Xen and the Art of Rails Deployment

Who am I?

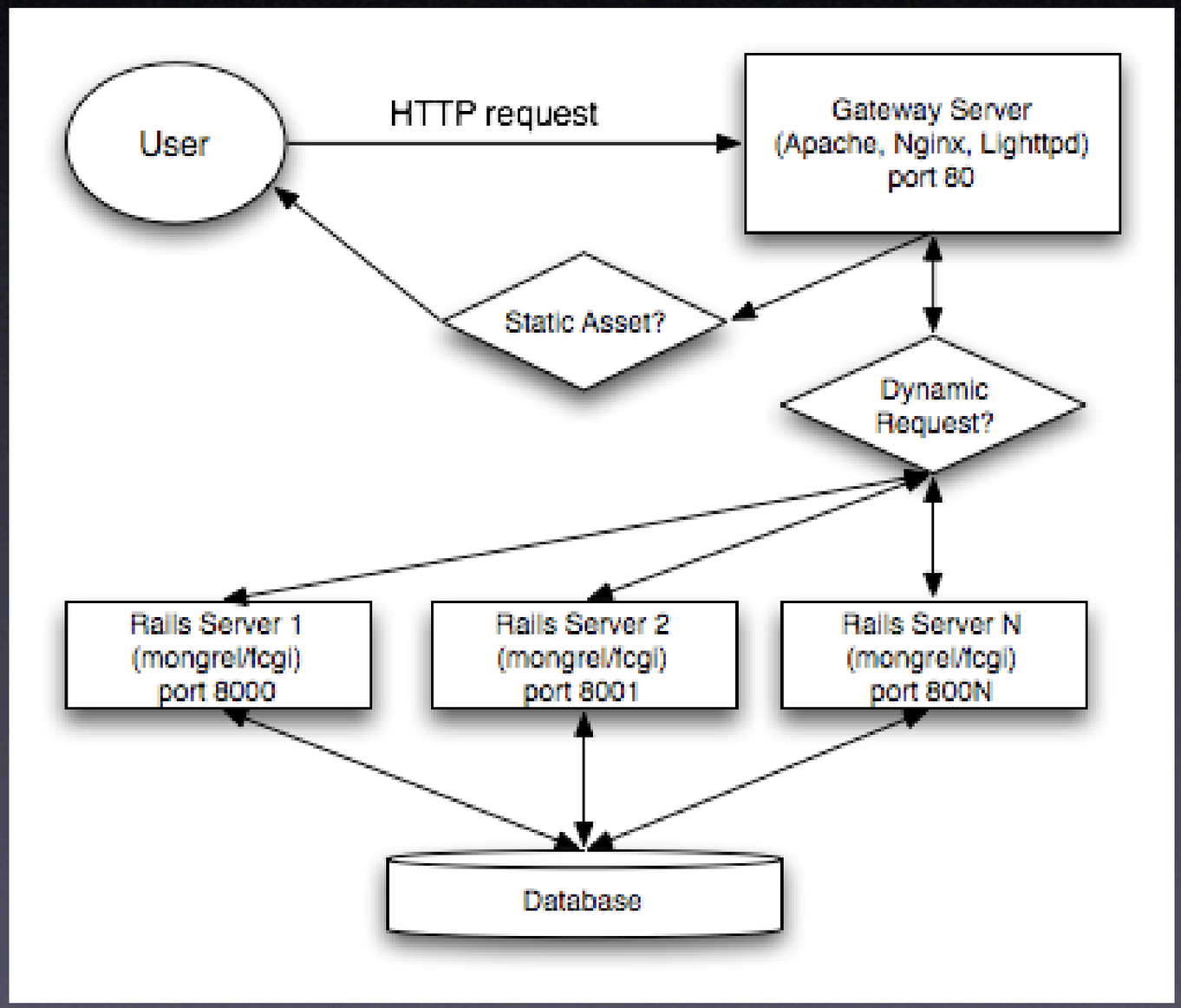
- Ezra Zygmuntowicz
- Rubyist for 4 years
- Engine Yard Founder and Architect
- Blog: http://brainspl.at

Deploying Rails

- Details have changed rapidly over the years
- Many different webservers have come and gone
- Basics remain the same

