Software Engineering Advice from Building Large-Scale Distributed Systems

Jeff Dean

http://labs.google.com/people/jeff
Context

• Lessons drawn from work across a broad range of areas
  – Products (ad serving systems, AdSense, four generations of web search crawling, indexing, and query serving systems, Google News, statistical machine translation, Google Codesearch, etc.)
  – Infrastructure (core indexing/search product components, MapReduce, BigTable, cluster scheduling systems, indexing service, core libraries, etc.)
  – Software tools (profiling systems, fast searching over source tree, etc.)
  – Other (system design advice, hiring process involvement)

• Talk is an unorganized set of tips drawn from this experience
  – Feel free to ask questions
Google Computing Environment

• Large clusters of commodity PCs
  – not ultra-reliable
  – … but cheap
  – best performance/$

• Lots of stuff can go very wrong
The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.
Google Engineering Environment

- Distributed systems
  - data or request volume or both are too large for single machine
    - careful design about how to partition problems
    - need high capacity systems even within a single datacenter
  - multiple datacenters, all around the world
    - almost all products deployed in multiple locations
Environment (cont.)

• Products are mostly services, not shrink-wrapped software
  – services used heavily even internally
    • web search might touch 50 separate services, thousands of machines
  – simpler from a software engineering standpoint
    • fewer dependencies, clearly specified
    • easy to test new versions
    • ability to run lots of experiments

  – development cycles of products largely decoupled
    • lots of benefits: small teams can work independently
    • easier to have many engineering offices around the world
Designing software systems is tricky

- Need to balance:
  - Simplicity
  - Scalability
  - Performance
  - Reliability
  - Generality
  - Features
Get Advice Early!

- Before you write any code
- Before you write any lengthy design documents

Instead:
- Jot down some rough ideas (a few paragraphs)
- Go find some people and chat at a whiteboard
  - Especially people familiar with building similar systems
- Even better: discuss a few different potential designs & evaluate…
Interfaces

- Think carefully about interfaces in your system!
- Imagine other hypothetical clients trying to use your interface
- Document precisely, but avoid constraining implementation
  - Very important to be able to reimplement
- Get feedback on your interfaces before implementing!
- Best way to learn is to look at well-designed interfaces
Protocols

- Good protocol description language is vital
- Desired attributes:
  - self-describing, multiple language support
  - efficient to encode/decode, compact serialized form

Our solution: Protocol Buffers (in active use since 2000)

```proto
message SearchResult {
  required int32 estimated_results = 1
  optional string error_message = 2;
  repeated group Result = 3 {
    required float score = 4;
    required fixed64 docid = 5;
    optional message<WebResultDetails> = 6;
    ...
  }
}
```
• Automatically generated language wrappers
• Graceful client and server upgrades
  – systems ignore tags they don’t understand, but pass the information through (no need to upgrade intermediate servers)
• Serialization/deserialization
  – lots of performance work here, benefits all users of protocol buffers
  – format used to store data persistently (not just for RPCs)
• Also allow service specifications:

```java
service Search {
  rpc DoSearch(SearchRequest) returns (SearchResponse);
  rpc DoSnippets(SnippetRequest)
    returns (SnippetResponse);
  rpc Ping(EmptyMessage) returns (EmptyMessage) {
    protocol=udp;
  }
};
```
Designing Efficient Systems

Important skill: given a basic problem definition, how do you choose the "best" solution?

• Best could be simplest, highest performance, easiest to extend, etc.

