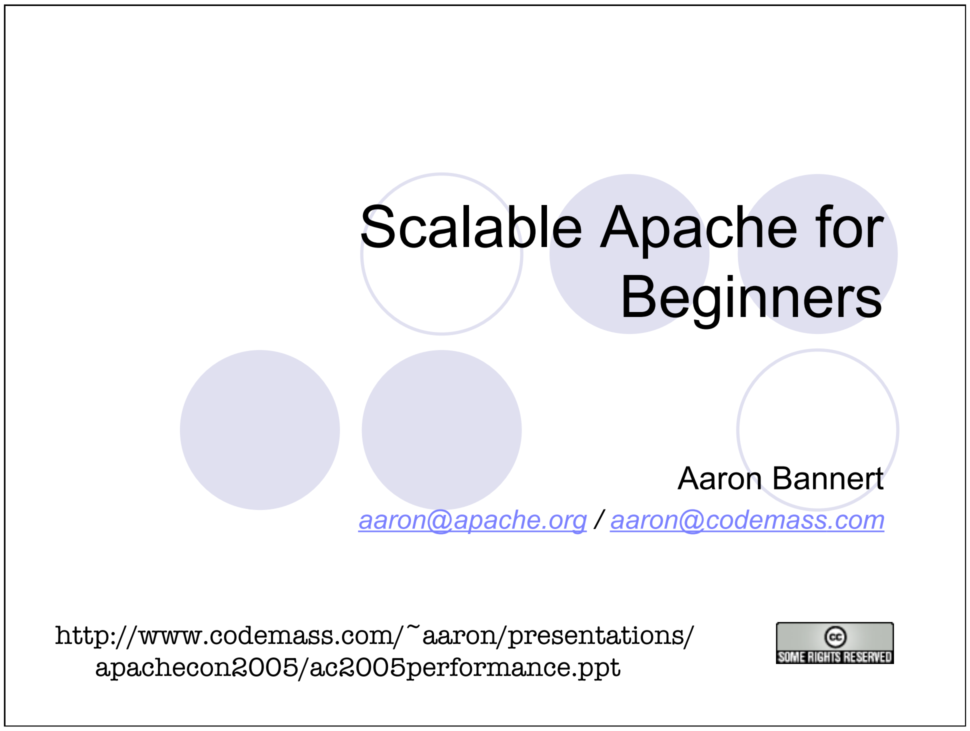


Scalable Apache for Beginners

Aaron Bannert
aaron@apache.org / aaron@codemass.com

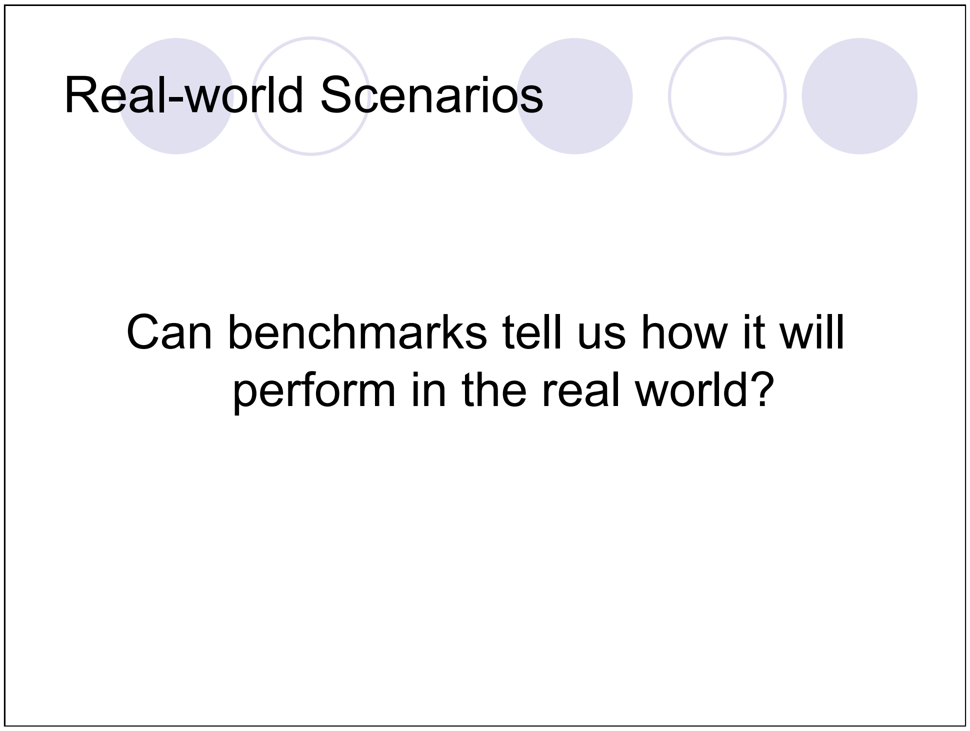
http://www.codemass.com/~aaron/presentations/apachecon2005/ac2005performance.ppt

Measuring Performance

What is *Performance*?

How do we measure performance?

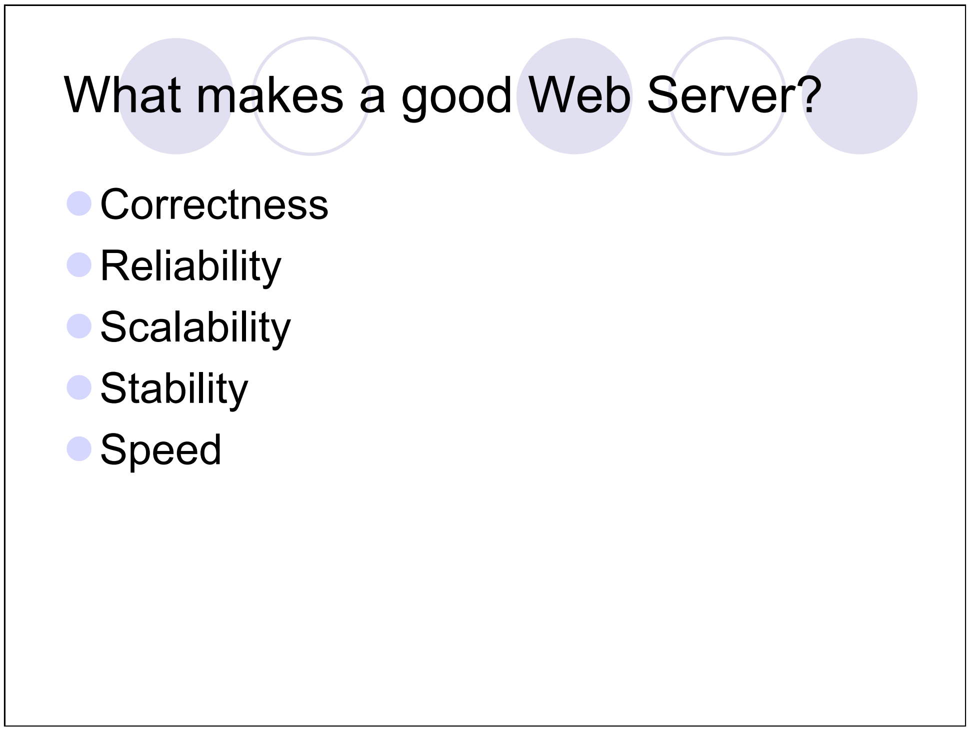
- Benchmarks
  - Requests per Second
  - Bandwidth
  - Latency
  - Concurrency (Scalability)



Real-world Scenarios

Can benchmarks tell us how it will perform in the real world?

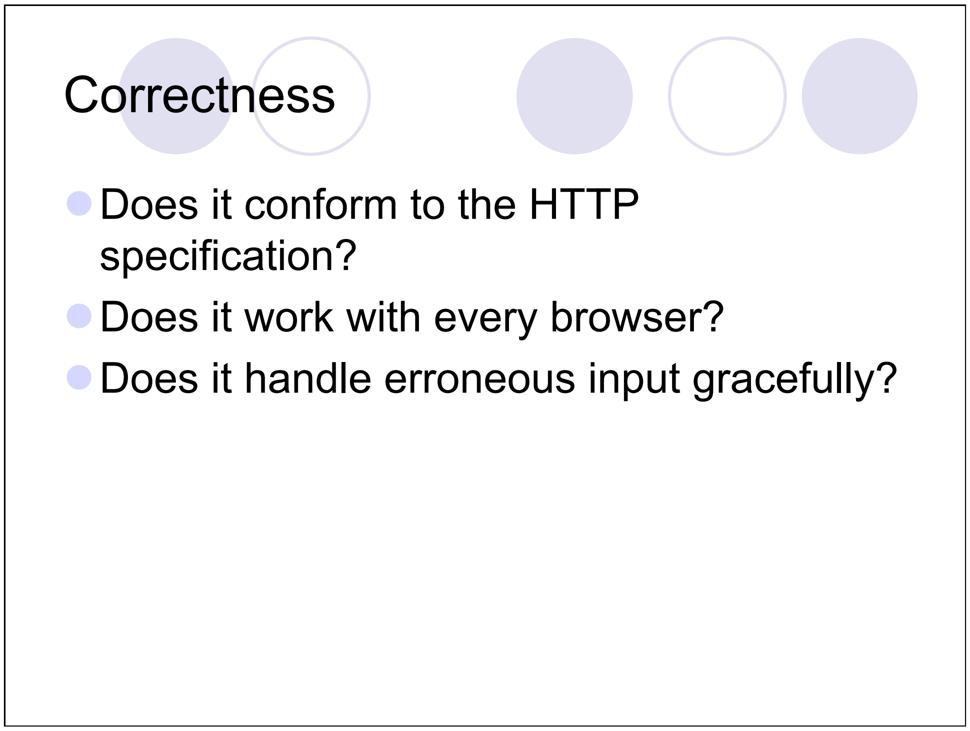
Benchmarks are a valuable tool for testing performance. All benchmarks strive to some degree to represent real-world scenarios. The key is knowing how well your benchmarks represent your workload. The accuracy of your benchmarks will directly affect your ability to improve your system using those results.



What makes a good Web Server?

- Correctness
- Reliability
- Scalability
- Stability
- Speed

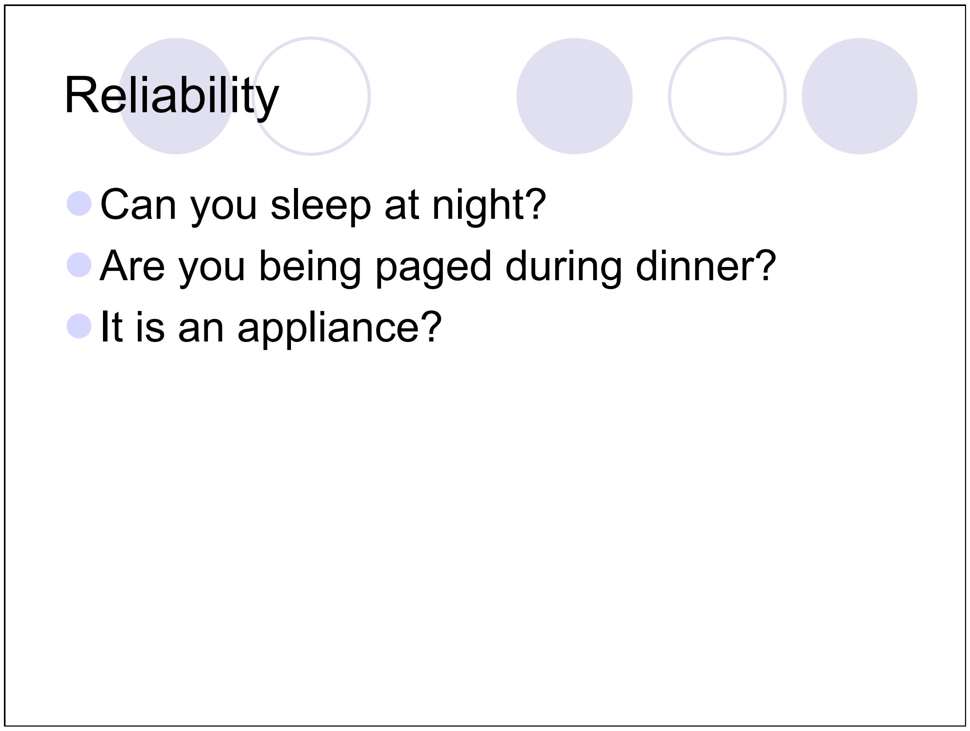
In decreasing order of importance, these are the most important values of a production web server.



Correctness

- Does it conform to the HTTP specification?
- Does it work with every browser?
- Does it handle erroneous input gracefully?

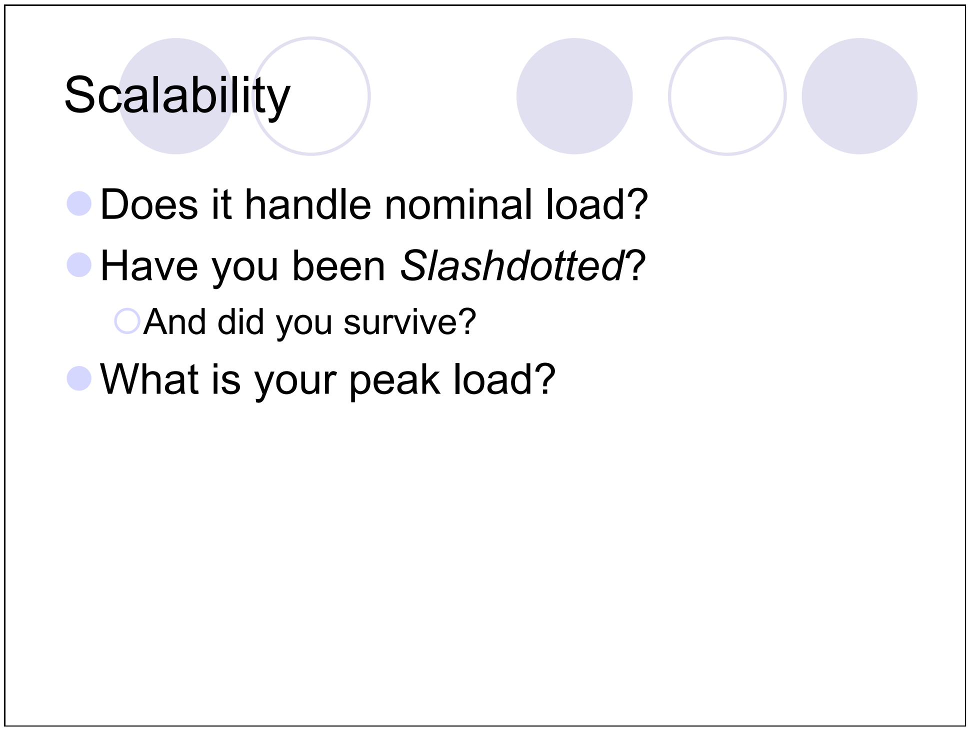
It MUST conform to the HTTP spec. It should be flexible with input and strict with output.



Reliability

- Can you sleep at night?
- Are you being paged during dinner?
- It is an appliance?

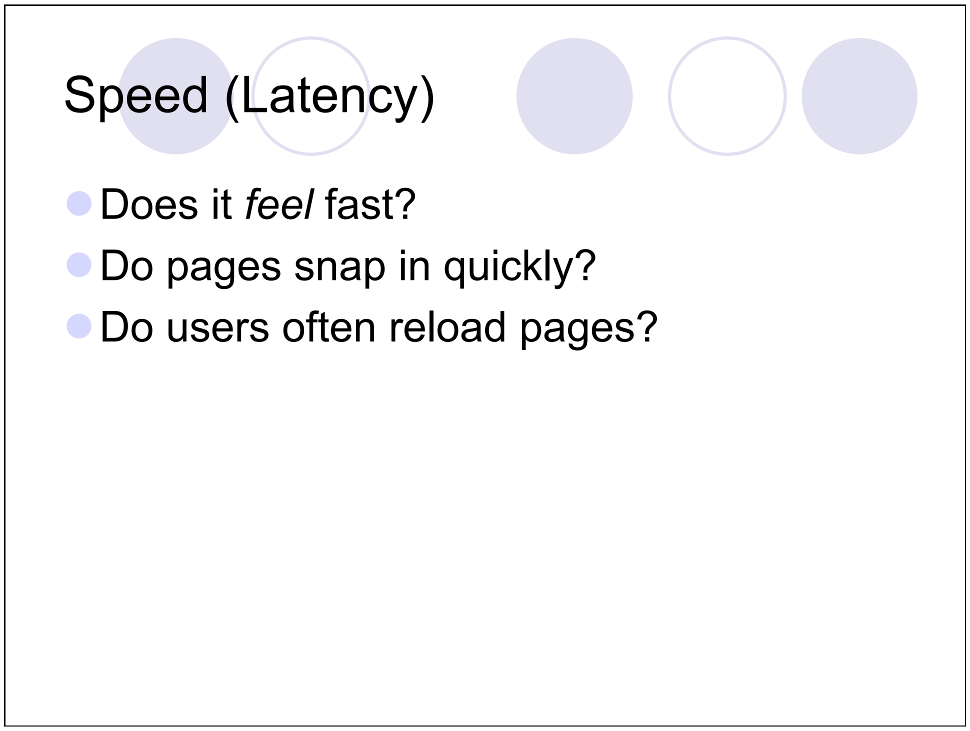
We don’t want our web servers waking us up in the middle of the night or requiring constant attention. We want a web server that we set up once and never touch again. We want this thing to behave like an appliance, never breaking down.



