Structured **P2P** Networks

Niels Olof Bouvin

Distributed Hash Tables

- **DHTs are designed to be *infrastructure* for other applications**
- **General concept**
  - Assign peers IDs evenly across an ID space (e.g., $[0, 2^n-1]$)
  - Assign resources IDs in the same ID space, and associate resources with the closest (in ID space) peer
  - Distance = distance in ID space
  - Peers have **broad** knowledge of the network, and **deep** knowledge about their neighbourhood
  - Arrange peers in a network that they easily (iteratively or recursively) can be found
  - Searching for a resource and searching for a peer become the same

Distributed Hash Tables

- **Challenges**
  - Routing information must be distributed – no central index
  - How is the routing information created and maintained?
  - How are peers inserted into the network? How do they leave? How are resources added?
  - Resources are stored at their closest peer
    - resources should be relatively small...

Overview

- Chord
- Pastry
- Kademlia
- Conclusions

Chord

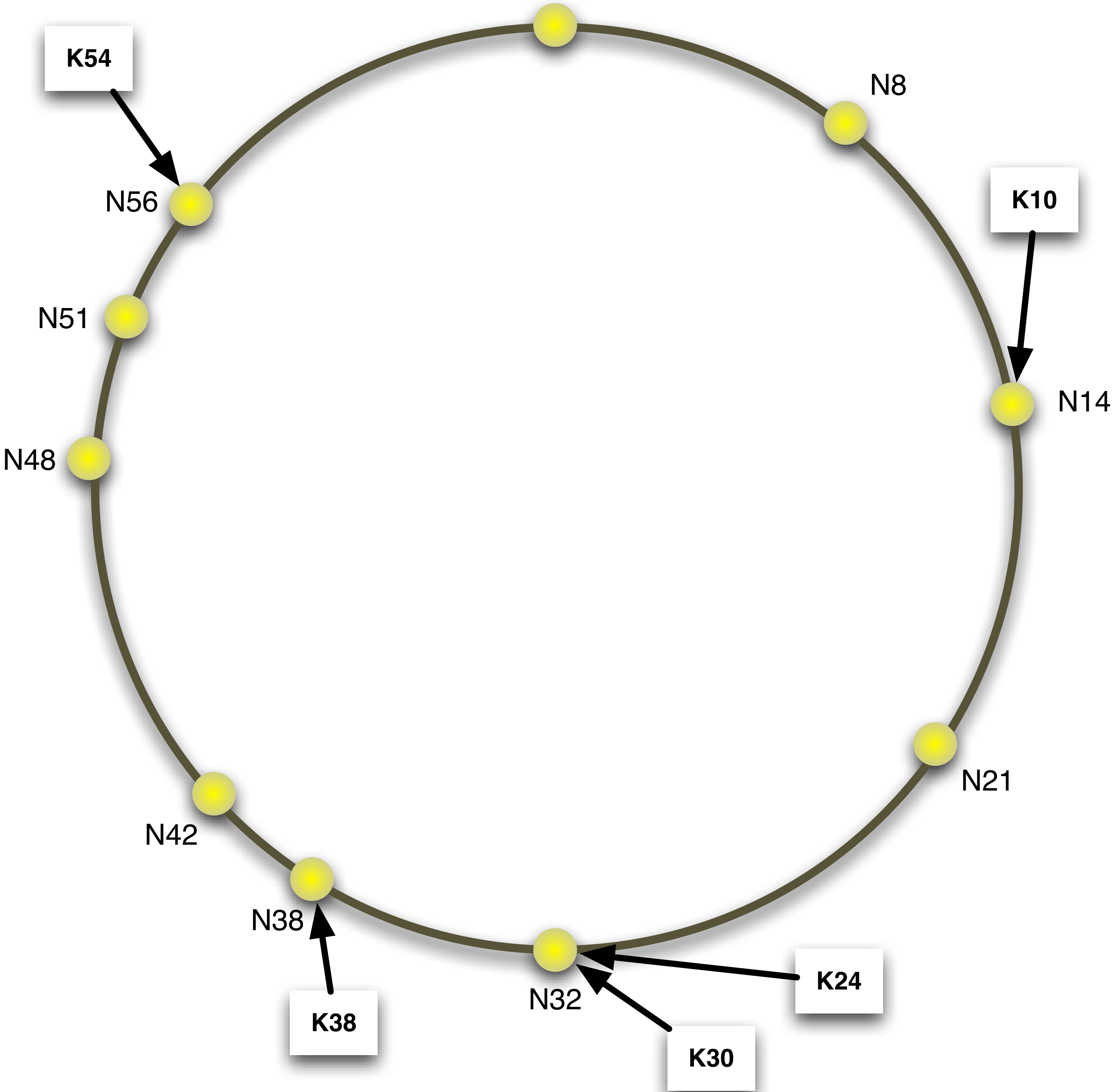
- **One operation:**
  - IP address = lookup(key): Given a key, find node responsible for that key
- **Goals**
  - load balancing, decentralisation, scalability, availability, flexible naming
  - performance and space usage:
    - lookup in O(log N)
    - each node needs information about O(log N) other nodes

