Chord

A Scalable Peer-to-Peer Lookup Protocol

About me

- Software Engineer @ MindTouch
- Interests
  - Distributed Systems
  - Algorithms
- Hobbies
  - Martial Arts
  - Snowboarding
  - Diving
  - Mountain Biking

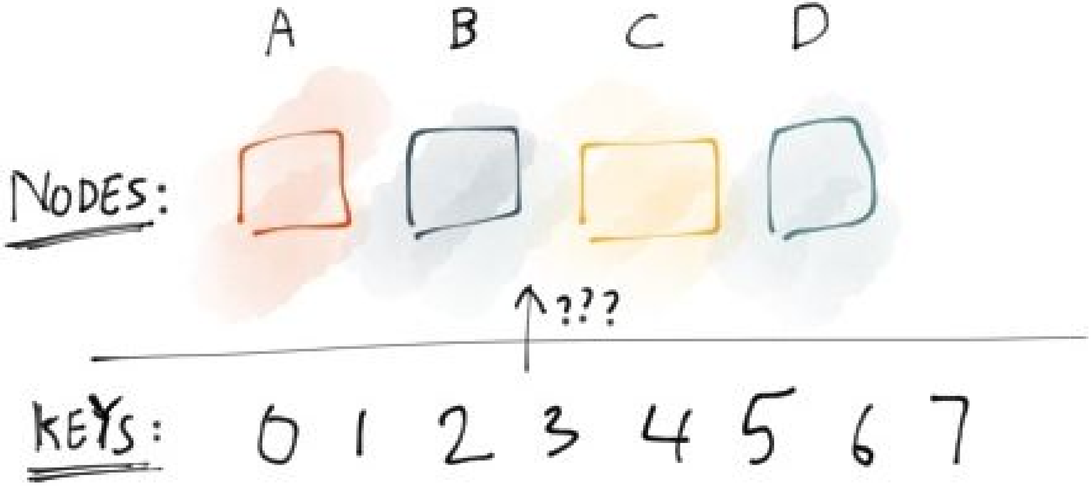
What is Chord?

- Distributed Hash Table
- Hash Table
    - Mapping of Key -> Value
    - Dictionary
    - Map
    - HashMap

Examples

- DNS
- K/V Stores (Memcache, Redis, DynamoDB)
- GlusterFS
- AWS S3

WHICH KEY GOES TO WHICH NODE?



Prior Art

- Central Repository (Napster)
- Query Flooding (Gnutella)
- Hierarchy (Kazaa, modern Gnutella)

How to organize Nodes

- Up to 90’s
  - Master-Slave model (Napster)
  - Unequal responsibilities
- More recently
  - P2P (Gnutella)
  - Equal responsibilities

Costs

	Memory	Lookup Latency	# Messages
Napster	O(N)	O(1)	O(1)
Gnutella	O(N)	O(N)	O(N)

Chord Design Goals

- Good Performance
- Simple and correct protocol
- Degrades gracefully under failure
- Scalable to large number of nodes
- No naming structure for keys
- Load balanced
- Decentralized (P2P)
- Available

Why?

- Distributed Storage System
  - Keys: filename
  - Values: node responsible for storing the file
  - Cooperative mirroring
- Distributed Indexes
  - Keys: search terms
  - Values: nodes containing files with those terms
- Large-Scale Combinatorial Search
  - Keys: candidate solution
  - Values: machines responsible for solution

What data structure is optimized for lookup speeds?

Hashing

- Maps n-bit key into k-buckets
- Function   fn(key) = node identifier
- Goals
  - Low cost
  - Deterministic
  - Load balanced
- Simplest Hash Function (I can think of)
  - fn(key) = key % #nodes

