Building Scalable Web Architectures

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

http://www.codemass.com/~aaron/presentations/apachecon2005/ac2005scalablewebarch.ppt

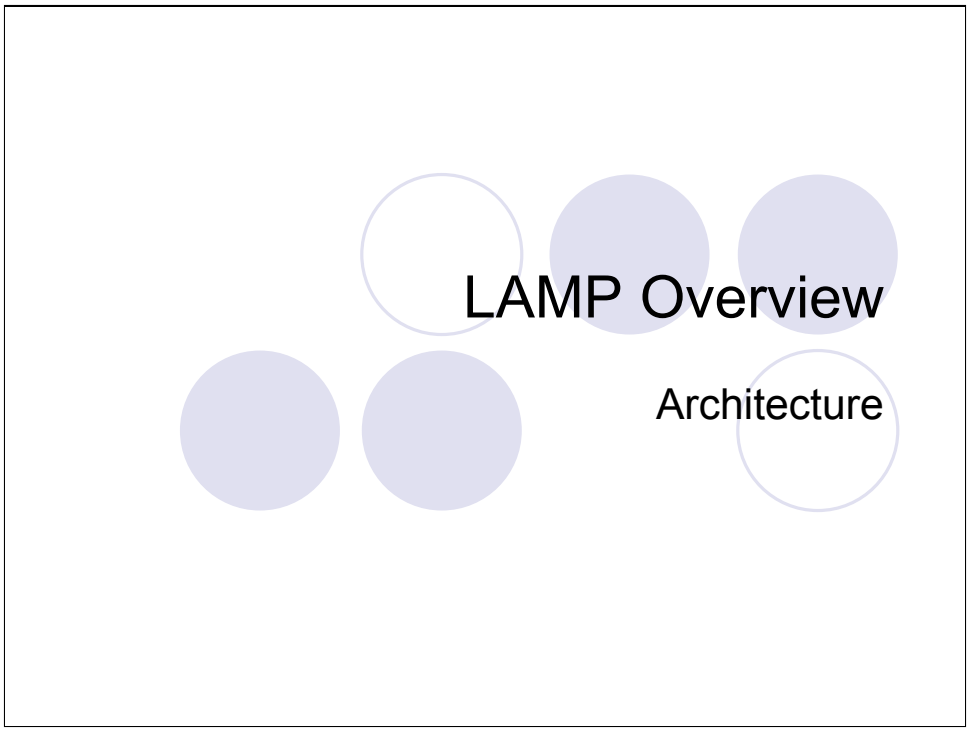
Goal

- How do we build a massive web system out of commodity parts and open source?

Agenda

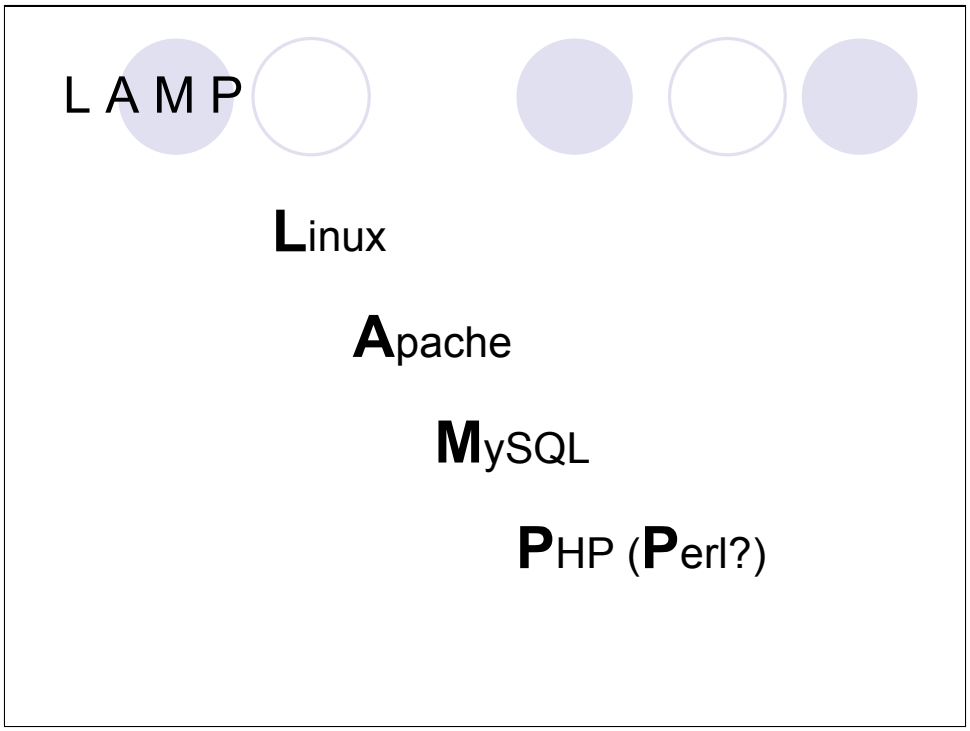
1. LAMP Overview
2. LAMP Features
3. Performance
4. Surviving your first Slashdotting
    a. Growing above a single box
    b. Avoiding Bottlenecks

This is just an overview of what will be presented today.

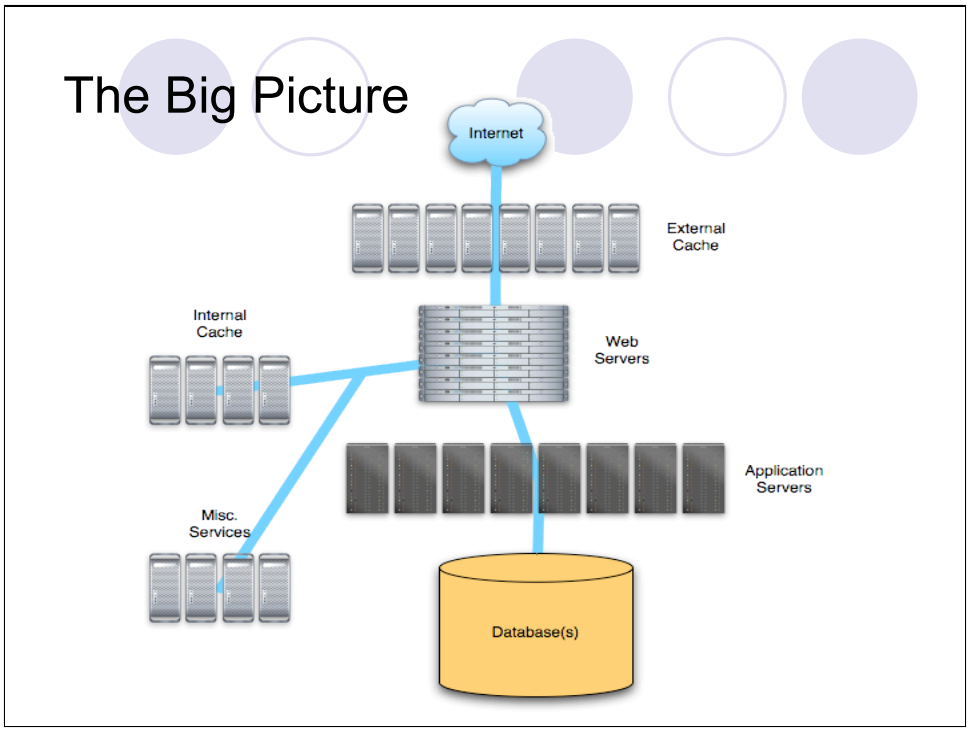


LAMP stands for Linux, Apache, MySQL and PHP (or Python).

At each level there are many alternatives, which I will go into more later on. First, let's review the big picture.

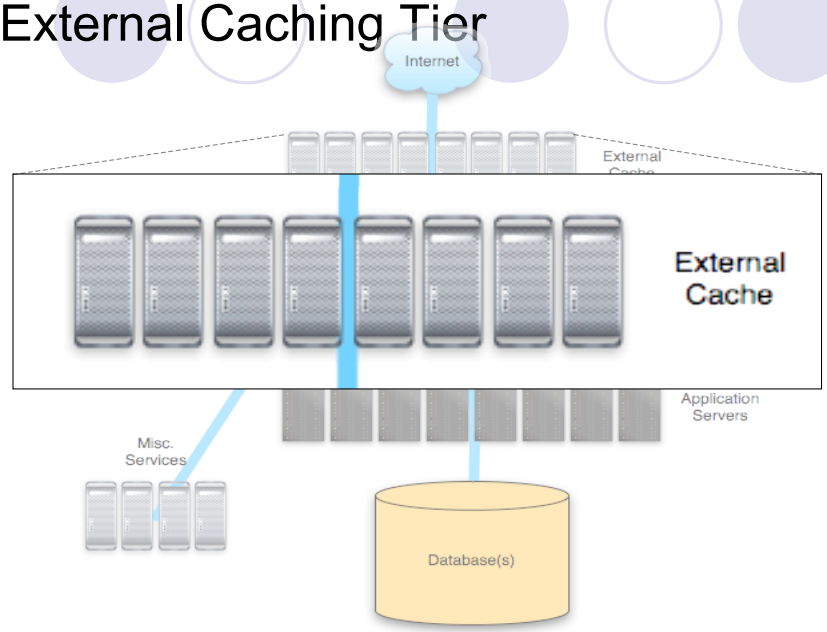


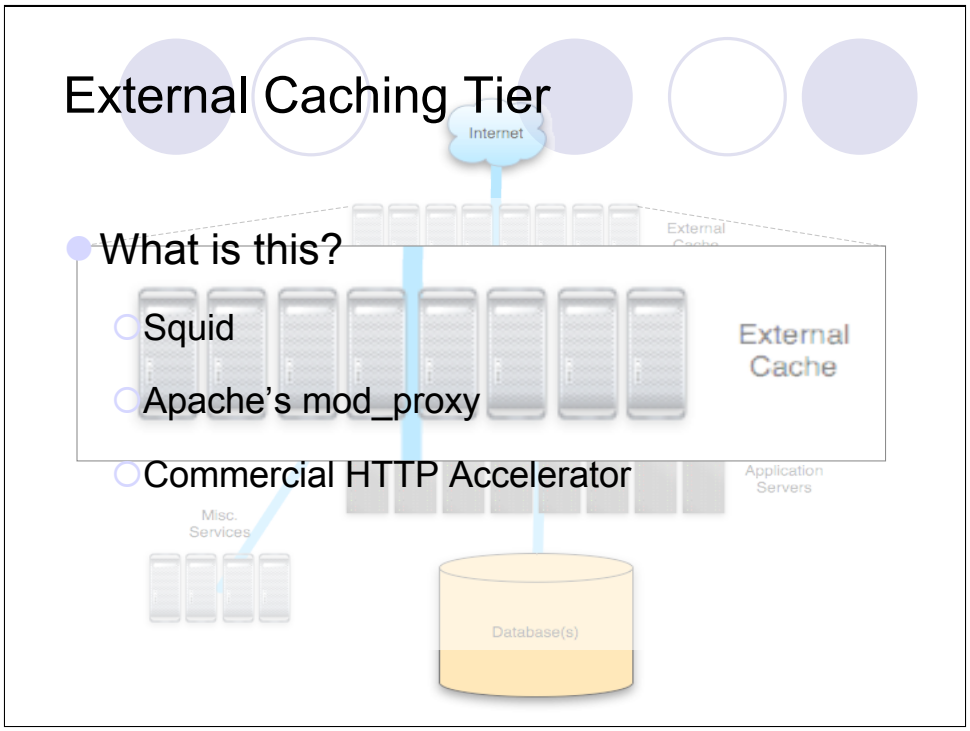
LAMP is an acronym which stands for Linux Apache MySQL PHP (or Perl, depending on who you ask).