Use of Hashing in Chord

- **Keys are assigned to nodes with hashing**
  - good hash function balances load
- **Nodes and keys are assigned m-bit identifiers**
  - using SHA-1 on nodes' IP addresses and on keys
  - m should be big enough to make collisions improbable
- **"Ring-based" assignment of keys to nodes**
  - identifiers are ordered on an identifier circle modulo 2^m
  - a key *k* is assigned to the first node *n* where $ID_n \geq ID_k$: $n = successor(k)$

Hash function?

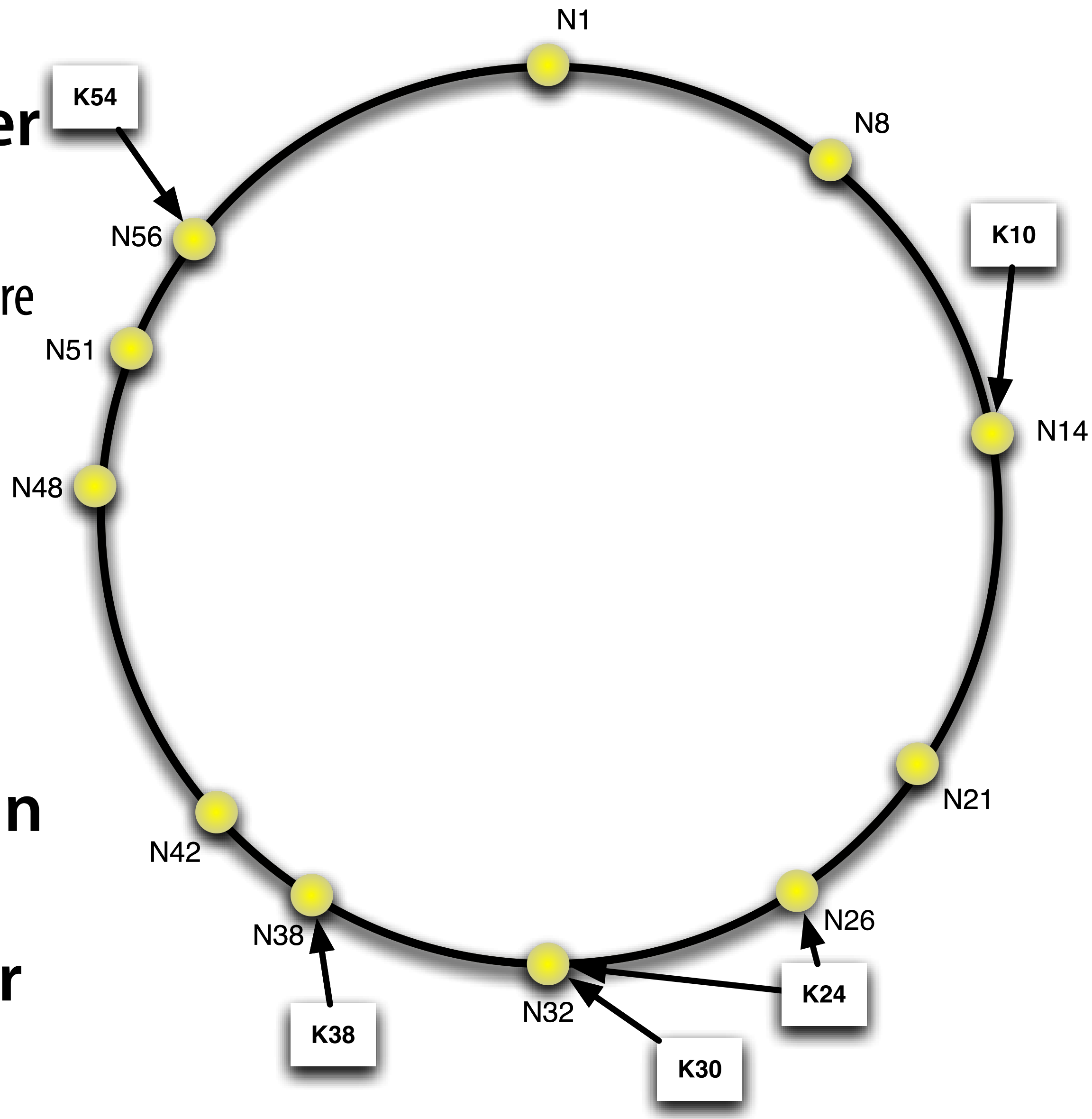
- ***"A hash function is any function that can be used to map data of arbitrary size to data of fixed size"***
  - e.g., from some data to a number belonging to some range
  - good hash functions generate a uniform distribution of numbers across its range
- ***Cryptographic* hashes (such as SHA-1, SHA-256, etc) are excellent hash functions where it is *very* hard to guess the data that led to a specific hash value**
  - even tiny changes in data leads to dramatically different hash values
  - the range is usually *very* large, e.g. SHA-1 is $[0, 2^{160} = 1{,}46 \times 10^{48}]$
  - (note that these days SHA-1 is no longer considered safe, so use SHA-256 instead)



