CS290b – Lecture 7
Server Architecture

Scalable Internet Services and Systems, Fall 2013

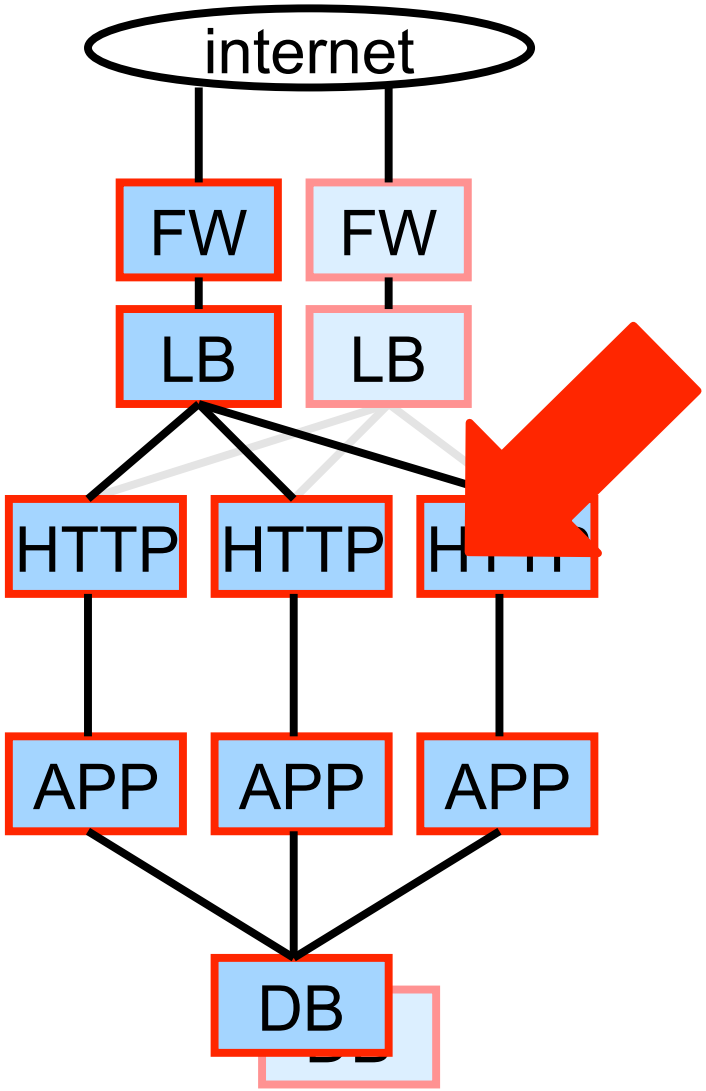
Jon Walker
Department of Computer Science
University of California at Santa Barbara

For today…

- HTTP Server Architecture
- App Servers
- Announcements

How to build scalable web services in one slide

- **Simple, share-nothing web stack**
  - Network devices – routers, firewall, load balancer, DNS, global LB, fail-over, …
  - HTTP server – thread/process model, HTTP protocol, caching, encryption (SSL)
  - Single threaded app server – model-view-controller programming model, RoR, concurrency, atomicity, clustering, caching, threading
  - Relational database – SQL, replication, redundancy, disaster recovery, partitioning



Web Servers

- What do we care about?
- What are the design considerations for a web server?

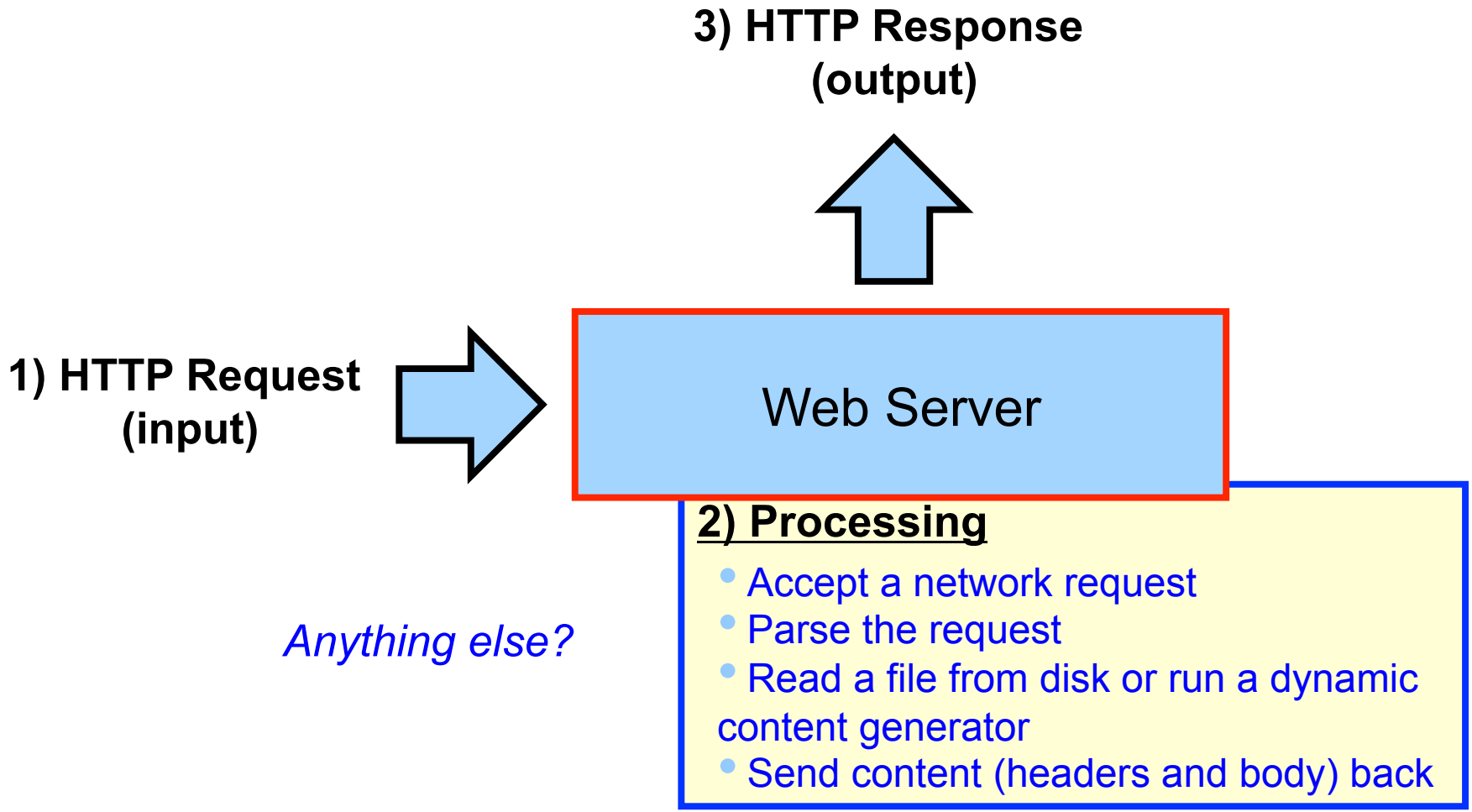
Server design considerations

- **Performance**
  - Responsiveness/Latency
  - Throughput
  - Concurrency
    - Parallelism & synchronization
  - Resources per request
    - CPU, memory, network, file desc.
  - Robustness
    - Graceful degradation
- **Ease of Development**
  - Implementation complexity
  - Plugin Architecture

Server architectures

- Single threaded
- Process per request
- Thread per request
- Process/thread worker pool
- Event driven
- Some combination of the above?

