Lecture 14

Content Lookup,
Small-world networks

10/15/2013
Outline

- Admin and recap
- P2P networks
  - Overview
  - The scalability problem
  - The lookup problem
Admin

- **Assignment three questions**
  - Please refresh class home page for FAQs.

- **Exam 1**
Recap: The Scalability Problem

- **Maximum throughput**

\[ R = \min \{C_0, \frac{(C_0 + \sum C_i)}{n} \} \]
Recap: Key Design Issues for Full Scalability

- Robustness
- Efficiency
- Incentive
- Lookup problem
Recap: BitTorrent

- If peer B has a piece that peer A needs, peer A sends interested to B.

- unchoke: indicate that B allows A to request.

- request: A requests a specific block from B.

- piece: specific data.

http://www.bittorrent.org/beps/bep_0003.html
BitTorrent Fluid Analysis

- Normalize file size to 1
- $x(t)$: number of downloaders (also known as leechers) who do not have all pieces at time $t$.
- $y(t)$: number of seeds in the system at time $t$.
- $\lambda$: the arrival rate of new requests.
- $\mu$: the uploading bandwidth of a given peer.
- $c$: the downloading bandwidth of a given peer, assume $c \geq \mu$.
- $\theta$: the rate at which downloaders abort download.
- $\gamma$: the rate at which seeds leave the system.
- $\eta$: indicates the effectiveness of downloader sharing, $\eta$ takes values in $[0, 1]$. 
System Evolution

\[
\frac{dx}{dt} = \lambda - \theta x(t) - \min\{cx(t), \mu(\eta x(t) + y(t))\},
\]
\[
\frac{dy}{dt} = \min\{cx(t), \mu(\eta x(t) + y(t))\} - \gamma y(t),
\]

Solving steady state:
\[
\frac{dx(t)}{dt} = \frac{dy(t)}{dt} = 0
\]

Define
\[
\frac{1}{\beta} = \max\left\{\frac{1}{c}, \frac{1}{\eta}\left(\frac{1}{\mu} - \frac{1}{\gamma}\right)\right\}
\]
\[
\bar{x} = \frac{\lambda}{\beta(1 + \theta \beta)}
\]
\[
\bar{y} = \frac{\lambda}{\gamma(1 + \theta \beta)}.
\]
System State

Q: How long does each downloader stay as a downloader?

\[ \bar{x} = \frac{\lambda}{\beta (1 + \frac{\theta}{\beta})} \]

\[ T = \frac{1}{\theta + \beta} \]

\[ \frac{1}{\beta} = \max \left\{ \frac{1}{c}, \frac{1}{\eta} (\frac{1}{\mu} - \frac{1}{\gamma}) \right\} \]
Summary: P2P Data

- BitTorrent is one of the most widely used P2P systems.
- Commercial systems often use simpler topologies or use servers.

http://eeweb.poly.edu/faculty/yongliu/docs/imc12.pdf
Outline

- Admin and recap
- P2P networks
  - Overview
  - The scalability problem
  - The lookup problem
The Lookup Problem

find where a particular file is stored
pay particular attention to see its equivalence of DNS

Key="title"
Value=MP3 data...
PUBLISHER

Client
Lookup("title")

Publisher

N1

N2

N3

N4

N5

N6

Internet
Outline

- Recap
  - P2P
    - the lookup problem
    - Napster
Centralized Database: Napster

- Program for sharing music over the Internet
- History:
  - 5/99: Shawn Fanning (freshman, Northeastern U.) founded Napster Online music service, wrote the program in 60 hours
  - 12/99: first lawsuit
  - 3/00: 25% UWisc traffic Napster
  - 2000: est. 60M users
  - 2/01: US Circuit Court of Appeals: Napster knew users violating copyright laws
  - 7/01: # simultaneous online users: Napster 160K
  - 9/02: bankruptcy

We are referring to the Napster before closure.
Napster: How Does it Work?

Application-level, client-server protocol over TCP

A centralized index system that maps files (songs) to machines that are alive and with files

Steps:
- Connect to Napster server
- Upload your list of files (push) to server
- Give server keywords to search the full list
- Select “best” of hosts with answers
Napster Architecture

THE ORIGINAL NAPSTER
Napster provided a central directory of users who had files to share.
Napster: Publish

I have X, Y, and Z!

