Distributed Computing

A-10. Search in P2P Systems

Peer-to-Peer (P2P)

- Distributed applications where nodes (**peers**) play "equal roles"
  - Self-organized and adaptive
  - Distributed and decentralized, hence fault-tolerant
    - An app owned by a single organization may be shut down
  - Censorship-resistant (as we've seen with Tor)
  - Uses resources that would be wasted otherwise

P2P Applications (and examples)

- Killer applications:
  - 1999: **file sharing** (Napster)
  - 2008: **cryptocurrencies** (Bitcoin)
  - 2013: **smart contracts**, i.e., “world computer” (Ethereum)
- Other uses:
  - Decentralized chat & audio calling (Skype)
  - Audio streaming (Spotify)
  - Censorship-resistant & private systems (Tor)
  - Cheap file/multimedia stream distribution & tolerance to flash-crowds (BitTorrent)
  - Decentralized private architectures in datacenters (Amazon Dynamo)
  - P2P storage & backup (IPFS)

Searching

- In a system with potentially millions of peers, how to find a given piece of content?
- It’s a non-trivial problem, which balances decentralization with performance

Napster: Centralized P2P

1999: Napster

- File-sharing used mostly for MP3s
- A central index server, to which users uploaded information about their songs
- Solved scalability issues for bandwidth
  - **Core** internet bandwidth was a bigger problem back then
- Legally tricky: uploading copyrighted content was illegal, but what about just **telling where it was**?
- Closed down in 2001 after reaching **26.4M users**

Unstructured P2P

Getting Decentralized

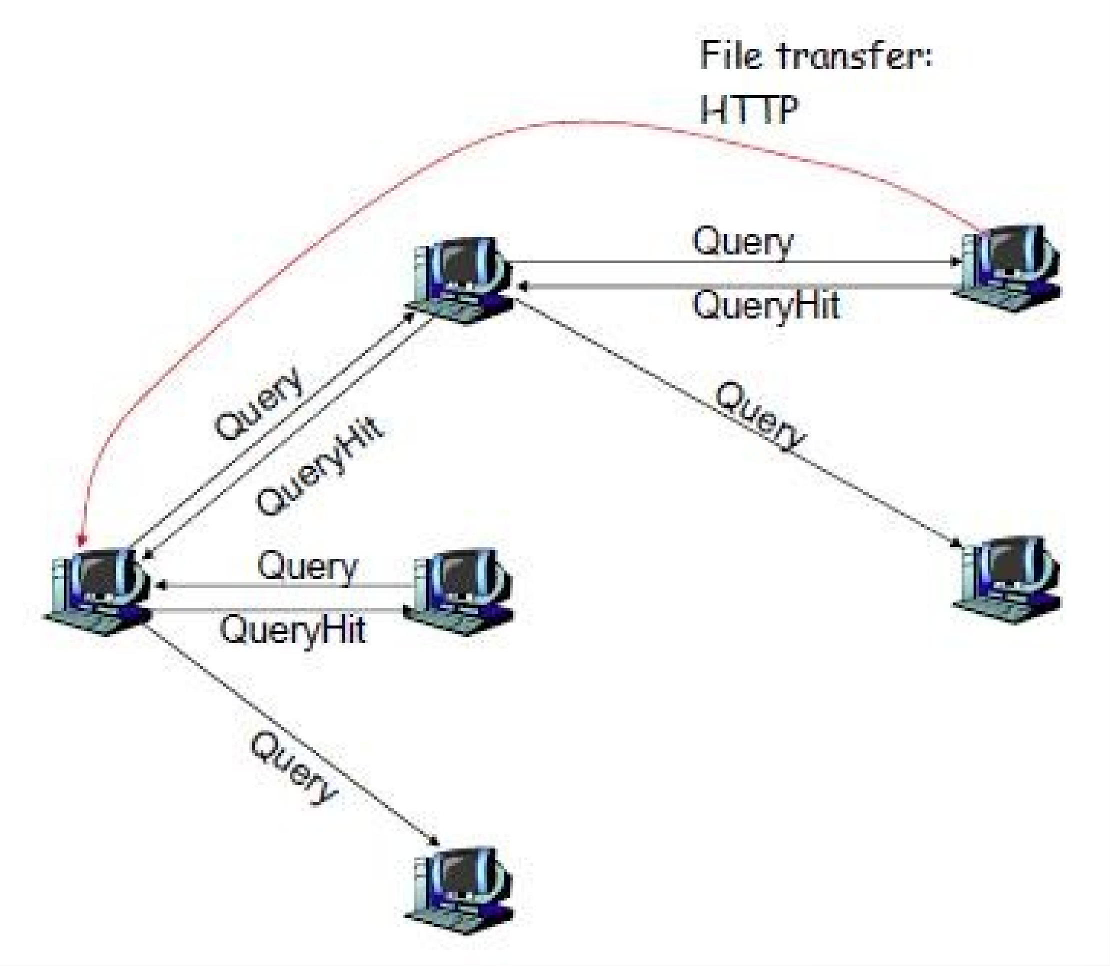
- The centralization of Napster arguably caused its demise:
  - There was a central server to shut down
  - There was a company to sue
- Work started on completely decentralized applications
- Idea: create **overlay networks**, i.e., networks **on top of other networks**
  - i.e., the P2P network is over the TCP/IP one

