15-440 Distributed Systems

Lecture 21 – CDN & Peer-to-Peer

Last Lecture: 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?

Outline

- Content Distribution Networks
- P2P Lookup Overview
- Centralized/Flooded Lookups
- Routed Lookups – Chord

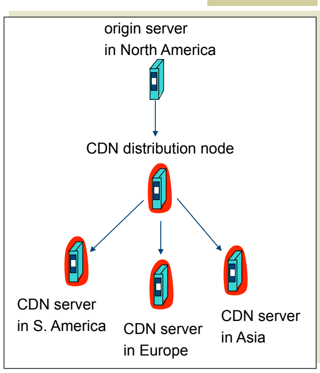
Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes are heavy-tailed
- Embedded references
- This plays havoc with performance. Why?
- Solutions?

•Lots of small objects & TCP
  •3-way handshake
  •Lots of slow starts
  •Extra connection state

Content Distribution Networks (CDNs)

- The content providers are the CDN customers.
- Content replication
- CDN company installs hundreds of CDN servers throughout Internet
  - Close to users
- CDN replicates its customers' content in CDN servers. When provider updates content, CDN updates servers



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

Note: Nice presentation on Akamai at www.cs.odu.edu/~mukka/cs775s07/Presentations/mklein.pdf

How Akamai Works

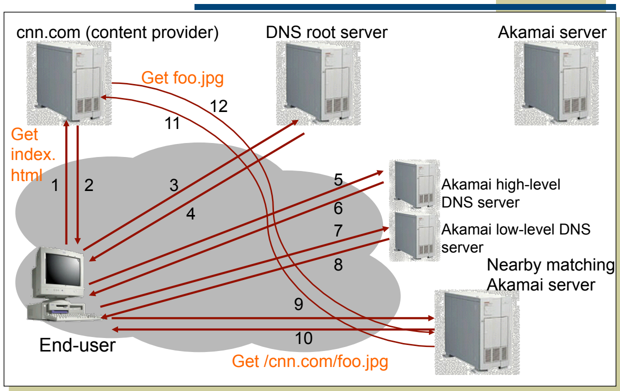
- How is content replicated?
- Akamai only replicates static content (*)
- Modified name contains original file name
- Akamai server is asked for content
  - First checks local cache
  - If not in cache, requests file from primary server and caches file

* (At least, the version we're talking about today. Akamai actually lets sites write code that can run on Akamai's servers, but that's a pretty different beast)

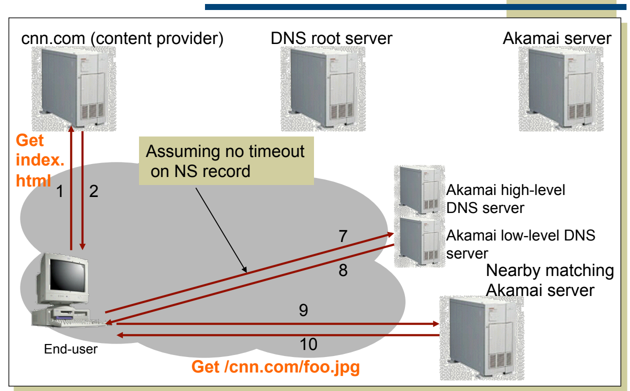
How Akamai Works

- Root server gives NS record for akamai.net
- Akamai.net name server returns NS record for g.akamaitech.net
  - Name server chosen to be in region of client's name server
  - TTL is large
- G.akamaitech.net nameserver chooses server in region
  - Should try to chose server that has file in cache - How to choose?
  - Uses aXYZ name and hash
  - TTL is small → why?

How Akamai Works



Akamai – Subsequent Requests



Simple Hashing

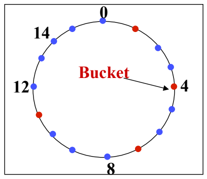
- Given document XYZ, we need to choose a server to use
- Suppose we use modulo
- Number servers from 1…n
  - Place document XYZ on server (XYZ mod n)
  - What happens when a servers fails? n → n-1
    - Same if different people have different measures of n
  - Why might this be bad?