123.2.21.23

insert(X, 123.2.21.23)
Napster: Search

Where is file A?

123.2.0.18

search(A)

--> 123.2.0.18

124.1.0.1

Query

Reply

124.1.0.1
Napster: Ping

123.2.0.18

ping

124.1.0.1

ping

124.1.0.1
Napster: Fetch

123.2.0.18

fetch

124.1.0.1
Napster Messages

General Packet Format

[chunksize] [chunkinfo] [data...]

CHUNKSIZE:
Intel-endian 16-bit integer
size of [data...] in bytes

CHUNKINFO: (hex)
Intel-endian 16-bit integer.

00 - login rejected
02 - login requested
03 - login accepted
0D - challenge? (nuprin1715)
2D - added to hotlist
2E - browse error (user isn't online!)
2F - user offline
5B - whois query
5C - whois result
5D - whois: user is offline!
69 - list all channels
6A - channel info
90 - join channel
91 - leave channel
.....
Centralized Database: Napster

- Summary of features: a hybrid design
  - control: client-server (aka special DNS) for files
  - data: peer to peer

- Advantages
  - simplicity, easy to implement sophisticated search engines (boolean exp) on top of the index system

- Disadvantages
  - application specific (compared with DNS)
  - lack of robustness, scalability: central search server single point of bottleneck/failure
  - easy to take down!
Variation: BitTorrent

- A global central index server is replaced by one tracker per file (called a swarm)
  - reduces centralization; but needs other means to locate trackers
Recap

- *P2P*
  - *the lookup problem*
    - Napster (central query server; distributed data servers)
  - *Gnutella*
Gnutella

- On March 14th 2000, J. Frankel and T. Pepper from AOL’s Nullsoft division (also the developers of the popular Winamp mp3 player) released Gnutella.

- Within hours, AOL pulled the plug on it.

- Quickly reverse-engineered and soon many other clients became available: Bearshare, Morpheus, LimeWire, etc.
Decentralized Flooding: Gnutella

- On startup, client contacts other servents (server + client) in network to form interconnection/peering relationships
  - servent interconnection used to forward control (queries, hits, etc)

- How to find a resource record: decentralized flooding
  - send requests to neighbors
  - neighbors recursively forward the requests

![Diagram of decentralized flooding in Gnutella network]
Decentralized Flooding
Decentralized Flooding

- Each node forwards the query to its neighbors other than the one who forwards it the query
Each node should keep track of forwarded queries to avoid loop!

- node state: nodes keep state (which will time out---soft state)
- packet state: carry the state in the query, i.e. carry a list of visited nodes
Decentralized Flooding: Gnutella

- Basic message header
  - Unique ID, TTL, Hops

- Message types
  - Ping - probes network for other servents
  - Pong - response to ping, contains IP addr, # of files, etc.
  - Query - search criteria + speed requirement of servent
  - QueryHit - successful response to Query, contains addr + port to transfer from, speed of servent, etc.
  - Ping, Queries are flooded
  - QueryHit, Pong: reverse path of previous message
Advantages and Disadvantages of Gnutella

Advantages:
- totally decentralized, highly robust

Disadvantages:
- not scalable; the entire network can be swamped with flood requests
  - especially hard on slow clients; at some point broadcast traffic on Gnutella exceeded 56 kbps
- to alleviate this problem, each request has a TTL to limit the scope
  - each query has an initial TTL, and each node forwarding it reduces it by one; if TTL reaches 0, the query is dropped (consequence?)
Flooding: FastTrack (aka Kazaa)

- Modifies the Gnutella protocol into two-level hierarchy
  - 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

- Search
  - Broadcast (Gnutella-style) search across supernodes
Outline

Recap

- **P2P**
  - *the lookup problem*
    - Napster (central query server; distributed data server)
    - Gnutella (decentralized, flooding)
  - *Freenet*
Freenet

- History
  - final year project Ian Clarke, Edinburgh University, Scotland, June, 1999

