CS5112: Algorithms and Data Structures for Applications

Lecture 7: Some distributed algorithms

Ramin Zabih

Some figures from Wikipedia/Google image search

Administrivia

- HW comments and minor corrections on Slack
  - Please keep an eye on it for announcements
- Q3 out today, coverage through Tuesday's lecture
- Non-anonymous survey coming re: speed of course, etc.
- Next week:
  - Tuesday lecture by Richard Bowen
  - Thursday lecture by Prof. Ari Juels
  - Thursday evening clinic by Richard Bowen

Office hours

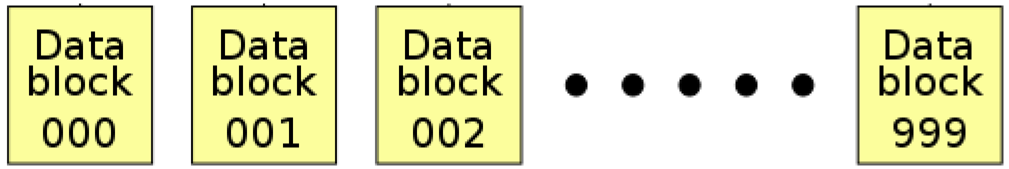
- Prof Zabih: after lecture or by appointment
- Tuesdays 11:30-12:30 in Bloomberg 277 with @Julia
- Wednesdays 2:30-3:30 in Bloomberg 277 with @irisz
- Wednesdays 3:30-4:30 in Bloomberg 277 with @Ishan
- Thursdays 10-12 in Bloomberg 267 with @Fei Li

Today

- Hash lists
- Merkle trees
- A few DHT (BitTorrent) issues
- Consistent hashing
- Perfect hashing

Motivation

- Definition of a distributed system
- Consider a large file like a video
- Blocks of the file are distributed for many reasons
  – Redundancy, cost, etc.
  – Different processors have different blocks



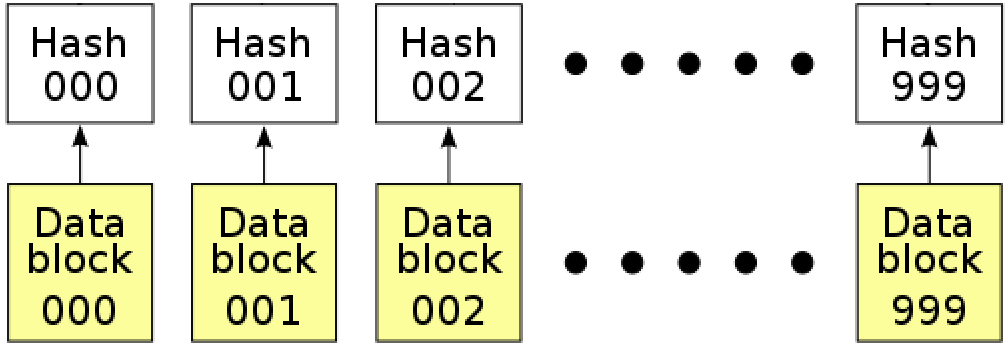
CORNELL TECH

How do we insure integrity?

- For a file on a single machine, we can use a checksum
  - Function which changes a lot if we change the input a little
  - Store the results along with the file
- Famous example: MD5, intended to be a cryptographic hash
  - You can find pairs such that $md5(x) = md5(y)$
  - “Flame” used this to forge a code-signing certificate for Windows
    - https://blogs.technet.microsoft.com/srd/2012/06/06/flame-malware-collision-attack-explained/
- But MD5 is fine as a checksum, and widely used

Hash lists and applications

- When the blocks are distributed we need:
  - Integrity of each block
  - Robustness to failure of computer holding a given block
- Solution: hash each data block