Basic HTTP Server operation



Request handling phases

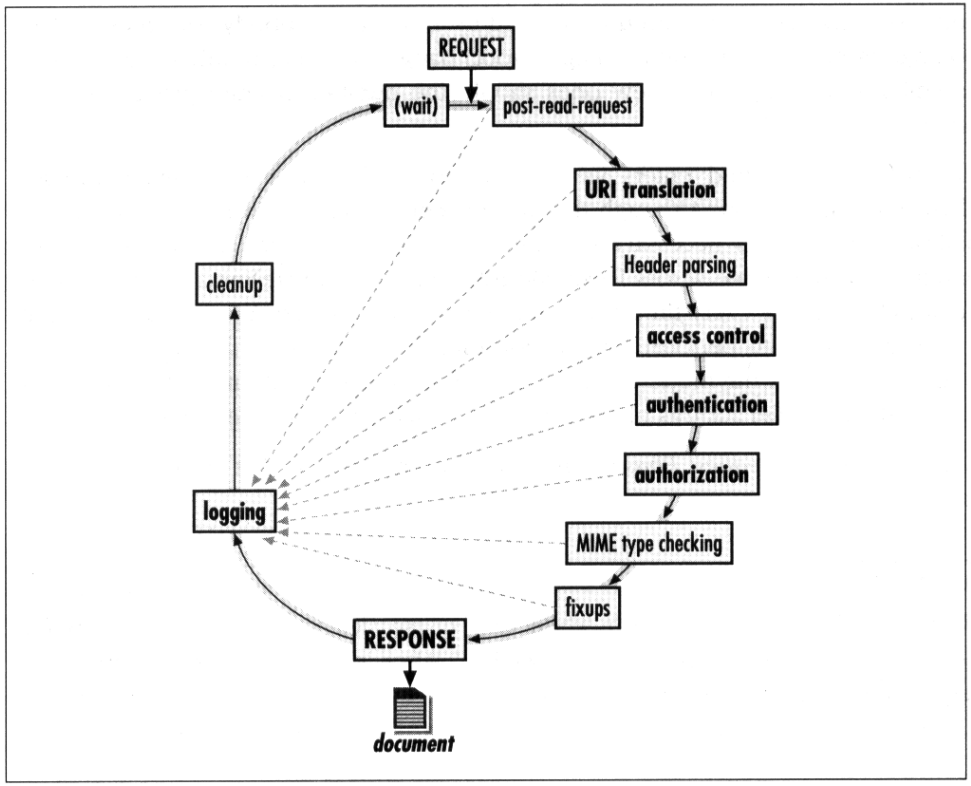


Figure 3-3. The Apache request. The main transaction path is shown in black, and the path taken when a handler returns an error is shown in gray. Phases that you are most likely to write handlers for are shown in bold.

- What is it for? (URI xlat)
  - Alias, files, etc…
- Where is it from? (access ctrl)
  - E.g. source IP based
- Who is it from? (authent.)
- Who is allowed? (authoriz.)
  - /etc/passwd, DB, …
- What is the type? (MIME chk)
  - File ext, directory, etc…
- Who will generate response?
  - Content-handlers
- Who will log response?
  - Log file, DB, …
- Who will clean-up?

HTTP Server operation

- **Server loop**
  - Bind `int bind(int s, struct sockaddr *name, int namelen);`
  - Listen `int listen(int s, int backlog);`
  - Forever *(loop over connections)*
    - Accept `int accept(int s, struct sockaddr *addr, socklen_t *addrlen);`
    - While !EOF *(loop over requests)*
      - Read
      - *Process request*
      - Write
    - Close

What is "EOF"?

What happens when a 2nd request comes in while one is processing?

Process per request

- **Master:**
  - Loop => Accept => Fork (, exec)
- **Child:**
  - Read request
  - Write response
  - Close, exit
- **e.g. inetd**
  - Performs bind, listen, accept
  - Forks & execs executable per /etc/inetd.conf
    - `ftp stream tcp6 nowait root /usr/sbin/in.ftpd in.ftpd`
- **Apache 1.x “prefork” model**
  - Variant on above model

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

Apache process worker pool

- **Aka. “pre-forking server”**
- **Processes**
  - “master process” + N child processes
  - Children listen+accept+serve
    - Max M times, M is configurable
  - Master
    - forks new children when too few are idle
    - Kills children if too many are idle
  - Communication via table in shared memory
- **Modules live in each process**
  - Same lifetime constraints as host process
  - Same communication constraints
  - Some modules do IPC to external process

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

Thread per request

- **Same as "process per request"**
  - But all in one address space
  - Very common model for Java app servers

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

Apache 2.x

- **Apache 2.x MPM “pre-fork” is a variation of thread per request**
  - Native process/thread implementation
  - Better build environment (autoconf)
- **Also supports hybrid multithreaded & multiprocess**
  - Processes isolate faults
  - Threads reduce resource requirements

Multiprocess & multithreaded

- **"MPM worker" module**
  - Fixed `ThreadsPerChild` threads per process
  - Start with `StartServers` processes
  - Maintain between `MinSpareThreads` and `MaxSpareThreads` by creating/terminating processes
  - Hard bounds: `MaxClients` threads and `ServerLimit` processes
- **Implications on applications?**
  - E.g. mod_perl, mod_php, …

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

Single-threaded, non-block I/O

- **Loop:**
  - Select on accept
    - Accept connection
  - Select on read (all socks with expected input)
    - Read request
    - Mark ready when request complete
  - Forall ready requests
    - Handle
  - Select on write (all socks for which there's output)
    - Write response
    - Close when done
- **Select & poll semantics/performance**
  - Select: bit-set, must loop through bitset to find active sockets
  - Poll: array of file descriptor numbers that are ready
  - Epoll: triggered events on "watched" file descriptors

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

Event-driven

- **Loop**
  - Same as for “single-threaded, non-blocking I/O”
  - But more general event mechanism
- **Events**
  - All I/O is non-blocking (network, IPC, disk, …)
  - I/O completion generates event
  - Modules written as event-handlers
  - Example: serve static file
    - Request-start event
    - Request-read-ready event
    - Disk-read-ready event
    - Response-write-ready event

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

Threads vs. Events

- **Threads**
  - Many potential points of contention
  - Straight-line code
- **Events**
  - Few well-known points of suspension
  - More code fragments
  - Can leads to complex “request state” descriptors:

```
Event_handler(request_record, event_descriptor);
```

Issues:

- Responsiveness
- Throughput
- Concurrency
- Resources
- Robustness
- Complexity
- Plug-ins

C10K Problem

- **Web servers should be able to handle 10K simultaneous connections**
- **1000MHz machine, 2GB RAM, 1000Mbit/sec ethernet card for $1000 (or less)**
- **10000 clients**
  - 100KHz, 200Kbytes, 100Kbits/second
  - Shouldn't we be able to move 4Kb from disk to network once a second for these clients
- **http://www.kegel.com/c10k.html**
- **Introduced over 10 years ago – now C1M**
  - http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-1

Event Driven Server Explained

- From a talk by Simon Willison about node.js
- Using bunnies!

YourWeb Server
(using a bunny)

Source: SimonWillison

YourWeb Server
(using a bunny)

Single threaded (one bunny), so can only handle one request at a time

Happy hamster

Source: SimonWillison

Impatient hamsters

(The hamsters are using web browsers to visit your site)

YourWeb Server (using a bunny)

Source: SimonWillison

5 bunnies = can handle 5 requests at the same time

YourWeb Server (using threads or processes, aka bunnies)

Happy hamsters

Source: SimonWillison

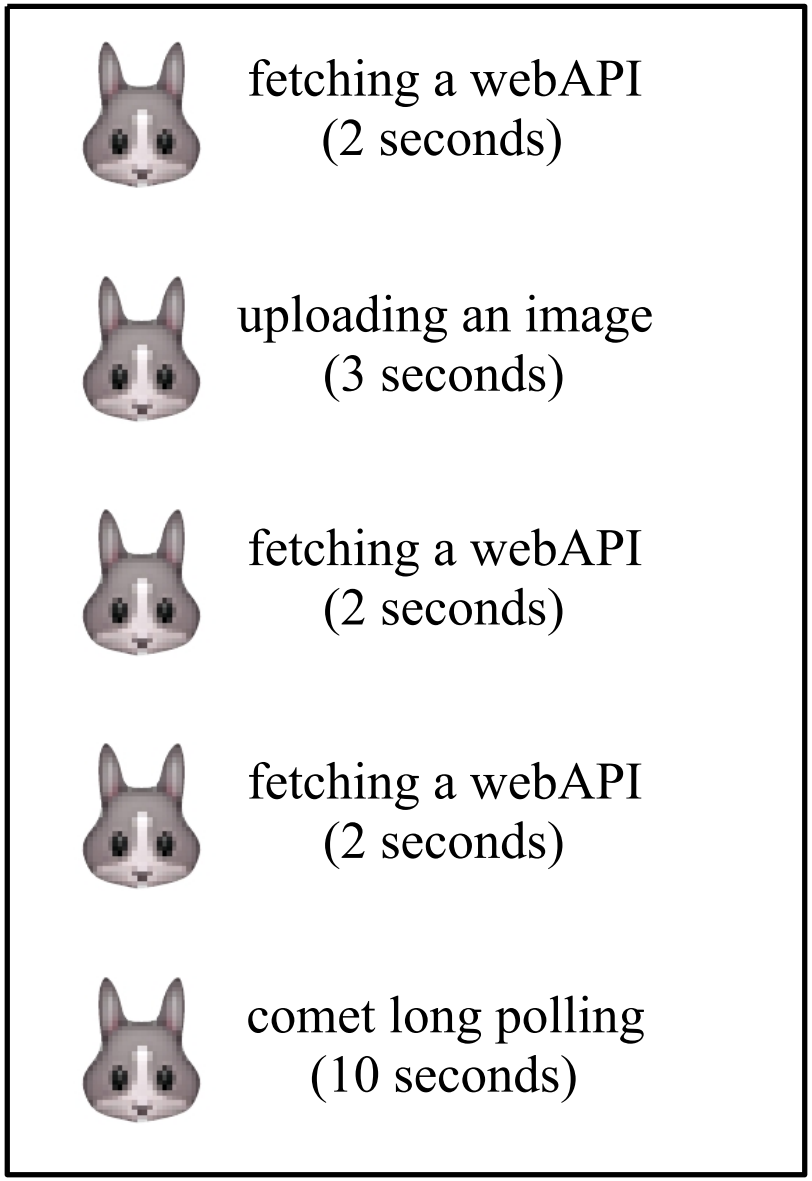
Long running operations cause a thread to block, causing requests to build up in a queue

fetching a webAPI
(2 seconds)