2000: Gnutella

- The first decentralized P2P file-sharing network
- **Overlay network** where each node is connected to a few others (5 by default)
- **Bootstrap**: each node contacts some services (“caches”) to get some nodes to connect with
- If a node is “full” with connections, it will forward the connection request to its neighbors
- New nodes discovered will be saved for the next sessions

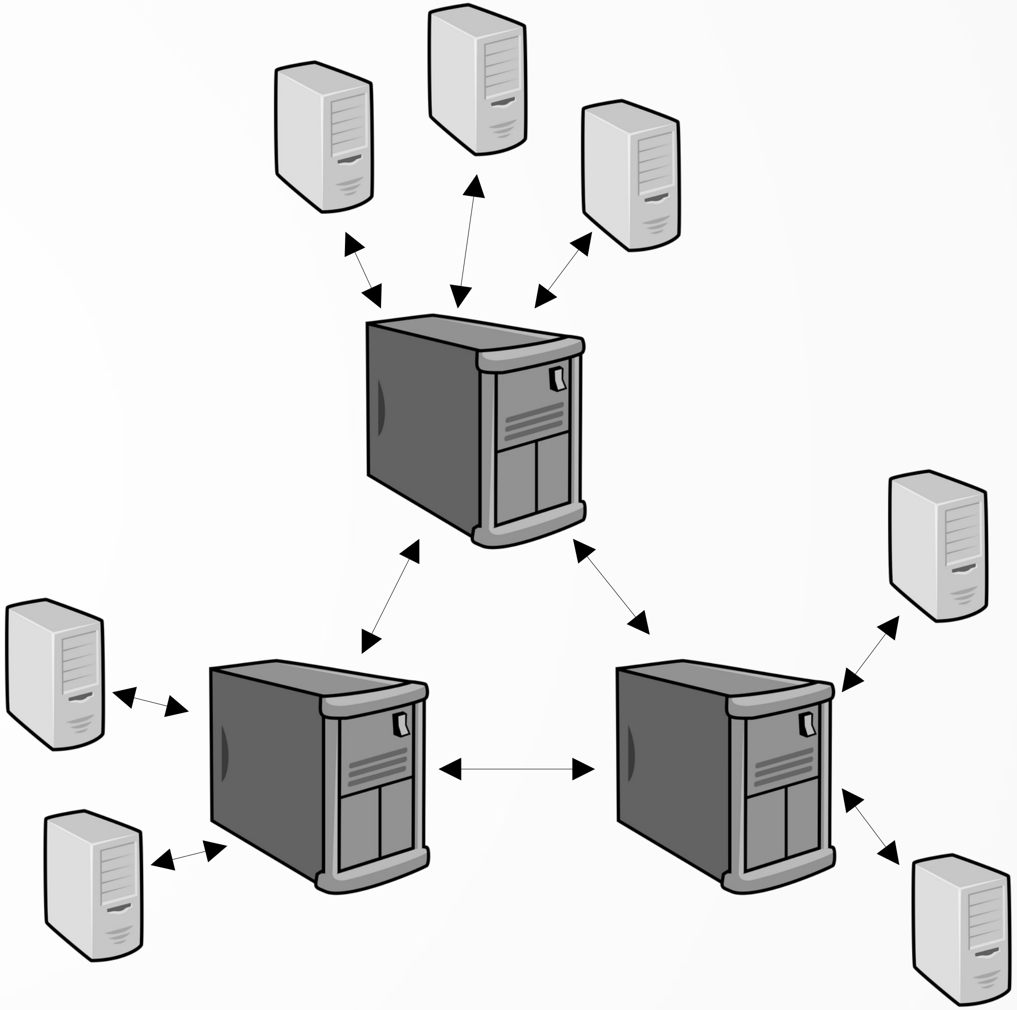
Flooding: Searching Everybody

- New query: set a time-to-live (TTL, max 7) and forward it to all neighbors
  - Each of them will decrease the TTL by 1 and forward the query to its neighbors
- Many duplicate queries and very slow, but working to some extent
- However, routing messages could overload machines—especially the weakest ones
- Lots of redundant messages



2002: Gnutella 0.6 & Ultrapeers

- There was a big difference between dial-up and ADSL nodes—dial-up ones were often overwhelmed just by sending queries around
- Separation between **leaf nodes** and **ultrapeers**
- Nodes with bandwidth and stability could get upgrated to **ultrapeers**
- Leaf nodes connect to 3 ultrapeers
- Ultrapeers connect to 32+ other ultrapeers
- “Superpeers” were a good design pattern used by several other apps (e.g., Skype)



