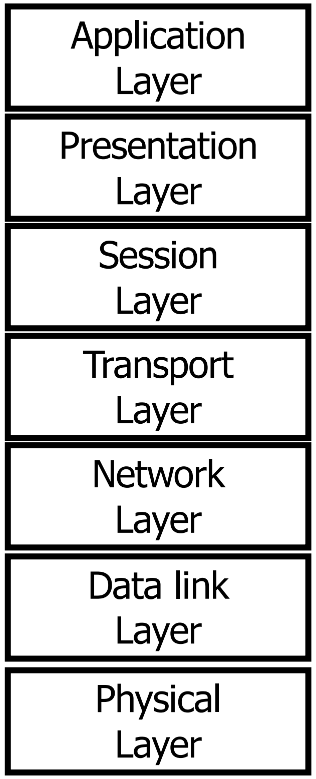
Technische Universität Berlin

Computer Networks

Distributed Hash Tables

Chapter

Top-Down-Approach



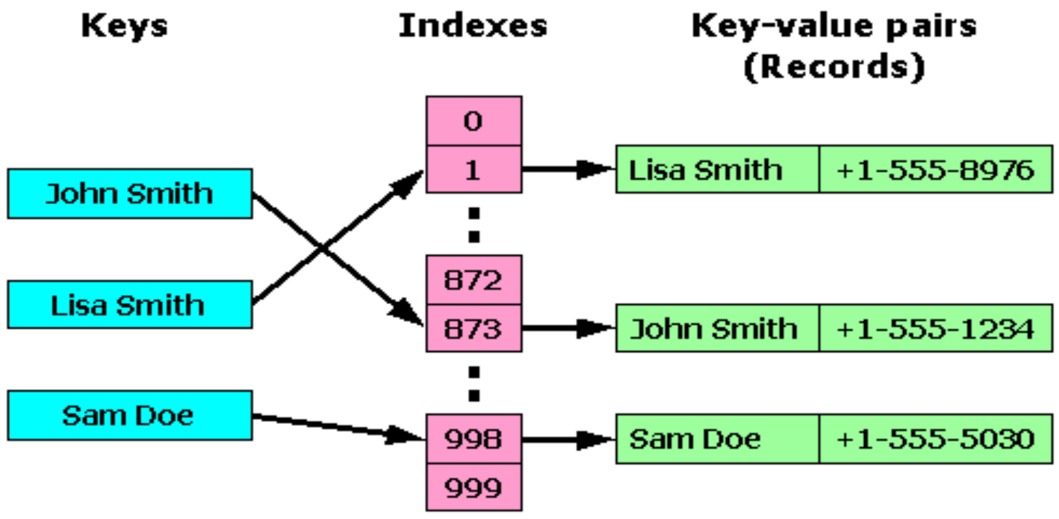
Distributed Hash Tables (DHT)

Reminder

- We have discussed the mapping: **name → address**
- The solution has been DNS
  - Hierarchical organization
  - Distributed
  - Using redundancy
- Challenge:
  - How to provide storage of pairs like the above one in a distributed way, even if there is no strong hierarchy?

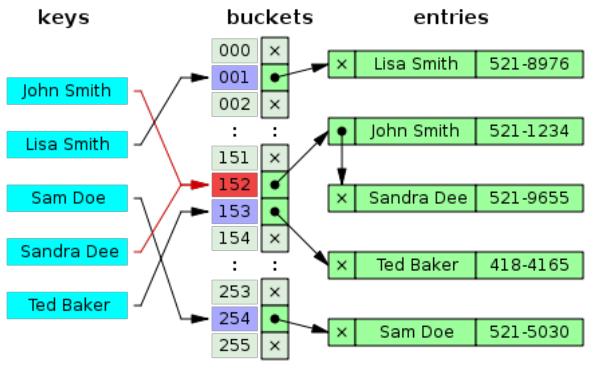
Hash Table

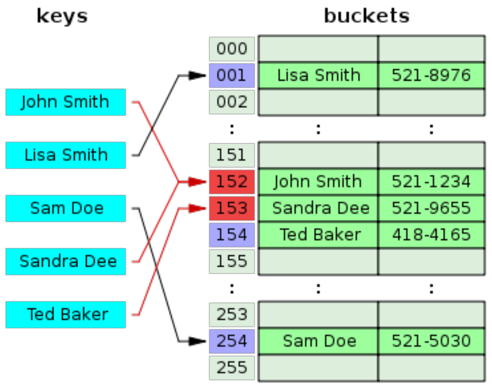
- Items: [Key, Value] are stored
- The key is hashed, i.e., transformed (using a hash function) so that the result – the **hash** – can be used to locate a **bucket** in which the pair is stored.
- The bucket is identified by an **index**.
- The bucket might contain multiple such items (pairs)

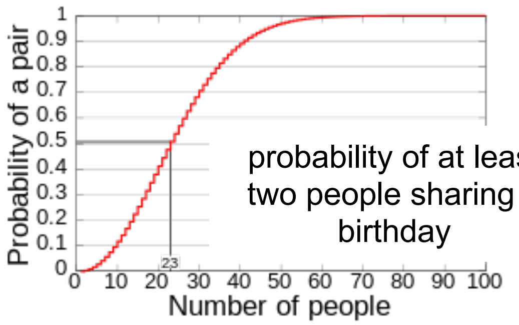


Hash Table - Collision Resolution

- **Hash collisions** are practically unavoidable when hashing a random subset of a large set of possible keys.
  - E.g., if 2450 keys are hashed into 10^6 buckets, even with a perfectly uniform random distribution there is a ~95% chance of at least two of the keys being hashed to the same slot → **birthday paradox**
- Widely used collision resolution strategies:
  - **Separate chaining**: each bucket is independent & has some sort of list of entries with the same index (left)
  - **Open addressing**: all entry records are stored in the bucket array itself & usage of probe sequence (middle)