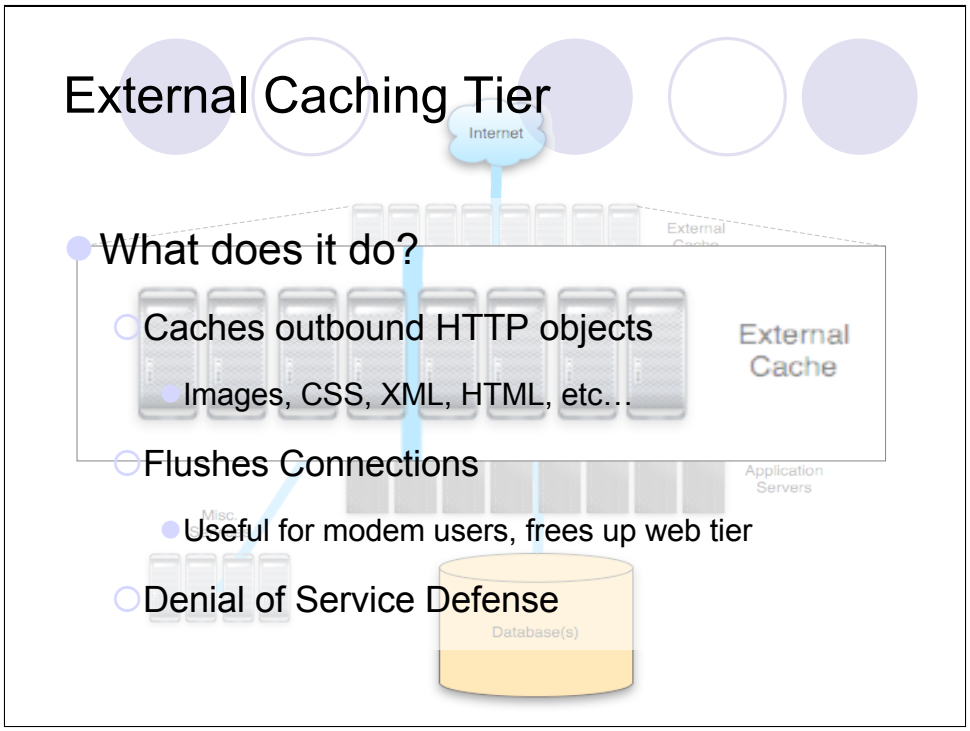
This is what a typical Web Architecture looks like these days.

External Caching Tier





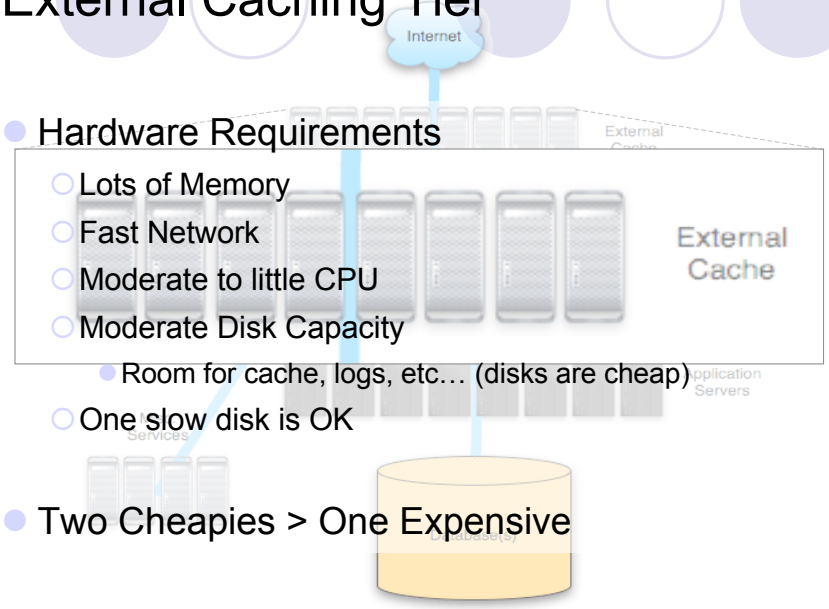
The main thing here is that all of these products operate only on the HTTP or HTTPS layer. There are SSL accelerators available that would fit here. Each of these systems are capable of caching, but they are confined to the rules defined by the HTTP specification.



Depending on the type of HTTP response being made, the Caching Tier may decide to store the object for future use. As soon as another request is made for the same object, the Caching Tier may decide to reuse the previous object, saving a round trip to the next layer. On other occasions, the Caching Tier may ignore the local object entirely and force a refresh of the object from the backend. All of these semantics are governed by the HTTP spec.

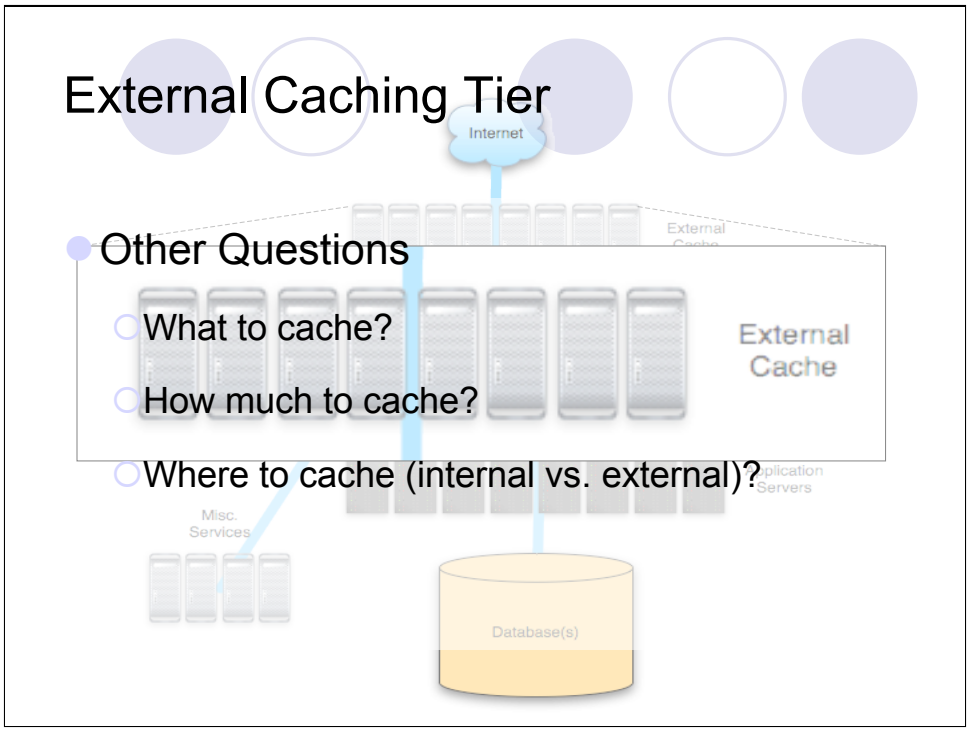
Also, since each connection to a web server uses a portion of that server's very finite resources, it's important to complete responses as quickly as possible. The Caching Tier has a nice side affect in that it quickly receives responses from the Web Tier, freeing it up very quickly, and can then slowly drain that response to the user. This is particularly useful for modem users, which, despite their slow speed, are actually quite a strain on high-traffic servers.

External Caching Tier



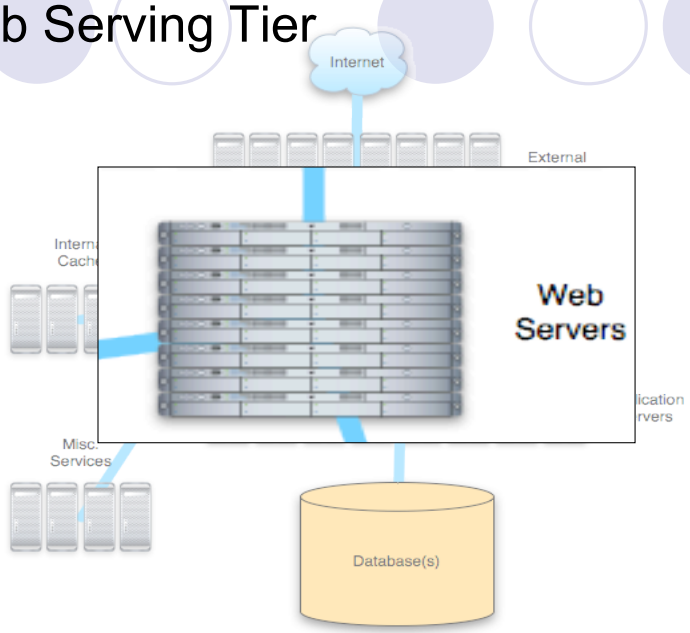
- Hardware Requirements
  - Lots of Memory
  - Fast Network
  - Moderate to little CPU
  - Moderate Disk Capacity
    - Room for cache, logs, etc… (disks are cheap)
  - One slow disk is OK
- Two Cheapies > One Expensive

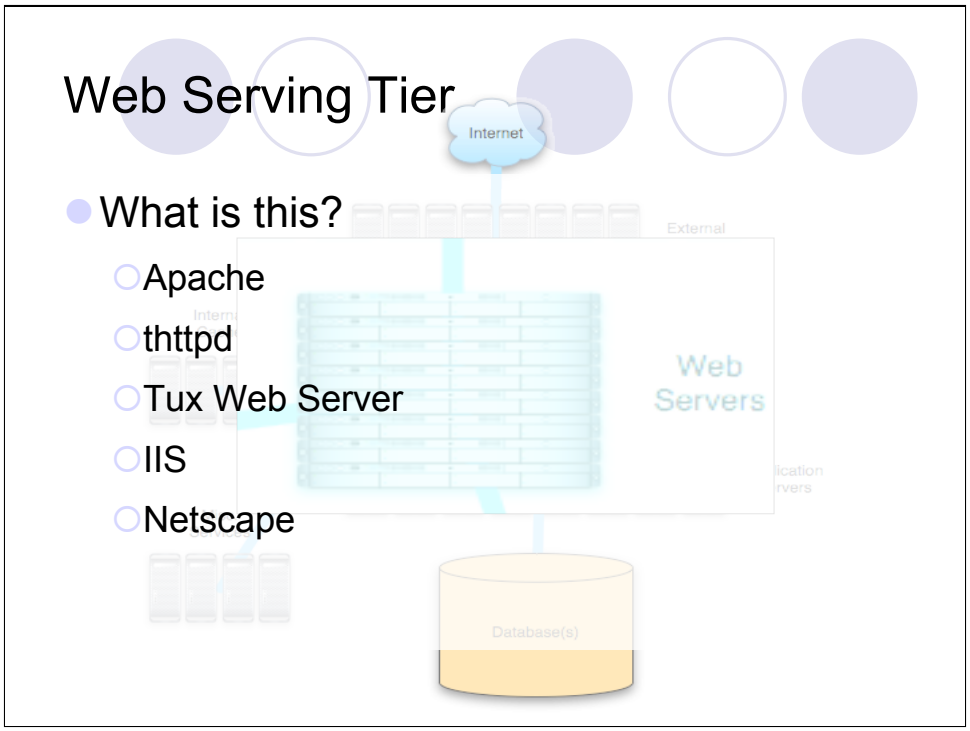
Buy lots of these for redundancy, and buy cheap parts (expect them to break).



The nice thing about HTTP is that the web server, aka the “origin server”, is the thing that gets to define what is cacheable and what is not. The Caching Tier should be entirely transparent, as far as the Users and the Web Servers are concerned. Therefore, the types of files to cache and the length of time to cache them should be defined by your web application (on a deeper layer).

