lookup services

Badri Nath
Rutgers University
badri@cs.rutgers.edu

1. CAN: A scalable content addressable network, Sylvia Ratnasamy et.al. SIGCOMM 2001
Distributed lookup services

A set of nodes cooperating

- Peers
- Run special purpose algorithms/software
- Doesn’t have to be deployed at every node
- Ignore underlay network
Search or lookup service

Find a host that satisfies some property
In a distributed, dynamic, internet-scale manner

- No central state, nodes come and go

Storage, overlay networks, databases, diagnosis

- Locate an item
- Locate a node
- Locate a tuple
- Locate an event
Lookup services classification

Centralized
  ● Napster

Flooding
  ● Gnutella

Distributed Hashing (DHT)
  ● CAN, CHORD, Tapestry, Pastry, Kademlia
Centralized

Central server directory

IP1
IP2
IP3
IP4
IP5

Slum Dog Millionaire?
IP3, IP5

Example: Napster
Napster

Client server protocol over TCP

- Connect to well-known napster server
- Upload <file names, keywords> that you want to share
- Select best client/peer to download files
- Selection done by pinging peers and selecting the one with lowest latency or best transfer rate

Popular for downloading music among peers

- Copyright issue
- Shutdown for legal reasons
Napster Pros and cons

Pros

- Simple
- Control easy
- Server can be made to scale with clusters etc

Cons

- Server bottleneck
- Need open servers (firewall, NAT)
- No security
- No authentication
- No anonymity
flooding

Gnutella a file sharing service based on flooding
A well known node acts as an anchor
  - Nodes who have files inform this node about their existence
  - Nodes select other nodes as “peers”
  - Form an overlay based on some criteria
Each node stores a select number of files
  - Send a query to peers if file not found locally
  - Peers respond to the query if found else
    - Reroute query to its neighbor peers and so on
Basically flood the network for a search of the file
Gnutella search and flooding

Search

- A request by a node NodeID for string S
- Check local system, if not found
- Form a descriptor <NodeID, S, N, T>
- N is a unique request ID, T is TTL time-to-live

Flood

- Send descriptor to fixed number (6 or 7) of peers
- If S not found, decrement TTL, send to its (6 or 7) peers
- Stop flooding if request ID already seen or TTL expiry
**Gnutella back propagation**

When NodeB receives a request (NodeA, S, ReqID, T)
If B already has seen reqid or TTL = 0, do nothing
Lookup locally, if found send it to NodeA
Eventually by back propagation, it reaches the originator
Gnutella protocol messages

Broadcast messages

- PING: initiate message to peers, **hello**, if booting send to anchor node
- Query: <nodeid, S, ReqID, TTL>

Response messages

- Pong: reply to ping, announce self, with Sharing information
- Query response: contains the NodeID that has the requested file

Point-to-point messages

- GET: retrieve the requested file
- PUT (PUSH) upload the file to me
Gnutella flooding

Query

Response

Query-hit

TTL=1

TTL=2

TTL=3

TTL=2

TTL=2

TTL=3

TTL=3

TTL=3

Download
issues

Peers not staying ON all the time
All peers the same
Modem users were the culprit
More intelligent overlay needed
hashing

Search for data based on a key

- Has function maps keys to a range \([0, \ldots, N-1]\)
- \(h(x)=x \mod N\)
- An item \(k\) is stored in index \(i=h(k)\)

Issues

- Collision
- Selection of hash function
- Handling Collisions
  - Chaining and linear probing
  - Double hashing
**Conventional**

Hashing, search tree etc

Hashing

Map key/index to bucket holding data

H(k) = k mod m (prime)

Secondary hash = p - k mod p

H(18) = 18 mod 13 = 5 - bucket

H(41) = 41 mod 13 = 2 - bucket

18, 44, 31

14

18, 44, 31

59
Changes?

Buckets increase / decrease, re-mapping
Add additional buckets
  - $H(k) = k \mod 17$
Remove 2 buckets
  - $H(k) = k \mod 11$
Move items to the new buckets
What to do in a distributed setting (internet scale)
Each node stores a portion of the key space
What Is a DHT?

