15-440 Distributed Systems

16 - Spanner, Naming, and Hashing
Today’s Topics

- **Spanner**
  - Motivation
  - Key ideas

- **Naming**
  - Name resolution with DNS (again)
  - More DNS (security/availability)
  - Consistent hashing
  - Name by hash
  - Properties of hash functions
Spanner Within Google

[Shute et al, F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business, SIGMOD, 2012]
What is Spanner?

• Distributed multiversion database
  • General-purpose transactions (ACID)
  • SQL query language
  • Schematized tables
  • Semi-relational data model

• Running in production
  • Storage for Google’s ad data
  • Replaced a sharded MySQL database
Example: Social Network
Overview

• Feature: Lock-free distributed read transactions
• Property: External consistency of distributed transactions
  – First system at global scale
• Implementation: Integration of concurrency control, replication (Paxos), and 2PC
  – Correctness and performance
• Enabling technology: TrueTime
  – Interval-based global time
Read Transactions

• Generate a page of friends’ recent posts
  – Consistent view of friend list and their posts

Why consistency matters
1. Remove untrustworthy person X as friend
2. Post P: “My government is repressive...”
Single Machine

Block writes

Friend1 post
Friend2 post
...
Friend999 post
Friend1000 post

User posts
Friend lists

Generate my page

OSDI 2012
Multiple Machines

Block writes

Friend1 post
Friend2 post
...
Friend999 post
Friend1000 post

User posts
Friend lists

Generate my page

OSDI 2012
Multiple Datacenters

Friend1 post
US

Friend2 post
Spain
...

Friend999 post
Brazil

Friend1000 post
Russia

Generate my page

OSDI 2012
Spanner Concurrency Control

• Key aspect of differentiating Spanner – using globally meaningful timestamps for distributed transactions in achieving external consistency

\[ T_1; s_1 \]

\[ s_1: \text{Timestamp} \]
Time-Based Version Management

- Transactions that write use strict 2PL
  - Each transaction $T$ is assigned a timestamp $s$
  - Data written by $T$ is timestamped with $s$

<table>
<thead>
<tr>
<th>Time</th>
<th>&lt;8</th>
<th>8</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>My friends</td>
<td>[X]</td>
<td>[]</td>
<td></td>
</tr>
<tr>
<td>My posts</td>
<td>[]</td>
<td>[P]</td>
<td></td>
</tr>
<tr>
<td>X’s friends</td>
<td>[me]</td>
<td>[]</td>
<td></td>
</tr>
</tbody>
</table>
Timestamps, Global Clock

• Strict two-phase locking for write transactions
• Assign timestamp while locks are held

Pick \( s = \text{now()} \)
Timestamp Invariants

- Timestamp order == commit order

- Timestamp order respects global wall-time order
TrueTime

• “Global wall-clock time” with bounded uncertainty
Spanner TrueTime API

**TT.now()** – TTInterval: [earliest, latest]
(Why is time an interval and not a specific instant?)
**TT.after(t)** – true if t has definitely passed
**TT.before(t)** – true if t has definitely not arrived

```
TT.after(t) = true  TT.now()  TT.before(t) = true
```

```
earliest                  latest
```

“Global wall-clock time” with bounded uncertainty
Why do you need to wait for TT.now().earliest > \( s \) before releasing locks?
Commit Wait and Replication

Acquired locks

Start consensus

Achieve consensus

Notify slaves

Release locks

Pick s

Commit wait done

T
Spanner External Consistency

• If a transaction $T_1$ commits before another transaction $T_2$ starts, then $T_1$'s commit timestamp is smaller than $T_2$

• Similar to how we reason with wall-clock time

\[ s_1 < s_2 \]
TrueTime Architecture

Datacenter 1

GPS timemaster

GPS timemaster

GPS timemaster

GPS timemaster

Atomic-clock timemaster

GPS timemaster

GPS timemaster

GPS timemaster

Client

Datacenter 2

... Datacenter n

Compute reference [earliest, latest] = now ± ε
TrueTime implementation

\[ \text{now} = \text{reference now} + \text{local-clock offset} \]

\[ \varepsilon = \text{reference } \varepsilon + \text{worst-case local-clock drift} \]
Spanner Summary