YourWeb Server
(using threads
or processes,
aka bunnies)

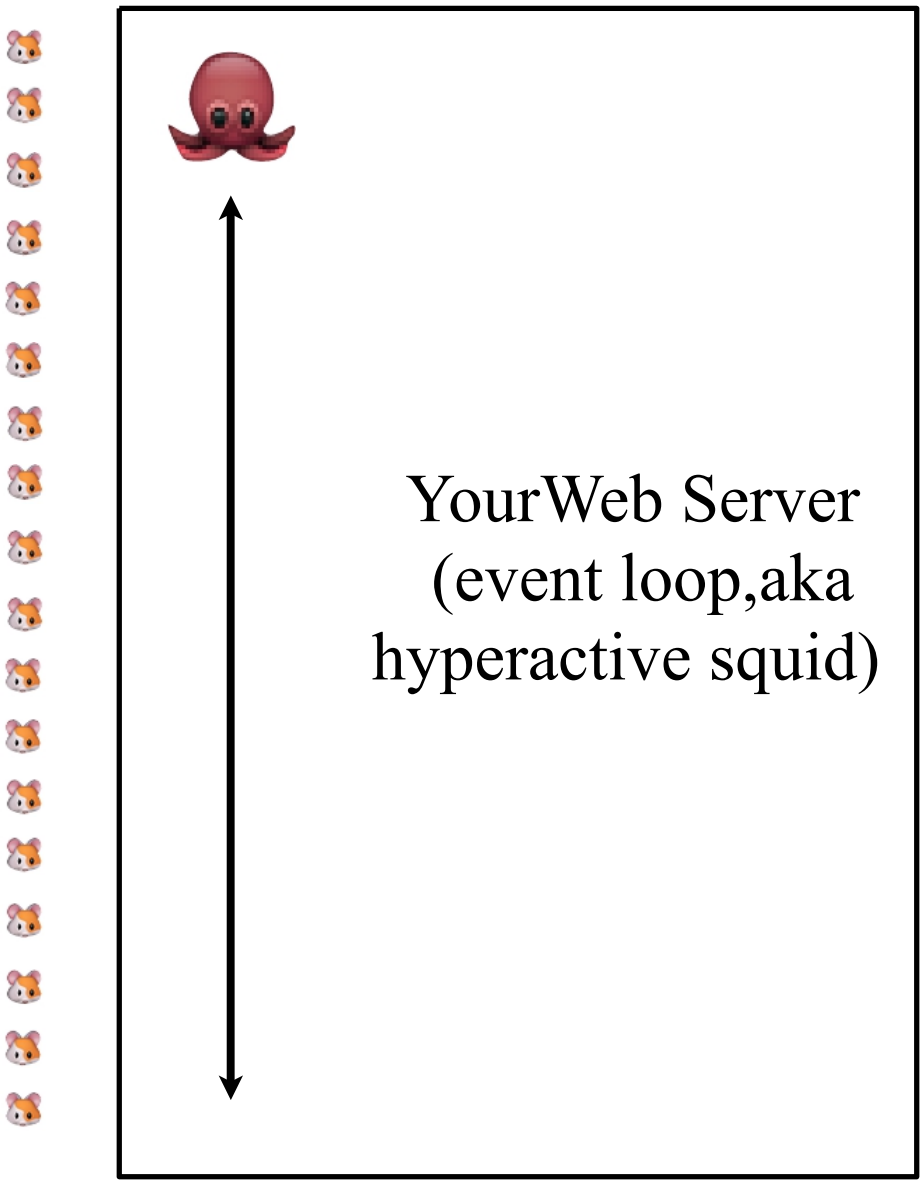
Source: SimonWillison

Impatient hamsters



Source: SimonWillison

Replace the bunnies with a single hyperactive squid.The squid runs up and down as fast as it can dealing with each hamster in turn.It fires off any long running I/O operations and then moves on to the next hamster.When the I/O operation reports progress,it does a little more work on behalf of the corresponding hamster.



Source: SimonWillison

Bad code

- rows = database.fetch(category = 'news')
- template = read_file('homepage.html')
- json = fetch_url('http://.../')

These functions block and wait for results - blocking the squid and causing the entire event loop (and hence server) to pause until they complete

Source: SimonWillison

Good code

- database.fetch(category = 'news',callback)
  - read_file('homepage.html',callback)
    - fetch_url('http://.../',callback)

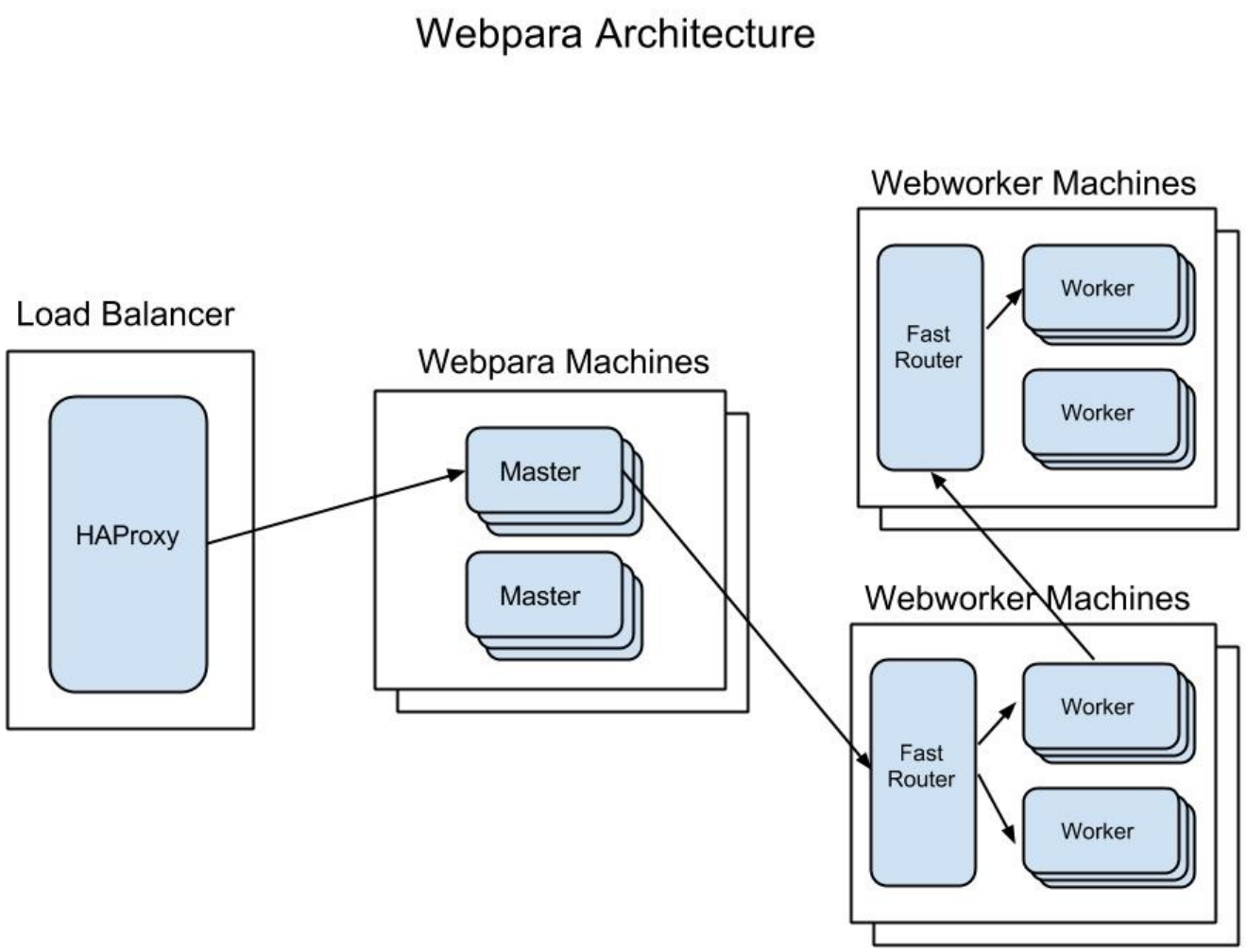
These functions specify a callback to be executed as soon as the I/O operation completes

Source: SimonWillison

Is this still a problem today?