Searching with Ultrapeers

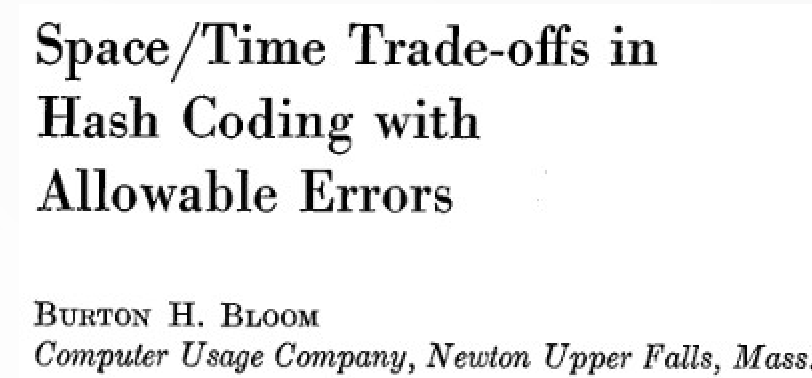
- Each node sends to its ultrapeers a “query routing table” (QRT): a representation of the set of files they have
- Ultrapeers aggregate their QRTs and those of all their leaves, and sends the results to all their neighbors
- Queries get sent to a peer only if they have a hope of having the requested file
- These modifications greatly improved scalability
- With higher degree, TTL was lowered to 4

Bloom Filters

What do QRTs do?

- QRTs should represent a set of keywords
- If my query is “foo bar” I want to get an answers for files that match with **both** keywords
- Maybe you have files that match “foo baz” and “bar qux”--in that case it’s ok to get a false positive: you match both keywords, but don’t have a single file that matches them
