eecs489 COMPUTER NETWORKS

Lecture 10: Content-based Routing and Consistent Hashing

Name-based Network

Today's Internet: address-based packet forwarding
- applications must first resolve a name to an address
- establish an end-to-end session with the returned address

Name-based network:
- name resolution and session establishment as one
- session establishment based on name (abstract ID) instead of an address
- no separate address beyond name
- a.k.a. information-centric, content-centric, content-oriented, content-addressable network

Characteristics of names:
- object agnostic: content, hosts, services, users, etc.
- cannot be easily aggregated by topological location

Network Architecture

Instead of DNS-based name resolution, objects are located using a publish-subscribe mechanism
- objects published by principal (owner of object)
- (replicated in caches by network)
- requested by name by subscriber
- (objects can be returned from any copy)

Examples:
- BitTorrent's Tracker, Skype's ID
- Amazon's Dynamo (paper linked to in syllabus)
  - highly-available key-value store
  - used for maintaining shopping cart, wish list, reviews, etc.

Key-Value Store

Database (DB) entries consist of <key, value> pairs, for example:
- key: title; value: song
- key: SSN; value: person's data
- key: sessionID; value: shopping cart
- key: sessionID; value: wish list
- key: itemID; value: reviews

Publish: object owner inserts value into DB by key
Subscribe: subscriber looks up value by key

Distributed Database

DB is distributed across several nodes

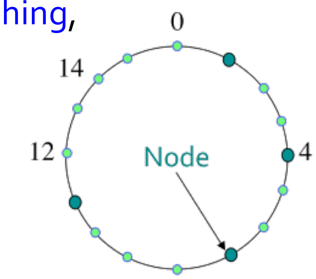
- each node stores only a portion of the DB

How to partition the DB to each node? Want:

- even spread: load is evenly spread across nodes
- fast lookup: faster than linear search
- localized changes: addition and removal of node requires only changes to nearby nodes, not to the whole network
  - consider conventional $\bmod m$ hashing: adding a node $(m+1)$ requires changing/rehashing the content of every node!

Consistent Hashing

One solution is to use consistent hashing, a.k.a., distributed hash table (DHT)



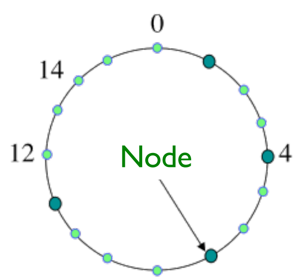
Chord is an example of a DHT:

- specify an identifier key size, n bits
  - here, $n = 4$
- arrange IDs in order on an identifier ring/circle
- given N nodes, assign each to a location on the ring $(\bmod 2^n)$
  - here, $N = 4$
- hash/map objects to positions on ring
- actual location of object is the node closest to object's position on ring in clockwise order

Chord

DB is distributed across several nodes

- each node stores only a portion of the DB



Given n-bit IDs ordered on an ID ring

Each node is assigned an integer ID from the range $[0, 2^n-1]$

Each key is hashed to an integer ID in the same range $[0, 2^n-1]$

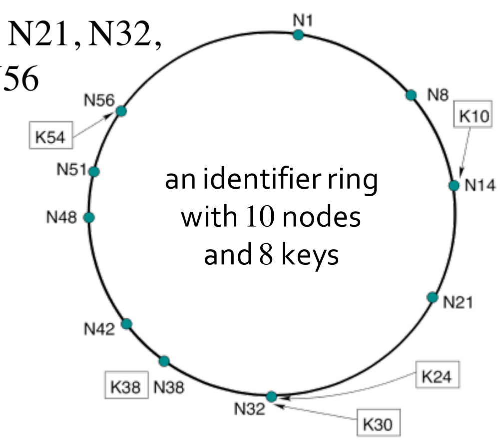
DB entry of a given key is stored at the smallest (or $=$) node ID following the ID the key hashes to $(\bmod 2^n)$

Chord

DB entry of a given key is stored at the smallest (or $=$) node ID following the ID the key hashes to $(\bmod 2^n)$

Example: $n = 6$ bits,
node IDs: N1, N8, N14, N21, N32, N38, N42, N48, N51, N56

$\text{hash}(\text{key1}) = \text{K10} \Rightarrow \text{N14}$
$\text{hash}(\text{key2}) = \text{K54} \Rightarrow \text{N56}$
$\text{hash}(\text{key3}) = \text{K24} \Rightarrow \text{N32}$
$\text{hash}(\text{key4}) = \text{K38} \Rightarrow \text{N38}$
$\text{hash}(\text{key5}) = \text{K30} \Rightarrow \text{N32}$
$\text{hash}(\text{key6}) = \text{K58} \Rightarrow \text{N?}$
$\text{hash}(\text{key7}) = \text{K15} \Rightarrow \text{N?}$
$\text{hash}(\text{key8}) = \text{K1} \Rightarrow \text{N?}$



