IN.5022 — Concurrent and Distributed Computing

Decentralised Lookup and Storage: DHTs

Prof. P. Felber

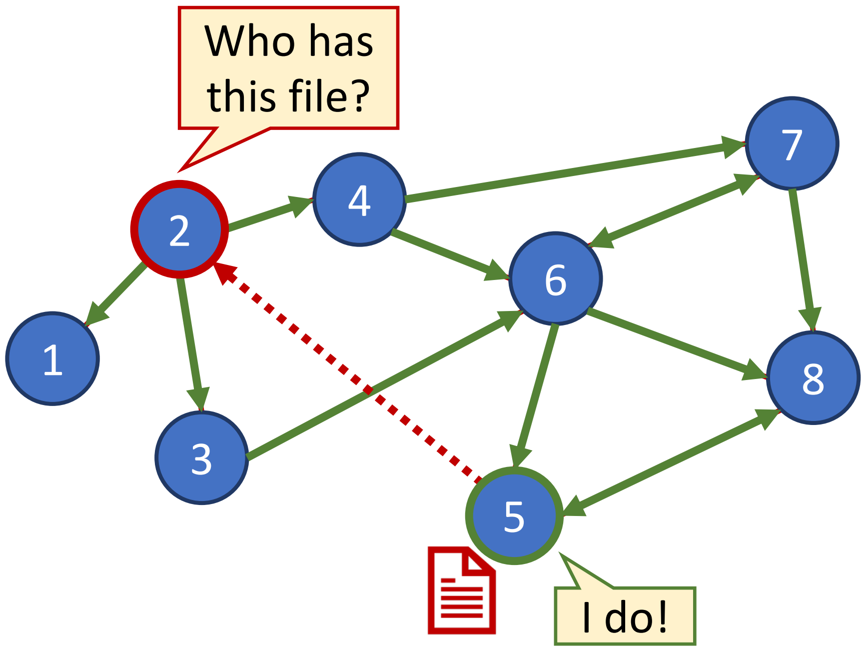
pascal.felber@unine.ch

Agenda

- What are DHTs? Why are they useful?
- What makes a “good” DHT design
- Case study: Chord

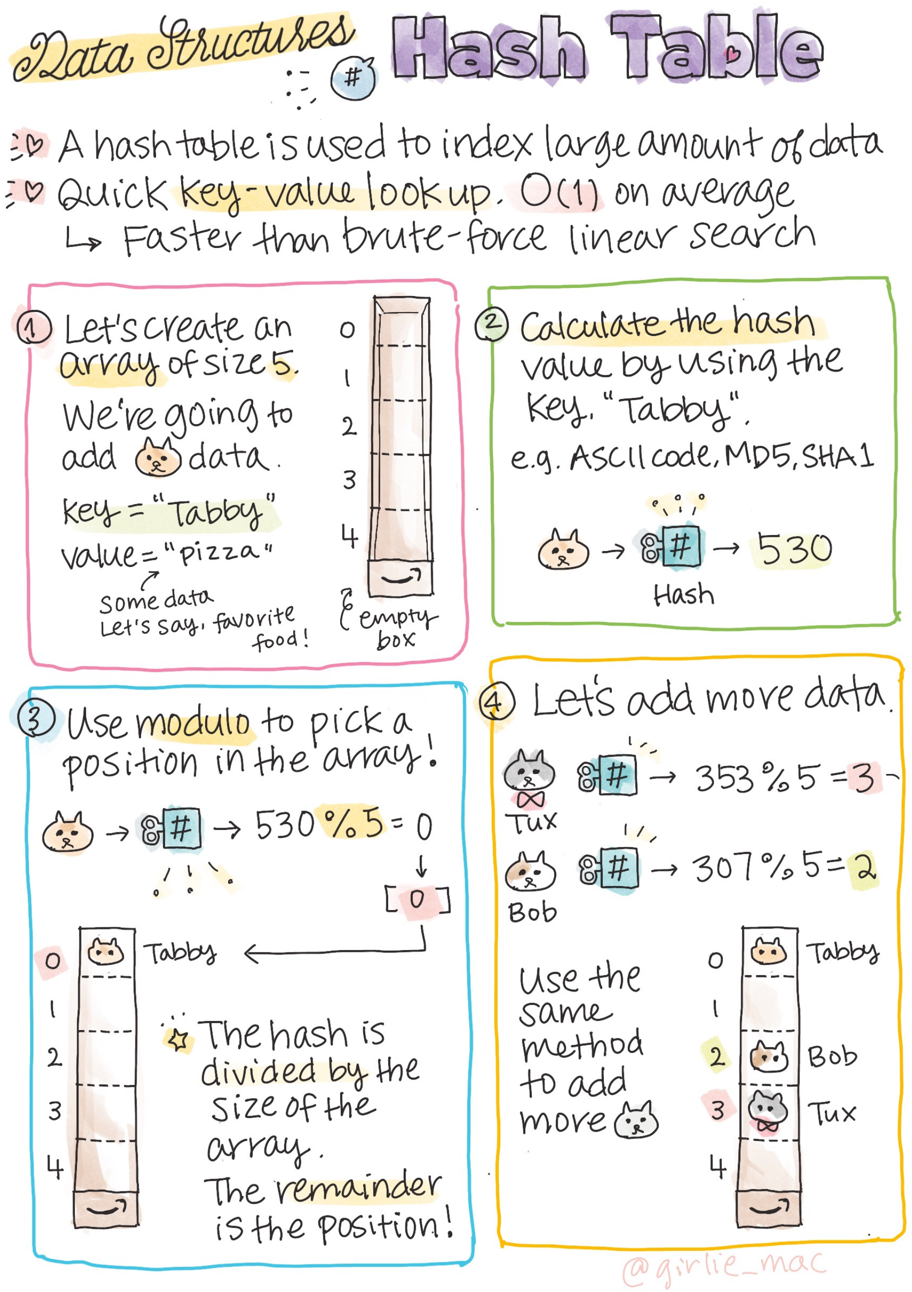
Challenge: locate content in P2P systems

- How to locate content in decentralised settings?
  - Centralized index (“Napster” design)
    - Single point of failure, high load
  - Expanding ring search until content is found (“Gnutella” design)
    - If r of N nodes have copy, the expected search cost is at least N/r, i.e., O(N)
    - Need many copies to keep overhead small