Important skill: ability to estimate performance of a system design
   – without actually having to build it!
# Numbers Everyone Should Know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>10,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from network</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

\[
30 \text{ seeks } \times 10 \text{ ms/seek} + 30 \times 256K \text{ } / \text{ } 30 \text{ MB/s} = 560 \text{ ms}
\]
Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

\[ 30 \text{ seeks} \times 10 \text{ ms/seek} + 30 \times 256K / 30 \text{ MB/s} = 560 \text{ ms} \]

Design 2: Issue reads in parallel:

\[ 10 \text{ ms/seek} + 256K \text{ read} / 30 \text{ MB/s} = 18 \text{ ms} \]
(Ignores variance, so really more like 30-60 ms, probably)
How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

- 30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms

Design 2: Issue reads in parallel:

- 10 ms/seek + 256K read / 30 MB/s = 18 ms
  (Ignores variance, so really more like 30-60 ms, probably)

Lots of variations:
- caching (single images? whole sets of thumbnails?)
- pre-computing thumbnails
- ...

Back of the envelope helps identify most promising…
How long to quicksort 1 GB of 4 byte numbers?

Comparison: lots of unpredictable branches

\[ \log(2^{28}) \] passes over \( 2^{28} \) numbers = \( \sim 2^{33} \) comparisons

\( \sim 1/2 \) will mispredict, so \( 2^{32} \) mispredicts * 5 ns/mispredict = 21 secs

Memory bandwidth: mostly sequential streaming

\( 2^{30} \) bytes * 28 passes = 28 GB. Memory BW is \( \sim 4 \) GB/s, so \( \sim 7 \) secs

So, it should take \( \sim 30 \) seconds to sort 1 GB on one CPU
Write Microbenchmarks!

- Great to understand performance
  - Builds intuition for back-of-the-envelope calculations
- Reduces cycle time to test performance improvements

Examples: `gsearch -i -w BENCHMARK -f='(cc|java)$'`

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time(ns)</th>
<th>CPU(ns)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_VarintLength32/0</td>
<td>2</td>
<td>2</td>
<td>291666666</td>
</tr>
<tr>
<td>BM_VarintLength32Old/0</td>
<td>5</td>
<td>5</td>
<td>124660869</td>
</tr>
<tr>
<td>BM_VarintLength64/0</td>
<td>8</td>
<td>8</td>
<td>89600000</td>
</tr>
<tr>
<td>BM_VarintLength64Old/0</td>
<td>25</td>
<td>24</td>
<td>42164705</td>
</tr>
<tr>
<td>BM_VarintEncode32/0</td>
<td>7</td>
<td>7</td>
<td>80000000</td>
</tr>
<tr>
<td>BM_VarintEncode64/0</td>
<td>18</td>
<td>16</td>
<td>39822222</td>
</tr>
<tr>
<td>BM_VarintEncode64Old/0</td>
<td>24</td>
<td>22</td>
<td>31165217</td>
</tr>
</tbody>
</table>
Know Your Basic Building Blocks

Core language libraries, basic data structures, SSTables, protocol buffers, GFS, BigTable, indexing systems, MySQL, MapReduce, …

Not just their interfaces, but understand their implementations (at least at a high level)

If you don’t know what’s going on, you can’t do decent back-of-the-envelope calculations!
Building Infrastructure

Identify common problems, and build software systems to address them in a general way

• Important not to try to be all things to all people
  – Clients might be demanding 8 different things
  – Doing 6 of them is easy
  – ...handling 7 of them requires real thought
  – ...dealing with all 8 usually results in a worse system
    • more complex, compromises other clients in trying to satisfy everyone

Don't build infrastructure just for its own sake
• Identify common needs and address them
• Don't imagine unlikely potential needs that aren't really there
• Best approach: use your own infrastructure (especially at first)
  – (much more rapid feedback about what works, what doesn't)
Design for Growth

Try to anticipate how requirements will evolve
keep likely features in mind as you design base system

Ensure your design works if scale changes by 10X or 20X
but the right solution for X often not optimal for 100X
Design for Low Latency

• Aim for low avg. times (happy users!)
  – 90%ile and 99%ile also very important
  – Think about how much data you’re shuffling around
    • e.g. dozens of 1 MB RPCs per user request -> latency will be lousy

• Worry about variance!
  – Redundancy or timeouts can help bring in latency tail
• Judicious use of caching can help
• Use higher priorities for interactive requests
• Parallelism helps!
Consistency