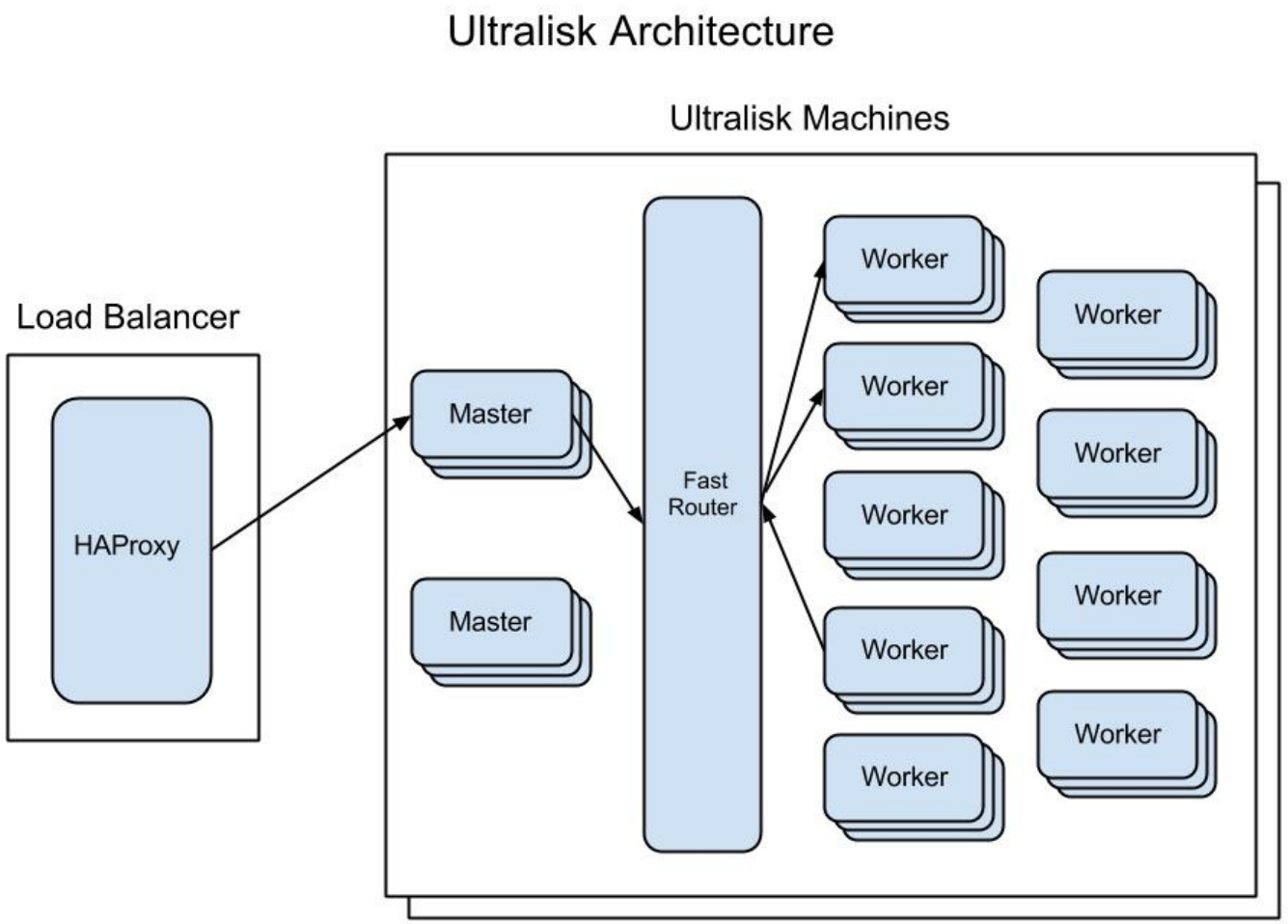
Source: http://www.modrails.com/

Quora Architecture – 2012



Source: http://www.modrails.com/

Quora Architecture – Today



Source: http://www.modrails.com/

Server architectures

- Single threaded
- Process per request
- Thread per request
- Process/thread worker pool
- Event driven
- Some combination of the above?

Server design considerations

- **Performance**
  - Responsiveness/Latency
  - Throughput
  - Concurrency
    - Parallelism & synchronization
  - Resources per request
    - CPU, memory, network, file desc.
  - Robustness
    - Graceful degradation
- **Ease of Development**
  - Implementation complexity
  - Plugin Architecture

Why do we need extensibility?

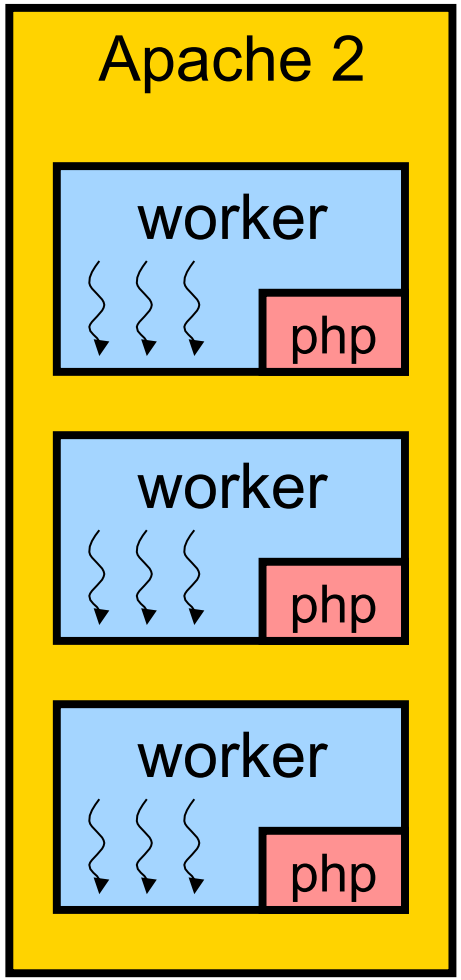
- **Apache for example has many plug-in modules**
  - Multiple multiprocessing/multithreading modules
  - Extensive URI rewriting
  - Authentication/authorization
  - HTTPS/SSL/TLS
  - Proxying requests, including load balancing
  - Connectors to external servers, CGI, FastCGI, …
  - **Connectors to Ruby/Java app servers, etc.**
  - Virtual hosts

Plug-in architecture options

- **CGI**
  - Original CGI
    - Fork a process for every request
    - Pass HTTP headers & processed info in ENV variables
  - FastCGI
    - Long lived process & connections
    - Complex custom protocol to forward requests
    - Supports more than just content handlers (e.g. auth, …)
    - Abandoned for years, revived by Rails (and abandoned again)
  - SCGI
    - Recent attempt at replacing FastCGI with something simpler
    - Long lived process, simple forwarding of requests (conn/req)
- **In-memory modules**
  - Best performance
  - Custom to web server (and server version)
  - Suited to extend core functionality, not build apps!

