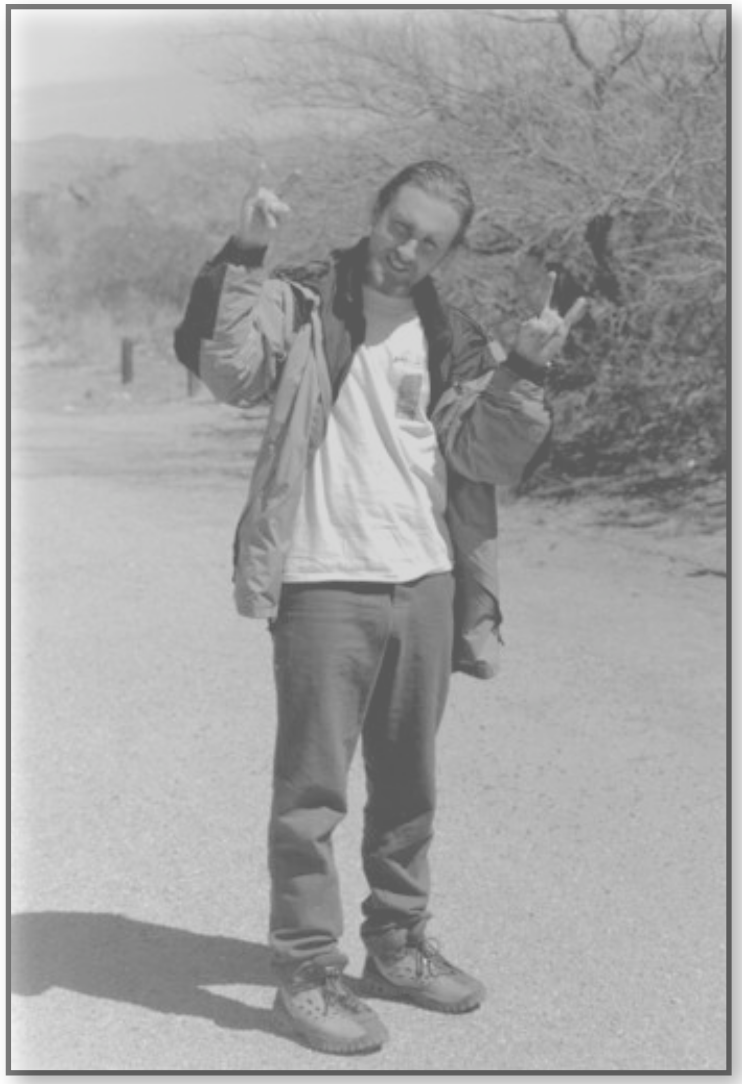
Scalable Internet Architectures

how to build scalable production Internet services and...

how not to build them

A bit about the speaker



Principal
OmniTI Computer Consulting, Inc.

- open-source developer
  - mod_backhand, wackamole, Daiquiri, Spread, OpenSSH/SecurID, a variety of CPAN modules, etc.
- closed-source developer
  - Ecelerity (MTA), EC Cluster (MTA Clustering)
- open-source advocate
  - Closed source software has technical risk.
- closed-source advocate
  - It's about business, *not* software. Finding the right tool for the job sometimes leads to closed-source solutions.

What is this talk really about?

It's about scalability...

(not really)

It's about how to build really big stuff...

(not really)

It's about learning to question the solutions to the problems we face.

(that's as close as we're going to get)

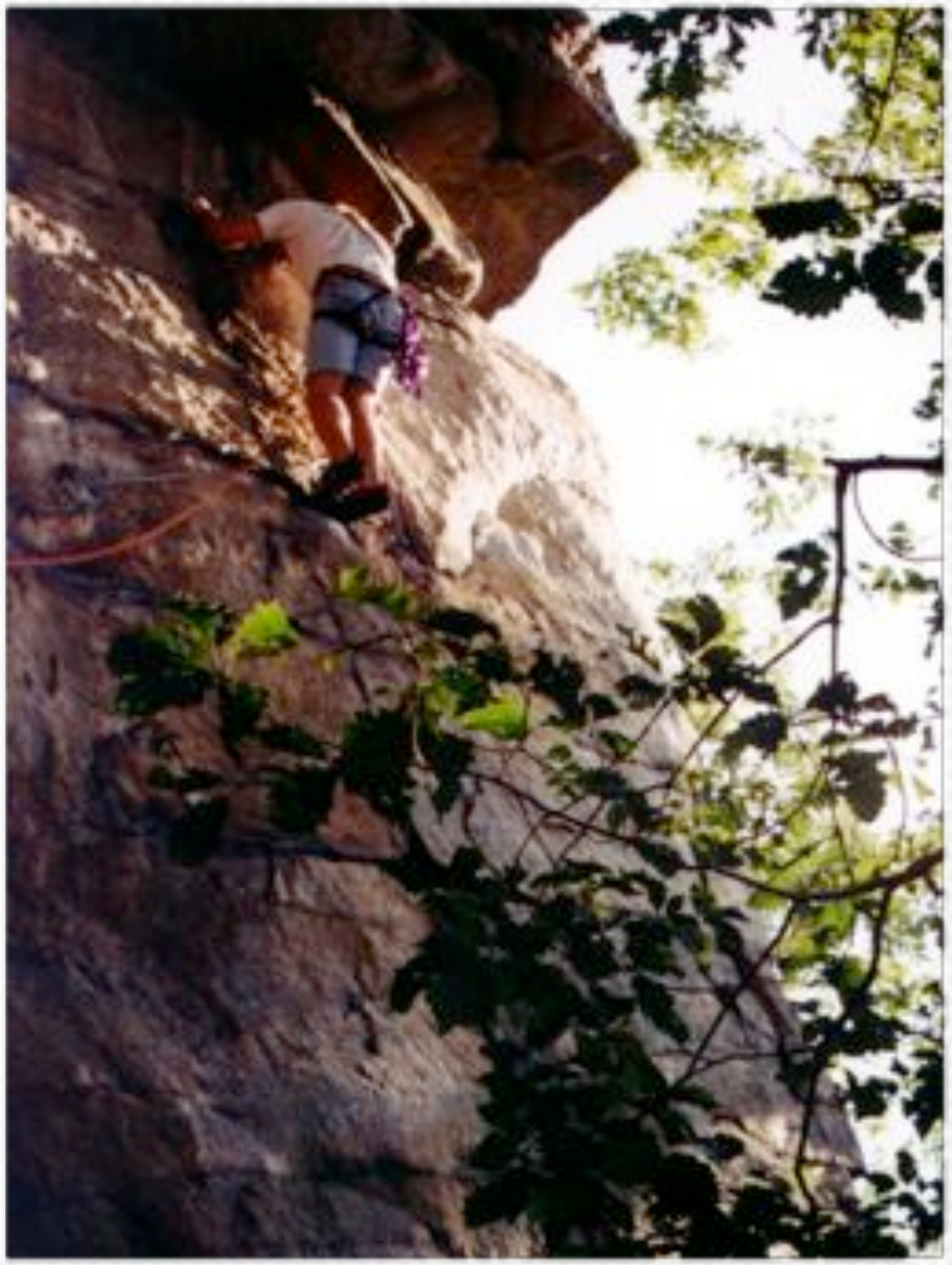
What is Scalability?

Definition:

How well a solution to some problem will work when the size of the problem increases.

What's missing?

... when the size decreases
the solution must fit



What is Scalability not?

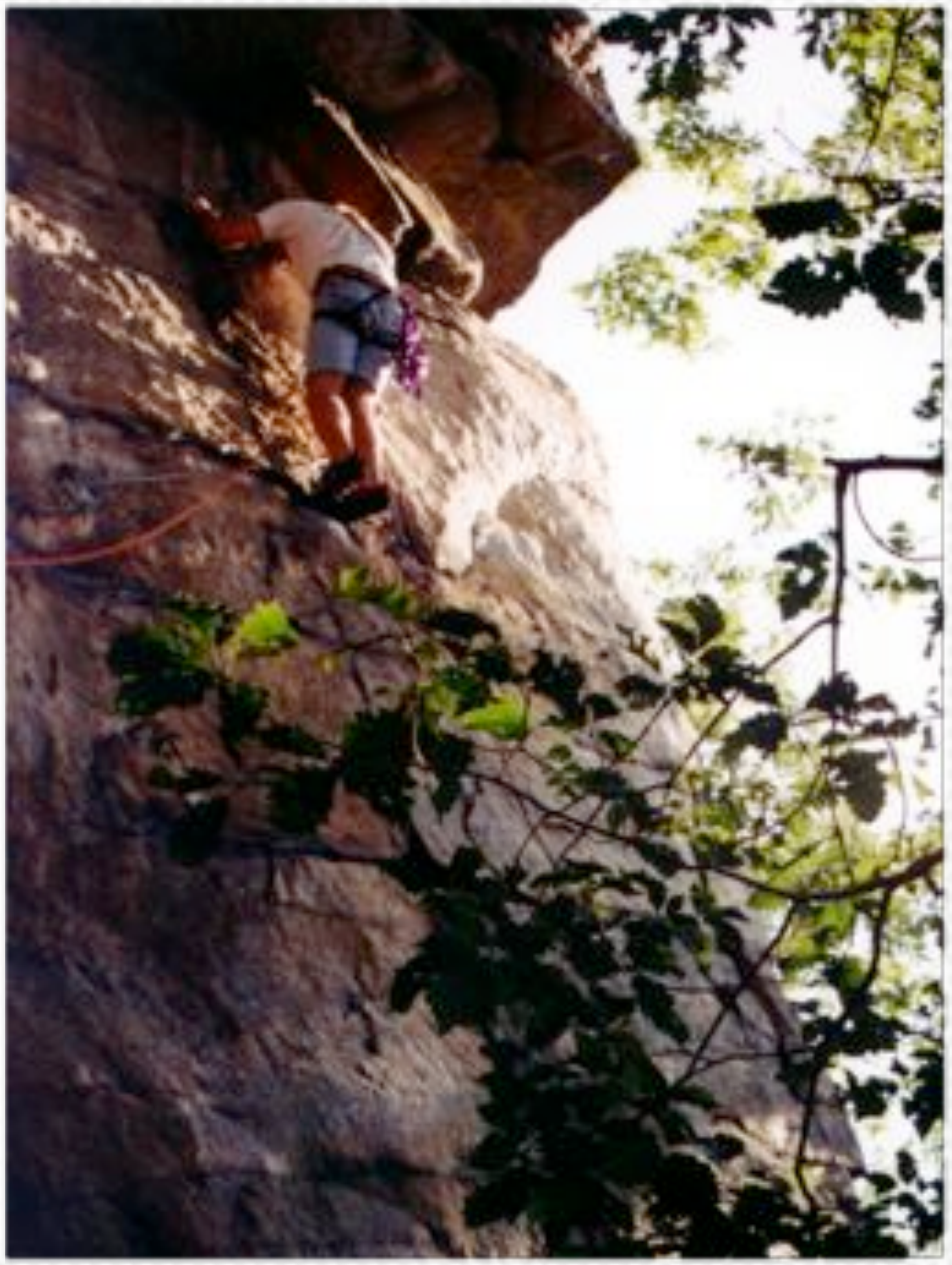
Performance:

The capabilities of a machine or product, esp. when observed under particular conditions

It has a different definition than scalability

because it is different.

This talk is not
“Performance Internet Architectures”



Production Environments

high uptime

low maintenance

formal procedures

cost controlled

High Uptime
Availability despite individual system failures

- **parallel servers**
  - all servers are live and can handle transactions
    - cheap and common for web servers
    - expensive for databases
- **hot spare/standby**
  - fail-over system that is seamless and immediate (automated)
    - common for HA/LB solutions
    - many databases have built-in facilities providing hot-spare service
- **warm spare/standby**
  - fail-over system is nearly immediate, but not seamless (not automated)
    - common technique for databases, cheap and easy
- **cold spare/standby**
  - “I have the equipment and backups to get it running if it were to fail.”