Full Stack Request/Response Life-Cycle

- Request comes into gateway server
- Rewrite rules are evaluated and request gets served directly if it's a static asset
- Dynamic requests are proxied to one Mongrel in the Mongrel Cluster
- Mongrel dispatches request through Rails and returns response to client



History Lesson

- CGI
- Apache1.3.x/mod_fastcgi
- Lighttpd/fcgi
- Apache2.x/mod_fcgid
- Lighttpd/SCGI
- Lightspeed

Enough Already

- CGI
- Apache 1.3.x/n fastcgi
- Lighttpd
- Apache mod_fcgid
- Lighttpd/SCGI
- Lightspeed

Enter Mongrel

The year of the Dog

What is Mongrel?

Mongrel is an HTTP Server Library written by Zed Shaw

- Fast HTTP Parser written in Ragel + C
- Fast URI Classifier written in C
- Stackable Request Handlers
- Flexible Configuration
- Secure and RFC Compliant HTTP Parser

Ragel State Machine Defined HTTP Parser

```
#### HTTP PROTOCOL GRAMMAR
# line endings
  CRLF = "\r\n";

# character types
  CTL = (cntrl | 127);
  safe = ("$" | "-" | "_" | ".");
  extra = ("!" | "*" | "'" | "(" | ")" | ",");
  reserved = (";" | "/" | "?" | ":" | "@" | "&" | "=" | "+");
  unsafe = (CTL | " " | "\"" | "#" | "%" | "<" | ">");
  national = any -- (alpha | digit | reserved | extra | safe | unsafe);
  unreserved = (alpha | digit | safe | extra | national);
  escape = ("%" xdigit xdigit);
  uchar = (unreserved | escape);
  pchar = (uchar | ":" | "@" | "&" | "=" | "+");
  tspecials = ("(" | ")" | "<" | ">" | "@" | "," | ";" | ":" | "\\" | "\"" | "/" | "[" | "]" | "?" | "=" | "{" | "}" | " " | "\t");

# elements
  token = (ascii -- (CTL | tspecials));

# URI schemes and absolute paths
  scheme = ( alpha | digit | "+" | "-" | "." )* ;
  absolute_uri = (scheme ":" (uchar | reserved )*);

  path = (pchar+ ( "/" pchar* )*) ;
  query = ( uchar | reserved )* %query_string ;
  param = ( pchar | "/" )* ;
  params = (param ( ";" param )*) ;
  rel_path = (path? %request_path (";" params)?) ("?" %start_query query)?;
  absolute_path = ("/"+ rel_path);

  Request_URI = ("*" | absolute_uri | absolute_path) >mark %request_uri;
  Method = (upper | digit | safe){1,20} >mark %request_method;

  http_number = (digit+ "." digit+) ;
  HTTP_Version = ("HTTP/" http_number) >mark %http_version ;
  Request_Line = (Method " " Request_URI " " HTTP_Version CRLF) ;
```

Why is Mongrel better?

