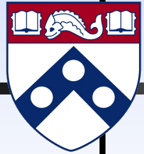


CIS 5530: Networked Systems

Chord

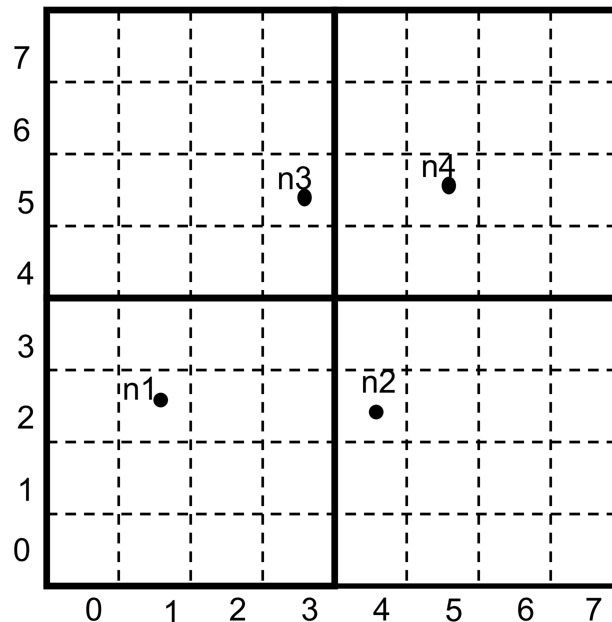
March 13, 2023

Agenda

- Overlay Networks
  - DHTs
  - Content Addressable Networks
  - Chord NEXT
  - PennSearch

Review

- Define an “overlay network”
- Consider a node that wants to join a CAN. It joins with the “landmark” n3 and wants to join at (6,1)
  - What is the sequence of messages it needs to send?



Chord

- Associate to each node and item a unique id in an uni-dimensional space $0..2^m-1$
- Key design decision
  - Decouple correctness from efficiency
- Properties
  - Routing table size O(log(N)) , where N is the total number of nodes
  - Guarantees that a item is found in O(log(N)) steps using its key

Review: 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? $n \rightarrow n-1$
  - Why might this be bad?

Consistent Hash [Karger 97]

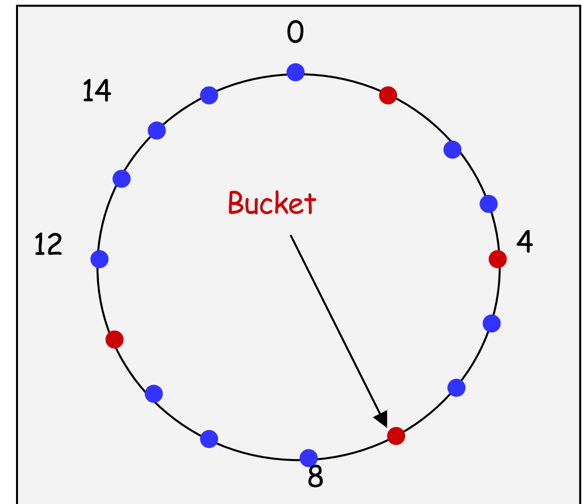
- David Karger, et al. Consistent Hashing and Random Trees: Tools for Relieving Hot Spots on the World Wide Web. STOC ’97.

- Used by Akamai CDN for load balancing

- Desired features
  - Balanced – load is equal across buckets
  - Smoothness – little impact on hash bucket contents when buckets are added/removed

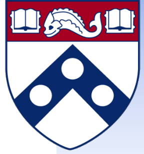
Consistent Hash [Karger 97]

- Construction
  - Assign each of C hash buckets to random points on mod 2n circle, where, hash key size = n.
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket



- Smoothness → addition of bucket does not cause movement between existing buckets

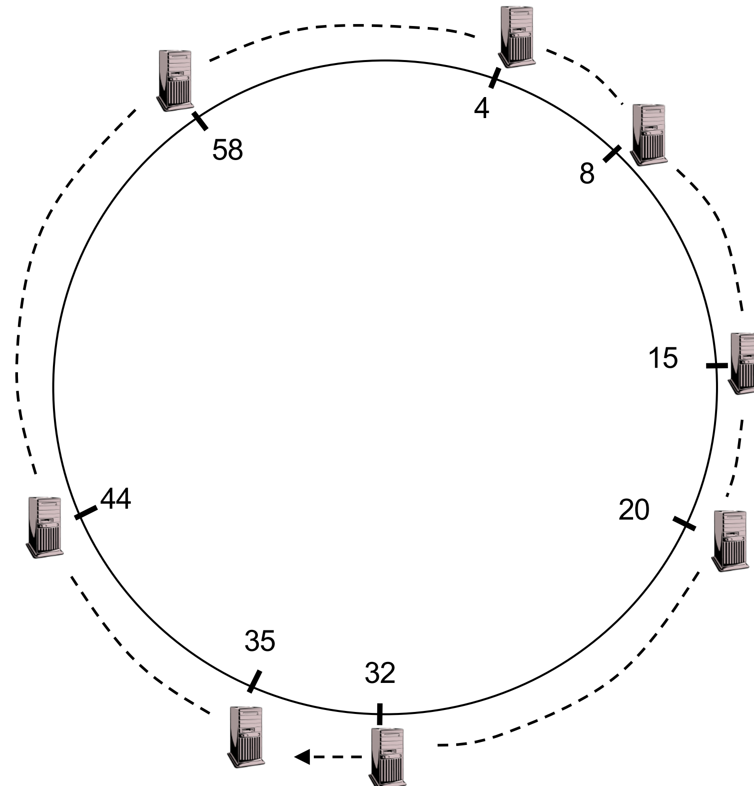
- Balance → no bucket is responsible for large number of objects