- We can have false positives, but no false negatives
- We want QRTs to be **as small as possible**

Bloom Filters


Space/Time Trade-offs in
Hash Coding with
Allowable Errors

BURTON H. BLOOM
Computer Usage Company, Newton Upper Falls, Mass.

- A data structure invented in 1970
- It represents sets, giving two methods:
  - **add**(*x*): add an element to the set
  - **test**(*x*): tell me if *x* is in the set
    - There’s a possibility for **false positives**
- No way of retrieving the original elements
- On the other hand, **very compact**: use very little space
- Great for our use case!

How Bloom Filters Work

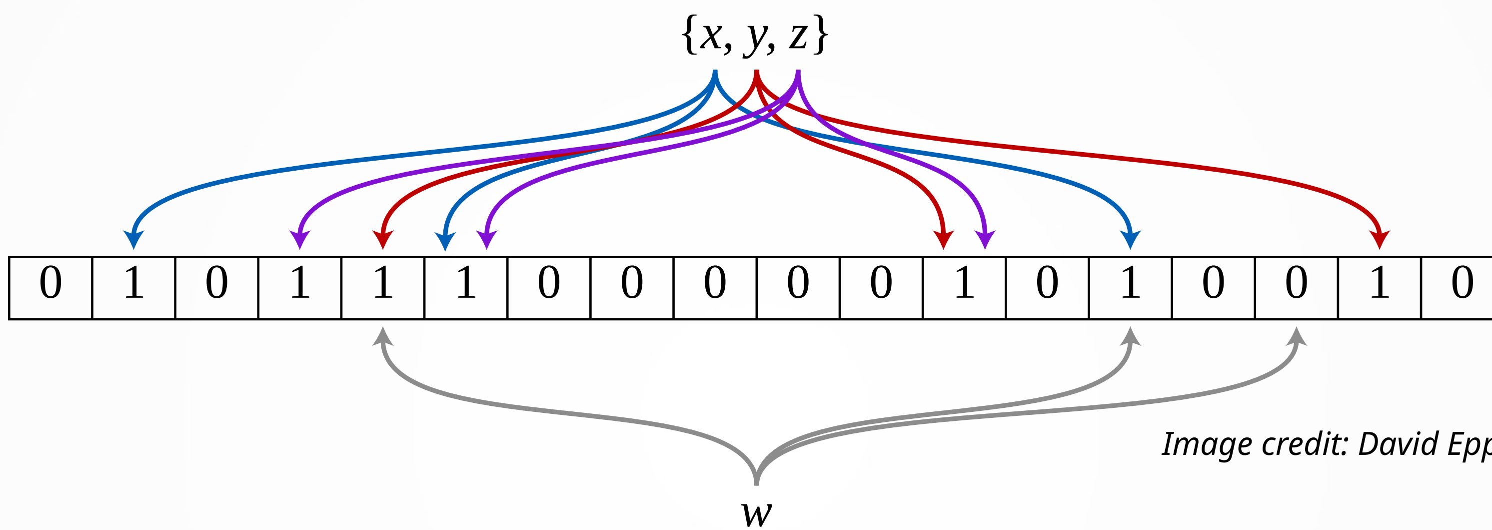
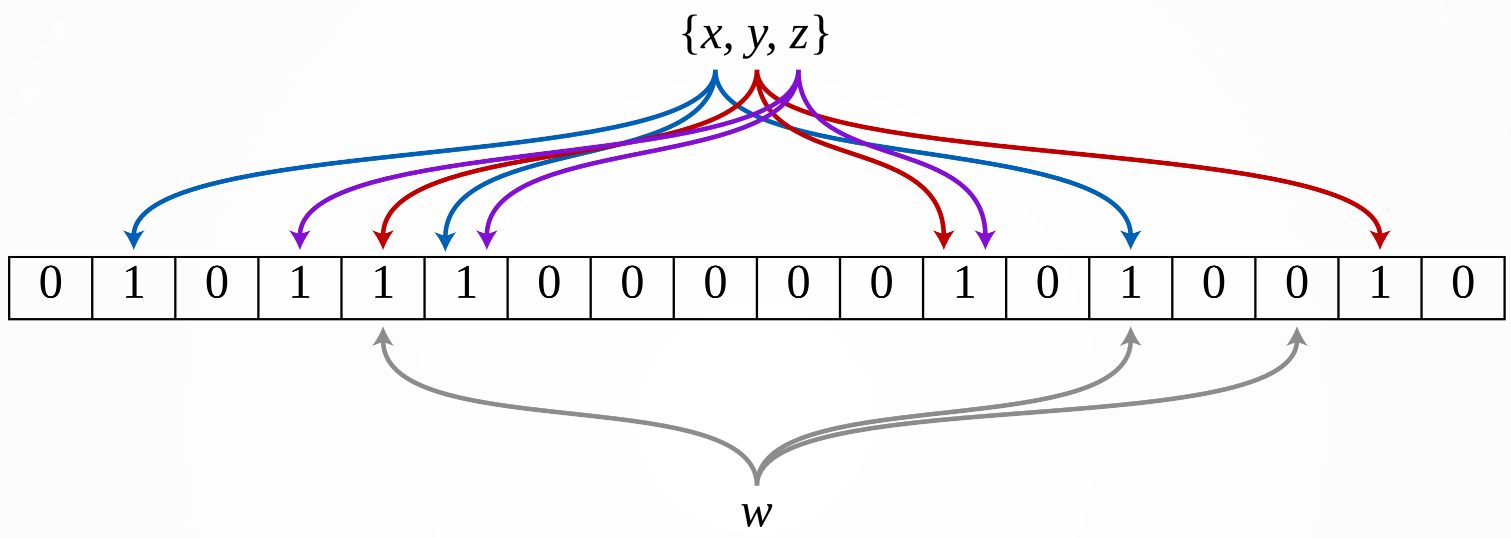