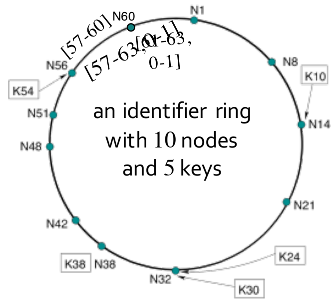
[Stoica+'03]

Chord: Basic Construction

Each node knows only the neighbors immediately behind (predecessor) and ahead (successor) of it, creating an overlay network

New node takes over keys in its identifier space from its successor

- N1 is responsible for IDs [57-63, 0-1]
- if a new node N60 joins the network, it takes over IDs [57-60] from N1
- and N1 is left with IDs [61-63, 0-1]



Departing node returns its key range to its successor

- when N60 leaves, N1 reclaims its original range of [57-63, 0-1]

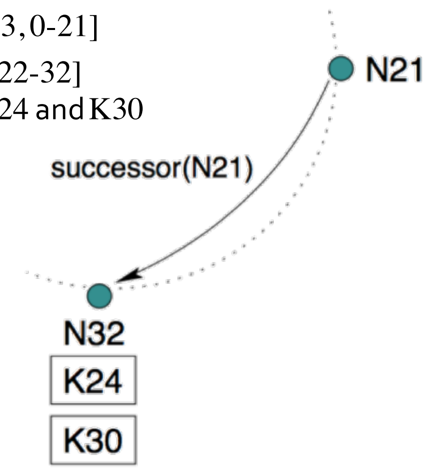
[Stoica+'03]

Chord: Adding a Node

Another example; let $n = 6$ bits

Assume there are only 2 nodes on the identifier ring

- N21 is responsible for IDs [33-63, 0-21]
- and N32 is responsible for IDs [22-32]
  two items are stored at N32: K24 and K30

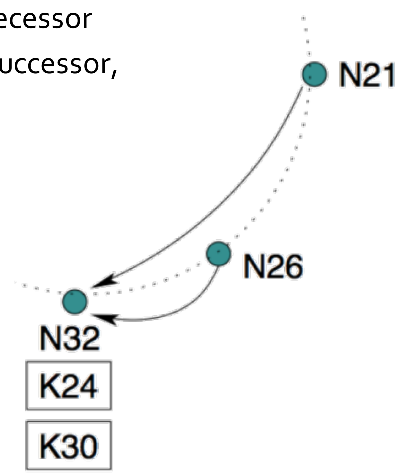


[Stoica+'03]

Chord: Adding a Node

A new node N26 joins the DHT at node N21

- N21 forwards it to N32, why?
- N32 accepts N26 as its new predecessor
- N32 informs N26 that N32 is its successor, N21 its predecessor



[Stoica+'03]

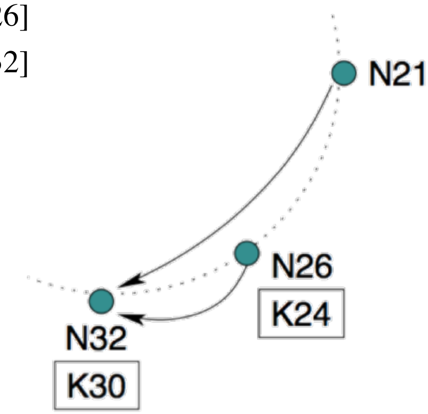
Chord: Adding a Node

N26 has N32 as its successor
(and N21 as its predecessor, not shown):

- N26 is responsible for IDs [22-26]
- N32 is responsible for IDs [27-32]
- item K24 is migrated to N26

But:

- N21 still has N32 as successor



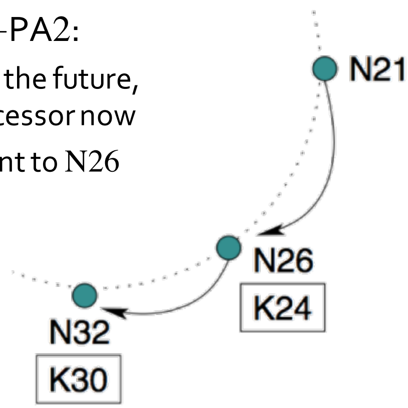
[Stoica+'03]

Chord: Adding a Node

Immediate predecessor and periodic fingers stabilization in Chord (`lookup()` always undershoot)

On-demand/lazy fix in Lab4+PA2:

- when contacted by N21 again in the future, N32 tells N21 that N26 is its successor now
- N21 updates its successor to point to N26
- N21 remains responsible for IDs [33-63, 0-21] throughout



[Stoica+'03]