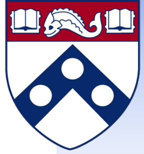


Identifier To Node Mapping Example

- Node 8 maps [5,8]
- Node 15 maps [9,15]
- Node 20 maps [16, 20]
- …
- Node 4 maps [59, 4]

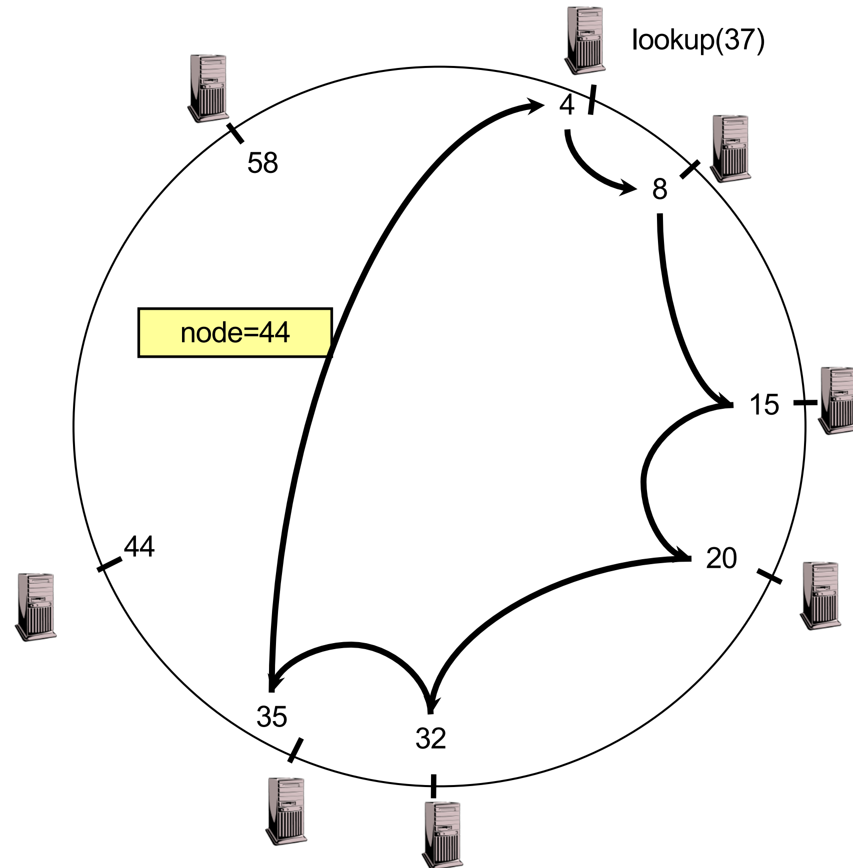
- Each node maintains a pointer to its successor





Lookup

- Each node maintains its successor
- Route packet (ID, data) to the node responsible for ID using successor pointers

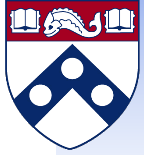


Joining the Ring (from paper)

Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications: https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