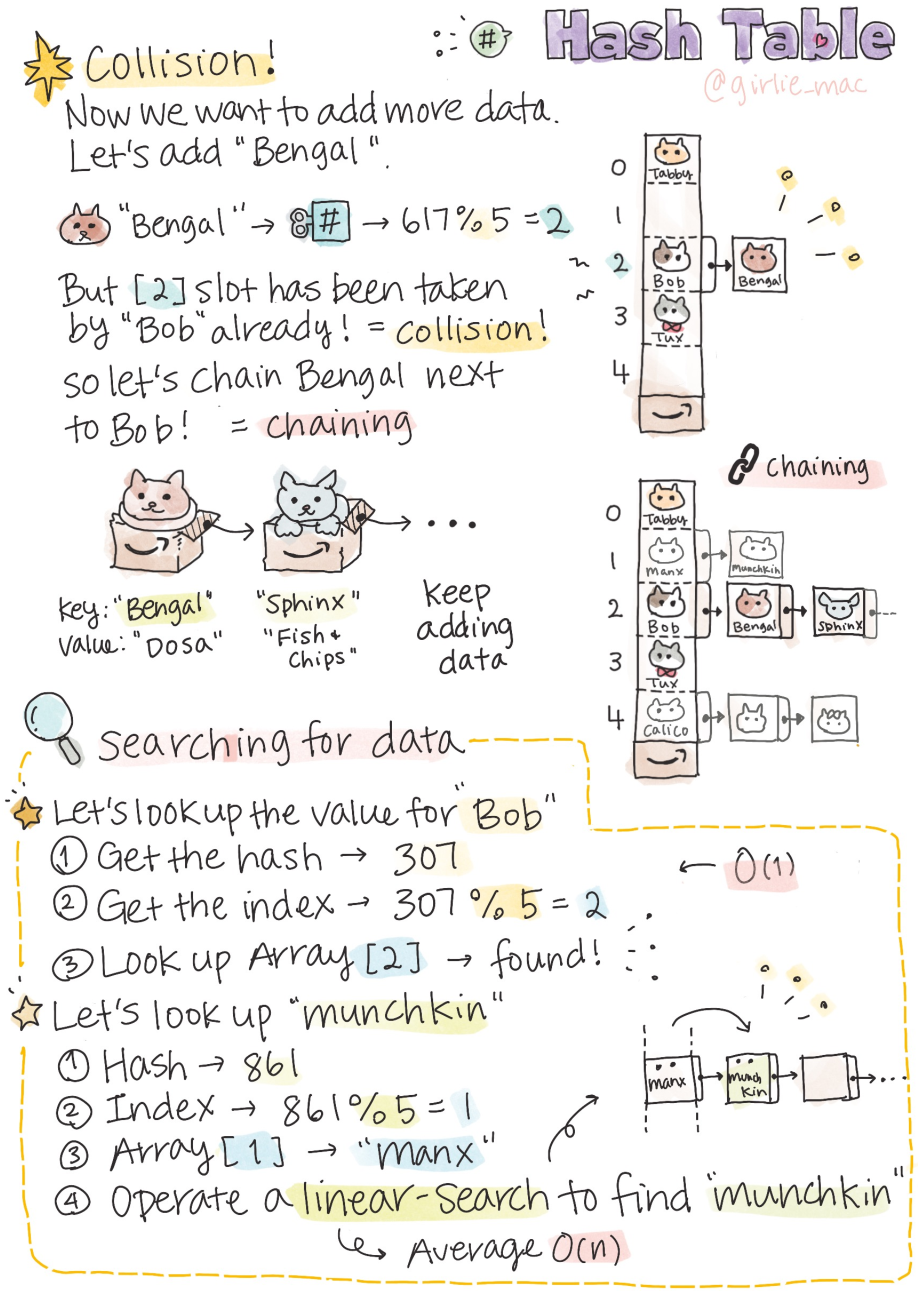


Directed searches

- Idea
  - Assign to particular nodes the responsibility to hold specific content (or know where it is)
  - When a node wants this content, go to the node that is supposed to hold it (or know where it is)
- Challenges
  - Avoid bottlenecks
    - Distribute the responsibilities “evenly” among the existing nodes
  - Adaptation to nodes joining or leaving (or failing)
    - Give responsibilities to joining nodes
    - Redistribute responsibilities from leaving nodes



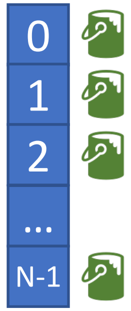
Data Structures
Hash Table
A hash table is used to index large amount of data
Quick key-value look up. O(1) on average
↳ Faster than brute-force linear search
① Let's create an array of size 5.
We're going to add data.
key = "Tabby"
value = "pizza"
Some data Let's say, favorite food!
empty box
0 1 2 3 4
② Calculate the hash value by using the key, "Tabby".
e.g. ASCII code, MD5, SHA1
→ 530
Hash
③ Use modulo to pick a position in the array!
→ 530 % 5 = 0
[0]
Tabby
The hash is divided by the size of the array.
The remainder is the position!
④ Let's add more data.
Tux → 353 % 5 = 3
Bob → 307 % 5 = 2
Use the same method to add more
Tabby
Bob
Tux
@girlie_mac



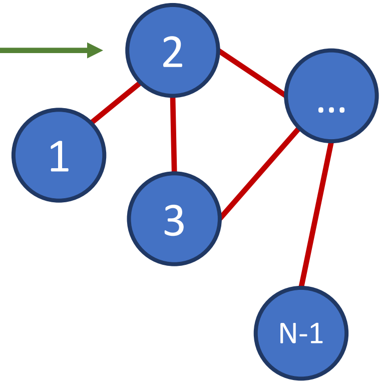
Hash Table
@girlie_mac
Collision!
Now we want to add more data.
Let's add "Bengal".
"Bengal" → 617 % 5 = 2
But [2] slot has been taken by "Bob" already! = collision!
so let's chain Bengal next to Bob! = chaining
Tabby
Bob
Bengal
Tux
Key: "Bengal"
Value: "Dosa"
"Sphinx"
"Fish & Chips"
Keep adding data
chaining
Tabby
Manx
Munchkin
Bob
Bengal
Sphinx
Tux
Calico
Searching for data
Let's look up the value for "Bob"
① Get the hash → 307
② Get the index → 307 % 5 = 2
③ Look up Array[2] → found!
← O(1)
Let's look up "munchkin"
① Hash → 861
② Index → 861 % 5 = 1
③ Array[1] → "manx"
④ Operate a linear-search to find "munchkin"
↳ Average O(n)
manx
munch kin

Distributed hash tables

- Idea: exploit the principle of hash tables, but in distributed settings
- A hash table associates data with keys
  - Key is hashed to find *bucket* in hash table: *hash(key) → id*
    - Typically, bucket index is: `idx = hash("name") % #buckets`
  - Each bucket is expected to hold #items/#buckets items



- In a distributed hash table (DHT), the nodes are the hash buckets
  - Key is hashed to find responsible *peer node*
    - Node identifier becomes: `id = hash("name") % #nodes`
  - Data and load are balanced across nodes



DHTs: Problems

- Problem 1 (dynamicity): adding or removing nodes
  - With hash mod N, virtually every key will change its location!
    - $h(k) \% N \neq h(k) \% (N+1) \neq h(k) \% (N-1)$
- Solution: 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 identifier in hash space (according to some proximity metric)

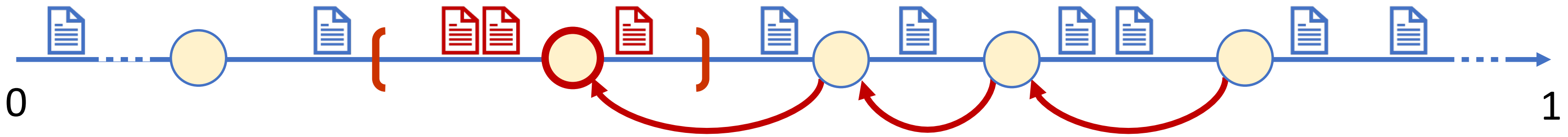
DHT hashing

- Based on consistent hashing (designed for Web caching)
  - Each cache (server) and each page (hash of URL) have an identifier uniformly distributed in range [0, 1]
  - A page is stored to the closest cache in the identifier space

    hash(URL) → id ⇒ under the responsibility of closest server
  - Good load balancing: each cache covers roughly equal intervals and stores roughly the same number of pages
  - Adding or removing a server invalidates few keys

DHTs: Problems (cont'd)

- Problem 2 (size): all nodes must be known a priori to insert or lookup data
  - Works with *small and* static server populations
- Solution: each peer knows of only a few "neighbours"
  - Messages are routed through neighbours via multiple hops (overlay routing)



What makes a good DHT design?