Maintenance

The single largest expense in most environments

- Contributing factors:
  - The number of unique required products in the architecture
  - The stability and "replaceability" of required products
  - Uneducated development and implementation decisions
  - The complexity and frequency of staging and pushing new code

Formal Procedures

"Scalability marginally impacts procedure
Procedure grossly impacts scalability"

- developer code review
- religious use of revision control
- planned and reviewed upgrade strategies
- intelligent, low-cost (resources) push procedures

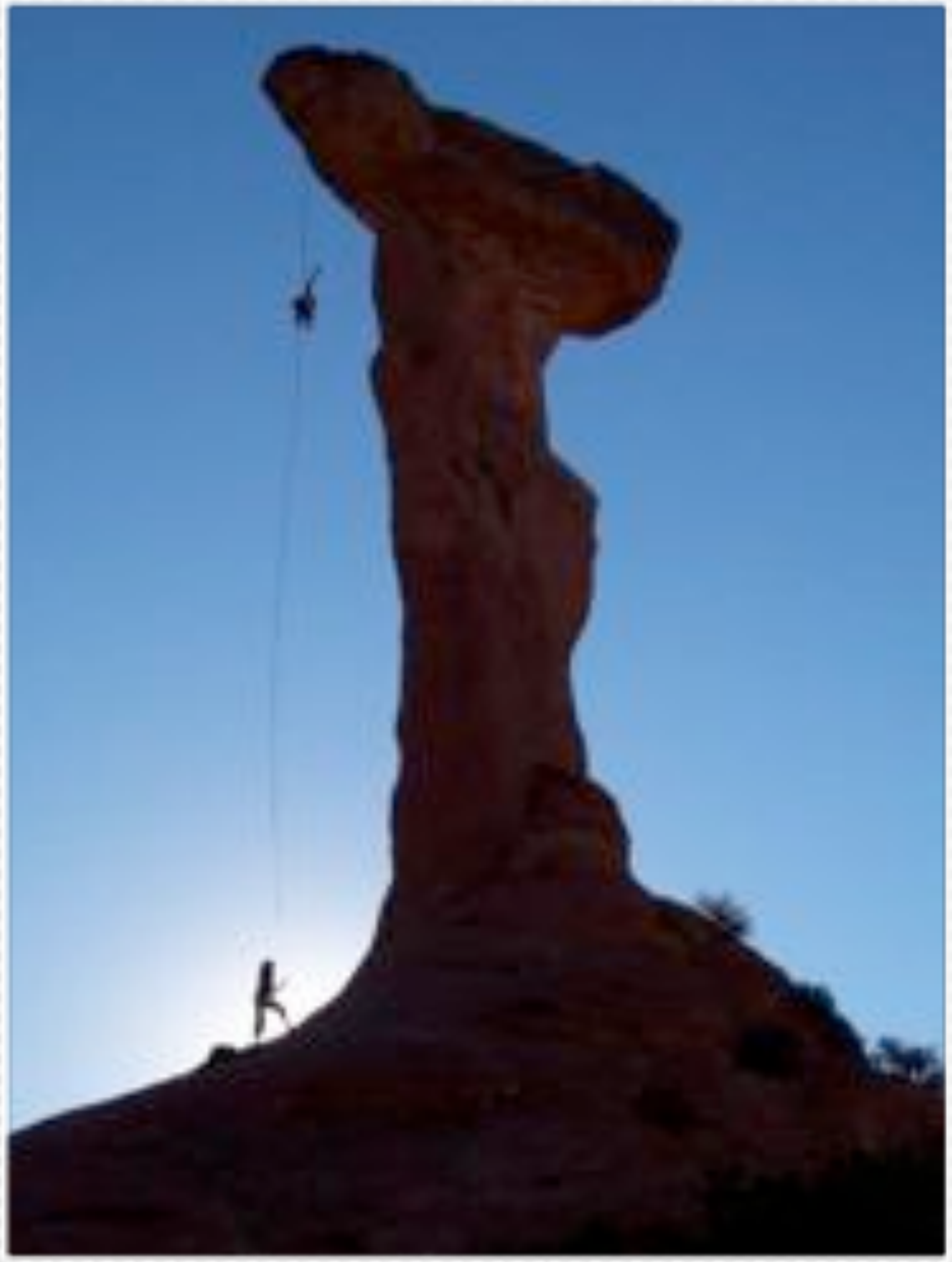
Building Production Systems

Production Fundamentals

- understand the velocity of development
- understand administrative aspects
- understand the likelihood of failure and the support for each component
- understand the stability of the software

Software Stability

- Stability is not just reliability
- Also consider:
  - release cycles
  - upgrade paths
  - feature additions, “deprications”, and removals



The Need For Speed
vs.
The Need For Control

- Understand the velocity of development
  - For Small Projects: use revision control
  - For Large Projects: use revision control
  - For ALL Projects: use revision control
- No revision control?
  - Accident waiting to happen

The Need For Speed
vs.
The Need For Control

- Unchecked speed is costly
- Rapid release cycles (once/day) are needed in some businesses
- An equilibrium must achieved or the situation will explode
- Properly used revision control allows for speed and control
- It is challenging, but meticulous unwavering adherence to policy and procedure will deliver you from disaster.

Administration

- This deserves a lot of attention
  (despite the single slide here)
- Systems Administration costs money
  - Short release cycles on components means perpetual administration
  - Constant change in development product results in different stress on:
    - Databases, Networks, Systems... and the people that maintain them
  - Adding components or complicating the architecture complicates:
    - Monitoring
    - Upgrading
    - Scaling down should the need arise

Likelihood of Failure
(the hidden administrative nightmare)

- Internally Developed Application Failures Suck.
- Third-Party Component Failures Suck More!
  - It is seen as an administration responsibility
  - Regardless if developers dictated their inclusion in the architecture
  - SAs, NAs, and DBAs suddenly become responsible for the ongoing maintenance of all third-party products -- open source or commercial
  - This is often beyond the expertise/attention of the individual or team
  - Systems fail, it's part of life
  - Chronic problems and failures will explode your TCO

Likelihood of Failure
(solution to the hidden administrative nightmare)

- Don’t leave “requirement” assessments at:
  - “This won’t work... but you’re the boss”
- Worst
  - implementing something that won’t work
  - being responsible for making it work
  - getting fired for perceived incompetence

- Bad
  - getting fired for refusing to implement something that has no hope of working
- Best
  - work with the development team to revise requirements and architectural needs

Three Simple Rules

optimize where it counts

complexity has costs

use the right tool

Three Simple Rules

#1: Amdahl's Law

- **Good**

  improving execution time by 50%
  of code that executes 2% of the time
  results in 1% performance improvement

- **Better**

  improving execution time by 10%
  of code that executes 80% of the time
  results in 8% performance improvement

Three Simple Rules
#2: Complex architectures are expensive

- adding an additional architectural component to a service or set of services increases the system complexity linearly
- requiring an additional architectural component for a service increases the system complexity exponentially

Three Simple Rules

#3: Using the wrong tool is expensive (and stupid)

- using a tool because it is easy or familiar doesn't make it right
- it is often a gratuitous waste of resources
- white papers are marketing tools and may not represent the most practical solution
- it's about good design and implementation practices

Three Simple Rules

#1: Amdahl's Law (again)

- **Good**

  improving execution time by 50%
  of code that executes 2% of the time
  results in 1% performance improvement

- **Better**

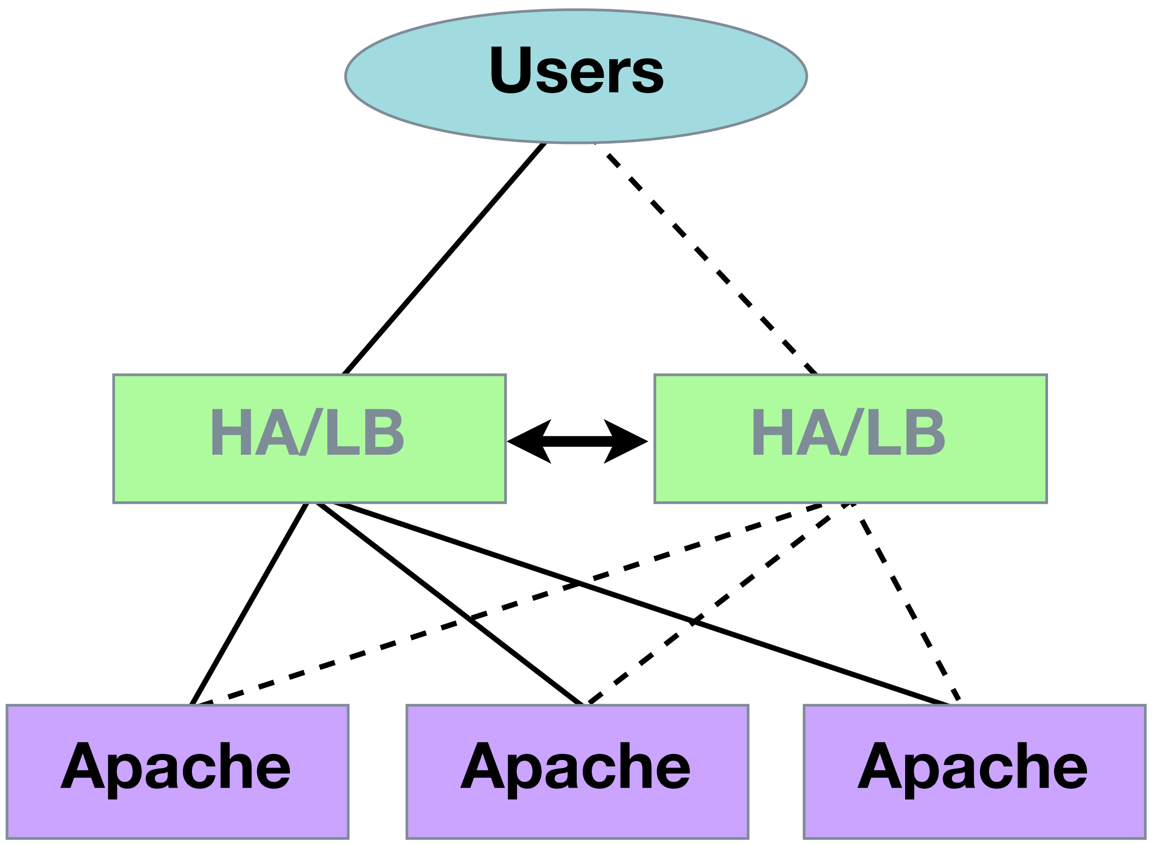
  improving execution time by 10%
  of code that executes 80% of the time
  results in 8% performance improvement

Clustered Image Serving

Goal

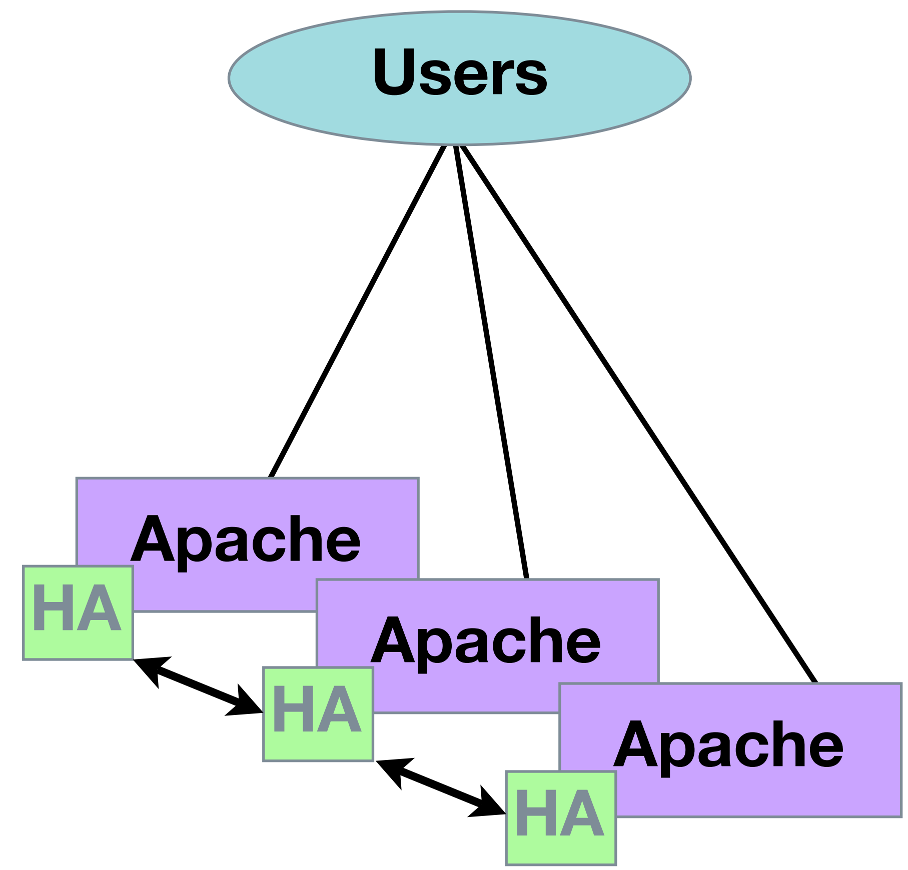
- Static image serving
- 120MBs throughput
- 24x7 uptime requirements
- Three geographically distributed sites

