickr File System

- StorageMaster nodes only used for write operations
- Reading and writing can scale separately

Amazon S3

- A big disk in the sky
- Multiple ‘buckets’
- Files have user-defined keys
- Data + metadata

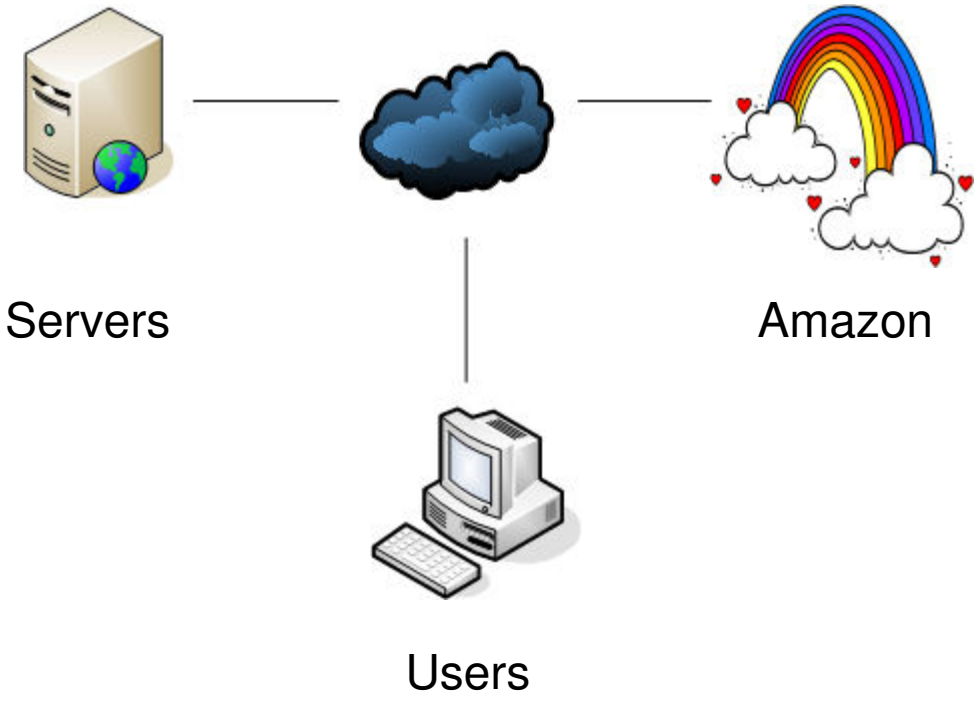
Amazon S3



Servers

Amazon

Amazon S3

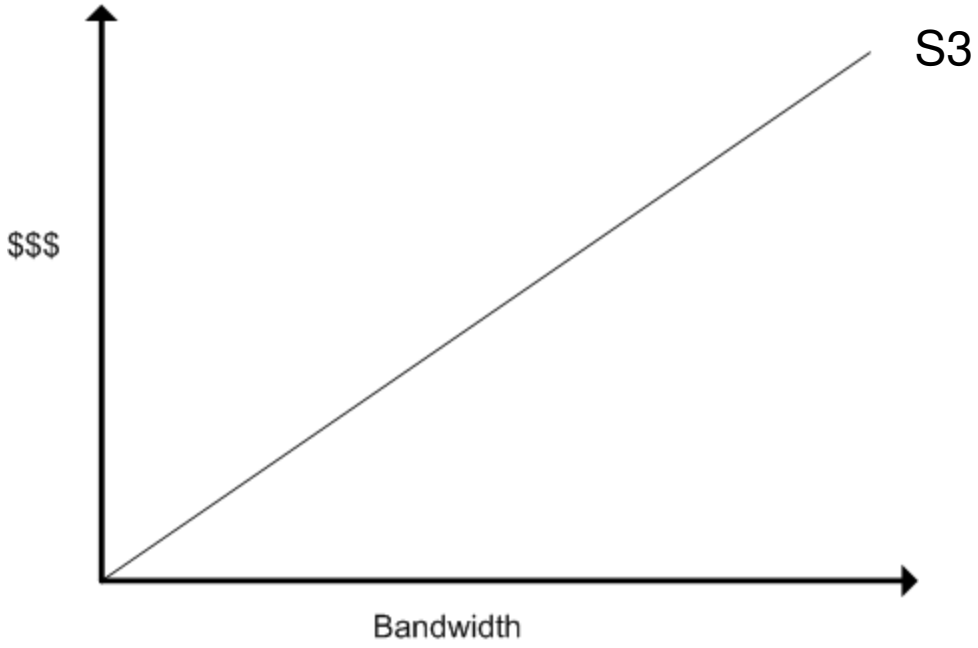


The cost

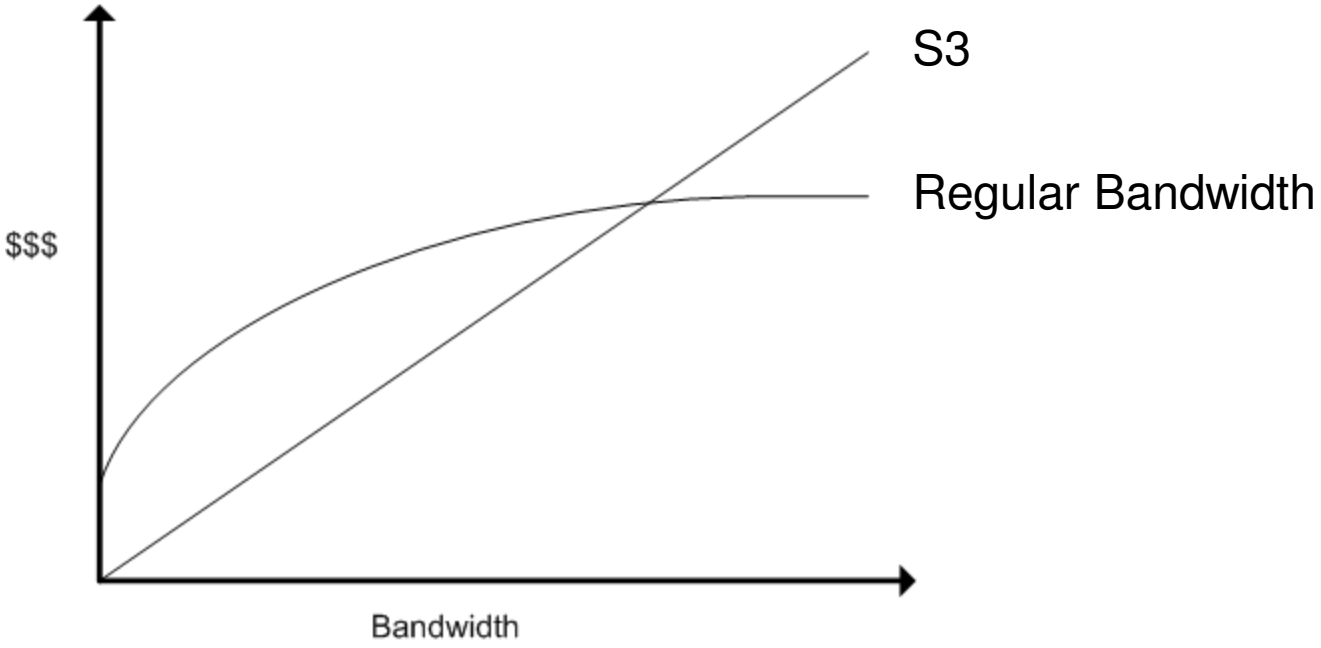
- Fixed price, by the GB