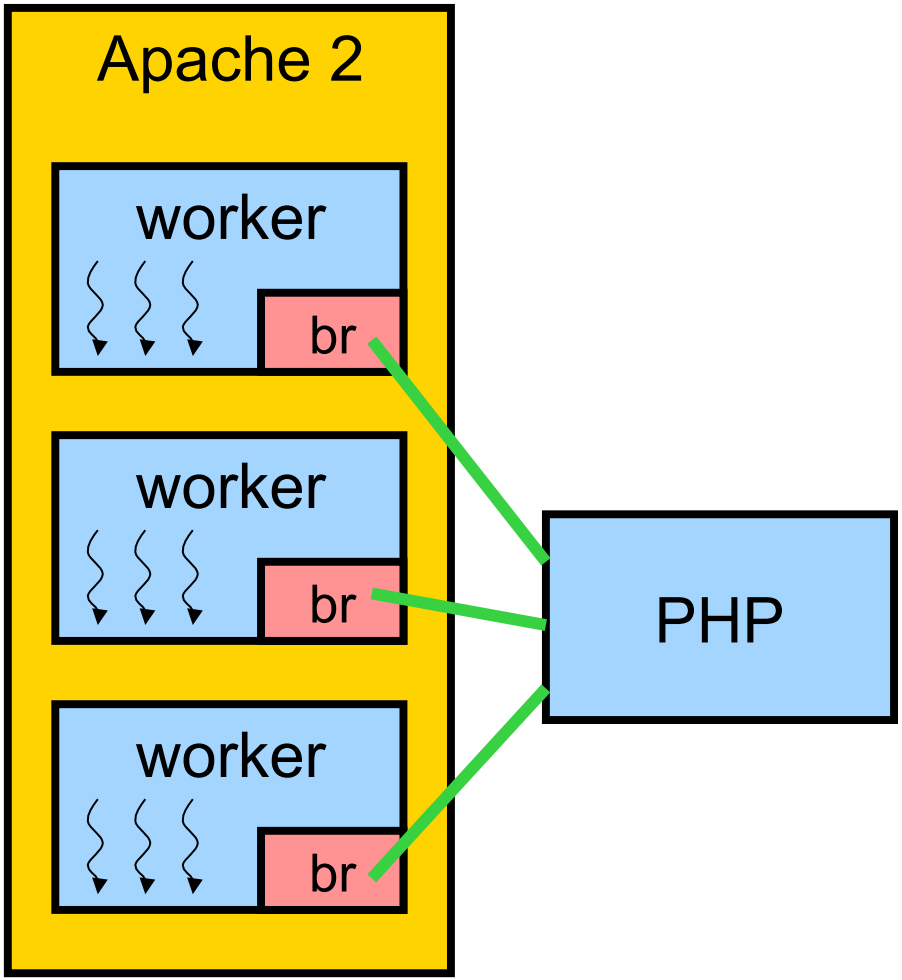
Plug-in Arch: In-Memory

- **In-memory interpreters**
  - mod_perl, mod_php, mod_ruby
- **Loads and loads of problems…**
  - Multiple users
  - Multiple applications
  - Threading models
  - Multiple processes
  - Stability
  - Memory footprint
- **Still widely used**



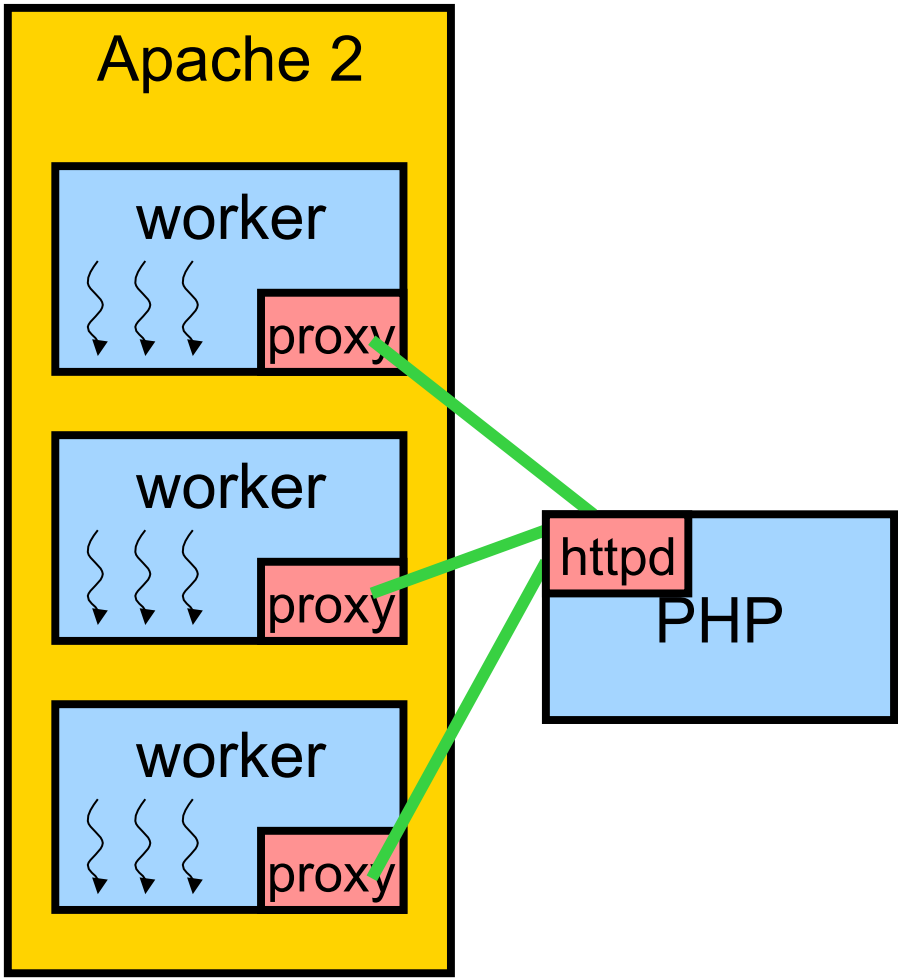
Plug-in Arch: “Bridging”

- **Bridges**
  - External application server process
  - Custom web server module
  - Custom comm protocol
  - Connection via
    - Unix sockets
    - TCP connections
    - Shared memory
- **Pros and cons**
  - Isolation
  - Custom



Plug-in Arch: Proxying

- **Similar to bridging, but:**
  - Use standard reverse-proxy module in web server
  - Use HTTP protocol
- **Advantages**
  - Standard
  - Tools (s/w & h/w)
  - Troubleshooting
- **Disadvantages**
  - Lack of info about orig request
  - Other?



Other HTTP Servers

NGINX

- **Original motivation to solve the C10K problem**
  - Russian open source web server (now commercial)
  - Hosts 7.67% of all domains worldwide
- **Single-threaded, event driven**
  - Written in C
  - Can run N instances in parallel for multiprocessors (*N* ~ # cores)
- **Uses small and *predictable* amounts of memory under load**
- **Less features than Apache**
  - Has compile time module based plugin architecture

Tengine

- **Originated by Taobao**
  - Large e-commerce site - Amazon of China
- **Based on Nginx**
  - Fully compatible
  - Better logging
  - Health checks of upstream servers
  - Etc…
- **Adds many apache type features**
  - Dynamic module support
  - Control over worker processes and CPU affinity
  - Write modules in Lua
  - Etc…

LightTPD aka. “Lighty”

- **Original motivation PHP support**
  - Written as a distraction from a PhD thesis ☺
- **Single-threaded, non-blocking I/O**
  - Uses epoll, if available
  - Can run N instances in parallel for multiprocessors (*N* ~ # cores)
- **Request handling stages:**
  - accept connection
  - read the request header
  - set up the request-handling filter-chain
    - authentication, rewriting,
  - forward request content to the backend
    - cgi, fastcgi, …
  - read the backend response header
  - set up the response-handling filter-chain
    - logging, compression, …
  - forward the response content to the client
  - on keep-alive go to 2 otherwise close the connection