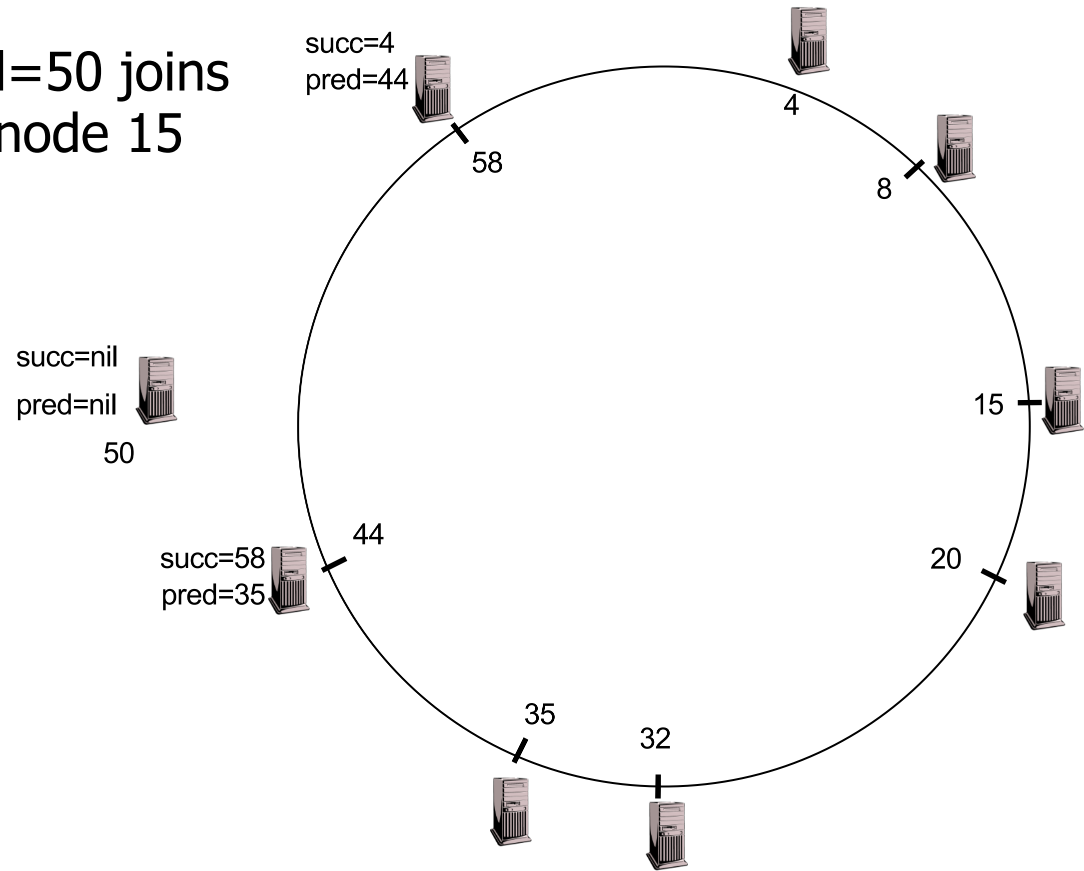
```
// create a new Chord ring.
n.create()
        predecessor = nil;
        successor = n;

// join a Chord ring containing node n'.
n.join(n')
        predecessor = nil;
        successor = n'.find_sucessor(n);
```

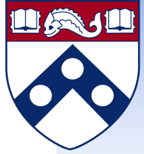
INSERT code to transfer items from successor to itself.



Joining Operation

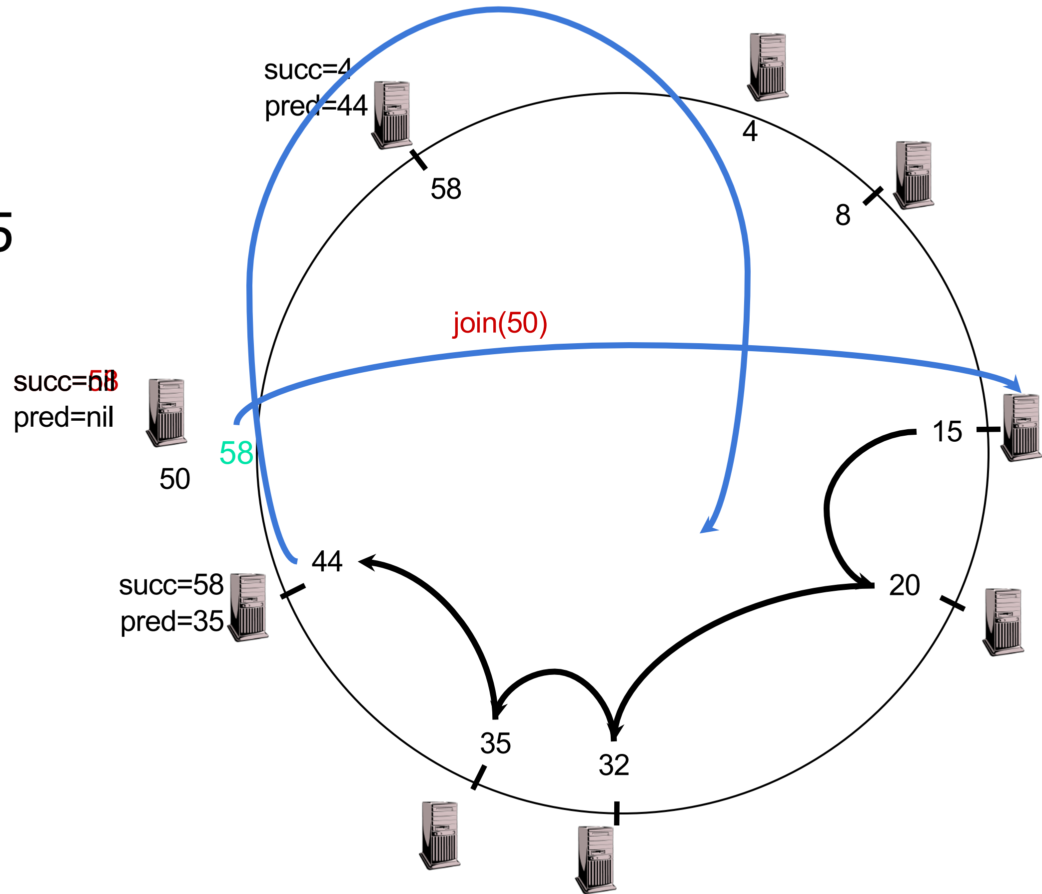
- Node with id=50 joins the ring via node 15





Joining Operation

- Node 50: send join(50) to node 15
- Node 44: returns node 58
- Node 50 updates its successor to 58



Ring Stabilization (from paper)

```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
        x = successor.predecessor;
        if (x  ∈ (n,successor))
                successor = x;
        successor.notify(n);
```

Ask my succ for its current pred.