- Store: $0.15 per GB per month
- Serve: $0.20 per GB

The cost



The cost



End costs

- ~$2k to store 1TB for a year
- ~$63 a month for 1Mb
- ~$65k a month for 1Gb

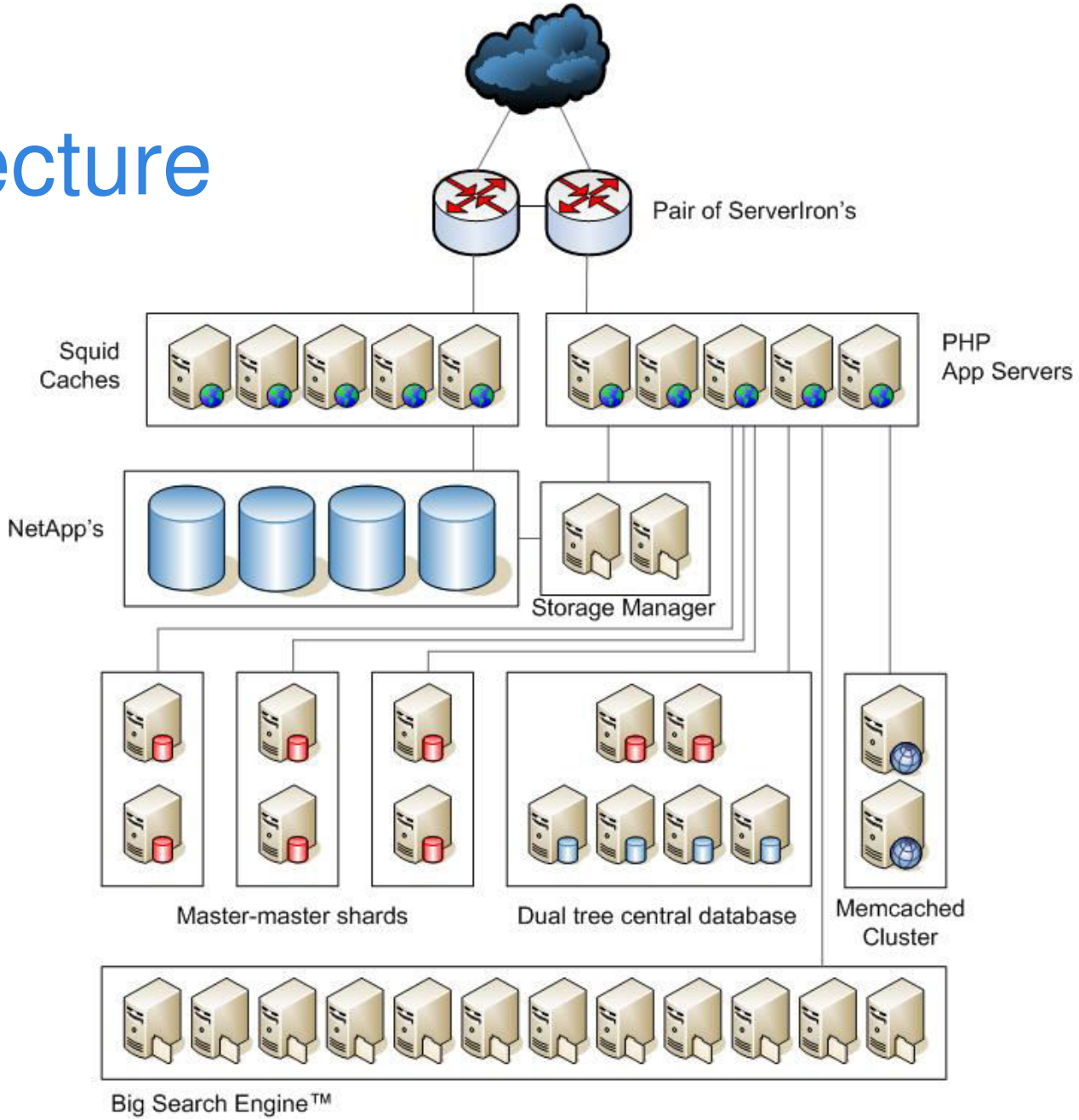
12.