Distributed Hash Table:
  table is distributed among a set of nodes
  nodes use the same hashing function
  key = Hash(data)
  lookup(key) -> node id that holds data

Two problems
  Partitioning of data
  Lookup or routing
**CAN design**

Key hashes to a point in d-dim space
A node randomly decides on a point P
Use existing CAN nodes to determine the node that owns the zone to which P belongs
Split the zone between existing CAN node and new node
**Dynamic zone creation**

Rotate the dimension on which to split
**CAN Key Assignment**

![Diagram of CAN Key Assignment]

- K1 connects to n1.
- K2 connects to n2.
- K3 connects to n3.
- n4 is located at the middle of the right side.
- n5 is located at the bottom right corner.

Axes: 0,Y and X/2,0.
CAN: node assignment
CAN: node assignment

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

(1) $a = h_x(K)$
**CAN: node assignment**

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

(1) \( a = h_x(K) \)
\( b = h_y(K) \)
CAN: simple example
CAN: simple example

node I::insert(K,V)
**CAN: simple example**

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

(1) $a = h_x(K)$
**CAN: simple example**

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

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

![Diagram](image-url)
**CAN: Phase II routing**

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

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

(2) route($K,V$) $\rightarrow$ (a,b)
**CAN: Phase II routing**

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

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

(2) $\text{route}(K,V) \rightarrow (a,b)$

(3) $(a,b)$ stores $(K,V)$
**CAN: lookup**

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

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

2. route "\text{retrieve}(K)" to $(a,b)$
A node only maintains state for its immediate neighboring nodes
**CAN: Routing/Lookup**

Use neighbors that minimizes distance to destination

- Use neighbors that minimizes distance to destination
- K1
- K2
- n1
- n2
- n3
- K3
- K3?
- n4
- n5
**CAN: routing table**
CAN: routing

![Diagram of CAN routing with coordinates (a,b) and (x,y).]
**CAN: node insertion**

1) Discover some node “I” already in CAN
**CAN: node insertion**

1) discover some node “I” already in CAN
**CAN: node insertion**

1. Pick a random point in space $(p,q)$.
2. Insert the new node.
CAN: node insertion

3) I routes to \((p,q)\), discovers node J

(new node)
CAN: node insertion

4) split J’s zone in half... new owns one half
**CAN: node insertion**

Inserting a new node affects only a single other node and its immediate neighbors.
CAN: Node failures

Simple failures

- know your neighbor’s neighbors

Only the failed node’s immediate neighbors are required for recovery

- when a node fails, one of its neighbors takes over its zone
**CAN: performance**

For a uniformly partitioned space with $n$ nodes and $d$ dimensions
- per node, number of neighbors is $2d$
- average routing path is $o(\sqrt{N})$ hops for 2-dimension
- simulations confirm analysis

Can scale the network without increasing per-node state

Next Chord
- $\log(n)$ fingers with $\log(n)$ hops
CAN: Discussion

Scalable
- State information is $O(d)$ at each node

Locality
- Nodes are neighbors in the overlay, not in the physical network
- Latency stretch
- Paper suggests other metrics
- RTT relative to distance progress
**Consistent hashing**

Hash keys and bucket-id to some uniform name space
Assign key to the first bucket encountered in the name space
Make collisions rare (for the bucket-ids)

When servers/buckets come and go small local movements
Maintain a directory to quickly locate server holding items
**Consistent hashing**

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

Centralized directory
- One node knows cache points
- $O(n)$ state, fixed distance
- Single point of failure
- Napster

Flat
- Every node knows about every other node
- $O(n^2)$ state, $O(n^2)$ communication, fixed distance
- RON

Hierarchical
- Maintain a tree, $\log(n)$ distance, root is bottleneck
**Distributed hash table (DHT)**

- Application may be distributed over many nodes
- DHT distributes data storage over many nodes

**Distributed application**
- `put(key, data)` to `get(key)`
- `lookup(key)` to `node IP address`

**Distributed hash table**
- `node`
- `node`
- `....`
- `node`

**Lookup service**
- (File sharing)
- (DHash)
- (Chord)
A hash table allows you to insert, lookup and delete objects with keys
A *distributed* hash table allows you to do the same in a distributed setting
(objects=files)

Performance Concerns:
- Load balancing
- Fault-tolerance
- Efficiency of lookups and inserts
- Locality

Chord, Pastry, Tapestry, Plaxton, Viceroy, Kademlia, Skipnet, Symphony,
Koorde, Ulysses, Apocrypha, Land, ORDI,....