Decisions
high uptime



“White Paper” Approach

expensive, dedicated, single-purpose
HA/LB devices



Peer-based HA

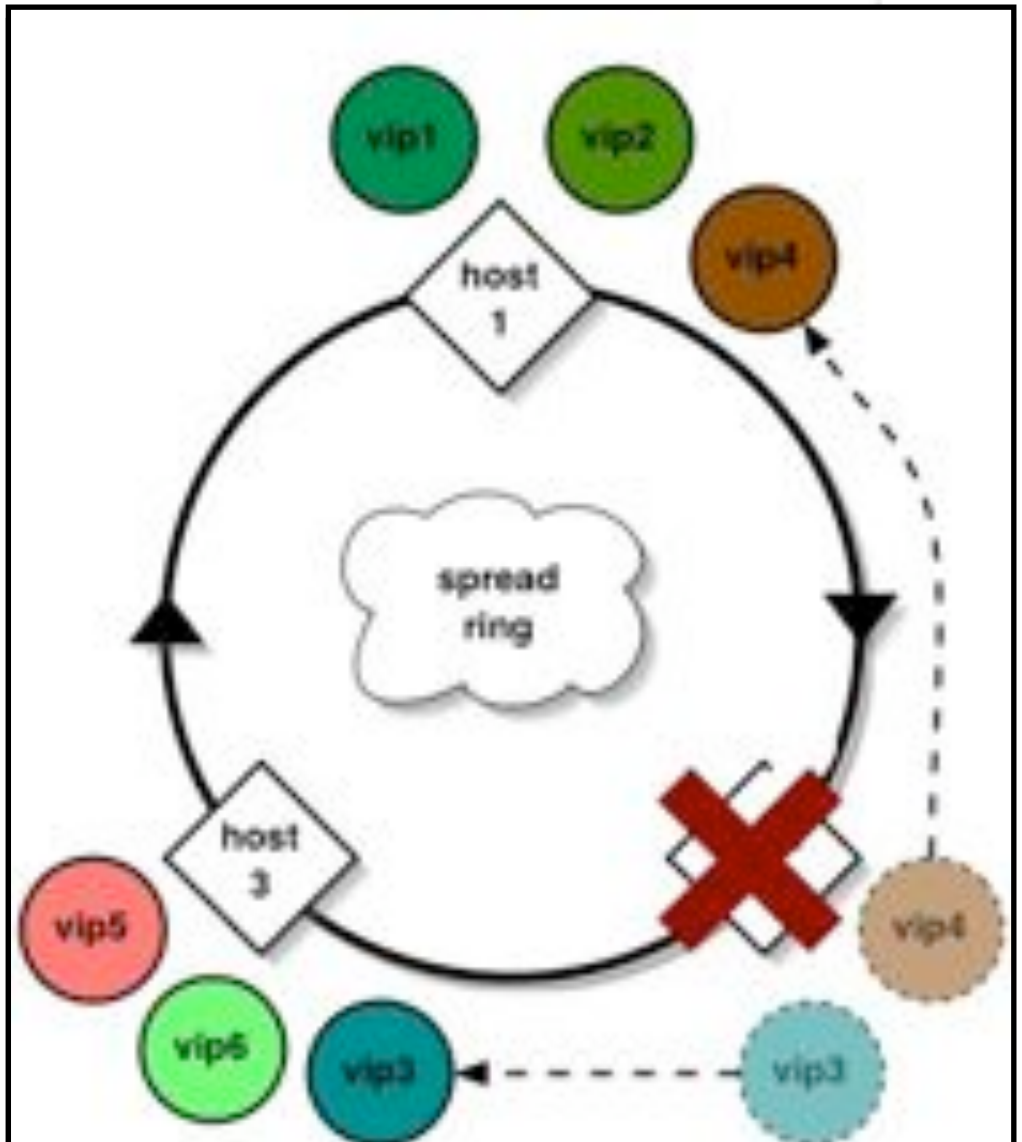
cheap and reusable
commodity machines

The ~~Tired~~ Tiered Approach

- **Pros:**
  - Fine-grained, connection-based request distribution (load balancing)
  - 100,000+ concurrent connections
  - Session management (sticky)
  - One IP per service
- **Cons:**
  - Expensive
  - Single purpose
  - Your HA solution needs HA!!!
  - 3 locations requires 6 units
  - High maintenance (additional hardware component)

Peer-based HA

Wackamole



- **Pros:**
  - No specialized hardware
  - Low maintenance (software daemon)
  - Simple
  - Free
- **Cons:**
  - Naïve load balancing (DNS RR)
  - Requires multiple IPs for a single service (bad for multi-SSL)

Policy & Procedure

- Pushing content
  - Even for small (~100Mb) image repositories, pushes are expensive
    - dumb protocols have horrible network costs
    - rsync still incurs substantial I/O for each “mirror”
    - multicast rsync could work, but there are no solid implementations
- Pulling content
  - Assuming a slow rate of change, cache-on-demand is solid
    - Use Apache + mod_proxy (Reverse Proxy + Caching)
    - Fine-grained cache purging is a challenge

Scaling Up
3 Sites

- Goal
  - 200Mbs throughput requirement
  - The goal is lower latency
  - only 2 web servers per site needed for fault tolerance
- Traditional “White Paper” Approach
  - 3 x dual HA/LB
  - 3 x 2 image web servers

$$\begin{array}{r} 3 \times 2 \times \$10000 \\ +\ 3 \times 2 \times \quad \$2000 \\ \hline \$72000 \end{array}$$

- Peer-based HA Solution
  - 3 x 2 image web servers

$$\begin{array}{r} 3 \times 2 \times \quad \$2000 \\ \hline \$12000 \end{array}$$

Scaling Down
1 Site

- Goal
  - 10Mbs throughput requirement
  - The goal is lower latency
  - only 2 web servers per site needed for fault tolerance
- Traditional "White Paper" Approach
  - dual HA/LB
  - 2 image web servers

2 x	$10000
+ 2 x	$2000
	$24000

- Peer-based HA Solution
  - 2 image web servers

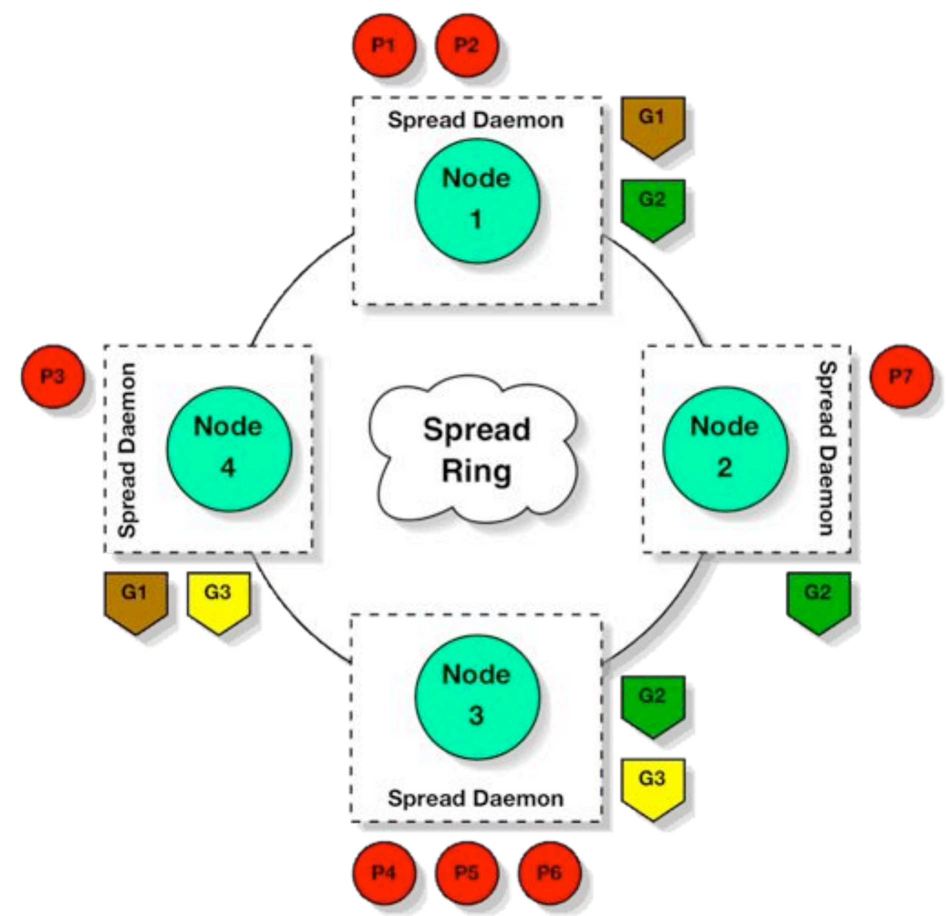
2 x	$2000
	$4000

Technical Details

- Each box running FreeBSD 4-stable
  - http://www.freebsd.org/
- Spread v3.17.3
  - http://www.spread.org/
- wackamole 2.1.2
  - http://www.backhand.org/wackamole/
- Apache 1.3.33/mod_ssl + mod_proxy + patches
  - http://www.apache.org/
  - http://www.omniti.com/~george/

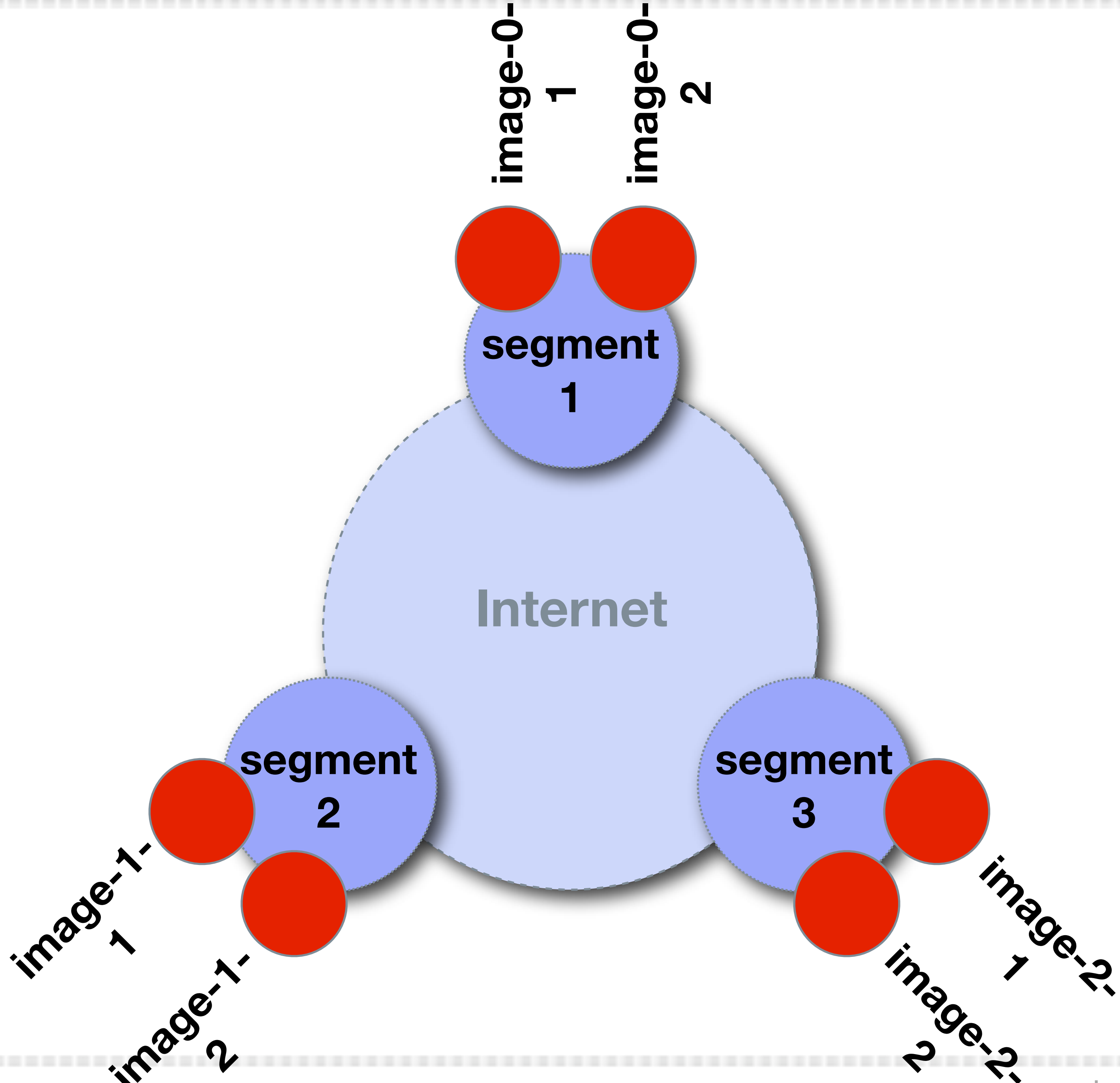
Spread: What is it?

- Group Communication
  - Messaging Bus
  - Membership
- Clear Delivery Semantics
  - Reliable or Unreliable
  - FIFO, Causal
  - Agreed, Safe
  - View of membership for delivery
- Fast and Efficient
  - N subscribers != N x bandwidth
  - Multicast or broadcast
- Usable
  - C, Perl, Python, Java, PHP, Ruby API



Spread: Configuration

```
# New Jersey Site
Spread_Segment 225.0.1.1:4803 {
    image-0-1           a.b.c.101
    image-0-2           a.b.c.102
}

# San Jose Site
Spread_Segment 225.0.1.2:4803 {
    image-1-1           d.e.f.101
    image-1-2           d.e.f.102
}

# Germany Site
Spread_Segment 225.0.1.3:4803 {
    image-2-1           g.h.i.101
    image-2-2           g.h.i.102
}
```



Wackamole: Configuration

Spread Daemon & Group →

```
Spread = 4803
SpreadRetryInterval = 5s
Group = wack1
Control = /var/run/wack.it
```

Virtual Interfaces Controlled →

```
Prefer None
VirtualInterfaces {
        { fxp0:a.b.c.111/32 }
        { fxp0:a.b.c.112/32 }
}
```

Notifications of ownership →

```
Arp-Cache = 90s
Notify {
        fxp0:a.b.c.1/32
        fxp0:a.b.c.0/24 throttle 16
        arp-cache
}
```

Balancing parameters →

```
balance {
    AcquisitionsPerRound = all
    interval = 4s
}
mature = 5s
```

Apache: Configuration

```
# Don't act as a free image caching service!
<Directory proxy:*>
        deny from all
</Directory>

# But act provide service to us
<Directory proxy:http://www.example.com/*>
        allow from all
</Directory>

RewriteEngine      on
RewriteLogLevel    0
RewriteRule   ^proxy:              -              [F]
RewriteRule   ^(http:|ftp:)        -              [F]
RewriteRule   ^\/*([^\/]+)(.*)$    http://$1$2    [P,L]
RewriteRule   .*                   -              [F]

ProxyRequests      on
CacheRoot /data/cache
CacheSize 5120000
ProxyPassReverse  /  http://www.example.com/
```

Why patch mod_proxy?