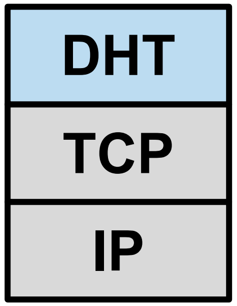


probability of at least two people sharing a birthday

src: wikipedia

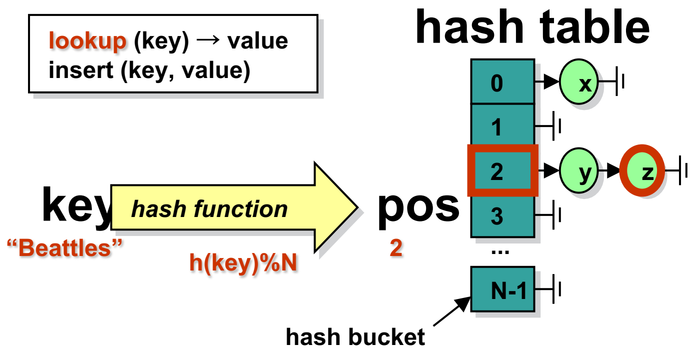
A Distributed Hash Table (DHT)

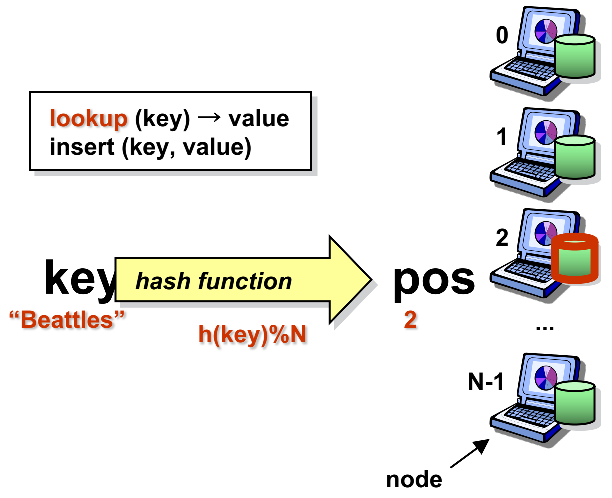
- Remember the mapping of names to IP addresses?
  Could we use hash tables? Remember the scaling issue …
- **Distributed Hash Tables** (DHT) spread the pairs across a number of computers (buckets) located arbitrarily across the world.
  - Note: copies of a single pair can be stored in one or in multiple locations!
- When a user queries the system, i.e., provides the key, the system uses the hash to find the pair from one of the computers where it's stored and returns the result.
- All the nodes are assumed to be reachable by some kind of unicast communication.
- DHT posses the features of **scaling, robustness, self-organization**.



Hash Table vs. DHT

- The key is hashed to find the proper bucket in a hash table
- In a Distributed Hash Table (DHT), nodes are the hash buckets
  - Key is hashed to find the responsible Node
  - Pairs are distributed among the nodes with respect to load balancing





[Khalifeh, op. cit.]

DHT Interface

- Minimal interface (data-centric)
  - **lookup(key) → value**
  - **insert(key, value)**
  - **delete(key)**
- Supports a wide range of applications, because few restrictions
  - Value is application dependent
  - Keys have no semantic meaning
- **Note**: DHTs do **not** have to store data useful to end users, e.g., data files … data storage can be built on top of DHTs

DHTs: Problems

- Problem 1 (dynamicity): adding or removing nodes
  - With hash mod M (= no. of nodes), virtually every key will change its location!

$$h(k) \bmod m \neq h(k) \bmod (m+1) \neq h(k) \bmod (m-1)$$

- Solution: use consistent hashing
  - Define a fixed hash space
  - All hash values fall within that space and do not depend on the number of peers (hash bucket)
  - Each key goes to peer closest to its ID in hash space (according to some proximity metric)
- Problem 2 (size): all nodes must be known (in order to insert or lookup items!)
  - Works with *small* and *static* server populations
- Solution: each peer knows of only **a few "neighbors"**
  - Messages are routed through neighbors via multiple hops