Image credit: David Eppstein

- A list of *m* bits
- *k* hash functions mapping elements to value in *[0, m-1]*
- **Add** sets to 1 the corresponding *k* bits
- **Test** verifies if the *k* bits are all set to 1

Bloom Filters: Example



- In this case, we created a filter representing elements *{x,y,z}*
  - Set to 1 the corresponding bits
- If we test for any of them, we'll see that all corresponding bits are 1s, so we get a **yes**
- When we test for *w*, we get 2 collisions but one bit is 0
  - We're certain *w* is not there

Applications

- Why do we want a data structures that gives us **worse functionality** than, e.g., a hashtable?
  - It uses **less memory**
- In databases: keeping a “cache” in RAM before accessing a disk
  - If we **know** an item isn’t on disk, we spare a disk access
- Web caches: avoid storing data requested only once
  - Only cache things at the second time they’re asked

Bloom Filters: Demo and Analysis

- Bloom filter demo: https://llimllib.github.io/bloomfilter-tutorial/
- See the other presentation
  - (if you ever tried formulas in LibreOffice, you know why)
- Cuckoo filters: http://bdupras.github.io/filter-tutorial/

Distributed Hash Tables

Better than Superpeers

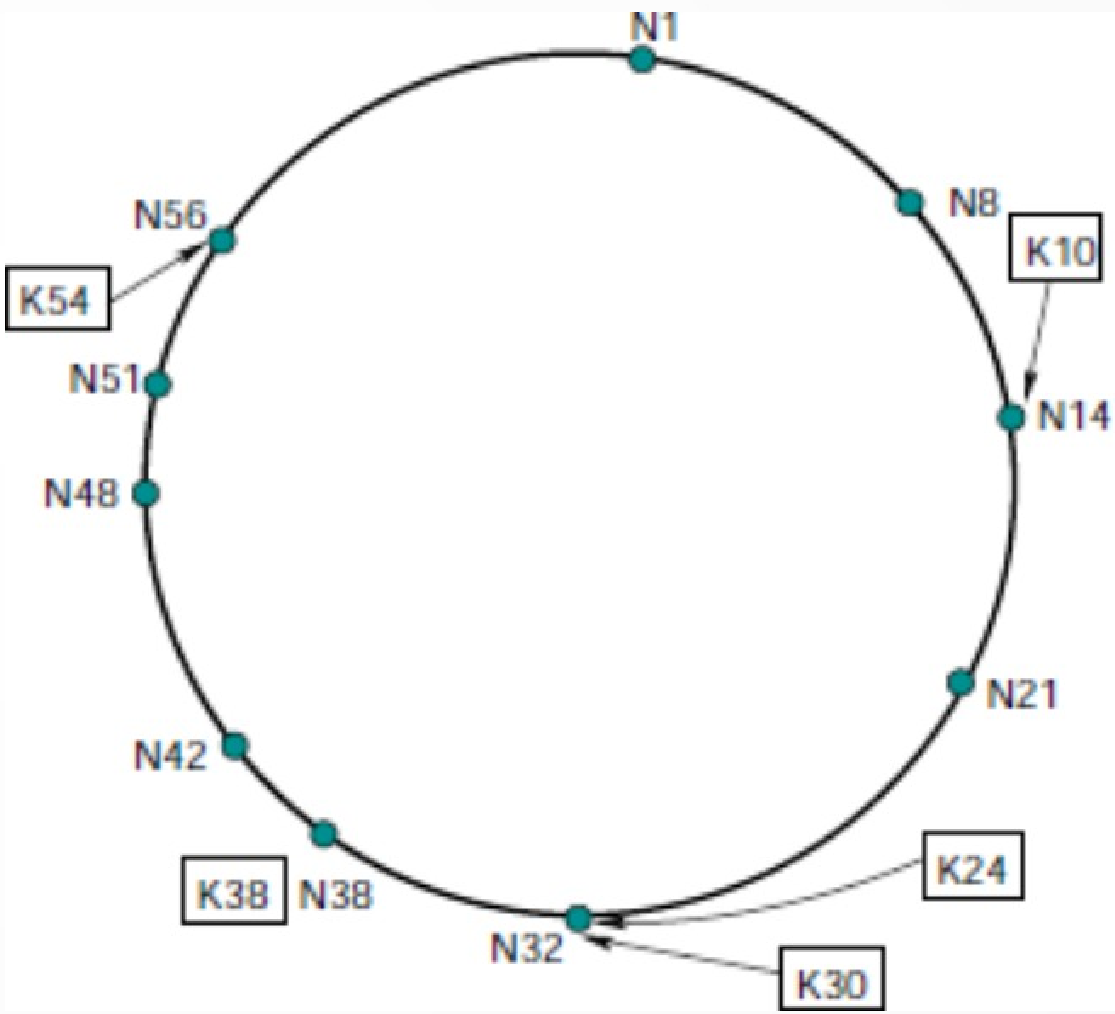
- **Distributed Hash Table (DHT)**: a decentralized system giving **efficient key-value lookups**
- Key idea: a **structured** peer-to-peer overlay
  - We choose **who connects to whom**
  - We use that freedom to **obtain efficient routing**!

How DHTs Work

- Every peer handles **a portion** of the hash table
  - For redundancy, more than one peer per portion actually
- **Consistent Hashing**: adding or removing peers has a **small impact** on resource allocation (i.e., which data a peer stores)
  - Perfect to handle **churn**: nodes arriving and leaving all the time
- Item *x* will be stored on node corresponding to address *h(x)*