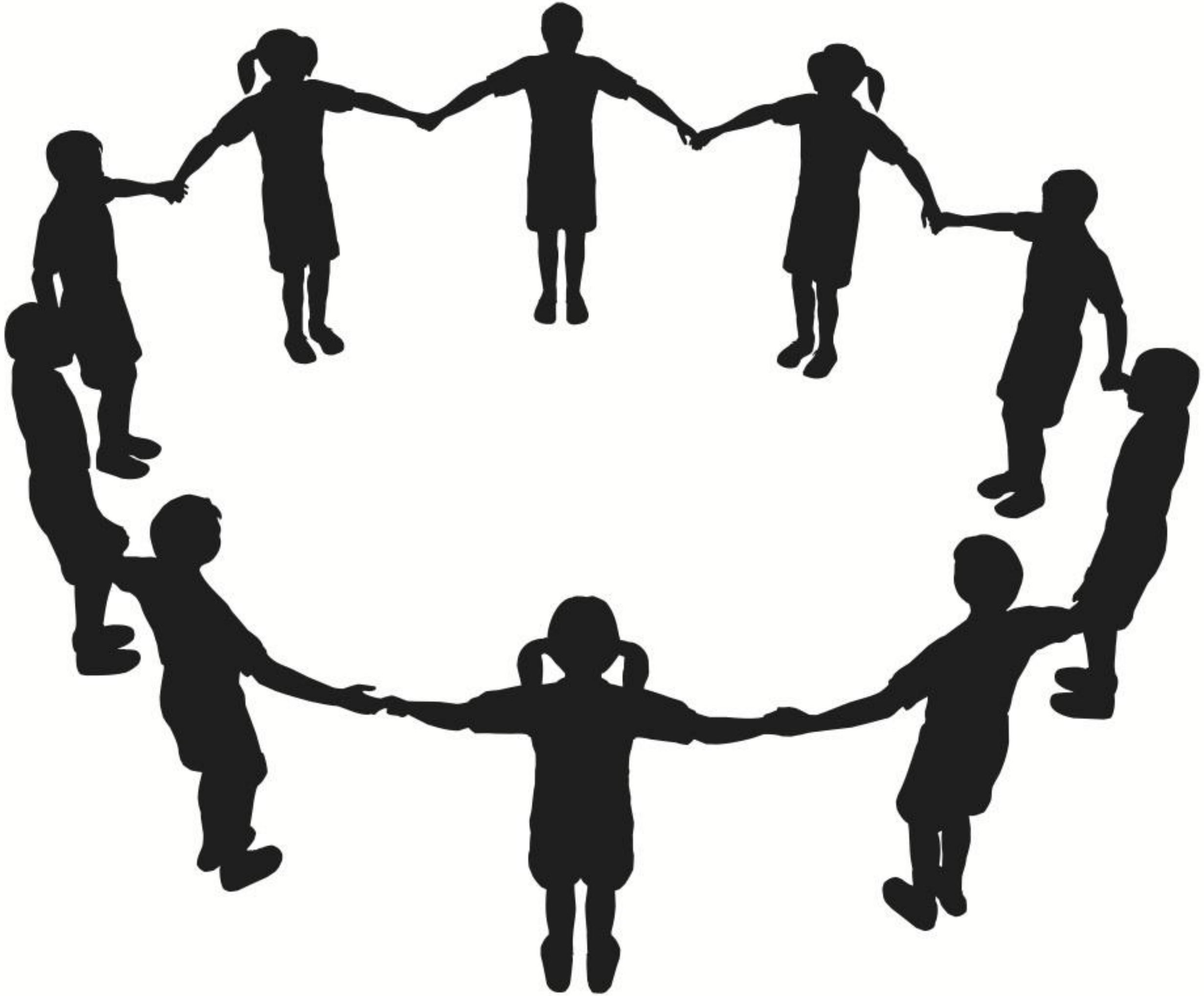
Modulo Hashing

- Good Performance
- Simple and Correct (provably) Protocol
- ~~Degrades gracefully under failure~~
- Scalable to large number of nodes
- No naming structure for keys
- ~~Load balanced~~
- Decentralized (P2P)
- ~~Available~~