- Goals:
  - totally distributed system without using centralized index or broadcast (flooding), instead search by routing
  - routing/storing system responds adaptively to usage patterns, transparently moving, replicating files as necessary to provide efficient service
  - provide publisher anonymity, security
  - free speech: resistant to attacks - a third party shouldn’t be able to deny (e.g., deleting) the access to a particular file (data item, object)
Basic Structure of Freenet

- Each machine stores a set of files; each file is identified by a unique identifier (called key or id)

- Each node maintains a “routing table”
  - $id$ - file id, key
  - $next_{\text{hop \hspace{1pt} node}}$ - where to search for a file with id’ that is similar to id
  - $file$ - local copy, if exists, of file with id
Freenet Query

- API: `file = query(id);`

- Upon receiving a query for file `id`
  - check whether the queried file is stored locally
  - check TTL to limit the search scope
    - each query is associated a TTL that is decremented each time the query message is forwarded
    - when TTL=1, the query is forwarded with a probability
    - TTL can be initiated to a random value (why random value?)
  - look for the "closest" `id` in the table with an unvisited `next_hop` node
    - if found one, forward the query to the corresponding `next_hop`
    - otherwise, backtrack
      - ends up performing a Depth First Search (DFS)-like traversal
      - search direction ordered by closeness to target

- When file is returned it is cached along the reverse path (any advantage?)
Beside the routing table, each node also maintains a query table containing the state of all outstanding queries that have traversed it → to backtrack.
INSERT

API: insert(id, file);

Two steps

- first attempt a “search” for the file to be inserted
- if found, report collision

- if not found, insert the file by sending it along the query path (why?)
  - a node probabilistically replaces the originator with itself (why?)
Insert Example

- Assume query returned failure along the shown path (backtrack slightly complicates things); insert f10

```
insert(10, f10)
```

```
1 n1  3 n1  5 n3
4 f4  9 f9
12 f12
```

```
3 n1  14 n4  5 n3
3 f3  14 f14
14 n2 f14
```

```
14 n5 f14
13 n2 f13
3 n6
```

```
4 n1 f4
11 n5 f11
8 n6
```
Insert Example

```
insert(10, f10)
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>n1</td>
<td>f10</td>
</tr>
<tr>
<td>4</td>
<td>n1</td>
<td>f4</td>
</tr>
<tr>
<td>12</td>
<td>n2</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>n3</td>
<td>f9</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>n1</td>
<td>f3</td>
</tr>
<tr>
<td>14</td>
<td>n4</td>
<td>f14</td>
</tr>
<tr>
<td>5</td>
<td>n3</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>n5</td>
<td>f14</td>
</tr>
<tr>
<td>13</td>
<td>n2</td>
<td>f13</td>
</tr>
<tr>
<td>3</td>
<td>n6</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>n1</td>
<td>f4</td>
</tr>
<tr>
<td>11</td>
<td>n5</td>
<td>f11</td>
</tr>
<tr>
<td>8</td>
<td>n6</td>
<td></td>
</tr>
</tbody>
</table>
```
Insert Example

insert(10, f10)

n1

<table>
<thead>
<tr>
<th>10</th>
<th>n1</th>
<th>f10</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>n1</td>
<td>f4</td>
</tr>
<tr>
<td>12</td>
<td>n2</td>
<td></td>
</tr>
</tbody>
</table>

n2

<table>
<thead>
<tr>
<th>10</th>
<th>n1</th>
<th>f10</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>n3</td>
<td>f9</td>
</tr>
</tbody>
</table>

n3

<table>
<thead>
<tr>
<th>3</th>
<th>n1</th>
<th>f3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>n4</td>
<td>f14</td>
</tr>
<tr>
<td>5</td>
<td>n3</td>
<td></td>
</tr>
</tbody>
</table>

n4

<table>
<thead>
<tr>
<th>14</th>
<th>n5</th>
<th>f14</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>n2</td>
<td>f13</td>
</tr>
<tr>
<td>3</td>
<td>n6</td>
<td></td>
</tr>
</tbody>
</table>

n5

<table>
<thead>
<tr>
<th>4</th>
<th>n1</th>
<th>f4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>n5</td>
<td>f11</td>
</tr>
<tr>
<td>8</td>
<td>n6</td>
<td></td>
</tr>
</tbody>
</table>
Insert Example