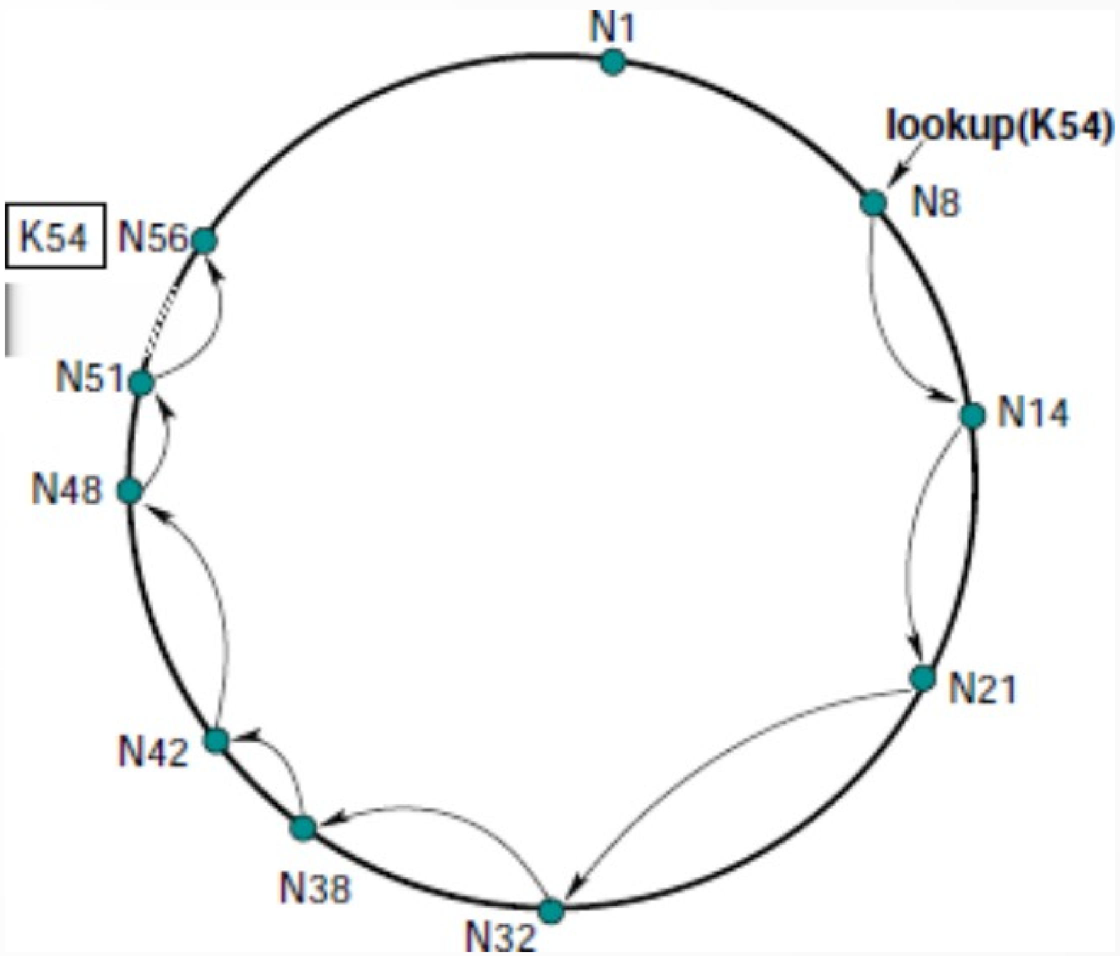
The Chord Ring

- Reference: paper by Stoica et al. (ACM SIGCOMM ‘01)
- Nodes take random identifiers, and get in the ring with a link to predecessor and followers
  - In the example: identifiers in [0, 63]
- Item *x* get inserted at the first peer with hash greater than *h(x)*
  - ...and a few (e.g., 2) predecessors, for redundancy



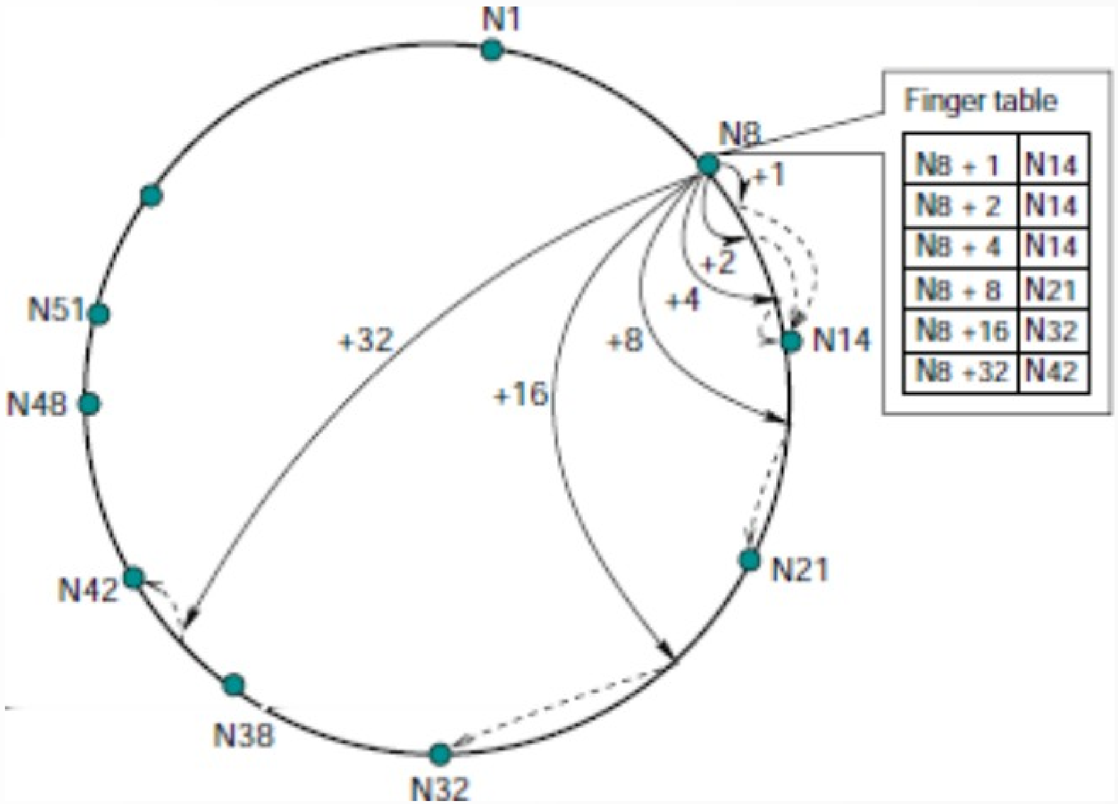
Chord: Lookup, the Slow Version

- You can get to the node responsible for a given key by following successor link until you get to the destination
- Very slow, and breaks if any node in the middle disappears
- *O(n)* hops