Field Work

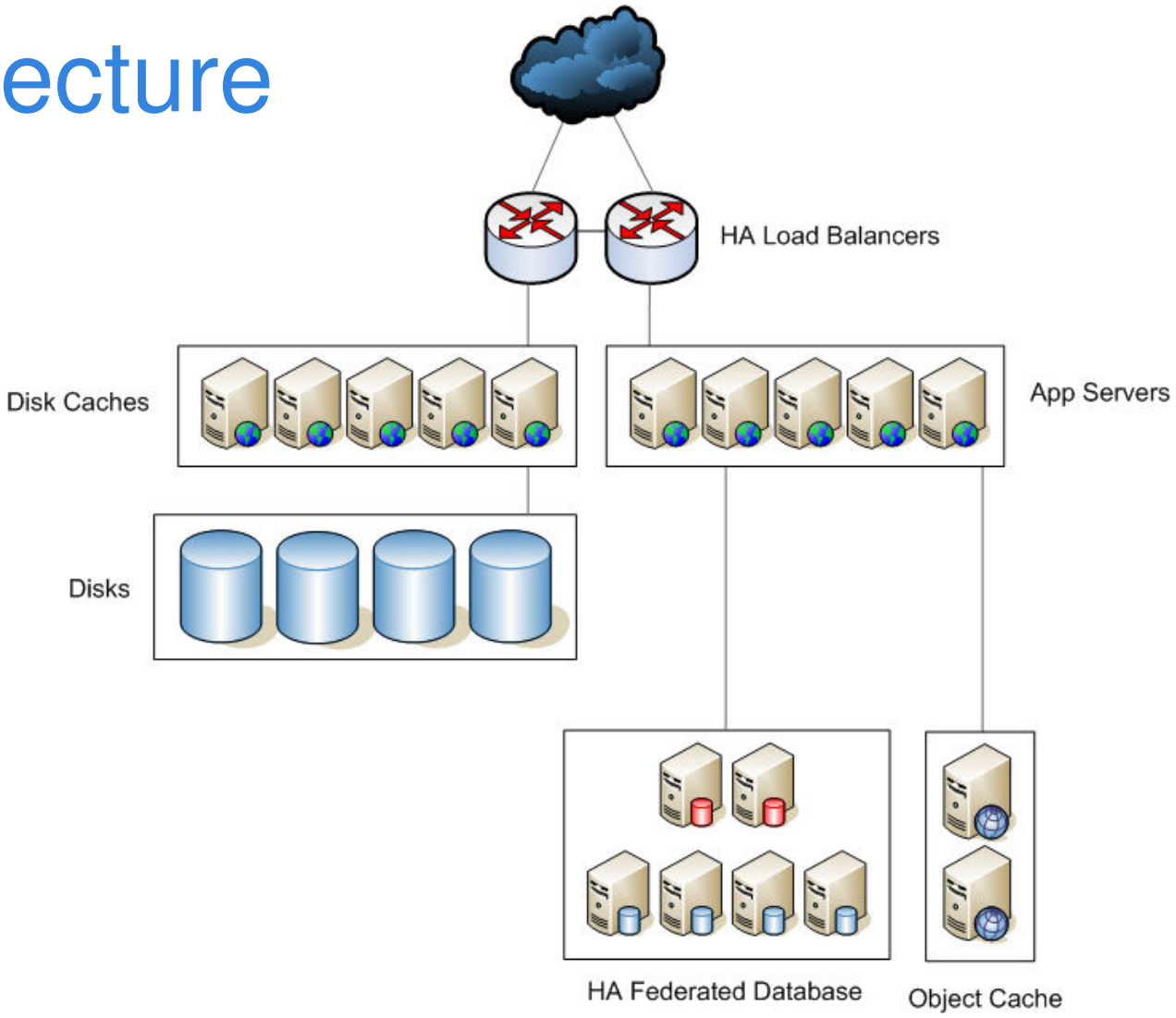
Real world examples

- Flickr
  - Because I know it
- LiveJournal
  - Because everyone copies it