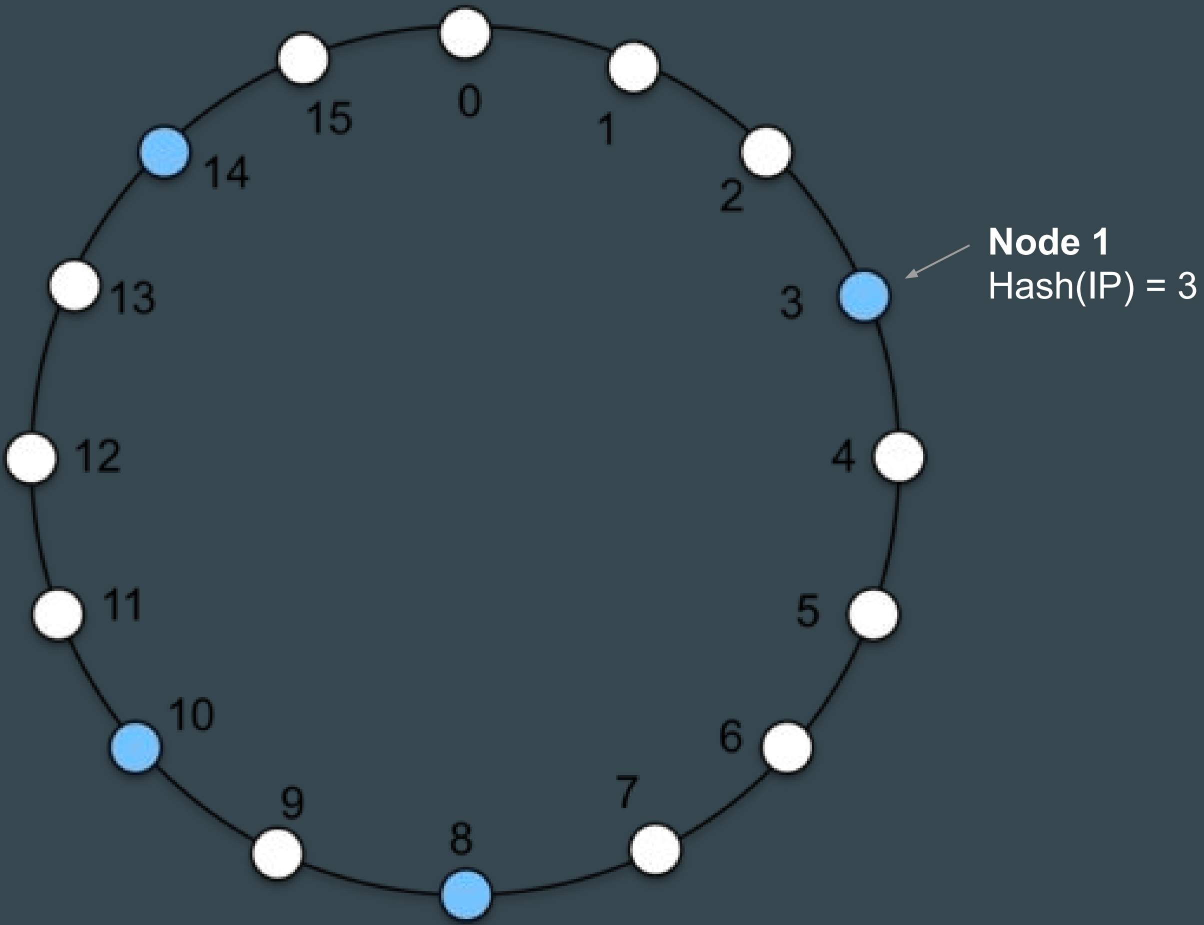
Consistent Hashing

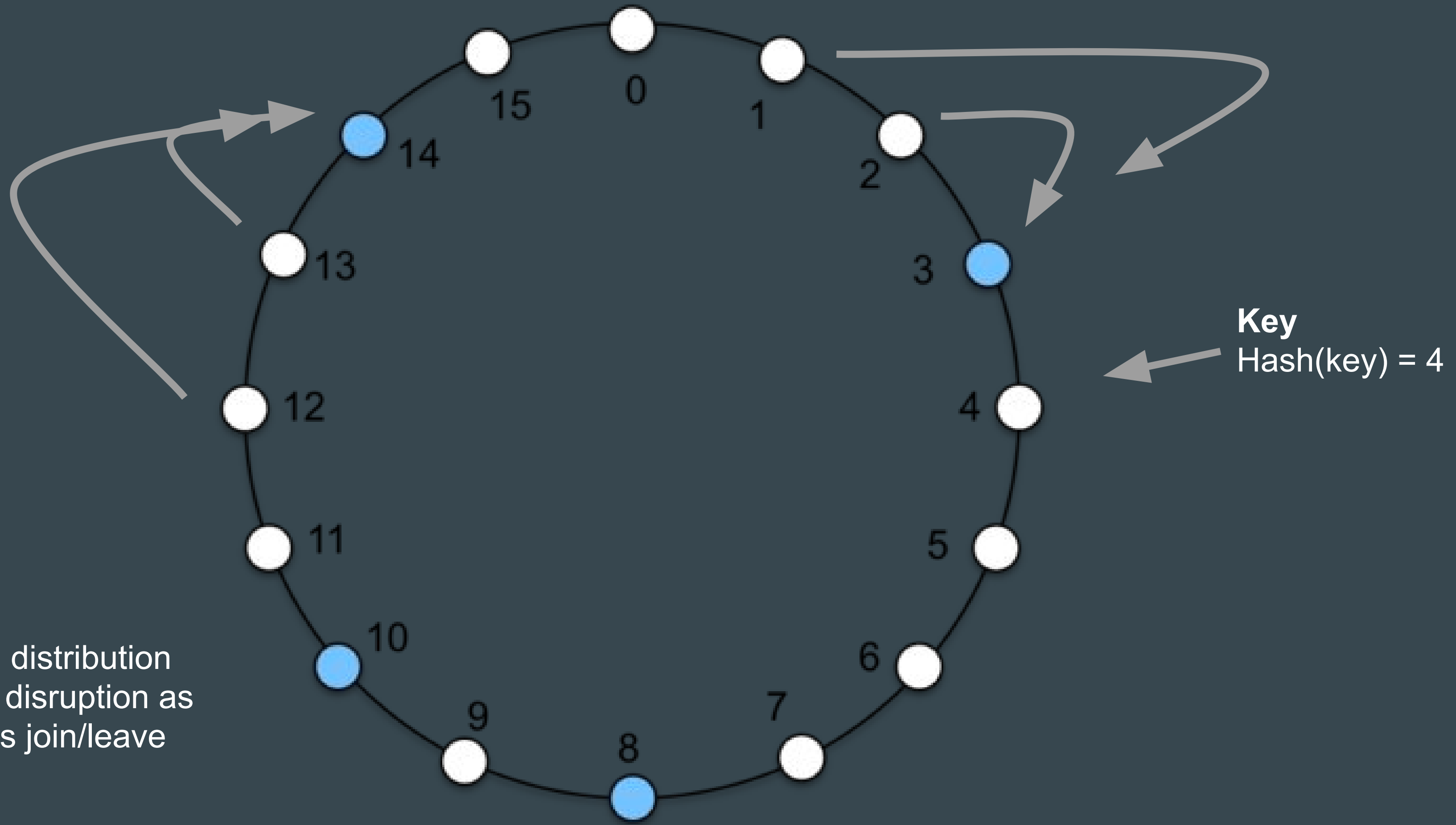
Consistent Hashing

- Organize Nodes in a Ring (Structured P2P)
- Maintains neighbors