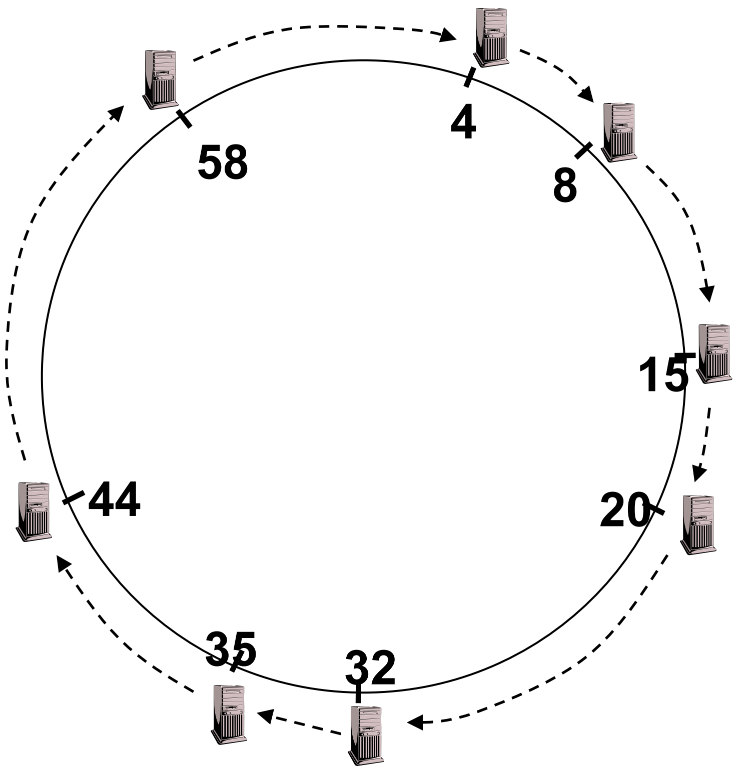
[Felber, op. cit.]

Chord: Identifier to Node Mapping

- Chord is a DHT developed in 2001

Chord: A scalable peer-to-peer lookup service for internet applications
I Stoica, R Morris, D Karger, MF Kaashoek... - ACM SIGCOMM ..., 2001 - dl.acm.org
A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents **Chord**, a distributed lookup protocol that addresses this problem. **Chord** provides support for just one operation: given a ...
☆ 99 Cited by 14275 Related articles All 325 versions

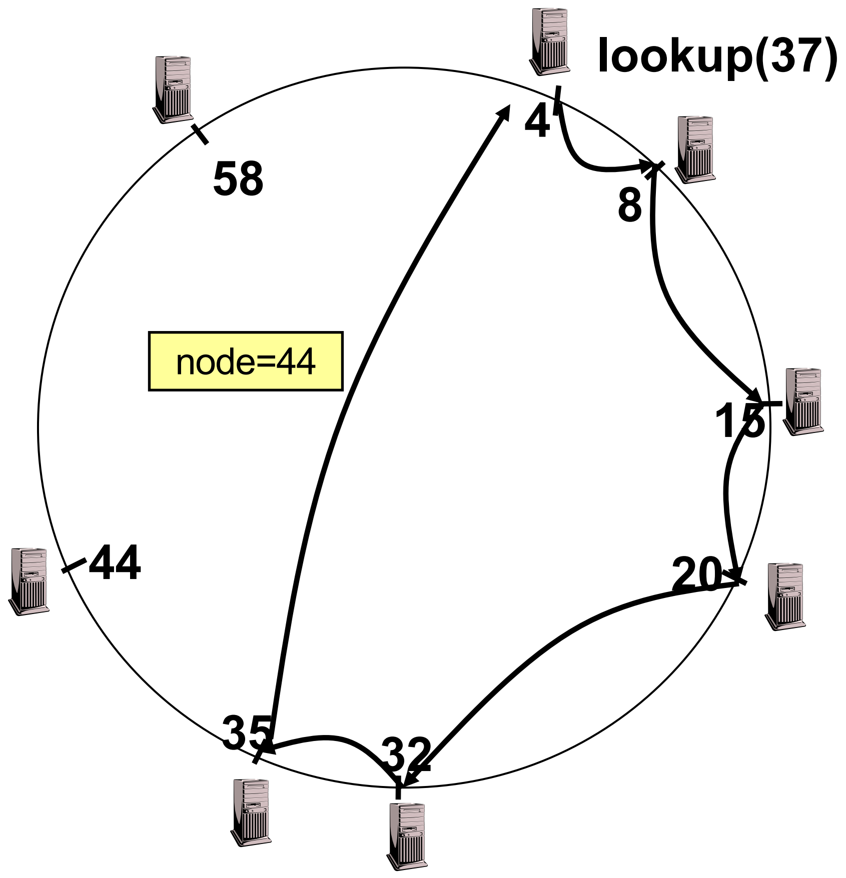
- Associate to each node and item a unique id in an unidimensional space 0 to 2^m-1
  - Node 8 maps [5, 8]
  - Node 15 maps [9, 15]
  - Node 20 maps [16, 20]
  - …
  - Node 4 maps [59, 4]
- Each node maintains a pointer to its successor



[Shenker, op. cit.]