- n2 replaces the originator (n1) with itself

```
insert(10, f10)
```

```
<table>
<thead>
<tr>
<th>orig= n2</th>
</tr>
</thead>
</table>
```

```n1
10   11   14
\[
\begin{array}{ccc}
10 & n1 & f10 \\
11 & n5 & f11 \\
14 & n4 & f14 \\
15 & n3 & f3 \\
\end{array}
\]
```

```n2
10   9   13
\[
\begin{array}{ccc}
10 & n1 & f10 \\
9 & n3 & f9 \\
13 & n2 & f13 \\
14 & n4 & f14 \\
15 & n3 & f3 \\
\end{array}
\]
```

```n3
10   9   13
\[
\begin{array}{ccc}
10 & n1 & f10 \\
9 & n3 & f9 \\
13 & n2 & f13 \\
14 & n4 & f14 \\
15 & n3 & f3 \\
\end{array}
\]
```

```n4
14   13   11
\[
\begin{array}{ccc}
14 & n5 & f14 \\
13 & n2 & f13 \\
11 & n5 & f11 \\
14 & n4 & f14 \\
15 & n3 & f3 \\
\end{array}
\]
```

```n5
4   11   8
\[
\begin{array}{ccc}
4 & n1 & f4 \\
11 & n5 & f11 \\
8 & n6 & f6 \\
\end{array}
\]
```
Insert Example

insert(10, f10)

n1
10 n1 f10
4 n1 f4
12 n2

n2
10 n1 f10
9 n3 f9

n3
10 n2 10
3 n1 f3
14 n4

n4
14 n5 f14
13 n2 f13
3 n6

n5
4 n1 f4
11 n5 f11
8 n6
Insert Example

Insert(10, f10)
Freenet Analysis

Authors claim the following effects:

- nodes eventually specialize in locating similar keys
  - if a node is listed in a routing table, it will get queries for related keys
  - thus will gain “experience” answering those queries
- popular data will be transparently replicated and will exist closer to requestors
- as nodes process queries, connectivity increases
  - nodes will discover other nodes in the network

Caveat: lexicographic closeness of file names/keys may not imply content similarity
We create a Freenet reference graph
- creating a vertex for each Freenet node
- adding a directed link from A to B if A refers to an item stored at B
**Experiment: Freenet Graph: Init**

- Assume a network of 1000 nodes, with node id 0 to 999
- Each node can store 50 data items, and 200 references
- Assume initially each node $i$ has $i$, and knows the storage of $i-2$, $-1$, $i+1$, $i+2$ (all mod 1000)
- thus a regular, locally-clustered graph with avg path length $\sim \frac{1000}{8} = 125$
- What is the effect that if the first search is node 0 searching for item 490 (assume no probabilistic replacement to hide origin)?
  - Nodes 0, 2, 4, 6, ..., 488 all cache item 490, and has a pointer to node 490
  - The search forms many long-distance links
Experiment: Evolution of Freenet Graph

- At each step
  - pick a node randomly
  - flip a coin to determine search or insert
    - if search, randomly pick a key in the network
    - if insert, pick a random key

Evolution of path length and clustering; Clustering is defined as percentage of local links
Freenet Evolves to Small-World Network

- With usage, the regular, highly localized Freenet network evolved into one irregular graph.

- High percentage of highly connected nodes provide shortcuts/bridges:
  - make the world a “small world”
  - most queries only traverse a small number of hops to find the file
First discovered by Milgrom

- in 1967, Milgram mailed 160 letters to a set of randomly chosen people in Omaha, Nebraska
- goal: pass the letters to a given person in Boston
  - each person can only pass the letter to an intermediary known on a first-name basis
  - pick the person who may make the best progress
- result: 42 letters made it through!
- median intermediaries was 5.5---thus six degree of separation

- a potential explanation: highly connected people with non-local links in mostly locally connected communities improve search performance!
Question: what kind of long distance links to maintain so that distributed network search is effective?