- Lookups follow neighbor links clockwise around the ring
- Nodes and Keys are “hashed” onto the Ring



- Maintains a doubly linked list (successor/predecessor)
- Separate Join/Leave protocol





- Even distribution
- Little disruption as nodes join/leave

Lookup

// ask node n to find the successor of id

n.**find_successor**(id)

    if ($id \in (n, successor]$)

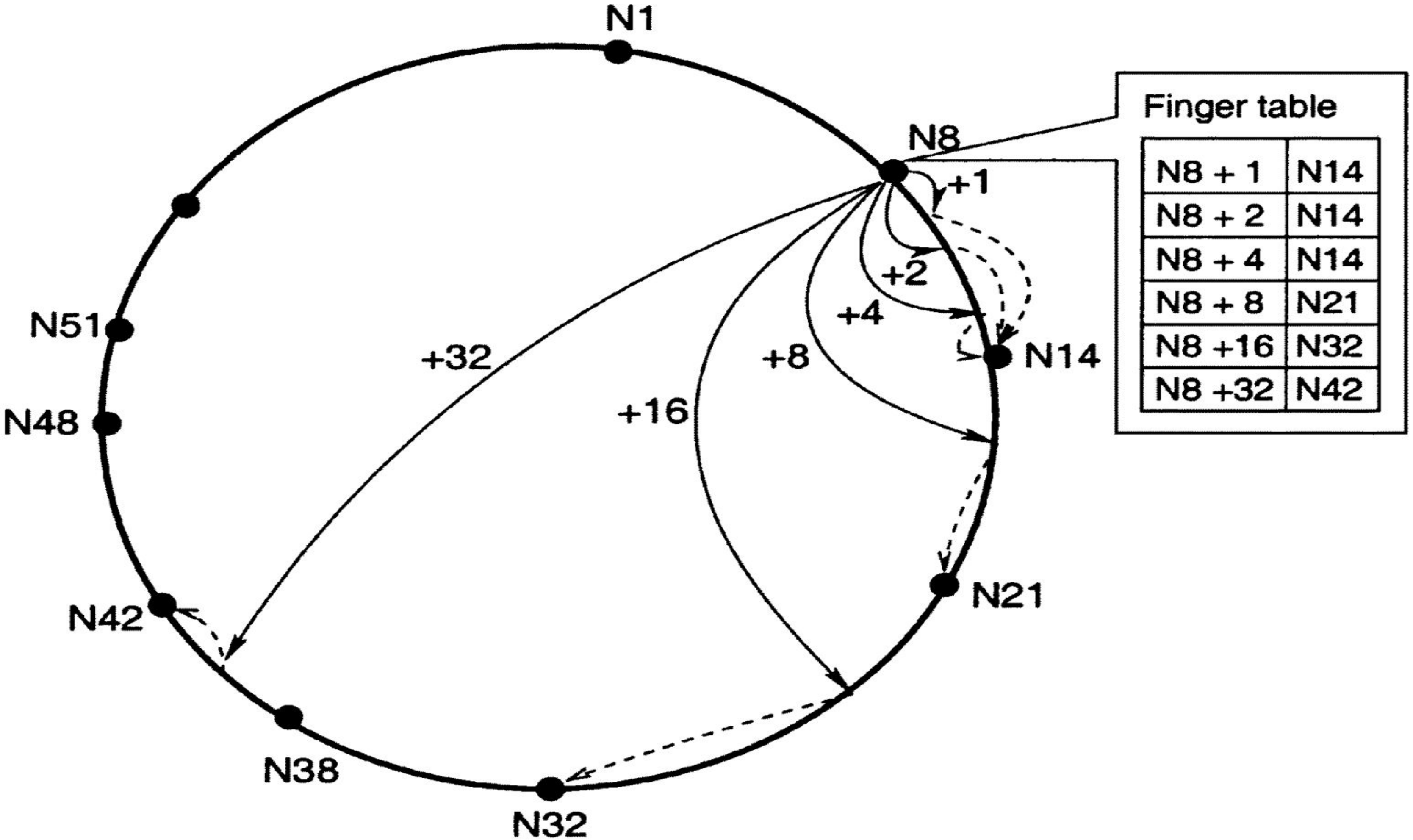
        return *successor*;