Consistent Hash

- "view" = subset of all hash buckets that are visible
- Desired features
  - 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 unit interval
  - Hash of object = closest bucket



- Monotone → 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

Consistent Hashing not just for CDN

- Finding a nearby server for an object in a CDN uses centralized knowledge.
- Consistent hashing can also be used in a distributed setting
- P2P systems like BitTorrent need a way of finding files.
- Consistent Hashing to the rescue.

Summary

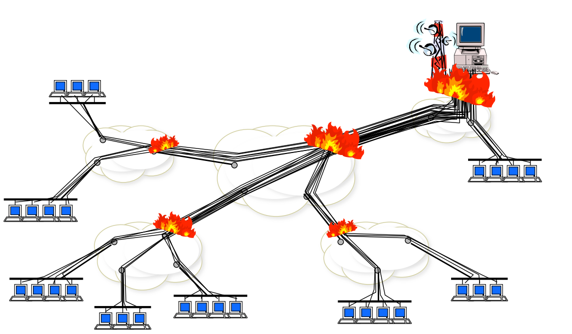
- Content Delivery Networks move data closer to user, maintain consistency, balance load
- Consistent hashing maps keys AND buckets into the same space
- Consistent hashing can be fully distributed, useful in P2P systems using structured overlays

Outline

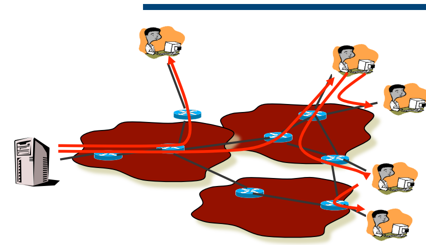
- Content Distribution Networks
- P2P Lookup Overview
- Centralized/Flooded Lookups
- Routed Lookups – Chord

Scaling Problem

- Millions of clients ⇒ server and network meltdown



P2P System

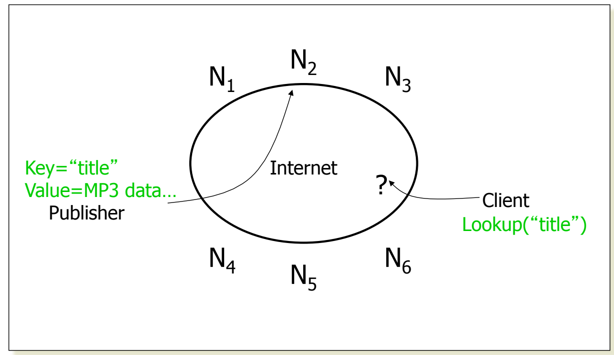


- Leverage the resources of client machines (peers)
  - Computation, storage, bandwidth

Peer-to-Peer Networks

- Typically each member stores/provides access to content
- Basically a replication system for files
  - Always a tradeoff between possible location of files and searching difficulty
  - Peer-to-peer allow files to be anywhere → searching is the challenge
  - Dynamic member list makes it more difficult
- What other systems have similar goals?
  - Routing, DNS

The Lookup Problem



Searching

- Needles vs. Haystacks
  - Searching for top 40, or an obscure punk track from 1981 that nobody's heard of?
- Search expressiveness
  - Whole word? Regular expressions? File names? Attributes? Whole-text search?
    - (e.g., p2p gnutella or p2p google?)

Framework

- Common Primitives:
  - **Join**: how to I begin participating?
  - **Publish**: how do I advertise my file?
  - **Search**: how to I find a file?
  - **Fetch**: how to I retrieve a file?

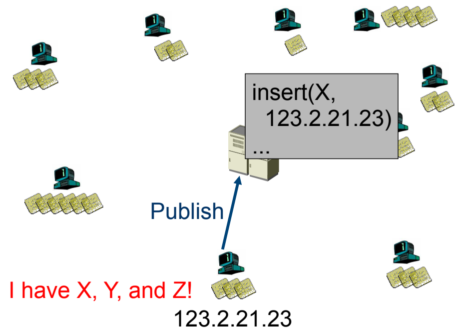
Outline

- Content Distribution Networks
- P2P Lookup Overview
- Centralized/Flooded Lookups
- Routed Lookups – Chord