- Globally consistent replicated database system
- Implements distributed transactions
  - Uses 2PC
- Fault-tolerance and replicated writes
  - Uses Paxos based
- Sync-free reads, external consistency
  - Uses TrueTime API
- Able to survive data center wipeouts
- GCD Cloud Spanner ⇔ Azure Cosmos DB, AWS Dynamo DB
Today’s Topics

- Spanner
  - Motivation
  - Key ideas

- Naming
  - Name resolution with DNS (again)
  - More DNS (security/availability)
  - Consistent hashing
  - Name by hash
  - Properties of hash functions
Names

- Names are associated with objects
  - Enables passing of references to objects
  - Indirection
  - Deferring decision on meaning/binding

- Examples
  - Registers $\rightarrow$ R5
  - Memory $\rightarrow$ 0xdeadbeef
  - Host names $\rightarrow$ srini.com
  - User names $\rightarrow$ sseshan
  - Email $\rightarrow$ srini@cmu.edu
  - File name $\rightarrow$ /usr/srini/foo.txt
  - URLs $\rightarrow$ http://www.srini.com/index.html
  - Ethernet $\rightarrow$ f8:e4:fb:bf:3d:a6
Name Discovery

- Well-known name
  - www.google.com, port 80...
- Broadcast
  - Advertise name → e.g. 802.11 Beacons
- Query
  - Use google
- Broadcast query
  - Ethernet ARP
- Use another naming system
  - DNS returns IP addresses
- Introductions
  - Web page hyperlinks
- Physical rendezvous
  - Exchange info in the real world

- OTHER KEY DIFFERENCES AS WELL....
Today’s Topics

• Spanner
  • Motivation
  • Key ideas

• Naming
  • Name resolution with DNS (again)
  • More DNS (security/availability)
  • Consistent hashing
  • Name by hash
  • Properties of hash functions
DNS Records

RR format: (class, name, value, type, ttl)

• DB contains tuples called resource records (RRs)
  • Classes = Internet (IN), Chaosnet (CH), etc.
  • Each class defines value associated with type

FOR IN class:

• Type=A
  • **name** is hostname
  • **value** is IP address
• Type=NS
  • **name** is domain (e.g. foo.com)
  • **value** is name of authoritative name server for this domain
• Type=CNAME
  • **name** is an alias name for some “canonical” (the real) name
  • **value** is canonical name
• Type=MX
  • **value** is hostname of mailserver associated with **name**
DNS Design: Zone Definitions

- Zone = contiguous section of name space
  - E.g., Complete tree, single node or subtree
- A zone has an associated set of name servers
  - Must store list of names and tree links

![Diagram of DNS zone definitions]

- Subtree
- Single node
- Complete Tree
Typical Resolution

- Client
- Local DNS server
- www.cs.cmu.edu
- ns1.cs.cmu.edu
- ns1.cs.cmu.edu DNS server
- www.cs.cmu.edu DNS server
- root & edu DNS server
- A www=IPaddr
- NS ns1.cs.cmu.edu
- NS ns1.cmu.edu
Subsequent Lookup Example

- Client
- Local DNS server
- ftp.cs.cmu.edu
- root & edu DNS server
- cmu.edu DNS server
- cs.cmu.edu DNS server
- ftp=IPaddr
Querying DNS: the “dig” Lookup Utility