Scalability

- Does it handle nominal load?
- Have you been *Slashdotted*?
  - And did you survive?
- What is your peak load?

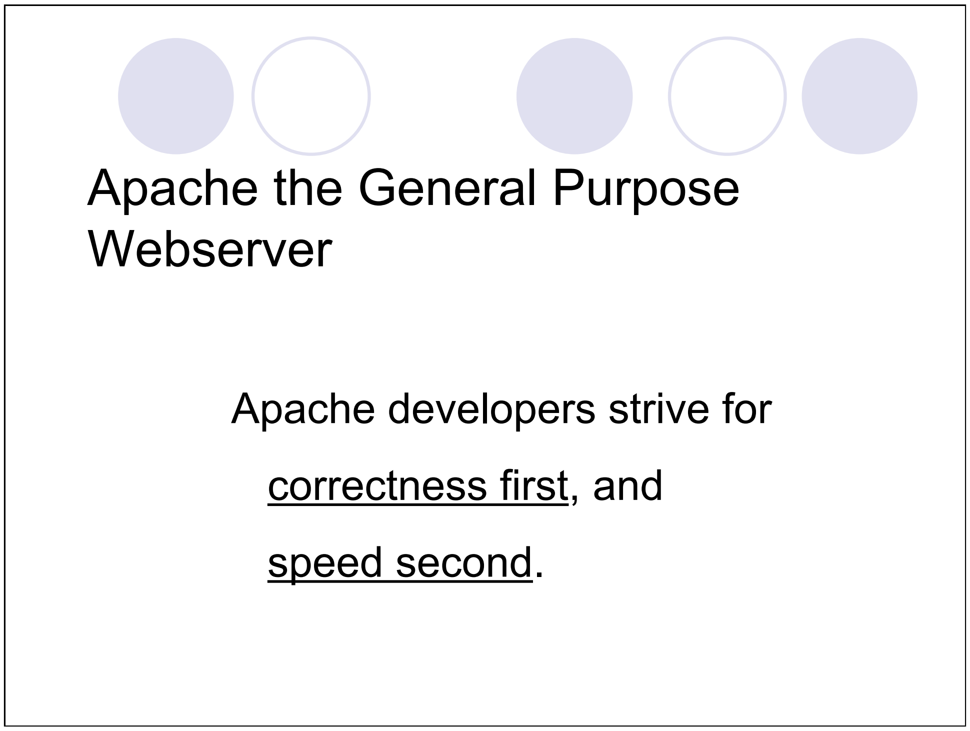
Many systems are deployed without significant scalability testing. This is unfortunate, since these are the sites that fail when uptime is most critical.



Speed (Latency)

- Does it *feel* fast?
- Do pages snap in quickly?
- Do users often reload pages?

Although Latency and Concurrency are different values that we strive to achieve in a web server, they are actually very closely related. Often, as a site struggles to serve more and more concurrent users, the speed at which each page is served will increase. One critical point is the point when users begin to hit stop on their browsers and reload the page. This can place even more load on an already strained system, causing further service denial.



Apache the General Purpose Webserver

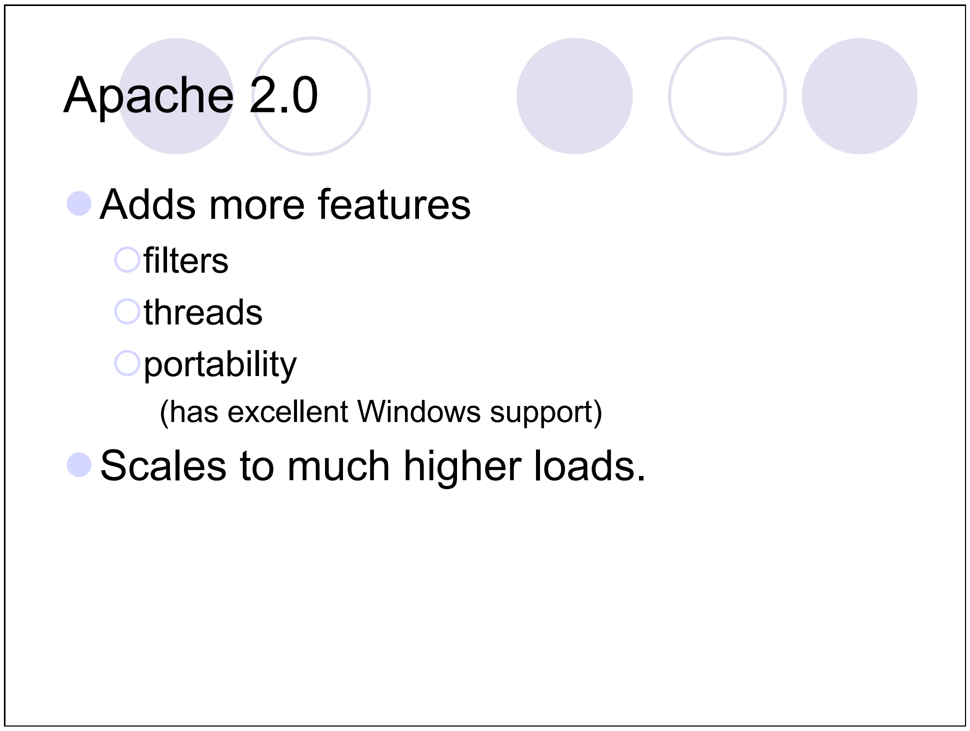
Apache developers strive for

correctness first, and

speed second.

Apache 1.3

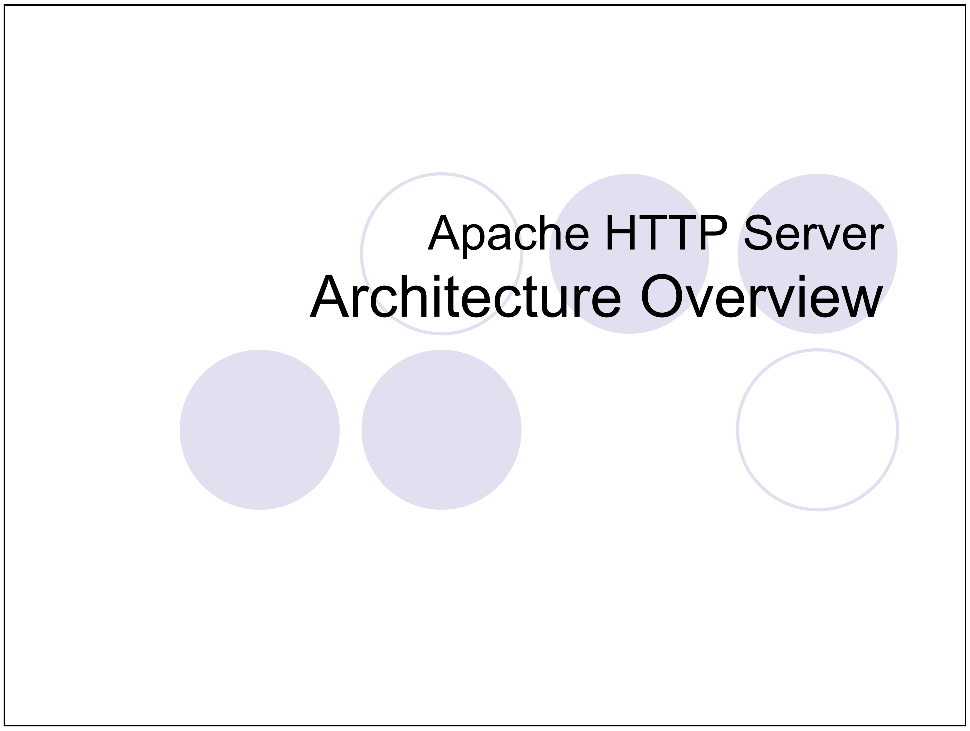
- Fast enough for most sites
- Particularly on 1 and 2 CPU systems.



Apache 2.0

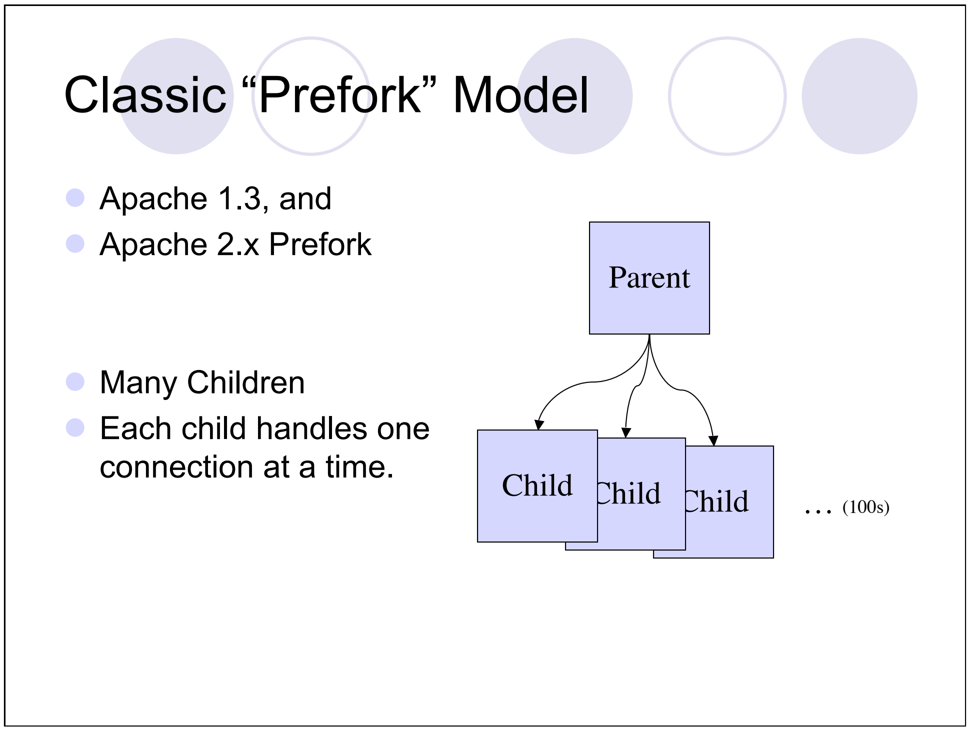
- Adds more features
  - filters
  - threads
  - portability

    (has excellent Windows support)
- Scales to much higher loads.



Apache HTTP Server

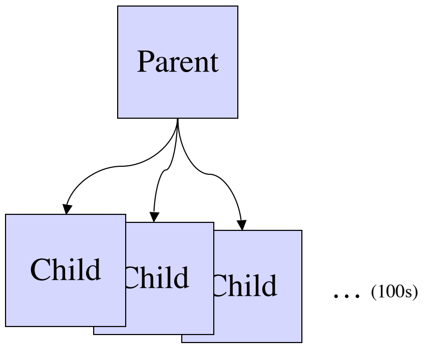
Architecture Overview



Classic “Prefork” Model

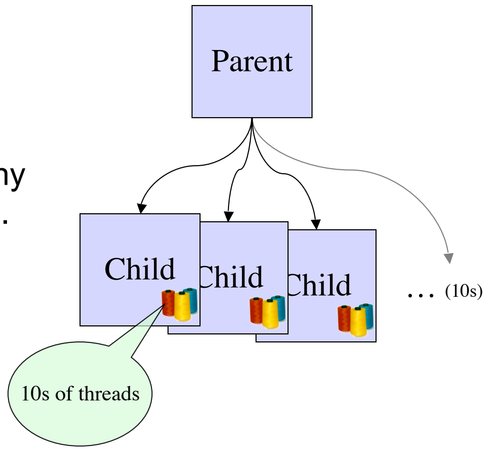
- Apache 1.3, and
- Apache 2.x Prefork

- Many Children
- Each child handles one connection at a time.



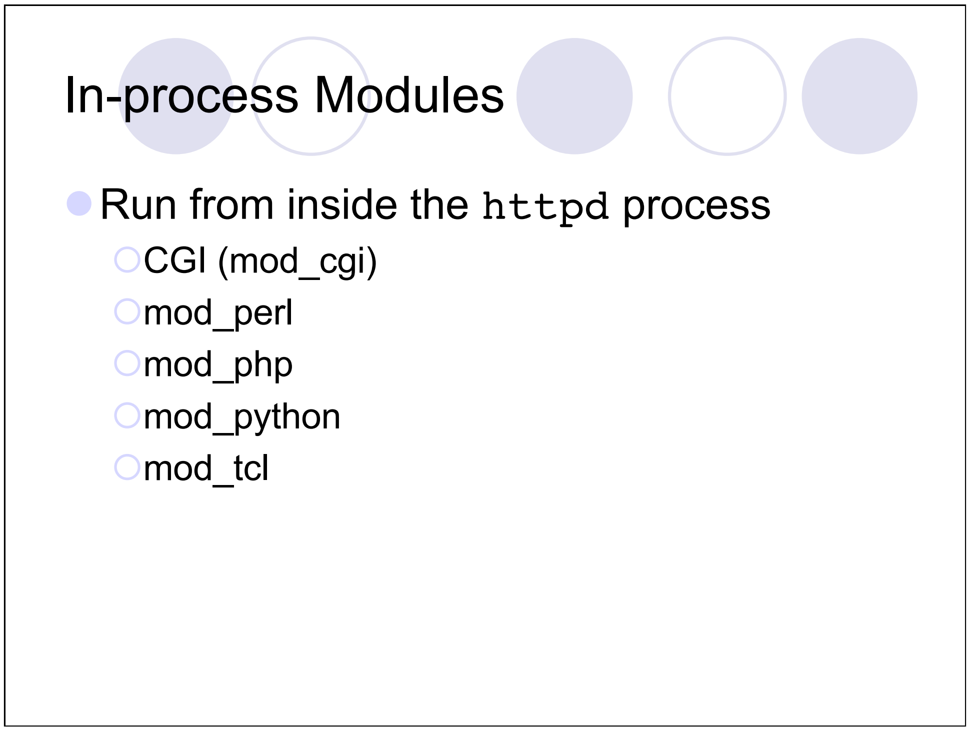
Multithreaded “Worker” Model

- Apache 2.x Worker
- Few Children
- Each child handles many concurrent connections.



Dynamic Content: Modules

- Extensive API
- Pluggable Interface
- Dynamic or Static Linkage

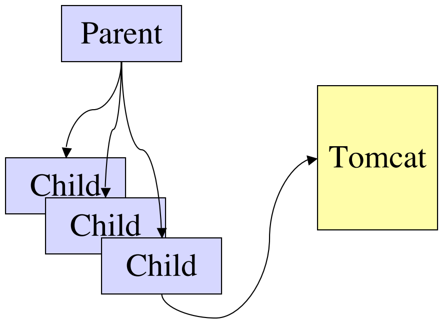


In-process Modules

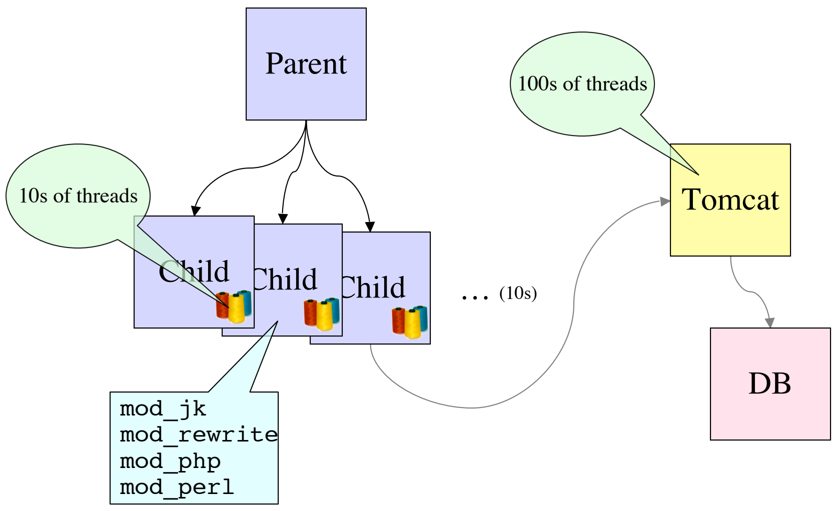
- Run from inside the `httpd` process
  - CGI (mod_cgi)
  - mod_perl
  - mod_php
  - mod_python
  - mod_tcl