- Small diameter: for each object, the node(s) responsible for that object should be reachable via a “short” path
  - The different DHTs differ fundamentally only in the routing approach
- Small degree: the number of neighbours for each node should remain “reasonable”
- Diameter and degree should be balanced
  - Ring has small degree but high diameter
  - Full mesh has small diameter but high degree

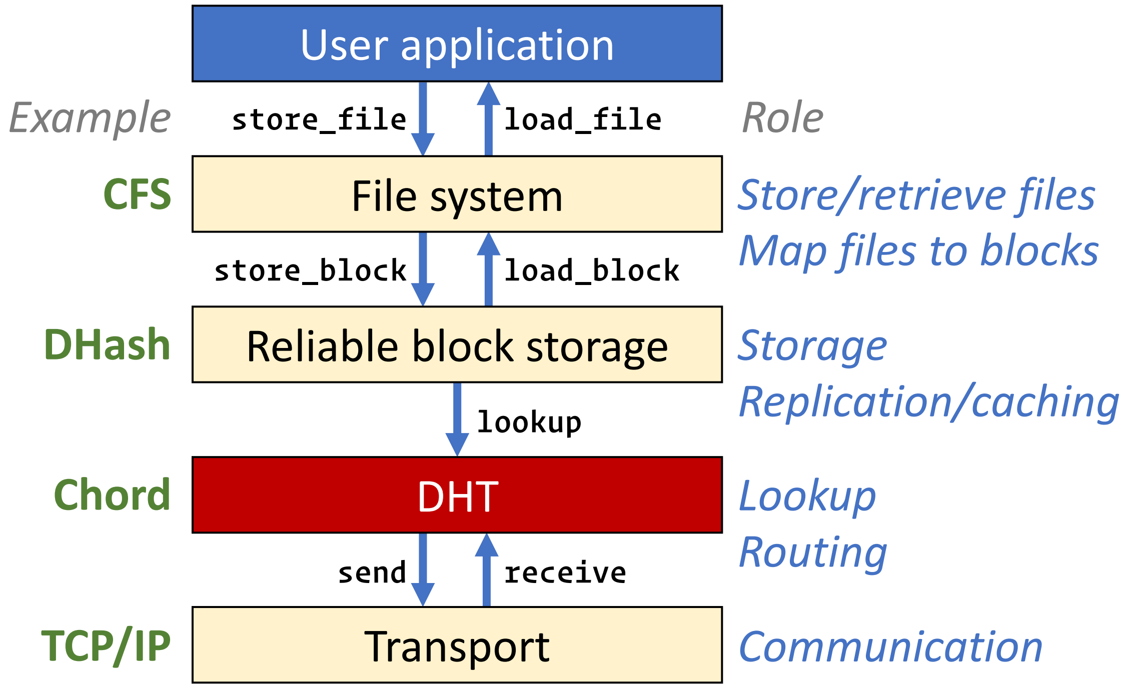
What makes a good DHT design? (cont'd)

- No single point of failure or bottleneck: DHT routing mechanisms should be decentralized
  - Should gracefully handle nodes joining and leaving
  - Repartition the affected keys over existing nodes
  - Reorganize the neighbour sets
  - Bootstrap mechanisms to connect new nodes into the DHT
- Low stretch: to achieve good performance, multi-hop messages should (topologically) progress toward destination
  - Minimize ratio of DHT routing vs. unicast latency

DHT interface

- Minimal interface (data-centric)
  - Lookup(key) → IP address
- Supports a wide range of applications, because few restrictions
  - Keys have no semantic meaning
  - Value is application dependent
- DHTs do *not* store the data
  - Data storage can be built on top of DHTs
    - Insert(key, data)
    - Lookup(key) → data

DHTs in context



DHTs as generic building blocks

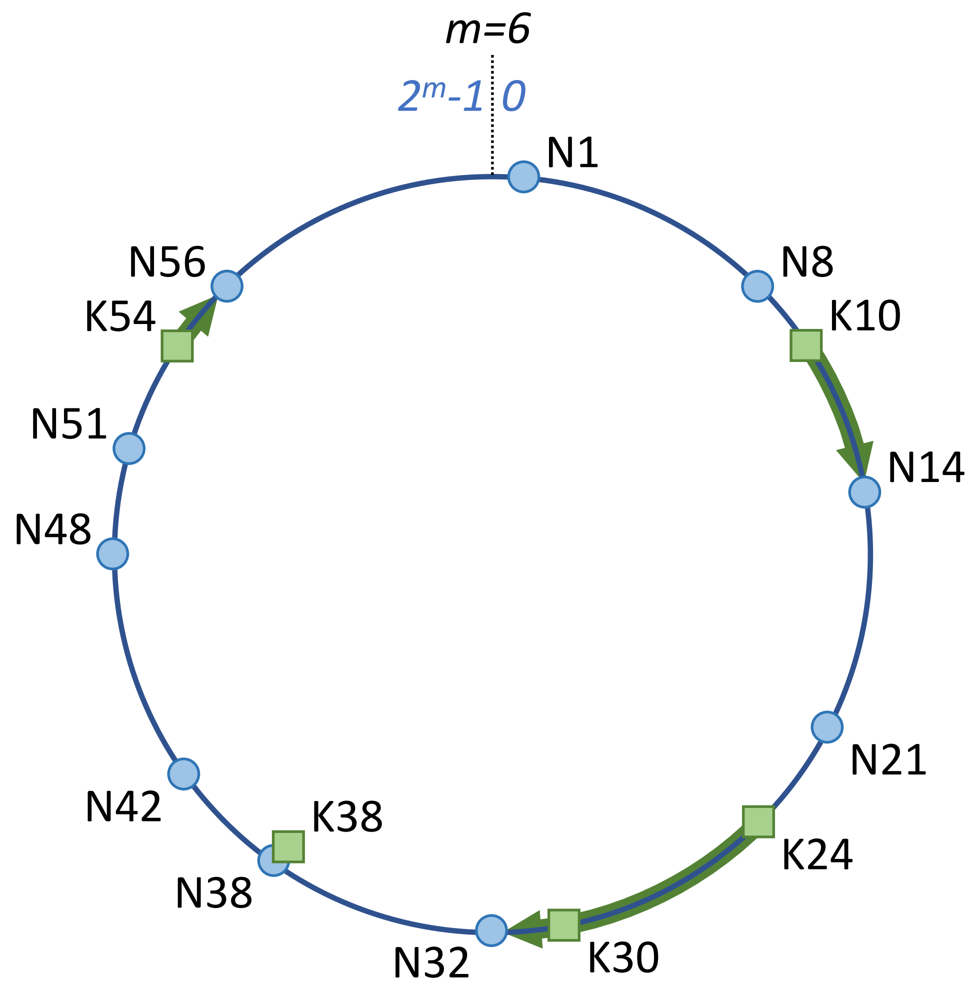
- Many distributed services have been designed on DHTs
  - File sharing (e.g., CFS, OceanStore, PAST)
  - Web cache (e.g., Squirrel)
  - Censor-resistant stores (e.g., Eternity, FreeNet)
  - Application-layer multicast (e.g., Narada)
  - Event notification (e.g., Scribe)
  - Naming systems (e.g., ChordDNS, INS)
  - Query and indexing (e.g., Kademlia)
  - Communication primitives (e.g., I3)
  - Backup store (e.g., HiveNet)
  - Web archive (e.g., Herodotus)