- mod_proxy hashes URLs for local caching
  - better distribution of files over directories
  - nice to your filesystem
  - good for forward caches
  - makes purging individual URLs less intuitive
- Patched to write intuitive filenames
  - a URL like: http://www.example.com/logo.gif
    becomes /data/cache/www.example.com/logo.gif
  - SAs can troubleshoot issues with certain URLs
  - cached files can be purged easily with ‘rm’
  - use Spread to distribute and coordinate cache purging operations

The Next Step

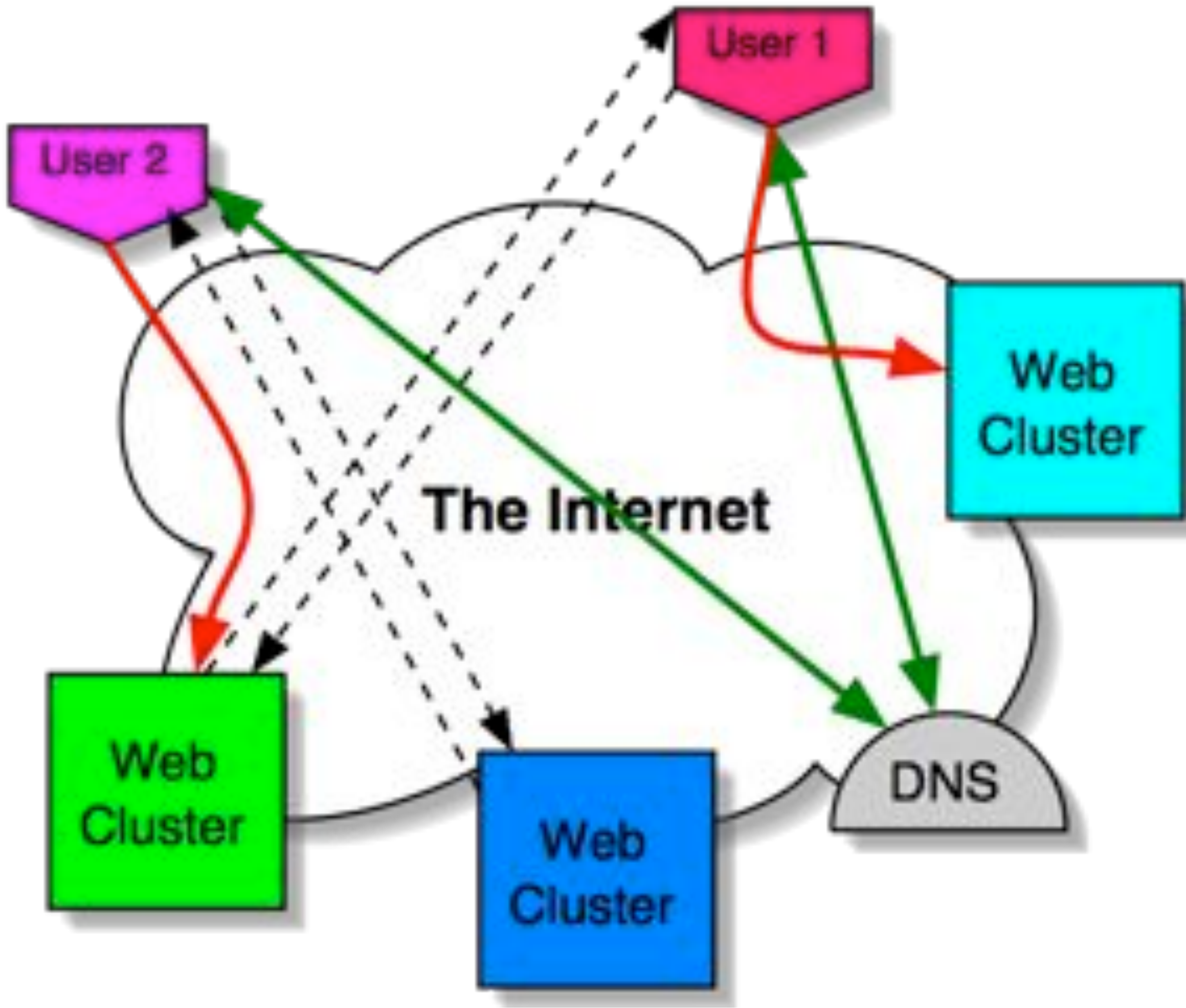
- We have three mini-clusters installed and configured, ready to throw bits by the trillions
- How people find them?

  ...DNS, obviously
- How do people find the closest cluster to them?

  ...clever DNS, [not so] obviously

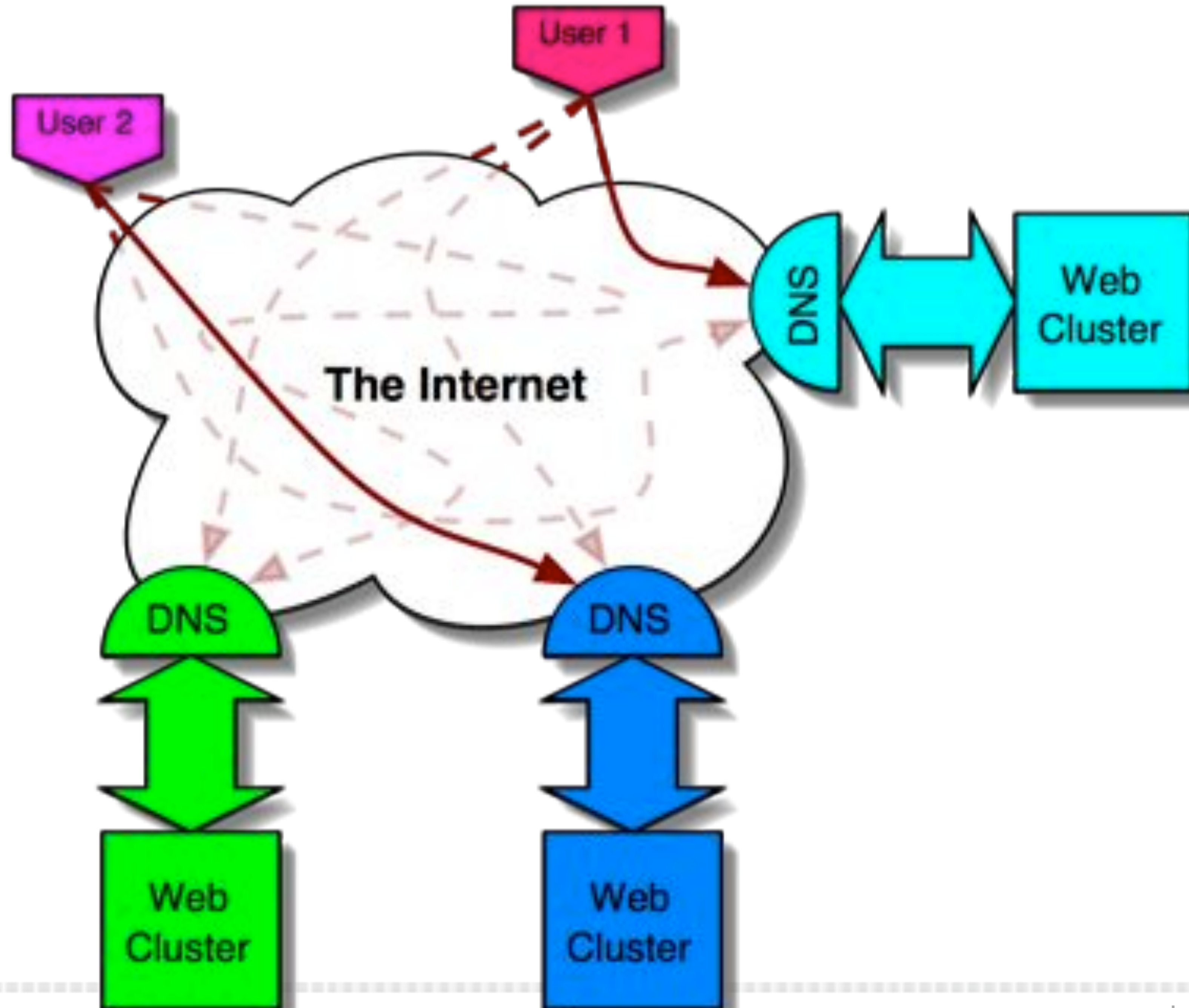
Replica Location
server-side redirection

- Application is responsible
- Use HTTP redirects:
  - images-sj.example.com
  - images-nj.example.com
  - images-de.example.com



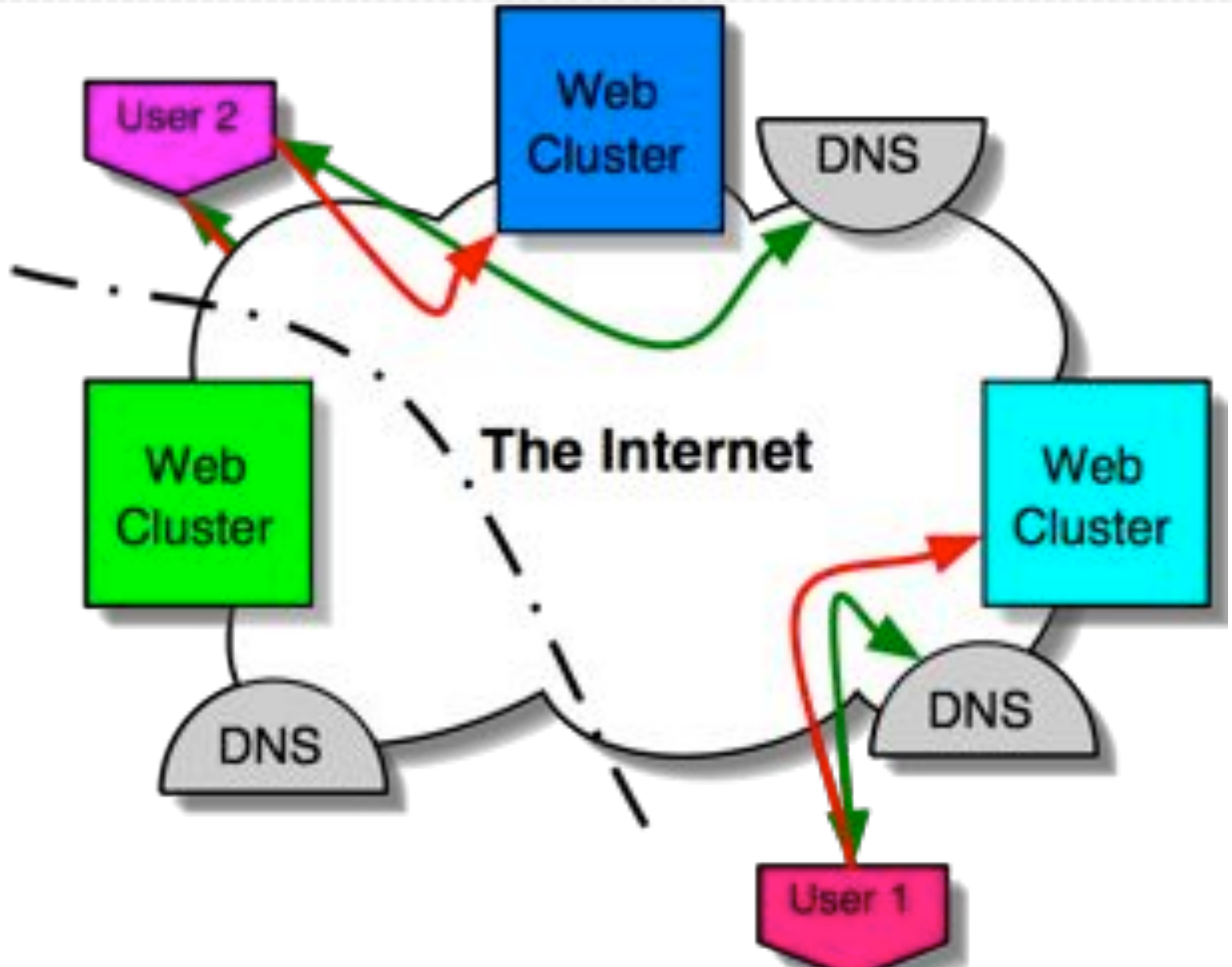
Replica Location
Proximity-based DNS

- DNS Provides Convergence
  - Can take hours/days
- DNS servers near clusters:
  - ns-sj.example.com
  - ns-nj.example.com
  - ns-de.example.com



Replica Location
DNS Shared IP (a.k.a. AnyCast)

- DNS servers near clusters:
  - ns-sj.example.com
  - ns-nj.example.com
  - ns-de.example.com
- All DNS servers have the same IP
- Network block is announced from all sites via BGP
- Routing protocols provide immediate convergence