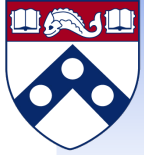
If succ.pred is better than my current successor, switch to new successor

Informs new successor

```
// n' thinks it might be our predecessor:
n.notify(n')
        if (predecessor is nil or n' ∈  (predecessor, n))
        predecessor = n';
```

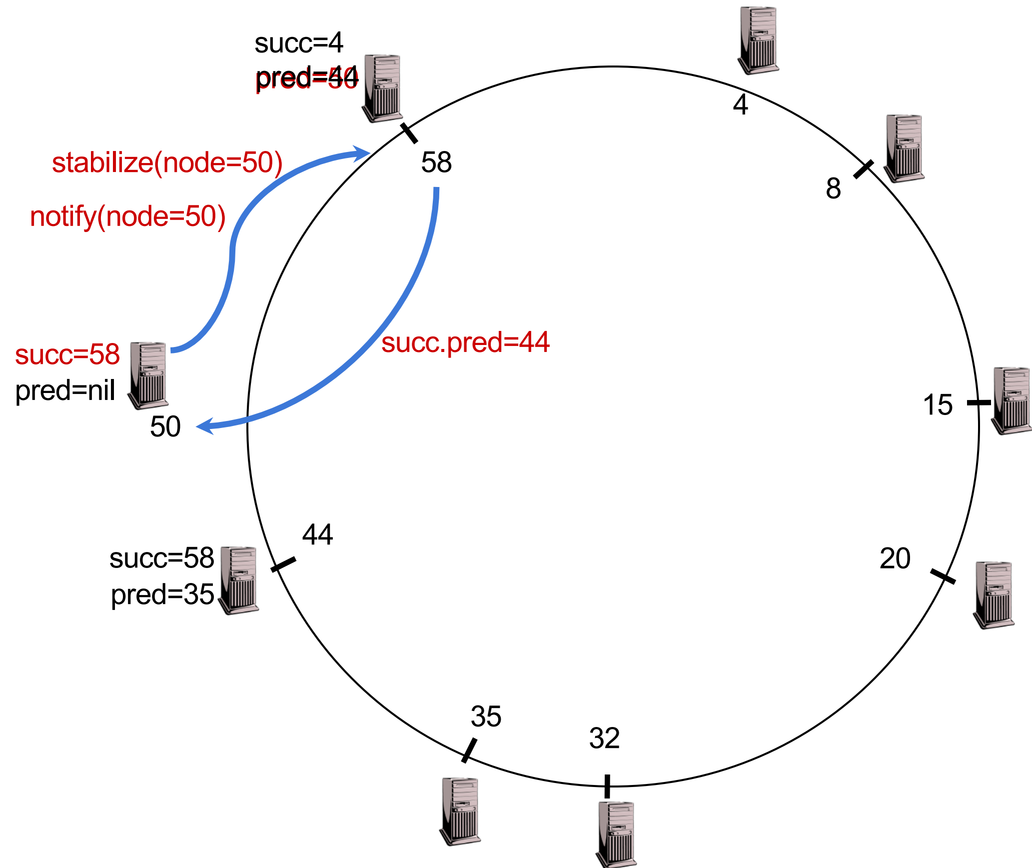
n found a better (closer) pred

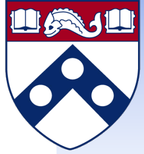
In the steady converged state, stabilization maintains the invariant n.successor.predecessor=n.