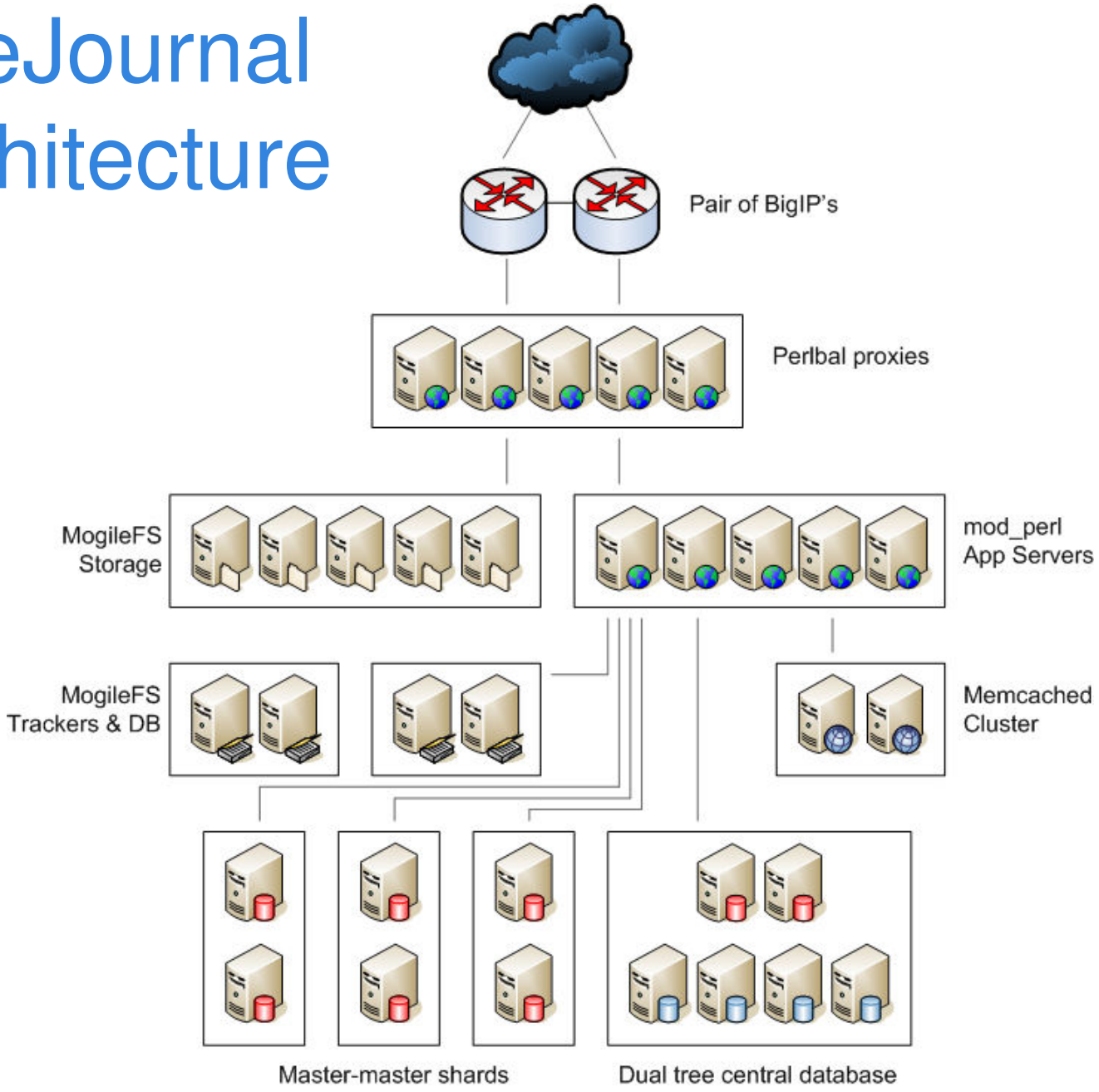
Flickr Architecture



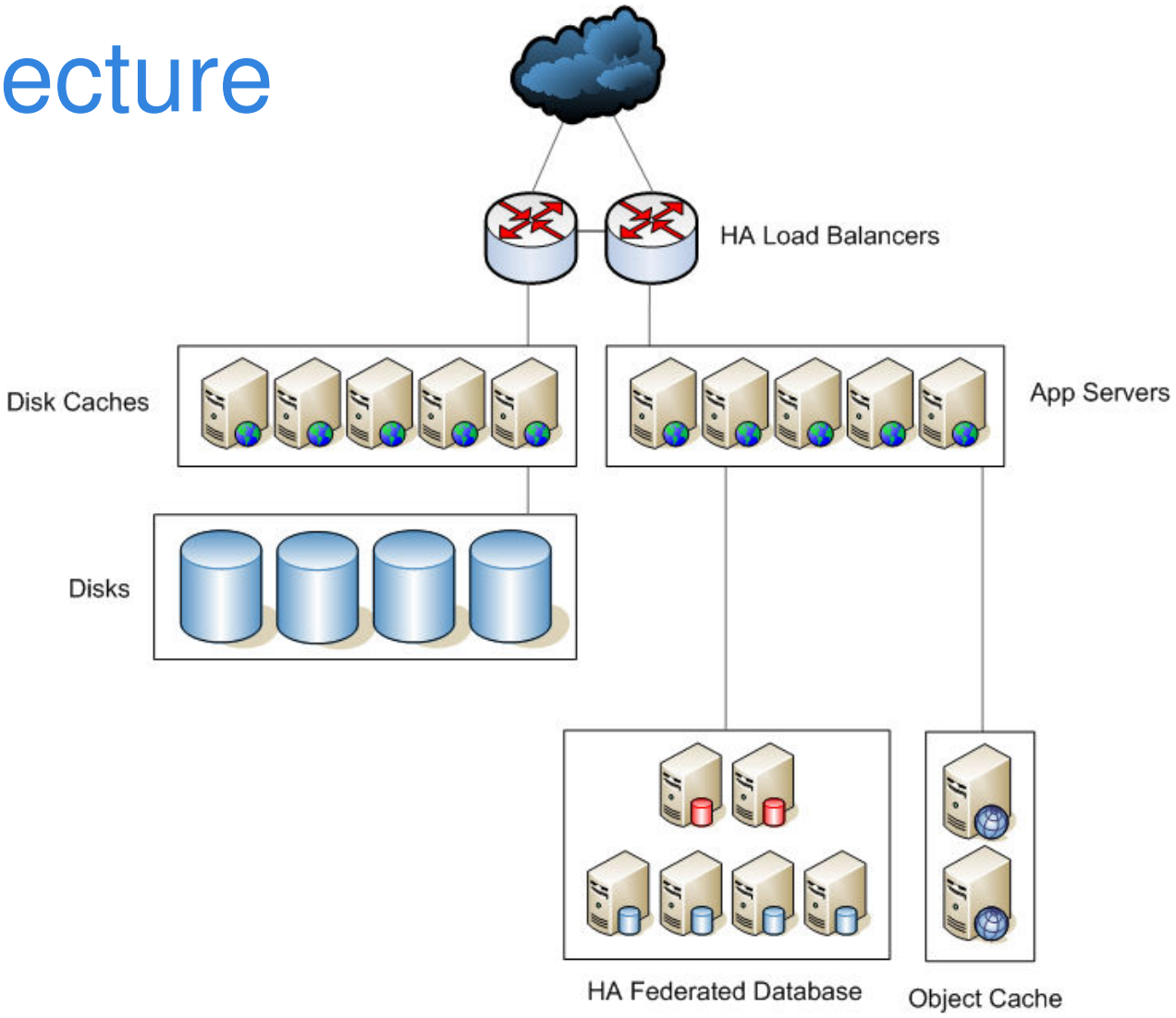
Flickr Architecture



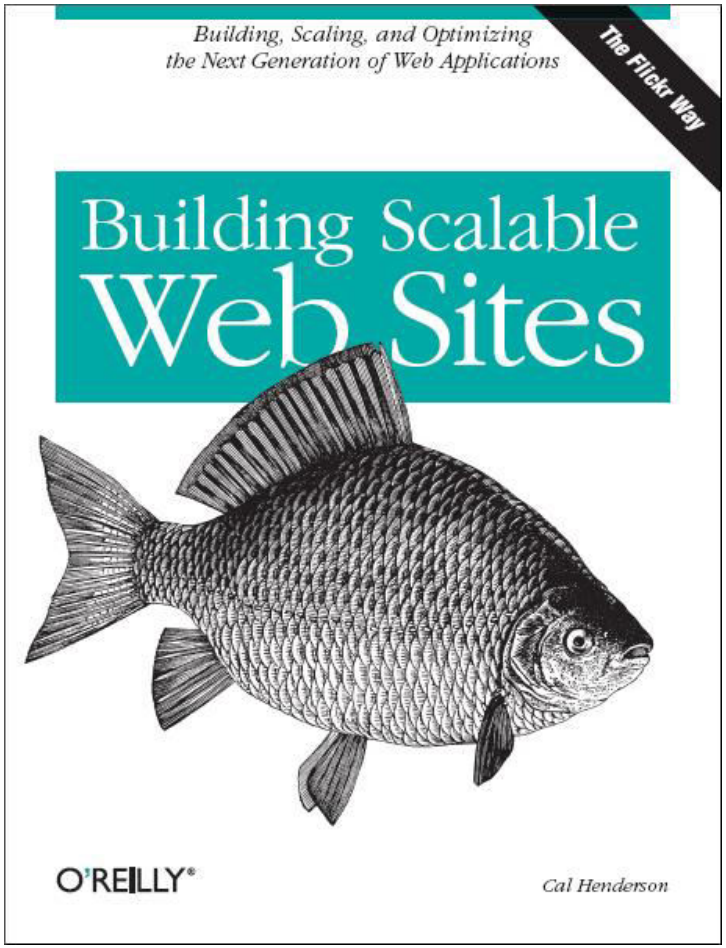