Web Cluster Logging

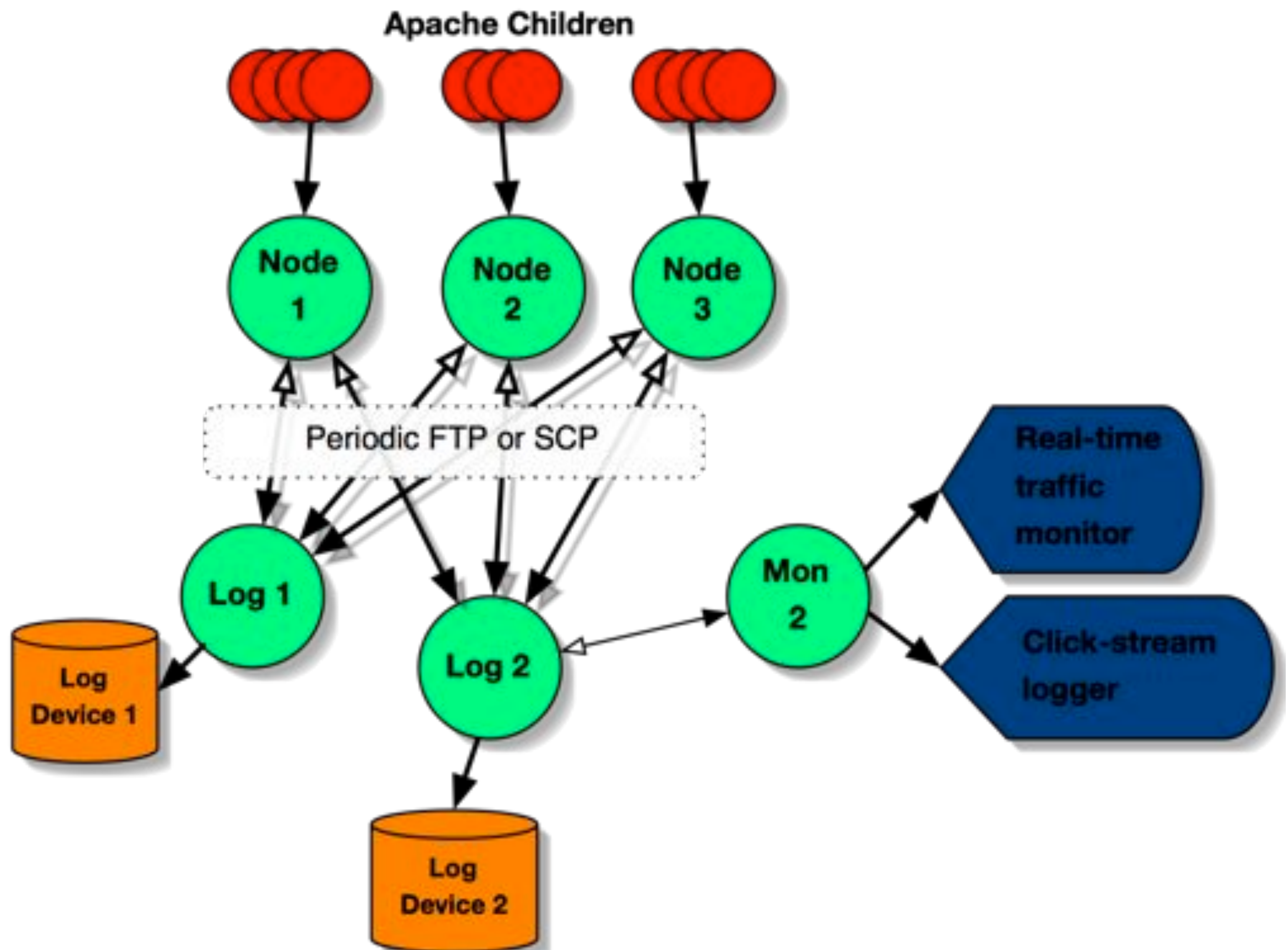
The Setup and The Goal

- Cluster of web servers
  - Apache
  - thttpd
- Logs are vital
  - must be stored in more than one place
- Real-time assessments
  - hit rates
  - load balancing
  - HTTP response code rates

Traditional Configuration
Local Logging, Post-process Aggregation

- Log written locally on web servers
  - space must be allocated
- Consolidation happens periodically
  - crashes will result in missing data
  - aggregators must preserve chronology (expensive)
  - real-time metrics cannot be calculated
- Monitor(s) must run against log server
  - monitors must tail log files
  - requires resources on the logging server

Traditional Configuration
Local Logging, Post-process Aggregation

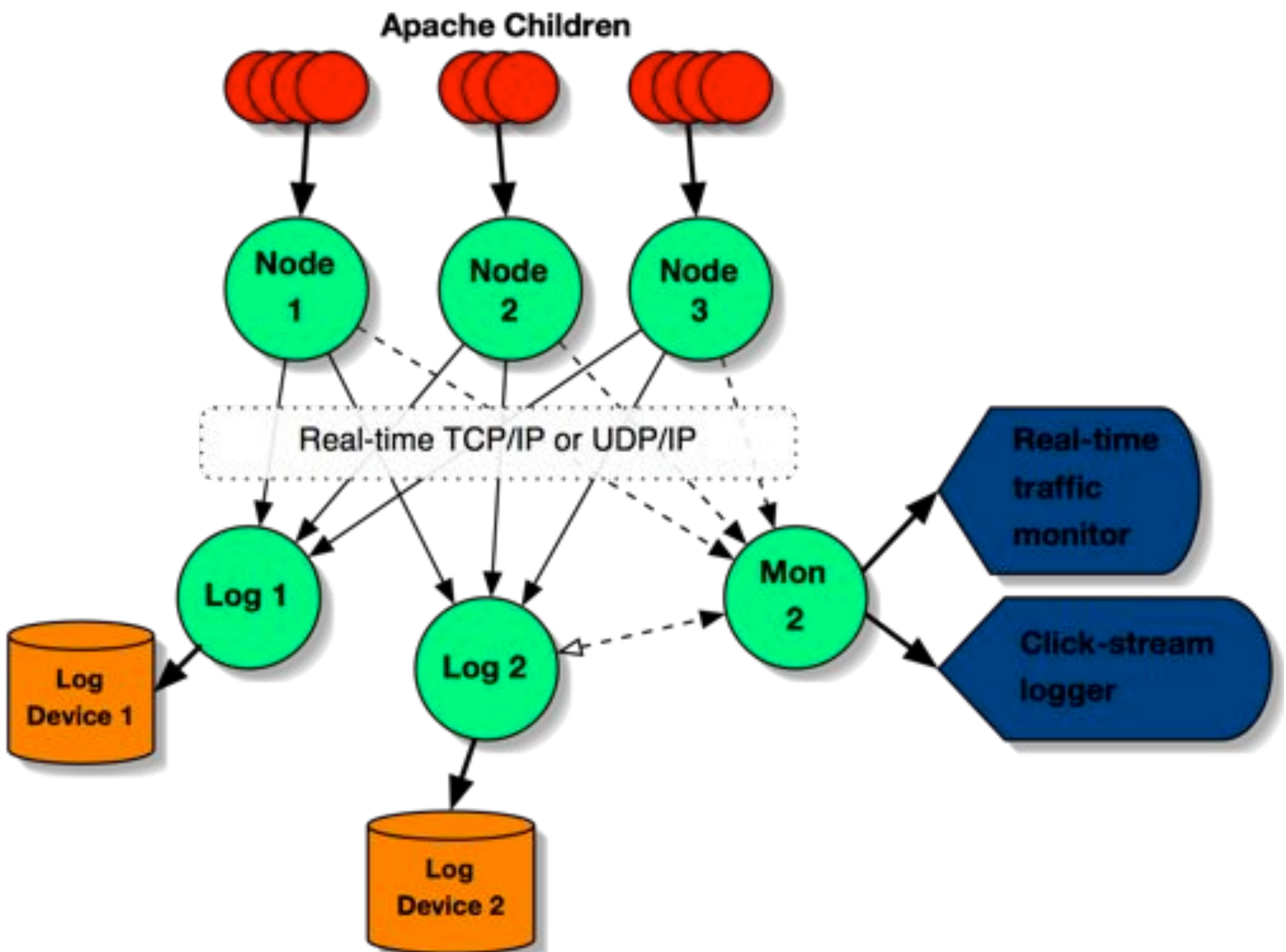


TCP/IP or UDP/IP Logging

Syslog, Syslog-NG

- Logs are written directly to logging server(s)
  - UDP is unreliable and thus not useful
  - TCP is a point-to-point protocol
    - Two logging servers means all info gets sent twice.
    - Add a monitor and that's three times!
- Real-time metrics can now be collected
  - monitors must still be run against log servers (or the publishers must be reconfigured)

TCP/IP or UDP/IP Logging
Syslog, Syslog-NG



Passive Network Logging

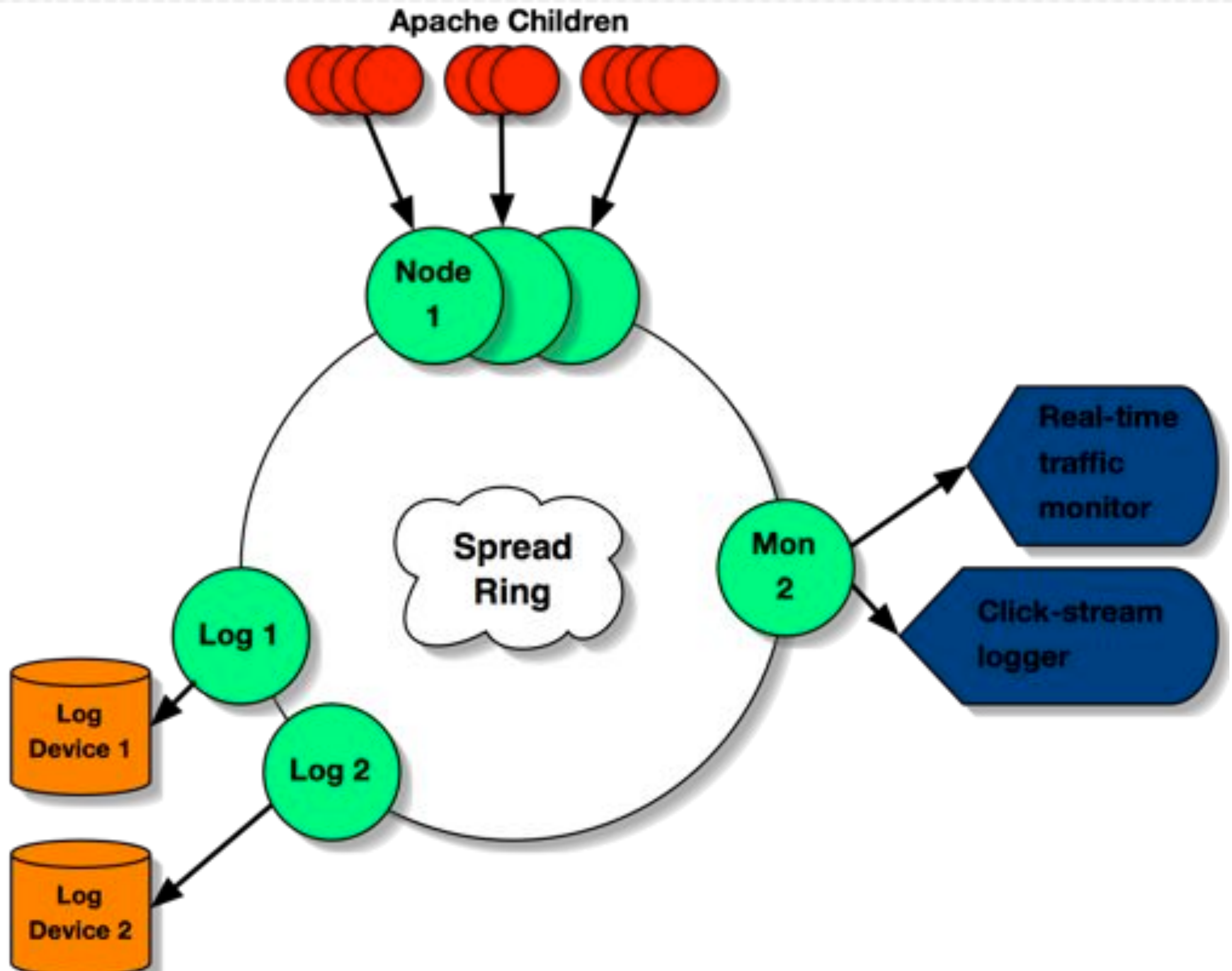
sniffers

- Requires no modification of architecture
  - Add/remove publishers (web severs) on-the-fly
- Drops Logs!
  - When tested head-to-head with traditional logging mechanisms we see loss
  - “Missing” logs are unacceptable
- **Clean the white dust off your upper lip and choose another implementation**

Reliable Multicast Logging
mod_log_spread

- Flexible Operations
  - Add/remove publishers (web severs) on-the-fly
  - Add/remove subscribers (loggers/monitors) on-the-fly
- Reliable Multicast (based on Spread)
  - Multiple subscribers don’t incur addition network overhead
- Individual real-time passive and active monitors
  - Monitors can be “attached” without resource consumption concerns
  - Passive monitors that draw graphs, assess trends, detect failures
  - Active monitors that feed metrics back into a production system
    - Who’s online
    - Real-time page access metrics