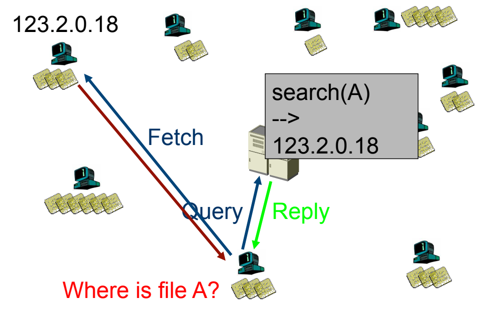
Napster: Overiew

- Centralized Database:
  - **Join:** on startup, client contacts central server
  - **Publish:** reports list of files to central server
  - **Search:** query the server => return someone that stores the requested file
  - **Fetch:** get the file directly from peer

Napster: Publish



Napster: Search



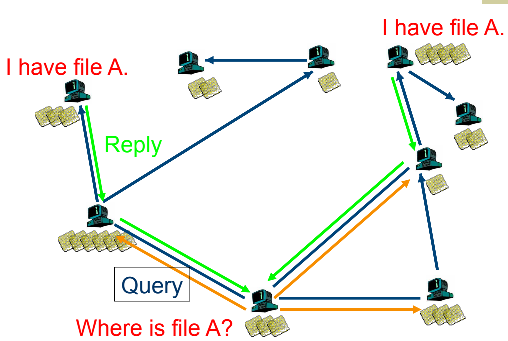
Napster: Discussion

- Pros:
  - Simple
  - Search scope is O(1)
  - Controllable (pro or con?)
- Cons:
  - Server maintains O(N) State
  - Server does all processing
  - Single point of failure

“Old” Gnutella: Overview

- Query Flooding:
  - **Join**: on startup, client contacts a few other nodes; these become its “neighbors”
  - **Publish**: no need
  - **Search**: ask neighbors, who ask their neighbors, and so on... when/if found, reply to sender.
    - TTL limits propagation
  - **Fetch**: get the file directly from peer

Gnutella: Search



Gnutella: Discussion

- Pros:
  - Fully de-centralized
  - Search cost distributed
  - Processing @ each node permits powerful search semantics
- Cons:
  - Search scope is O(*N*)
  - Search time is O(???)
  - Nodes leave often, network unstable
- TTL-limited search works well for haystacks.
  - For scalability, does NOT search every node. May have to re-issue query later

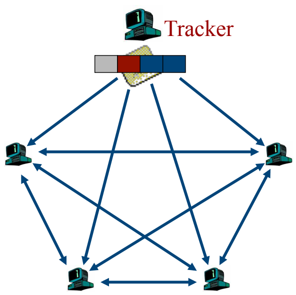
Flooding: Gnutella, Kazaa

- Modifies the Gnutella protocol into two-level hierarchy
  - Hybrid of Gnutella and Napster
- Supernodes
  - Nodes that have better connection to Internet
  - Act as temporary indexing servers for other nodes
  - Help improve the stability of the network
- Standard nodes
  - Connect to supernodes and report list of files
  - Allows slower nodes to participate
- Search
  - Broadcast (Gnutella-style) search across supernodes
- Disadvantages
  - Kept a centralized registration → allowed for law suits ☹

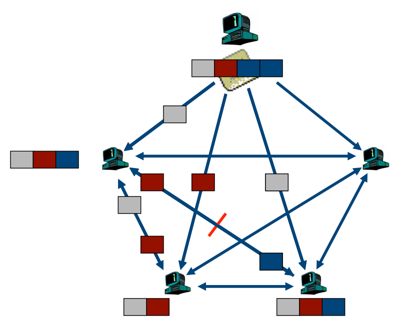
BitTorrent: Overview

- Swarming:
  - **Join**: contact centralized “tracker” server, get a list of peers.
  - **Publish**: Run a tracker server.
  - **Search**: Out-of-band. E.g., use Google to find a tracker for the file you want.
  - **Fetch**: Download chunks of the file from your peers. Upload chunks you have to them.
- Big differences from Napster:
  - Chunk based downloading
  - “few large files” focus
  - Anti-freeloading mechanisms

BitTorrent: Publish/Join



BitTorrent: Fetch



BitTorrent: Sharing Strategy

- Employ “Tit-for-tat” sharing strategy
  - A is downloading from some other people
    - A will let the fastest N of those download from him
  - Be optimistic: occasionally let freeloaders download
    - Otherwise no one would ever start!
    - Also allows you to discover better peers to download from when they reciprocate
- Goal: Pareto Efficiency
  - Game Theory: “No change can make anyone better off without making others worse off”
  - Does it work? (not perfectly, but perhaps good enough?)