• Multiple data centers implies dealing with consistency issues
  – disconnected/partitioned operation relatively common
    • e.g. datacenter down for maintenance
  – insisting on strong consistency likely undesirable
    • "We have your data but can't show it to you because one of the replicas is unavailable"
  – most products with mutable state gravitating towards "eventual consistency" model
    • a bit harder to think about, but better from an availability standpoint
Threads

If you’re not using threads, you’re wasting ever larger fractions of typical machines

Machines have 4 cores now, 8 soon, going to 16.
  – think about how to parallelize your application!
  – multi-threaded programming really isn't very hard if you think about it upfront

• Threading your application can help both throughput and latency
Understand your Data Access &

• Data access:
  – Disks: seeks, sequential reads, etc.
  – Memory: think about caches, branch predictors, etc.

• RPCs:
  – know how much data you’re sending/receiving
  – will you saturate your machine’s network interface?
  – what about the rack switch?
Encoding Your Data

- CPUs are fast, memory/bandwidth are precious, ergo…
  - Variable-length encodings
  - Compression
  - Compact in-memory representations

- Compression very important aspect of many systems
  - inverted index posting list formats
  - storage systems for persistent data

- We have lots of core libraries in this area
  - Many tradeoffs: space, encoding/decoding speed, etc.
Making Applications Robust Against Failures

Canary requests
Failover to other replicas/datacenters
Bad backend detection:
  stop using for live requests until behavior gets better
More aggressive load balancing when imbalance is more severe

Make your apps do something reasonable even if not all is right
  Better to give users limited functionality than an error page
Add Sufficient Monitoring/Status/Debugging Hooks

All our servers:

• Export HTML-based status pages for easy diagnosis
• Export a collection of key-value pairs via a standard interface
  – monitoring systems periodically collect this from running servers
• RPC subsystem collects sample of all requests, all error requests, all requests >0.0s, >0.05s, >0.1s, >0.5s, >1s, etc.

• Support low-overhead online profiling
  – cpu profiling
  – memory profiling
  – lock contention profiling

If your system is slow or misbehaving, can you figure out why?
Source Code Philosophy

• Google has one large shared source base
  – lots of lower-level libraries used by almost everything
  – higher-level app or domain-specific libraries
  – application specific code

• Many benefits:
  – improvements in core libraries benefit everyone
  – easy to reuse code that someone else has written in another context

• Drawbacks:
  – reuse sometimes leads to tangled dependencies

• Essential to be able to easily search whole source base
  – gsearch: internal tool for fast searching of source code
  – huge productivity boost: easy to find uses, defs, examples, etc.
  – makes large-scale refactoring or renaming easier
Software Engineering Hygiene

- Code reviews
- Design reviews
- Lots of testing
  - unit tests for individual modules
  - larger tests for whole systems
  - continuous testing system

- Most development done in C++, Java, & Python
  - C++: performance critical systems (e.g. everything for a web query)
  - Java: lower volume apps (advertising front end, parts of Gmail, etc.)
  - Python: configuration tools, etc.
Multi-Site Software Engineering

- Google has moved from one to a handful to 20+ engineering sites around the world in last few years

- Motivation:
  - hire best candidates, regardless of their geographic location

- Issues:
  - more coordination needed
  - communication somewhat harder (no hallway conversations, time zone issues)
  - establishing trust between remote teams important

- Techniques:
  - online documentation, e-mail, video conferencing, careful choice of interfaces/project decomposition
  - BigTable: split across three sites
Fun Environment for Software Engineering

- Very interesting problems
  - wide range of areas: low level hw/sw, dist. systems, storage systems, information retrieval, machine learning, user interfaces, auction theory, new product design, etc.
  - lots of interesting data and computational resources
- Service-based model for software development is very nice
  - very fluid, easy to make changes, easy to test, small teams can accomplish a lot
- Great colleagues/environment
  - expertise in wide range of areas, lots of interesting talks, etc.
- Work has a very large impact
  - hundreds of millions of users every month
Questions?