Chord: Lookup

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

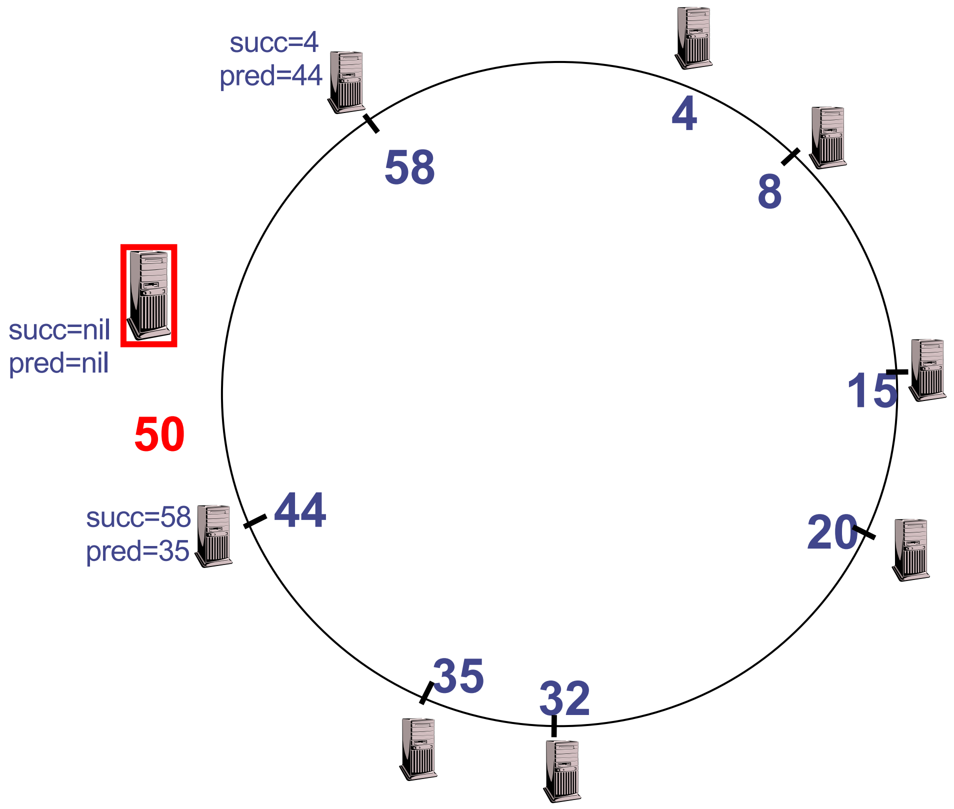


Chord: Joining Operation

- Each node A periodically sends a **stabilize()** message to its successor B
- Upon receiving a **stabilize()** message, node B
  - returns its predecessor B’= pred(B) to A by sending a **notify(B’)** message
- Upon receiving **notify(B’)** from B,
  - if B’ is between A and B, A updates its successor to B’
  - A doesn’t do anything, otherwise

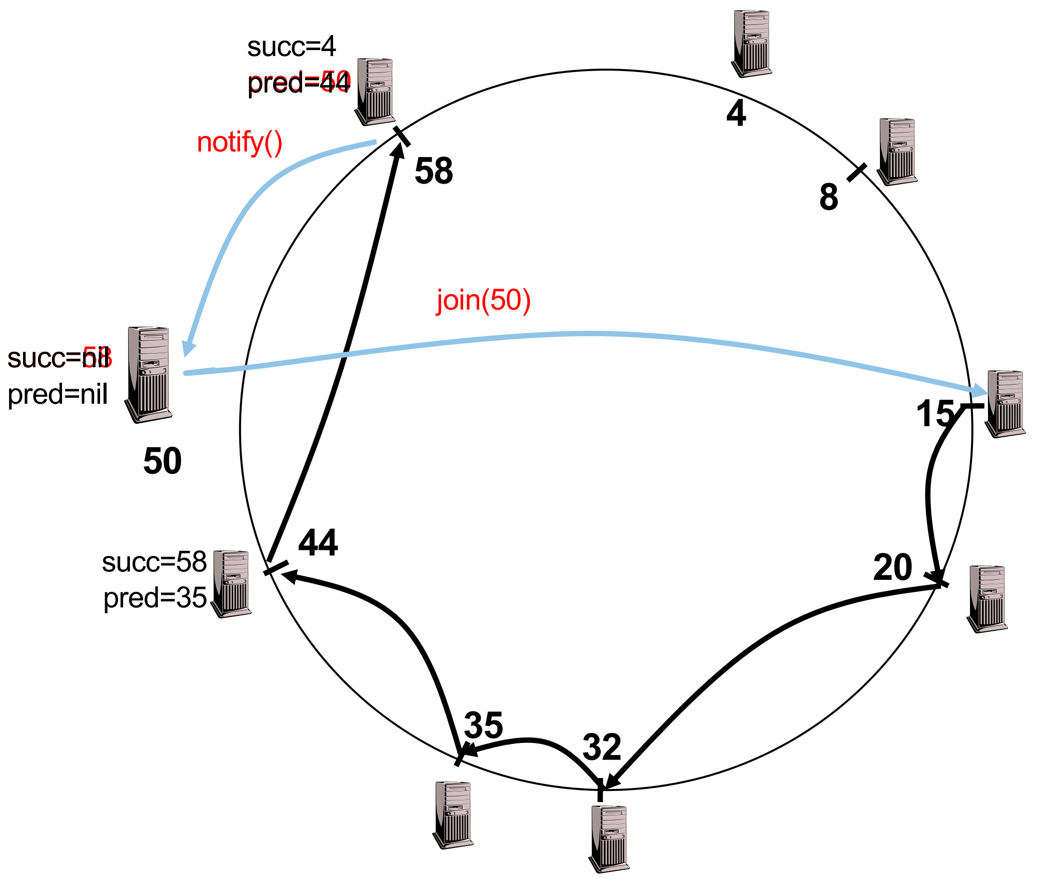
Chord: Joining Operation (II)

- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15



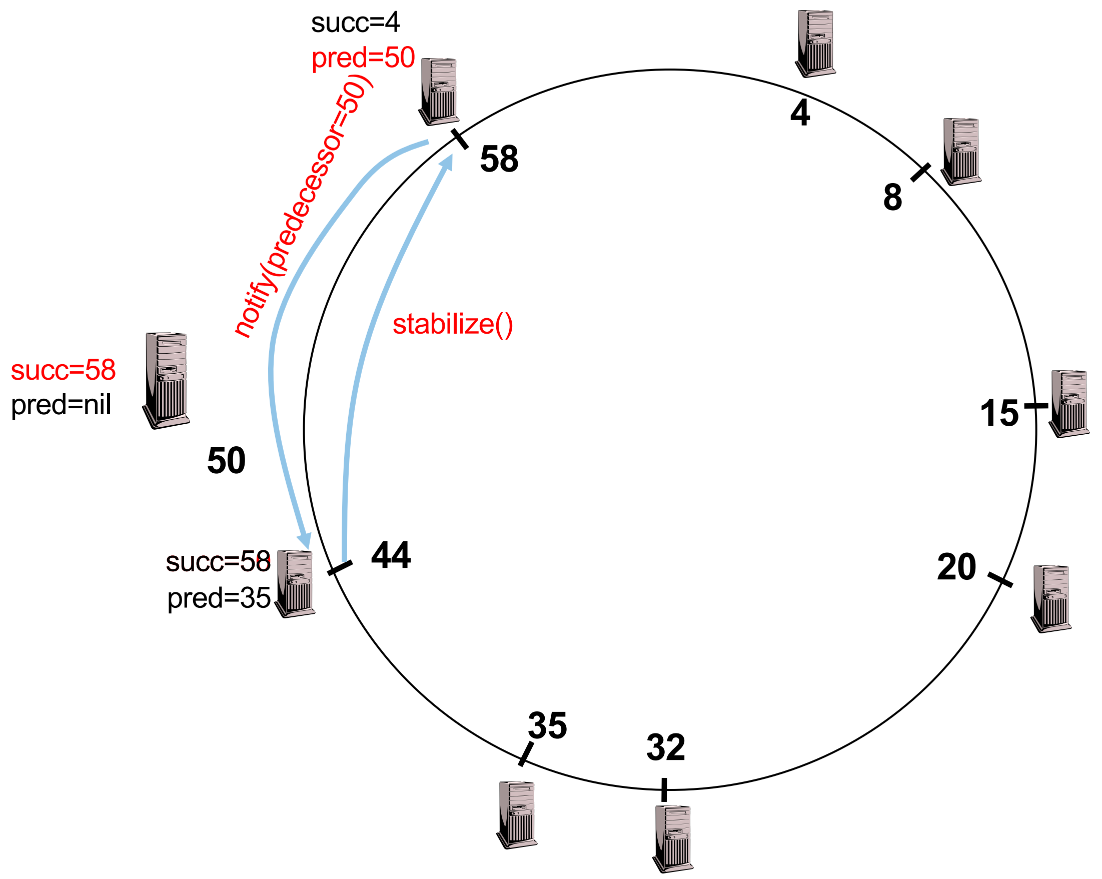
Chord: Joining Operation

- Node 50 asks node 15 to forward join message to successor(50) which is 58
- When join(50) reaches the destination (i.e., node 58), node 58
- updates its predecessor to 50,
- returns a notify message to node 50
- Node 50 updates its successor to 58



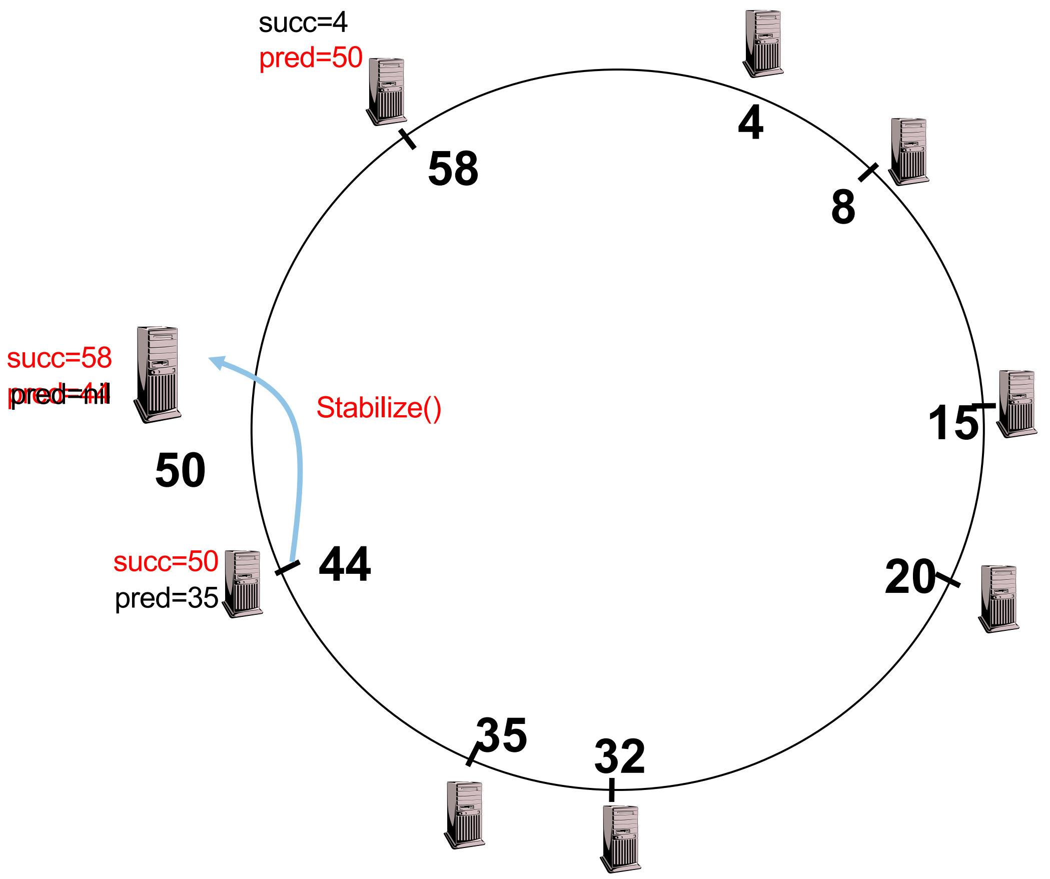
Chord: Joining Operation (cont'd)