YADHT .... Yet another DHT papers have started appearing
Variations

Geometry
- Underlying graph structure
- Ring, tree, butterfly

Distance
- Distance between the nodes

Routing
- Protocol for selecting neighbor and route
The lookup problem

Key="boyle"
Value=Slum dog Millionaire
Publisher

Lookup("boyle")
Centralized lookup (Napster)

SetLoc("boyle", N4)
Publisher@N4
  Key="Boyle"
  Value=Slum dog millionaire

N1 N2
DB

N3

N4

N9 N7
N6
N8

Client

Lookup("boyle")

Simple, but $O(N)$ state and a single point of failure
Flooded queries (Gnutella)

Robust, but worst case $O(N)$ messages per lookup
Routed queries (Chord)

Publisher → N₄
Key = “Boyle”
Value = Slum Dog Millionaire...

Client → N₃

Lookup(“Boyle”) → N₆ → N₇ → N₈ → N₉ → N₁ → N₂
### Comparative Performance

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Lookup Latency</th>
<th>#Messages for a lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>$O(1)$ ((O(N)\text{@server}))</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Gnutella</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
## Comparative Performance

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Lookup Latency</th>
<th>#Messages for a lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>($O(N)@$server)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gnutella</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Chord</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
</tr>
</tbody>
</table>
Routing challenges

Define a useful key nearness metric
Keep the hop count small
Keep the tables small
Stay robust despite rapid change
Chord: emphasizes efficiency and simplicity
Chord overview

Provides peer-to-peer hash lookup:
- Lookup(key) → IP address
- Chord does not store the data

How does Chord route lookups?
How does Chord maintain routing tables?
O(n) lookup, O(1) state

“Where is key 80?”

“N90 has K80”
A Simple Key Lookup

If each node knows only how to contact its current successor node on the identifier circle, all nodes can be visited in linear order.

Queries for a given identifier could be passed around the circle via these successor pointers until they encounter the node that contains the key.
Simple lookup algorithm

```
Lookup(my-id, key-id)
    n = my successor
    if my-id < n < key-id
        call Lookup(id) on node n  // next hop
    else
        return my successor          // done
```

Correctness depends only on successors
Join and Departure

When a node \( n \) joins the network, certain keys previously assigned to \( n \)'s successor now become assigned to \( n \).

When node \( n \) leaves the network, all of its assigned keys are reassigned to \( n \)'s successor.
Direct lookup, $O(n)$ state

“Where is key 80?”
“N90 has K80”
Direct Key Lookup

If each node knows all current nodes on the identifier circle, the node that has the key can be directly visited.

Node leave/join expensive
Chord IDs

Key identifier = SecureHASH(key)
Node identifier = SecureHASH(IP address)
Both are uniformly distributed
Both exist in the same ID space

How to map key IDs to node IDs?
What directory structure to maintain
Chord properties

Efficient: $O(\log(N))$ messages per lookup
- $N$ is the total number of servers

Scalable: $O(\log(N))$ state per node

Robust: survives massive failures

Since chord number of proposals for DHT
Routing table size vs distance

Routing table size

Worst case distance

O(1) O(log n) O(dn^{1/d}) O(n)

O(log_{log n} n)

Maintain full state

Plaxton et al., Chord, Pastry, Tapestry

Ulysses, de Bruijn Graph (Koorde etc.)