With load balancing

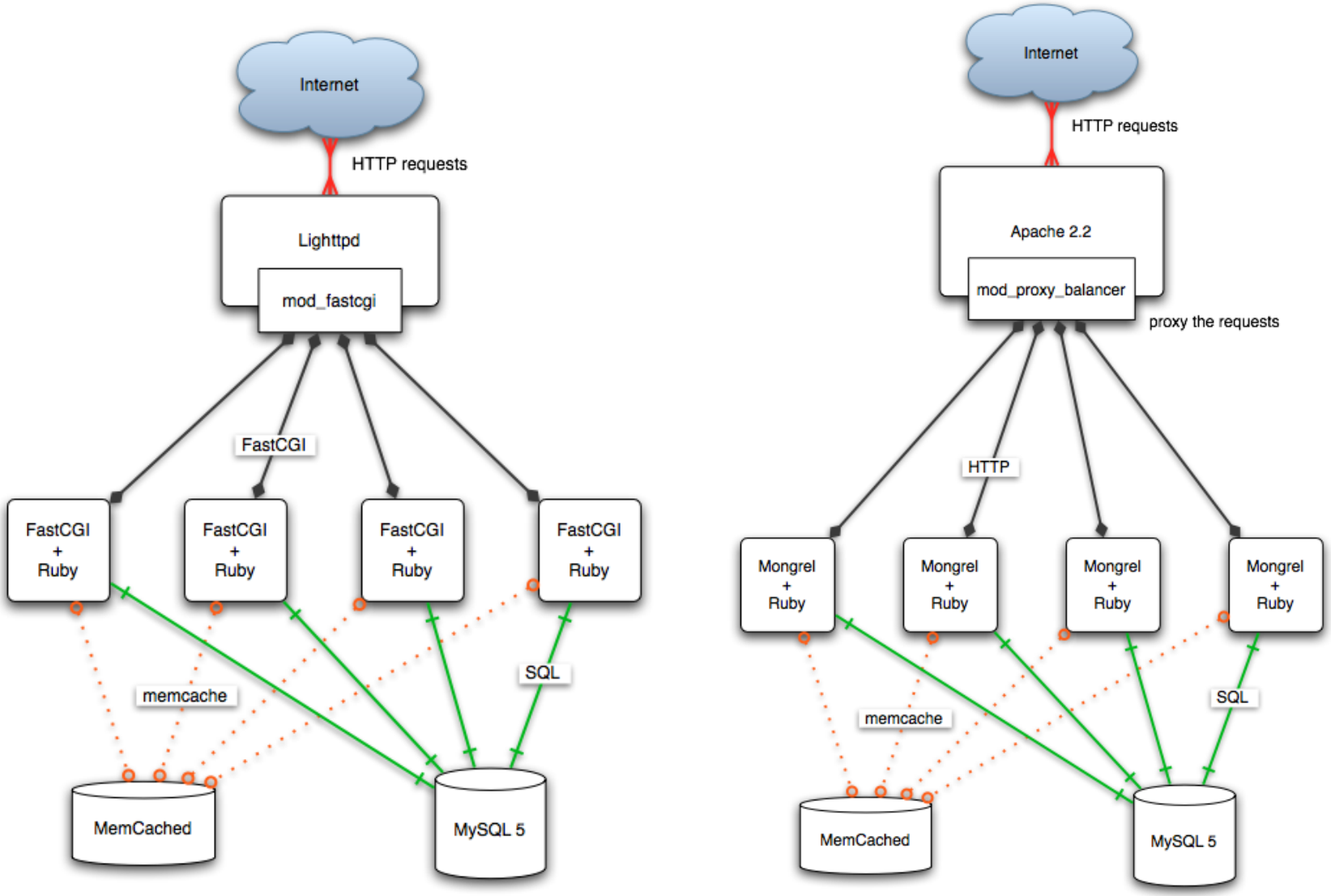
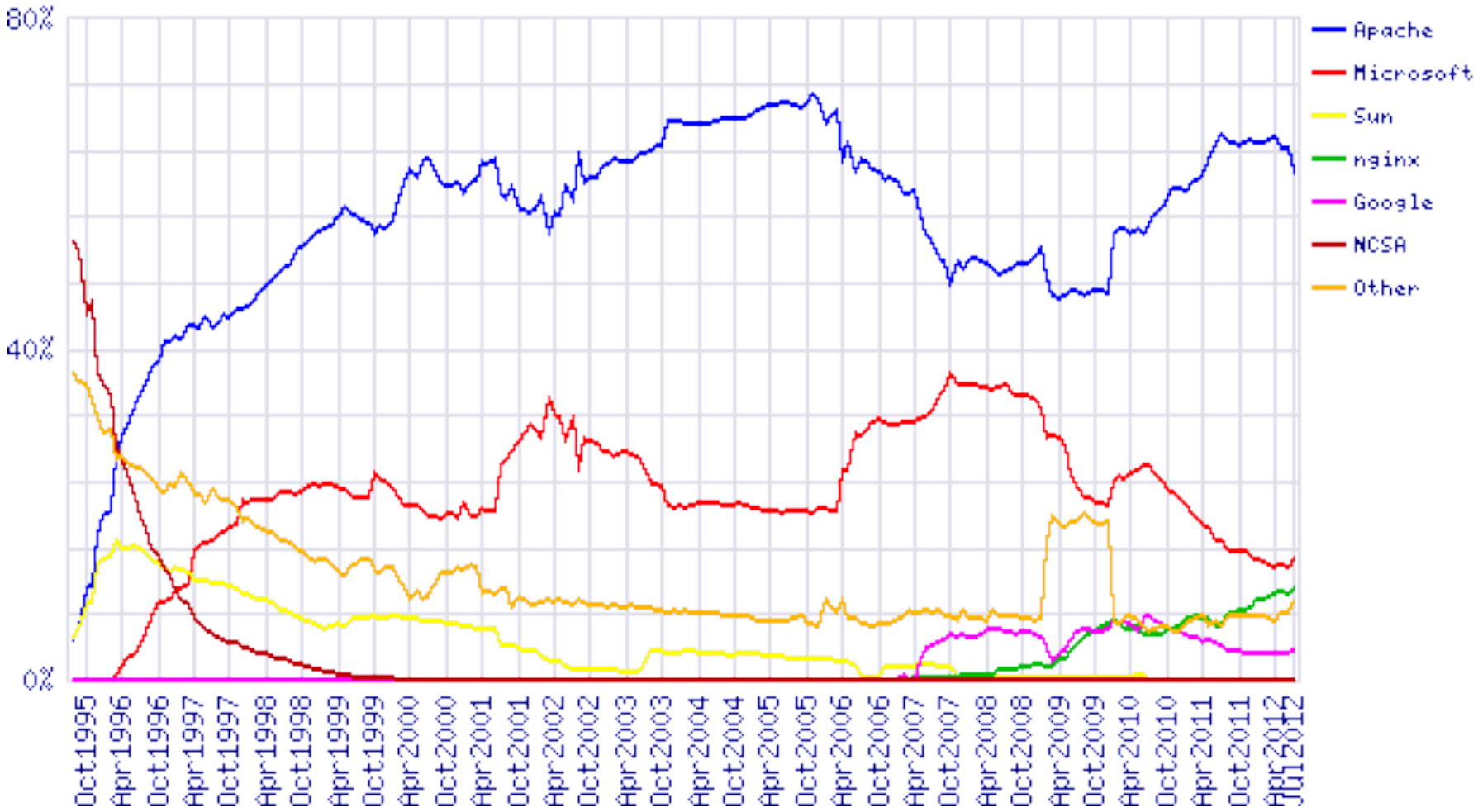


ımage from Jonathan Weiss' blog

Web Server Market Share

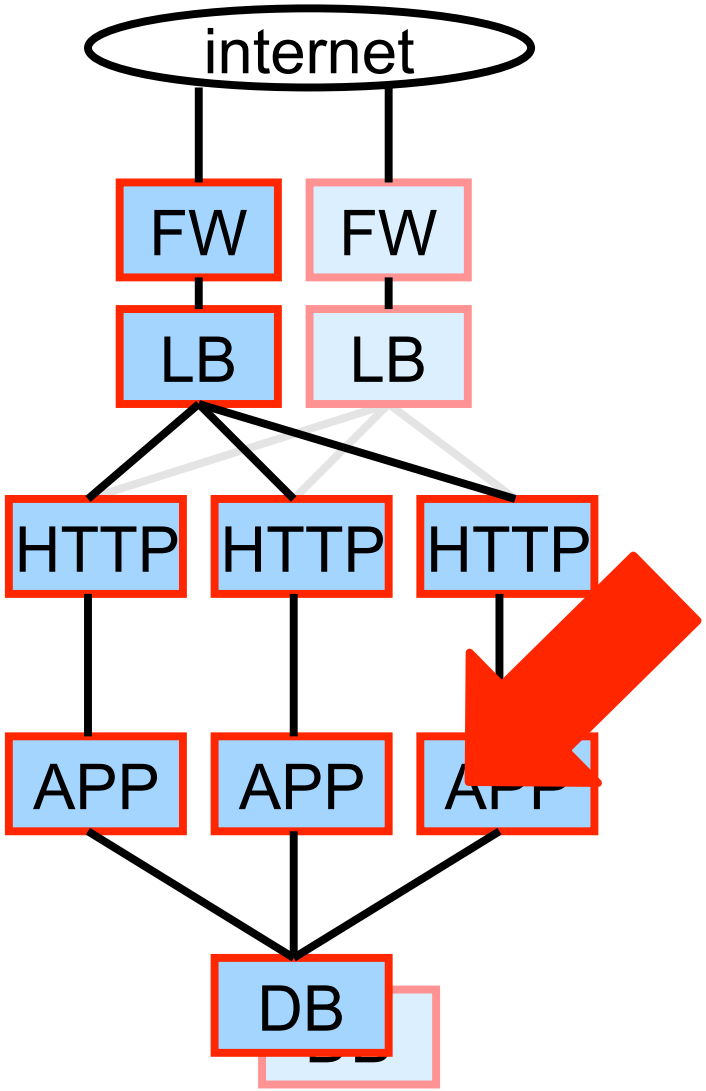


Source: netcraft.com

Questions about HTTP Servers?