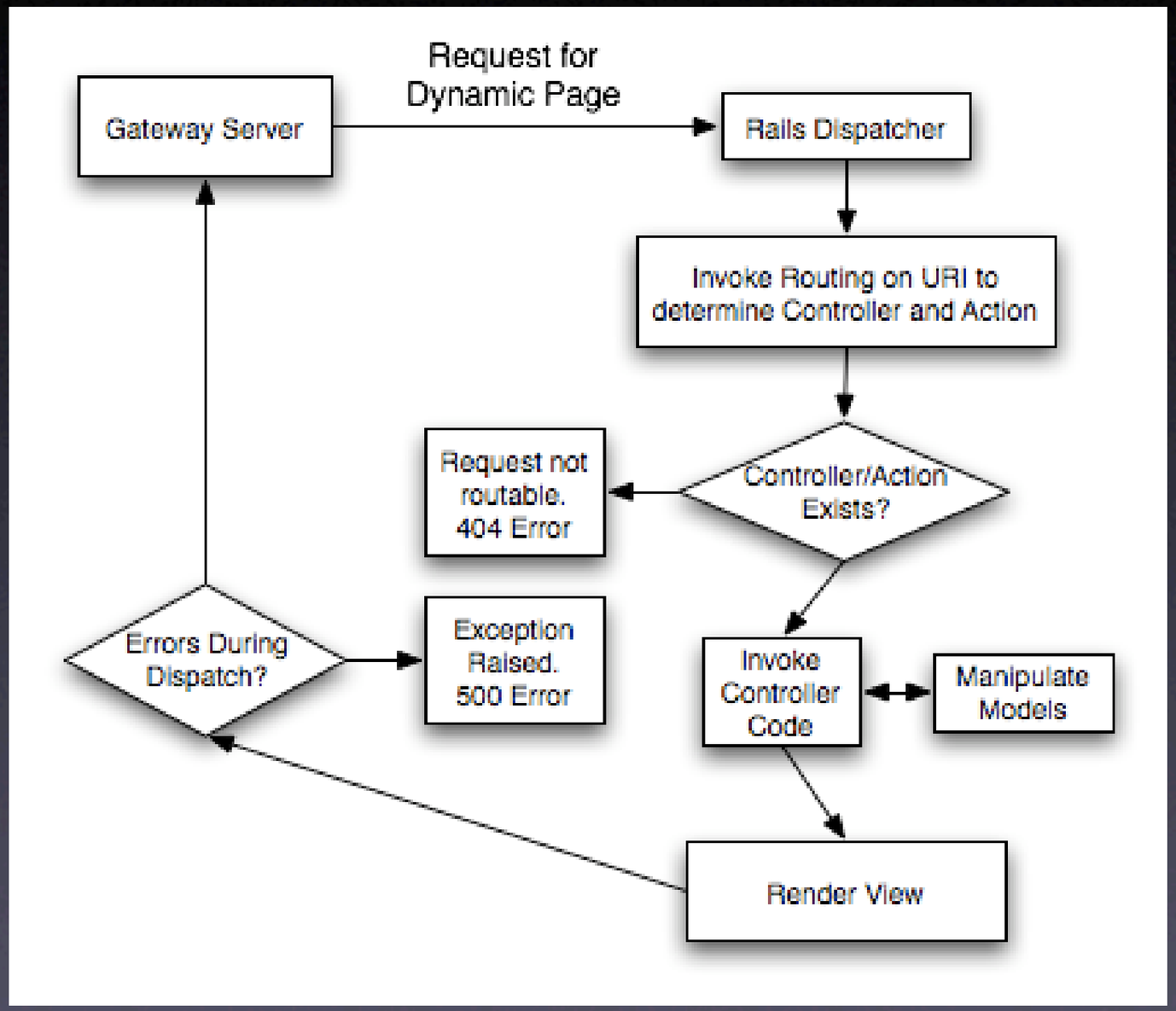
- HTTP is a well known and well tooled protocol
- Mongrel is way easier to setup and use
- Transparent wire protocol

But Rails isn't Thread Safe!

- Giant Mutex Lock around Rails Dispatch
- Only one request served at a time by one mongrel
- Use mongrel_cluster to scale with multiple processes

Rails Internal Request/Response Life-Cycle

- Mongrel Locks Mutex
- Rails Dispatcher is invoked with request/response objects
- Routing is invoked and returns the proper Controller object or 404 if no route found
- Filter chain is invoked
- Controller's Action is called, manipulates Models
- View is rendered and any after filters are called
- Mongrel Unlocks Mutex
- Final response or error page returned to client



New dog seeking old tricks

- Wide array of options for HTTP tools to front mongrel clusters
- Pen, Pound, Balance, Haproxy (No static file serving, just proxies)
- Lightspeed can serve static files and proxy to mongrel
- Apache2.2.x/mod_proxy_balancer can do the same

On the prowl for the perfect stack

- Pen(no ssl support, no connection rate limiting)
- Pound(Falls down under high load, no connection rate limiting)
- Haproxy(supports conn rate limits, very high perf, no static files so more moving parts in a full stack)
- Lightspeed(free version is crippled)
- Apache2.2.x(Does work but.. bloat, bloat, bloat...)

Nginx:
From Russia, with Love

- Seriously bent on performance
- Super small resource footprint
- Stands up under the heaviest loads without leaking memory
- Killer rewrite and proxy modules
- Approachable author and growing community

Nginx + Mongrel

- This is *the* stack to be on
- Only keep apache around for mod_dav_svn
- Flexible nginx.conf syntax allows for serving static files and rails caches and proxying dynamic requests to mongrel
- Fast, fast, fast
- Did I say it's fast yet?