BitTorrent: Summary

- Pros:
  - Works reasonably well in practice
  - Gives peers incentive to share resources; avoids freeloaders
- Cons:
  - Pareto Efficiency relative weak condition
  - Central tracker server needed to bootstrap swarm
    - Alternate tracker designs exist (e.g. DHT based)

Outline

- Content Distribution Networks
- P2P Lookup Overview
- Centralized/Flooded Lookups
- Routed Lookups – Chord

DHT: Overview (1)

- Goal: make sure that an item (file) identified is always found in a reasonable # of steps
- Abstraction: a distributed hash-table (DHT) data structure
  - insert(id, item);
  - item = query(id);
  - Note: item can be anything: a data object, document, file, pointer to a file…
- Implementation: nodes in system form a distributed data structure
  - Can be Ring, Tree, Hypercube, Skip List, Butterfly Network, ...

DHT: Overview (2)

- Structured Overlay Routing:
  - **Join**: On startup, contact a "bootstrap" node and integrate yourself into the distributed data structure; get a *node id*
  - **Publish**: Route publication for *file id* toward a close *node id* along the data structure
  - **Search**: Route a query for file id toward a close node id. Data structure guarantees that query will meet the publication.
  - **Fetch**: Two options:
    - Publication contains actual file => fetch from where query stops
    - Publication says "I have file X" => query tells you 128.2.1.3 has X, use IP routing to get X from 128.2.1.3

DHT: Example - Chord

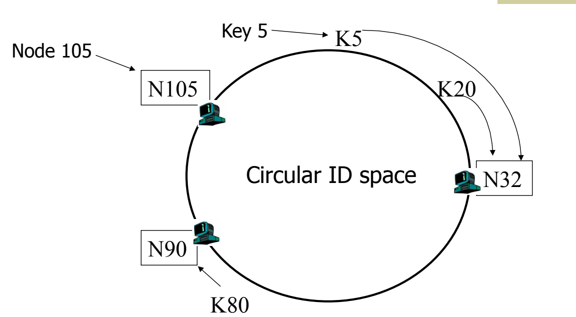
- Associate to each node and file a unique *id* in an *uni*-dimensional space (a Ring)
  - E.g., pick from the range [0...2^m]
  - Usually the hash of the file or IP address
- Properties:
  - Routing table size is O(log *N*) , where *N* is the total number of nodes
  - Guarantees that a file is found in O(log *N*) hops

from MIT in 2001

Routing: Chord

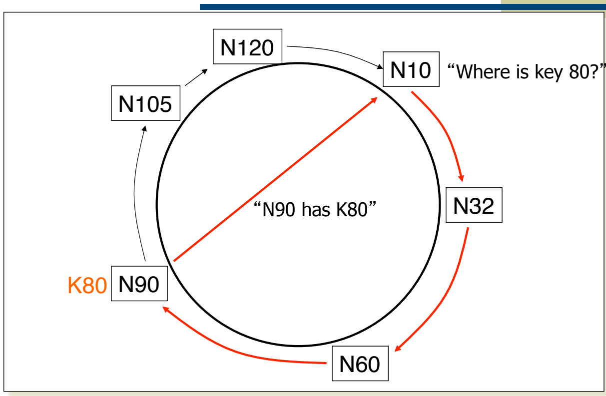
- Associate to each node and item a unique *id* in an *uni*-dimensional space
- Properties
  - Routing table size O(log(*N*)) , where *N* is the total number of nodes
  - Guarantees that a file is found in O(log(*N*)) steps