How to build scalable web services in one slide

- **Simple, share-nothing web stack**
  - Network devices – routers, firewall, load balancer, DNS, global LB, fail-over, …
  - HTTP server – thread/process model, HTTP protocol, caching, encryption (SSL)
  - Single threaded app server – model-view-controller programming model, RoR, concurrency, atomicity, clustering, caching, threading
  - Relational database – SQL, replication, redundancy, disaster recovery, partitioning



Ruby Application Servers

- **Same design, architecture, issues as HTTP servers**
- **Ruby is not thread safe**
  - GIL – Global Interpreter Lock
    - Python has a GIL as well
  - Rails itself can only process one request at a time
    - PHP and CPython have the same limitation
- **What does this mean for a ruby based architecture?**

Mongrel – web server in Ruby

Overview

- Simple server
  - use a custom HTTP 1.1 parser and URIClassifier to process requests
- Run as a separate process
- Very robust
- Used in conjunction with Apache mod_proxy_balancer
- Create a pool of Mongrels to serve application
- Mongrel 2 is Event Driven

Cons

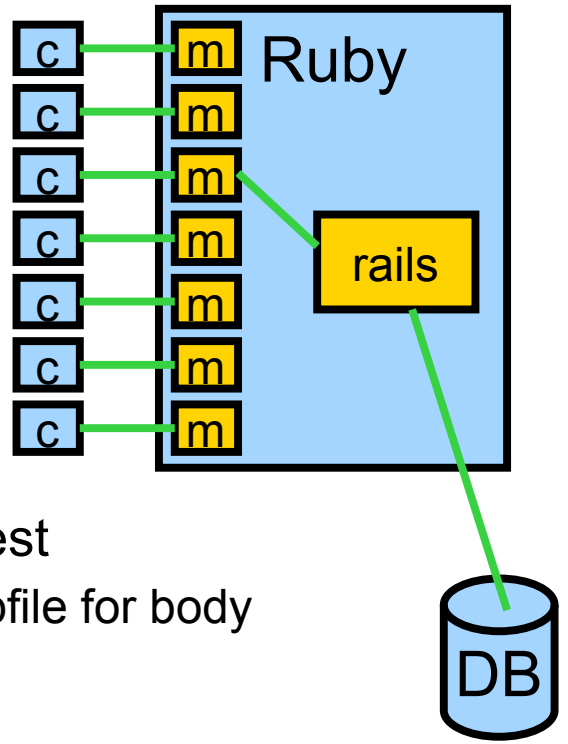
- No built-in mechanism for resizing the mongrel pool dynamically
- Can use a lot of memory
  - Most of the memory in a Rails application is used to load the AST (code)

What do you do when a mongrel process dies?

Mongrel & threading

Details for a connection

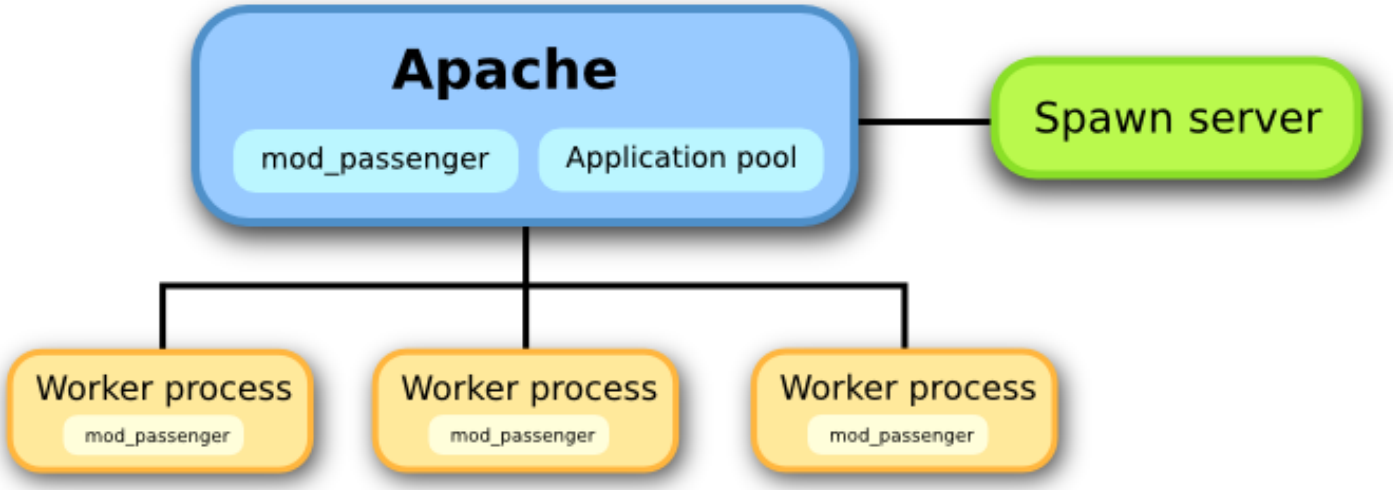
- create thread and parse HTTP request headers
- Read request body
  - Small: put in into a StringIO object
  - Large: stream it to a temp file
- Call the RailsHandler
- RailsHandler checks page cache
  - Hit: return the cached page
- Acquire Rails lock
- Calls Rails Dispatcher to handle the request
  - pass in the headers, and StringIO or Tempfile for body
  - output returned in StringIO object
- Release Rails lock
- Stream output headers & StringIO output to client



Phusion Passenger

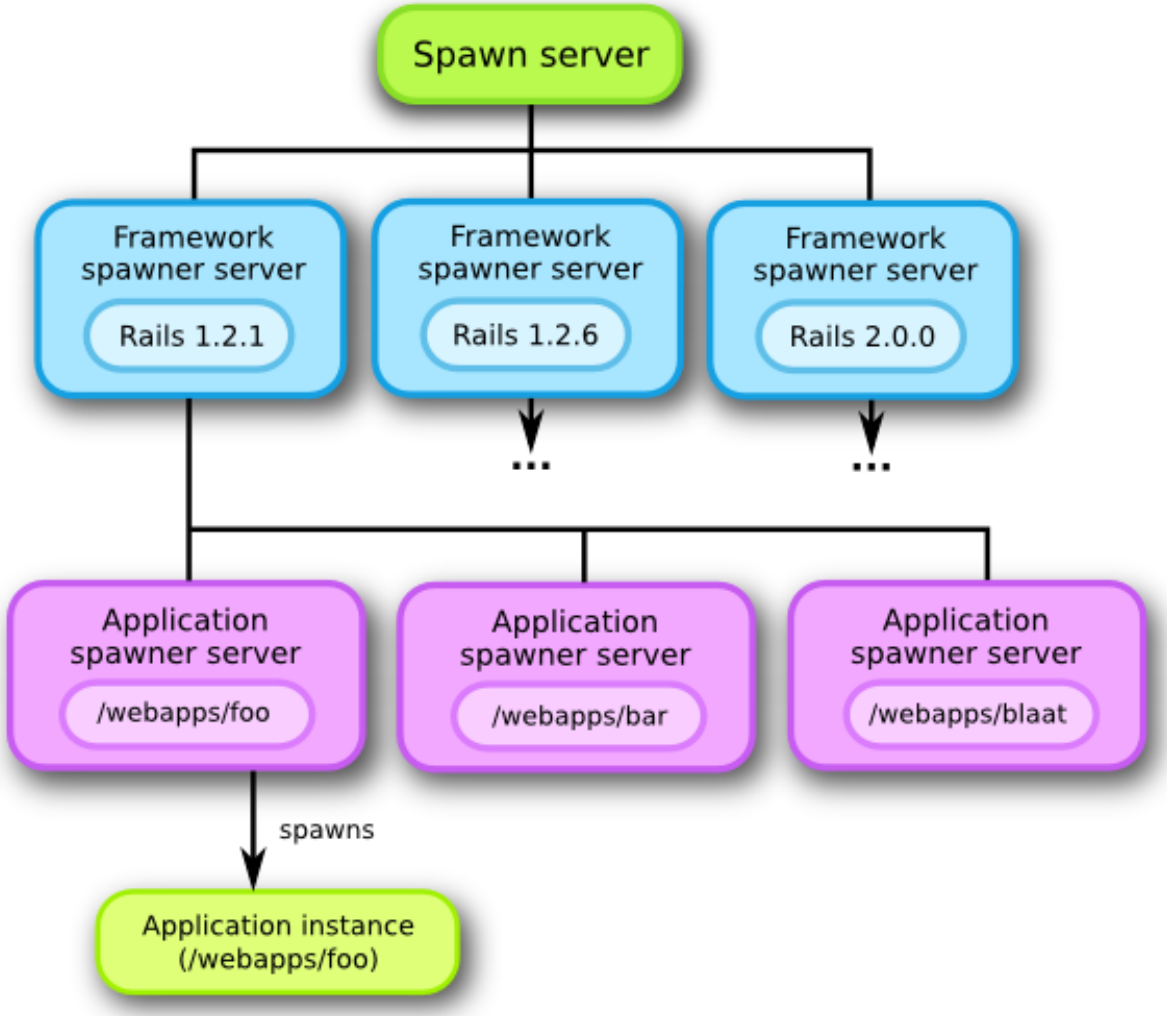
- **Overview**
  - Also referred to as *mod_rails*
  - Designed for performance
  - Works within Apache or Nginx to spawn new instances of applications
    - Can be done dynamically
  - Designed to be robust
    - Will not crash Apache/Nginx
  - Easy setup – no port management
  - Supports Ruby and Python
- **Reduces memory use – How?**
- **Latest version uses Event Driven (“Evented I/O”) architecture**