DHT case studies

- A complete case study
  - Chord
- Other classical designs
  - Pastry, CAN, Kademlia, …
- Questions
  - How is the hash space divided evenly among nodes?
  - How do we locate a node?
  - How does we maintain routing tables?
  - How does we cope with (rapid) changes in membership?

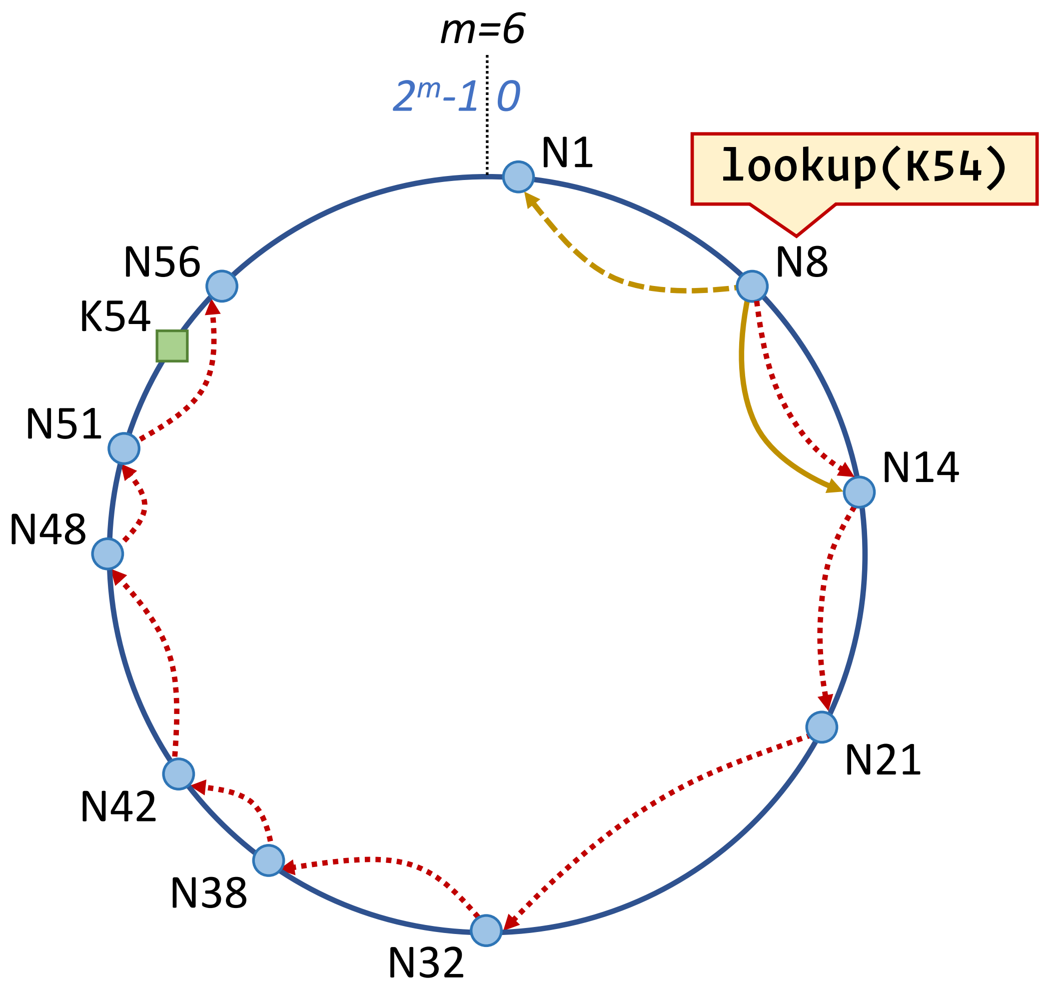
Chord (MIT)

- Circular m-bit identifier space for both keys and nodes
  - Node ID = SHA-1(IP address)
  - Key ID = SHA-1(key)
- A key is mapped to the first node whose identifier is equal to or follows that of the key
  - Each node is responsible for O(K/N) keys
  - When a node joins or leaves, O(K/N) keys move