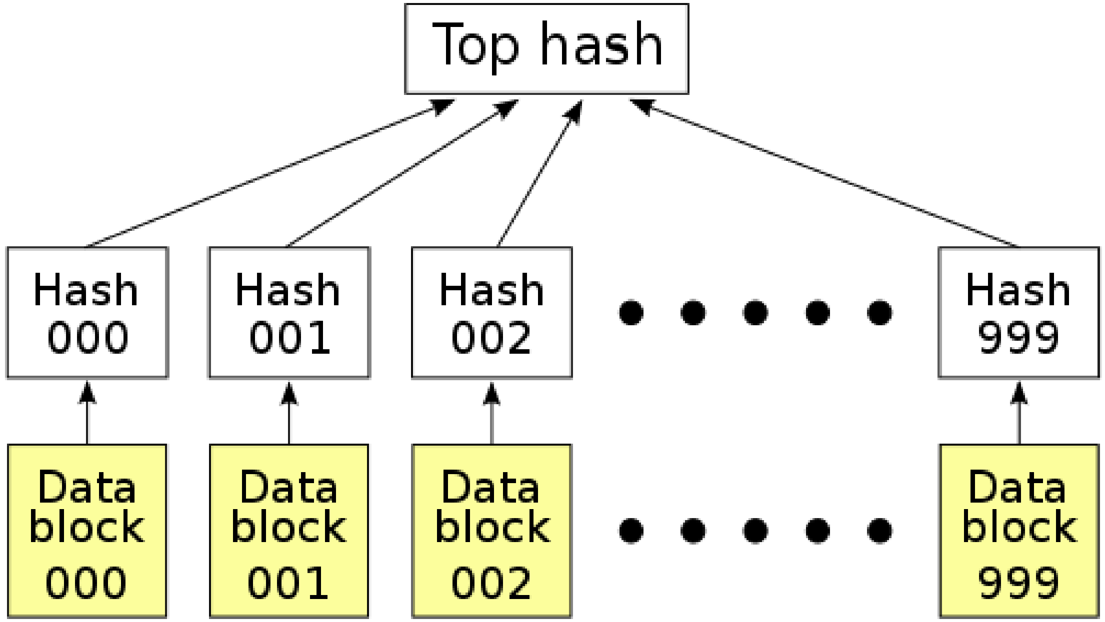
- Sounds cool, but is it useful?

Widely used application of hash lists

```
{
    'info':
    {
        'name': 'debian-503-amd64-CD-1.iso',
        'piece length': 262144,
        'length': 678301696,
        'pieces': <binary SHA1 hashes>
    }
    'info':
    {
        'name': 'directoryName',
        'piece length': 262144,
        'files':
        [
            {'path': ['111.txt'], 'length': 111},
            {'path': ['222.txt'], 'length': 222}
        ],
        'pieces': <binary SHA1 hashes>
    }
}
```

Hash trees/Merkle trees

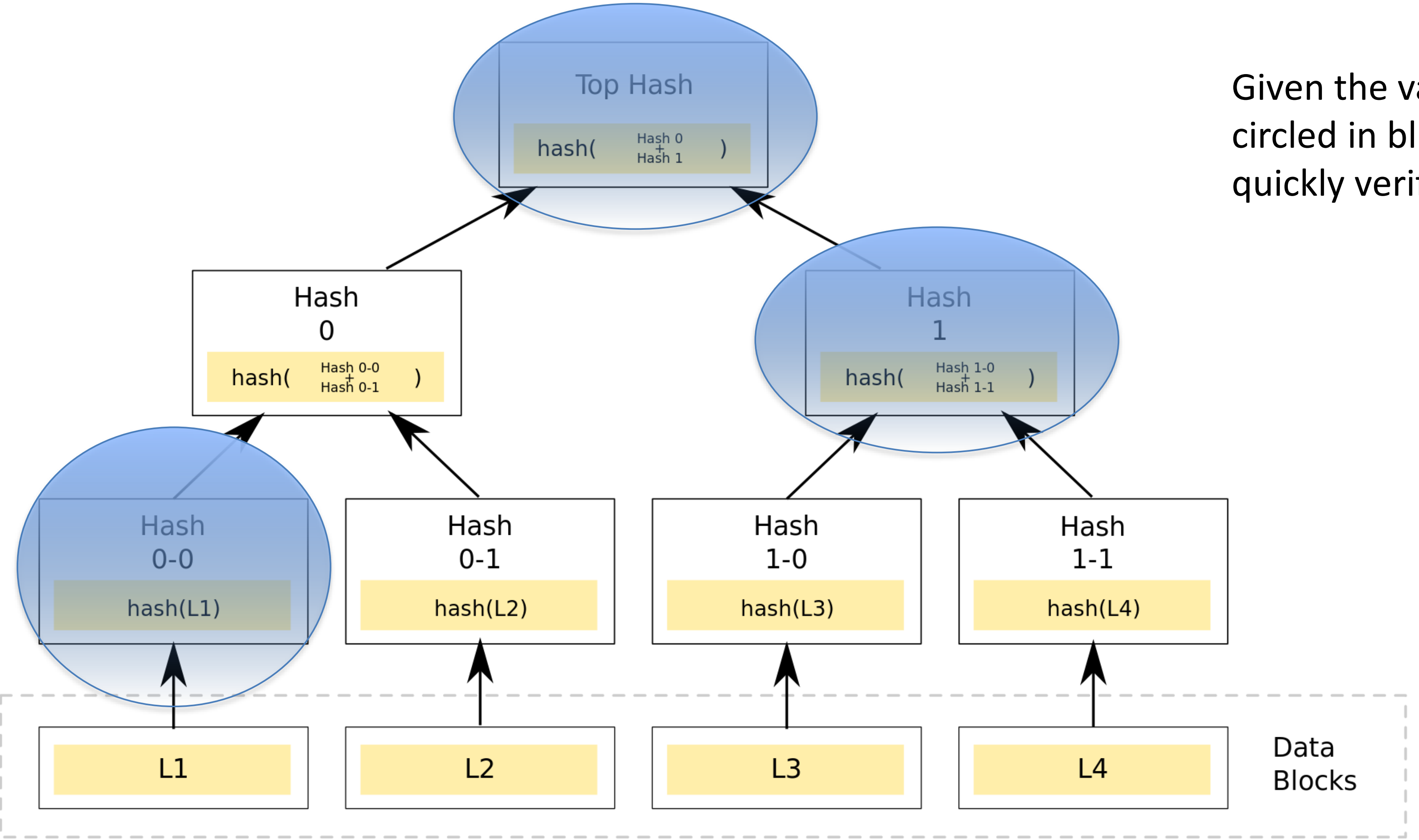
- Many applications has the block hashes together
  - Create a trusted “top hash” by hashing the concatenation



