Hashing

- Today is a "building blocks" day (we've had two days of seeing our toolkit used in the real world, so back to basics)
- Two uses of hashing that are becoming wildly popular in distributed systems
  - Content-based naming
  - Consistent Hashing of various forms

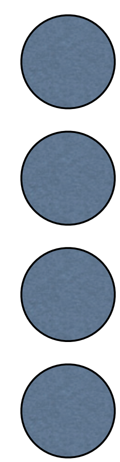
Example systems that use them

- BitTorrent & many other modern p2p systems use content-based naming
- Amazon, Linkedin, etc., all have built very large-scale key-value storage systems (databases--) using consistent hashing

Simple scenario

Suppose you're building a big web cache that holds copies of web pages your users have downloaded

Web page



Cache servers

How do you allocate pages/images to the cache servers?

(This is a realistic scenario)

Dividing items onto storage servers

- Option 1: Static partition (items a-c go there, d-f go there, ...)
  - Requires thinking. e.g., 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. (May be reasonable design for huge objects, as we saw in GFS!)
  -

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?
  - The bucket that every item is assigned to changes, pretty much.

Simple Hashing

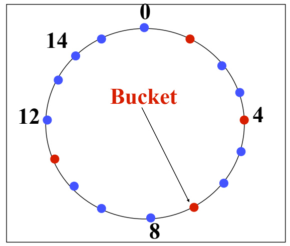
- Given document XYZ, we need to choose a server to use
- Suppose we use modulo
- Number servers from 1…n
  - Place document XYZ on server (XYZ mod n)
  - What happens when a servers fails? n → n-1
    - Same if different people have different measures of n
  - Why might this be bad?

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^n circle, where, hash key size = n.
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket ("successor")



- Smoothness → addition of bucket does not cause 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 popularized by a scalable lookup system called Chord
  - 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

Hashing 2: For naming

- Many filesystems 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 kinda more like names. :)

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. $$$. Can we do better? Yes, we can!

"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
- That'd look, in practice, like:
- /Users/dga/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!

A second example

- Several p2p systems operate something like:
- Search for "briney spars music", find a particular file name (badmusic.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)

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*.

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

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

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