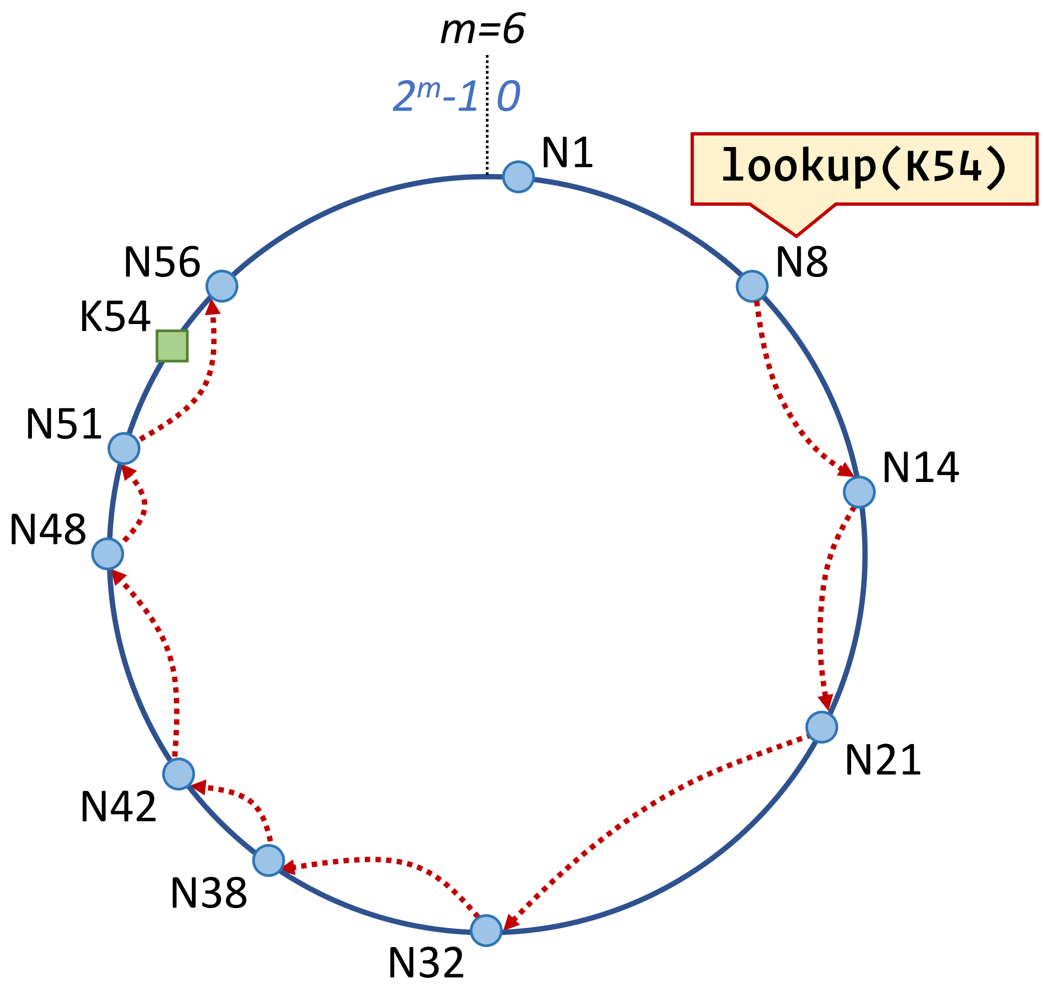
Chord state and lookup

- Basic Chord: each node knows only 2 other nodes
  - Successor
  - Predecessor (ring management)
- Lookup by forwarding requests around the ring through successor pointers
  - Requires O(N) hops



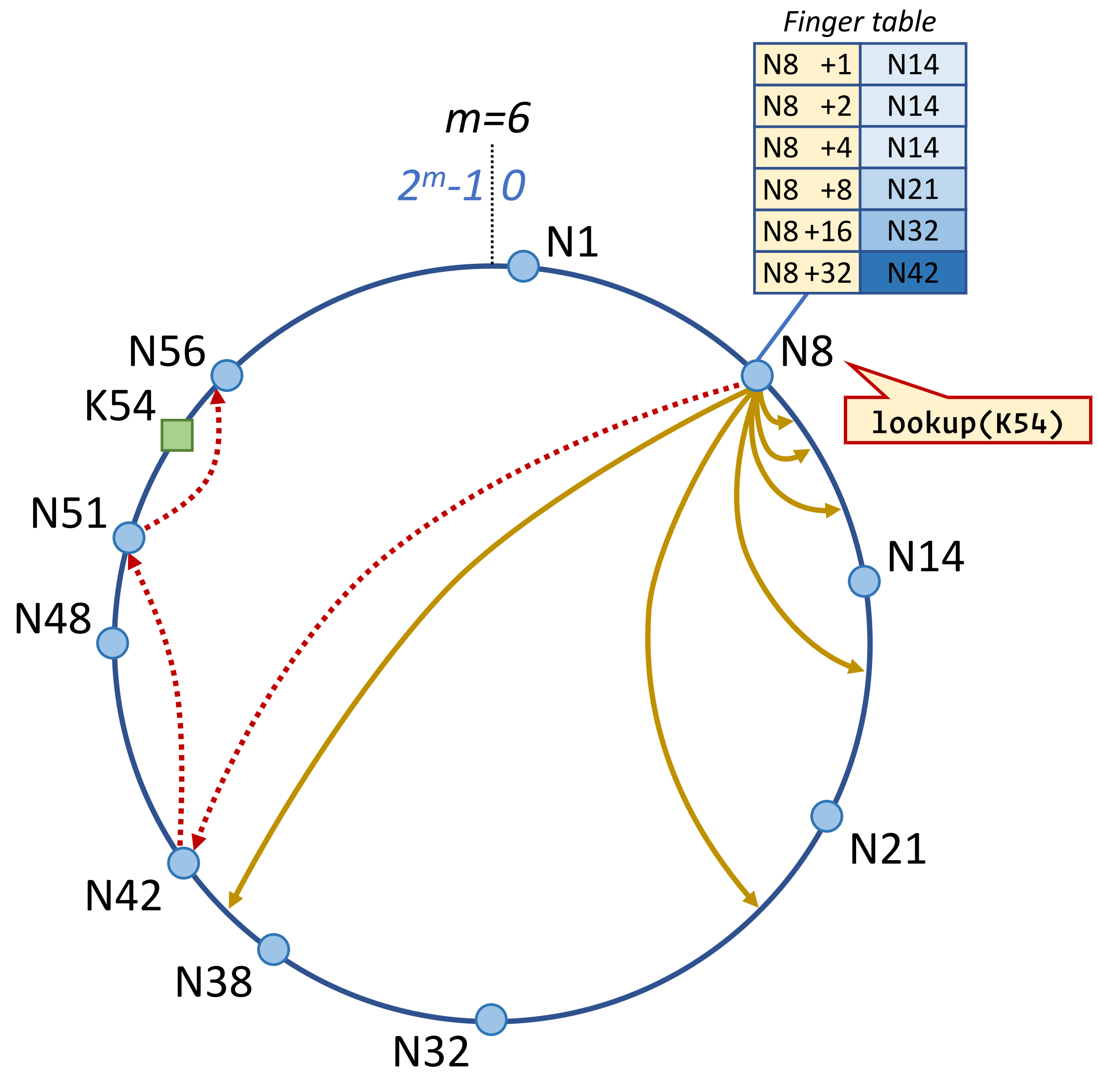
Chord state and lookup (cont'd)