A few gotchas

- Nginx buffers file uploads, so no mongrel_upload_progress. This will be addressed soon
- No connection rate limiting for proxy module yet, this too shall pass

A bright future for nginx

- mod_rewrite is going away
- To be replaced with http_script_module
- This will embed the NekoVM(http://nekovm.org/) directly in nginx so customizing behavior for rewriting and proxying will become infinitley flexible

Perfect Simple Stack

- Linux
- Nginx
- Mongrel(mongrel_cluster)
- Monit

Swiftiply:
Teaching the Dog new tricks

http://swiftiply.swiftcore.org

Swiftiply: Evented Mongrel

- Hot patch to Mongrel
- Removes Ruby's Thread's and Socket handling from Mongrel Core
- Replace with EventMachine event loop
- Mongrel becomes Single threaded, event driven
- Noticable Speed and IO throughtput increase
- Stands up much better under higher concurrent load without starting to slow down or leak memory

But how does a single threaded event driven mongrel outperform a multithreaded mongrel?

- Ruby's green threads have a lot of overhead in context switching and have to copy a lot of state context for each thread
- Mutual exclusion locks are expensive
- One process can only do so much IO
- Event driven means running in a tight loop and firing callbacks in response to network 'events'
- Since there is no context switching between threads, a single process has less overhead to deal with which allows for higher throughput and faster networking IO

Mongrel VS Evented Mongrel in a Hello World dogfight

Mongrel: 1 concurrent user

```
Concurrency Level:      1
Time taken for tests:   2.944 seconds
Complete requests:      5000
Failed requests:        0
Broken pipe errors:     0
Total transferred:      655000 bytes
HTML transferred:       55000 bytes
Requests per second:    1698.37 [#/sec] (mean)
Time per request:       0.59 [ms] (mean)
Time per request:       0.59 [ms] (mean, across all concurrent requests)
Transfer rate:          222.49 [Kbytes/sec] received
```

Evented Mongrel: 1 concurrent user

```
Concurrency Level:      1
Time taken for tests:   1.914 seconds
Complete requests:      5000
Failed requests:        0
Broken pipe errors:     0
Total transferred:      655000 bytes
HTML transferred:       55000 bytes
Requests per second:    2612.33 [#/sec] (mean)
Time per request:       0.38 [ms] (mean)
Time per request:       0.38 [ms] (mean, across all concurrent requests)
Transfer rate:          342.22 [Kbytes/sec] received
```

Mongrel:
100 concurrent
users

```
Concurrency Level:      100
Time taken for tests:   6.508 seconds
Complete requests:      5000
Failed requests:        0
Broken pipe errors:     0
Total transferred:      655156 bytes
HTML transferred:       55000 bytes
Requests per second:    768.29 [#/sec] (mean)
Time per request:       130.16 [ms] (mean)
Time per request:       1.30 [ms] (mean, across all concurrent requests)
Transfer rate:          100.67 [Kbytes/sec] received
```

Evented Mongrel:
100 concurrent
users

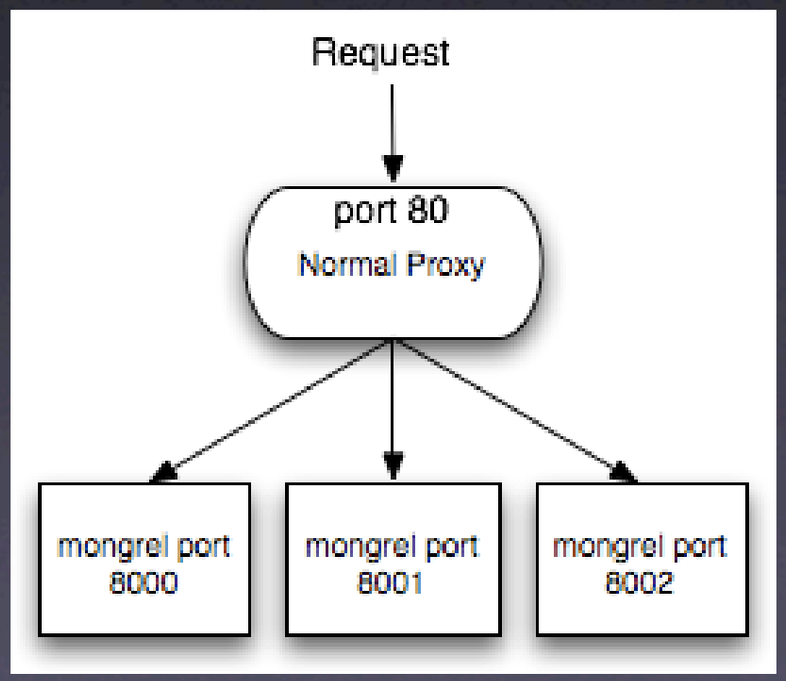
```
Concurrency Level:      100
Time taken for tests:   1.332 seconds
Complete requests:      5000
Failed requests:        0
Broken pipe errors:     0
Total transferred:      655000 bytes
HTML transferred:       55000 bytes
Requests per second:    3753.75 [#/sec] (mean)
Time per request:       26.64 [ms] (mean)
Time per request:       0.27 [ms] (mean, across all concurrent requests)
Transfer rate:          491.74 [Kbytes/sec] received
```