    else

        $n' = \cancel{closest\_preceding\_node}(id)$; **n.get_successor()**

        return $n'.find\_successor(id)$;

Lookup Complexity?



N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Lookup

// *search the local table for the highest predecessor of* id

n.**closest_preceding_node**(id)

  for $i = m$ **downto** 1

    if (*finger*[i] $\in (n, id)$)

      return *finger*[i];

  return n;

// *ask node* n *to find the successor of* id

n.**find_successor**(id)

  if ($id \in (n, successor]$)

    return *successor*;

  else

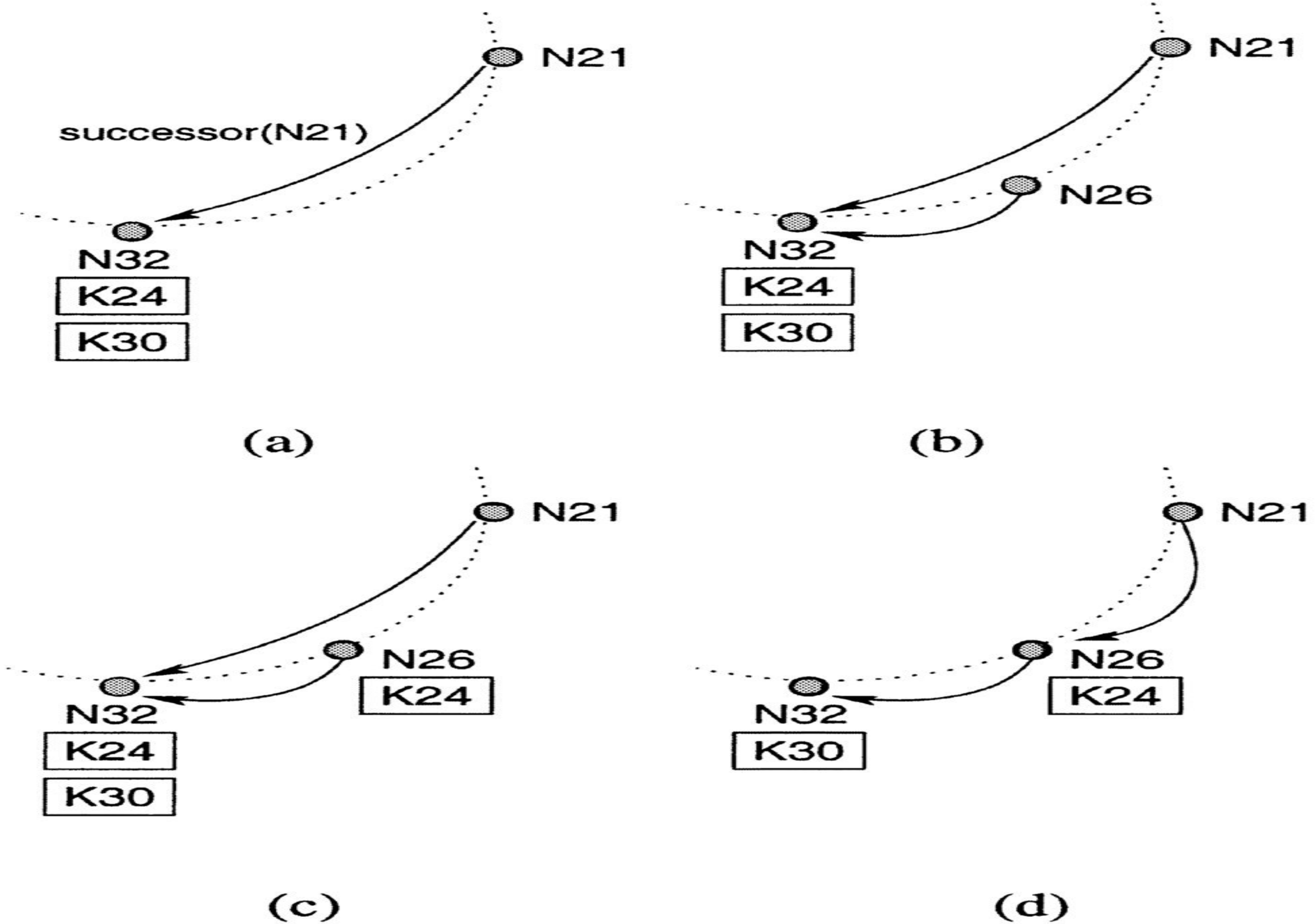
    $n' = closest\_preceding\_node(id)$;