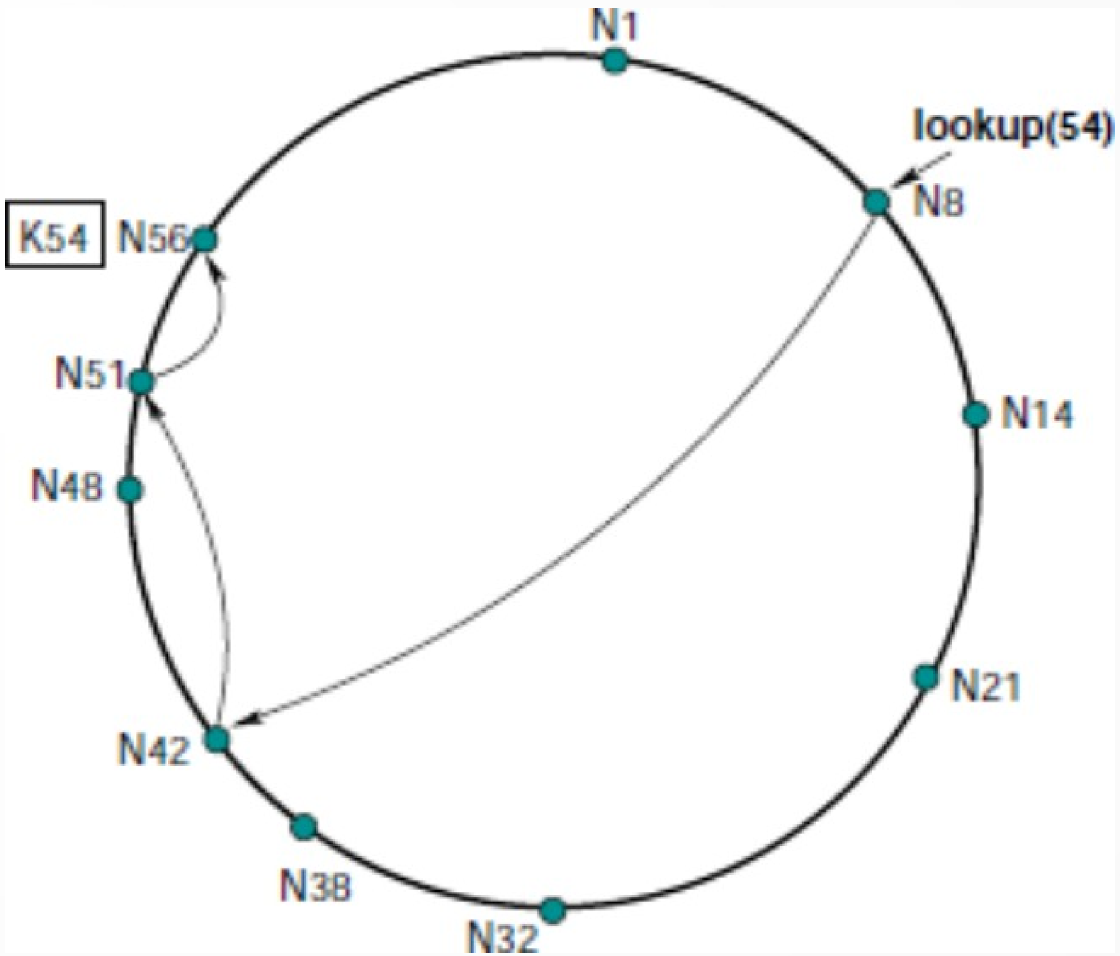
Chord: Accelerated Lookup

- To get to destination faster, nodes save **shortcuts** ("*fingers*") to destinations at exponential distances
- In the example, powers of 2
- Node at position *x* can get finger at position *x*+*y* by looking it up on the network