Swiftiply Proxy

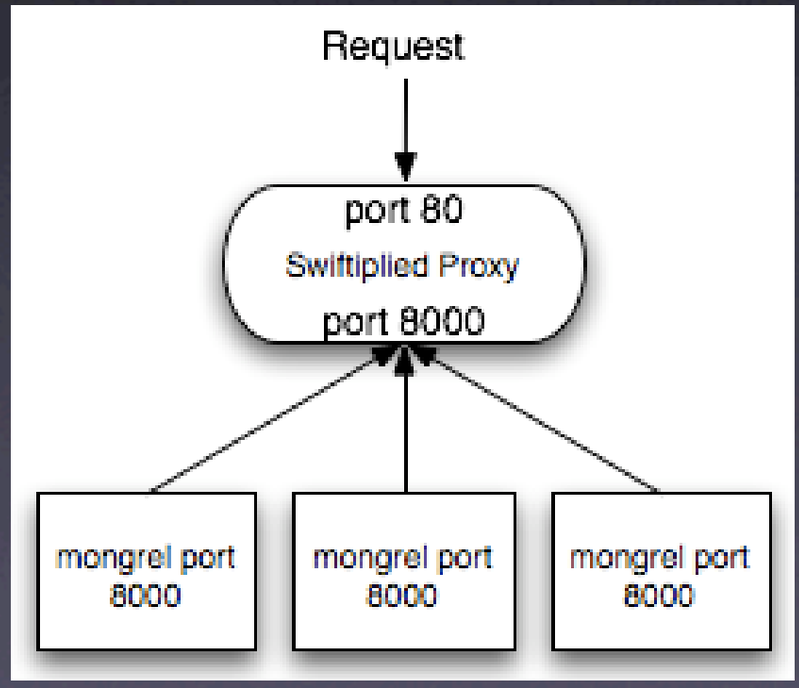
- Event driven proxy, small memory footprint(7-10Mb)
- Faster then Haproxy
- Did I mention Fast?