Reliable Multicast Logging
mod_log_spread



mod_log_spread
“The Publisher”

mod_log_spread is really a patch to mod_log_config

Like pipes in mod_log_config:

```
“|/path/to/rotatelogs filename 3600”
```

mls adds a Spread group destination:

```
$groupname
```

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %T" combined
CustomLog $example combined
```

spreadlogd
“The Subscriber”

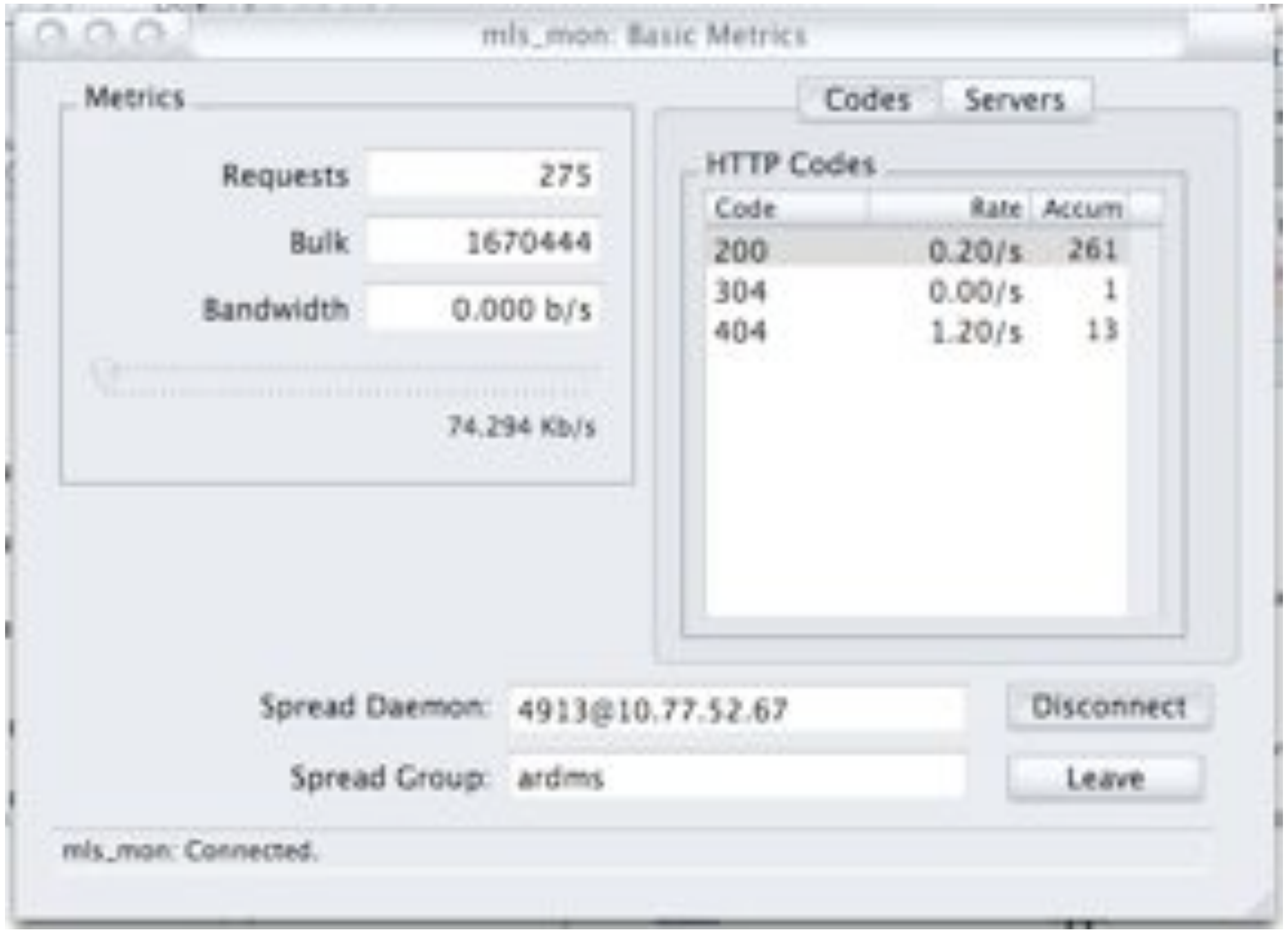
spreadlogd writes logs...

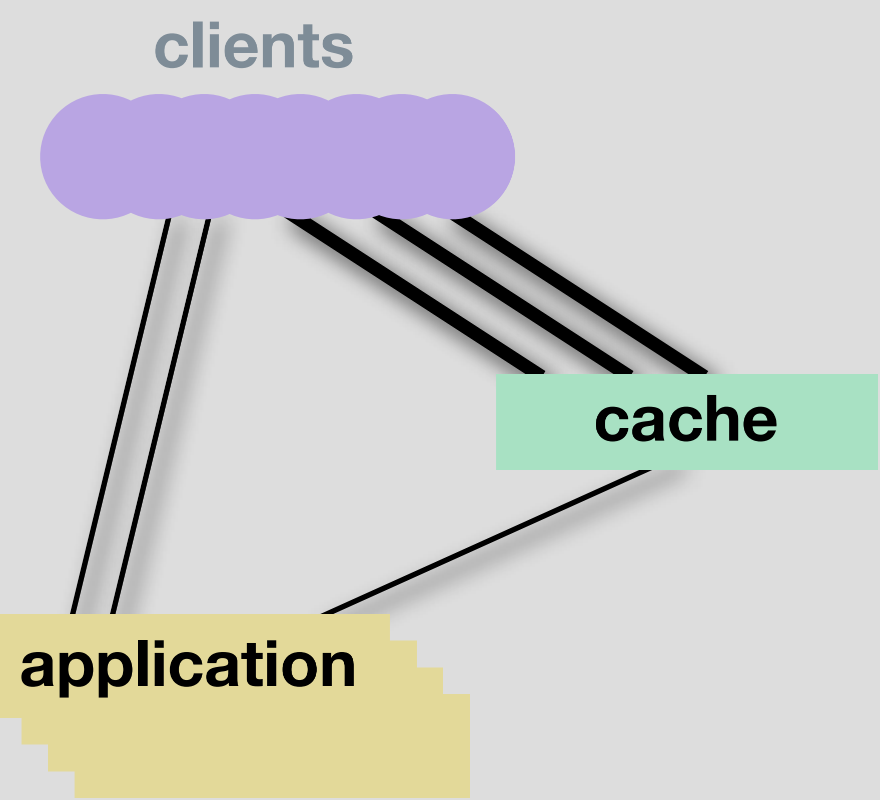
```
BufferSize = 65536

Spread {
  Port = 4803
  Log {
    RewriteTimestamp = CommonLogFormat
    Group = "example"
    File = /data/logs/apache/www.example.com/combined_log
  }
}
```

Other Real-Time Tools

"Other Subscribers"

- ApacheTop
  - C++ "top"-style real-time hit display
- mls_mon
  - graphical hit rates by server and by code





Caching Architectures

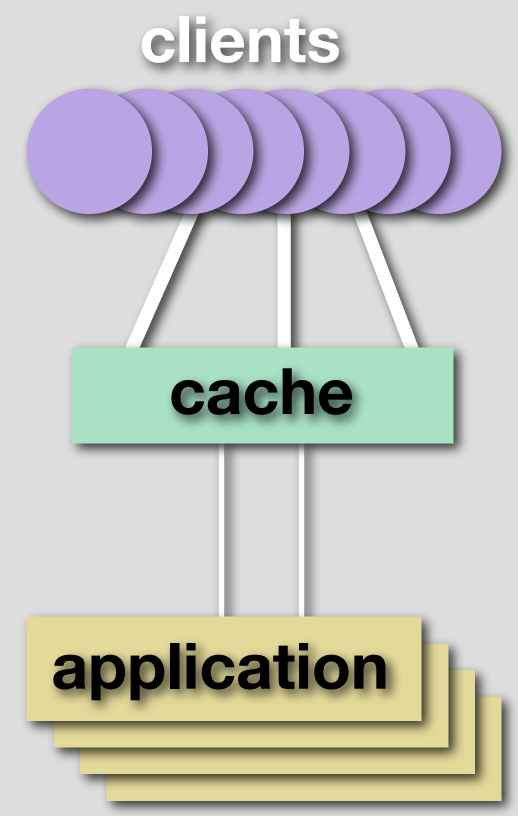
What is a cache?

Cache:

A small fast memory holding recently accessed data, designed to speed up subsequent access to the same data. Most often applied to processor-memory access but also used for a local copy of data accessible over a network etc.

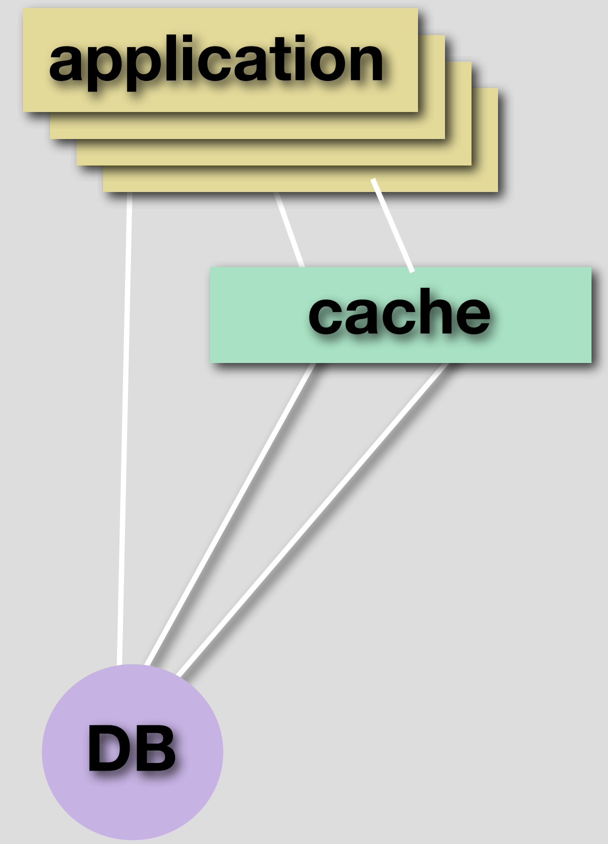
The Layered Cache

- Exists above/in-front
- Knows little or nothing about what's underneath
- Works fabulously for static data (like images)



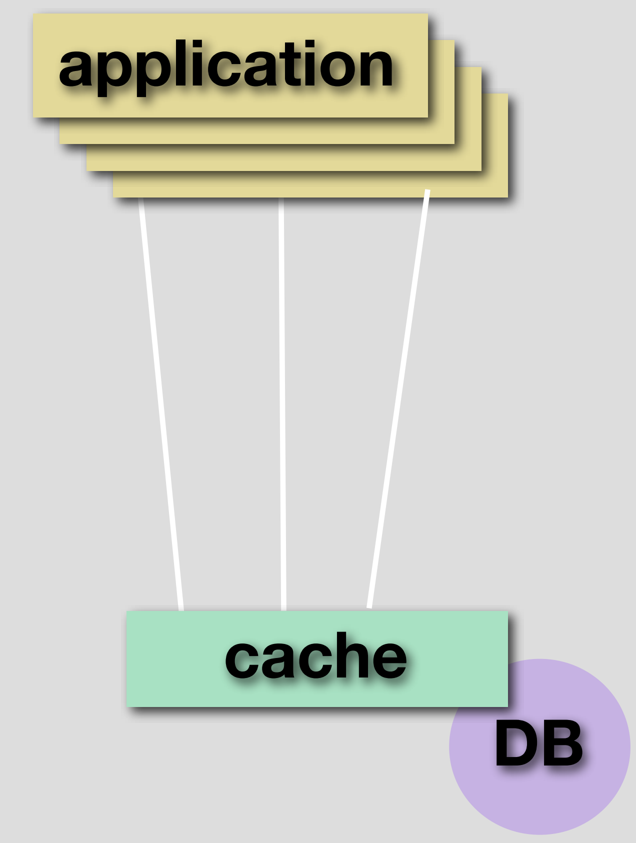
The Integrated Cache

- Exists in the application.
- Knows the data and how the application uses it.
- Works well for data that doesn't change rapidly but is relatively expensive to query.



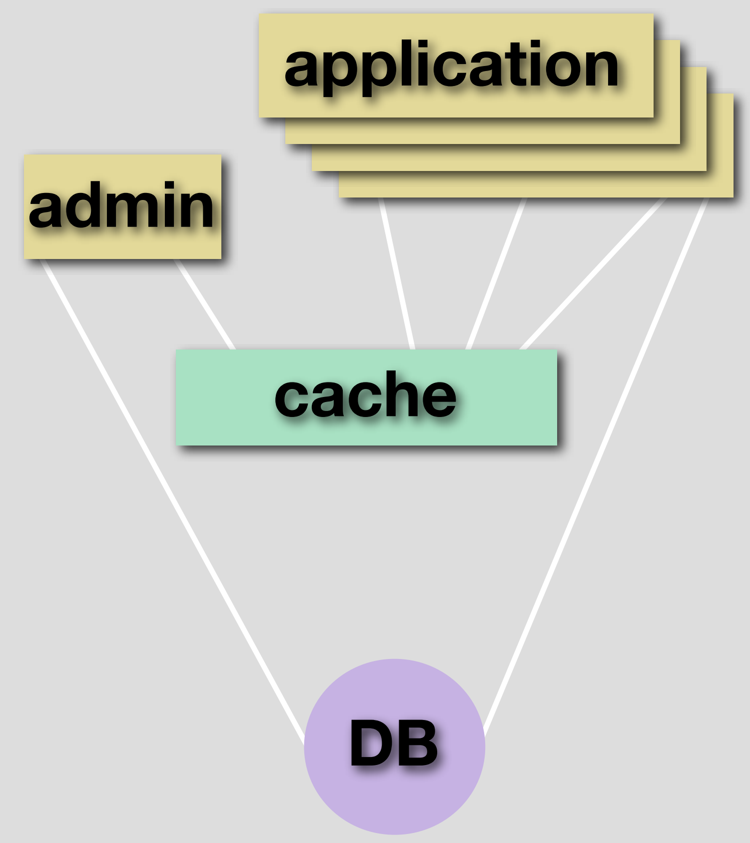
The Data Cache

- Exists in the data store
- Knows the data, the queries and how the data has changed.
- Works well always
  - called computational reuse
  - oldest trick in the book
- MySQL 4 has this
  they call it a Query Cache



Write-thru Cache

- Occurs at update location
- Knows the data, the app, the queries and how the data has changed.
- Works well for administrative updates
  - many WebLogs work this way
- Can be very adaptive and flexible



A “Real-World” Example

- News site
  - News items are stored in Oracle
  - User Preferences are stored in Oracle
- Hundreds of different sections
  - Each with thousands of different articles

- Pages:
  - 1000+ hits/second
  - shows personalized user info on EVERY page
  - front page shows top N_F articles for forum F (limit 10)

The Approach

- Oracle is fast enough
  - why abuse Oracle for this purposes?
  - surely there are better things for Oracle to be doing
- Updates are controlled
  - updates to news items only happen from a publisher
  - news update:read ratio is miniscule
  - user preferences are only ever updated by the user

Articles

- Article publishing
  - sticks news items in Oracle
- The straight forward way
  - http://news.example.com/news/article.php?id=12345
  - page pulls user prefs from cookie (or bounces off a cookie populator)
  - page pulls news item from database
- I hate query strings
  - I like: http://news.example.com/news/items/12345.html

```
RewriteRule ^/news/items/([^/]*).html$ /www/docs/news/article.php?id=$1 [L]
```

Articles Cached

- We pull the item that is likely to never change
  - cheaper if the page just hard coded the news item
  - writing the news article out into a PHP page is a hassle
    ... or is it?

- Have the straight forward page cache it
  - /news/article.php writes /news/items/12345.html as a PHP page that still expands personal info from cookie, but has the news item content statically included as HTML.