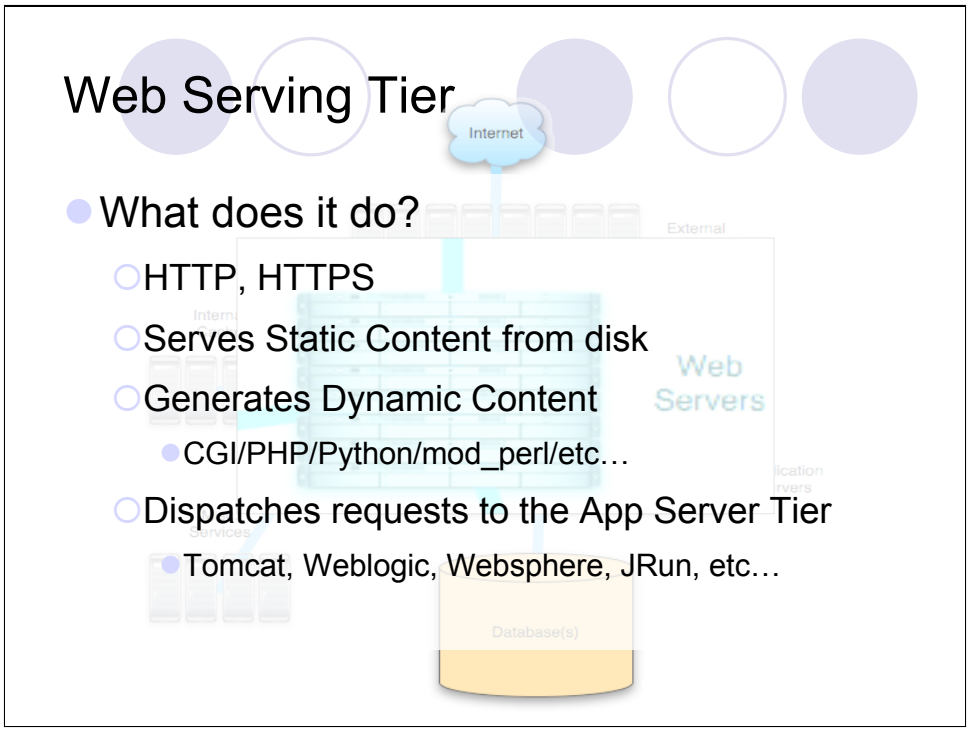
Web Serving Tier





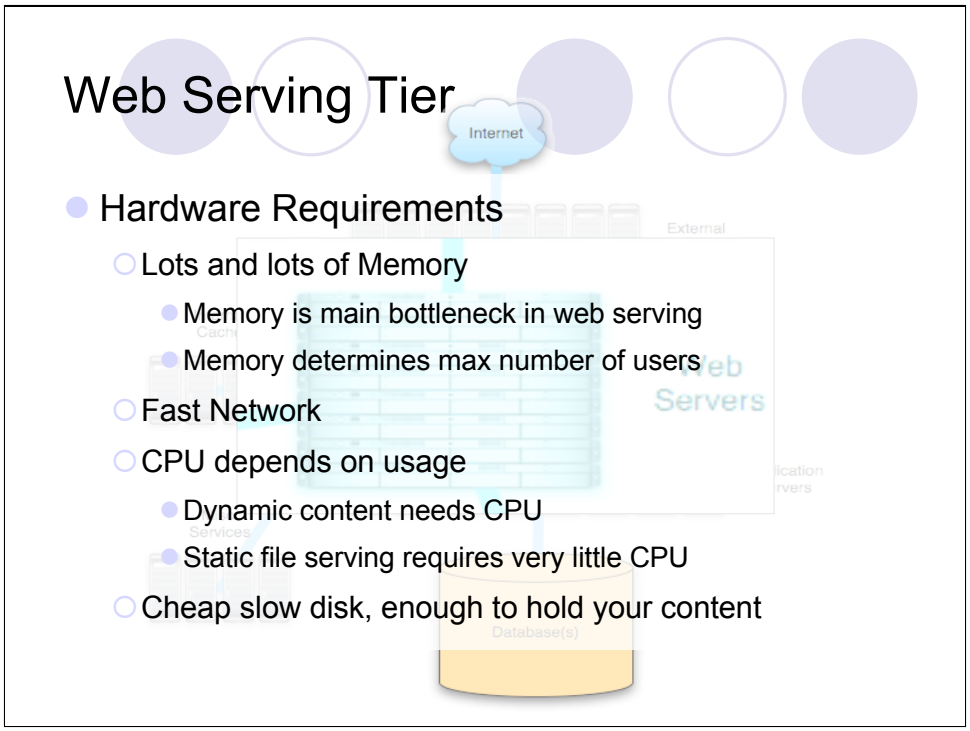
This is the “A” in LAMP.

Each of these have their strengths and weaknesses. The reason I believe Apache is the most popular because 1) it’s free 2) it’s very flexible (lots of modules are available) and 3) it has a strong support community.



The main three things that a web server does, are: 1) Serve Static Content, 2) Generate Dynamic Content, and 3) Dispatch Requests.

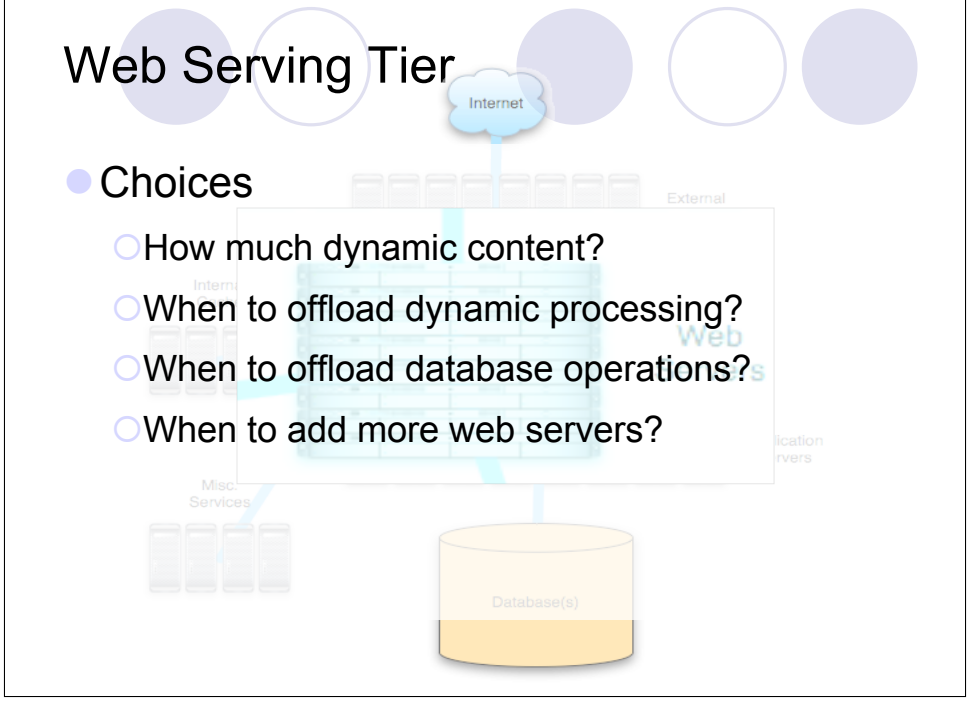
Secondary things that a web server may do are thing such as logging, user tracking, etc.



Web servers almost always run out of memory before then run out of anything else. With heavy dynamic content, memory usage can skyrocket. Apache 2.x can easily fill a 100-mbit pipe with a cheap desktop PC from a couple years ago, but only to one user at a time. To serve multiple simultaneous users, you need memory!

If your dynamic generators are consuming a lot of CPU, then you might want to optimize your code. Heavy SSL usage can put a lot of pressure on the CPU, and in that case you might want to consider using an SSL accelerator card. Normal HTML-generating web server applications (eg. PHP, mod_perl, etc…) don't typically consume much CPU under normal conditions.

Web Serving Tier

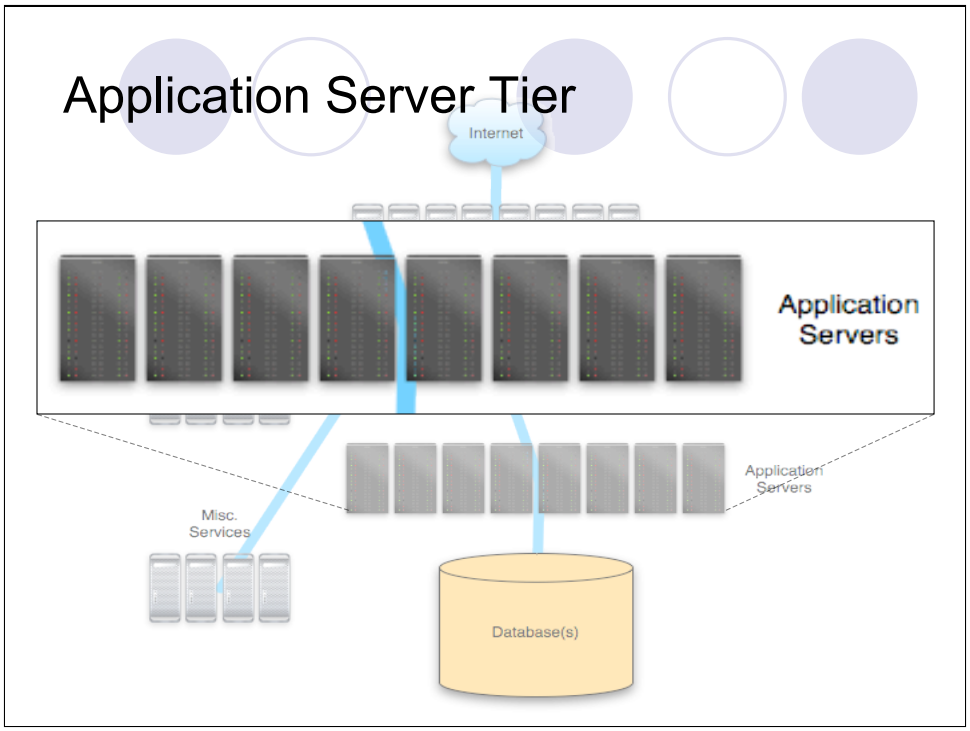


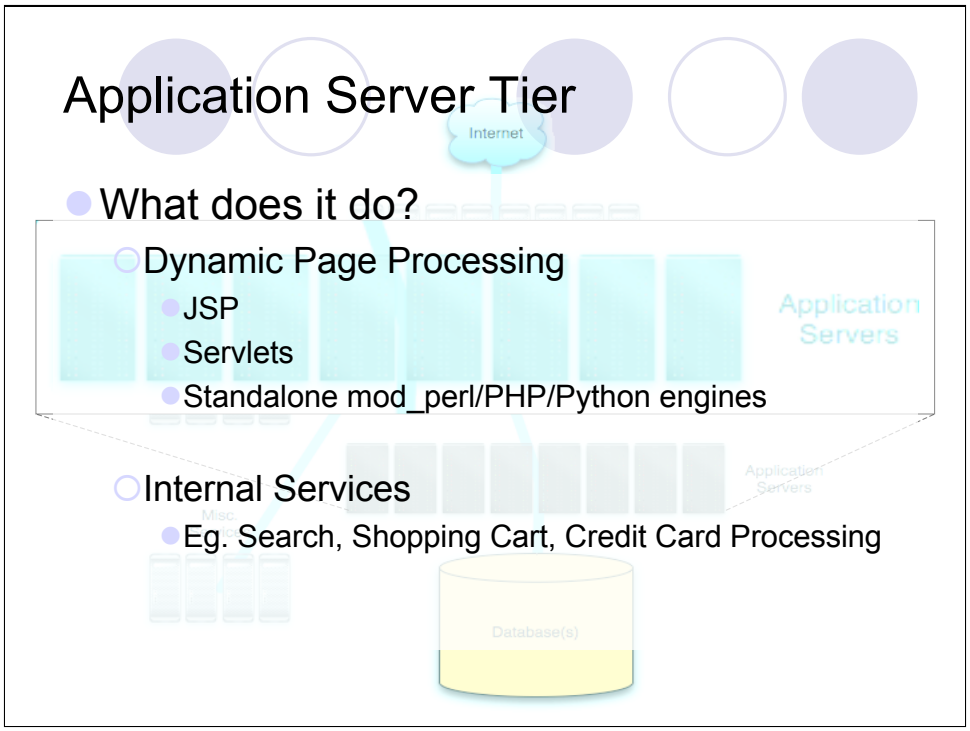
- Choices
  - How much dynamic content?
  - When to offload dynamic processing?
  - When to offload database operations?
  - When to add more web servers?

Dynamic content, as mentioned before, is very heavy on system resources. At some point you're going to want to split your dynamic components into smaller pieces, and possibly move them to dedicated systems. We'll talk more about that later however.

Knowing when to add more servers is difficult. Often, the systems will appear to be performing nominally, then one day they "fall off a cliff". Performance slowdowns tend to cause a cascading effect. Load spikes can also trigger this avalanche of slowdowns. Be aware of traffic spikes, performance slowdowns and other problems and troubleshoot them early to avoid service interruptions later.