Periodic Stabilize

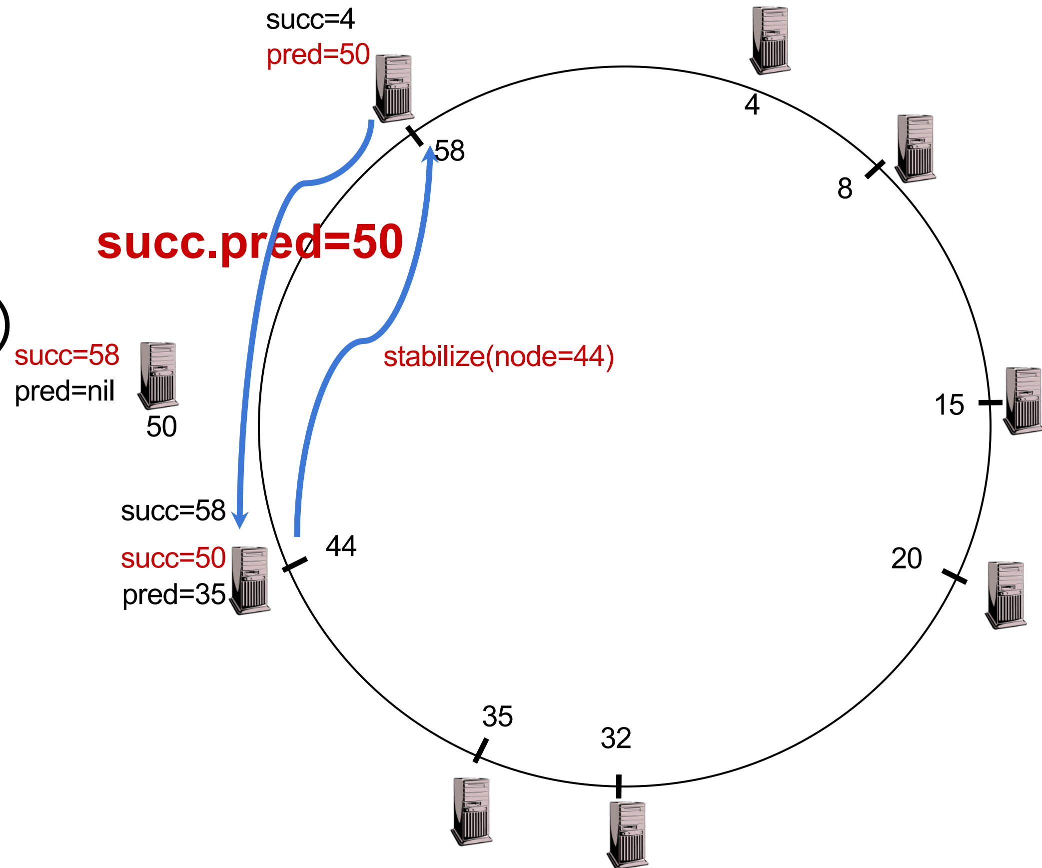
- Node 50: periodic stabilize
  - Sends stabilize message to 58
- Node 50: send notify message to 58
  - Update pred=50

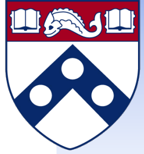




Periodic Stabilize

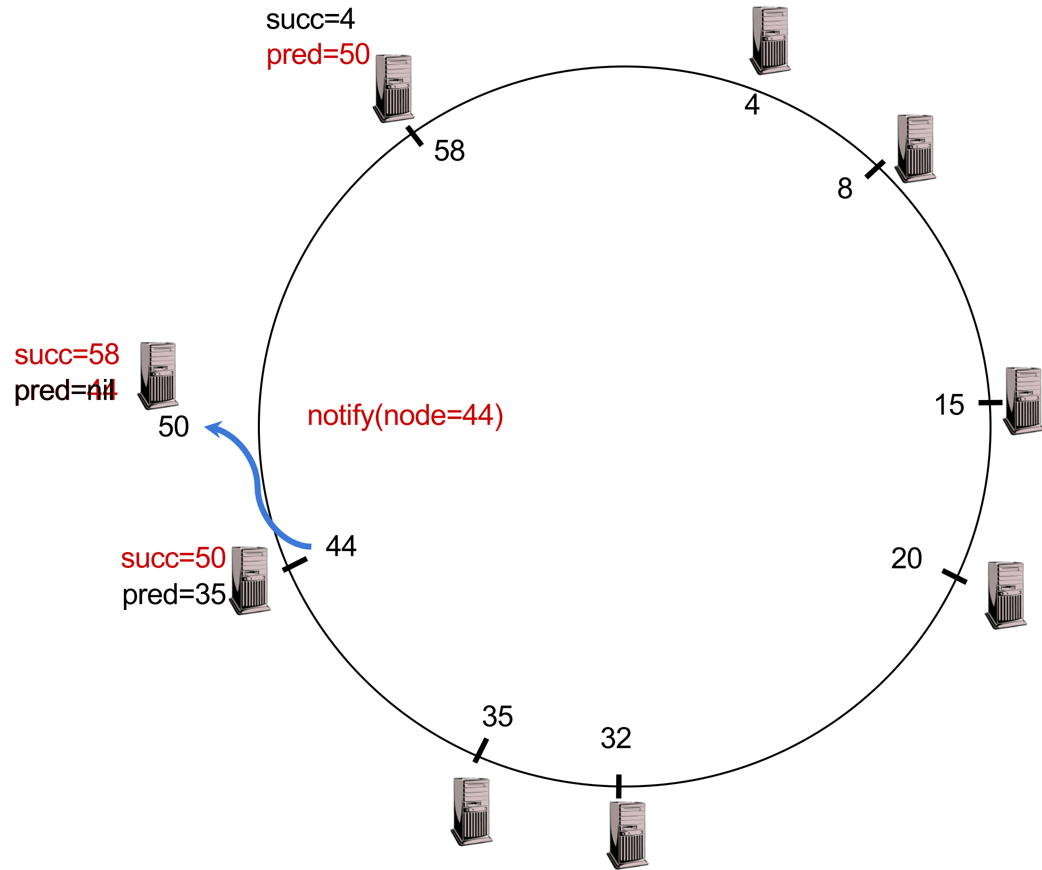
- Node 44: periodic stabilize
- Asks 58 for pred (50)
- Node 44 updates its successor to 50

