CS-310 Scalable Software Architectures

Lecture 1:
Types of Scaling

Steve Tarzia

Gaps in traditional CS curriculum

- In the first few CS classes, students learn all about writing **programs**.
  - These are **single-machine**, and **single-threaded**.
  - Take an input and produce an output.
  - Goals are: *correctness*, *efficiency*, (and hopefully *clarity* or readability).
- After that, most of the upper-level classes are introductions to various computing research fields.
  - These are conceptually difficult, but involve only very small programs.
  - This is preparation for a PhD program, not for Software Engineering.
- **The Result**: most CS graduates are not ready to be productive in even a junior-level software engineering job.

What you'll learn in this class

- In short, you'll learn to build **real, complex, big** software **services**.
  - Eg., how to build something like Google Search or Netflix.
  - Writing correct & efficient code is only a small part of the challenge.
- Learn about:
  - Coordinating multiple apps
  - Scaling load
  - Big data storage and processing
  - Operating in the cloud, and different computing platform models
  - … and more.
- **The Goal**: to learn enough to build your own scalable startup product. Bypass the "on the job training" or self-study usually required.

Topics we'll cover

Control:

- Separation of concerns: Microservices, APIs
- Asynchronicity: Distributed Message Queues, Push Notifications
- Parallel processing: Load balancing, Map Reduce, Spark
- Platforms: Cloud computing, VMs, Containers, Serverless functions

Data Storage:

- Relational vs NoSQL databases
- Caching, Content Delivery Networks (CDNs)

Case Studies:

- Wikipedia, Netflix, Twitter, etc.
- We'll look at a small piece of each of these companies' architecture.

What you will **not** learn

- Machine learning
- Database internals
- Cloud infrastructure internals (virtualization, SDN)
- Distributed systems details

What you'll learn in this class

- In short, you'll learn to build **real, complex, big** software **services**.
  - Eg., how to build something like Google Search or Netflix.
  - Writing correct & efficient code is only a small part of the challenge.
- Learn about:
  - Coordinating multiple apps
  - Scaling load
  - Big data storage and processing
  - Operating in the cloud, and different computing platform models
  - … and more.
- **The Goal**: to learn enough to build your own scalable startup product. Bypass the "on the job training" or self-study usually required.

What you will **not** learn

- Machine learning
- Database internals
- Cloud infrastructure internals (virtualization, SDN)
- Distributed systems details

Topics we'll cover

- Service Oriented Architectures (Microservices)
- Relational vs NoSQL databases
- Caching, Content Delivery Networks (CDNs)
- Distributed Message Queues
- Parallel processing with: Map Reduce, Spark

Case Studies:

- Wikipedia, Netflix, Twitter, etc.
- We'll look at a small piece of each of these companies' architecture.

Today's introduction

- Differences between standalone apps and **services**.
- What do we mean by “scalability” and why is it difficult?

Traditional view of Software **Scalability**

In Data Structures & Algorithms we consider a kind of scalability:

- As **input data size** "n" gets bigger, program should run quickly.
- Complexity analysis lists program runtime as a function of input size.

For example:

- Given a list of size ***n***, mergesort takes O(n log n) time to run.
- Given a hashtable of size ***n***, finding a value takes O(1) time.

- This assumes one problem to solve, one computer, and all operations having the same cost.

Services vs Programs

- A **service** is different than a simple program because it listens for requests from clients/users, and may handle multiple requests concurrently.
- External user provides an input (request) and service outputs a response.
- Requests are usually delivered as messages that arrive over a network.
- The service runs constantly, waiting for requests that it should process.
  - Thus, you can't just run the code on your laptop. You need a machine that is always powered-on (probably located in a data center or server room).

For example:

- a website, like: https://www.ebay.com/sch/i.html?_nkw=guitar

Defining *Service Scalability*

- Roughly speaking, a service is scalable if it can easily handle growth in the number of concurrent users/requests.

Scalability metrics are measures of **work throughput**:

- Requests/queries per second
- Concurrent users
- Monthly-active users

- So far, we don't care about the **costs** to achieve this scale (time per request or number of machines required), just the scale achieved.