How it differs from a normal proxy

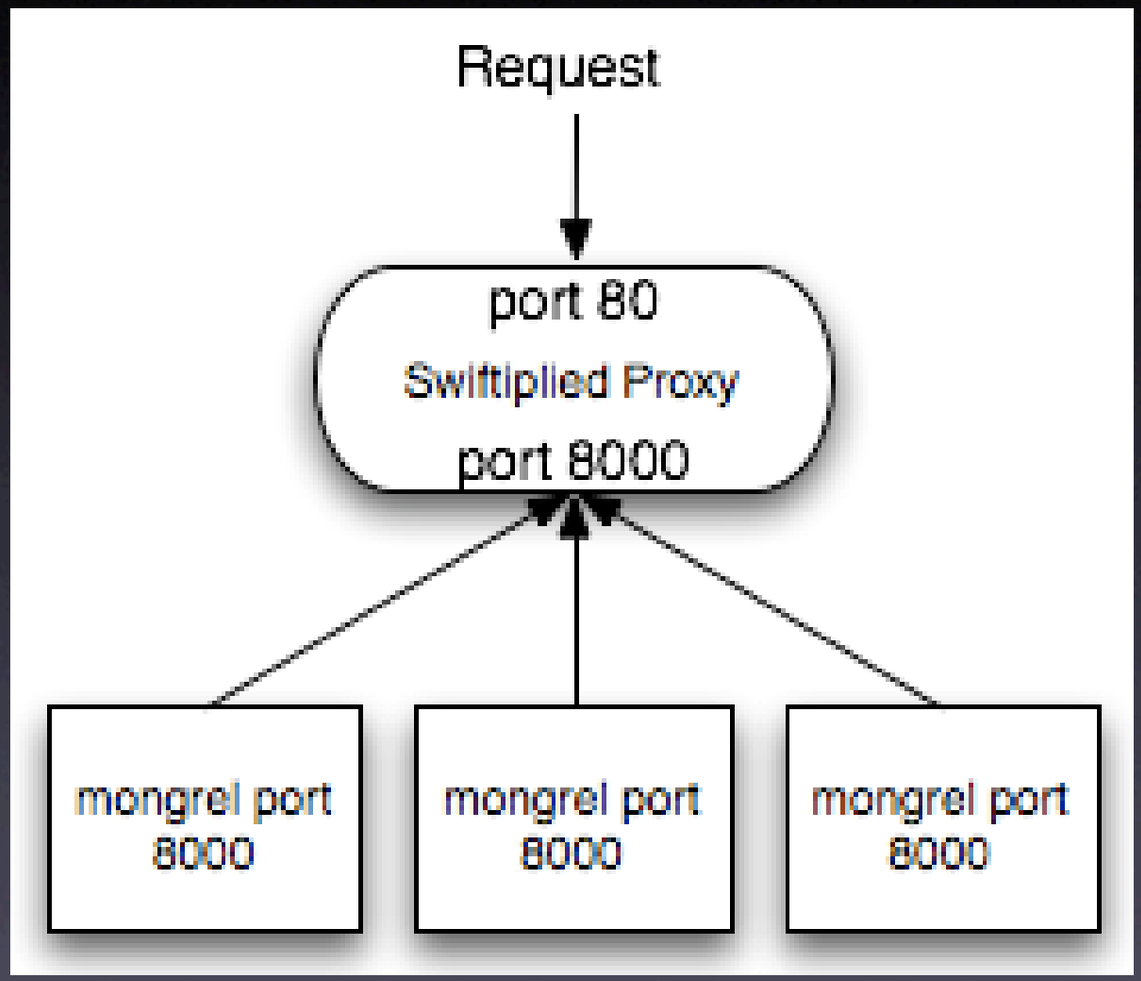
Standard proxy must know about the ports of all backends. Usually requires restart to add more backends



With swiftiply, the backends connect to the proxy. So all mongrels get started on the same port and then they open a persistent connection to the proxy



This means you can start and stop as many mongrels as you want and they get auto configured in the proxy!



This opens the door for scaling the number of mongrels automatically in response to increased load on the fly!