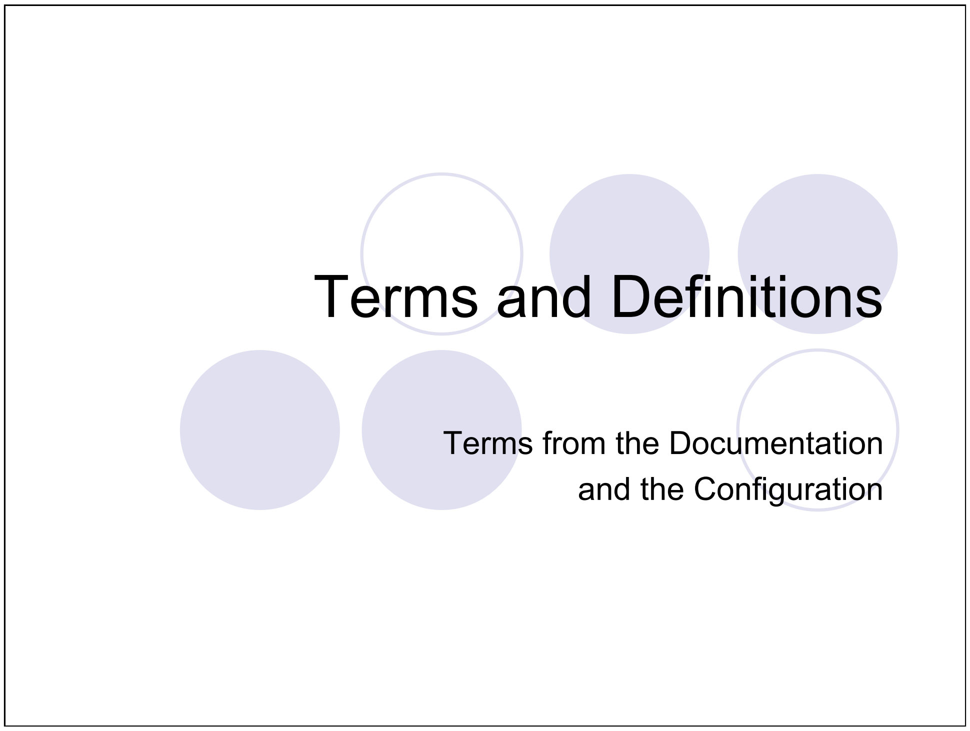
Out-of-process Modules

- Processing happens outside of httpd (eg. Application Server)
- Tomcat
  - mod_jk/jk2, mod_jserv
- mod_proxy
- mod_jrun



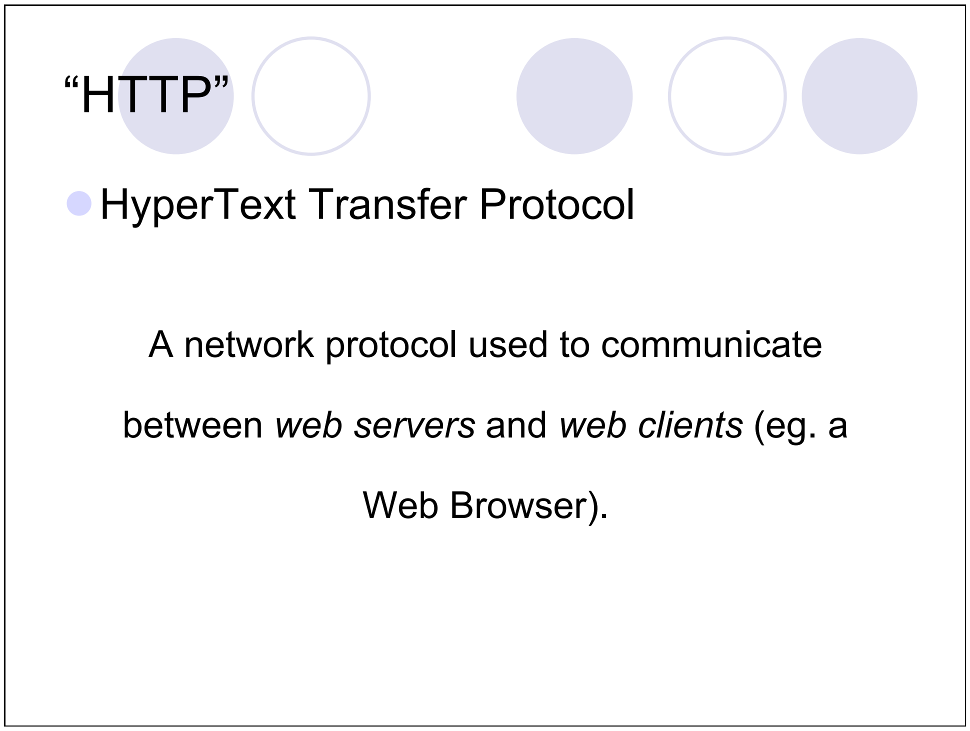
Architecture: The Big Picture





Terms and Definitions

Terms from the Documentation
and the Configuration

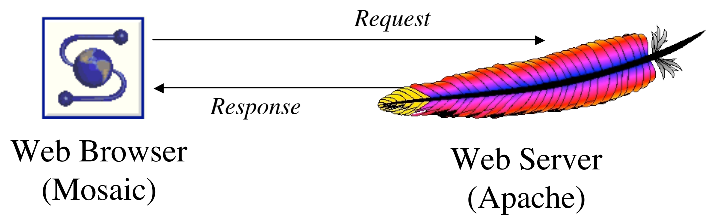


“HTTP”

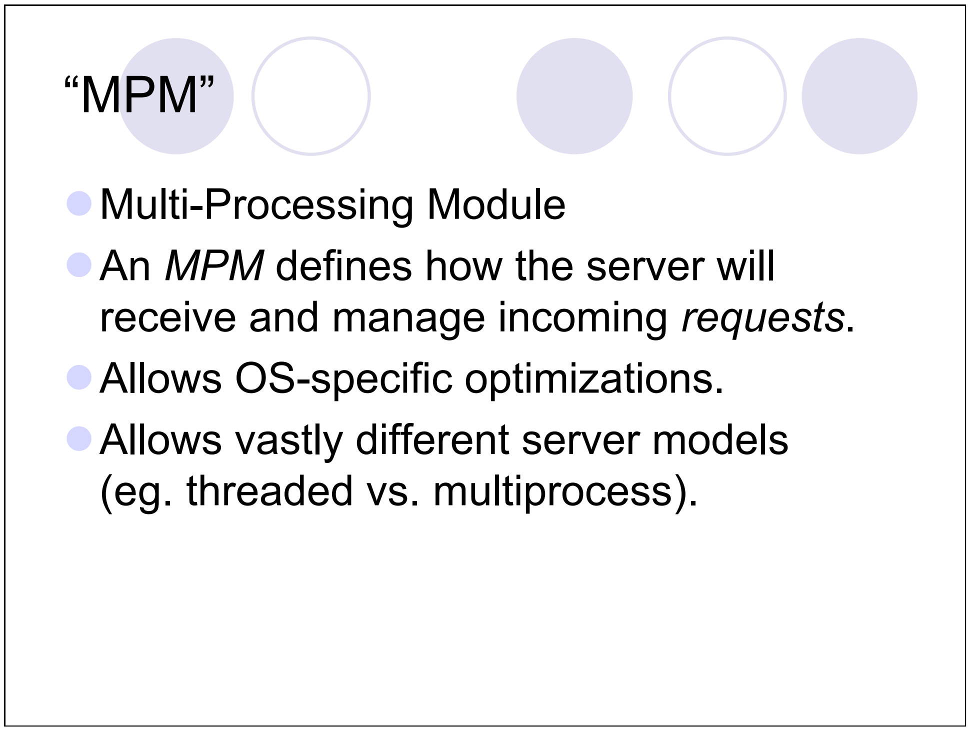
- HyperText Transfer Protocol

A network protocol used to communicate between *web servers* and *web clients* (eg. a Web Browser).

“Request” and “Response”



- Web browsers *request* pages and web servers *respond* with the result.

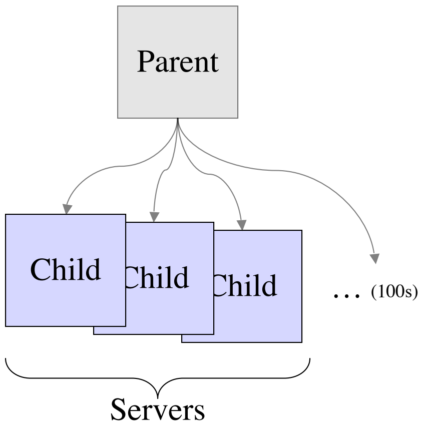


“MPM”

- Multi-Processing Module
- An *MPM* defines how the server will receive and manage incoming *requests*.
- Allows OS-specific optimizations.
- Allows vastly different server models (eg. threaded vs. multiprocess).

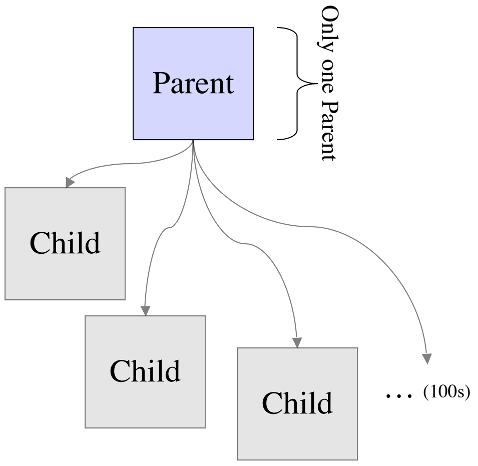
“Child Process” aka “Server”

- Called a “Server” in `httpd.conf`
- A single `httpd` process.
- May handle one or more concurrent requests (depending on the *MPM).*



“Parent Process”

- The main `httpd` process.
- Does not handle connections itself.
- Only creates and destroys children.



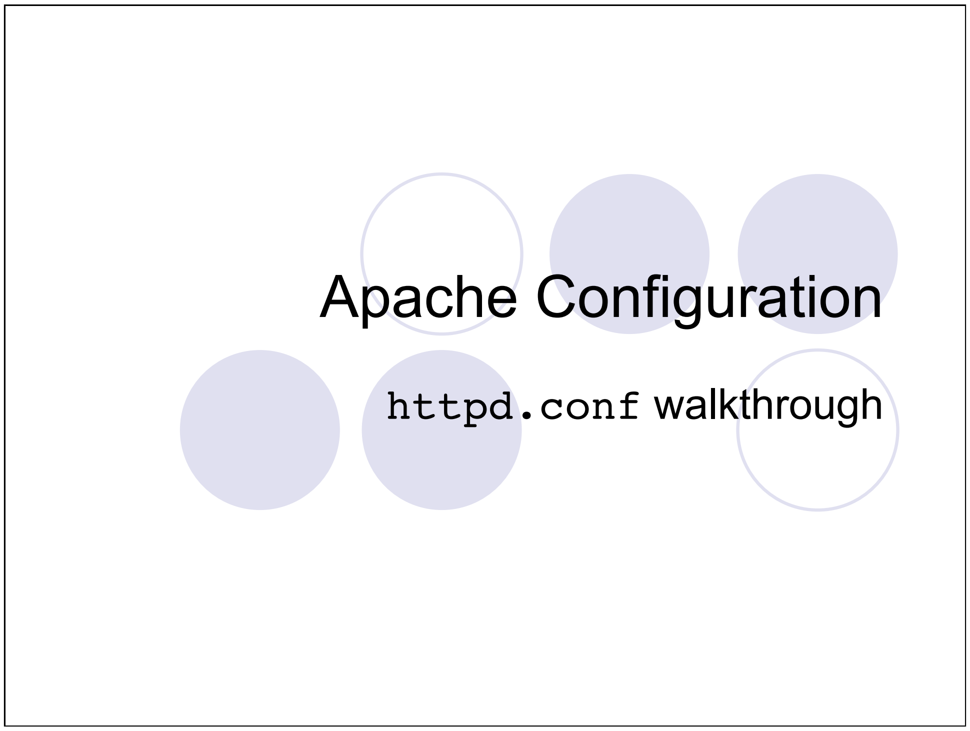
“Client”

Web Browser
(Mosaic)
Web Server
(Apache)

- Single HTTP connection (eg. web browser).
  - Note that many web browsers open up multiple *connections*. Apache considers each *connection* uniquely.

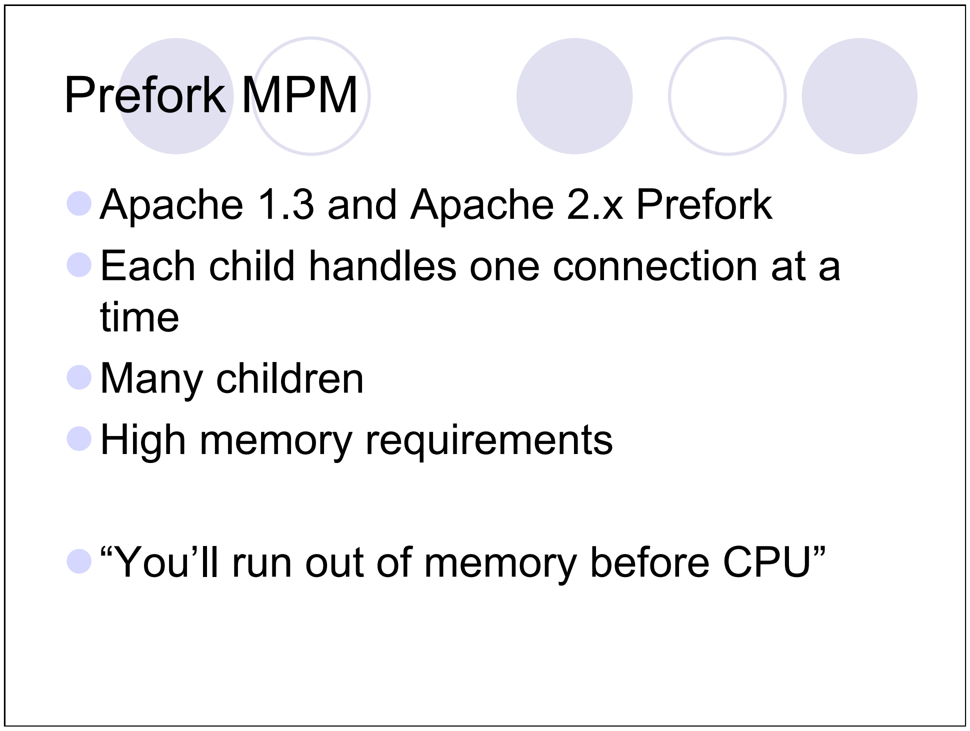
“Thread”

- In multi-threaded MPMs (eg. *Worker*).
- Each thread handles a single connection.
- Allows Children to handle many connections at once.



Apache Configuration

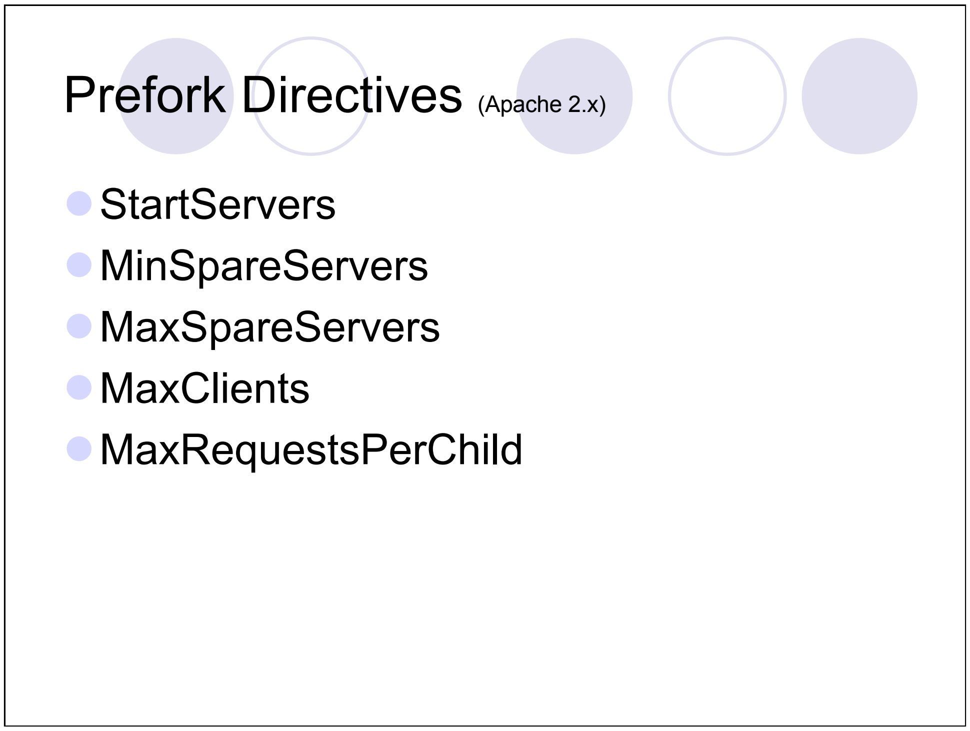
`httpd.conf` walkthrough



Prefork MPM

- Apache 1.3 and Apache 2.x Prefork
- Each child handles one connection at a time
- Many children
- High memory requirements

- "You'll run out of memory before CPU"



Prefork Directives (Apache 2.x)

- StartServers
- MinSpareServers
- MaxSpareServers
- MaxClients
- MaxRequestsPerChild

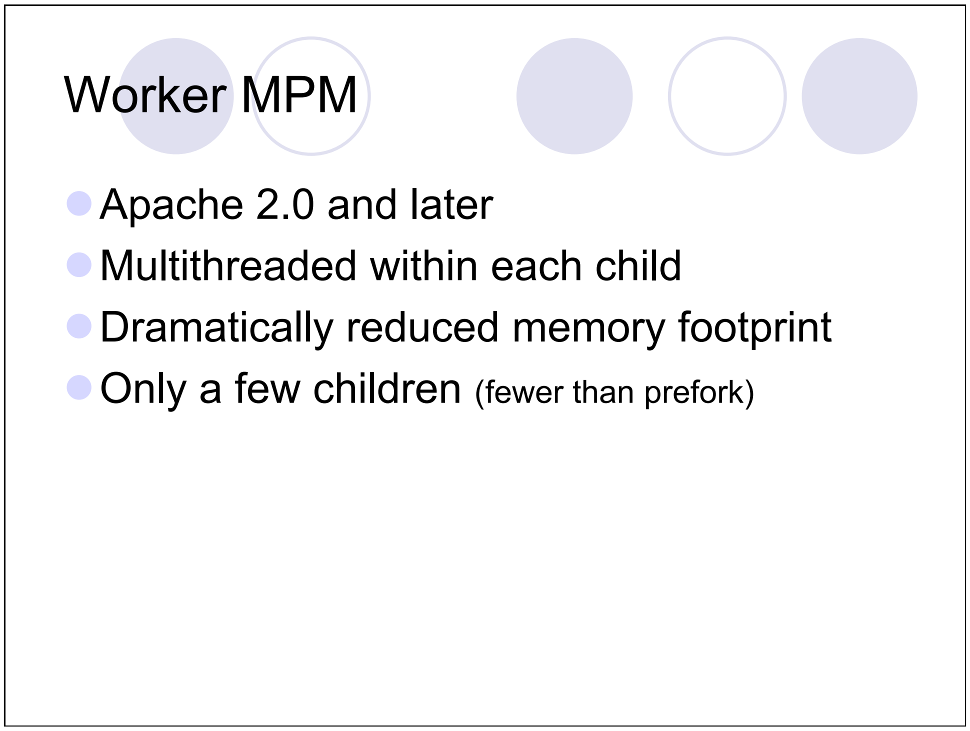
StartServer - The number of child processes to create at start-time.