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
   // forward the query around the circle
   return successor.find_successor(id);
```



Chord state and lookup

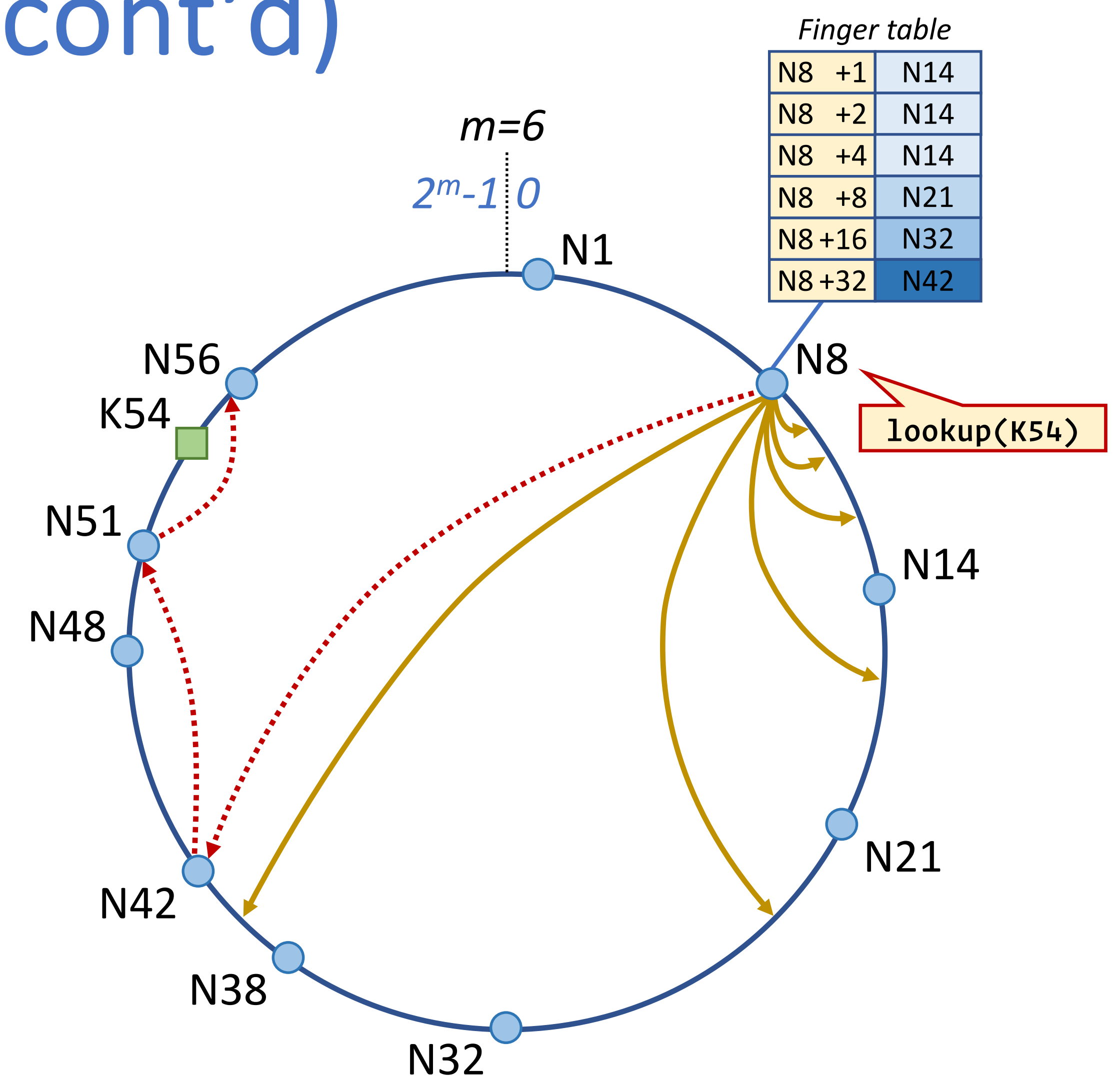
- Each node knows m other successor nodes on the ring
  - "Finger" i of n points to node at $n+2^i$
  - State is O(log N) per node
- Lookup is achieved by following closest preceding fingers, then successor
  - Lookup takes O(log N) hops



Chord state and lookup (cont'd)