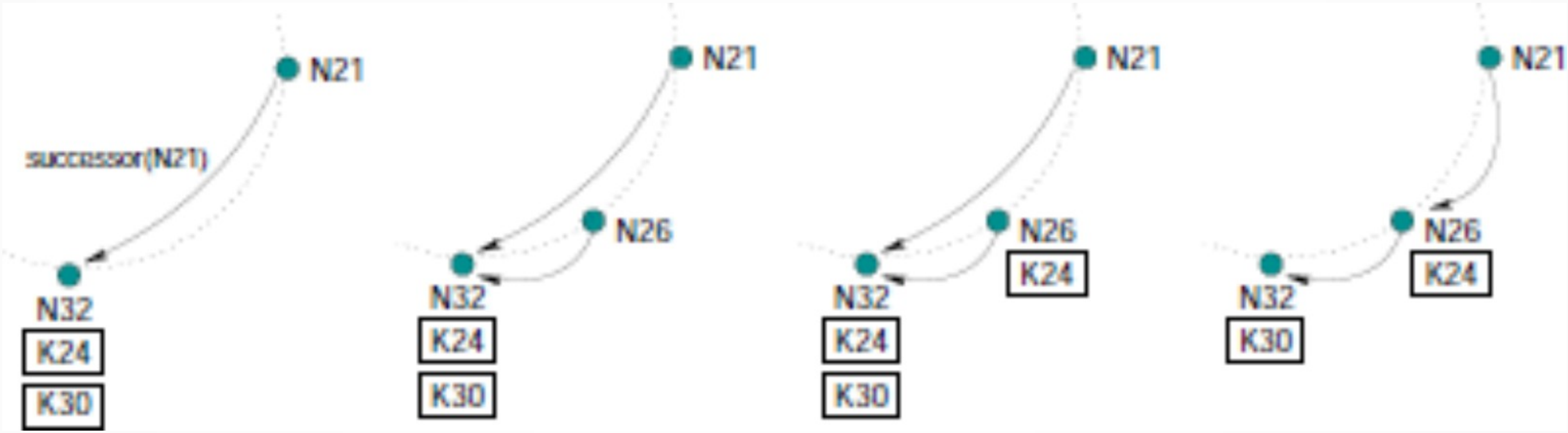


Chord: Greedy Routing

- If at each step we follow the finger closest—but before—the destination, our lookup is **much faster**
- **How many steps?**
- *O(log n)*



Chord: New Nodes



- Adding a new node just requires
  - A lookup
  - Exchanging data with two other nodes

Chord: Fault Tolerance

- To avoid breaking the successor chain when peers leave the system, nodes keep a list of *r* successors
  - If all of them leave the system, the ring can be broken
- A periodic **stabilization** procedure maintains the links
  - Predecessors and successors get pinged, fingers are re-queried

Chord: Scalability

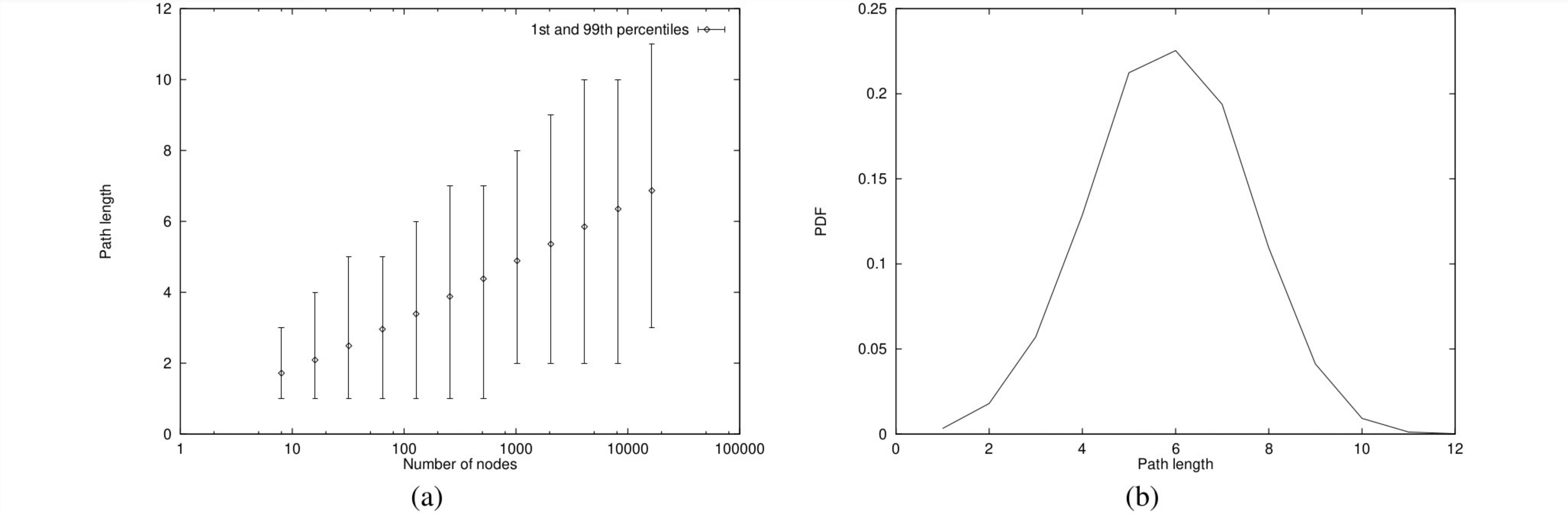


Figure 10: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

- Right: $n=2^{12}$=4,096; average path length=$\log_2(n)/2$=6
  - **Why**?

Kademlia

- Reference: paper
- The most used DHT in practice (e.g., BitTorrent)
- Very similar idea: logarithmic number of steps to get to destination
- Here, you get a finger table for nodes that have the first 0, 1, …, k bit in common with you
- Added value: links are symmetrical, so you can exploit information about them when they reach you