- Node 44 sends a stabilize message to its successor, node 58
- Node 58 reply with a notify message
- Node 44 updates its successor to 50



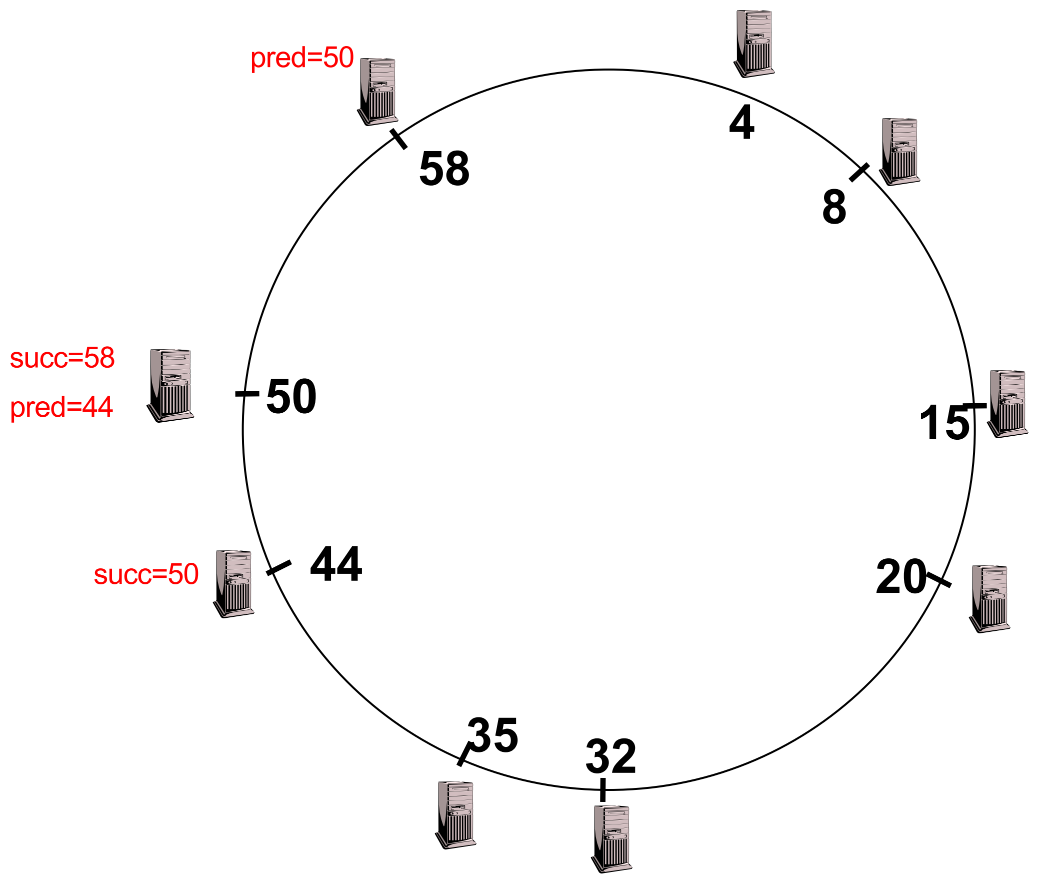
Chord: Joining Operation (cont'd)

- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44



Chord: Joining Operation (cont'd)

- This completes the joining operation!

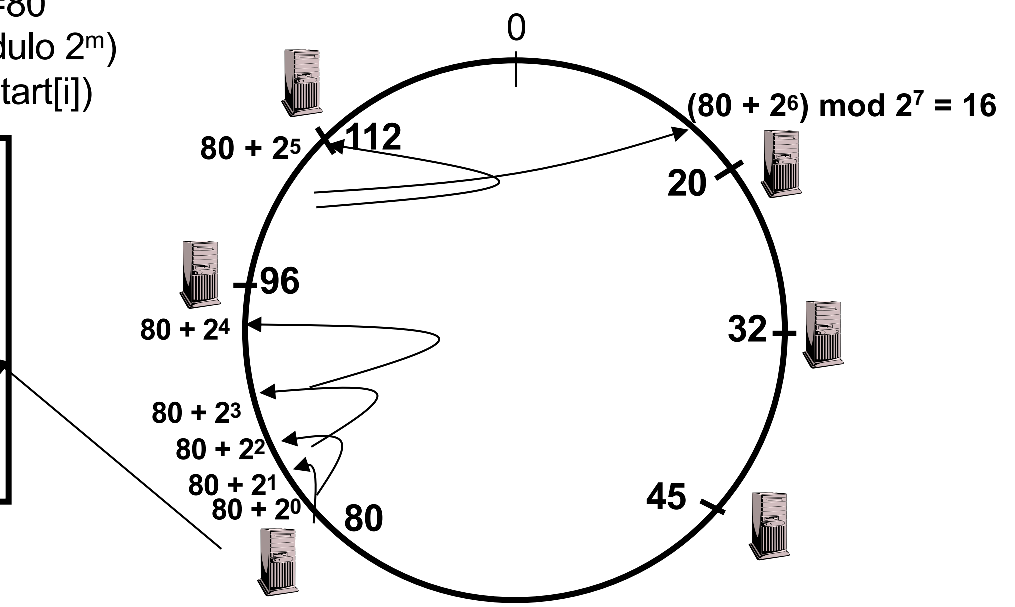


Achieving Efficiency: Finger Tables

Finger Table at k=80
start = k + 2^i (modulo 2^m)
ft[i] = successor(start[i])