DHT: Consistent Hashing

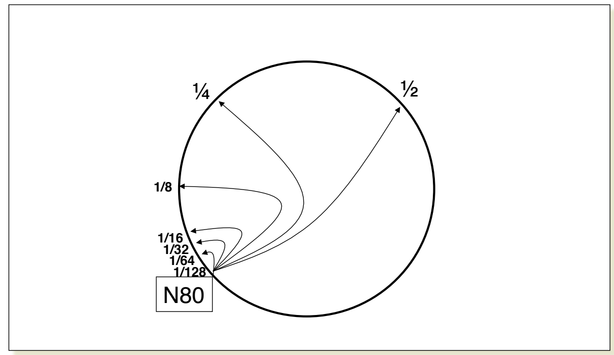


A key is stored at its successor: node with next higher ID

Routing: Chord Basic Lookup



Routing: Finger table - Faster Lookups

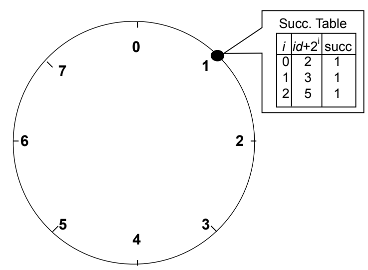


Routing: Chord Summary

- Assume identifier space is $0\ldots 2^m$
- Each node maintains
  - Finger table
    - Entry *i* in the finger table of *n* is the first node that succeeds or equals $n + 2^i$
  - Predecessor node
- An item identified by *id* is stored on the successor node of *id*

Routing: Chord Example

- Assume an identifier space 0..7
- Node n1:(1) joins→all entries in its finger table are initialized to itself

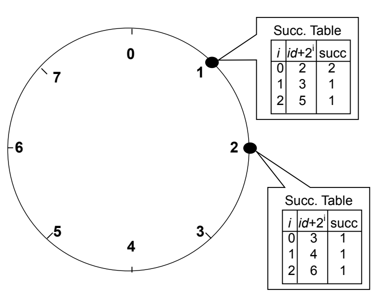


Succ. Table

i	$id+2^i$	succ
0	2	1
1	3	1
2	5	1

Routing: Chord Example

- Node n2:(3) joins



Succ. Table

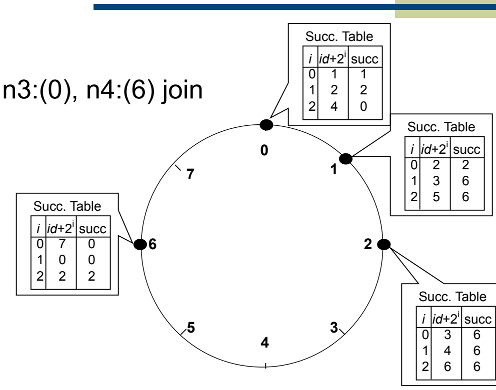
i	$id+2^i$	succ
0	2	2
1	3	1
2	5	1

Succ. Table

i	$id+2^i$	succ
0	3	1
1	4	1
2	6	1

Routing: Chord Example

- Nodes n3:(0), n4:(6) join



Succ. Table

i	$id+2^i$	succ
0	1	1
1	2	2
2	4	0

Succ. Table

i	$id+2^i$	succ
0	2	2
1	3	6
2	5	6

Succ. Table

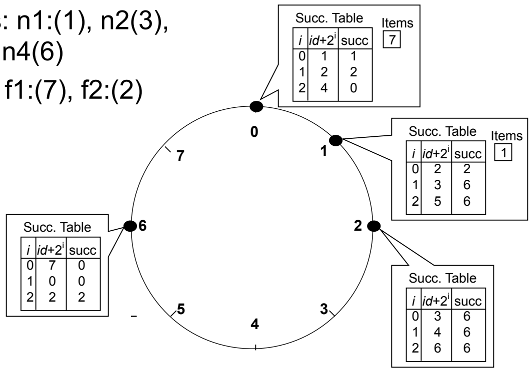
i	$id+2^i$	succ
0	7	0
1	0	0
2	2	2

Succ. Table

i	$id+2^i$	succ
0	3	6
1	4	6
2	6	6

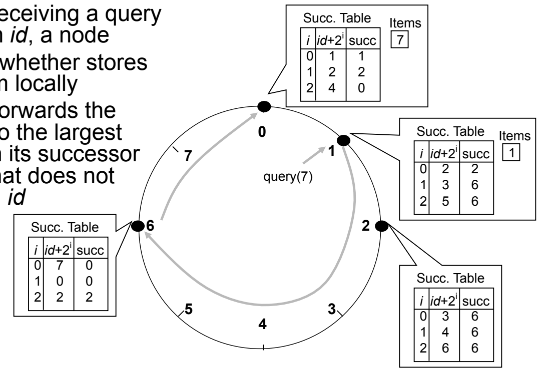
Routing: Chord Examples

- Nodes: n1:(1), n2(3), n3(0), n4(6)
- Items: f1:(7), f2:(2)