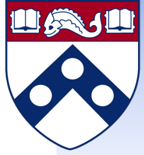




Periodic Stabilize

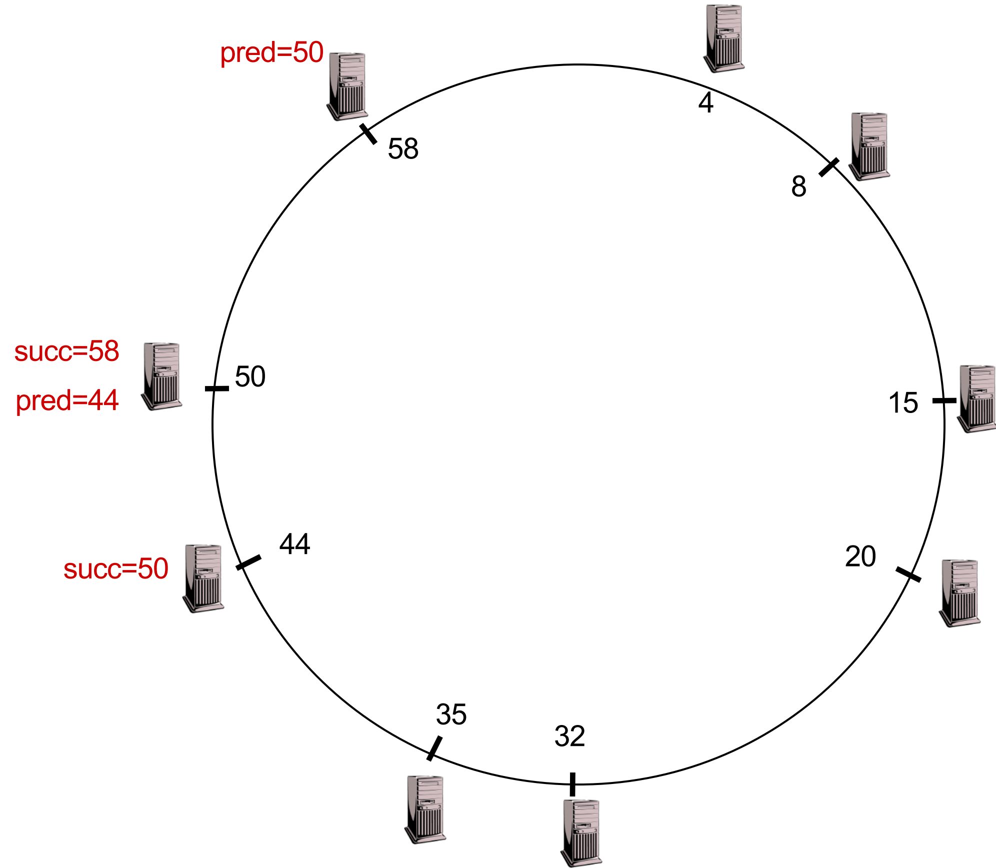
- Node 44 has a new successor (50)
- Node 44 sends a notify message to node 50





Periodic Stabilize Converges!

This completes the joining operation!

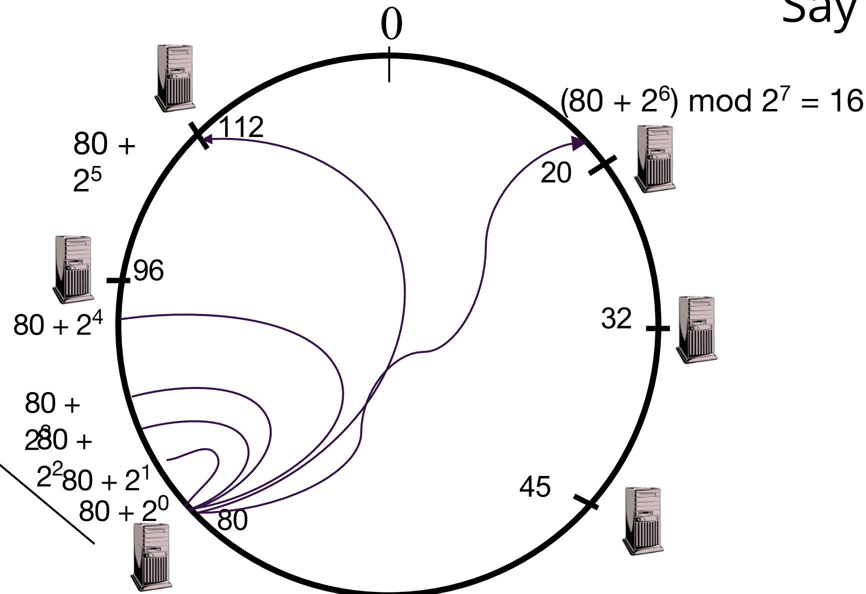


Achieving Efficiency: *Finger Tables*

Finger Table at 80

i	*ft[i]*
0	96
1	96
2	96
3	96
4	96
5	112
6	20

Say *m*=7



ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

Finger "Fixing" (from paper)

```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix
n.fix_fingers()
        next = next + 1;
        if (next > m)
             next = 1;
        finger[next] = find successor(n + 2 next-1);
```