The Zen of Xen

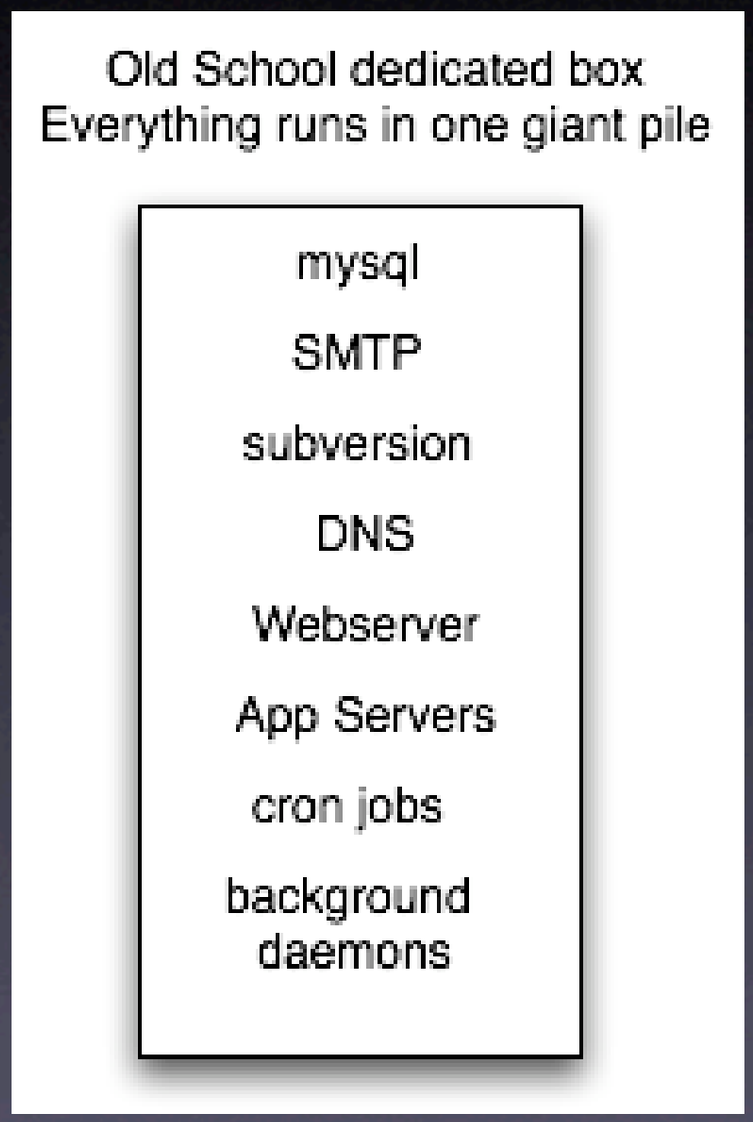
Monolithic Linux VS Modularized Linux

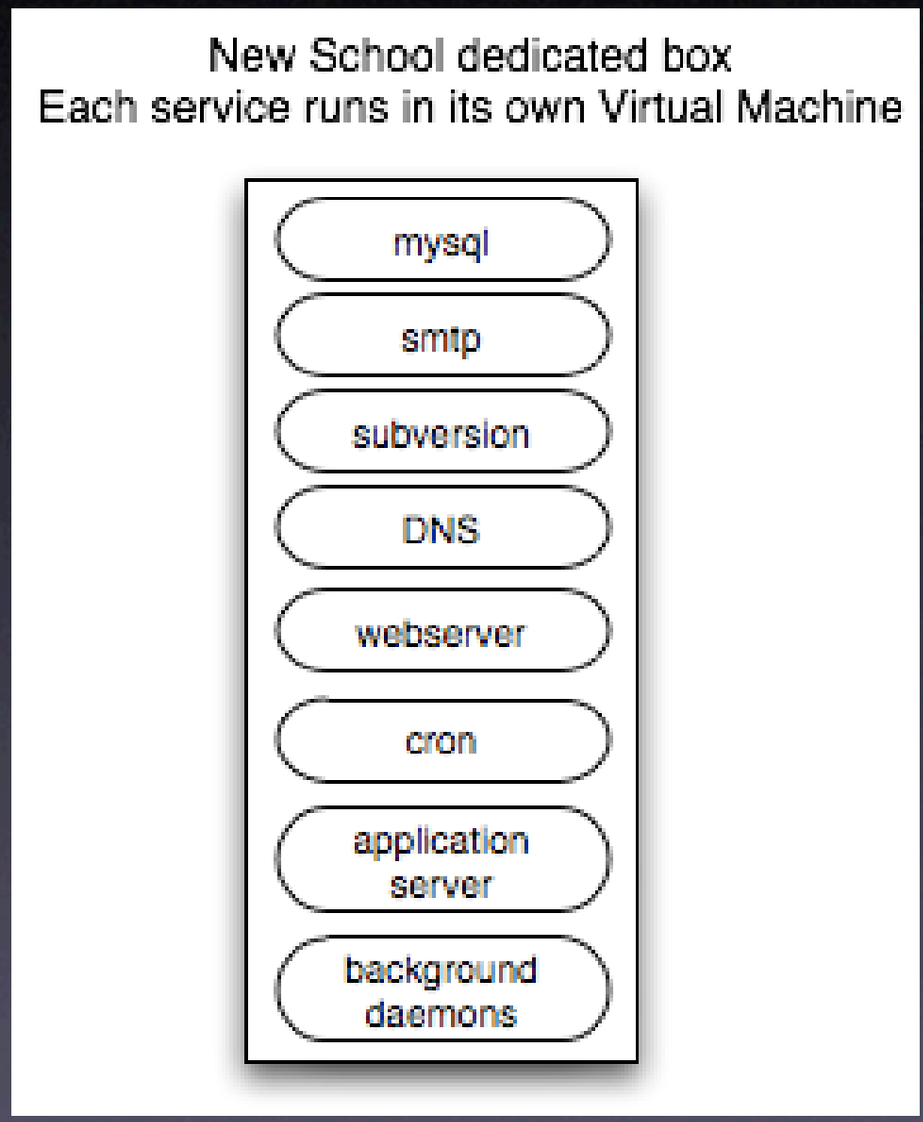
- Old way of thinking is dedicated boxes running all services in one big hodgepodge on one kernel
- New school is sharply targeted virtualized linux with each VM running a single tier or service