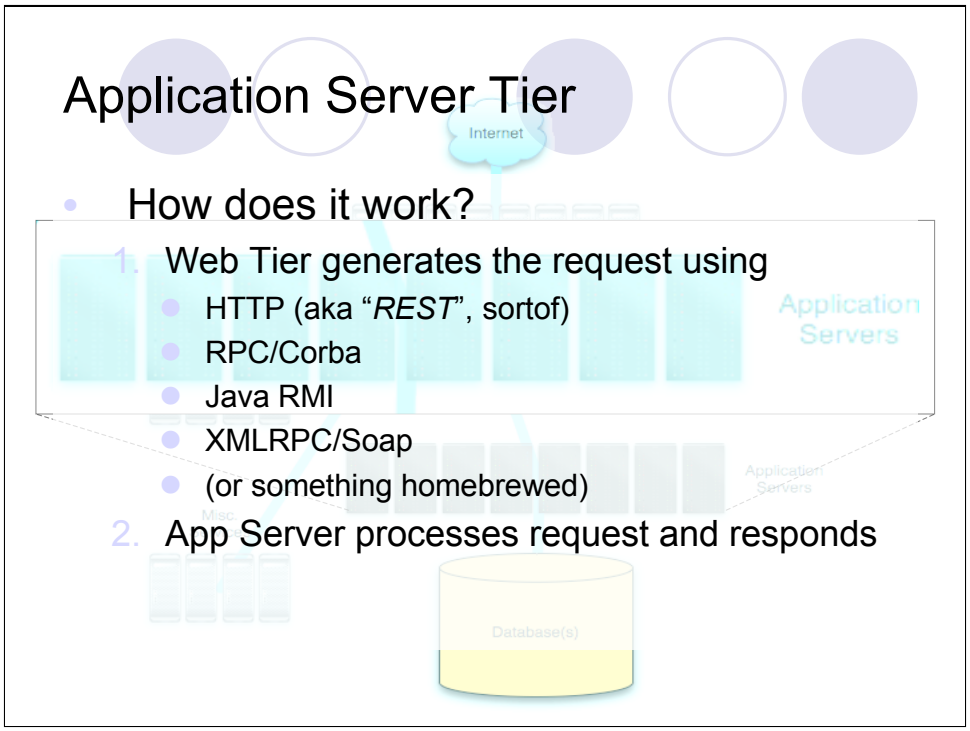
Application Server Tier



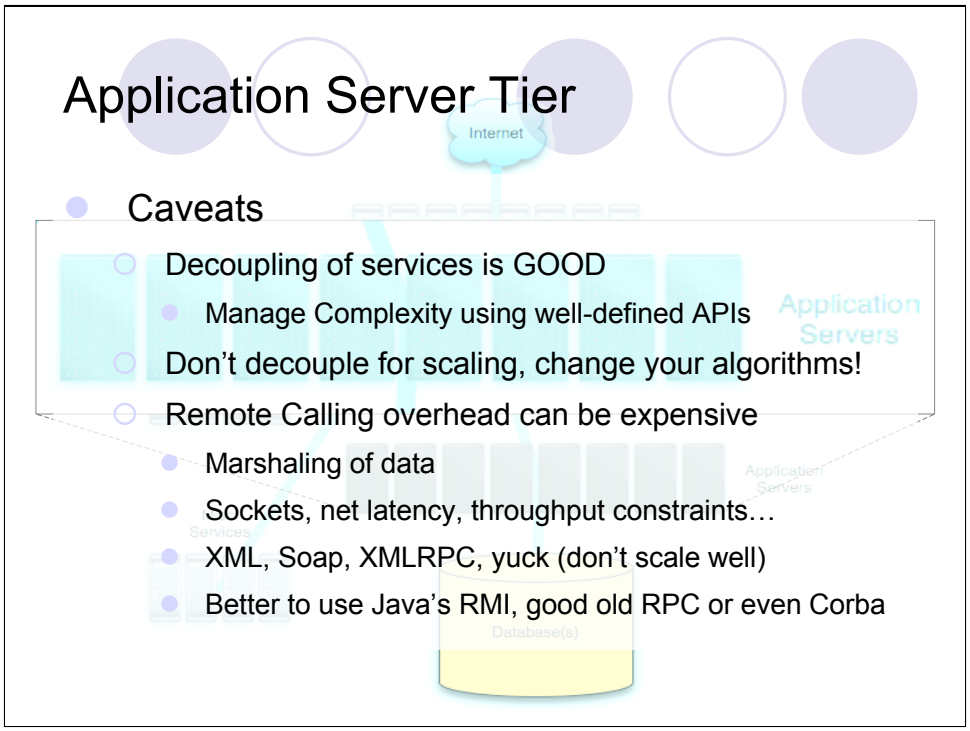


This is the “P” in LAMP.

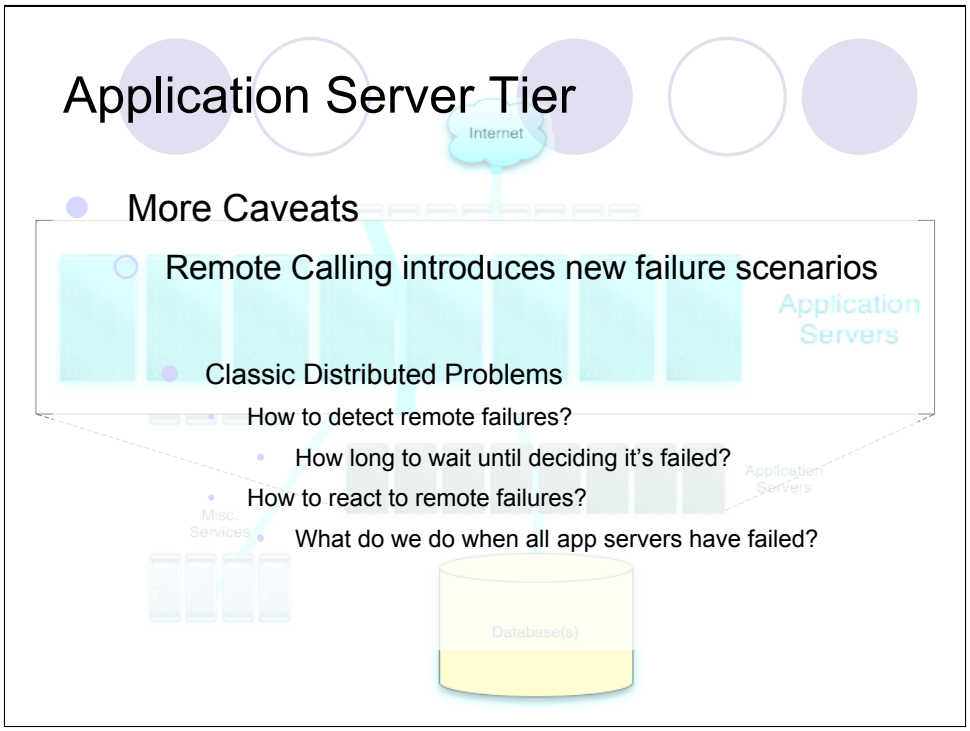
Application Servers are a convenient way of decoupling code. Separating code into distinct services helps manage complexity and can often reveal bottlenecks that were not otherwise obvious.



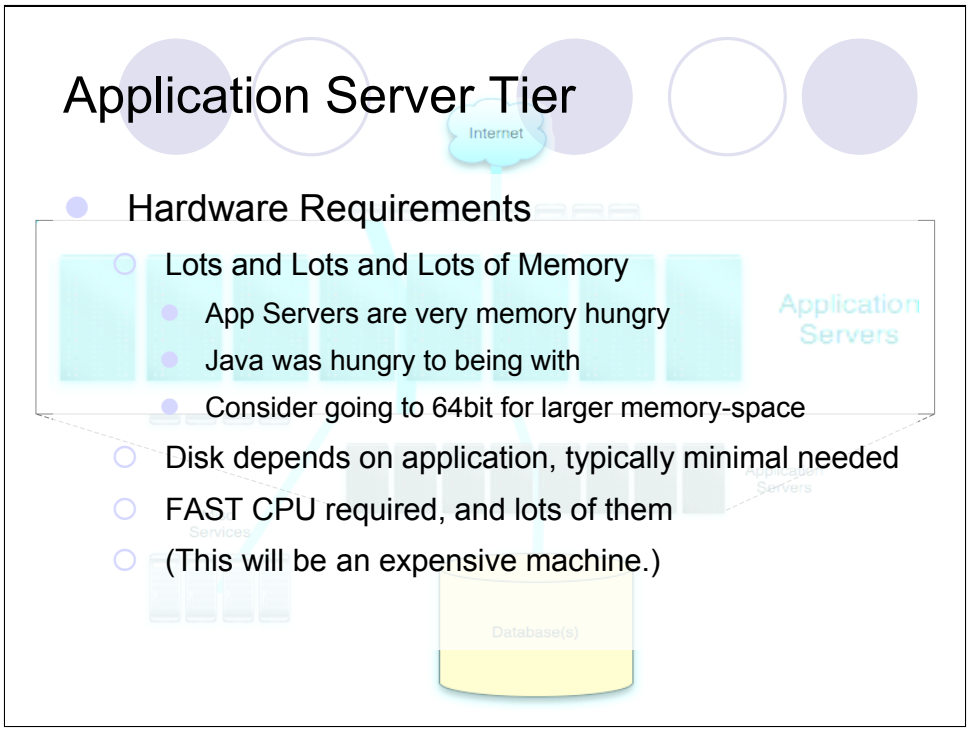
REST is an overused term, in my humble opinion. Most of the time when people use the term REST, all they're really saying is "remote procedure call".



The modern Web Application is a complex distributed system. Distributed systems need communication infrastructure so that the components can relay information to each other. Designing the system in this way provides some benefits and some drawbacks. The choice of communication medium comes down to performance and usability. It should be easy to use while still maintaining a sufficient level of performance to satisfy the needs of the system. Most of the technologies mentioned above will perform well enough under low-load conditions, but many of them fall down under high-load scenarios.



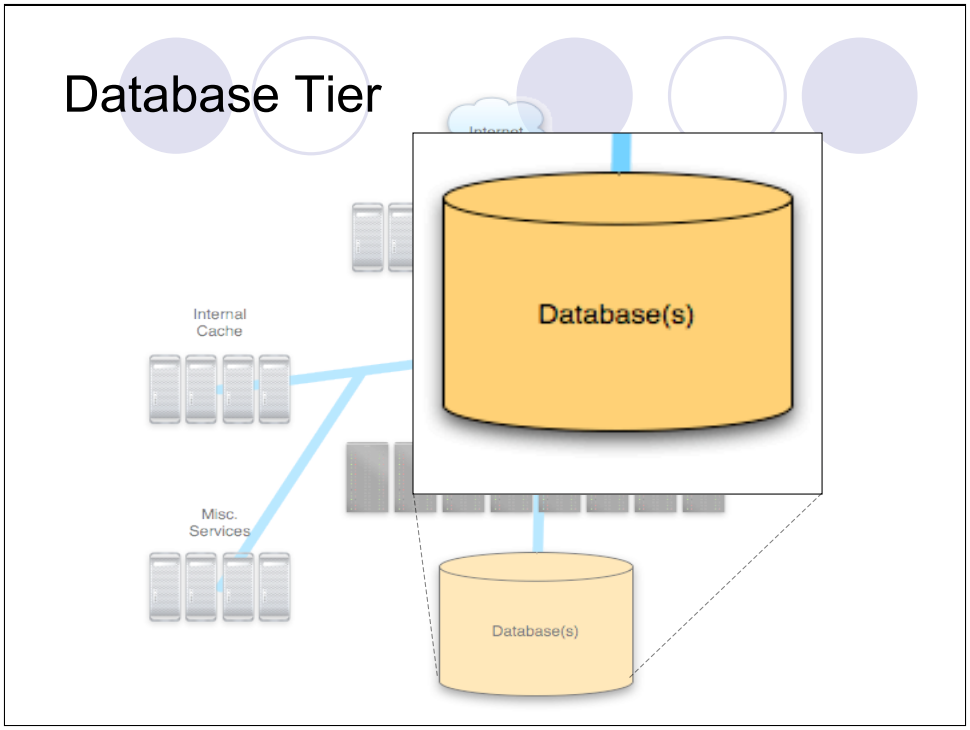
It's important to identify failure scenarios before writing any code, to ensure that the system will behave as expected under all conditions. This is much more important within a distributed system, since the failure scenarios are much more complex and difficult to deal with.



Application Servers are typically the most resource hungry machines in a Web Architected System, aside from maybe a central database.