Lookup With Fingers (from paper)

```
// ask node n to find the successor of id
n.find_successor(id)
        if (id ∈ (n,successor])
                return successor;
        else
                n' = closest_preceding_node (id);
                return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
        for i = m downto 1
                if (finger[i] ∈ (n, id))
                        return finger[i];
        return n;
```

Achieving Robustness

- Ring robustness: each node maintains the k (> 1) immediate successors instead of only one successor
  - If smallest successor does no respond, substitute the second entry in its successor list
  - Unlikely all successors fail simultaneously
- Modifications to stabilize protocol (see Chord paper!)
- Lookup robustness:
  - Hop-by-hop reliability (along each call to `find_successor`)
  - End-to-end reliability (how?)

DHT Optimizations

- Reduce latency
  - Chose finger that reduces expected time to reach destination ("route selection")
  - Chose the closest node from range [N+2i-1,N+2i) as ith finger ("proximity neighbor selection")
- Accommodate heterogeneous systems
  - Multiple virtual nodes per physical node

What Is the Right DHT API?

- Subject of much debate
- Isn't it a hash table?
  - Item ← get(k)
  - put(k, item)
- Consensus today:
  - Addr ← lookup(k)
  - Much more general API. DHT is a key to location map.
- To store/retrieve string pairs:
  - Compute k = hash(str1)
  - Addr ← lookup(k)
  - Send STORE to Addr asking to store <str1, str2>
  - Send RETRIEVE to Addr asking for all <str1, *>

Agenda

- Overlay Networks
  - DHTs
  - Content Addressable Networks
  - Chord
  - PennSearch

Goal Of Project 2

- Build PennChord
  - Finger tables (160-bit)
  - One successor and predecessor
  - Light churn: one node join/leave at any time
  - No node failures
  - One API: IPAddr <- lookup(K)
- PennSearch: DHT-based search engine
  - P2P application over Chord
  - Able to execute queries of the form “T1 AND T2 AND T3”

Components

Publish *inverted index* hashed by keyword, issue search queries (hash on multiple keywords)

PennSearch Node		PennSearch Node
Search Engine	⟷	Search Engine
Addr ⟵ lookup(K)		Addr ⟵ lookup(K)
Chord Overlay	⟷	Chord Overlay

Chord Protocol, e.g. Stabilization, finger fixing. Uses UDP for communication.

PennSearch Node

PennSearch Node

Where does your Project 1 code get used?

Scenario File

```
TIME 120000                     [Gives time for routing protocol in project 1 to converge]
0 PENNSEARCH CHORD JOIN 0       [Node 0 starts Chord ring]
TIME 10000
1 PENNSEARCH CHORD JOIN 0       [Node 1 joins Chord ring via node 0]
TIME 10000
2 PENNSEARCH CHORD JOIN 1       [We have Chord overlay of size 3.]
TIME 10000
0 PENNSEARCH PUBLISH ./upenn-cis553/keys/metadata0.keys  [Node 0 publishes inverted index]
TIME 10000
1 PENNSEARCH PUBLISH ./upenn-cis553/keys/metadata1.keys [Node 1 publishes inverted index]
TIME 10000
0 PENNSEARCH SEARCH 0 T1 T2  [Node 0 issues a search for documents matching “T1” and “T2”.]
TIME 10000
1 PENNSEARCH SEARCH 1 T2     [Node 1 issues a search for documents matching “T2”.]
TIME 10000
4 PENNSEARCH SEARCH 2 T10    [Node 4 issues a search via 2 for documents matching “T1”.]
TIME 10000
1 PENNSEARCH CHORD LEAVE     [Node 1 leaves the ring, no data must be lost]
TIME 10000
2 PENNSEARCH SEARCH 2 T12 T33 T14  [Same results obtained even if 1 did not leave the ring.]
```

Format Of Input Data For Publishing

- metadata0.keys published by node 0
  - doc0 T1 T2 T3 T4
  - doc1 T2 T3 T4 T5
  - doc2 T3 T3 T4 T5 T6
  - doc3 T7 T8 T9 T10
  - doc4 T1 T2 T4 T5

- Each line represents keywords for each document, identified by docID (which can be a URL or catalog number, etc).

Format Of Input Data For Publishing

- This is post-extraction, i.e., we assume there is a separate process that gets the raw documents (e.g. web pages), and generate each line above.
- You may assume no nodes publish for the same documents.
- For “T1 and T2”, our search results turns all matching document identifiers. Downloading actual content is out of scope of this project.