i	*start*	*ft[i]*
0	81	96
1	82	96
2	84	96
3	88	96
4	96	96
5	112	112
6	16	20

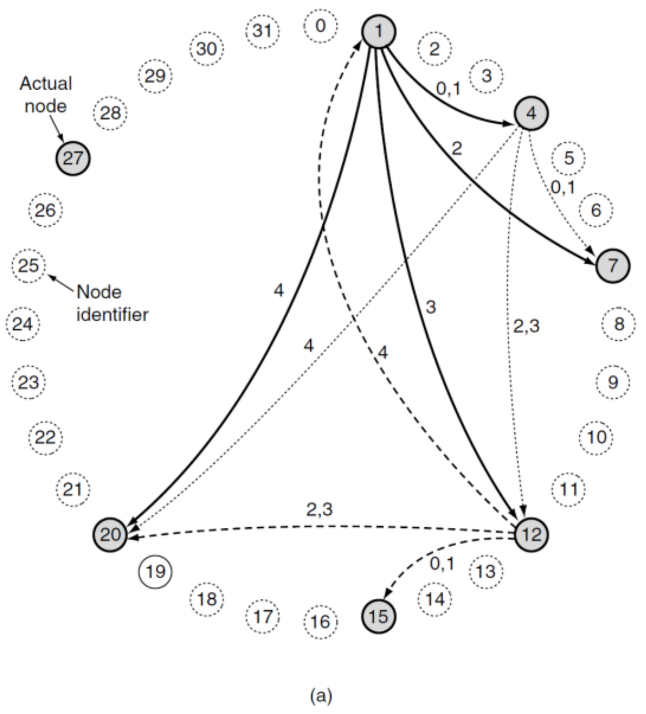
Say *m*=7



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

Lookups using Finger Tables (II)

- **Ex. 1**: key=3 at node 1:
  - Node 1 knows that 3 lies between it & its successor (4) → desired node is 4
- **Ex. 2**: key=16 at node 1:
  - Using FT we see that closest predecessor to 16 is 9 → request is **forwarded** to node 12
  - Node 12 uses FT to find out that closest predecessor to 16 is 14 → request is **forwarded** to node 15
  - Node 15 observes that 16 lies between it and its successor (20) → it returns address of node 20 to caller



Start	IP addr of successor	
2	4	Node 1's finger table
3	4	
5	7	
9	12	
17	20	

Start	IP addr of successor	
5	7	Node 4's finger table
6	7	
8	12	
12	12	
20	20	

Start	IP addr of successor	
13	15	Node 12's finger table
14	15	
16	20	
20	20	
28	1	

(b)

Figure 7-71. (a) A set of 32 node identifiers arranged in a circle. The shaded ones correspond to actual machines. The arcs show the fingers from nodes 1, 4, and 12. The labels on the arcs are the table indices. (b) Examples of the finger tables.

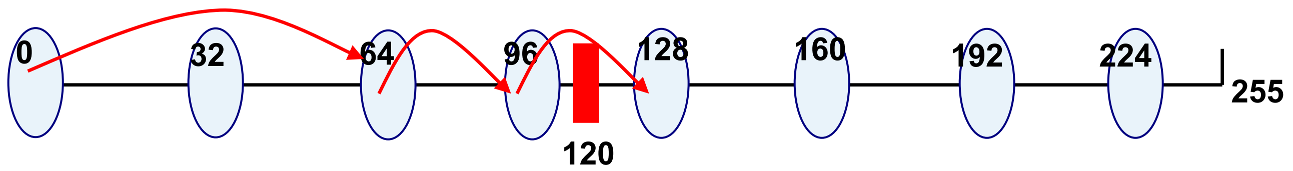
Chord Performance (Improvements)

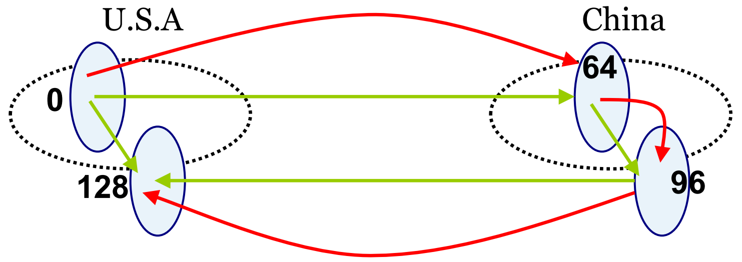
- Chord properties
  - Routing table size *O(log(N))*, where *N* is the total number of nodes
  - Guarantees that a file is found in *O(log(N))* steps
- Reducing latency
  - Chose finger that reduces expected time to reach destination
  - Chose the closest node from range $[N+2^{i-1}, N+2^{i})$ as successor
- Stretch is another parameter:

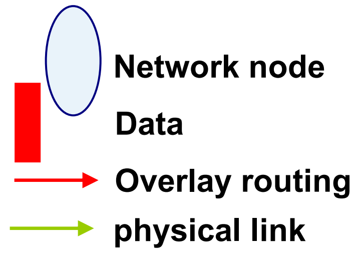
$$= \frac{\text{latency for each lookup on the overlay topology}}{\text{average latency on the underlying topology}}$$

- Nodes **close** on ring, but **far away** in Internet
- Goal: put nodes in routing table that result in few hops and low latency

Latency Stretch in Chord







A Chord network with N (=8) nodes and m (=8)-bit key space

[Ratnasamy, op. cit.]