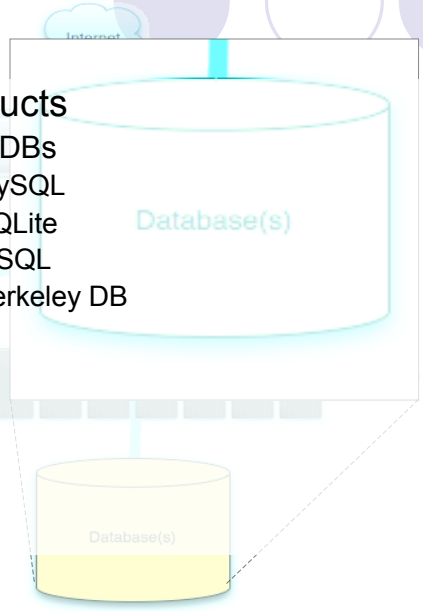
Considering the availability of 64-bit systems with large memory capacities, and the availability of 64-bit applications such as Java and Apache, writing large-scale memory-intensive applications has become much more possible.

Database Tier



Database Tier

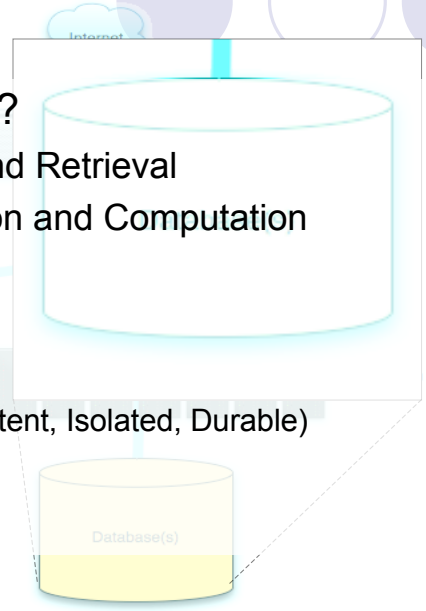
- Available DB Products
  - Free/Open Source DBs
    - PostgreSQL
    - GNU DBM
    - Ingres
    - SQLite
    - MySQL
    - SQLite
    - mSQL
    - Berkeley DB
  - Commercial
    - Oracle
    - MS SQL
    - IBM DB2
    - Sybase
    - SleepyCat



There are a huge number of free and commercial database packages available, each with their own benefits and drawbacks.

Database Tier

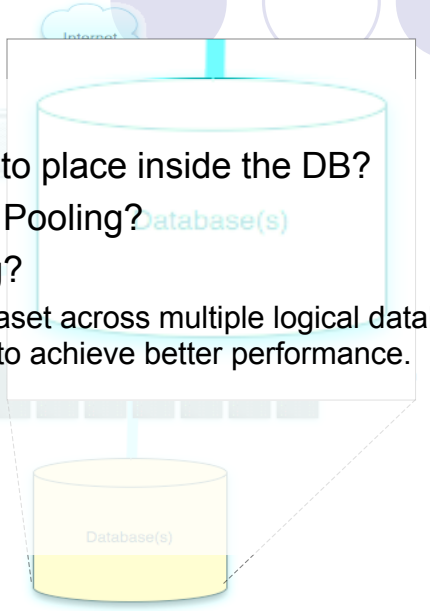
- What does it do?
  - Data Storage and Retrieval
  - Data Aggregation and Computation
  - Sorting
  - Filtering
  - ACID properties
    - (Atomic, Consistent, Isolated, Durable)



The database is one of the central points of contention within the Web Architecture. This means that huge pressure is being placed on the database from simultaneous interests throughout the system. Reliability and Performance are both crucial. Data warehousing and availability are also important.

Database Tier

- Choices
  - How much logic to place inside the DB?
  - Use Connection Pooling?
  - Data Partitioning?
    - Spreading a dataset across multiple logical database "slices" in order to achieve better performance.

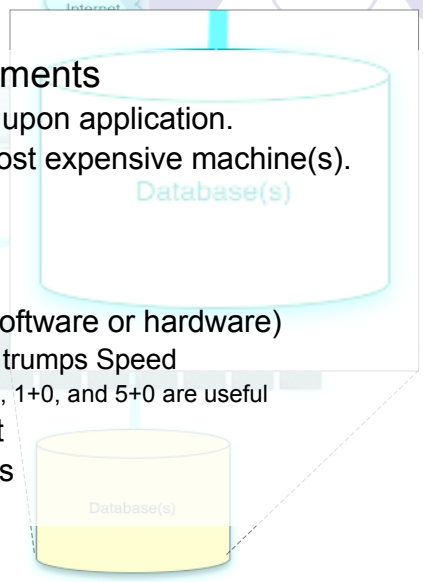


Deciding how much business logic to put into the database vs. in application space is a whole can of worms that is outside of the scope of this presentation. Some systems such as MySQL place huge burdens on the application developer to write their own data integrity constraints, while other systems such as Oracle encourage putting everything inside the database. There is a happy medium.

Don't forego data integrity for the sake of performance, it will bite you later.

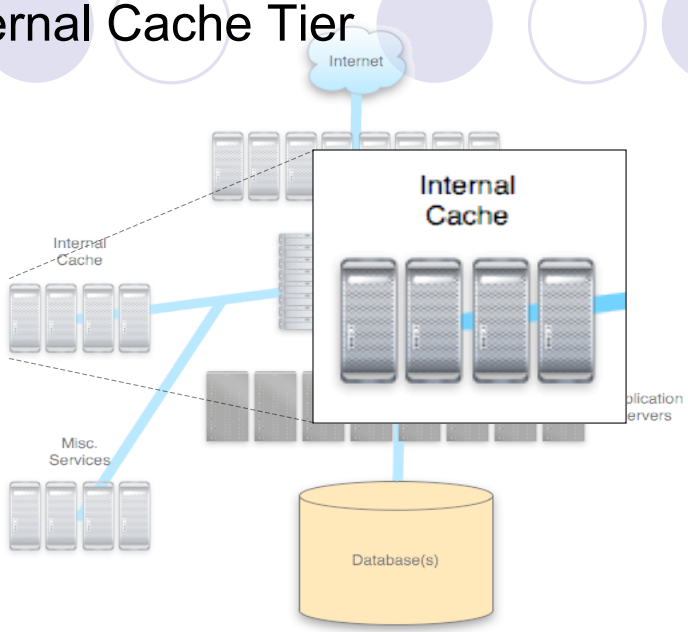
Database Tier

- Hardware Requirements
  - Entirely dependent upon application.
  - Likely to be your most expensive machine(s).
  - Tons of Memory
  - Spindles galore
  - RAID is useful (in software or hardware)
    - Reliability usually trumps Speed
      - RAID levels 0, 5, 1+0, and 5+0 are useful
  - CPU also important
  - Dual power supplies
  - Dual Network

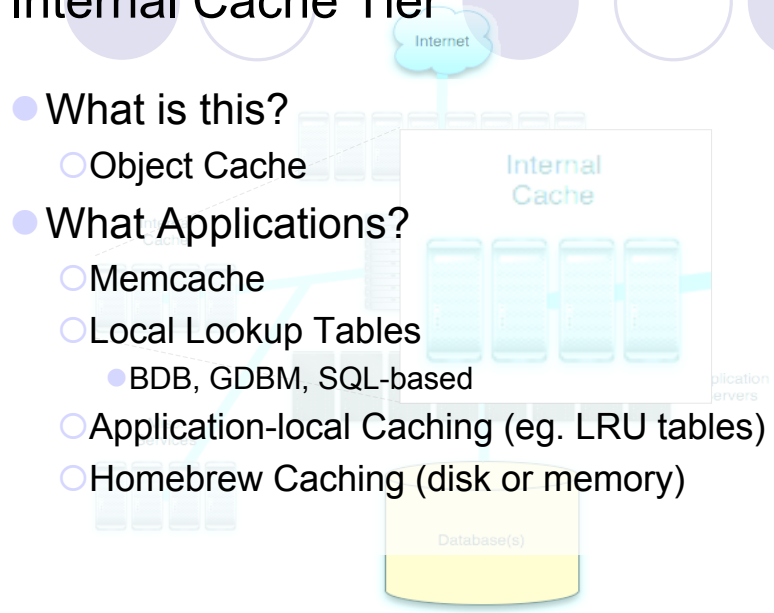


The size, speed, and number of disks, CPU, memory, etc… that is required by your database is entirely dependent upon the problem you are trying to solve. In general, spindles are important for databases to be able to perform concurrent operations. Memory is also important for the database to be able to perform sorts and filtering of data sets.

Internal Cache Tier



Internal Cache Tier

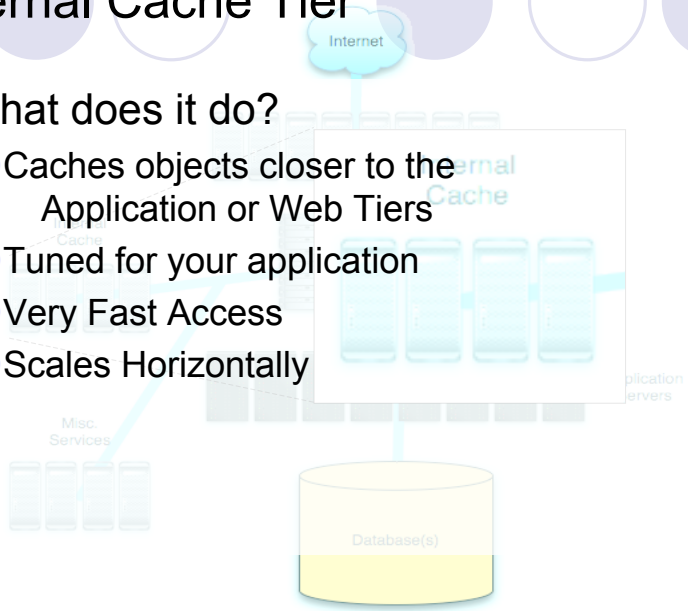


- What is this?
  - Object Cache
- What Applications?
  - Memcache
  - Local Lookup Tables
    - BDB, GDBM, SQL-based
  - Application-local Caching (eg. LRU tables)
  - Homebrew Caching (disk or memory)

Memcache is becoming a very popular form of Internal Cache. It has its own load-balancing algorithm, and is essentially a distributed network-based key-value LRU cache system. It's very fast. Use it to store short-lived objects from your web or application tiers. You could write your own on top of a flat file, a DB or an SQL engine, but only do that if you need to.

Internal Cache Tier

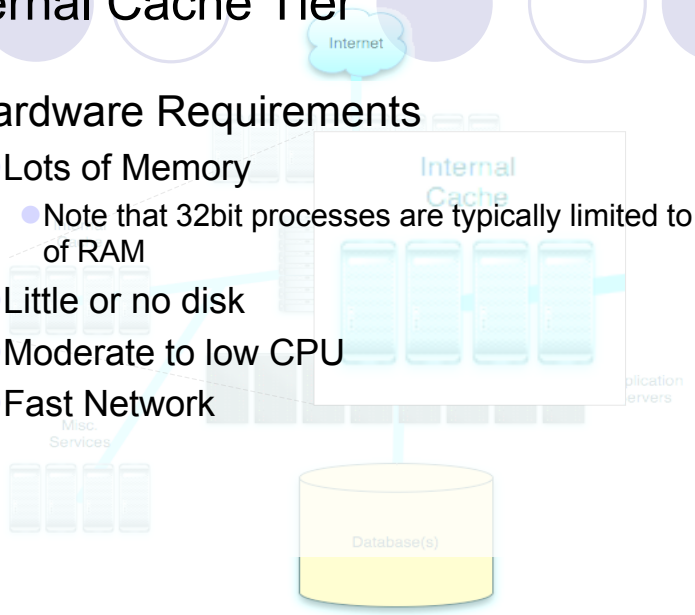
- What does it do?
  - Caches objects closer to the Application or Web Tiers
  - Tuned for your application
  - Very Fast Access
  - Scales Horizontally



Add more memcache boxes as needed.