- Linux/MacOS, ssh into unix.andrew.cmu.edu
- Usage (see man dig)

```bash
> dig +norecurse @a.root-servers.net NS dberger.sp.cs.cmu.edu
;; AUTHORITY SECTION:
edu. 172800 IN NS a.edu-servers.net.
edu. 172800 IN NS c.edu-servers.net.
...```
DNS (Summary)

- Motivations → large distributed database
  - Scalability
  - Independent update
  - Robustness
- Hierarchical database structure
  - Zones
  - How is a lookup done
- Caching/prefetching and TTLs
- Reverse name lookup
- What are the steps to creating your own domain?
Today’s Topics

• Spanner
  • Motivation
  • Key ideas

• Naming
  • Name resolution with DNS (again)
  • More DNS (security/availability)
  • Consistent hashing
  • Name by hash
  • Properties of hash functions
Protecting the Root Nameservers

**Attack On Internet Called Largest Ever**

By David McGuire and Brian Krebs  
washingtonpost.com Staff Writers  
Tuesday, October 22, 2002; 5:40 PM

The heart of the Internet sustained its largest and most sophisticated attack ever, starting late Monday, according to officials at key online backbone organizations.

Around 5:00 p.m. EDT on Monday, a "distributed denial of service" (DDOS) attack struck the 13 "root servers" that provide the primary roadmap for almost all Internet communications. Despite the scale of the attack, which lasted about an hour, Internet users worldwide were largely unaffected, experts said.

**Defense Mechanisms**

- Redundancy: 13 root nameservers
- IP Anycast for root DNS servers {c,f,i,j,k}.root-servers.net
  - RFC 3258
  - Most *physical* nameservers lie outside of the US

Sophisticated? Why did nobody notice?

What Happened on Oct 21st 2016?

- DDoS attack on Dyn
- Dyn provides core Internet services for Twitter, SoundCloud, Spotify, Reddit and a host of other sites
- Why didn’t DNS defense mechanisms work in this case?
- Let’s take a look at the DNS records…
What was the source of attack?

- **Mirai botnet**
  - Used in 620Gbps attack last month

- **Source:** bad IoT devices, e.g.,
  - White-labeled DVR and IP camera electronics
  - username: root and password: `xc3511`
  - password is hardcoded into the device firmware
Attack Waves

- DNS lookups are routed to the nearest data center
- First wave
  - On three Dyn data centers – Chicago, Washington, D.C., and New York
- Second wave,
  - Hit 20 Dyn data centers around the world.
  - Required extensive planning.
  - Since DNS request go to the closest DNS server, the attacker had to plan a successful attack for each of the 20 data centers with enough bots in each region to be able to take down the local Dyn services
Solutions?

- Dyn customers
  - Going to backup DNS providers
  - Signing up with an alternative today after the attacks, as PayPal did

- Lowering their time-to-life settings on their DNS servers
  - Redirect traffic faster to another DNS service that is still available
DNS Dist Sys Lessons

- Availability and reliability
- Caching and consistency
- Federated design
- Security
- Scalability
- Coordination/standardization
Today’s Topics

• Spanner
  • Motivation
  • Key ideas

• Naming
  • Name resolution with DNS (again)
  • More DNS (security/availability)
  • Consistent hashing
  • Name by hash
  • Properties of hash functions
Hashing

Two uses of hashing that are wildly popular in distributed systems:

- Consistent Hashing of various forms
- Content-based naming
Example systems that use them

• Content distribution networks such as Akamai use consistent hashing to place content on servers
• Amazon, Linkedin, etc., all have built very large-scale key-value storage systems (databases--) using consistent hashing
• BitTorrent & many other modern p2p systems use content-based naming
Problem: Dividing items onto storage servers

- Option 1: Static partition (items a-c go there, d-f go there, ...)
  - If you used the server name, what if “cowpatties.com” had 1000000 pages, but “zebras.com” had only 10? → Load imbalance
  - Could fill up the bins as they arrive → Requires tracking the location of every object at the front-end.
Hashing

- Let nodes be numbered 1..m