```
// 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;
```



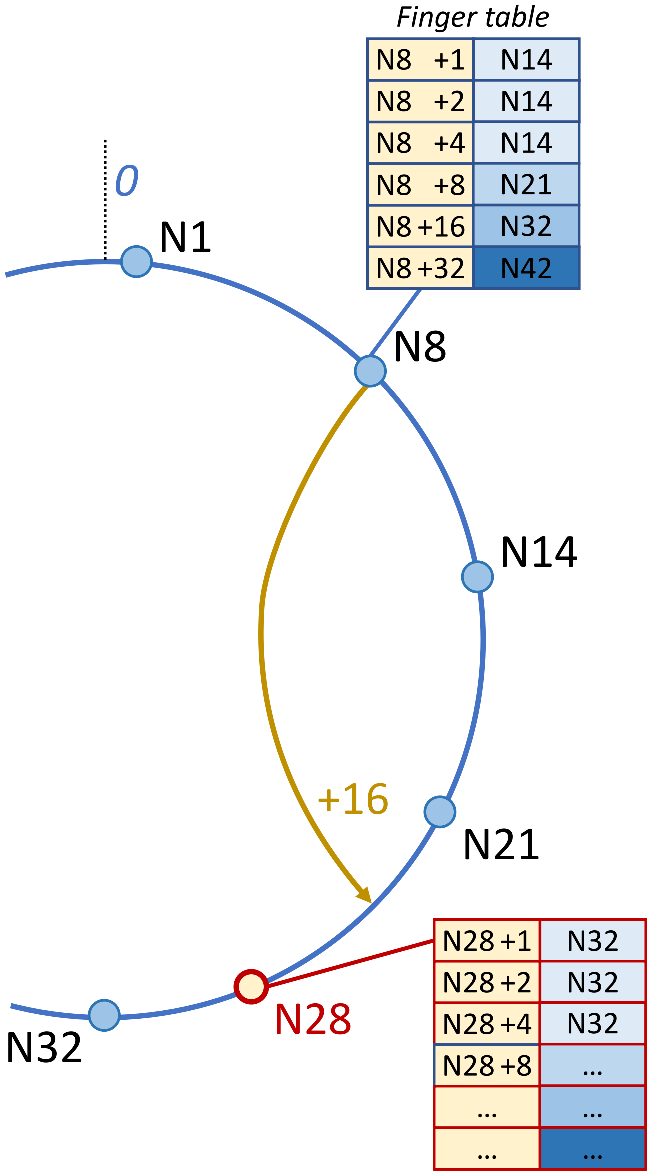
Chord ring management

- For correctness, Chord needs to maintain the following invariants
  - For every key k, the successor of k is responsible for k
  - Successor pointers are correctly maintained
- Finger tables are not necessary for correctness
  - One can always default to successor-based lookup
  - Finger table can be updated lazily

Joining the ring

- Three step process
  1. Initialize all fingers of new node
  2. *Update fingers of existing nodes*
  3. *Transfer keys from successor to new node*

1. Initialize the new node finger table
   - Locate any node n in the ring
   - Ask n to lookup the peers at $j+2^0$, $j+2^1$, $j+2^2$…
   - Use results to populate finger table of j

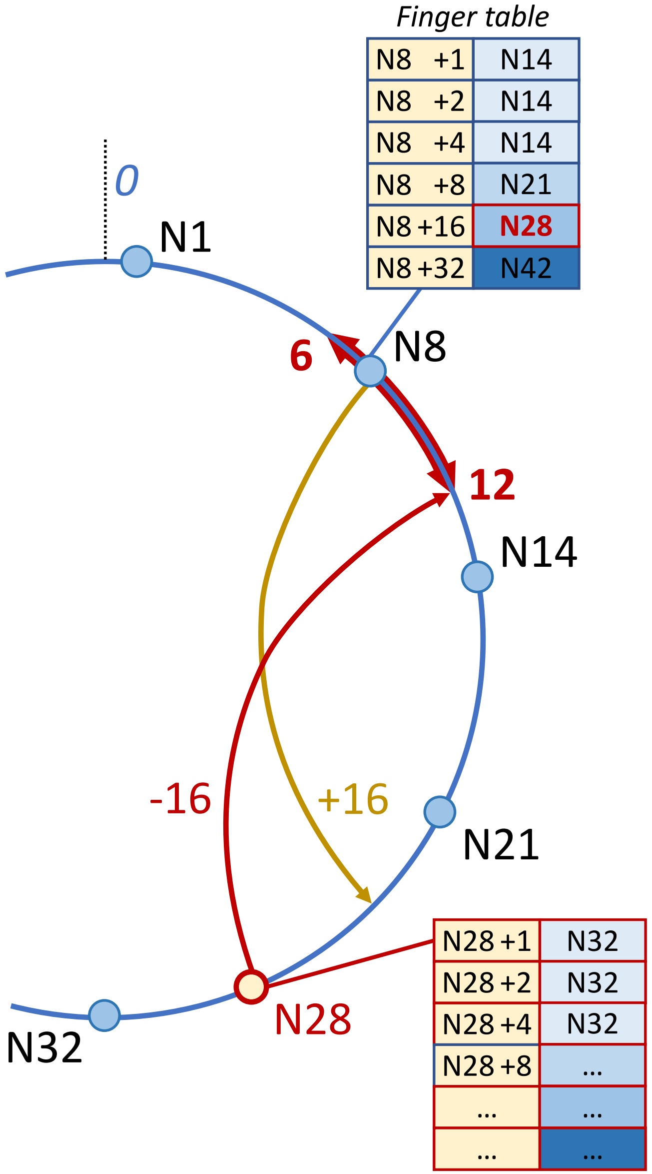


Joining the ring (cont')

- Three step process
  1. *Initialize all fingers of new node*
  2. Update fingers of existing nodes
  3. *Transfer keys from successor to new node*

2. Update fingers of existing nodes

- New node j calls update function on existing nodes that must point to j
  - Nodes in the ranges [j-2^i, predecessor(j)-2^i+1]
- O(log N) nodes need to be updated

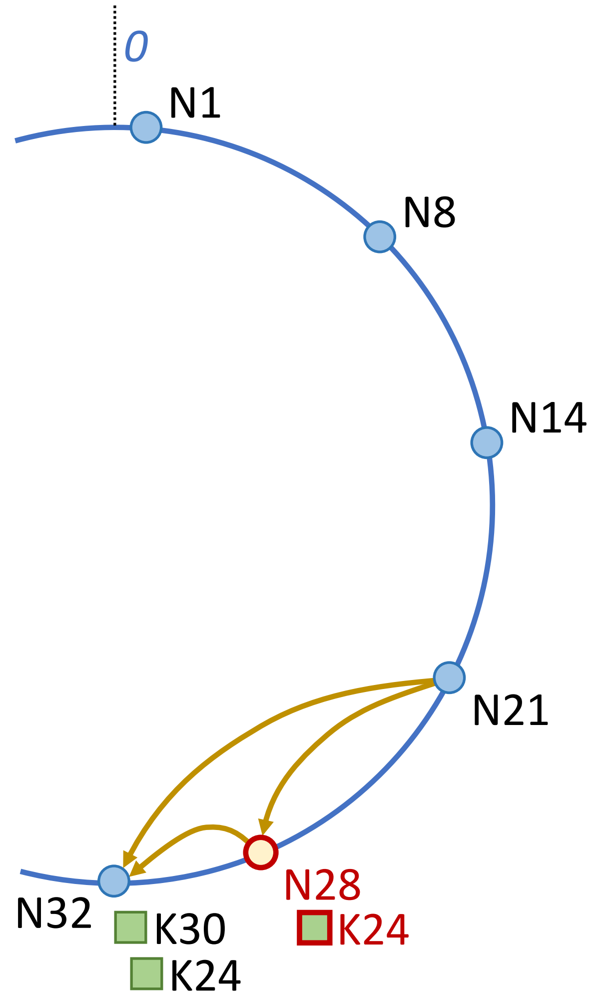


Joining the ring (cont')

- Three step process
  1. *Initialize all fingers of new node*
  2. *Update fingers of existing nodes*
  3. Transfer keys from successor to new node

3. Transfer key responsibility
   - Connect to successor
   - Copy keys from successor to new node
   - Update successor pointer and remove keys

   Only keys in the range are transferred!

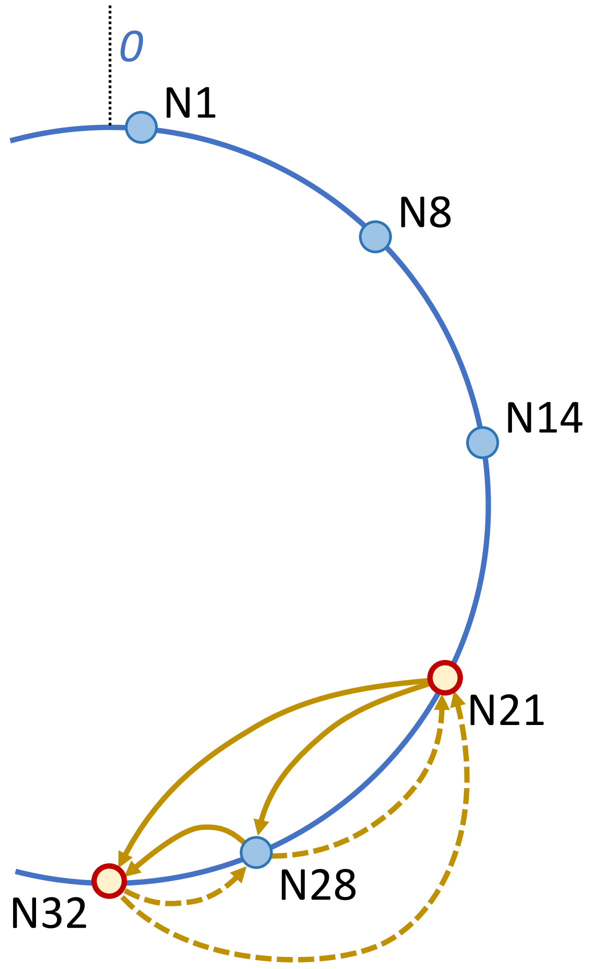


Stabilization

- Three situations to handle
  1. Finger tables are reasonably fresh
  2. Successor pointers are correct, not fingers
  3. Successor pointers are inaccurate or key migration is incomplete — TO AVOID!
- Stabilization algorithm periodically verifies and refreshes pointers/fingers

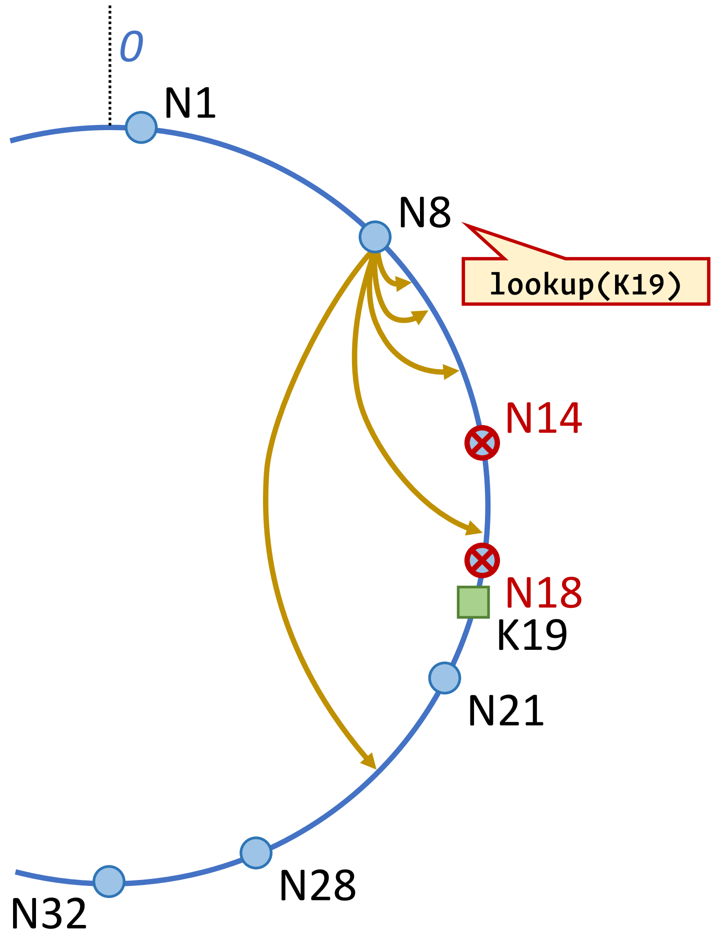
  Eventually stabilizes the system when no node joins or fails