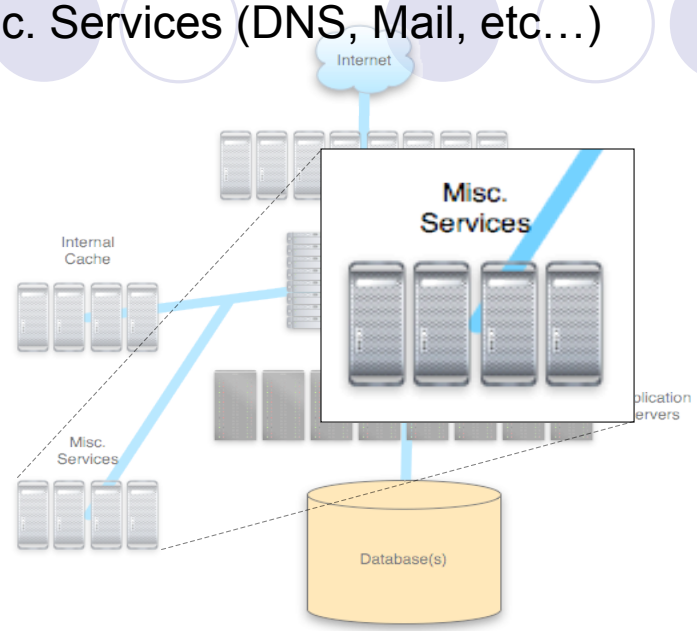
Internal Cache Tier

- Hardware Requirements
  - Lots of Memory
    - Note that 32bit processes are typically limited to 2GB of RAM
  - Little or no disk
  - Moderate to low CPU
  - Fast Network

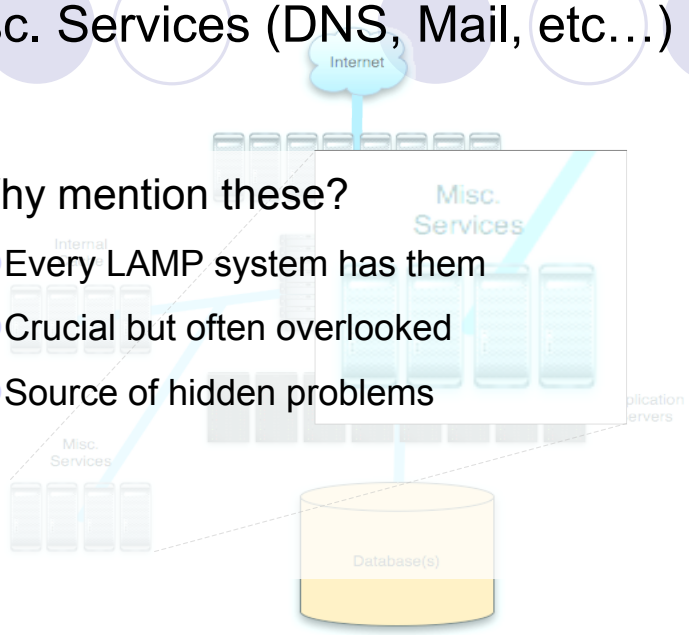


The main cost of these boxes will be the memory.

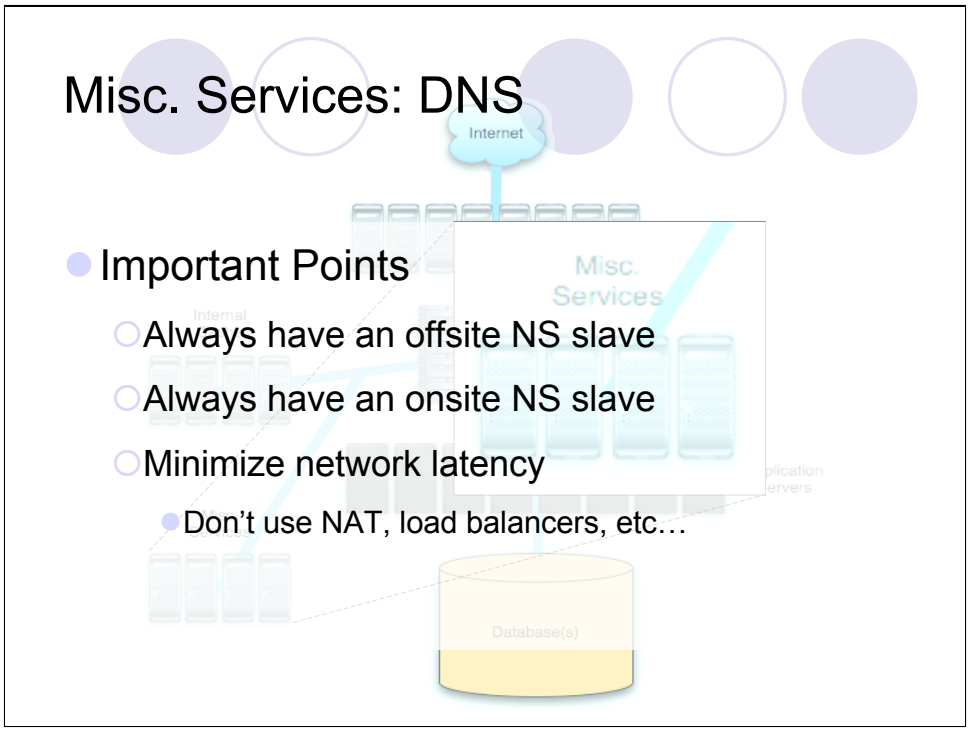
Misc. Services (DNS, Mail, etc…)



Misc. Services (DNS, Mail, etc…)

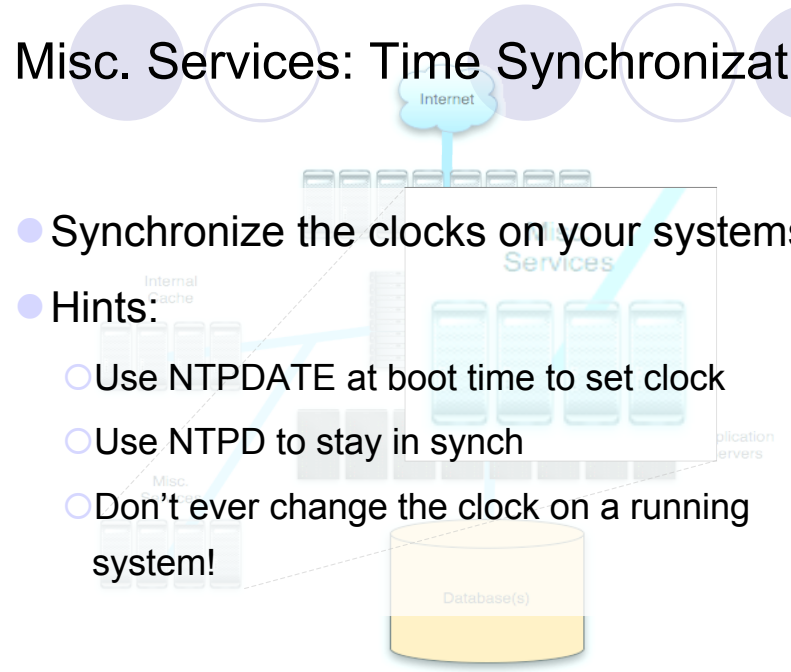


- Why mention these?
  - Every LAMP system has them
  - Crucial but often overlooked
  - Source of hidden problems



Managing DNS is an art form. DNS is one of the largest distributed systems ever made. It is also probably one of the most misunderstood of the internet technologies. Make sure you have a backup offsite, and a primary server onsite for your servers to use. Use redundant switches, routers, network cards, etc. This thing should always stay up.

Misc. Services: Time Synchronization

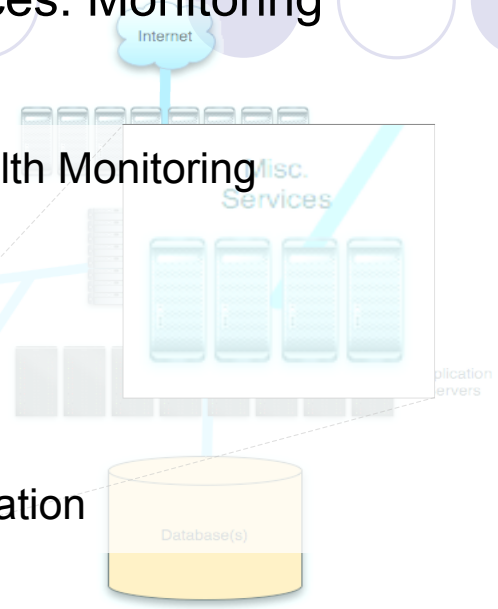


- Synchronize the clocks on your systems!
- Hints:
  - Use NTPDATE at boot time to set clock
  - Use NTPD to stay in synch
  - Don't ever change the clock on a running system!

It's important to keep your systems accurately synchronized. NTPD is free and very useful.

NEVER EVER change the clock on a running system. You can cause harm to your programs in ways that are often very difficult to diagnose.

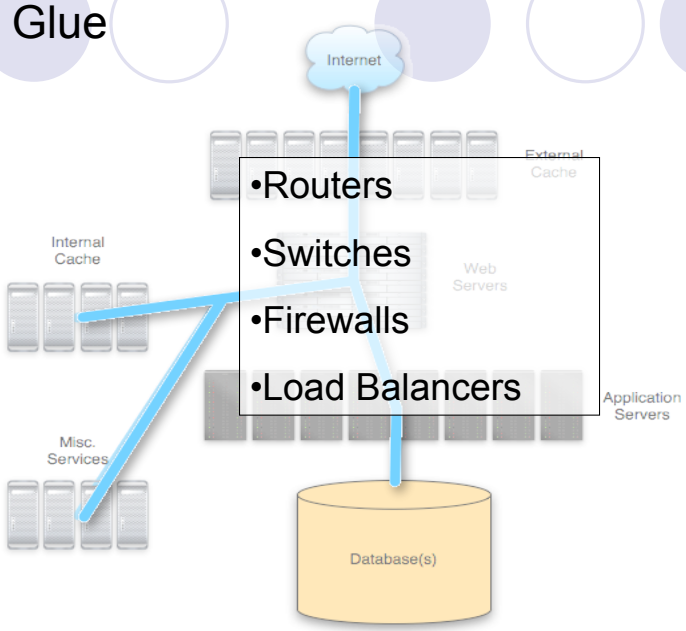
Misc. Services: Monitoring



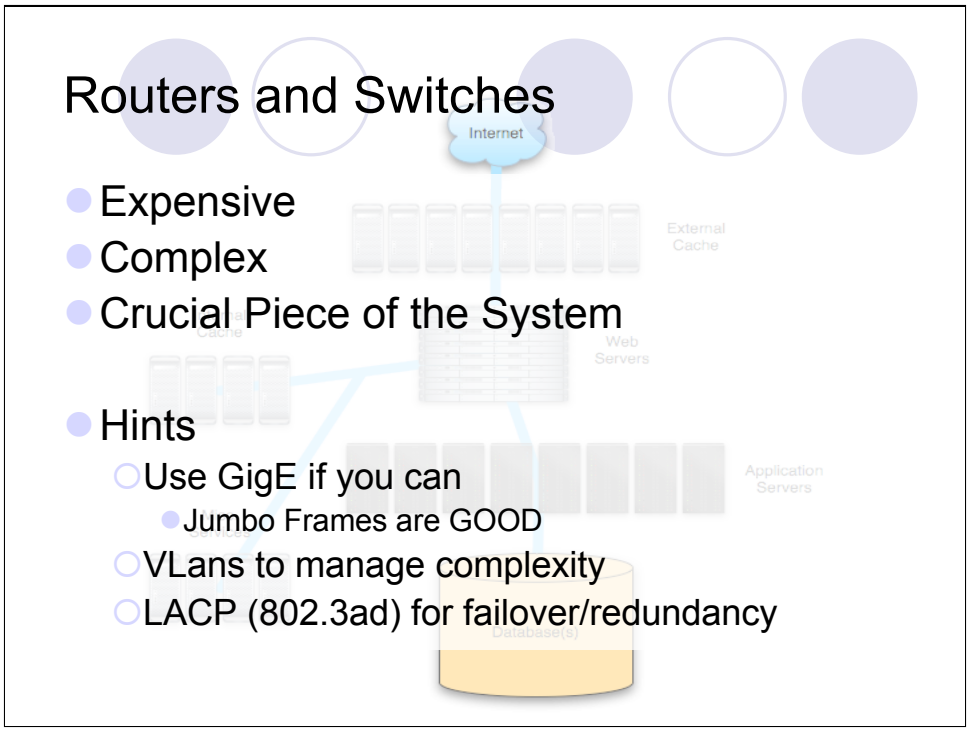
- System Health Monitoring
  - Nagios
  - Big Brother
  - Orcalator
  - Ganglia
- Fault Notification

There are numerous monitoring and notification systems available. I didn't even mention any commercial alternatives, of which there are numerous. The main thing here is that the notification system should not be a replacement for good troubleshooting. It should merely be a tool for monitoring the health of the system.