- Client uses a *good* hash function to map a URL to 1..m
- Say hash (url) = x, so, client fetches content from node x
- No duplication – not being fault tolerant.
- Any other problems?
  - What happens if a node goes down?
  - What happens if a node comes back up?
  - What if different nodes have different views?
Option 2: Conventional Hashing

- bucket = hash(item) / num_buckets
- Sweet! Now the server we use is a deterministic function of the item, e.g., sha1(URL) → 160 bit ID / 20 → a server ID
- But what happens if we want to add or remove a server?
Option 2: Conventional Hashing

- 90 documents, node 1..9, node 10 which was dead is alive again
- Simple assignment:
  - ID/9 (for 9 servers) vs. ID/10 (for 10 servers)
- % of documents in the wrong node?
  - 10, 19-20, 28-30, 37-40, 46-50, 55-60, 64-70, 73-80, 82-90
  - Disruption coefficient = ½ 😞
Consistent Hash

• “view” = subset of all hash buckets that are visible
• Desired features
  • Balanced – in any one view, load is equal across buckets
  • Smoothness – little impact on hash bucket contents when buckets are added/removed
  • Spread – small set of hash buckets that may hold an object regardless of views
  • Load – across all views # of objects assigned to hash bucket is small
Consistent Hashing

- **Main idea:**
  - map both *keys* and *nodes* to the same (metric) *identifier space*
  - find a “rule” how to assign keys to nodes

*Ring is one option.*
Consistent Hashing

• The consistent hash function assigns each node and key an $m$-bit identifier using SHA-1 as a base hash function

• **Node identifier:** SHA-1 hash of IP address

• **Key identifier:** SHA-1 hash of key
Identifiers

- \( m \) bit identifier space for both keys and nodes

- **Key identifier**: SHA-1(key)
  
  \[
  \text{Key=“LetItBe”} \xrightarrow{\text{SHA-1}} \text{ID=60}
  \]

- **Node identifier**: SHA-1(IP address)
  
  \[
  \text{IP=“198.10.10.1”} \xrightarrow{\text{SHA-1}} \text{ID=123}
  \]

- How to map key IDs to node IDs?
**Rule**: A key is stored at its successor: node with next higher or equal ID

**Consistent Hashing Example**

- **IP**: “198.10.10.1”
- **Key**: “LetItBe”
Consistent Hash – Properties

- Ring-based construction using hash of key and node
- Smoothness → addition of bucket does not cause much movement between existing buckets
- Load → For $N$ nodes and $K$ keys, with high probability, each node holds at most $(1+\varepsilon)K/N$ keys
  - (for large $K$ compared to $N$)
- Spread → small set of buckets that lie near object
- Balance → no bucket is responsible for large number of objects
Today’s Topics

• Spanner
  • Motivation
  • Key ideas

• Naming
  • Name resolution with DNS (again)
  • More DNS (security/availability)
  • Consistent hashing
  • Name by hash
  • Properties of hash functions
Hashing 2: For naming

- Many file systems split files into blocks and store each block on a disk.
- Several levels of naming:
  - Pathname to list of blocks
  - Block #s are addresses where you can find the data stored therein. (But in practice, they’re logical block #s – the disk can change the location at which it stores a particular block... so they’re actually more like names and need a lookup to location :)}
Another problem to solve...

- Imagine you’re creating a backup server
- It stores the full data for 1000 CMU users’ laptops
- Each user has a 100GB disk.
- That’s 100TB and lots of $$$
- How can we reduce the storage requirements?
“Deduplication”

- A common goal in big archival storage systems. Those 1000 users probably have a lot of data in common -- the OS, copies of binaries, maybe even the same music or movies.

- How can we detect those duplicates and coalesce them?

- One way: Content-based naming, also called content-addressable foo (storage, memory, networks, etc.).

- A fancy name for...
Name items by their hash

• Imagine that your filesystem had a layer of indirection:
  • pathname → hash(data)
  • hash(data) → list of blocks

• For example:
  • /src/foo.c → 0xfff32f2fa11d00f0
  • 0xfff32f2fa11d00f0 → [5623, 5624, 5625, 8993]

