Tunes

“Good Ol’ Fashion Nightmare” (Matt & Kim)

Matt & Kim

“I think it’s pretty clear what we were singing about.”

- Matt & Kim
YES.

418 exams are long.

(we did warn you)
But...
In many situations.
THE QUESTIONS.

That were attempted.

had answers...

that seem to indicate...
that perhaps.
SOME FUNDAMENTAL CONCEPTS
may not be understood
may not be understood at the level
your instructor might prefer.
AND SO
HERE
THE DEAL.
The Exam 1 Deal

- No exam 1 solutions will be distributed at this time
- You have the opportunity to redo up to two questions (of your choosing) from the exam, on your own time.
  - You may discuss the problems with your classmates and the TAs.
  - You must write your solutions on your own.
  - You will get 50% credit for lost points on regraded questions.
  - This must be completed by April 11th

But... there’s a catch.
Exam 1 Deal: The Catch

- You must hand in your solution to Kayvon at office hours

- And you are not allowed to hand in unless you are able to successfully answer a series of questions I ask you

- The questions will a subset of the seven questions on exam 1 (or simple follow up variants)
It’s time to start thinking about projects

- **Timeline**
  - Project proposal due: April 4th
  - Project checkpoint: April 18th
  - Parallelism competition finals! (project presentations): May 9th

- **Ideas**
  - Pick an application, parallelize it, and analyze its performance
  - Modify a parallel library or compilation tool
  - Write a hardware simulator, play around with FPGAs, do real hardware design
  - Free to experiment with fun new parallel platforms: FPGAs, mobile devices, Tegra devkits, Raspberry Pis, etc.

- We will be making a web page of ideas over spring break

- See examples from last year:
  - [http://15418.courses.cs.cmu.edu/spring2013/article/34](http://15418.courses.cs.cmu.edu/spring2013/article/34)
Today’s focus: the basics of scaling a web site

- I’m going to focus on performance issues
  - Parallelism and locality

- Many other issues in developing a successful web platform
  - Reliability, security, privacy, etc.
  - There are other great courses at CMU for these topics
A simple web server for static content

```c
while (1)
{
    request = wait_for_request();
    filename = parse_request(request);
    contents = read_file(filename);
    send contents as response
}
```

Question: is site performance a question of throughput or latency? (we’ll revisit this question later)
A simple parallel web server

What factors would you consider in setting the value of N for a multi-core web server?

- Parallelism: use all the server’s cores
- Latency hiding: hide long-latency disk read operations (by context switching between worker processes)
- Concurrency: many outstanding requests, want to service quick requests while long requests are in progress (e.g., large file transfer shouldn’t block serving index.html)
- Footprint: don’t want too many threads so that aggregate working set causes thrashing

```c
while (1)
{
    request = wait_for_request();
    filename = parse_request(request);
    contents = read_file(filename);
    send contents as response
}
```
Example: Apache’s parent process dynamically manages size of worker pool

Desirable to maintain a few idle workers in pool (avoid process creation in critical path of servicing requests)
Limit maximum number of workers to avoid excessive memory footprint (thrashing)

Key parameter of Apache's "prefork" multi-processing module: MaxRequestWorkers
Aside: why partition server into processes, not threads?

- **Protection**
  - Don’t want a crash in one worker to bring down the whole web server
  - Often want to use non-thread safe libraries (e.g., third-party libraries) in server operation

- **Parent process can periodically recycle workers**
  (robustness to memory leaks)

- **Of course, multi-threaded web server solutions exist as well**
  (e.g., Apache’s “worker” module)
Dynamic web content

"Response" is not a static page on disk, but the result of application logic running in response to a request.
Consider the amount of logic and the number database queries required to generate your Facebook News Feed.
Scripting language performance (poor)

- Two popular content management systems (PHP)
  - Wordpress ~ 12 requests/sec/core (DB size = 1000 posts)
  - MediaWiki ~ 8 requests/sec/core  
    [Source: Talaria Inc.]

- Recent interest in making scripted code execute faster
  - Facebook’s HipHop: PHP to C source-to-source converter
  - Google’s V8 Javascript engine: JIT Javascript to machine code
“Scale out” to increase throughput
Use many web servers to meet site’s throughput goals.

Requests

Load Balancer

Web Server
Worker Process

Web Server
Worker Process

Web Server
Worker Process

Database (e.g., mySQL)

Load balancer maintains list of available web servers and an estimate of load on each.

Distributes requests to pool of web servers. (Redistribution logic is cheap: one load balancer typically can service many web servers)
Load balancing with persistence

All requests associated with a session are directed to the same server (aka. session affinity, “sticky sessions”)

Good:
- Do not have to change web-application design to implement scale out

Bad:
- Stateful servers can limit load balancing options. Also, session is lost if server fails
Desirable: avoid persistent state in web server

Maintain stateless servers, treat sessions as persistent data to be stored in the DB.