Routing: Query

- Upon receiving a query for item *id*, a node
- Check whether stores the item locally
- If not, forwards the query to the largest node in its successor table that does not exceed *id*



DHT: Chord Summary

- Routing table size?
  - Log *N* fingers
- Routing time?
  - Each hop expects to 1/2 the distance to the desired id => expect O(log *N*) hops.

DHT: Discussion

- Pros:
  - Guaranteed Lookup
  - O(log *N*) per node state and search scope
- Cons:
  - No one uses them? (only one file sharing app)
  - Supporting non-exact match search is hard

What can DHTs do for us?

- Distributed object lookup
  - Based on object ID
- De-centralized file systems
  - CFS, PAST, Ivy
- Application Layer Multicast
  - Scribe, Bayeux, Splitstream
- Databases
  - PIER

When are p2p / DHTs useful?

- Caching and "soft-state" data
  - Works well! BitTorrent, KaZaA, etc., all use peers as caches for hot data
- Finding read-only data
  - Limited flooding finds hay
  - DHTs find needles
- BUT

A Peer-to-peer Google?

- Complex intersection queries (“the” + “who”)
  - Billions of hits for each term alone
- Sophisticated ranking
  - Must compare many results before returning a subset to user
- Very, very hard for a DHT / p2p system
  - Need high inter-node bandwidth
  - (This is exactly what Google does - massive clusters)

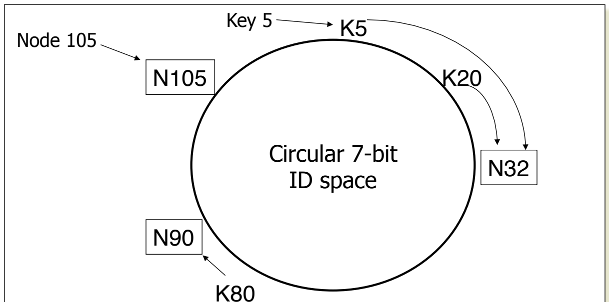
Writable, persistent p2p

- Do you trust your data to 100,000 monkeys?
- Node availability hurts
  - Ex: Store 5 copies of data on different nodes
  - When someone goes away, you must replicate the data they held
  - Hard drives are *huge*, but cable modem upload bandwidth is tiny - perhaps 10 Gbytes/day
  - Takes many days to upload contents of 200GB hard drive. Very expensive leave/replication situation!

P2P: Summary

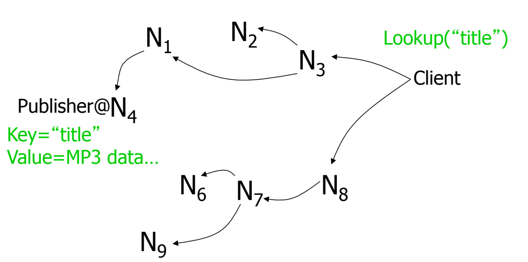
- Many different styles; remember pros and cons of each
  - centralized, flooding, swarming, unstructured and structured routing
- Lessons learned:
  - Single points of failure are very bad
  - Flooding messages to everyone is bad
  - Underlying network topology is important
  - Not all nodes are equal
  - Need incentives to discourage freeloading
  - Privacy and security are important
  - Structure can provide theoretical bounds and guarantees

Aside: Consistent Hashing [Karger 97]



A key is stored at its successor: node with next higher ID

Flooded Queries (Gnutella)



Robust, but worst case O(N) messages per lookup

Flooding: Old Gnutella

- On startup, client contacts any servent (**serv**er + cli**ent**) in network
  - Servent interconnection used to forward control (queries, hits, etc)
- Idea: broadcast the request
- How to find a file:
  - Send request to all neighbors
  - Neighbors recursively forward the request
  - Eventually a machine that has the file receives the request, and it sends back the answer
  - Transfers are done with HTTP between peers