Assume that each node has

- a fixed # (say p distance away) local links
- a small # (say a total of q) long-distance links s.t. the probability of a link between nodes x and y is some ($\alpha$) inverse-power of the distance $d(x, y)$ of x and y

\[ \frac{d(x, y)^{-\alpha}}{\Delta_\alpha} \]

- Different alpha's give diff types of links.
- Q: what is a good alpha?
What Should the Long-Distance Links Look?

- Consider the simple case of one dimensional space.

- Which alpha leads to best performing distributed search alg?
What Should the Long-Distance Links Look?

[Graph showing the different a values]
What Should the Long-Distance Links Look?

- For 1-d space, for any distributed algorithm, the expected # of search steps, for different $\alpha$'s:
  - $0 \leq \alpha < 1 : \geq k_1 n^{(1-\alpha)/2}$
  - $\alpha > 1 : \geq k_1 n^{(\alpha-1)/\alpha}$
  - $\alpha = 1 : O(\log^2 n)$ greedy search
In the general case, \( \alpha \) should be the dimension.

In other words, a guideline on long-distance links: roughly the same number of nodes in each region \( A_1, A_2, A_4, A_8 \), where \( A_1 \) is the set of nodes who are one lattice step away, \( A_2 \) is those two steps away, \( A_4 \) is four steps away...

probability is proportional to (lattice steps)^{-d}
Small World

Saul Steinberg; View of World from 9th Ave
Freenet: Issues

- Does **not** always guarantee that a file is found, even if the file is in the network.

- Good average-case performance, but a potentially **long search path** in a large network.
  - approaching small-world...
Summary

- All of the previous p2p systems are called **unstructured** p2p systems
  - Napster (central query server)
  - Gnutella (decentralized, flooding)
  - Freenet (search by routing)

- Advantages of unstructured p2p
  - algorithms tend to be simple

- Disadvantages
  - hard to make performance guarantee
  - failure even when files exist
Distributed Hash Tables (DHT)

- In 2000-2001, academic researchers jumped on to the P2P bandwagon

**Motivation:**
- frustrated by popularity of all these “half-baked” P2P apps. We can do better! (so they said)
- guaranteed lookup success for data in system
- provable bounds on search time (motivated by the Kleinberg observation)
- provable scalability to millions of node

**Examples**
- CAN, Chord
  - See optional slides at the end of this lecture
Optional Slides
DHT: Overview

- Abstraction: a distributed “hash-table” (DHT) data structure
  - put(key, value) and get(key) → value
  - DHT imposes no structure/meaning on keys
  - one can build complex data structures using DHT

- Implementation:
  - nodes in system form an interconnection network: ring, zone, tree, hypercube, butterfly network, ...

**Distributed application**

put(key, data) ↑ get (key) ↓

**DHT**

node node .... node
DHT Applications

- File sharing and backup [CFS, Ivy, OceanStore, PAST, Pastiche ...]
- Web cache and replica [Squirrel, Croquet Media Player]
- Censor-resistant stores [Eternity]
- DB query and indexing [PIER, Place Lab, VPN Index]
- Event notification [Scribe]
- Naming systems [ChordDNS, Twine, INS, HIP]
- Communication primitives [I3, ...]
- Host mobility [DTN Tetherless Architecture]
DHT: Basic Idea
DHT: Basic Idea (2)
DHT: Basic Idea (3)
DHT: Basic Idea (4)
DHT: Basic Idea (5)
Abstraction

- map a key to a “point” in a multi-dimensional Cartesian space

- a node “owns” a zone in the overall space

- route from one “point” to another
CAN Example: Two Dimensional Space

- Space divided among nodes

- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
CAN Example: Two Dimensional Space

- Space divided among nodes

- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
CAN Example: Two Dimensional Space

- Space divided among nodes
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
CAN Insert: Example (1)

node $I::\text{insert}(K,V)$
CAN Insert: Example (2)

```
ode I::insert(K,V)
(1) a = h_x(K)
    b = h_y(K)
```

Example: Key="Matrix3" h(Key)=60
CAN Insert: Example (3)

node $I::\text{insert}(K,V)$

(1) $a = h_x(K)$
    $b = h_y(K)$

(2) $\text{route}(K,V) \rightarrow (a,b)$
CAN Insert: Routing

- A node maintains state only for its immediate neighboring nodes.
- Forward to neighbor which is closest to the target point:
  - a type of greedy, local routing scheme.
**CAN Insert: Example (4)**

node `I::insert(K,V)`

1. \( a = h_x(K) \)
   \( b = h_y(K) \)