We all strive for code modularization right?

- Why not do the same thing with our servers?
- Each VM runs one or two related services
- Simplifies deployment and scaling
- Even if you only have one box you absolutely should run Xen on it from the start

Old VS New


Old School dedicated box
Everything runs in one giant pile
mysql
SMTP
subversion
DNS
Webserver
App Servers
cron jobs
background daemons


New School dedicated box
Each service runs in its own Virtual Machine
mysql
smtp
subversion
DNS
webserver
cron
application server
background daemons

What happens when you need to scale to more then one box?

Old School

- Get another box and move mysql on there
- Get another box to run some of the other services
- Lots of setup required, downtime to migrate
- Complex Linux installs with many services running are harder to debug when performance problems happen
- This *can* scale but is way less flexible

New School

- Add another box with Xen installed
- Pick a few services that need more resources and migrate them *live* to the other machine
- Each VM runs one thing and runs it well
- Easy to target performance problems
- Scales much better

Advanced Clustering

- Virtualized compute nodes that boot Xen dom0 off of USB thumb drives
- SAN storage for all Xen domU(VPS's)
- Red Hat Clustering Suite for fencing and cluster quorems
- GFS for 100% posix compliant clustered filesystem(no shitty NFS)
- Hardware load balancers or dedicated boxes running Ultra Monkey or just straight LVS

Fabric of Compute and Storage

- When a compute node fails just swap it out for a new one and plug in the thumbdrive and you're back in business
- Move hot VM's to less loaded nodes easily as they are not tied to a single machine
- Deploy your app code to one node and then bounce the mongrels on all nodes with a clustered filesystem like GFS
- Fragment and page caching consistency across all nodes instantly
- Scale from one or 2 VM's to as many as traffic requires *and* back down again once traffic subsides.

RAM RAM RAM

- Most Rails apps are RAM bound way before they are CPU bound
- Average mongrel size on 64bit EngineYard is 70-120Mb *per* mongrel. Slightly less on 32 bit systems
- Rmagick and :include the worst culprits
- 95% of Rails apps will leak memory at one point or another

Rails eats Database resources for breakfast

- Majority of app in the wild have *no* indexes in their databases
- Learn when and where to apply indexes, it will save your ass
- ActiveRecord insulates developers from SQL to the point of massive ineficiencies. Look at your logs and see what SQL is being generated. Do not fear the SQL and don't think you can get away without some denormalization and custom SQL if you plan on your app having a chance of scaling

Other tips & tricks

- *Don't* use filesystem sessions, AR or SQLSession or memcached if you don't need persistance
- script/runner is massively ineficient. Try as hard as possible to not load all of rails in your background processes. Use the raw Mysql library and plain ruby if you can and your servers will thank you for it
- *Do not* use script runner to process incoming email. Run a daemon in a loop and poll a mail server with net/pop2 or net/imap. Forking a whole rails process for each incoming email will never work in a production environment period

Rails is great for the 80/20 rule

- But you are on your own when you need the last 20%
- Learn how to write custom mongrel handlers for perf critical sections of your app
- When is optimization not premature?
- Ruby is plenty fast, it's rails that tends to be on the slow side
- Cache, cache, cache. It doesnt get much faster then service cached static html files

Parting Thought

- Don't take what I or anyone else says about this stuff as gospel
- Test it and benchmark it for yourself to be sure
- Trust but verify and you will stay in good shape

Questions?