The Glue



- Routers
- Switches
- Firewalls
- Load Balancers



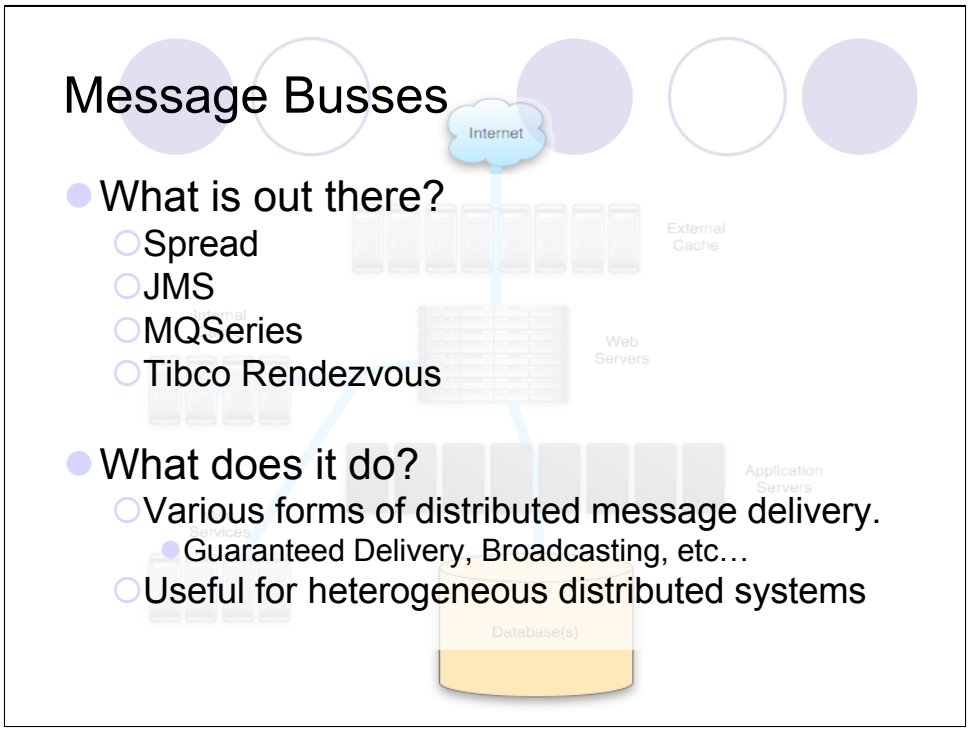
Gigibit Ethernet is best utilitized with Jumbo Frames. Jumbo Frames are a modification to the ethernet stack that increases the maximum packet size from 1500 bytes to 9000 bytes. This reduces OS overhead for each packet being sent and received, and improves throughput. Make sure your OS, your network card and your switch all support Jumbo Frames. You may have to enable Jumbo Frame support in your switch before it will work.

LACP stands for Link Aggregation Control Protocol, and is also know as Trunking, Bonding and Aggregation, and is described in the IEEE 802.3ad specification. It can be used for bonding two physical layer ethernet connections into a single logical connection. It can also be used to connect a machine with two or more ethernet ports to two or more switches for instant failover. I suggest getting GigE networks and using LACP in the failover mode. This will protect against switch, cable or network controller failure.

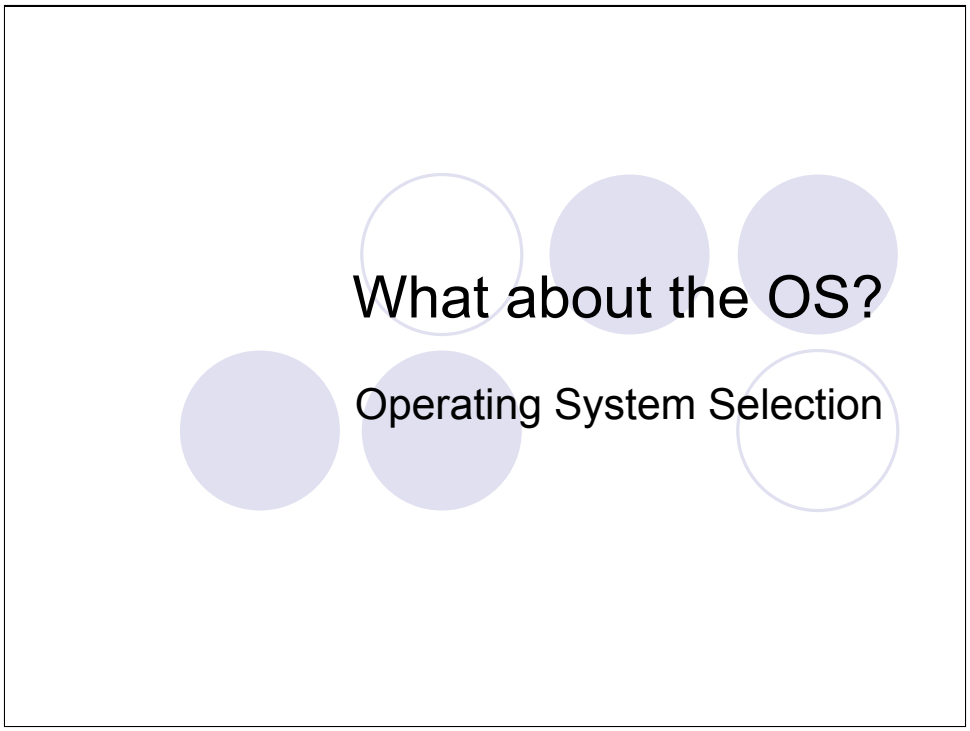
Load Balancers

- What services to balance?
  - HTTP Caches and Servers, App Servers, DB Slaves
- What NOT to balance?
  - DNS
  - LDAP
  - NIS
  - Memcache
  - Spread
  - Anything with it's own built-in balancing

Load Balancing inevitably introduces a small amount of network latency in exchange for greatly improving network application throughput and concurrency. Applications that have their own built-in load balancing or distribution shouldn't use a load balancer.



Message Busses are another form of communication glue that exists between components in a distributed system. Message Busses like Spread and JMS are extremely useful for information flow and control. One excellent use of a message bus is in prototyping new data processing techniques without disturbing the production systems. Most message delivery systems optionally provide delivery guarantees, which are important for transaction-oriented systems or systems with data integrity requirements (hint: every system requires some level of data integrity, so you should all be taking advantage of delivery guarantees).


What about the OS?
Operating System Selection

Lots of OS choices

- Linux
- FreeBSD
- NetBSD
- OpenBSD
- OpenSolaris
- Commercial Unix

Plenty of Operating Systems to chose from. The free/open source systems are significantly more mature than they were a number of years ago.

What's Important?

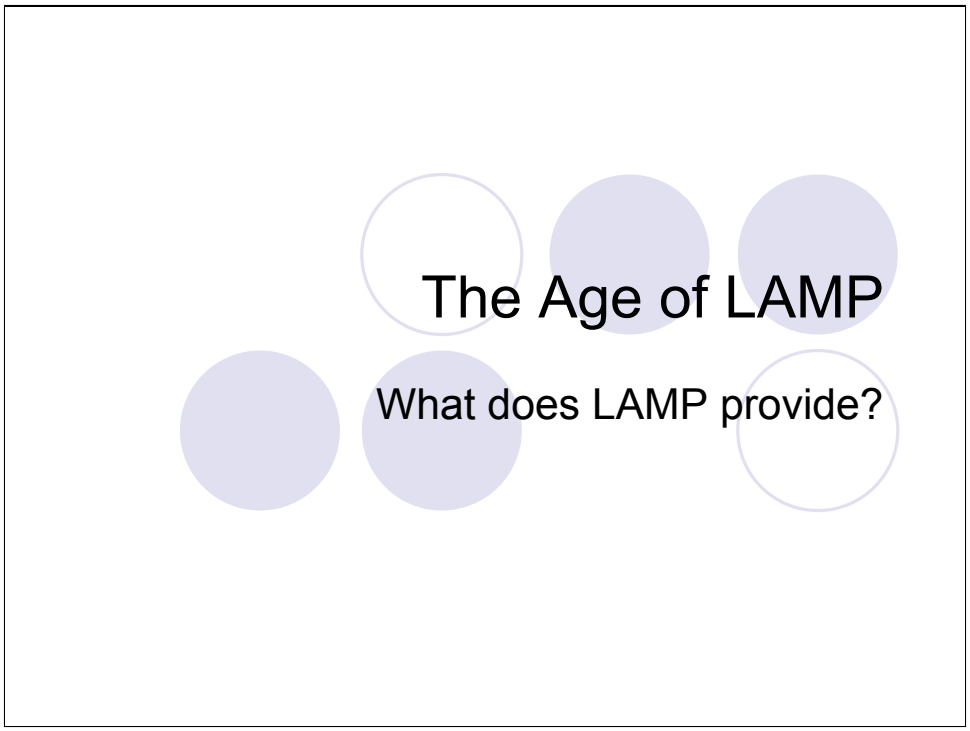
- Maintainability
  - Upgrade Path
  - Security Updates
  - Bug Fixes
- Usability
  - Do your engineers like it?
- Cost
  - Hardware Requirements
  - (you don't need a commercial Unix anymore)

When choosing an operating system, there are many things to consider. Since your distributed web application will likely consist of many different types of software applications running on top of many different types of hardware, it's important to minimize the operational overhead of running all these machines. Standardizing on a single login technology (NIS, LDAP, RADIUS, Kerberos, etc…) may be one level of acceptable consistency. Alternatively (although not recommended) you may wish to restrict all systems to the same operating system and version.

Features to look for

- Multi-processor Support
- 64bit Capable
- Mature Thread Support
- Vibrant User Community
- Support for your devices

As operating systems mature, support for things like thread libraries and 64-bit processors becomes more readily available. Almost as important as the technology is the existence of a vibrant user community. A vibrant user community will provide bug fixes, improve stability, react rapidly to security vulnerabilities, and among other things will port software packages and document useful features. All of these things will combine to improve the system and reduce usage costs.


The Age of LAMP
What does LAMP provide?

Scalability

- Grows in small steps
- Stays up when it counts
- Can grow with your traffic
- Room for the future

The scalability of large systems is a difficult aspect to quantify. In an ideal world, every component of a complex system would be tested to measure its capacities. In the real world, this is a little more vague. It's still important to measure your system under load and under artificial conditions designed to mimic real-world usage. In the end, what this really means is confidence that your system will grow as your business grows.

Reliability

- High Quality of Service
- Minimal Downtime
- Stability
- Redundancy
- Resilience

Reliability is very easy to measure. The common metric in our industry is “uptime”, or the percentage of downtime over a period of time (typically 12 months).

Not everyone needs five-nines, but the amount of resources you dedicate to keeping your system up and running will directly affect the experience of your users.

Low Cost

- Little or no software licensing costs
- Minimal hardware requirements
- Abundance of talent
- Reduced maintenance costs

One of the fundamental aspects of LAMP is that it's built from Open Source Components. Open source components are typically provided for free (as in beer), even for commercial uses.

Another benefit of using open source components is the availability of talented engineers experienced in the various technologies. Since the tools themselves are open and available, the learning curve is often greatly improved compared to closed source alternatives.

Using commodity parts such as open source allows builders to rely on the abundance of documentation and shared knowledge, as well as the shared bug reports of the open source community. All of these theoretically combine to allow greatly reduced maintenance costs.

Flexible

- Modular Components
- Public APIs
- Open Architecture
  - Vendor Neutral
  - Many options at all levels

Using components that share open and well publicized APIs also improves the flexibility of the system. The parts of the system are modular and interchangeable, allowing for many competitive choices (most of which are free).

Extendable

- Free/Open Source Licensing
  - Right to Use
  - Right to Inspect
  - Right to Improve
- Plugins
  - Some Free
  - Some Commercial
  - Can always customize

Having the source to a commodity piece of software has been a boon for system developers. Bugs can be found and squashed. Improvements made and often contributed back to the system.

Note that not all open source licenses allow for use, inspection, and improvements, so pay attention to those licenses.