1. SessionId = X
2. SessionId = Y
3. SessionId = X
4. SessionId = X

Database (e.g., mySQL)

Session State
Dealing with database contention

Option 1: “scale up”: buy better hardware for database server, buy professional-grade DB that scales
Good: no change to software
Bad: High cost, limit to scaling
Scaling out a database: replicate

Replicate data and parallelize reads
(most DB accesses are reads)
Cost: extra storage, consistency issues

Adopt relaxed consistency models:
propagate updates “eventually”
Scaling out a database: partition

Can tune database for access characteristics of data stored (common to use different databases: SQL vs. nosql)
Inter-request parallelism

Parallelize generation of a single page

Amount of user traffic is directly correlated to response latency.

How many web servers do you need?
Web traffic is bursty

Amazon.com Page Views

HuffingtonPost.com Page Views Per Week

Holiday shopping season

HuffingtonPost.com Page Views Per Day

(fewer people read news on weekends)

More examples:
- Facebook gears up for bursts of image uploads on Halloween and New Year’s Eve.
- Twitter topics trend after world events
Exam 1
Problem

- Site load is bursty

- Provisioning site for the average case load will result in poor quality of service (or failures) during peak usage
  - Peak usage tends to be when users care the most... since by the definition the site is important at these times

- Provisioning site for the peak usage case will result in many idle servers most of the time
  - Not cost efficient (must pay for many servers, power/cooling, datacenter space, etc.)
Elasticity!

- Main idea: site automatically adds or shuts down web servers based on measured load

- Need source of servers available on-demand
  - Example: Amazon.com EC2 instances
Example: Amazon’s elastic compute cloud (EC2)

- Amazon had an over-provisioning problem
- Solution: make machines available for rent to others in need of compute
  - For those that don’t want to incur cost of, or have expertise to, manage own machines at scale
  - For those that need elastic compute capability

<table>
<thead>
<tr>
<th>vCPU</th>
<th>ECU</th>
<th>Memory (GiB)</th>
<th>Instance Storage (GB)</th>
<th>Linux/UNIX Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>3.75</td>
<td>1 x 4 SSD</td>
</tr>
<tr>
<td>m3.large</td>
<td>2</td>
<td>6.5</td>
<td>7.5</td>
<td>1 x 32 SSD</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>15</td>
<td>2 x 40 SSD</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>8</td>
<td>26</td>
<td>30</td>
<td>2 x 80 SSD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vCPU</th>
<th>ECU</th>
<th>Memory (GiB)</th>
<th>Instance Storage (GB)</th>
<th>Linux/UNIX Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>1.7</td>
<td>1 x 160</td>
</tr>
<tr>
<td>m1.medium</td>
<td>1</td>
<td>2</td>
<td>3.75</td>
<td>1 x 410</td>
</tr>
<tr>
<td>m1.large</td>
<td>2</td>
<td>4</td>
<td>7.5</td>
<td>2 x 420</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>4</td>
<td>8</td>
<td>15</td>
<td>4 x 420</td>
</tr>
</tbody>
</table>
Site configuration: normal load

Requests

Load Balancer

Web Server

Web Server

Web Server

Database (potentially multiple machines)

DB Slave 1

DB Slave 2

Master

Perf. Monitor
Load: moderate
Event triggers spike in load

Heavily loaded servers: slow response times

Database (potentially multiple machines)
- DB Slave 1
- DB Slave 2
- Master

Load Balancer

Requests

Perf. Monitor
Load: high
Heavily loaded servers = slow response times

- If requests arrive faster than site can service them, queue lengths will grow
- **Latency** of servicing request is wait time in queue + time to actually process request
  - Assume site has capability to process $R$ requests per second
  - Assume queue length is $L$
  - Time in queue = $L/R$

- How does site **throughput** change under heavy load?
Site configuration: high load

Site performance monitor detects high load
Instantiates new web server instances
Informs load balancer about presence of new servers

Requests

Load Balancer

Perf. Monitor
Load: moderate

Web Server

Web Server

Web Server

Web Server

Web Server

Database
(potentially multiple machines)

DB Slave 1

DB Slave 2

Master
Site configuration: return to normal load

Site performance monitor detects low load
Released extra server instances (to save operating cost)
Informs load balancer about loss of servers

Note convenience of stateless servers in elastic environment:
can kill server without loss of important information.