```
RewriteCond %{REQUEST_FILENAME} ^/news/items/([^/]*).html
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^/news/items/([^/]*).html$ /www/docs/news/article.php?id=$1 [L]
```

Articles Cache Invalidation

- Run a cache invalidator on each web server
  - connects to Spread as a subscriber
  - accepts /www/docs/news/items/####.html deletion requests
  - accepts full purge requests
- Article publishing
  - stash item #### in Oracle (insert or update)
  - publish through Spread an invalidation of ####
- Changing the look of the article pages
  - change article.php to have the desired effect (and write the appropriate php cache pages)
  - publish through Spread a full purge

The Result

- All news item pages require zero DB requests
  - the business can now make your life difficult by requesting new crap on these pages that can't be so easily cached
- Far fewer database connections required
  - all databases appreciate that (Oracle, MySQL, Postgres)
- Bottleneck is now Apache+mod_php
  - crazy fast with tools like APC
  - inherently scalable... just add more web servers
  - room for more application features

Tiered Architectures

Why Tier?

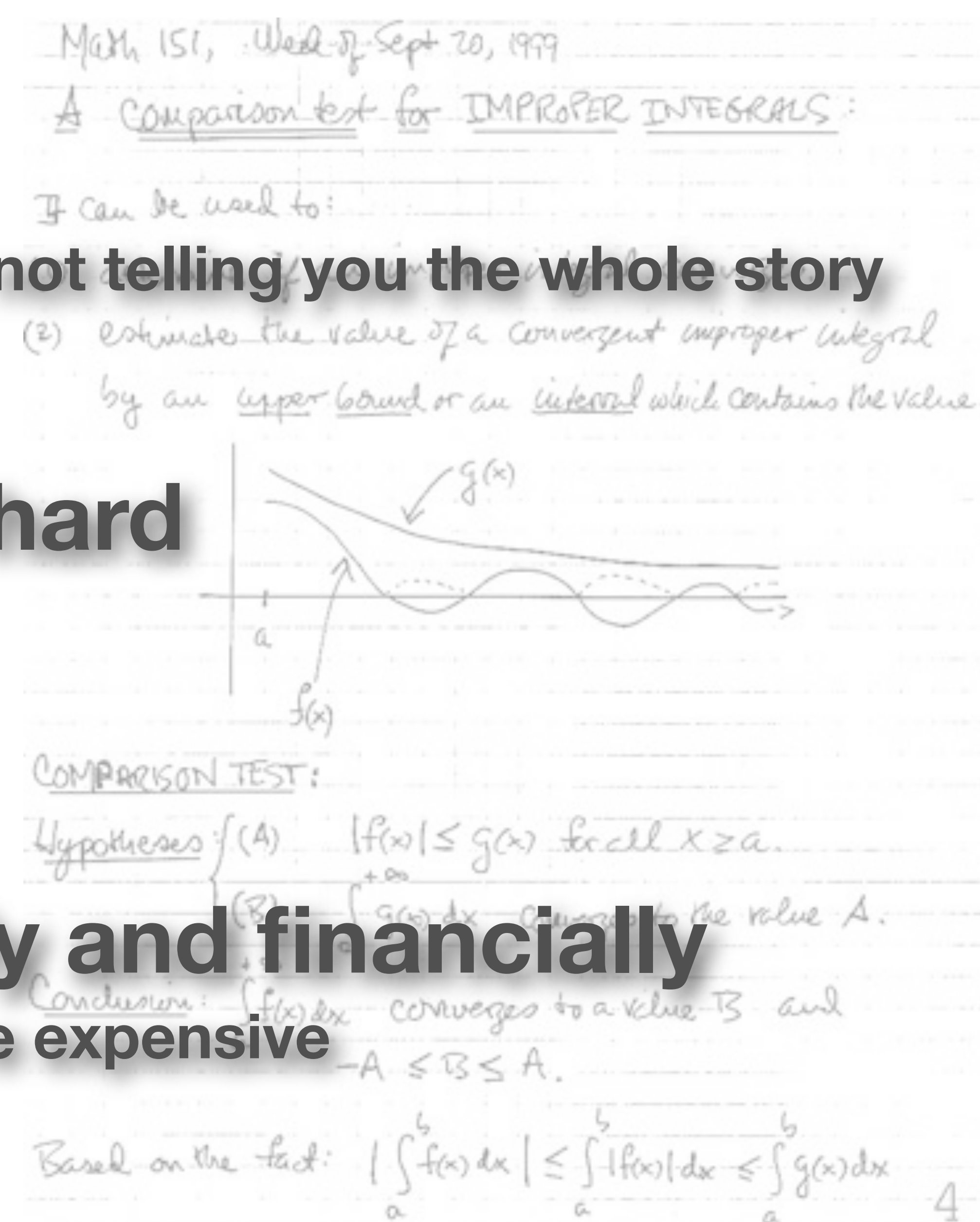
- Dedicate resources to specific components
  - Often a good approach to scaling systems up
  - Requiring single purpose components is a good way to lock into a big (expensive) architecture
- Tiers make computer science problems easier
  - Understand the trade off of solving hard problems vs. maintaining tiered solutions
- Tiers add complexity and increase maintenance costs
  - More components, more pieces, more moving parts... More can (and does) go wrong.

Dedicated Resources

- Classic Example:
  - Apache on dedicated web servers
  - Database on dedicated machine
  - Why? it is easy to have 4 web server, hard to have 4 databases
- Lock-in Example:
  - Web application on several servers
  - Requires local session state and sticky sessions
  - Why? scaling down to 2 servers will still require a load balancer that can “stick” sessions.

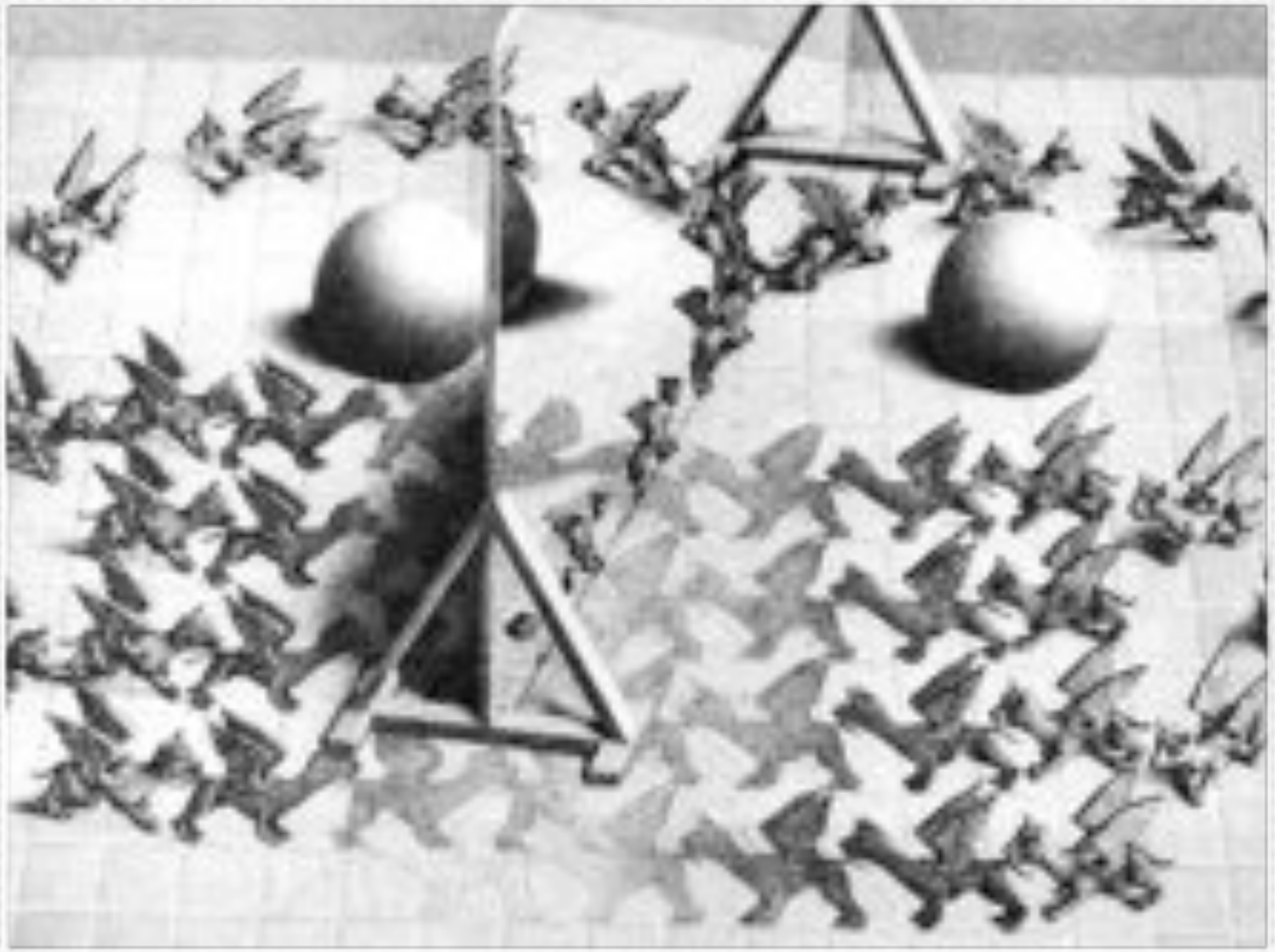
Tiering to Compensate
(for problems that are hard to solve)

- **Database Replication is hard**
  - Anyone who tells you otherwise is lying or not telling you the whole story
- **Session Replication is not so hard**
  - use a technology like Splash!
  - offload responsibility to the client
- **Tiers are expensive technically and financially**
  - Some problems more difficult than tiers are expensive



Tiers are expensive

- Intrinsically difficult to scale down
- If it is a real production system...
  - Complete staging environment
  - Complete development environment



Effective Replication

Eliminating The Need To Tier

Types of Database Replication

Master-Slave

- a data set has a master server
- changes to the data set are sent to slaves
- dml must be performed at the master
- read-only queries can be performed anywhere
- no challenging synchronization algorithms

Types of Database Replication

Master-Master

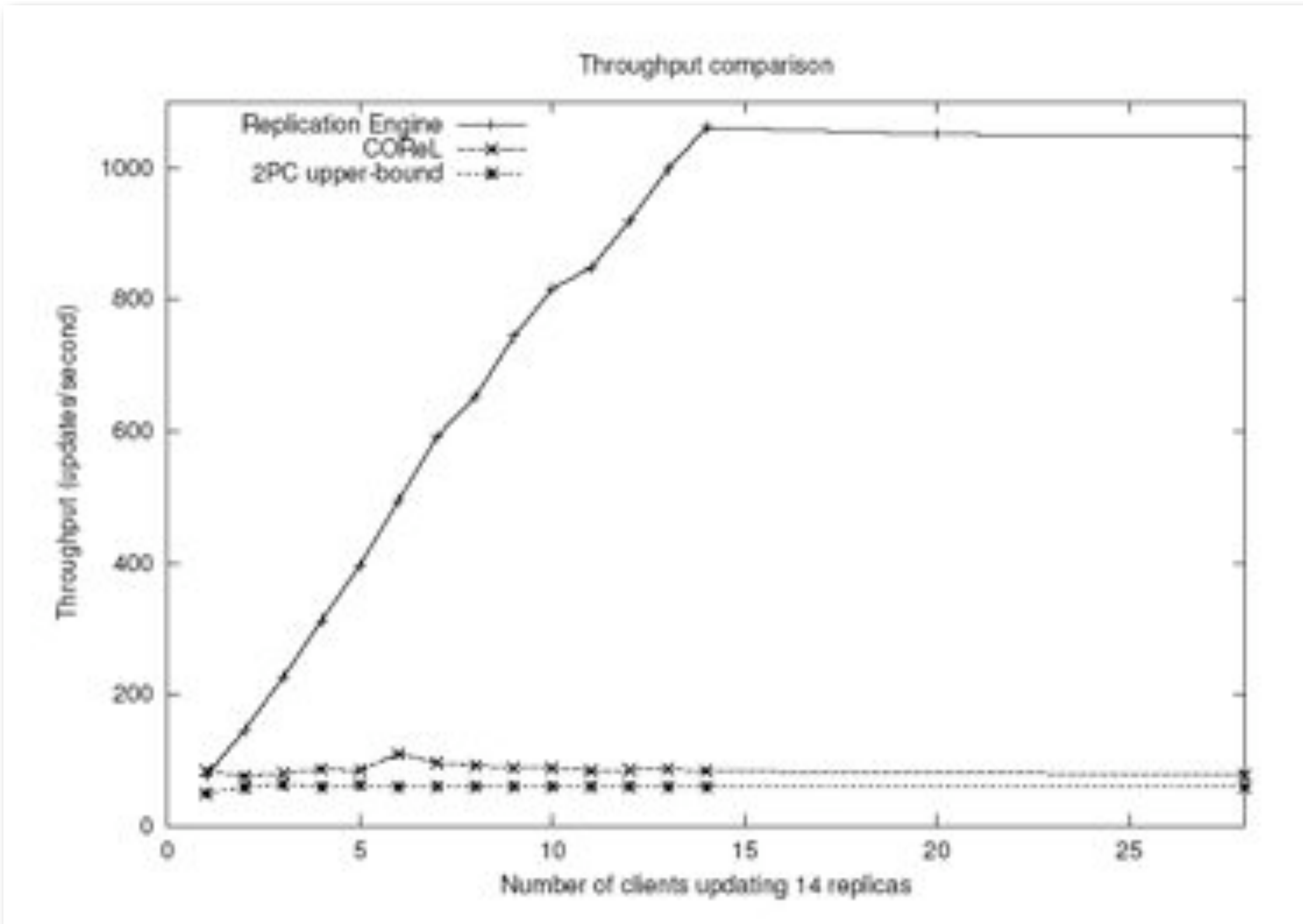
- data modification can be performed anywhere
- coordinating ACID and XA constraints is hard
  - manage full transactions
  - view consistency
  - initial synchronization
- synchronization algorithms
  - 2-phase commit (2PC)
  - 3-phase commit (3PC)

Types of Database Replication
Multi-Master

- data modification can be performed anywhere
- coordinating ACID and XA constraints is hard
  - manage full transactions
  - view consistency
  - initial synchronization
- synchronization algorithms are complex
  - 2PC and 3PC are unrealistic as N^2 handshakes must happen
  - EVS Engine
  - COReL

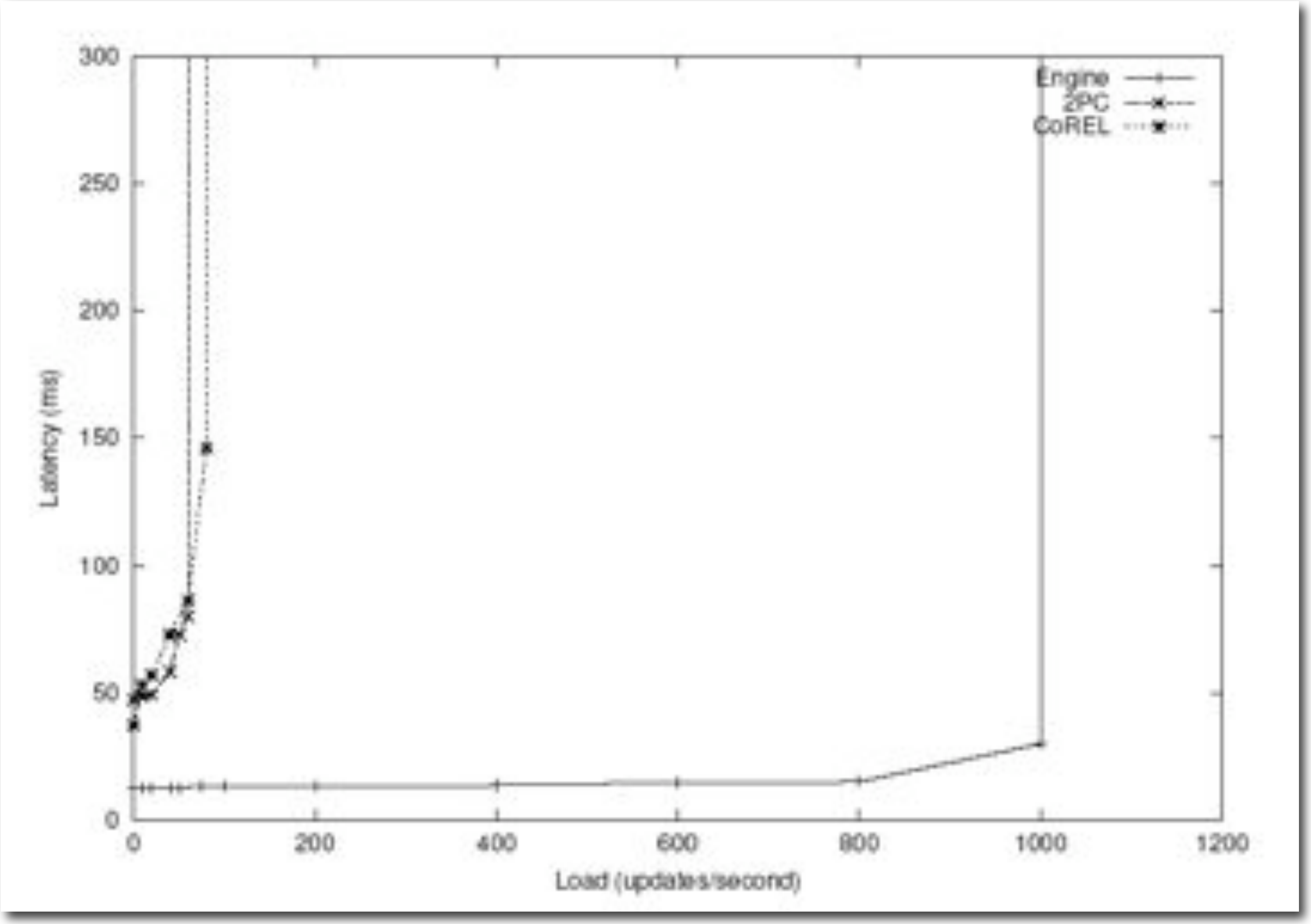
Relative Performance

http://www.cnds.jhu.edu/pub/papers/AT02_icdcs.pdf
http://www.cnds.jhu.edu/pub/papers/cnds-2002-4.pdf



Relative Performance

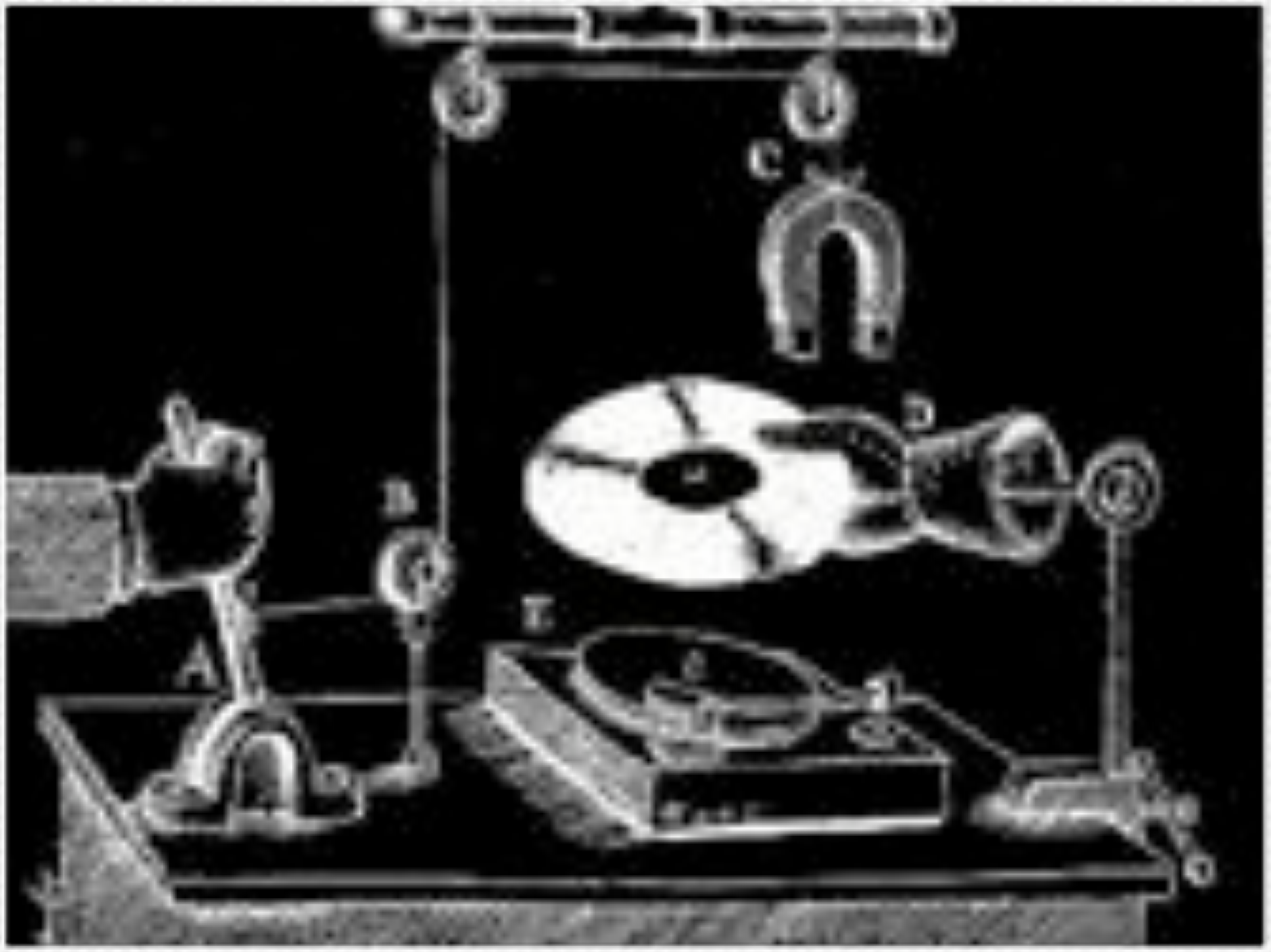
http://www.cnds.jhu.edu/pub/papers/AT02_icdcs.pdf
http://www.cnds.jhu.edu/pub/papers/cnds-2002-4.pdf



State of Affairs

News Site Revisited

- Replicate the database on each web server