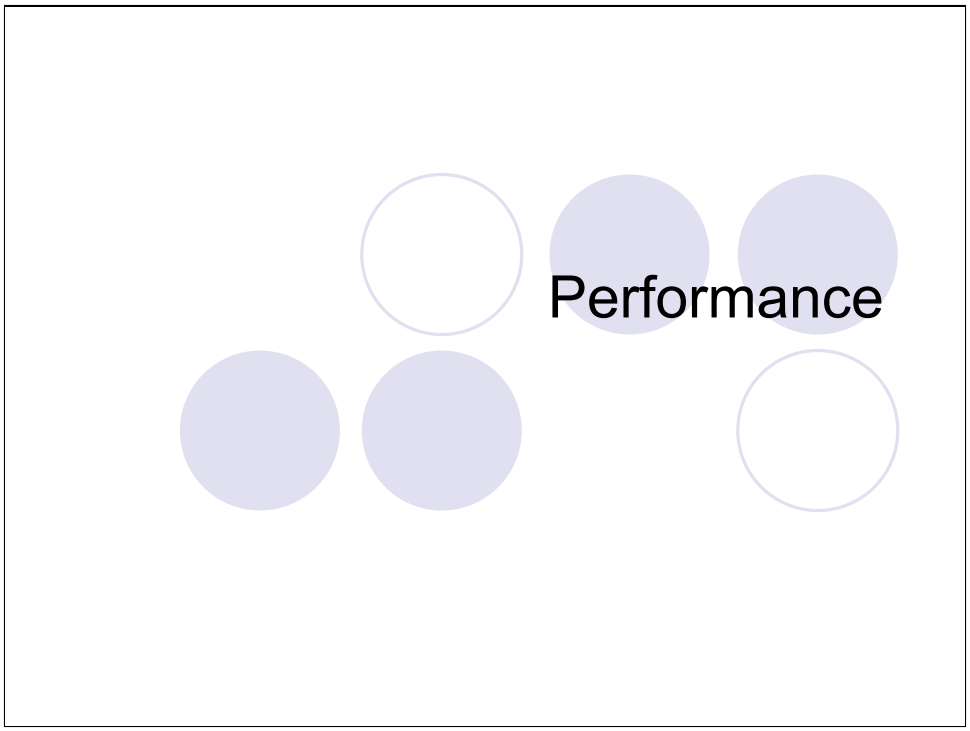
Free as in Beer?

Price
Speed
Quality

Pick any two.

Alas, everything has a price.

There are tradeoffs in everything, including software. LAMP, unfortunately, doesn't solve this. But it does make some outstanding improvements in each area of Price, Speed and Quality.


Performance

What is Performance?

- For LAMP?
  - Improving the **User Experience**

Architecture affects user experience?

- It affects it in two ways
  - Speed → Fast Page Loads (*Latency*)
  - Availability → Uptime

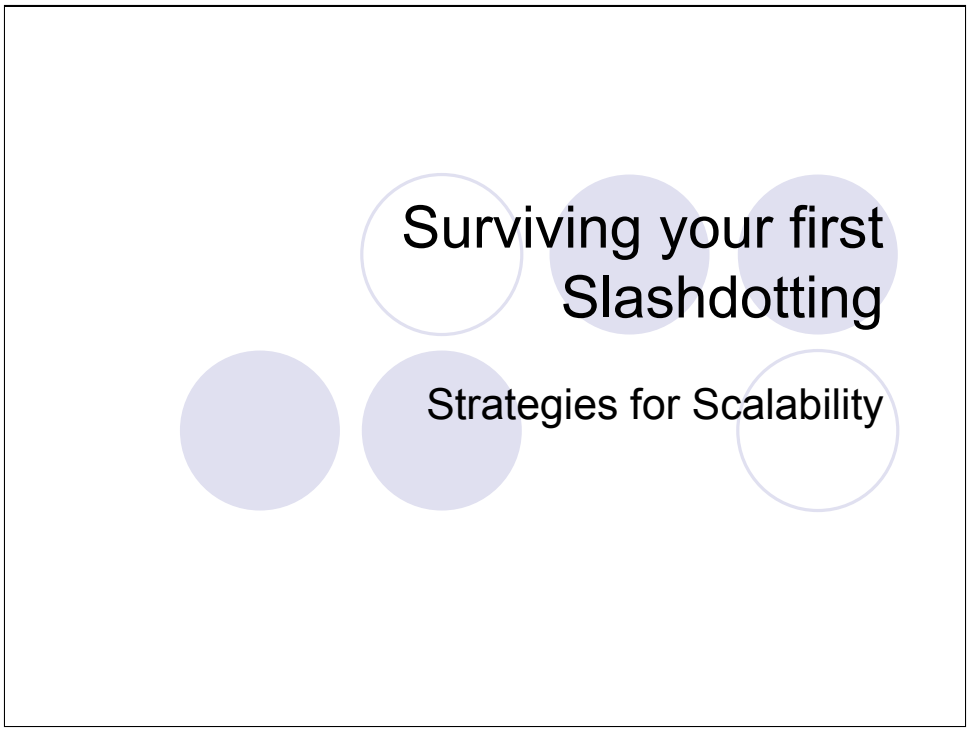
Problem: Concurrency

- Concurrency causes slowdowns
- Latency suffers
- Pages load more slowly

Solution: Design for Concurrency

- Build parallel systems
- Eliminate bottlenecks
- Aim for horizontal scalability

Now for some real-world examples…


Surviving your first Slashdotting
Strategies for Scalability

What is a “Slashdotting”?

- Massive traffic spike (1000x normal)
- High bandwidth needed
- VERY high concurrency needed
- Site inaccessible to *some* users
- If your system **crashes**, nobody gets in

Approach

1. Keep the system up, no crashing
   - Some users are better than none
2. Let as many users in as possible

Strategies

1. Load Balancers (of course)
2. Static File Servers
3. External Caching

Load Balancers

- Hardware vs. Software
  - Software is complex to set up, but cheaper
  - Hardware is expensive, but dedicated
  - IMHO: Use SW at first, graduate to HW

Software Load Balancers such as LVS (Linux Virtual Server) and others allow failover load balancer support. There are also numerous commercial hardware load balancer vendors out there that provide the same product. I suggest starting with a software-based system, and then transitioning to a hardware-based solution as traffic increases.

Static File Servers: Zero-copy

- Separate Static from Dynamic
  - Scale them independently
- Later, dedicate static content serers
  - Modern web servers are ***very*** good at serving static content such as
    - HTML
    - CSS
    - Images
    - Zip/GZ/Tar files

Starting with Apache 2.0, support for zero-copy was added (using something called sendfile()). Zero-copy is a technique where content can be delivered directly from the disk (or the buffer cache) to the network card, with only minimal coordination by the CPU. This gives a dramatic improvement in speed, and a dramatic reduction in latency and system overhead for serving static content. For an even bigger boost, try the latest Apache 2.x with the Worker MPM.

External Caching

- Reduces internal load
- Scales horizontally
- Obeys HTTP-defined rules for caching
  - Your app defines the caching headers
  - Behaves no differently than other proxy servers or your browser, only it's dedicated
- Hint: Use mod_expires to override

Outgrowing your First Server

Strategies for Growth

Design for Horizontal Scalability

- Manage Complexity
- Design Stateless Systems (hard)
- Identify Bottlenecks (hard)
- Predict Growth
- Commodity Parts

Predict your traffic needs and plan your capacity accordingly.

Manage Complexity

- Decouple internal services
  - Services scale independently
  - Independent maintenance
- Well-defined APIs
  - Facilitates service decoupling
  - Scales your engineering efforts

One bottleneck you will likely run in to is how to scale your engineering efforts. Whether you are working alone or in a large team, separating your subsystems into independent services (with their own web/app servers) and connected via well-defined APIs will help you manage complexity. Simplifying the pieces in your system will help you identify bottlenecks and allow you to better concentrate on algorithmic and architectural changes that will help overall scalability.

What is a Stateless System?

- Each connection hits a new server
- Server remembers nothing

- Benefits?
  - Allows Better Caching
  - Scales Horizontally

A stateless session is an HTTP request/response cycle that is fully self-contained and relies on no other information about the client other than what is contained in the request. Designing systems in this way allow for horizontal scalability because the request can be dispatched to any of a number of identical web servers (or application servers) for processing, and the outcome will always be the same. This means that capacity can be added simply by adding more servers.

Designing Stateless Systems

- Decouple session state from web server
- Store session state in a DB
  - Careful: may just move bottleneck to DB tier
- Use a distributed internal cache
  - Memcached
  - Reduces pressure on session database

Example: Scaling your User DB

- Assume you have a user-centric system
  - Eg. User identity info, subscriptions, etc…

1. Group data by user
2. Distribute users across multiple DBs
3. Write a directory to track user->DB location
4. Cache user data to reduce DB round trips

Disadvantage: difficult to compare two users

Identify Bottlenecks

- Monitor your system performance
  - Use tools for this, there are many
- Store and plot historical data
  - Used to identify abnormalities
    - Check out ***rrdtool***
- Use system tools to troubleshoot
  - ***vmstat, iostat, sar, top, strace,*** etc…

Predict Growth

- Use performance metrics
  - Hits/sec
  - Concurrent connections
  - System load
  - Total number of users
  - Database table rows
  - Database index size (on disk/memory)
  - …

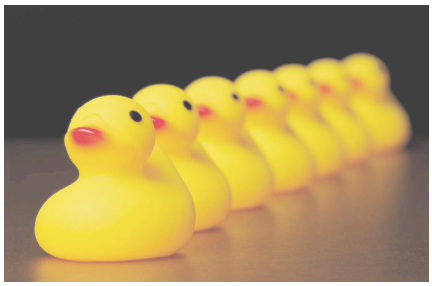
Machine-sized Solutions

- Design for last year's hardware
  - (it's cheaper)
- Leaves room for your software to grow
  - Hardware will get faster
  - And your systems will get busier

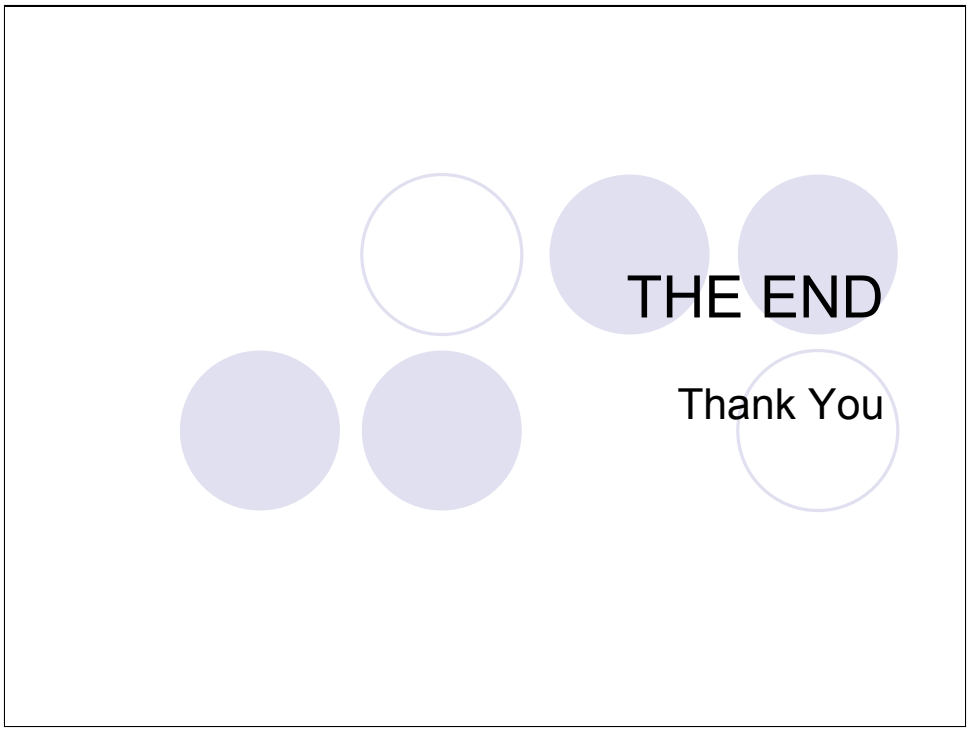
Use Commodity Parts

- Standardize Hardware
- Use Commodity Software

  (Open Source!)
- Avoid Fads



Avoid the latest and greatest technologies and focus on what works: standards, commoditized hardware, commoditized software.


THE END
Thank You