MinSpareServer - The minimum number of idle children to have at any time.

MaxSpareServer - The maximum number of idle children to have at any time.

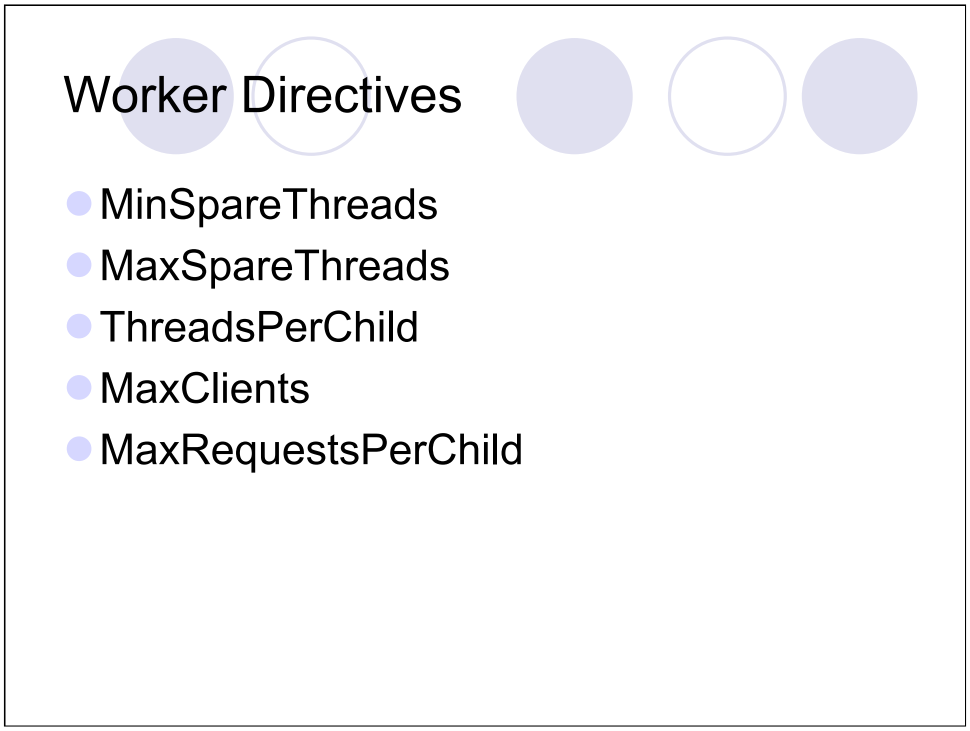
MaxClients - The maximum number of concurrent client connections to allow at any time.

MaxRequestsPerChild - The maximum number of requests that each child is allowed to serve before it must terminate and be replaced. Useful for starting with a clean slate every once in awhile, and particular useful for buggy 3rd party modules that leak precious system resources.



Worker MPM

- Apache 2.0 and later
- Multithreaded within each child
- Dramatically reduced memory footprint
- Only a few children (fewer than prefork)



Worker Directives

- MinSpareThreads
- MaxSpareThreads
- ThreadsPerChild
- MaxClients
- MaxRequestsPerChild

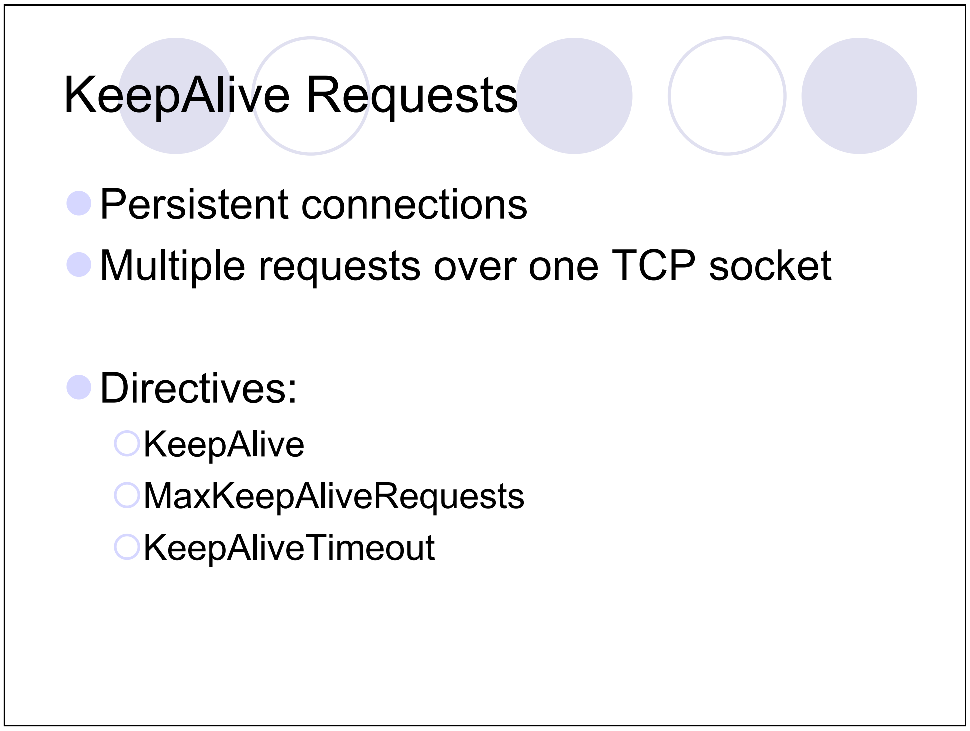
MinSpareThreads - The minimum number of idle threads to allow at any time across all children.

MaxSpareThreads - The maximum number of idle threads to allow at any time across all children.

ThreadsPerChild - The number of threads within each child.

MaxClients - The maximum number of concurrent client connections to allow at any time.

MaxRequestsPerChild - The maximum number of requests that each child is allowed to serve before it must terminate and be replaced. Useful for starting with a clean slate every once in awhile, and particular useful for buggy 3rd party modules that leak precious system resources.



KeepAlive Requests

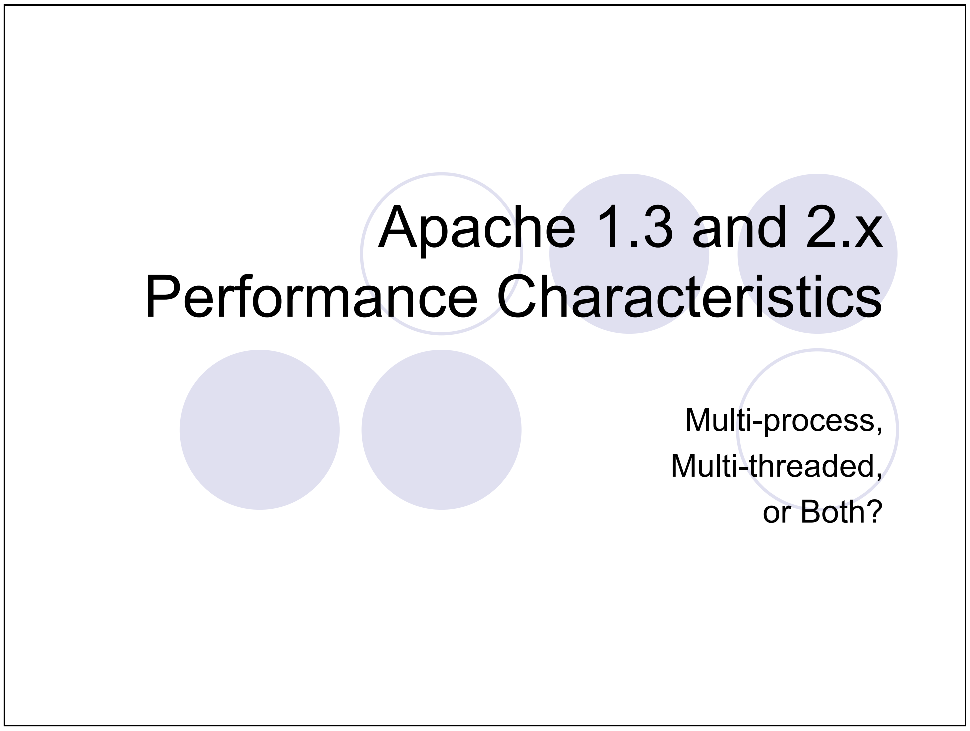
- Persistent connections
- Multiple requests over one TCP socket

- Directives:
  - KeepAlive
  - MaxKeepAliveRequests
  - KeepAliveTimeout

KeepAlive - Enable or Disable KeepAlive suport.

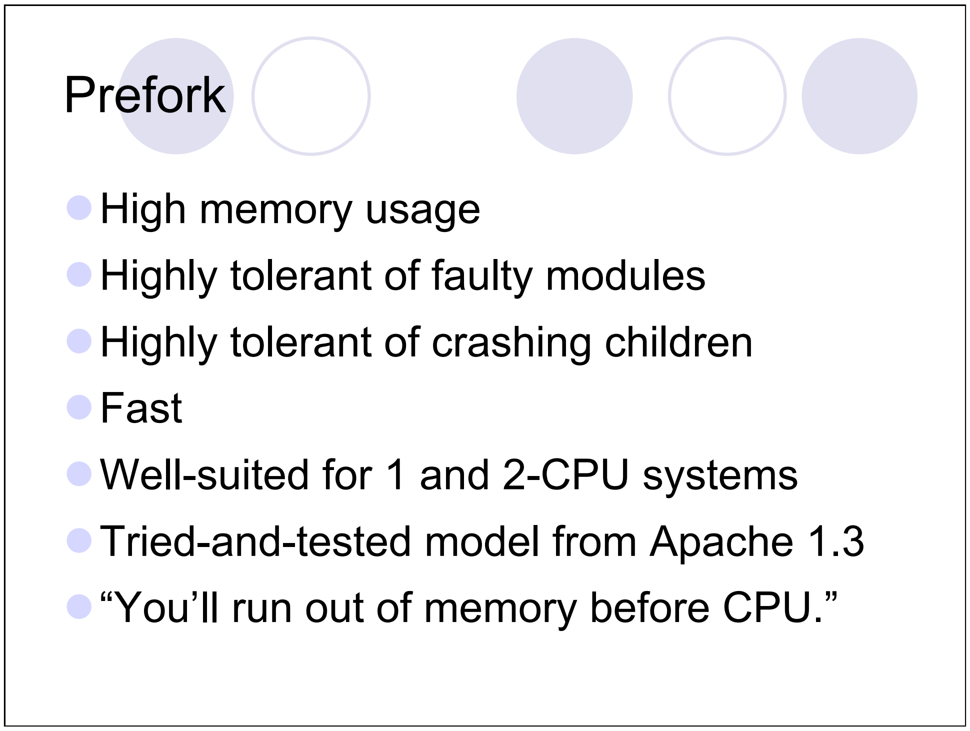
MaxKeepAliveRequests - The maximum number of requests to allow across one persistent TCP/IP connection.

KeepAliveTimeout - The maximum amount of time to wait between requests on one persistent TCP/IP connection.



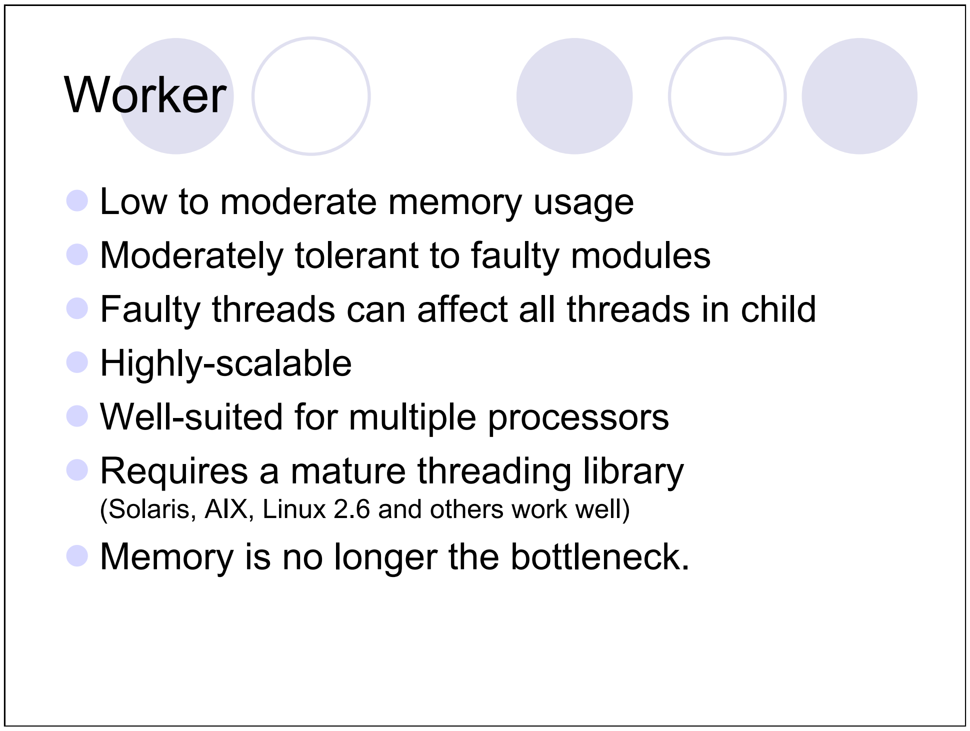
Apache 1.3 and 2.x Performance Characteristics

Multi-process,
Multi-threaded,
or Both?



Prefork

- High memory usage
- Highly tolerant of faulty modules
- Highly tolerant of crashing children
- Fast
- Well-suited for 1 and 2-CPU systems
- Tried-and-tested model from Apache 1.3
- “You’ll run out of memory before CPU.”

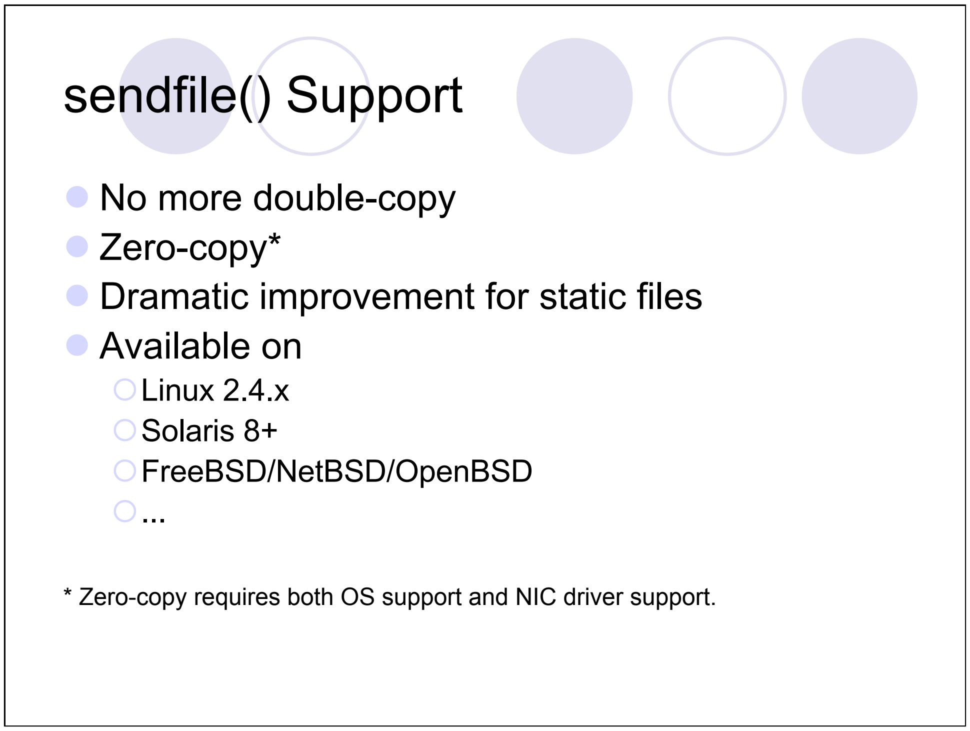


Worker

- Low to moderate memory usage
- Moderately tolerant to faulty modules
- Faulty threads can affect all threads in child
- Highly-scalable
- Well-suited for multiple processors
- Requires a mature threading library
  (Solaris, AIX, Linux 2.6 and others work well)
- Memory is no longer the bottleneck.