@justinbieber: WTF, parallel programming is 2 hrd. Buy my new album.
Today: many “turn-key” environment-in-a-box services

Offer elastic computing environments for web applications

CloudWatch+Auto Scaling
Amazon Elastic Beanstalk

Amazon Web Services

Google App Engine

RightScale

Engine Yard
The story so far: parallelism scale out, scale out, scale out

(+ elasticity to be able to scale out on demand)

Now: reuse and locality
Recall: basic site configuration

Example PHP Code

```php
$query = "SELECT * FROM users WHERE username='kayvonf';
$user = mysql_fetch_array(mysql_query($userquery));

echo "<div>" . $user['FirstName'] . " " . $user['LastName'] . "</div>";
```

Response Information Flow

HTML

<table>
<thead>
<tr>
<th>PHP 'user' object</th>
</tr>
</thead>
<tbody>
<tr>
<td>'users' table</td>
</tr>
</tbody>
</table>

<div>Kayvon Fatahalian</div>
Work repeated every page

Example PHP Code

```php
$query = "SELECT * FROM users WHERE username='kayvonf';
$user = mysql_fetch_array(mysql_query($userquery));

echo "<div>" . $user['FirstName'] . " " . $user['LastName'] . "</div>";
```

Response Information Flow

HTML → PHP 'user' object → 'users' table

Steps repeated to emit my name at the top of every page:
- Communicate with DB
- Perform query
- Marshall results from database into object model of scripting language
- Generate presentation
- etc...

Remember, DB can be hard to scale!
Solution: cache!

- Cache commonly accessed objects
  - Example: memcached, in memory key-value store (e.g., a big hash table)
  - Reduces database load (fewer queries)
  - Reduces web server load:
    - Less data shuffling between DB response and scripting environment
    - Store intermediate results of common processing
Caching example

```php
userid = $_SESSION['userid'];

check if memcache->get(userid) retrieves a valid user object

if not:
    make expensive database query
    add resulting object into cache with memcache->put(userid)
    (so future requests involving this user can skip the query)

continue with request processing logic
```

- Of course, there is complexity associated with keeping caches in sync with data in the DB in the presence of writes
  - Must invalidate cache
  - Very simple “first-step” solution: only cache read-only objects
  - More realistic solutions provide some measure of consistency
    - But we’ll leave this to your distributed computing and database courses
Site configuration

- Requests
- Load Balancer
- Perf. Monitor
- Web Server
- Web Server
- Web Server
- Web Server
- Web Server
- Database (potentially multiple machines)
  - DB Slave 1
  - DB Slave 2
  - Master

memcached servers

value = get(key)
put(key, value)
Example: Facebook memcached deployment

- Facebook, circa 2008
  - 800 memcached servers
  - 28 TB of cached data

- Performance
  - 200,000 UDP requests per second @ 173 msec latency
  - 300,000 UDP requests per second possible at “unacceptable” latency

More caching

- Cache web server responses (e.g. entire pages, pieces of pages)
  - Reduce load on web servers
  - Example: Varnish-Cache application “accelerator”
Caching using content distribution networks (CDNs)

- Serving large media assets can be expensive to serve (high bandwidth costs, tie up web servers)
  - E.g., images, streaming video
- Physical locality is important
  - Higher bandwidth
  - Lower latency

Source: http://www.telco2.net/blog/2008/11/amazon_cloudfront_yet_more_tra.html
CDN usage example

Facebook photo:
Page URL:

Image source URL:
https://sphotos-a.xx.fbcdn.net/hphotos-prn1/522152_10151325164543897_1133820438_n.jpg
CDN integration

Local CDN (Pittsburgh)

Page Requests

Media Requests

Local CDN (San Francisco)

Perf. Monitor

Load Balancer

Front-End Cache

Web Server

Database

Memcached servers

DB Slave 1

DB Slave 2

Master

Page Requests

Media Requests

Front-End Cache

Front-End Cache

Front-End Cache

Front-End Cache

Web Server
Summary: scaling modern web sites

- **Use parallelism**
  - Scale-out parallelism: leverage many web servers to meet throughput demand
  - Elastic scale-out: cost-effectively adapt to bursty load
  - Scaling databases can be tricky (replicate, shard, partition by access pattern)
    - Consistency issues on writes

- **Exploit locality and reuse**
  - Cache everything (key-value stores)
    - Cache the results of database access (reduce DB load)
    - Cache computation results (reduce web server load)
    - Cache the results of processing requests (reduce web server load)
  - Localize cached data near users, especially for large media content (CDNs)

- **Specialize implementations for performance**
  - Different forms of requests, different workload patterns
Final comments

- It is true that performance of straight-line application logic is often very poor in web-programming languages (orders of magnitude left on the table in Ruby and PHP).

- BUT... web development is not just trivial hacking in slow scripting languages. Scaling a web site is a very challenging parallel-systems problem that involves many of the optimization techniques and design choices studied in this class: just at different scales
  - Identifying parallelism and dependencies
  - Workload balancing: static vs. dynamic partitioning issues
  - Data duplication vs. contention
  - Throughput vs. latency trade-offs
  - Parallelism vs. footprint trade-offs
  - Identifying and exploiting reuse and locality

- Many great sites (and blogs) on the web to learn more:
  - www.highscalability.com has great case studies (see “All Time Favorites” section)
  - James Hamilton’s blog: http://perspectives.mvdirona.com
Have a nice spring break!