Scaling Challenges

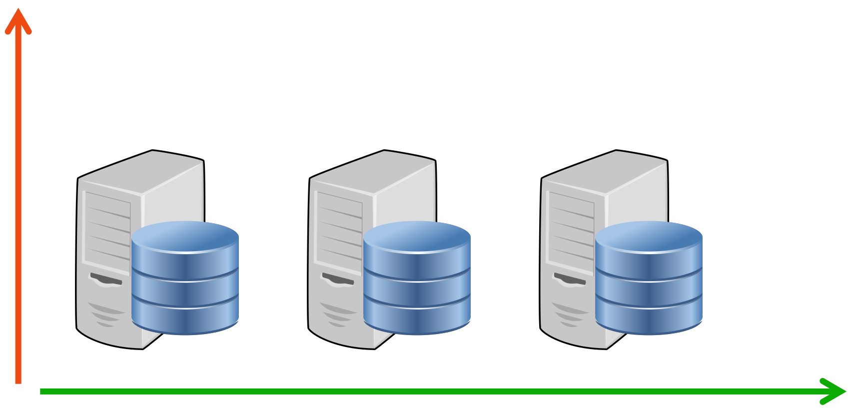
- Why is it difficult to make services big, even if *money ain't a thang?*

- Programs run on one machine, which has limited speed.
- **Coordinating** multiple machine can be difficult (who does what?)
- **Sharing data** among multiple machines is difficult (where is the data? how do we manage competing requests to change the same data?).
- More machines means there is high probability one will **fail** (die).
- Users can be distributed worldwide (communication **latency** is high).
- Service components must trust each other but ignore interference from attackers (**authentication**).
- Software **updates** must be deployed without downtime.

Vertical Scaling

- Let's assume that you're starting with a very light load and you can handle all the requests on one machine. Suddenly demand increases!
- The easiest approach to scaling is to just buy a faster machine to run your service.

Vertical scaling makes your machine(s) bigger and stronger.
Think "taller."



Horizontal scaling adds more machines.
Think of them standing side-by-side.

COMP_ENG 101: What affects computer performance?

Primarily:

- Number of CPU cores.
- Speed of each CPU core.
- (Lack of) competing processes running on the same machine.

Perhaps also:

- Amount of memory (RAM).
- Type of disk (SSD vs magnetic).
- Number of disks (parallel access).
- Type of network connectivity.
- Presence of GPUs, TPUs, and other special-purpose accelerators.

https://aws.amazon.com/ec2/instance-types/

Parallelism within a machine

- At any given time, you probably have about 100 processes (programs) "running" on your laptop (which probably has about 4 CPU cores).
- The OS kernel schedules processes, so they take turns using CPU.
- Often, processes **block** (wait) while doing input/output (IO).
  - For example, reading a file from disk, or waiting for a message to arrive from the network.
- While a process is blocked, another process can take over the CPU.
- A single process can have multiple **threads** which execute concurrently while sharing the same memory.
  - This is called Shared Memory Parallelism.