Important Performance Considerations

- sendfile() support
- DNS considerations
- stat() calls
- Unnecessary modules


sendfile() Support

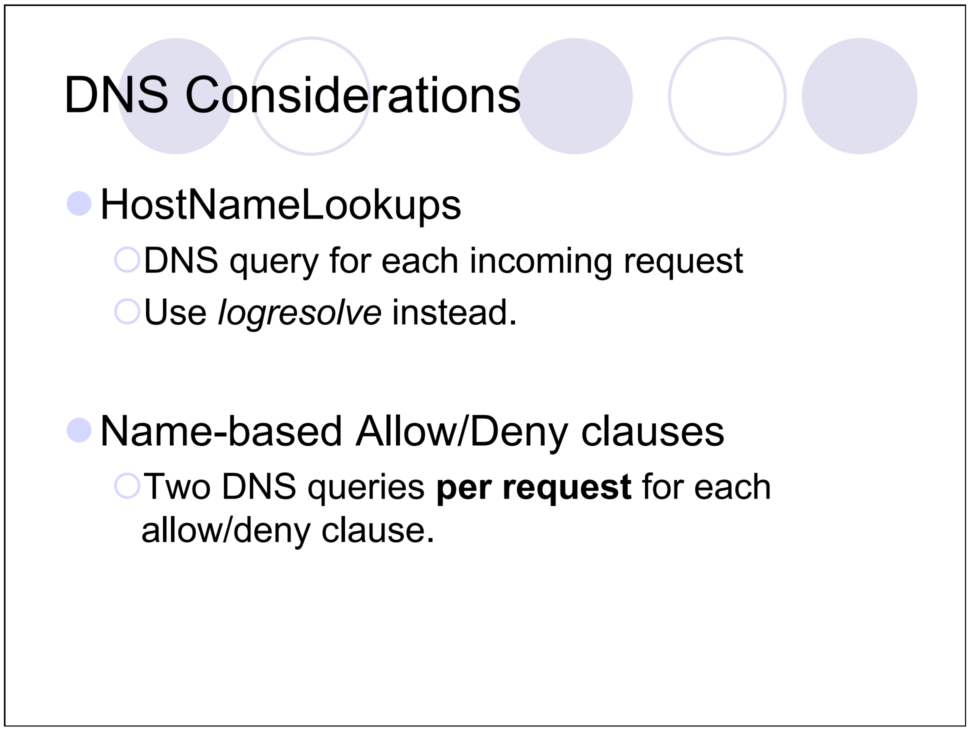
- No more double-copy
- Zero-copy*
- Dramatic improvement for static files
- Available on
  - Linux 2.4.x
  - Solaris 8+
  - FreeBSD/NetBSD/OpenBSD
  - ...

* Zero-copy requires both OS support and NIC driver support.

If you serve a lot of static content (eg. html pages, images, downloaded files, etc) then you should seriously consider running under an OS that supports sendfile.

Double-copy is what happens when a process reads data from a file and sends it to a network device. The first copy happens when the kernel reads the file into the userspace process memory area. The second copy happens when the kernel copies the data back out of userspace into kernel space, forms a full data packet, and then copies that to the network card. When sendfile is involved, the process simply instructs the kernel to send a particular file out to a network connection. The kernel is then able to skip one of the copy steps, dramatically decreasing processing. In Apache 2.0 is can actually be much worse than double-copy, since filters that manipulate data sometimes need a third or subsequent copy.

Zero-copy is the best-case scenario, since that is when the kernel and the network card cooperate together to read and assemble data directly from a disk straight to the network. The data can actually pass to a network socket without ever having to be copied into main memory. Modern UNIX (and Windows) operating systems support this, but it is important that the network driver also have support for this, otherwise you fall back to simple sendfile() support.

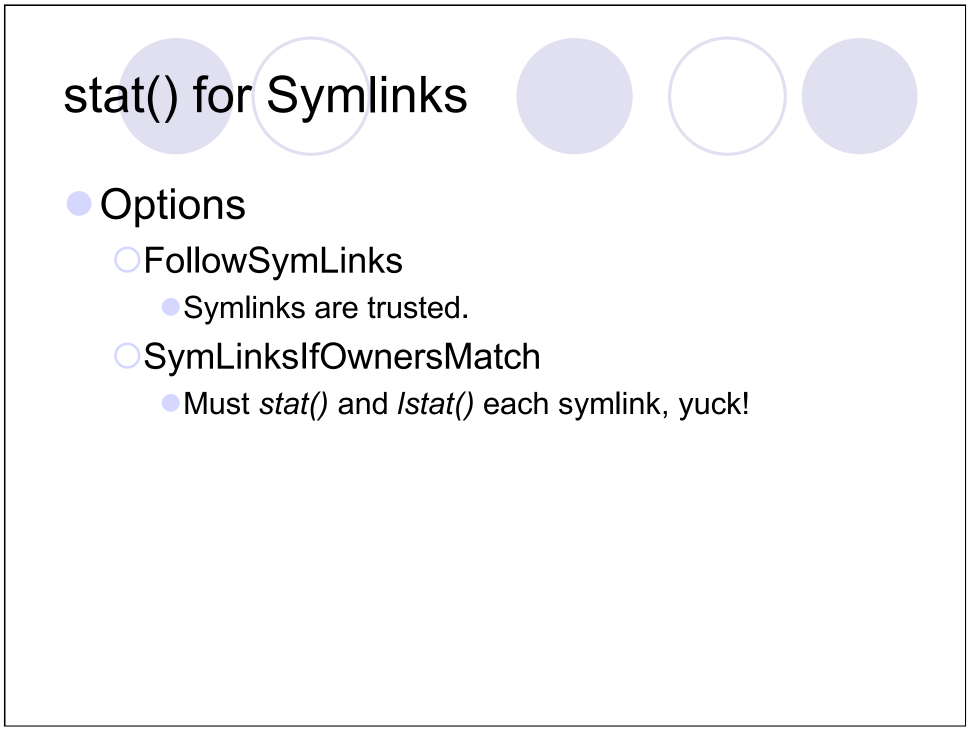


DNS Considerations

- HostNameLookups
  - DNS query for each incoming request
  - Use *logresolve* instead.
- Name-based Allow/Deny clauses
  - Two DNS queries **per request** for each allow/deny clause.

HostNameLookups are disabled by default since Apache 1.3. If you need to get equivalent functionality, the logresolve program that comes with Apache will often suffice. It is suggested that the processing occur on a different box than the web server, so as to not adversely affect server performance.

If possible, restrict Allow/Deny clauses to a small set of users and directory locations and only use IP addresses.

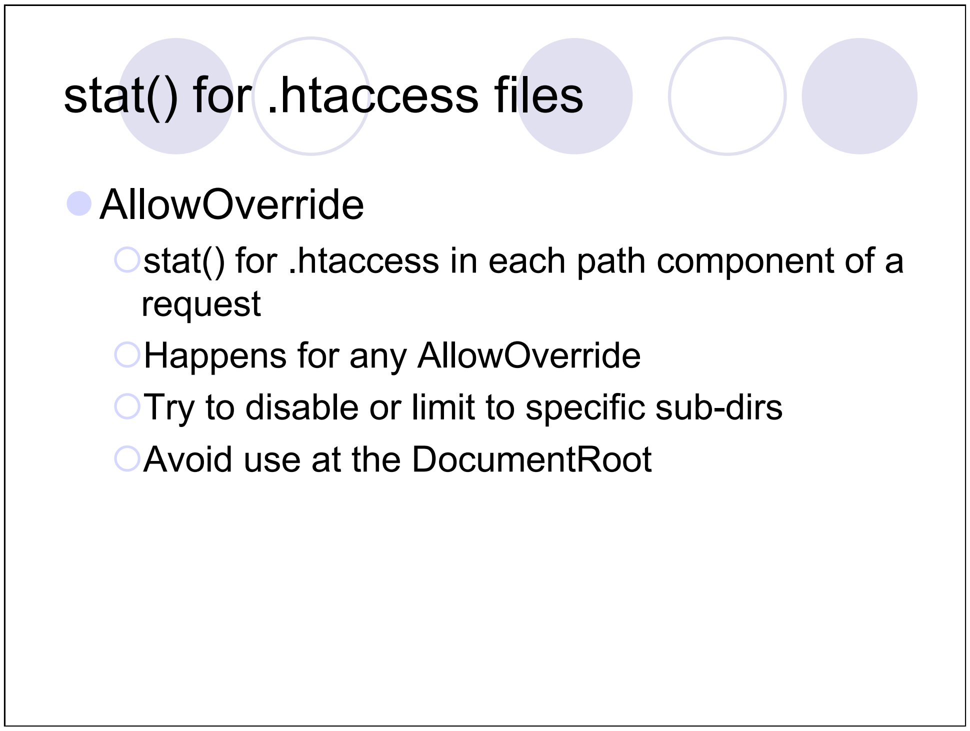


stat() for Symlinks

- Options
  - FollowSymLinks
    - Symlinks are trusted.
  - SymLinksIfOwnersMatch
    - Must *stat()* and *lstat()* each symlink, yuck!

By default, Apache will check each path component of a URL with stat() to see if it is a symlink. This doesn’t happen if FollowSymLinks is enabled, since symlinks are simply trusted. However, If the SymLinksIfOwnersMatch Option is enabled in a directory, then each stat() call that results in a symlink will also have lstat() called on it.

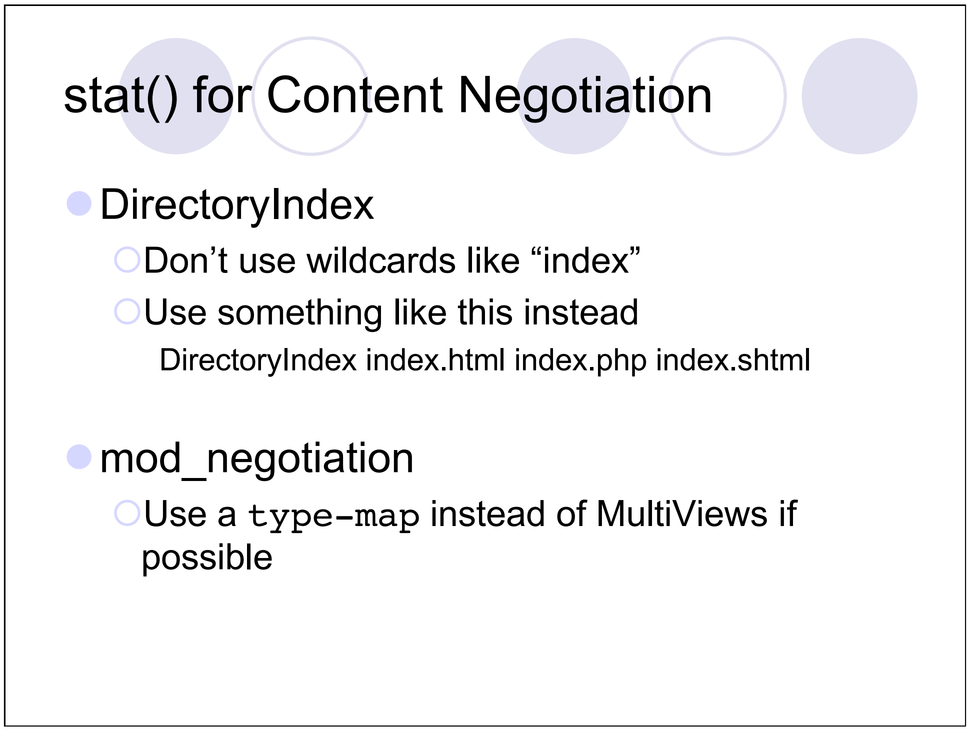
To reduce the number of system calls per request, avoid using symlinks, or try to isolate them to a place that you trust so that you do not have to incur the extra stat()+lstat() penalty across the entire DocumentRoot hierarchy.



stat() for .htaccess files

- AllowOverride
  - stat() for .htaccess in each path component of a request
  - Happens for any AllowOverride
  - Try to disable or limit to specific sub-dirs
  - Avoid use at the DocumentRoot

If you don’t want any .htaccess support, use “AllowOverride None” and you will get the best performance. If that is not possible, try to restruct .htaccess support to a certain part of your directory-space.



stat() for Content Negotiation

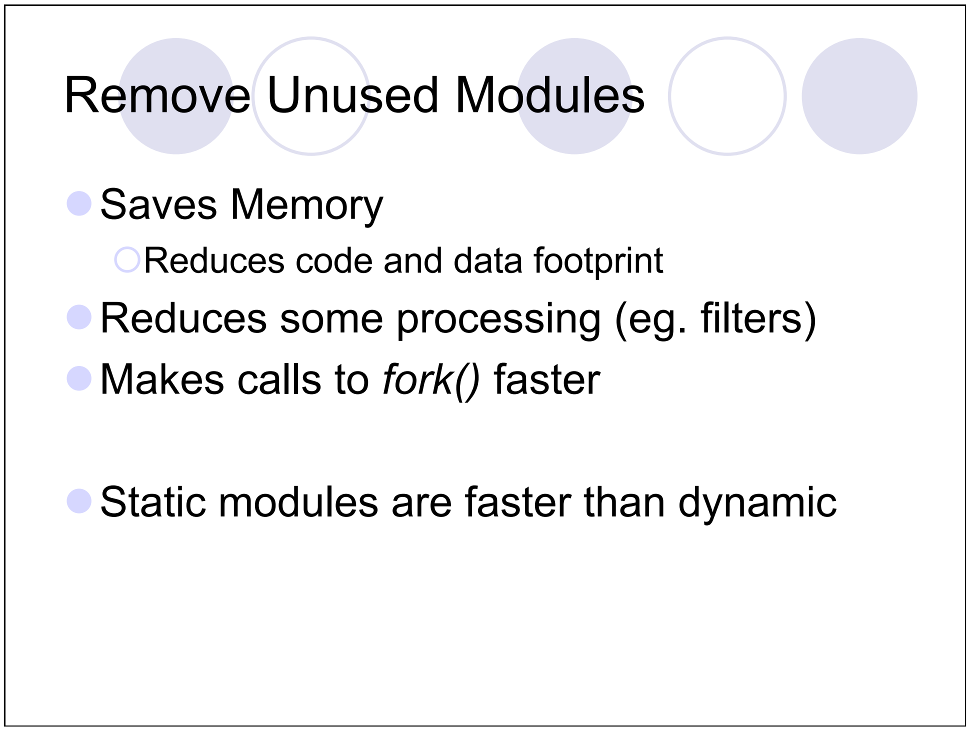
- DirectoryIndex
  - Don’t use wildcards like “index”
  - Use something like this instead

    DirectoryIndex index.html index.php index.shtml

- mod_negotiation
  - Use a `type-map` instead of MultiViews if possible

DirectoryIndex is a very handy feature of Apache, but if you want to squeeze a little more performance out of Apache then you might considering disabling it and referring to all pages directly.

An in-depth discussion of the features and benefits of mod_negotiation is outside the scope of this presentation, however if you wish to use this wonderful feature, you might consider using a type-map instead of MultiViews for the sake of performance.



Remove Unused Modules

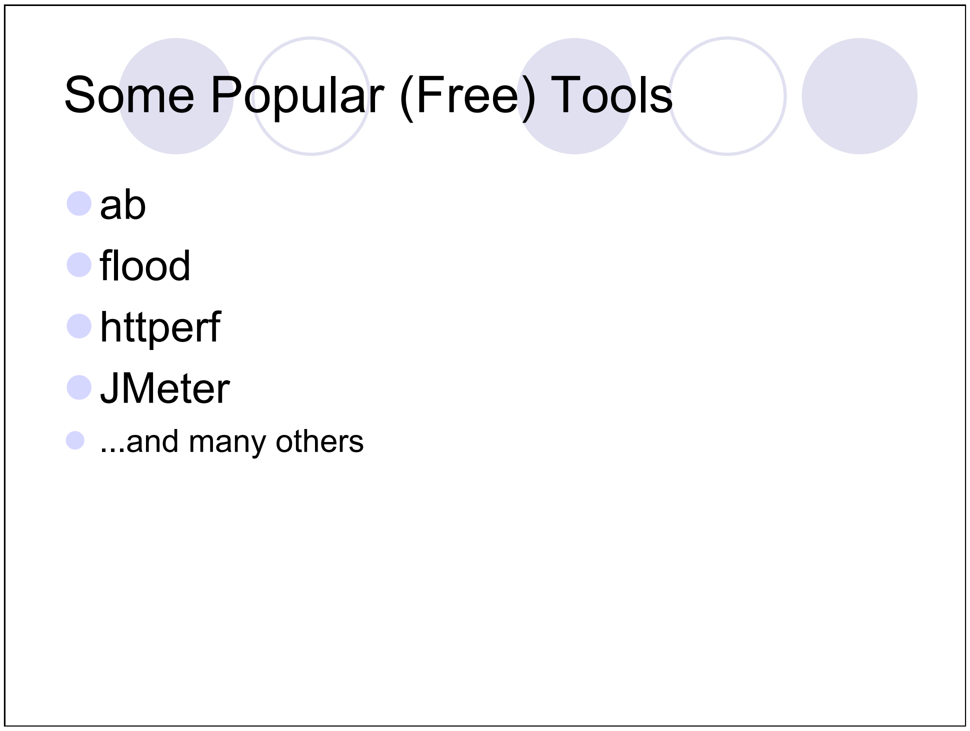
- Saves Memory
  - Reduces code and data footprint
- Reduces some processing (eg. filters)
- Makes calls to *fork()* faster

- Static modules are faster than dynamic

Read the documentation about build-configuration. Specifically the configure parameter --enable-mods-static, --enable-static-apr, etc are useful for improving performance in a stable server configuration where you don't need dynamic module support or dynamic library support.