  - Oracle on each web server
  - replicate the needed tables
  - certainly doesn't scale financially
- If the site used MySQL...
  - zero capital investment
  - news items don't need to be cached in PHP pages
  - legitimizes more intense queries in live site pages
  - or.. if you'd like to replicate from Oracle into MySQL
    ... go to my talk on Wednesday morning.



The Right Tool For The Job

Who’s Online?

a real-world example

- A “service” requires who’s online info
- Users that have loaded an object within x minutes
- Need to know the last page the user hit
- The “service” is exposed throughout the site

Scalability Requirements

- x = 30 (minutes)
- 5000 hits/second
- 1,000,000 concurrent users
- Queries:
  - current users online (count)
  - current users on “this” page sorted by last access (limit 30)

Let’s use a familiar tool

We use MySQL anyway

- We use MySQL to “drive” the site
- We are familiar with MySQL
- Queries are cake:
  - select count(1) from recent_hits
    where hitdate > SUBDATE(NOW(), INTERVAL 30 MINUTE)
  - select username, hitdate from recent_hits
    where url = ? and hitdate > SUBDATE(NOW(), INTERVAL 30 MINUTE)
    order by hitdate desc
    limit 30

Getting More Specific

- 1,000,000+ row table
  - assuming we sweep out stale data
- 5,000 replaces/second
  - indexes required on hitdate and url
- 10,000 queries/second
  - MySQL's query cache doesn't help at all, the updates invalidate it
  - Replaces cannot block queries
    MyISAM is not an option, we use InnoDB
  - Both queries require a full table scan!

All in addition to the existing demands of the site!

Try Some Tests

- On the development box (Idle dual Xeon, SCSI drives)
  - 1,400 replaces/second
  - 800 queries/second
- On the production box (dual Xeon, SCSI disk array w/ 1GB cache)
  - 200 replaces/second
  - 150 queries/second

ARE YOU INSANE?!

Build a Custom Tool

- We need which urls/users/timestamp tracking
  - So... we need to add an “update” to each page
  - No... that won’t catch images, let’s use a mod_perl log hook.
  - Wait... we are already writing logs, let’s aggregate passively if we use mod_log_spread, we just need to add a subscriber
- Passive aggregation
  - can handle “bursty” traffic by lagging behind a bit
  - it can’t slow down the app!
- Pick a data structure
  - Multi-Indexed Skiplist -- why?
    - Free “balancing” -- randomized
    - O(lg n) insertion, deletion, location
    - O(1) popping (for culling expired sessions)
    - it precisely meets the requirements... and I like them.

Choose a Language
I like C... so sue me.

The concept:

- Skiplist Index on:
  - username
  - url,hitdate
  - hitdate
- Single thread, event driven
- Receive messages from Spread:
  - parse username, url, hitdate
  - delete from skiplist by username O(lg n)
  - insert into skiplist O(lg n)
  - pop the end of the skiplist of any hitdate > 30 minutes O(1)
- Receive client requests
  - Cardinality query is O(1), single write()
  - Users on a url query is O(lg), O(1) to fill out iovec, single writev()

The Result

- 6 hours worth of coding and testing
- 800 lines of C code (server)
  - use libspread and libskiplist
- 40 lines of perl (client module for web app)
- On commodity hardware ($2k box)
  - ~80,000 inserts/second
    - more hits than we'll ever see!
  - ~100,000,000 counts/second -- not including write()
  - ~500,000 users for urls/second -- not including writev()

That'll do.

The Right Tool For The Job

- MySQL is a great tool
  - it is used to drive the rest of the site... spectacularly
- Using it for this project would:
  - wasted valuable resources in the architecture
  - saved a few hours of work
  - allowed you to not use your brain

This Image
Intentionally
Left Blank

Conclusion

Scalable Internet Architectures

- Building them just isn't that hard... if you
  - carefully analyze the problem at hand
  - don't make sloppy or rash technical decisions
  - always think like a computer scientist

Thank You

- OmniTI Computer Consulting, Inc.
- My biggest fan club Lisa, Zoe & Gianna

Buy the book!