• If there were two identical copies of foo.c on disk ... We’d only have to store it once!
  • Name of second copy can be different
A second example

- Several p2p systems operate something like:
  - Search for “national anthem”, find a particular file name (starspangled.mp3).
  - Identify the files by the hash of their content (0x2fab4f001...)
  - Request to download a file whose hash matches the one you want
  - Advantage? You can verify what you got, even if you got it from an untrusted source (like some dude on a p2p network)
Today’s Topics

- Spanner
  - Motivation
  - Key ideas

- Naming
  - Name resolution with DNS (again)
  - More DNS (security/availability)
  - Consistent hashing
  - Name by hash
  - Properties of hash functions
Desirable Properties of Hashes

- Compression: Maps a variable-length input to a fixed-length output
- Ease of computation: A relative metric...
- Pre-image resistance: For all outputs, computationally infeasible to find input that produces output.
- 2nd pre-image resistance: For all inputs, computationally infeasible to find second input that produces same output as a given input.
- Collision resistance: For all outputs, computationally infeasible to find two inputs that produce the same output.
Hash functions

• Given a universe of possible objects $U$, map $N$ objects from $U$ to an $M$-bit hash.

• Typically, $|U| >>> 2^M$.
  • This means that there can be collisions: Multiple objects map to the same $M$-bit representation.

• Likelihood of collision depends on hash function, $M$, and $N$.
  • Birthday paradox $\rightarrow$ roughly 50% collision with $2^{M/2}$ objects for a well designed hash function
Longevity

  • Moore’s law
  • Some day, maybe, perhaps, sorta, kinda: Quantum computing.

• Hash functions are not an exact science yet.
  • They get broken by advances in crypto.
## Real hash functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Introduced</th>
<th>Weakened</th>
<th>Broken</th>
<th>Lifetime</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD4</td>
<td>1990</td>
<td>1991</td>
<td>1995</td>
<td>1-5y</td>
<td>MD5</td>
</tr>
<tr>
<td>MD5</td>
<td>1992</td>
<td>1994</td>
<td>2004</td>
<td>8-10y</td>
<td>SHA-1</td>
</tr>
<tr>
<td>MD2</td>
<td>1992</td>
<td>1995</td>
<td>abandoned</td>
<td>3y</td>
<td>SHA-1</td>
</tr>
<tr>
<td>RIPEMD</td>
<td>1992</td>
<td>1997</td>
<td>2004</td>
<td>5-12y</td>
<td>RIPEMD-160</td>
</tr>
<tr>
<td>HAVAL-128</td>
<td>1992</td>
<td>-</td>
<td>2004</td>
<td>12y</td>
<td>SHA-1</td>
</tr>
<tr>
<td>SHA-0</td>
<td>1993</td>
<td>1998</td>
<td>2004</td>
<td>5-11y</td>
<td>SHA-1</td>
</tr>
<tr>
<td>SHA-1</td>
<td>1995</td>
<td>2004</td>
<td>not quite yet</td>
<td>9+</td>
<td>SHA-2 &amp; 3</td>
</tr>
<tr>
<td>SHA-2 (256, 384, 512)</td>
<td>2001</td>
<td>still good</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA-3</td>
<td>2012</td>
<td></td>
<td>brand new</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using them

- How long does the hash need to have the desired properties (preimage resistance, etc)?
  - rsync: For the duration of the sync;
  - dedup: Until a (probably major) software update;
  - store-by-hash: Until you replace the storage system

- What is the adversarial model?
  - Protecting against bit flips vs. an adversary who can try 1B hashes/second?
• Hashing forms the basis for MACs - message authentication codes
  • Basically, a hash function with a secret key.
  • \( H(\text{key, data}) \) - can only create or verify the hash given the key.
  • Very, very useful building block
Final pointer 2: Rabin Fingerprinting

Rabin Fingerprints

Hash 1

Hash 2

File Data

Natural Boundary

Natural Boundary

Given Value - 8

4  7  8

2  8
Summary

- Hashes used for:
  - Splitting up work/storage in a distributed fashion
  - Naming objects with self-certifying properties

- Key applications
  - Key-value storage
  - P2P content lookup
  - Deduplication
  - MAC

- Many types of naming
  - DNS names, IP addresses, Ethernet addresses, content-based addresses
  - Make sure you understand differences