Testing Performance

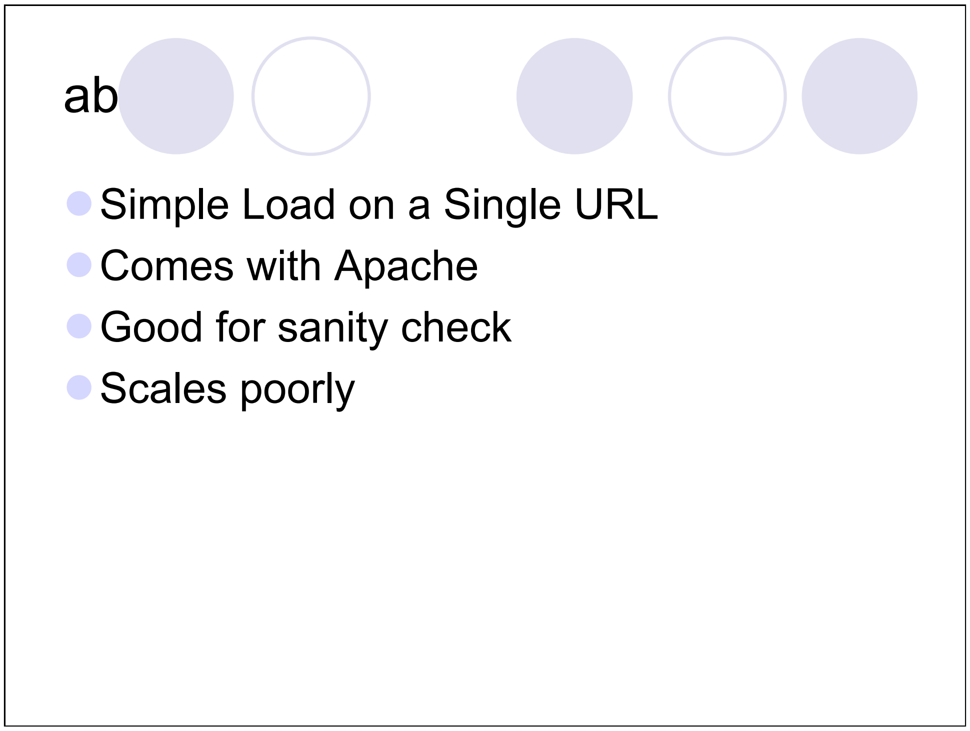
Benchmarking Tools



Some Popular (Free) Tools

- ab
- flood
- httperf
- JMeter
- ...and many others

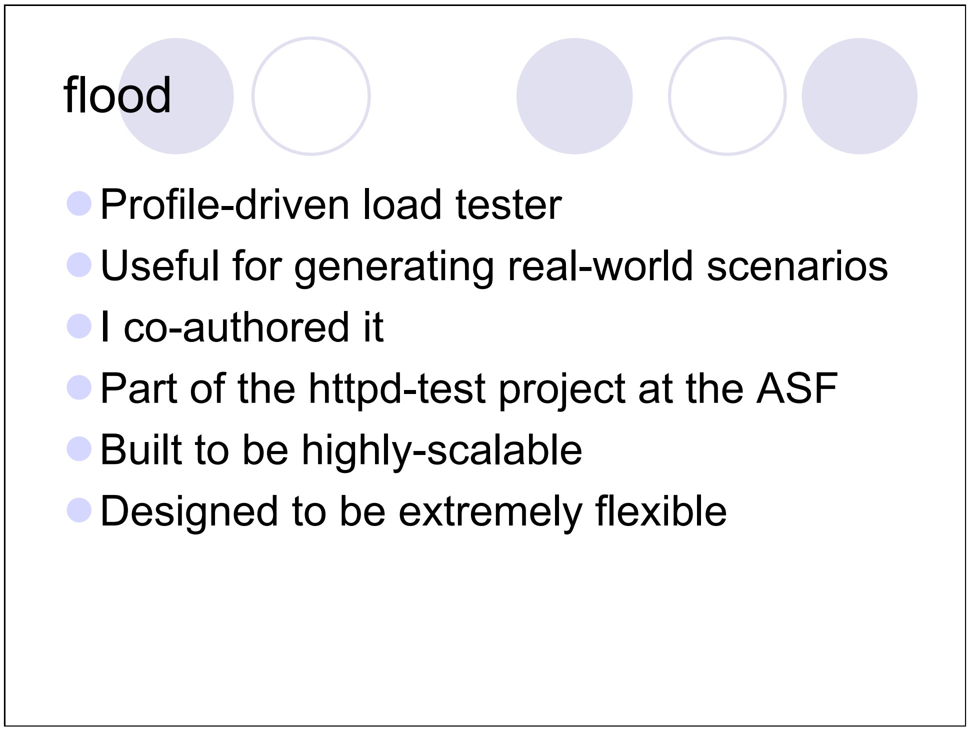
Note that these are only freely-available or open-source benchmarking tools. There are a number of commercial tools available, but those are outside the scope of this presentation.



ab

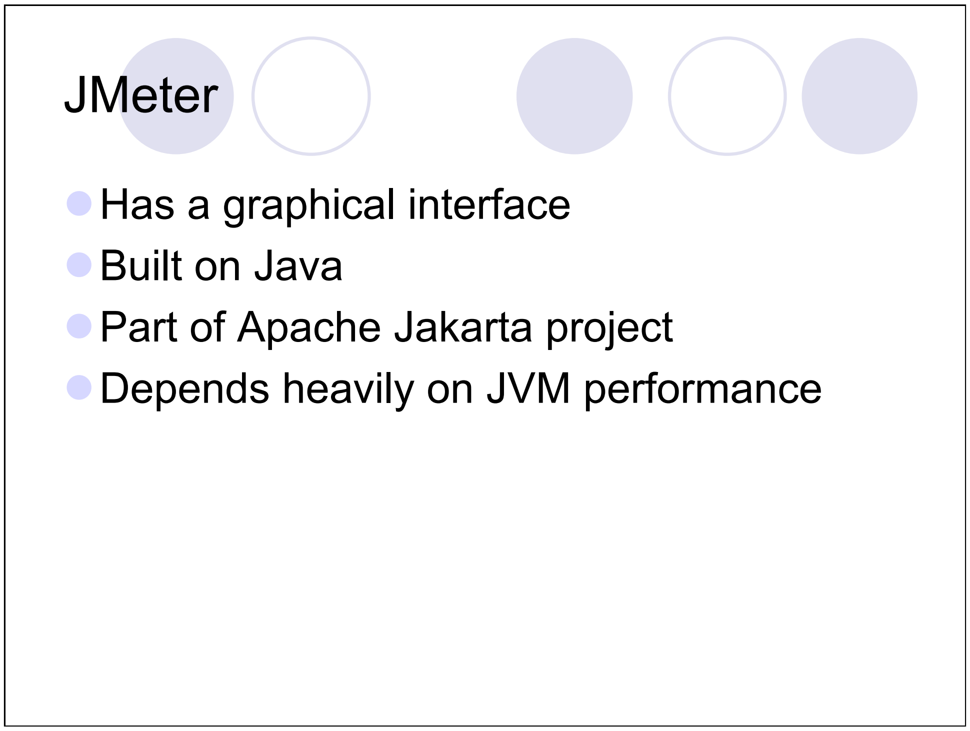
- Simple Load on a Single URL
- Comes with Apache
- Good for sanity check
- Scales poorly

Ab is probably already installed on your system. It is great for a quick check to see if your system can stand up to minimal load.



flood

- Profile-driven load tester
- Useful for generating real-world scenarios
- I co-authored it
- Part of the httpd-test project at the ASF
- Built to be highly-scalable
- Designed to be extremely flexible

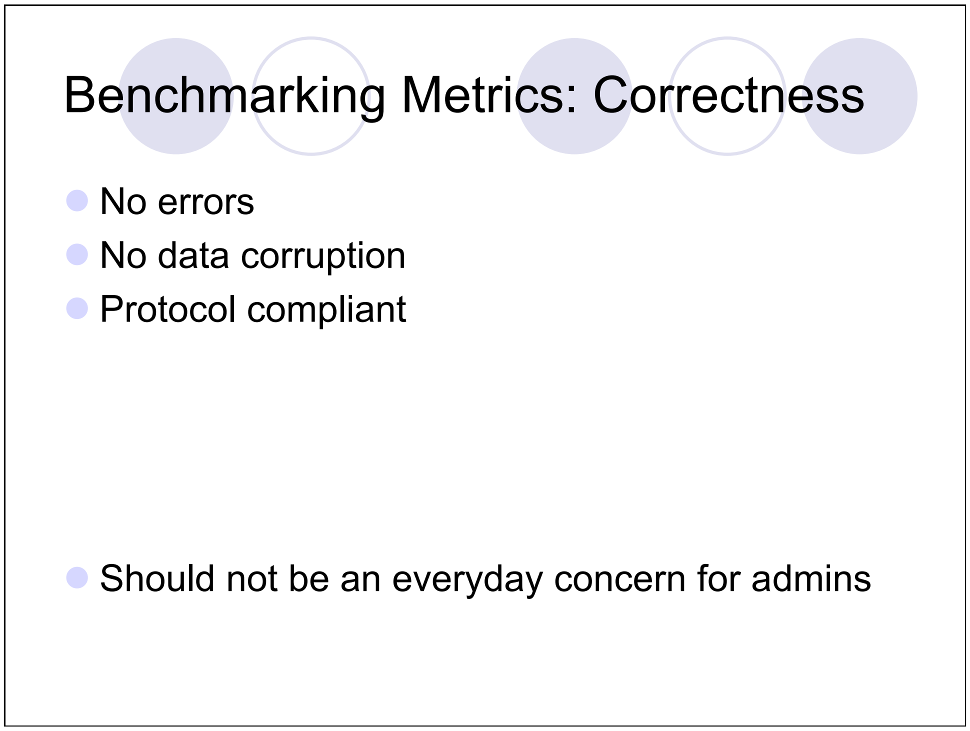


JMeter

- Has a graphical interface
- Built on Java
- Part of Apache Jakarta project
- Depends heavily on JVM performance

Benchmarking Metrics

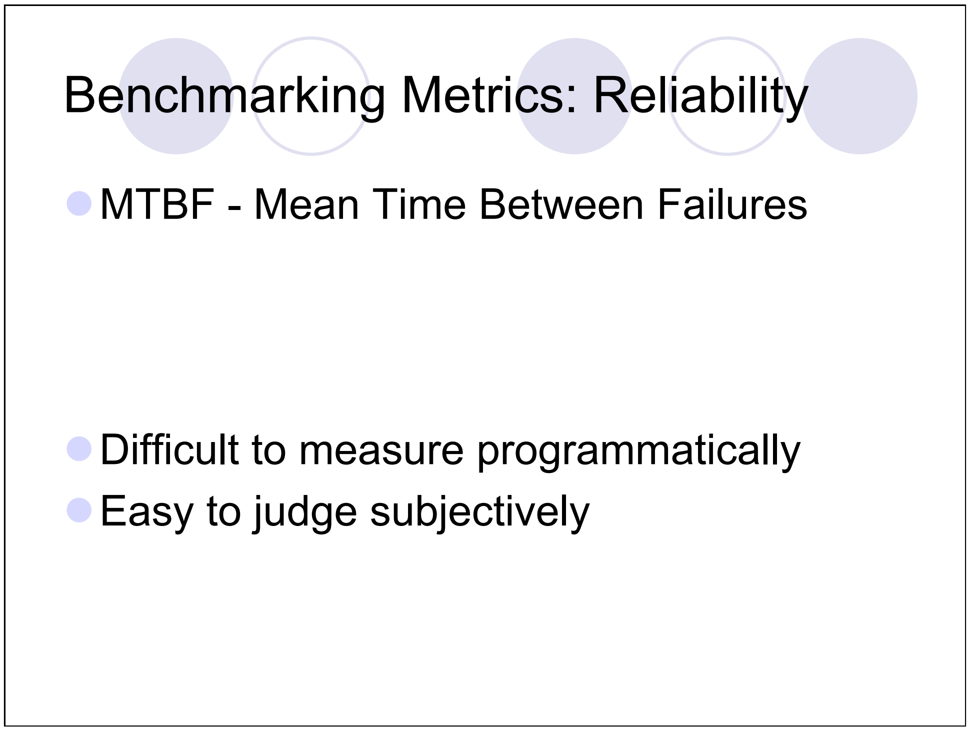
- What are we interested in testing?
  - Recall that we want our web server to be
    - Correct
    - Reliable
    - Scalable
    - Stable
    - Fast



Benchmarking Metrics: Correctness

- No errors
- No data corruption
- Protocol compliant

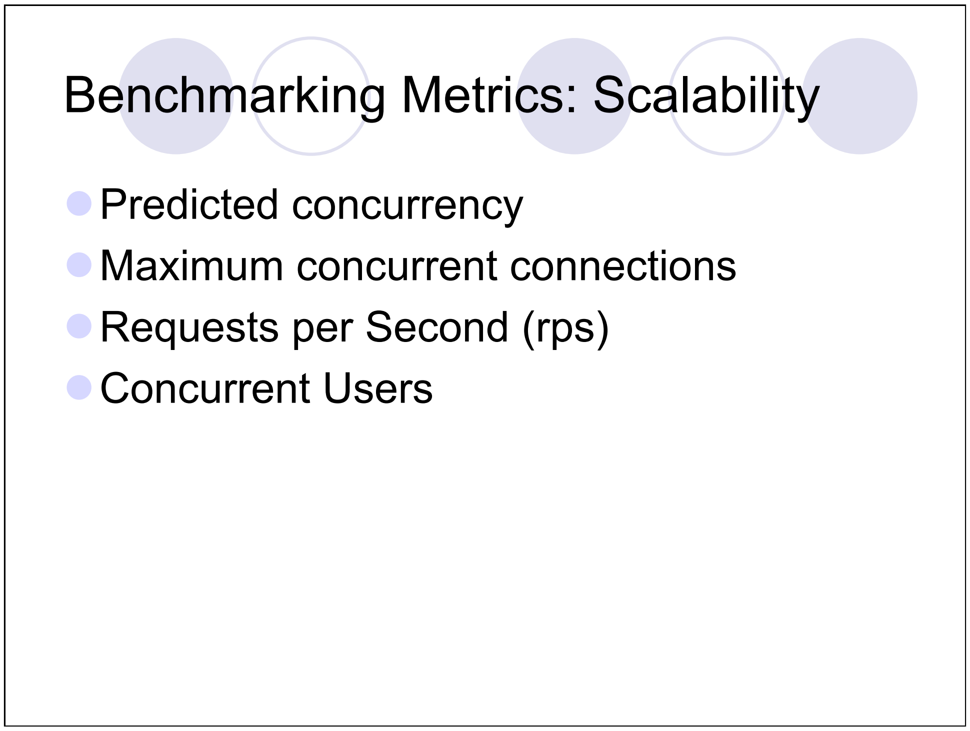
- Should not be an everyday concern for admins



Benchmarking Metrics: Reliability

- MTBF - Mean Time Between Failures
- Difficult to measure programmatically
- Easy to judge subjectively

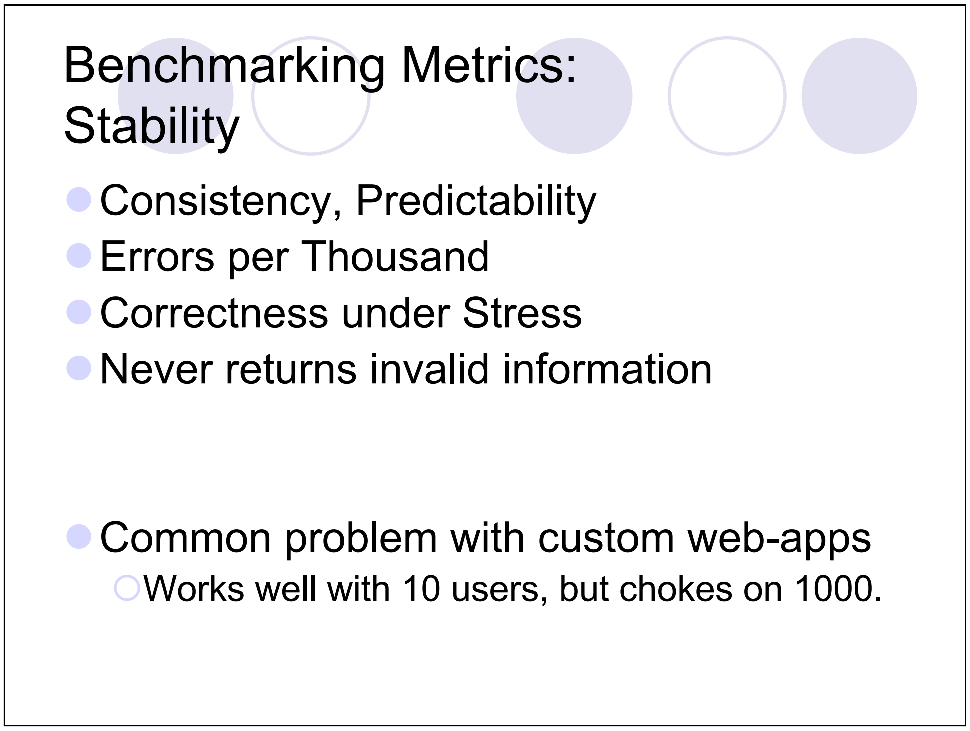
If you're spending a lot of time babying your system, monitoring log files and restarting Apache, then you might have a reliability problem. On the other hand, if the system runs so smoothly that you hardly notice it, then chances are you built yourself a reliable system. This isn't so much something that can change overnight, but is more a long-term goal of every administrator.