2. `route(K,V) -> (a,b)`

3. \((K,V)\) is stored at \((a,b)\)
CAN Retrieve: Example

node J::retrieve(K)
CAN Retrieve: Example

node $J::\text{retrieve}(K)$

1. $a = h_x(K)$
   
2. $b = h_y(K)$

(2) route “retrieve(K)” to $(a,b)$
1) Discover some node “J” already in CAN

2) pick a random point (p,q) in space
3) J routes to \((p,q)\), discovers node \(N\)
CAN Insert: Join (3)

Inserting a new node affects only a single other node and its immediate neighbors.

4) split N’s zone in half... new node owns one half
CAN Evaluations

- Guarantee to find an item if in the network

- Load balancing
  - hashing achieves some load balancing
  - overloaded node replicates popular entries at neighbors

- Scalability
  - for a uniform (regularly) partitioned space with \( n \) nodes and \( d \) dimensions
  - storage:
    - per node, number of neighbors is \( 2d \)
  - routing
    - average routing path is \( (dn^{1/d})/3 \) hops (due to Manhattan distance routing, expected hops in each dimension is dimension length * 1/3)
    - a fixed \( d \) can scale the network without increasing per-node state
Chord

- Space is a ring

- Consistent hashing: m bit identifier space for both keys and nodes
  - key identifier = SHA-1(key), where SHA-1() is a popular hash function,
    Key="Matrix3" → ID=60
  - node identifier = SHA-1(IP address)
    • IP="198.10.10.1" → ID=123
Chord: Storage using a Ring

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

IP="198.10.10.1"

Key="Matrix 3"
How to Search: One Extreme

- Every node knows of every other node
- Routing tables are large $O(N)$
- Lookups are fast $O(1)$
How to Search: the Other Extreme

- Every node knows its successor in the ring

Routing tables are small $O(1)$
Lookups are slow $O(N)$
Chord Solution: “finger tables”

- Node $K$ knows the node that is maintaining $K + 2^i$, where $K$ is mapped id of current node
  - increase distance exponentially
Joining the Ring

- use a contact node to obtain info
- transfer keys from successor node to new node
- updating fingers of existing nodes

Copy keys 21..36 from N40 to N36
DHT: Chord Node Join

- Assume an identifier space [0..8]
- Node n1 joins

- Succ. Table
<table>
<thead>
<tr>
<th>i</th>
<th>(id+2^i)</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Node n2 joins
DHT: Chord Node Join

- Node n6 joins

\[
\begin{array}{c|cc|c}
  i & id+2^i & \text{succ} \\
  0 & 7 & 1 \\
  1 & 0 & 1 \\
  2 & 2 & 2 \\
\end{array}
\]
DHT: Chord Node Join

- Node n0 starts join

Succ. Table

<table>
<thead>
<tr>
<th>i</th>
<th>id+2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Succ. Table

<table>
<thead>
<tr>
<th>i</th>
<th>id+2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
After Nodes n0 joins

DHT: Chord Node Join
DHT: Chord Insert Items

- **Nodes:** n1, n2, n0, n6
- **Items:** f7, f1
DHT: Chord Routing

- Upon receiving a query for item \( id \), a node:
  - checks whether stores the item locally
  - if not, forwards the query to the largest node in its successor table that does not exceed \( id \)

<table>
<thead>
<tr>
<th>Succ. Table</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>( id + 2^i )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Succ. Table</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>( id + 2^i )</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
Chord/CAN Summary

- Each node “owns” some portion of the key-space
  - in CAN, it is a multi-dimensional “zone”
  - in Chord, it is the key-id-space between two nodes in 1-D ring

- Files and nodes are assigned random locations in key-space
  - provides some load balancing
    - probabilistically equal division of keys to nodes

- Routing/search is local (distributed) and greedy
  - node X does not know of a path to a key Z
  - but if it appears that node Y is the closest to Z among all of the nodes known to X
  - so route to Y