Hash trees/Merkle trees

- Widely used (Bitcoin, Git, etc)
- Tree of hash values
  - Usually a binary tree
- Leaf nodes identify data blocks
- The parent of two nodes with hash value x, y has the hash of the concatenation of x and y

Example



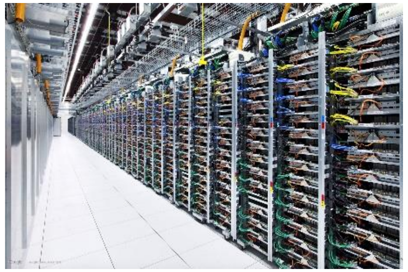
Given the values circled in blue we can quickly verify L2

Distributed hash tables (DHT)

- BitTorrent, etc.
- Given a file name and its data, store/retrieve it in a network
- Compute the hash of the file name
- This maps to a particular server, which holds the file
- Sounds good! Until the file you want is on a machine that is not responding…
  - But is this a real issue? Aren’t computers pretty reliable?

Google datacenter numbers (2008)

- *In each cluster's first year, it's typical that:*
  - *1,000 individual machine failures will occur;*
  - *thousands of hard drive failures will occur;*
  - *one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours;*
  - *20 racks will fail, each time causing 40 to 80 machines to vanish from the network;*
  - *5 racks will "go wonky," with half their network packets missing in action;*
  - *The cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span.*
  - *About a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.*
- Jeff Dean, "Google spotlights data center inner workings", CNET May 2008



From filename to processor

- Typically the result of a hash function is a large number
  - SHA-1 produces 160 bits (not secure!)
- Map into servers with modular arithmetic
  - Reminder: 4 + 7 = 1 (mod 10)
  - Mod with powers of 2 is just the low-order bits
  - Sneak preview: next lecture will be on bits
- How do we handle a server crashing or rejoining??

Consistent hashing

- Effectively the hash table itself is resized
  - Note that this is an important operation in general!
- With naïve hash functions, resizing is a disaster
  - Everything needs to be shuffled between buckets/servers
- Key idea is to give add **state**
  - Traditional hash functions are stateless/functional

CORNELL TECH

Hashing into the circle

- Let’s convert the output of our hash function into a circle
  - For example, using the low-order 8 bits of SHA-1
- We map both servers and data onto the circle
  - For a server, hash of IP address or something similar
- Data is stored in the “next” server on the circle
  - By convention we will move clockwise

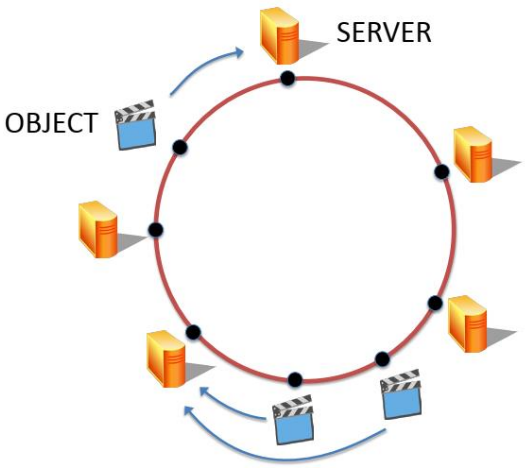


Figure from Maggs, Bruce M.; Sitaraman, Ramesh K. (July 2015), "Algorithmic nuggets in content delivery" (PDF), *SIGCOMM Computer Communication Review*, New York, NY, USA,**45** (3): 52–66

CORNELL TECH

Example of consistent hashing

- Data 1,2,3,4 stored on computers A,B
- Servers->data (good quiz/exam question):