    return $n'.find\_successor(id)$;

Lookup Complexity?

Join/Leaves

Join



N21
successor(N21)
N32
K24
K30
(a)
N21
N26
N32
K24
K30
(b)
N21
N26
K24
N32
K24
K30
(c)
N21
N26
K24
N32
K30
(d)

// join a Chord ring containing node n'.

n.**join**(n')

$\quad$ *predecessor* $=$ **nil**;

$\quad$ *successor* $= n'$.*find_successor*(n);

// called periodically. verifies n's immediate

// successor, and tells the successor about n.

n.**stabilize**()

$\quad$ $x =$ *successor.predecessor*;

$\quad$ **if** $(x \in (n, \mathit{successor}))$

$\quad\quad$ *successor* $= x$;

$\quad$ *successor.notify*(n);

// n' thinks it might be our predecessor.

n.**notify**(n')

$\quad$ **if** (*predecessor* **is nil or** $n' \in (\mathit{predecessor}, n)$)

$\quad\quad$ *predecessor* $= n'$;

```
// 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));
```

Fingers can be out of date!

- Lookups are still <u>correct</u>

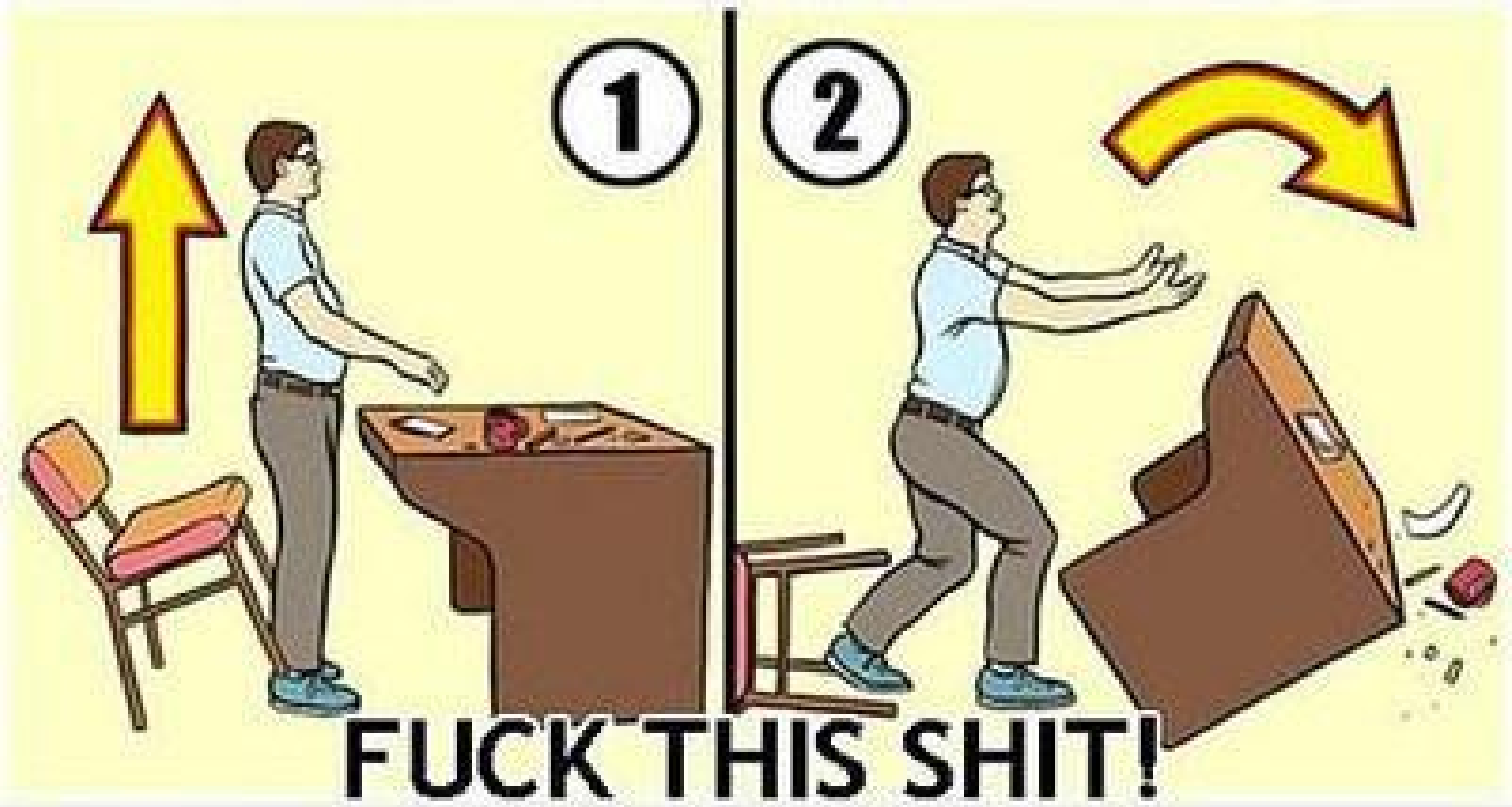
- Degrades to O(N) in the worst case!