Benchmarking Metrics: Scalability

- Predicted concurrency
- Maximum concurrent connections
- Requests per Second (rps)
- Concurrent Users

The main difference between the RPS and the Concurrent User count is the way the website is used. Typical users will load a page containing many graphics, and then they will appear idle as they interact with that page. The server is only involved when there is network activity, when a page is being loaded. The number of Concurrent Users is the total number of users visiting the site over any time period, even if they are idly reading the text of the webpage and not using network or server resources.

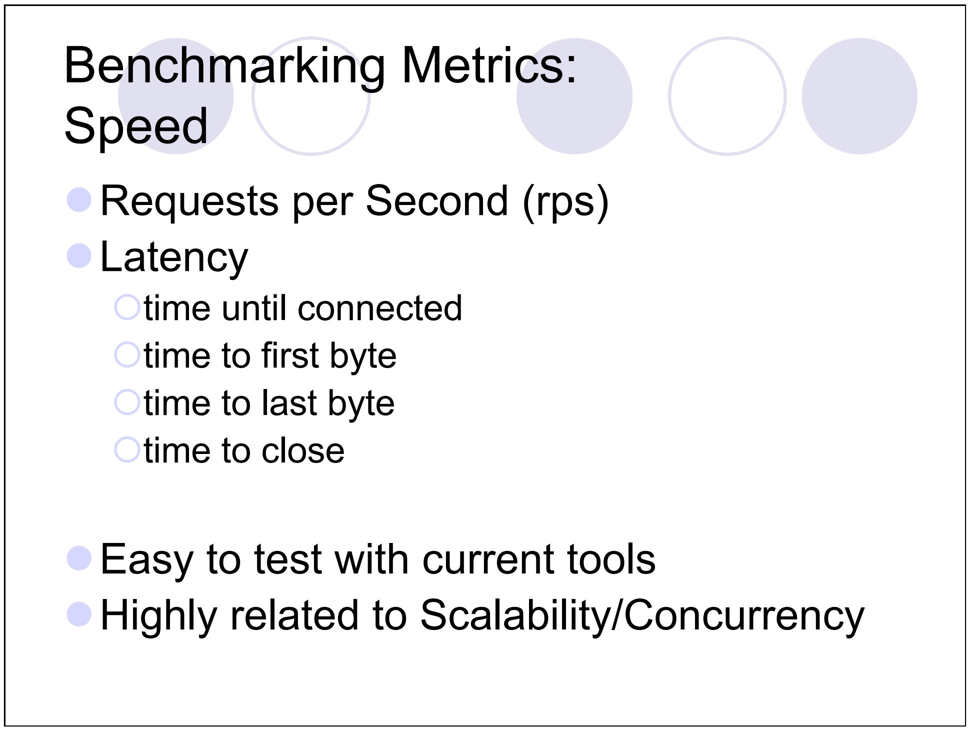


Benchmarking Metrics: Stability

- Consistency, Predictability
- Errors per Thousand
- Correctness under Stress
- Never returns invalid information

- Common problem with custom web-apps
  - Works well with 10 users, but chokes on 1000.

Predicting all the ways your server will be stressed is very difficult. At best one might hope to cover enough ground to be able to predict how the server will respond under a few important conditions, and feel confident that it will remain healthy. The worse case scenario is a deployment that fails catastrophically under very-high load but not under artificial load conditions.



Benchmarking Metrics: Speed

- Requests per Second (rps)
- Latency
  - time until connected
  - time to first byte
  - time to last byte
  - time to close
- Easy to test with current tools
- Highly related to Scalability/Concurrency

Latency is a very important factor in the user experience. Pages that snap in to place quickly allow a user to quickly navigate a site without any delays. Even a short delay can frustrate a user. The longer a delay, the more likely a user will try to reload a page in their browser, or perhaps leave the site entirely.

Method

1. Define the problem
   eg. Test Max Concurrency, Correctness, etc...
2. Narrow the scope of the problem
   Simplify the problem
3. Use tools to collect data
4. Come up with a hypothesis
5. Make minimal changes, retest

Troubleshooting

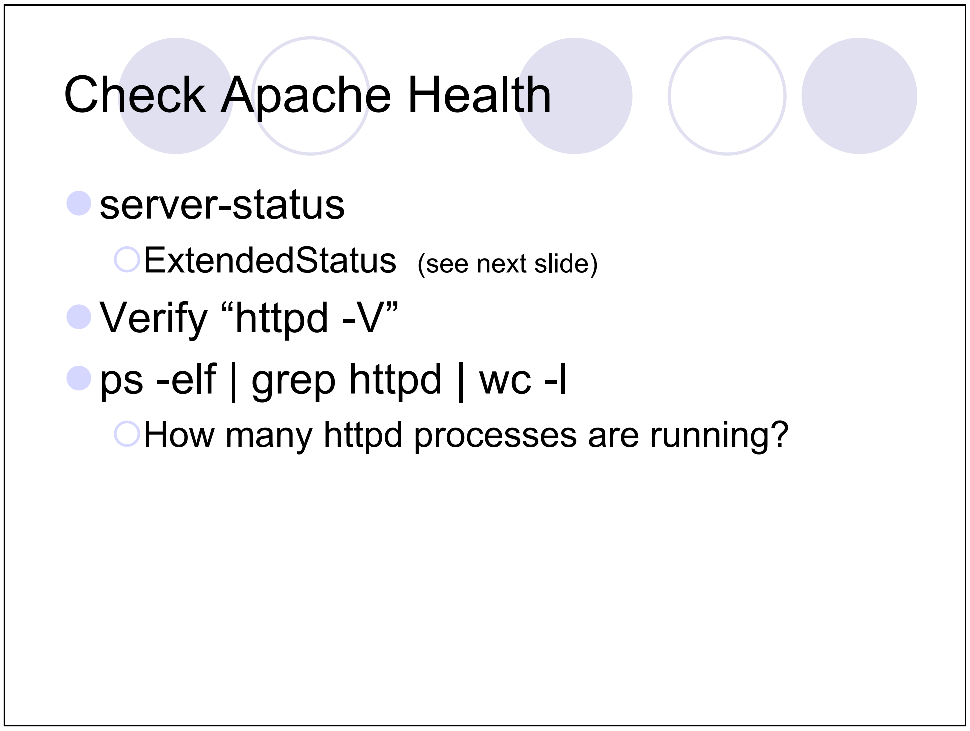
Common pitfalls
and their solutions

Check your error_log

- The first place to look
- Increase the LogLevel if needed
  - Make sure to turn it back down (but not off) in production

Check System Health

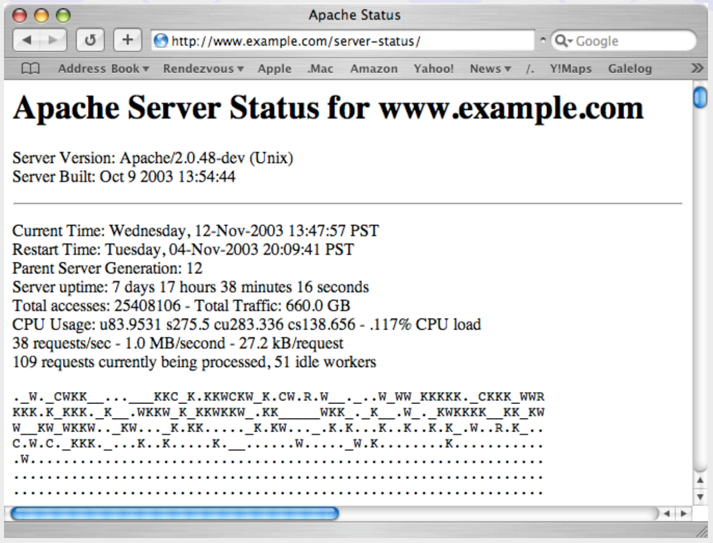
- vmstat, systat, iostat, mpstat, lockstat, etc...
- Check interrupt load
  - NIC might be overloaded
- Are you swapping memory?
  - A web server should ***never*** swap
- Check system logs
  - /var/log/message, /var/log/syslog, etc...



Check Apache Health

- server-status
  - ExtendedStatus (see next slide)
- Verify “httpd -V”
- ps -elf | grep httpd | wc -l
  - How many httpd processes are running?

server-status Example

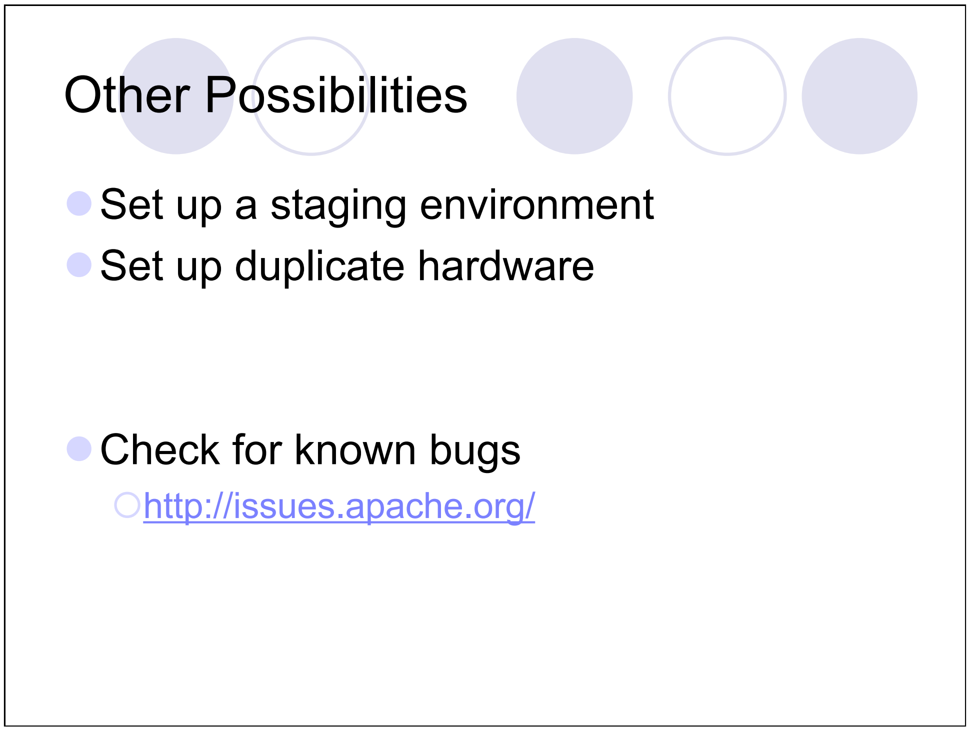


Apache Server Status for www.example.com

Server Version: Apache/2.0.48-dev (Unix)
Server Built: Oct 9 2003 13:54:44

Current Time: Wednesday, 12-Nov-2003 13:47:57 PST
Restart Time: Tuesday, 04-Nov-2003 20:09:41 PST
Parent Server Generation: 12
Server uptime: 7 days 17 hours 38 minutes 16 seconds
Total accesses: 25408106 - Total Traffic: 660.0 GB
CPU Usage: u83.9531 s275.5 cu283.336 cs138.656 - .117% CPU load
38 requests/sec - 1.0 MB/second - 27.2 kB/request
109 requests currently being processed, 51 idle workers

```
._W._CWKK__...___KKC_K.KKWCKW_K.CW.R.W__._..W_WW_KKKKK._CKKK_WWR
KKK.K_KKK._K__.WKKW_K_KKWKKW_.KK_____WKK_._K__.W_._KWKKKK__KK_KW
W__KW_WKKW.._KW..._K.KK....._K.KW..._.K.K...K..K..K.K_.W..R.K_..
C.W.C._KKK._...K..K.....K.__......W....._W.K........K...........
.W..............................................................
................................................................
................................................................
```



Other Possibilities

- Set up a staging environment
- Set up duplicate hardware

- Check for known bugs
  - http://issues.apache.org/

Common Bottlenecks

- No more File Descriptors
- Sockets stuck in TIME_WAIT
- High Memory Use (swapping)
- CPU Overload
- Interrupt (IRQ) Overload

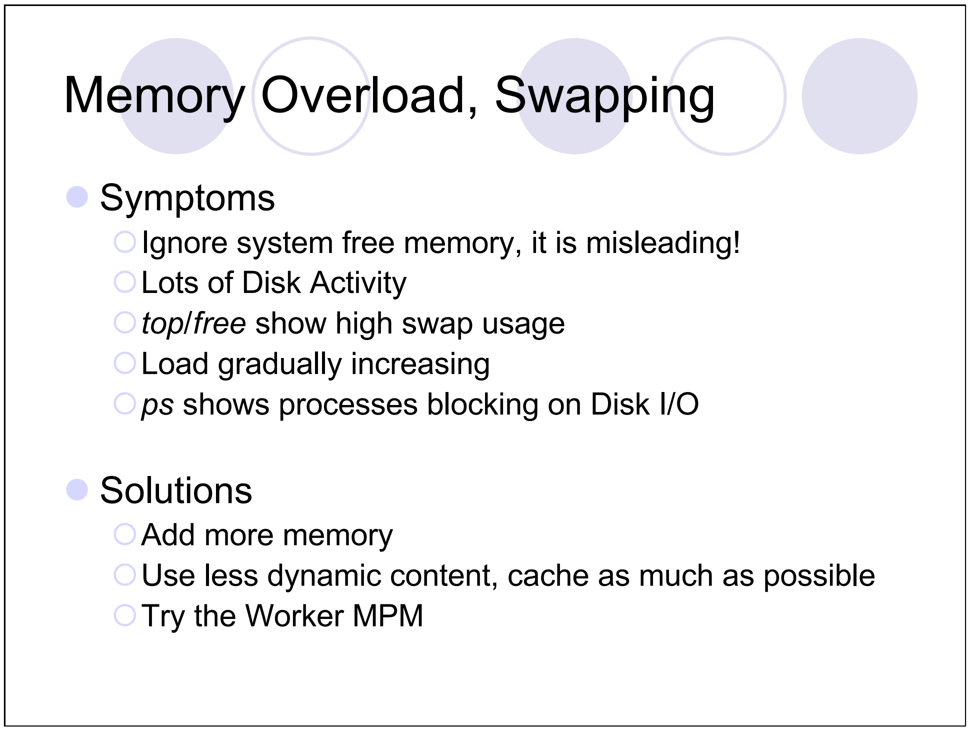
File Descriptors

- Symptoms
  - entry in error_log
  - new httpd children fail to start
  - fork() failing across the system

- Solutions
  - Increase system-wide limits
  - Increase ulimit settings in `apachectl`