```
// run periodically at node n
x = n.pred.succ              x = n.succ.pred
if x ∈ (n.pred, n)           if x ∈ (n, n.succ)
  n.pred = x                   n.succ = x
```



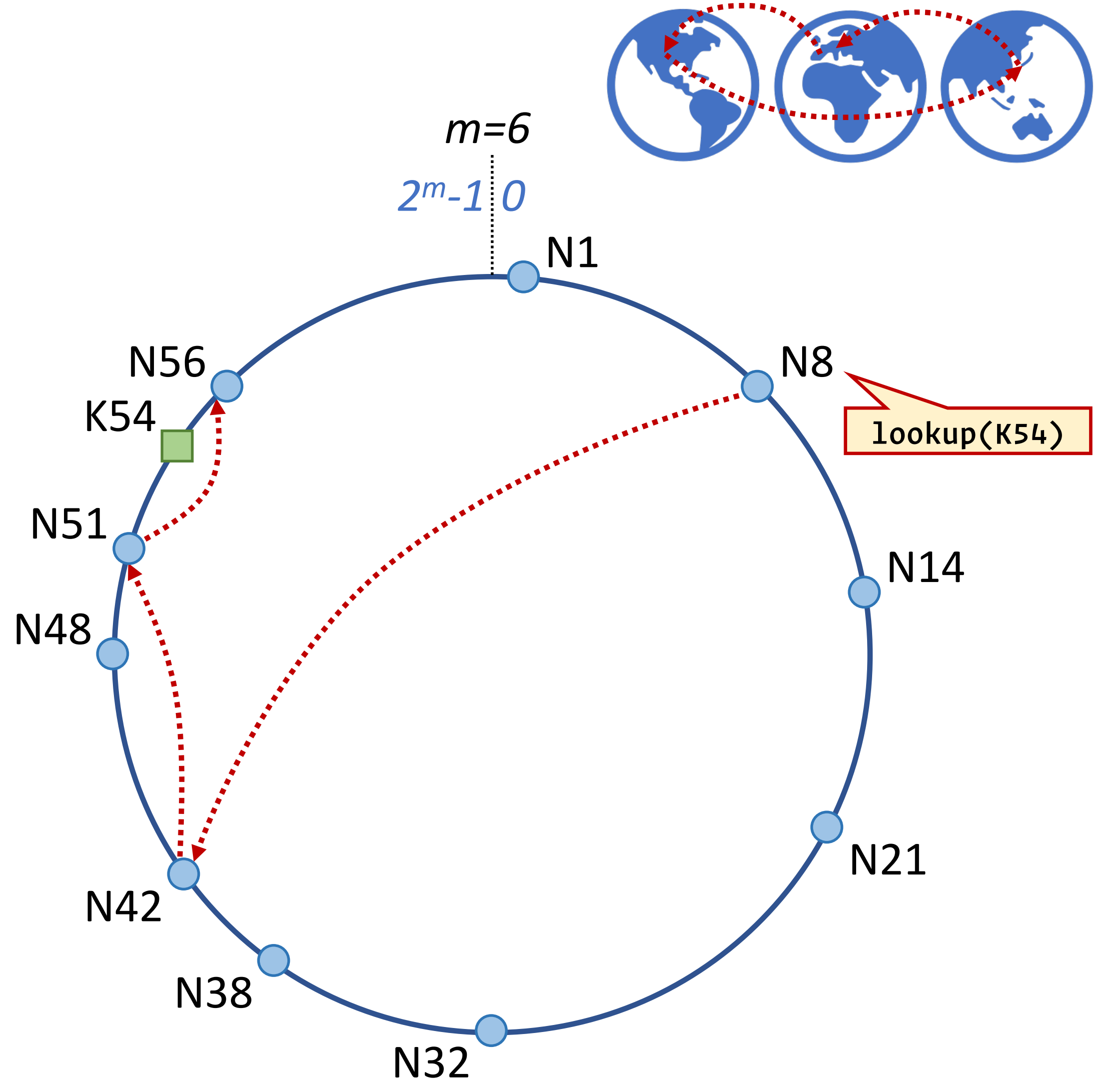
Dealing with failures

- Node failures may cause incorrect lookup
  - Node does not know correct successor, so lookup of key fails
- Solution: successor list
  - Each node n knows r immediate successors
  - After failure, n knows first *live* successor and updates successor list
  - Correct successors guarantee correct lookups
- Successor lists guarantee correct lookup with some probability (depending on r)
  - P(correct) = 1 – 1/N (w.h.p.) with r = 2 log(N)



Locality

- Nodes numerically close are not topologically close
  - Multi-hop lookups may have high latency (e.g., more than 10 hops with 1 million nodes)
- Some DHTs, such as Pastry, provide *low stretch*
  - Multiple choices for fingers
    - Pick those that are “closest” (according to proximity metric)
    - Limited stretch (~2–3)



Chord properties

- Search types: only equality
- Scalability
  - Degree: $O(\log(N))$
  - Diameter (search and update): $O(\log(N))$ w.h.p.
  - Construction: $O(\log^2(N))$ when a new node joins
- Robustness: can replicate keys at successor nodes
- Autonomy: IP address imposes a specific role to nodes
- Global knowledge
  - Mapping of IP addresses and data keys to common key space
  - Single origin (single initial node, cannot merge rings)

Other designs

- Many DHT designs have been proposed in the early 2000s
  - Pastry: improves on locality, robustness
  - CAN: based on cartesian space
  - Koorde: based on *De Bruijn* graph (reduced number of hops)
  - Kademlia: XOR-based distance metric (faster lookup, robustness)
  - Tapestry, P-Grid, …
- Each design has its advantages and drawbacks
  - Degree (space complexity), diameter (lookup latency), topology-awareness, maintenance cost, simplicity (structure, protocols, APIs), flexibility, robustness, security…

Summary

- DHTs are a simple, yet powerful abstraction
  - Building block of many distributed services (file systems, application-layer multicast, distributed caches, etc.)
- Many DHT designs, with various trade-offs
  - Balance between state (degree), speed of lookup (diameter) and ease of management
- System must support rapid changes in membership
  - Dealing with joins/leaves/failures is not trivial
  - Dynamics of P2P network is difficult to analyse