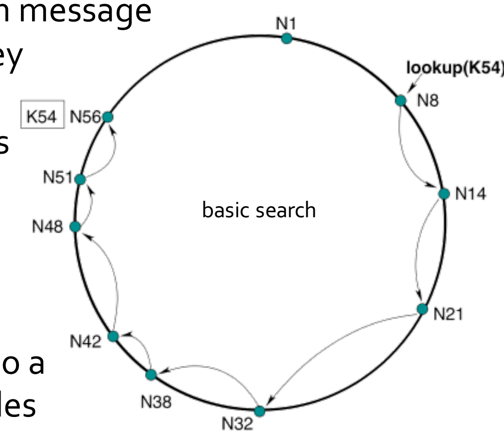
Chord: Basic Search

Given a key, route search message towards node holding key

Each node only knows its immediate successor

Example: **lookup(K54)**

It takes $O(N)$ time(!) to do a search, N number of nodes



[Stoica+'03]

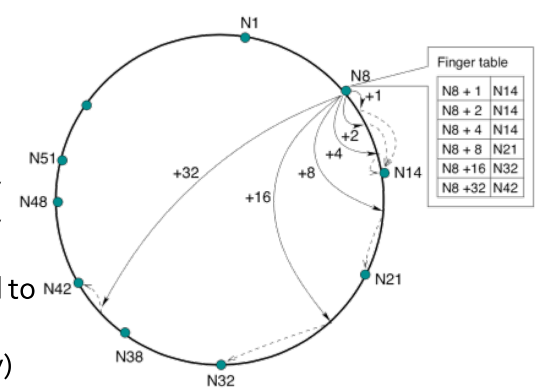
Chord: Finger Table Construction

Each node i knows of its successor and the nodes responsible for ID $i+2^k$ ($0 \leq k \leq 5$, for example)

- these nodes are kept in its finger table

Example: the finger table of N8 consists of:

- 8+1: at successor, N14
- 8+2: at successor, N14
- 8+4: at successor, N14
- 8+8: query N14 ⇒ N21
- 8+16: query N21 ⇒ N32
- 8+32: query N32 ⇒ N42 (from N32's finger table)
- in other cases, may need to query multiple nodes (recursively or iteratively)



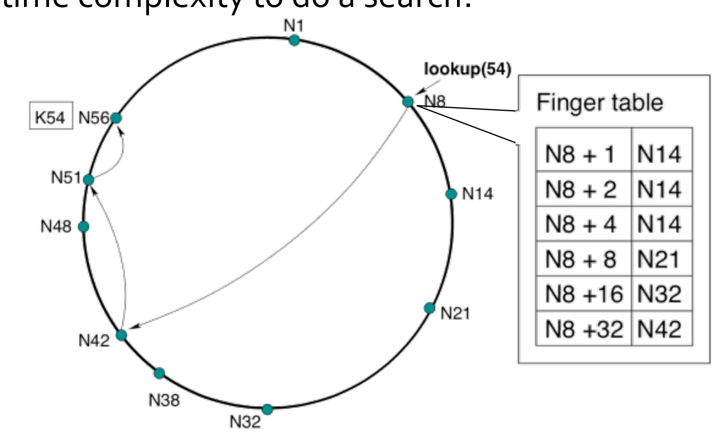
[Stoica+'03]

Chord: Search with Finger Table

Example: **lookup(K54)**

What is the finger table of N42, assuming $n = 6$ bits?

What is the time complexity to do a search?

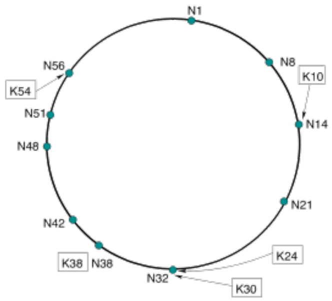


[Stoica+'03]

Chord: Node Failure

Each node must know both its immediate and subsequent successors
- sends periodic keep-alive pings

If ping fails, obtain new successor (new successor assumes ID range of old successor)



Example:
- N1 has N8 as immediate successor and N14 as subsequent successor
- if N8 fails, N1 makes N14 its immediate successor, and queries N14 for its immediate successor
- if N14 fails, N1 queries N8 for its new immediate successor

Inbound fingers fixed lazily

[Stoica+'03]

Storage Models

DHT can be used as "content-addressable network"

Where to backup the values of a node? Alternatives:
- only at the node's immediate successor in the identifier ring
  - immediate successor assumes node's ID range in case of failure
  - churn, routing issues, packet loss make lookup failure more likely
- on k successor nodes
  - when nodes detect successor/predecessor failure, replicate further
- cached along reverse lookup path
  - cache consistency and dynamic content issues
  - query and reply must both be recursive

Limitations of Consistent Hashing

Limited to <key, value> pair search
(What other kinds of search might you want to do?)

High overhead at node arrivals and departures

Complicated node failure recovery and topology maintenance

Suffers from "hot-spots" due to keyword-to-node mapping
- popular keywords concentrate traffic on a few nodes
- cannot spread load associated with a single keyword across multiple nodes