LiveJournal Architecture



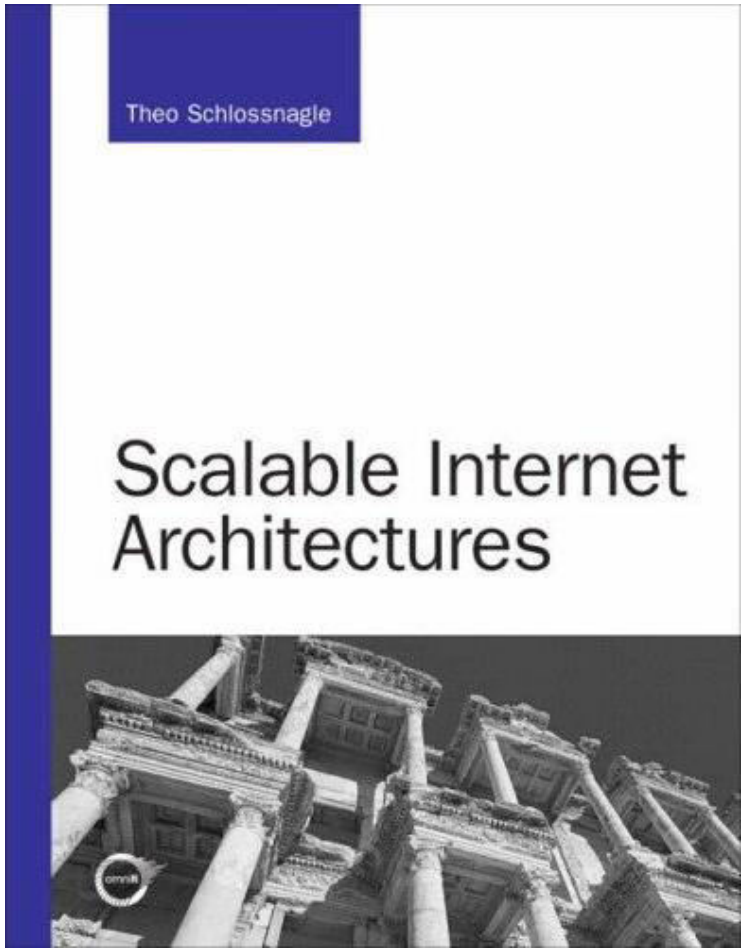
LiveJournal Architecture



Buy my book!


Building, Scaling, and Optimizing
the Next Generation of Web Applications
The Flickr Way
Building Scalable
Web Sites
O'REILLY
Cal Henderson

Or buy Theo's



The end!

Awesome!

These slides are available online:

iamcal.com/talks/