Large-Scale Web Applications

Mendel Rosenblum
Web Application Architecture

Web Browser

Web Server / Application server

Storage System

Chrome
Mozilla Firefox
Safari
Internet

HTTP

LAN

MySQL
mongoDB

CS142 Lecture Notes - Intro
Large-Scale: **Scale-Out Architecture**
Scale-out architecture

- Expand capacity by adding more instances
- **Contrast:** *Scale-up architecture* - Switch to a bigger instance
  - Quickly hit limits on how big of single instances you can build
- **Benefits of scale-out**
  - Can scale to fit needs: Just add or remove instances
  - Natural redundancy make tolerating failures easier: One instance dies others keep working
- **Challenge:** Need to manage multiple instances and distribute work to them
Scale out web servers: Which server do you use?

- Browsers want to speak HTTP to a web server
- Use load balancing to distribute incoming HTTP requests across many front-end web servers
- HTTP redirection (HotMail, now LiveMail):
  - Front-end machine accepts initial connections
  - Redirects them among an array of back-end machines
- DNS (Domain Name System) load balancing:
  - Specify multiple targets for a given name
  - Handles geographically distributed system
  - DNS servers rotate among those targets
Load-balancing switch ("Layer 4-7 Switch")

- Special load balancer network switch
  - Incoming packets pass through load balancer switch between Internet and web servers
  - Load balancer directs TCP connection request to one of the many web servers
  - Load balancer will send all packets for that connection to the same server.
- In some cases the switches are smart enough to inspect session cookies, so that the same session always goes to the same server.
- Stateless servers make load balancing easier (different requests from the same user can be handled by different servers).
- Can select web server based on random or on load estimates
nginx ("Engine X")

- Super efficient web server (i.e. speaks HTTP)
  - Handles 10s of thousands of HTTP connections

- Uses:
  - Load balancing - Forward requests to collection of front-end web servers
  - Handles front-end web servers coming and going (dynamic pools of server)
    - Fault tolerant - web server dies the load balance just quits using it
  - Handles some simple request - static files, etc.
  - DOS mitigation - request rate limits

- Popular approach to shielding Node.js web servers
Scale-out assumption: any web server will do

- Stateless servers make load balancing easier
  - Different requests from the same user can be handled by different servers
  - Requires database to be shared across web servers

- What about session state?
  - Accessed on every request so needs to be fast (memcache?)

- WebSockets bind browsers and web server
  - Can not load balance each request
Scale-out storage system

- Traditionally Web applications have started off using relational databases
- A single database instance doesn't scale very far.

- **Data sharding** - Spread database over scale-out instances
  - Each piece is called data shard
  - Can tolerate failures by replication - place more than one copy of data (3 is common)

- Applications must partition data among multiple independent databases, which adds complexity.
  - Facebook initial model: One database instance per university
  - In 2009: Facebook had 4000 MySQL servers - Use hash function to select data shard
Memcache: main-memory caching system

- Key-value store (both keys and values are arbitrary blobs)
- Used to cache results of recent database queries
- Much faster than databases:
  - 500-microsecond access time, vs. 10's of milliseconds
- Example: Facebook had 2000 memcache servers by 2009
  - Writes must still go to the DBMS, so no performance improvement for them
  - Cache misses still hurt performance
  - Must manage consistency in software (e.g., flush relevant memcache data when database gets modified)
Scale-out web architecture

Internet → Load Balancer → Web Server → Database Server

Memcache

Web Server

Database Server

Database Server

Database Server

Database Server
Building this architecture is hard

- Large capital and time cost in buying and installing equipment
- Must become expert in datacenter management
- Figuring out the right number of different components hard
  - Depends on load demand
Scaling issues were hard for early web app

- Startup: Initially, can't afford expensive systems for managing large scale.

- But, application can suddenly become very popular ("flash crowd"); can be disastrous if application can not scale quickly.

- Many of the early web apps either lived or died by the ability to scale
  - Friendster vs. Facebook
Virtualization - Virtual and Physical machines

Virtual Machines Images

<table>
<thead>
<tr>
<th>Disk Images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Balancer</td>
</tr>
<tr>
<td>Web Server</td>
</tr>
<tr>
<td>Database Server</td>
</tr>
<tr>
<td>Memcache</td>
</tr>
</tbody>
</table>

Load Balancer 1
Web Server 100
Database 50
Memcache 20

Virtualization layer

Physical Machines

<table>
<thead>
<tr>
<th>server</th>
<th>server</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
</tbody>
</table>
Cloud Computing

- Idea: Use servers housed and managed by someone else
  - Use Internet to access them

- Virtualization is a key enabler

  Specify your compute, storage, communication needs: Cloud provider does the rest

- Examples:
  - Amazon EC2
  - Microsoft Azure
  - Google Cloud
  - Many others

<table>
<thead>
<tr>
<th>Service</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load balancer</td>
<td>1</td>
</tr>
<tr>
<td>Web Server</td>
<td>100</td>
</tr>
<tr>
<td>Database</td>
<td>50</td>
</tr>
<tr>
<td>Memcache</td>
<td>20</td>
</tr>
</tbody>
</table>
Cloud Computing Advantages

● Key: Pay for the resources you use
  ○ No up front capital cost
  ○ Need 1000s machines right now? Possible
  ○ Perfect fit for startups:
    ■ 1998 software startup: First purchase: server machines
    ■ 2012 software startup: No server machines

● Typically billing is on resources:
  ○ CPU core time, memory bytes, storage bytes, network bytes

● Runs extremely efficiently
  ○ Buy equipment in large quantities, get volume discounts
  ○ Hirer a few experts to manage large numbers of machines
  ○ Place servers where space, electricity, and labor is cheap
Higher level interfaces to web app cloud services

- Managing a web app backend at the level of virtual machines requires system building skills.
- If you don't need the full generality of virtual machines you can use some already scalable platform.

Example: Google App Engine
Google App Engine

- You provide pieces of Python or Java code, URLs associated with each piece of code.

- Google does the rest:
  - Allocate machines to run your code
  - Arrange for name mappings so that HTTP requests find their way to your code
  - Scale machine allocations up and down automatically as load changes
  - AppEngine also includes a scalable storage system

- More constrained environment
  - Must use Python, Java, PHP, or Go
  - Must use specialized Google storage system

- Can work: Snapchat
Cloud Computing and Web Apps

- The pay-for-resources-used model works well for many web app companies
  - At some point if you use many resources it makes sense to build own data centers

- Many useful services available:
  - Auto scaling (spinning up and down instances on load changes)
  - Geographic distribution (can have parts of the backend in different parts of the world)
  - Monitoring and reporting (what parts of web app is being used, etc.)
  - Fault handling (monitoring and mapping out failed servers)
Content Distribution Network (CDN)

- Consider a read-only part of our web app (e.g. image, html template, etc.)
  - Browser needs to fetch but doesn't care where it comes from

- Content distribution network
  - Has many servers positions all over the world
  - You give them some content (e.g. image) and they give you an URL
  - You put that URL in your app (e.g. `<img src="...")
  - When user's browsers access that URL they are sent to the closest server (DNS trick)

- Benefits:
  - Faster serving of app contents
  - Reduce load on web app backend

- Only works on content that doesn't need to change often