Achieving Robustness

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- In the ***notify()*** message, node A can send its k-1 successors to its predecessor B
- Upon receiving ***notify()*** message, B can update its successor list by concatenating the successor list received from A with A itself

The Problem of Membership Churn

- In a system with 1,000s of machines, some machines failing / recovering at all times
- This process is called ***churn***
- Without repair, quality of overlay network degrades over time
- A significant problem deployed DHTs systems

Observation: in >50 % cases, mean time between failures (MTBF) in order of minutes.

[Rhea, op. cit.]

What Makes a Good DHT Design?

- The number of neighbors for each node should remain “reasonable” (**small degree**)
- DHT routing mechanisms should be decentralized (**no single point of failure or bottleneck**)
- Should **gracefully handle nodes joining and leaving**
  - Repartition the affected keys over existing nodes
  - Reorganize the neighbor sets
  - Bootstrap mechanisms to connect new nodes into the DHT
- DHT must provide **low stretch**
  - Minimize ratio of DHT routing vs. unicast latency between two nodes

[Felber, op. cit.]

Multiple Solutions

- Chord
- Tapestry
  - Uses locally optimal routing tables to reduce routing stretch
- Pastry
  - Routing overlay network to reduce cost of routing a packet
- CAN
  - Virtual multi-dimensional Cartesian coordinate space

DHTs Support Many Applications

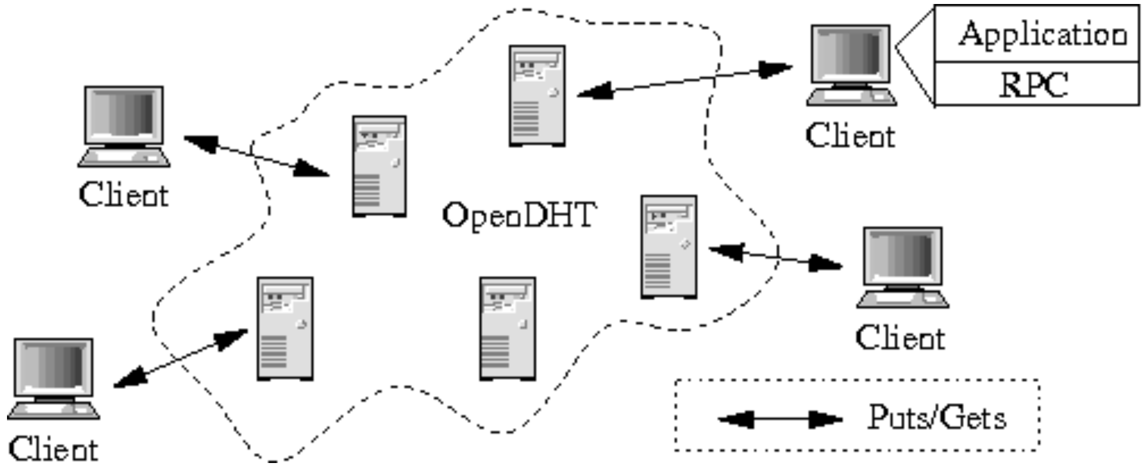
- File sharing [CFS, OceanStore, PAST, …]
- Web cache [Squirrel, …]
- Censor-resistant stores [Eternity, FreeNet, …]
- Application-layer multicast [Narada, …]
- Event notification [Scribe]
- Naming systems [ChordDNS, INS, …]
- Query and indexing [Kademlia, …]
- Communication primitives [I3, …]
- Backup store [HiveNet]
- Web archive [Herodotus]

DHT is a Good Shared Infrastructure

- A single DHT (namely, Open DHT) is shared across multiple applications, thus amortizing the cost of deployment.
- Applications inherit **some** security and robustness from DHT
  - DHT replicates data
  - Resistant to malicious participants
- Low-cost deployment
  - Self-organizing across administrative domains
  - Allows to be shared among applications

[Kashoek, op. cit.]

DHT as an Infrastructure



Open DHT Architecture

[Rhea, op. cit.]

Open DHT Deployment Model

- A single DHT (namely, Open DHT) is shared across multiple applications, thus amortizing the cost of deployment.
- Each DHT node serves as a **gateway** into the DHT for clients.
- Any Internet-connected computer can act as client:
  - Clients of Open DHT do not need to run a DHT node
  - Using DHT services, i.e., can store or put key-value pairs in Open DHT, and can retrieve or get the value stored under a particular key
- Each DHT node serves as a **gateway** into the DHT for clients.
- An Open DHT client communicates with the DHT through the gateway of its choice using an RPC over TCP. The **gateway** processes the operations on client´s behalf.
- Because of this, the service is easy to access from virtually every programming language.