Centralized Search

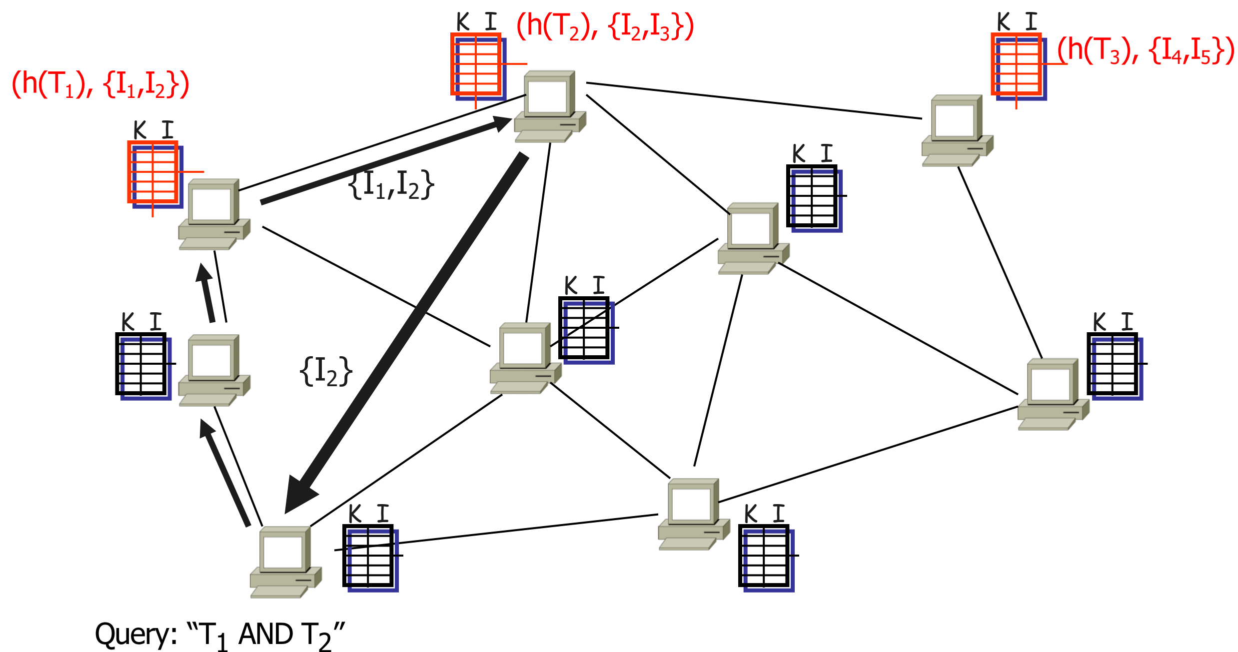
- Inverted lists, one for each keyword.
- For example:
  - T1, {Doc0, Doc4}
  - T2, {Doc0, Doc1, Doc2, Doc4}
  - T3, {Doc0, Doc1, Doc2}
- To solve "T1 and T2", take the inverted list T1 and intersect with inverted list of T2:
  - {Doc0, Doc4} intersects {Doc0, Doc1, Doc2,Doc4} = {Doc0,Doc4}

Centralized Search

- Query results: {Doc0,Doc4}
- Typically, another retrieval to fetch raw data for Doc0 and Doc4, but out of scope of our project.

DHT-based Keyword Search

- Inverted Lists hashed by keyword (term) in the DHT



Step-by-Step Publishing (Node 0)

- Read metadata file, e.g. metadata0.keys
- For each keyword, generate inverted list. E.g.
  - T1, {Doc0, Doc4}
  - T2, {Doc0, Doc1, Doc2, Doc4}

Step-by-Step Publishing (Node 0)

- For each inverted list:
  - S = chord_lookup(hash(T1))
  - Node 0 sends inverted list for T1 to S via a PennSearch message over UDP
  - S receives the list. If it already has a list for T1, it unions its original list with the new one. E.g. {A,B} union {B,C} = {A,B,C}
  - Repeat for all keywords

Step-by-Step Query Execution

- Consider node 4 issues a search query for “T1” and “T2” via PennSearch node 0.
  - Node 4 contacts node 0 with query “T1 AND T2”.
  - Node 0
    - issues N_T1 = chord_lookup(hash(T1)). [An asynchronous call]
    - Sends query to node N_T1.
  - Node N_T1
    - Retrieves inverted list for T1 {Doc0, Doc4}
    - Computes N_T2 = chord_lookup(hash(T2)). [An asynchronous call]
    - Sends {Doc0,Doc4} to N_T2.
  - Node N_T2
    - Retrieves inverted list for T2 {Doc0, Doc1, Doc2, Doc4}.
    - Computes intersection results {Doc0,Doc4}
    - Sends results direcly back to 4 (or to 0, which forwards to 4).

PennSearch

- Generalize search to any arbitrary number of search terms.

- If no results along intermediate nodes, return “no results” to query node.

- Some reminders:
  - Following our logging conventions
  - Start early
  - Test in piecemeal. Get Chord working first without fingers, add fingers, add nodes leaving, add single term search, do multiple search terms, run across multiple nodes, etc.