  A->1,4

  B->2

  C->3
- If C crashes, we just move 3 to A

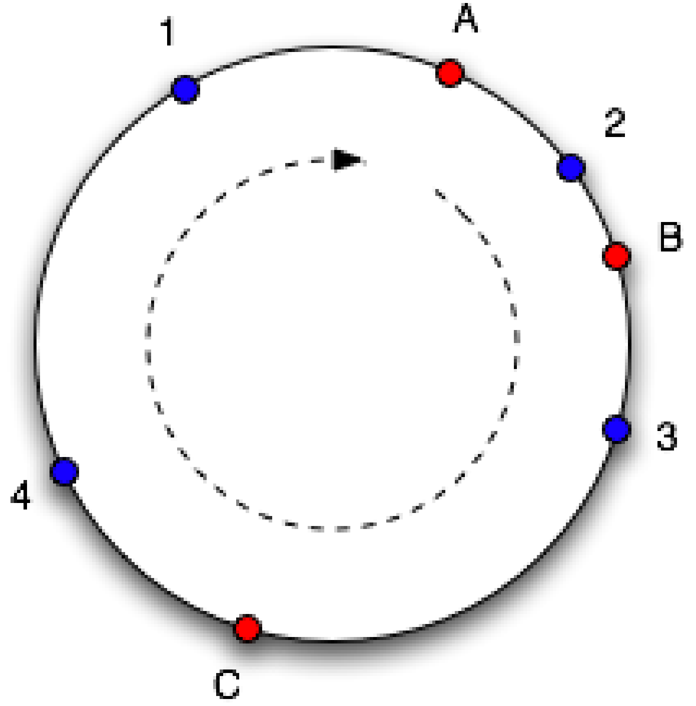


Diagram taken from Tom White based on original article

Gracefully adding/removing a server

- Add server D after C crashes
  - Takes 3,4 from A
- Servers->data:

  A->1

  B->2

  D->3,4
- This is a lot faster!
  - Naively, going from 3 to 4 servers moves 75% of data
  - With consistent hashing we move 25% of data
  - Advantage gets even larger for more servers

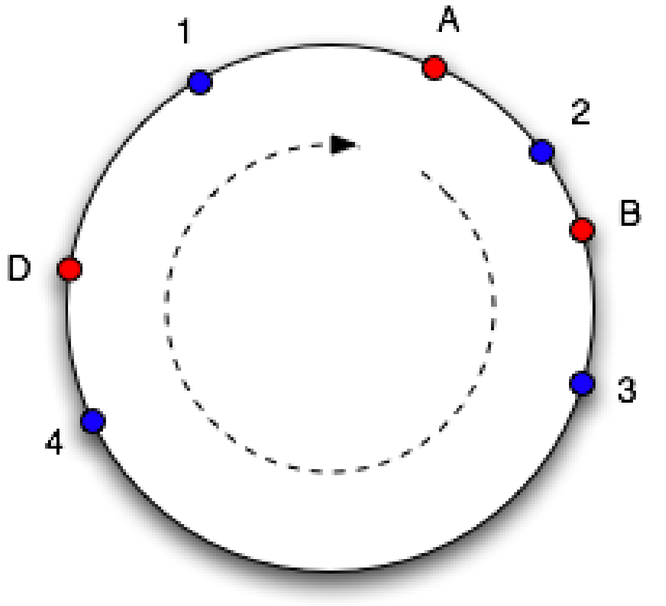


Diagram taken from Tom White based on original article

CORNELL TECH

Improving consistent hashing

- Need a uniform hash function, lots of them aren't
- Typically make replicas of servers for load balancing
  - About $\log m$ replicas from m servers for theoretical reasons
  - Can also replicate data items if they are popular
- Typically store a list of nearby nodes for redundancy
- Note that the data still needs to move after a crash
- Store the servers in a BST to efficiently find successor
  - This requires global knowledge about the servers

Handling popular objects

- Each object can have its own hash function
- Basically, it's view of the unit circle
- Ensures that you are very unlikely to have 2 popular objects share the same server

Perfect & minimal hashing

- Choice of hash functions is data-dependent!
- Let's try to hash 4 English words into the buckets 0,1,2,3
  - E.g., to efficiently compress a sentence
- Words: {"banana", "glib", "epic", "food"}
  - Can efficiently say sentence like "epic glib banana food" = 3,2,1,0
- Can you construct a minimal perfect hash function that maps each of these to a different bucket?
  - Needs to be efficient, not (e.g.) a list of cases

Perfect hashing example

- For this particular example, it is easy

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

CORNELL TECH

Recall: bitwise masking

- Bitwise AND operation:
  - AND(1,1) = 1
  - AND(0,1) = AND(1,0) = AND(0,0) = 0
- Note that AND(x,0) = 0 and AND(x,1) = x
- An AND with a binary number (mask) zeros out the bits where the mask is 0
  - Lets through the bits where the mask is 1
- So our perfect hash function is: AND with 3 = 0b11