Lecture 15 – Consistent hashing and name-by-hash

Outline

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

• Content-based naming
• Consistent Hashing of various forms

Example systems that use them

• BitTorrent & many other modern p2p systems use content-based naming
• 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

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 1

• 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.
• One hop access
• Any 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

- Let 90 documents, node 1..9, node 10 which was dead is alive again
- % 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 Hash – Example

- Construction
  - Assign each of C hash buckets to random points on mod 2^k circle, where, hash key size = n.
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket
  - Smoothness → addition of bucket does not cause much movement between existing buckets
  - Spread & Load → small set of buckets that lie near object
  - Balance → no bucket is responsible for large number of objects

Detail - “virtual” nodes

- The way we outlined it results in moderate load imbalance between buckets (remember balls and bins analysis of hashing?)
- To reduce imbalance, systems often represent each physical node as k different buckets, sometimes called “virtual nodes” (but really, it’s just multiple buckets).
- log n buckets gets you a very pleasing load balance - O(#items/n) with high probability, if #items large and uniformly distributed

Use of consistent hashing

- Consistent hashing was first widely used by Akamai CDN
- Also used in systems like Chord DHT
  - Provided key-value storage, designed to scale to millions or billions of nodes
  - Had a p2p lookup algorithm, completely decentralized, etc. Fun to read about; very influential, but not widely used outside of p2p systems.
- In practice, many more systems use consistent hashing where the people doing the lookups know the list of all storage nodes (tens to tens of thousands; not too bad) and directly determine who to contact

How Akamai Works

- Clients fetch html document from primary server
  - E.g. fetch index.html from cnn.com
- URLs for replicated content are replaced in html
  - E.g. `<img src="http://cnn.com/af/x.gif">` replaced with `<img src="http://a73.g.akamaitech.net/7/23/cnn.com/af/x.gif">`
- Client is forced to resolve aXYZ.g.akamaitech.net hostname
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 :)

A 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 \(\rightarrow\) hash(data)
  - hash(data) \(\rightarrow\) list of blocks
- For example:
  - `/src/foo.c` \(\rightarrow\) `0xfff32f2fa11d00f0`
  - `0xfff32f2fa11d00f0` \(\rightarrow\) `[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 (`0x2fab4f00f01...`).
- 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)

P2P-enabled Applications: Self-Certifying Names

- Name = Hash(pubkey, salt)
- Value = <pubkey, salt, data, signature>
  - can verify name related to pubkey and pubkey signed data
- Can receive data from caches, peers or other 3rd parties without worry
Hash functions

- Given a universe of possible objects U, map N objects from U to an M-bit hash.
- Typically, \(|U| \gg M\).
  - 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.

Desirable Properties

- 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.

Longevity

- “Computationally infeasible” means different things in 1970 and 2012.
  - 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>MD5</td>
<td>1990</td>
<td>1991</td>
<td>1995</td>
<td>1-5y</td>
<td>MD5</td>
</tr>
<tr>
<td>MD5-64</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>abandoned</td>
<td>2004</td>
<td></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>SHA-0</td>
<td>1992</td>
<td></td>
<td>2004</td>
<td>5-12y</td>
<td>SHA-1</td>
</tr>
<tr>
<td>SHA-1</td>
<td>1993</td>
<td>1998</td>
<td>2004</td>
<td>5-12y</td>
<td>SHA-1</td>
</tr>
<tr>
<td>SHA-2 (SHA-1)</td>
<td>1995</td>
<td>not quite yet</td>
<td>9+</td>
<td>SHA-3 &amp; 3</td>
<td></td>
</tr>
<tr>
<td>SHA-1 (SHA-2)</td>
<td>2001</td>
<td>still good</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA-2 (SHA-3)</td>
<td>2012</td>
<td></td>
<td></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?

Final pointer

- Hashing forms the basis for MACs - message authentication codes
  - Basically, a hash function with a secret key.
  - \(H(key, data)\) - can only create or verify the hash given the key.
  - Very, very useful building block
Summary

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

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