Apache web server example

```
top - 20:20:10 up 697 days, 17:04,  1 user,  load average: 0.00, 0.01, 0.00
Tasks:  91 total,   1 running,  90 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.3%us,  0.3%sy,  0.0%ni, 96.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.3%st
Mem:   1017368k total,   942832k used,    74536k free,     4988k buffers
Swap:  1048572k total,   308580k used,   739992k free,    40260k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7750 root      20   0  117m 6656 5556 S  0.0  0.7   0:00.00 sshd
 7727 root      20   0 79984 5800 4964 S  0.0  0.6   0:00.00 sshd
13729 tomcat    20   0 2666m 677m 4516 S  1.7 68.2 825:59.07 java
 7753 ec2-user  20   0  112m 3412 3008 S  0.0  0.3   0:00.00 bash
 7752 ec2-user  20   0  117m 3872 2772 S  0.0  0.4   0:00.00 sshd
 3879 root      20   0  526m  14m 2724 S  0.0  1.4   2383:27 cfn-hup
 7774 ec2-user  20   0 15308 2188 1904 R  0.0  0.2   0:00.18 top
17933 apache    20   0  823m 7888 1304 S  0.0  0.8  40:55.55 httpd.worker
17936 apache    20   0  823m 7496 1248 S  0.0  0.7  40:42.84 httpd.worker
17938 apache    20   0  823m 7712 1212 S  0.7  0.8  41:02.22 httpd.worker
 2546 root      20   0 89028 1224 1048 S  0.0  0.1  15:50.17 sendmail
17937 apache    20   0  823m 7616 1044 S  0.0  0.7  40:35.37 httpd.worker
17940 apache    20   0  823m 7168 1024 S  0.3  0.7  40:46.54 httpd.worker
17932 apache    20   0  823m 7764 1020 S  0.0  0.8  40:41.50 httpd.worker
17934 apache    20   0  823m 7444  896 S  0.3  0.7  41:05.20 httpd.worker
17939 apache    20   0  823m 7100  884 S  0.0  0.7  40:31.48 httpd.worker
17941 apache    20   0  823m 7308  836 S  0.0  0.7  40:32.70 httpd.worker
 3287 ntp       20   0 29288  944  792 S  0.0  0.1   0:56.44 ntpd
 2116 root      20   0  9356  872  728 S  0.0  0.1   0:16.25 dhclient
 2866 root      20   0  118m  824  728 S  0.0  0.1   4:59.63 crond
 2516 root      20   0 79984  700  592 S  0.0  0.1   3:43.41 sshd
 2555 smmsp     20   0 80492  640  500 S  0.0  0.1   0:05.54 sendmail
17935 apache    20   0  823m 6324  408 S  0.0  0.6  40:43.62 httpd.worker
17929 root      20   0 94836  588  192 S  0.0  0.1   1:24.51 httpd.worker
    1 root      20   0 19616  336  156 S  0.0  0.0   0:37.12 init
```

- At left is the output from the “top” command, showing process status on Linux.
- This is a t2.micro virtual machine with only one CPU core.
- It's running a webserver with at least 11 separate processes (httpd.worker).
- While one process is blocked (meaning *busy*, eg., waiting to read data from an HTML file) another process can handle a different user’s request.

Cloud Computing makes scaling easier

- **Vertical scaling:** change the **instance type** of a virtual machine. Eg., upgrade from:
  - **t3.nano** (<2 cores, 0.5GB RAM, remote SSD disk) $.0052/hour …to…
  - **m5d.24xlarge** (96 cores, 386GB RAM, local NVMe SSD disk) $5.424/hour
- Vertical scaling (up or down) just requires a reboot of the VM.

- **Horizontal scaling:** purchase more VM instances.
  - The new instance will be available to use in just a few minutes.

- We call cloud computing resources "**elastic**" because you can quickly change the size and quantity of the computing resources you are using.

Vertical Scaling pros and cons

- ✓ Easy to write your programs.
- ✓ Most languages have support for multithreading.
- ✓ Most "off the shelf" software (commercial or open source) is written to run on one machine.
  - Eg.: MySQL, Oracle DB, Apache, Nginx, Node.js, etc
- ✓ Modern servers can do a lot of work in parallel with ~96 cores.
- ✓ Can connects hundreds of disks to a machine before overwhelming I/O bandwidth.
- ✓ Avoids slow communication with outside machines.

- ✕ Cannot handle really huge loads.
- ✕ Cannot be scaled quickly in a fine-grained manner.
  - Ie., must replace entire machine instead of just adding one more node.
- ✕ Single point of failure.
- ✕ Price/performance ratio is poor for top-of-the-line machines.
  - ✕ Motherboards with many sockets are expensive.
  - ✕ Fastest CPUs are expensive.

- **Vertical scaling is not scalable!**

Horizontal Scaling is needed for global apps

- Public Cloud Computing providers can give you lots of machines, but making good use of them is very difficult.
- Most of this class will address the coordination of execution and data in horizontally-scaled systems.

Recap

- A software **service** is a program that runs continuously, giving responses to requests.
- **Scalability** is the ability of a service to grow to handle many concurrent users (ideally an arbitrarily large number).
- Two approaches to scaling that are useful in different scenarios:
- **Vertical scaling** is upgrading your machine(s).
  - The simplest and most efficient way of scaling… but there is a ceiling.
- **Horizontal scaling** is adding more machines.
  - Coordinating a cluster of machines is complicated, but it's necessary for global scale and massive throughput.