Flooding: Old Gnutella

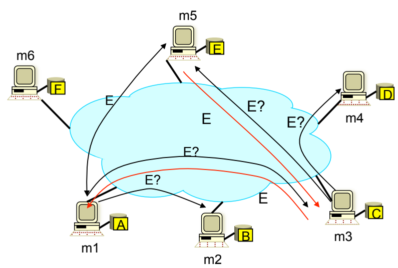
- Advantages:
  - Totally decentralized, highly robust
- Disadvantages:
  - Not scalable; the entire network can be swamped with request (to alleviate this problem, each request has a TTL)
  - Especially hard on slow clients
    - At some point broadcast traffic on Gnutella exceeded 56kbps – what happened?
    - Modem users were effectively cut off!

Flooding: Old Gnutella Details

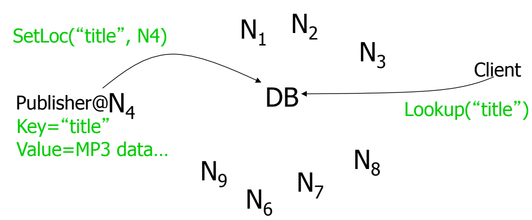
- Basic message header
  - Unique ID, TTL, Hops
- Message types
  - Ping – probes network for other servents
  - Pong – response to ping, contains IP addr, # of files, # of Kbytes shared
  - Query – search criteria + speed requirement of servent
  - QueryHit – successful response to Query, contains addr + port to transfer from, speed of servent, number of hits, hit results, servent ID
  - Push – request to servent ID to initiate connection, used to traverse firewalls
- Ping, Queries are flooded
- QueryHit, Pong, Push reverse path of previous message

Flooding: Old Gnutella Example

Assume: m1's neighbors are m2 and m3; m3's neighbors are m4 and m5;…

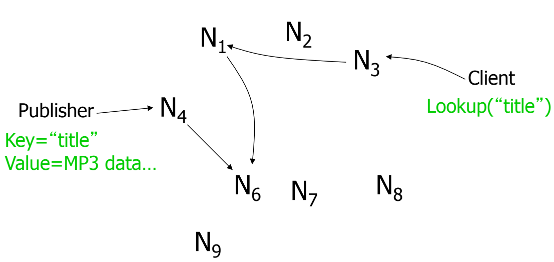


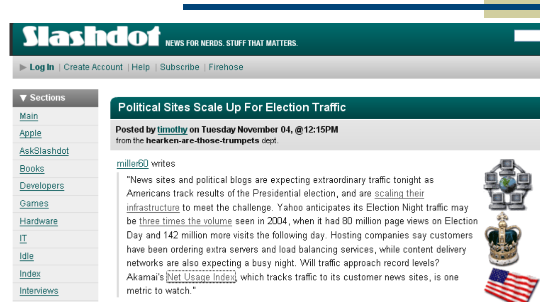
Centralized Lookup (Napster)



Simple, but O(N) state and a single point of failure

Routed Queries (Chord, etc.)





http://www.akamai.com/html/technology/nui/news/index.html

Content Distribution Networks & Server Selection

- Replicate content on many servers
- Challenges
  - How to replicate content
  - Where to replicate content
  - How to find replicated content
  - How to choose among known replicas
  - How to direct clients towards replica

Server Selection

- Which server?
  - Lowest load → to balance load on servers
  - Best performance → to improve client performance
    - Based on Geography? RTT? Throughput? Load?
  - Any alive node → to provide fault tolerance
- How to direct clients to a particular server?
  - As part of routing → anycast, cluster load balancing
    - Not covered ☹
  - As part of application → HTTP redirect
  - As part of naming → DNS

Application Based

- HTTP supports simple way to indicate that Web page has moved (30X responses)
- Server receives Get request from client
  - Decides which server is best suited for particular client and object
  - Returns HTTP redirect to that server
- Can make informed application specific decision
- May introduce additional overhead → multiple connection setup, name lookups, etc.
- While good solution in general, but…
  - HTTP Redirect has some design flaws – especially with current browsers

Naming Based

- Client does name lookup for service
- Name server chooses appropriate server address
  - A-record returned is "best" one for the client
- What information can name server base decision on?
  - Server load/location → must be collected
  - Information in the name lookup request
    - Name service client → typically the local name server for client