Passenger



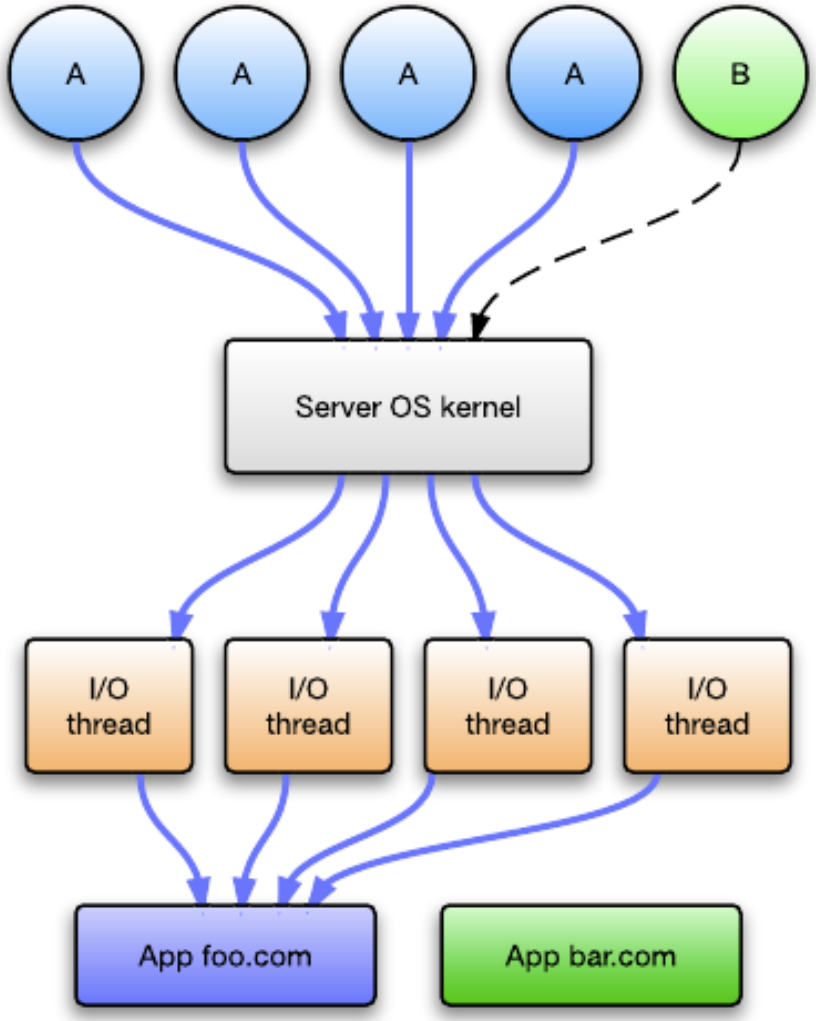
Source: http://www.modrails.com/

Passenger



How do you talk to the framework?

Passenger – Evented I/O



Source: http://blog.phusion.nl/

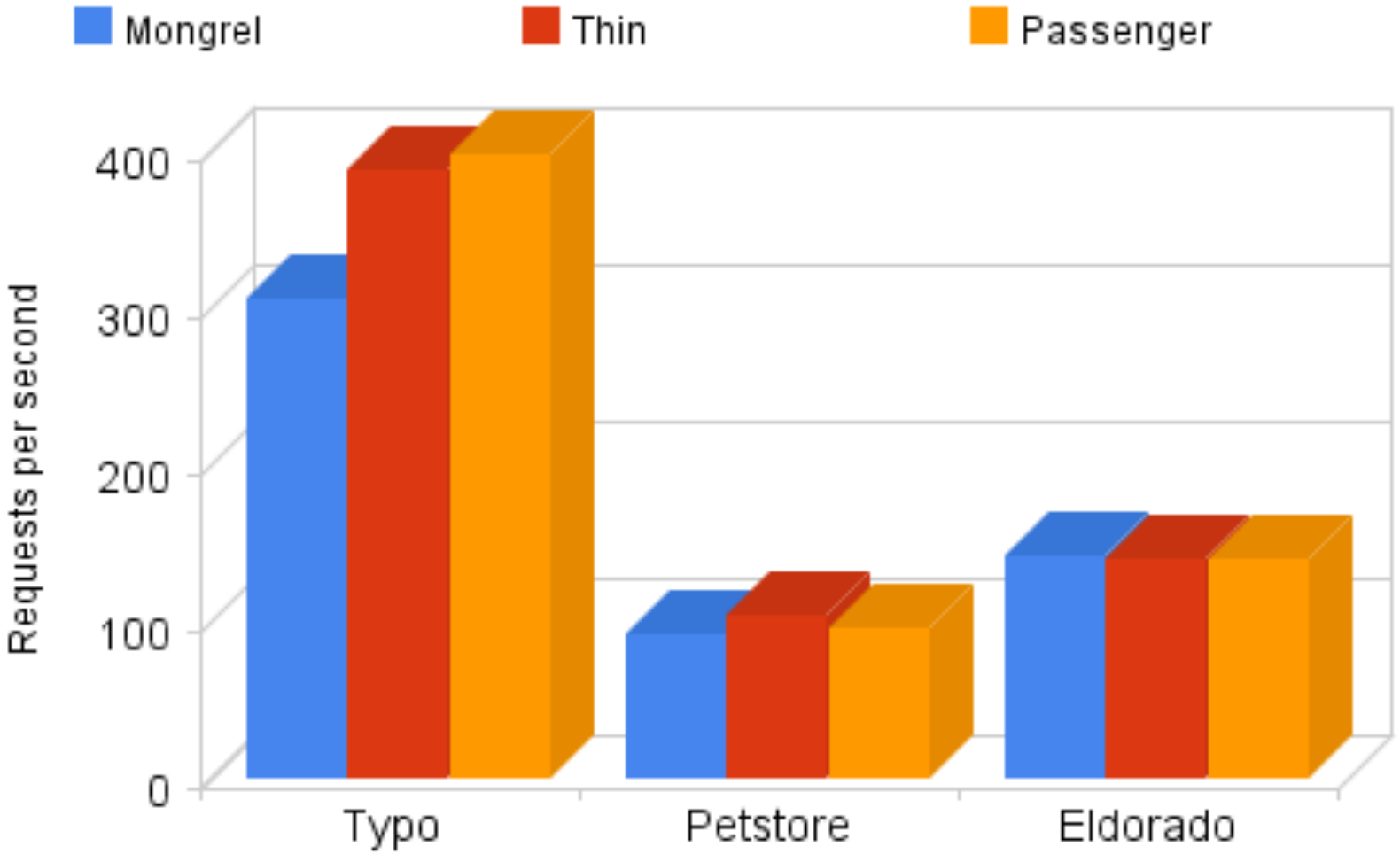
Other Ruby App Servers

- Thin - http://code.macournoyer.com/thin/
- Unicorn - http://unicorn.bogomips.org/
- Goliath – http://goliath.io/
- JRuby + Glassfish - http://www.jruby.org/
- And of course… WebBrick
  - Part of the ruby standard library

- **Many more app servers for other languages**
  - Like node.js

- **Build your own**
  - libev, libevent – cross OS libraries for evented I/O
  - Twisted – event driven networking engine

Performance



Source: www.modrails.com

Questions about Server Architecture?

Summary

- **Scalable Internet Services**
  - Involve both an HTTP and an App server
  - There are many choices for both (and new ones all the time)
  - Considerations are:
    - responsiveness/Latency, throughput, concurrency, resources per request, robustness, ease of development
- **This is a great research area!**
  - Event driven architectures are the current trend
  - Maybe you will solve the C1B problem?

  - TODO: move?
  - libev, libevent – cross OS libraries for evented I/O
  - Twisted – event driven networking engine

Announcements

- Monday – Guest Speaker
  - Jon Hsieh from Cloudera
  - Read **Map Reduce** and **Big Table** papers
- Sprint 1 – Demo in lab Monday 10/21
  - Complete team page
  - Create user stories for the functionality you will build this quarter and enter them into tracker
  - Create mockups of the user interface and post to team page
- If you want to work ahead
  - Get github setup for your team
  - Experiment with EC2