Viceroy (d=7)

SCAN

Maintain no state

Ulysses: A Robust, Low-Diameter, Low-Latency Peer-to-Peer Network in ICNP 2003
Node Entry

Keys: 4, 5, 6

Keys: 1

Keys: 2

Keys: 7
Node departure

- Node 0:
  - Keys: 7

- Node 6:
  - Keys: 4, 5, 6

- Node 7:
  - Keys: 1

- Node 4:
  - Keys: 2
Scalable Key Location

To accelerate lookups, Chord maintains additional routing information.

A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node.

The first finger of n is the immediate successor of n on the circle.
Scalable Key Location – Finger Tables

Each node $n$ maintains a routing table with up to $m$ entries (which is in fact the number of bits in identifiers), called \textit{finger table}.

The $i^{th}$ entry in the table at node $n$ contains the identity of the \textit{first} node $s$ that succeeds $n$ by at least $2^{i-1}$ on the identifier circle.

$s = \text{successor}(n+2^{i-1})$.

$s$ is called the $i^{th}$ \textit{finger} of node $n$, denoted by $n.finger(i)$.
Building finger table: Example

\( m = 3 \ <0..7> \)

Node \( n_1 \) joins all entries in its finger table are initialized to itself
Example (cont)

Node $n_2$ 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>2</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>

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>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>
Nodes $n_0$, $n_6$ join

Succ. Table

<table>
<thead>
<tr>
<th>i</th>
<th>id+2</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Succ. Table

<table>
<thead>
<tr>
<th>i</th>
<th>id+2</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>
Scalable Key Location - Example query

The path a query for key 54 starting at node 8:
### Finger Table Example

**Finger Table at N80**

<table>
<thead>
<tr>
<th>$i$</th>
<th>$ft[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81</td>
</tr>
<tr>
<td>1</td>
<td>82</td>
</tr>
<tr>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td>3</td>
<td>88</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

$i$th entry at peer with id $n$ is first peer with id $\geq n + 2^i \pmod{2^m}$

with $m=7$
Lookup with fingers

Lookup(my-id, key-id)

look in local finger table for

    highest node n s.t. my-id < n < key-id

if n exists

    call Lookup(id) on node n        // next hop

else

    return my successor            // done
Lookups take $O(\log(N))$ hops
Simulation Results: $\frac{1}{2} \log_2(N)$

- Error bars mark 1st and 99th percentiles
**node_join**

Introducer directs N40 to N32
N32 updates successor to N40
N40 initializes successor to N45

*N40 periodically talks to neighbors to update finger table*

Say $m=7$
node join

N40 may need to copy some files/keys from N45
(files with file-id between 32 and 40)

Say $m=7$

K34, K38, K42
Intermediate peer failure

Say \( m = 7 \)

Lookup Trump with key K40 stored in N45

Lookup fails (N16 does not know N45)
Replicate successor pointers

Maintain r successor pointers

Say $m=7$

Lookup Trump with key $K40$ stored in $N45$
Node failure

Node that stores the file has failed

Say $m=7$

Lookup Trump with key $K40$ stored in $N45$

Lookup fails $N45$—not responding

Trump with key $K40$ stored in $N45$
Node failure

Replicate files at K succ and pred nodes

Say \( m = 7 \)

Lookup Trump with key \( K40 \) stored in N45

Trump with key \( K40 \) stored in N32

Trump with key \( K40 \) stored in N45
Chord latency stretch

Neighbor on the virtual ring can be physically far apart
Chord Summary

Chord provides peer-to-peer hash lookup
Efficient: $O(\log(n))$ messages per lookup
Robust as nodes fail and join
Good primitive for peer-to-peer systems
**CAN vs CHORD**

From Chord to CAN
- From one dim hashing to d-dim hashing

Nodes form an overlay in $d$-dimensional space
- Node IDs are chosen randomly from the d-space
- Object IDs (keys) are chosen from the same d-space

Nodes form an overlay of a ring
- Finger tables used to improve lookup to $O(\log N)$

Nodes join/leave
Zones are split and merged as nodes join and leave
Finger tables are adjusted
Local movement of data objects