How do JOIN's affect operations

- Lookups are correct but may be slow (due to outdated pointers)
- Some lookups may fail
  - Data is in transit
  - Incorrect successor pointers
- SOLUTION:
  - Higher level software should handle it

Leaving the Ring!


TWO WEEKS
NOTICE
memegenerator.net


1
2
FUCK THIS SHIT!

Leaving the Ring (w/ proper 2 weeks notice)

- Notify your predecessor/successor by sending them your successor/predecessor
- Transfer them your data ahead of time (if you are nice)

Failure handling

- The last hop is always from predecessor -> successor
- It is the predecessor’s job to notice if the successor has failed
  - But what next?
- Instead of keeping a single successor, keep a list of successors
- When you notice your successor has failed, notify successor+1, and cut your successor out of the circle

Being accepting of new predecessors

// called periodically. checks whether predecessor has failed.

n.**check_predecessor**()

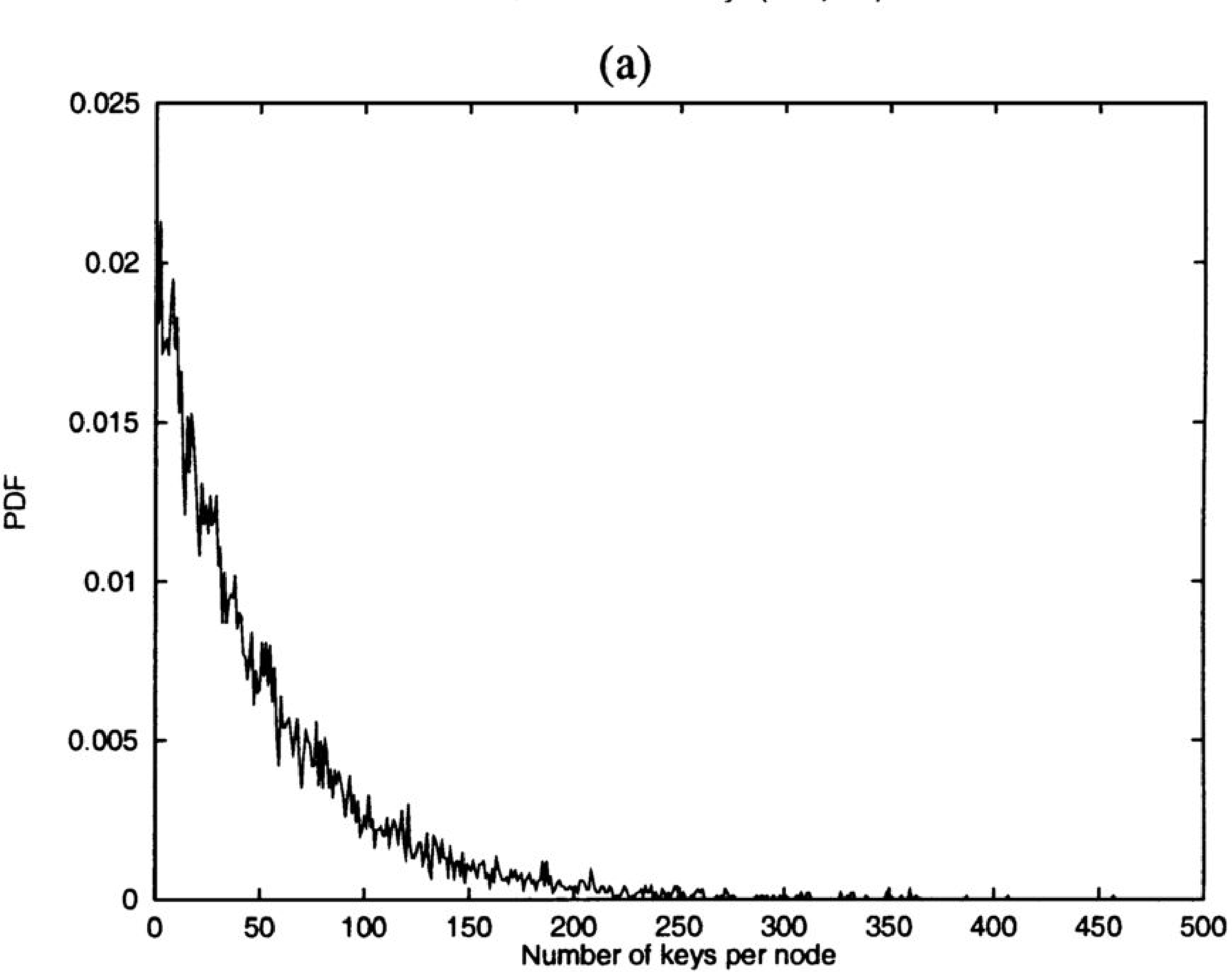
    if (*predecessor* **has failed**)

        predecessor = **nil**;

Chord does not...

- Prevent split brain
- Prevent rings that loop multiple times around
- SOLUTION:
  - Sample the ring topology every so often and fix it (thanks?)
  - Though they do hypothesize, an ordinary series of JOINS/LEAVES do not cause this.

Analysis



- 10^4 nodes
- 5 * 10^5 keys

TABLE II

PATH LENGTH AND NUMBER OF TIMEOUTS EXPERIENCED BY A LOOKUP AS FUNCTION OF THE FRACTION OF NODES THAT FAIL SIMULTANEOUSLY. THE FIRST AND 99TH PERCENTILES ARE IN PARENTHESES. INITIALLY, THE NETWORK HAS 1000 NODES

Fraction of failed nodes	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)
0	3.84 (2, 5)	0.0 (0, 0)
0.1	4.03 (2, 6)	0.60 (0, 2)
0.2	4.22 (2, 6)	1.17 (0, 3)
0.3	4.44 (2, 6)	2.02 (0, 5)
0.4	4.69 (2, 7)	3.23 (0, 8)
0.5	5.09 (3, 8)	5.10 (0, 11)

TABLE III

PATH LENGTH AND NUMBER OF TIMEOUTS EXPERIENCED BY A LOOKUP AS FUNCTION OF NODE JOIN AND LEAVE RATES. FIRST AND 99TH PERCENTILES ARE IN PARENTHESES. THE NETWORK HAS ROUGHLY 1000 NODES

Node join/leave rate (per second/per stab. period)	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)	Lookup failures (per 10,000 lookups)
0.05 / 1.5	3.90 (1, 9)	0.05 (0, 2)	0
0.10 / 3	3.83 (1, 9)	0.11 (0, 2)	0
0.15 / 4.5	3.84 (1, 9)	0.16 (0, 2)	2
0.20 / 6	3.81 (1, 9)	0.23 (0, 3)	5
0.25 / 7.5	3.83 (1, 9)	0.30 (0, 3)	6
0.30 / 9	3.91 (1, 9)	0.34 (0, 4)	8
0.35 / 10.5	3.94 (1, 10)	0.42 (0, 4)	16
0.40 / 12	4.06 (1, 10)	0.46 (0, 5)	15

Credits & Resources

- Chord: https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
- Great slides on Chord: http://slideplayer.com/slide/10698351/
- Great list of papers: http://cseweb.ucsd.edu/classes/sp14/cse223B-a/syllabus.html