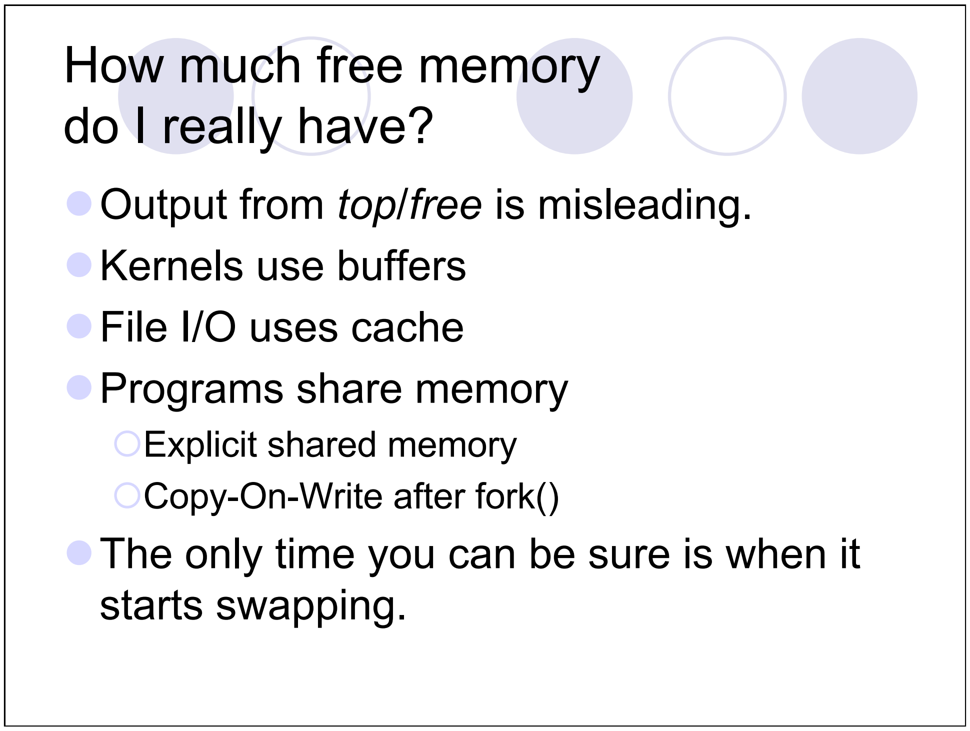
TIME_WAIT

- Symptoms
  - Unable to accept new connections
  - CPU under-utilized, httpd processes sit idle
  - Not Swapping
  - netstat shows huge numbers of sockets in TIME_WAIT
- Many TIME_WAIT are to be expected
- Only when new connections are failing is it a problem
  - Decrease system-wide TCP/IP FIN timeout



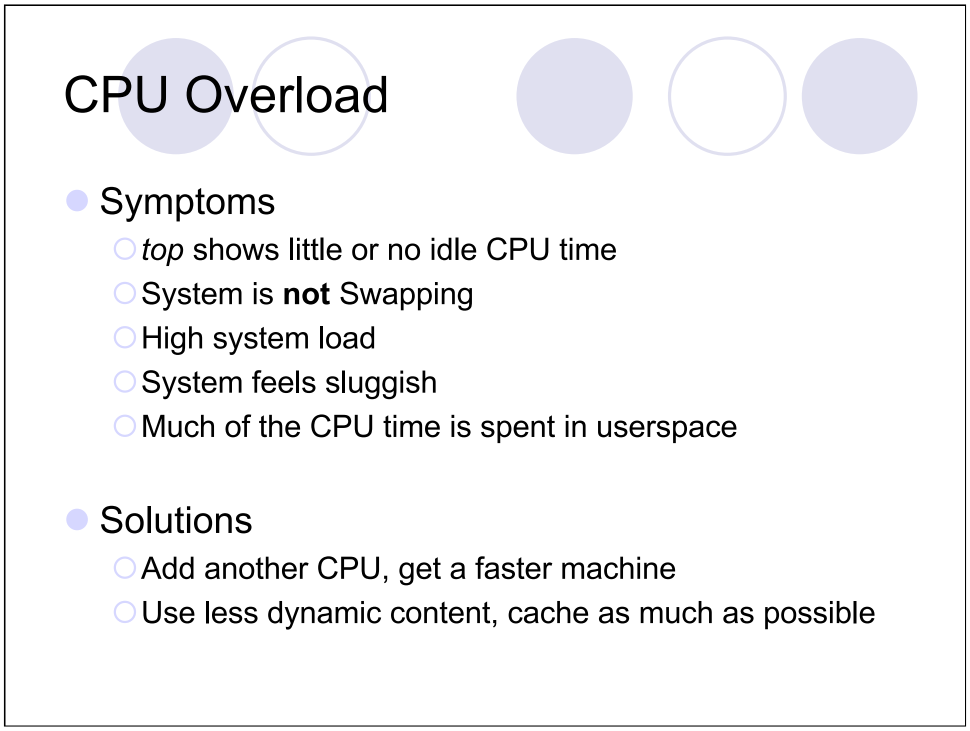
Memory Overload, Swapping

- Symptoms
  - Ignore system free memory, it is misleading!
  - Lots of Disk Activity
  - *top*/*free* show high swap usage
  - Load gradually increasing
  - *ps* shows processes blocking on Disk I/O
- Solutions
  - Add more memory
  - Use less dynamic content, cache as much as possible
  - Try the Worker MPM



How much free memory do I really have?

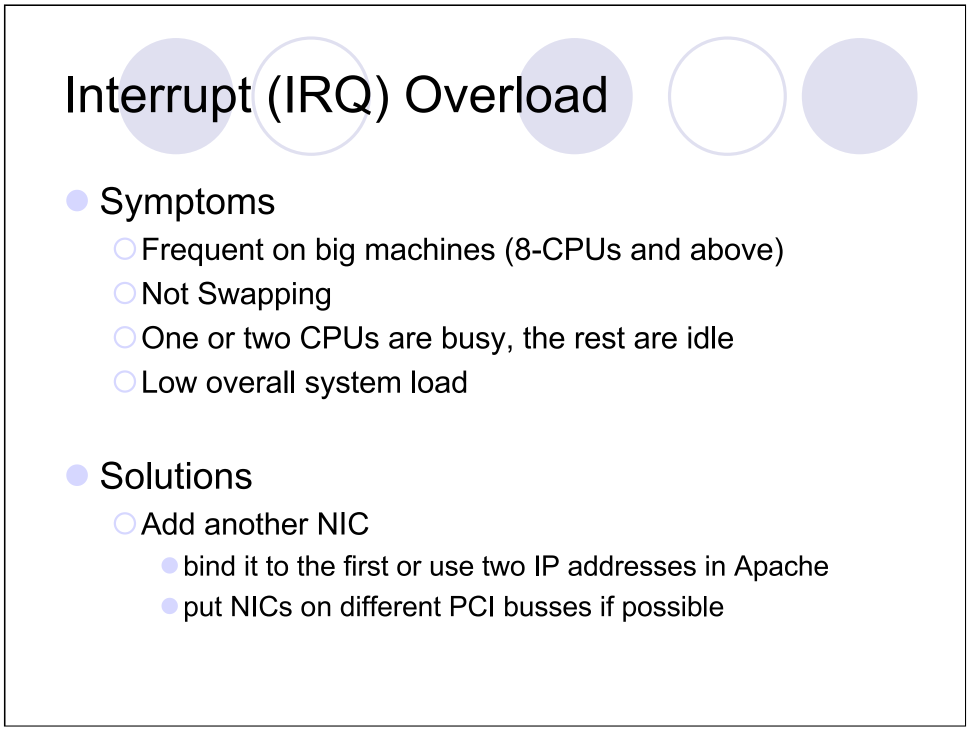
- Output from *top*/*free* is misleading.
- Kernels use buffers
- File I/O uses cache
- Programs share memory
  - Explicit shared memory
  - Copy-On-Write after fork()
- The only time you can be sure is when it starts swapping.



CPU Overload

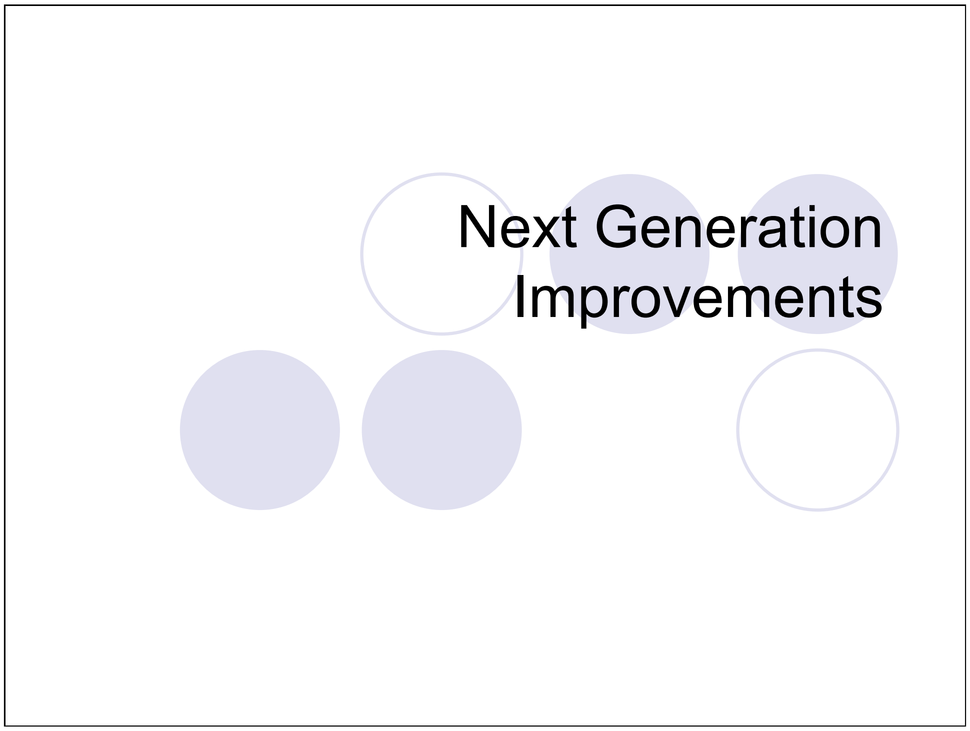
- Symptoms
  - *top* shows little or no idle CPU time
  - System is **not** Swapping
  - High system load
  - System feels sluggish
  - Much of the CPU time is spent in userspace

- Solutions
  - Add another CPU, get a faster machine
  - Use less dynamic content, cache as much as possible



Interrupt (IRQ) Overload

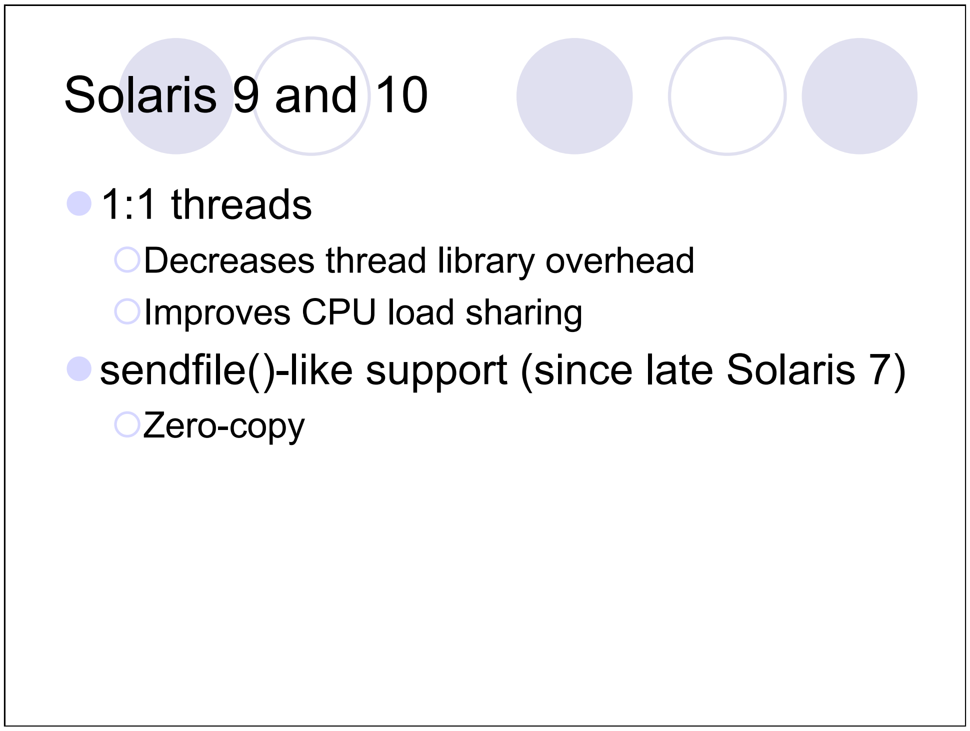
- Symptoms
  - Frequent on big machines (8-CPUs and above)
  - Not Swapping
  - One or two CPUs are busy, the rest are idle
  - Low overall system load
- Solutions
  - Add another NIC
    - bind it to the first or use two IP addresses in Apache
    - put NICs on different PCI busses if possible



Next Generation Improvements

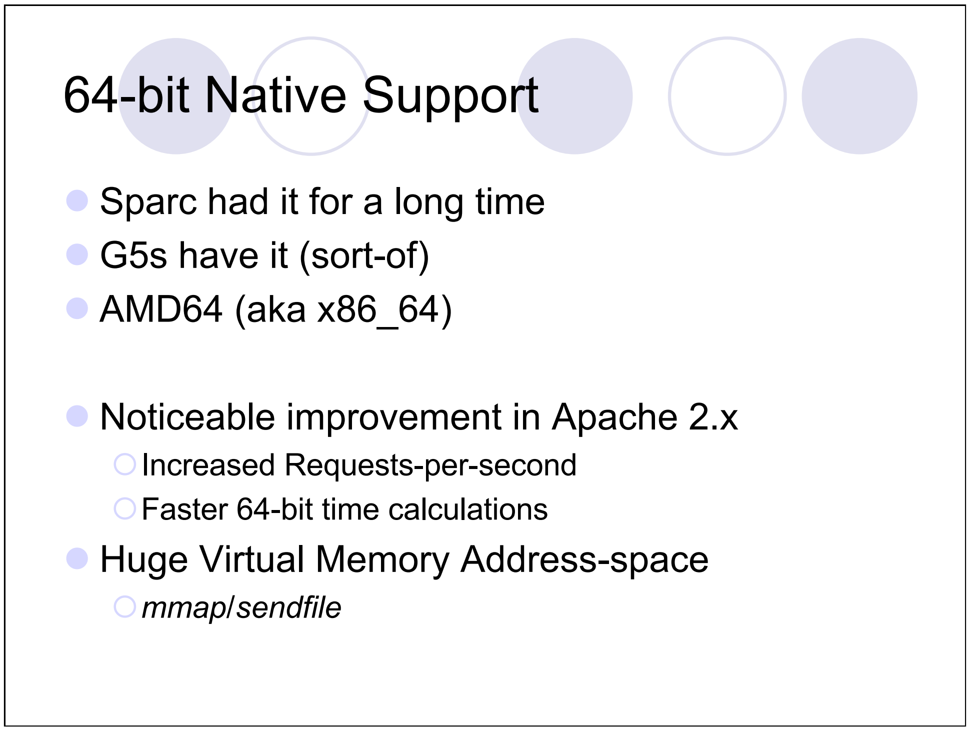
Linux 2.6

- NPTL and NGPT
  - Next-Gen Thread Libraries for Linux
  - Available in most modern Linux distros
- O(1) scheduling patch
- Preemptive Kernel patch
- All improvements affect Apache, but the Worker MPM will likely be the most affected.



Solaris 9 and 10

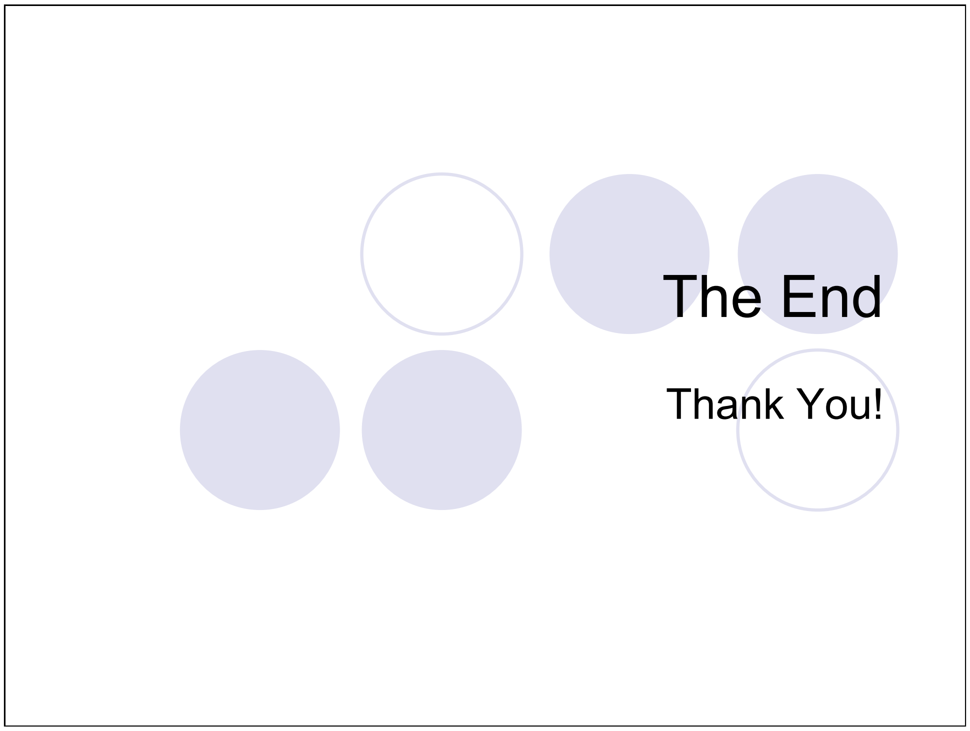
- 1:1 threads
  - Decreases thread library overhead
  - Improves CPU load sharing
- sendfile()-like support (since late Solaris 7)
  - Zero-copy



64-bit Native Support

- Sparc had it for a long time
- G5s have it (sort-of)
- AMD64 (aka x86_64)

- Noticeable improvement in Apache 2.x
  - Increased Requests-per-second
  - Faster 64-bit time calculations
- Huge Virtual Memory Address-space
  - *mmap*/*sendfile*



The End

Thank You!