A ring consisting of 10 nodes storing 5 keys

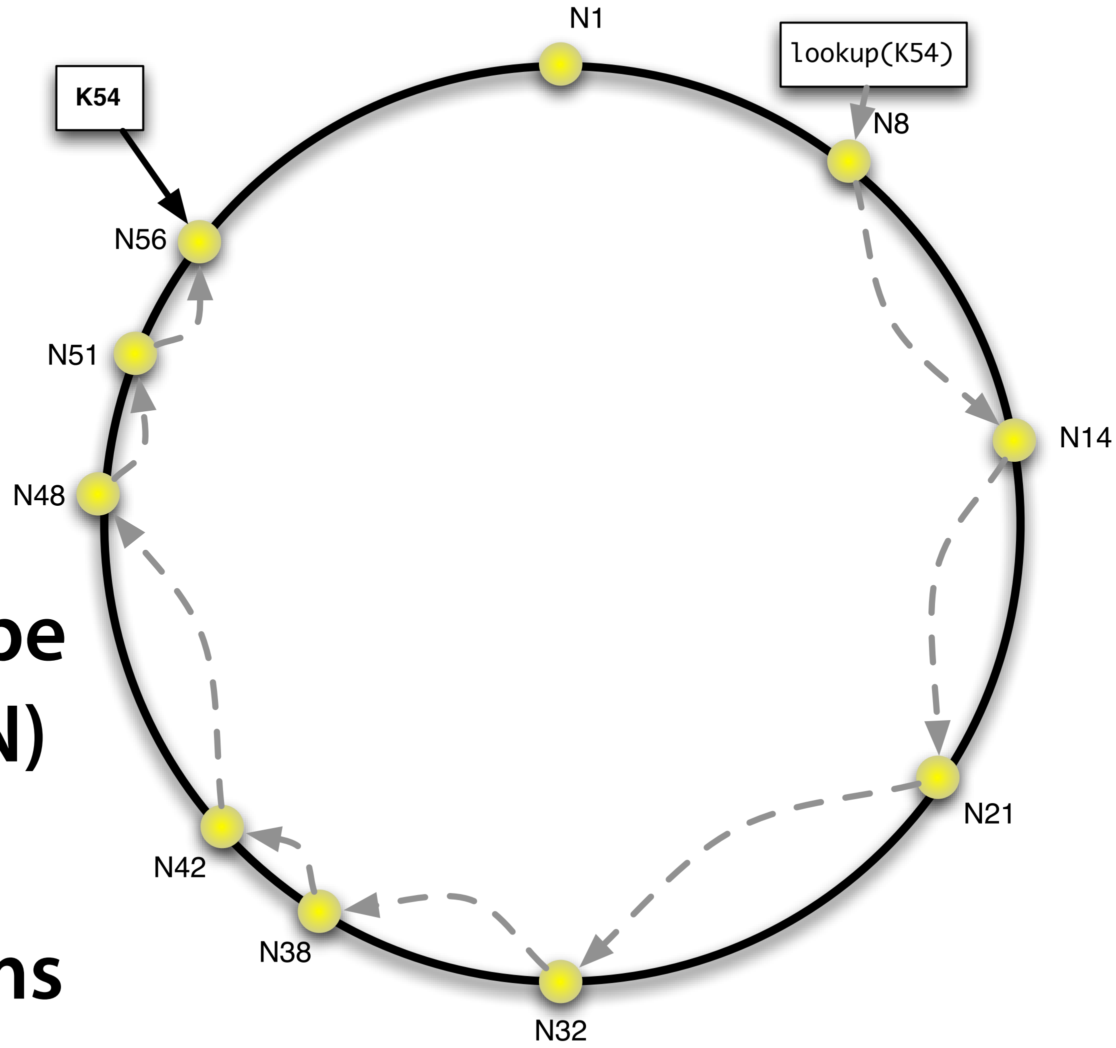
Key Allocation in Chord

- **Designed to let nodes enter and leave network easily**
  - Node n leaves: all of n's assigned keys are assigned to successor(n)
  - Node n joins: keys k ≤ n assigned to successor(n) are assigned to n
  - Example: N26 joins ⇒ K24 becomes assigned to N26
- **Each physical node may run a number of virtual nodes, each with its own identifier to balance the load**



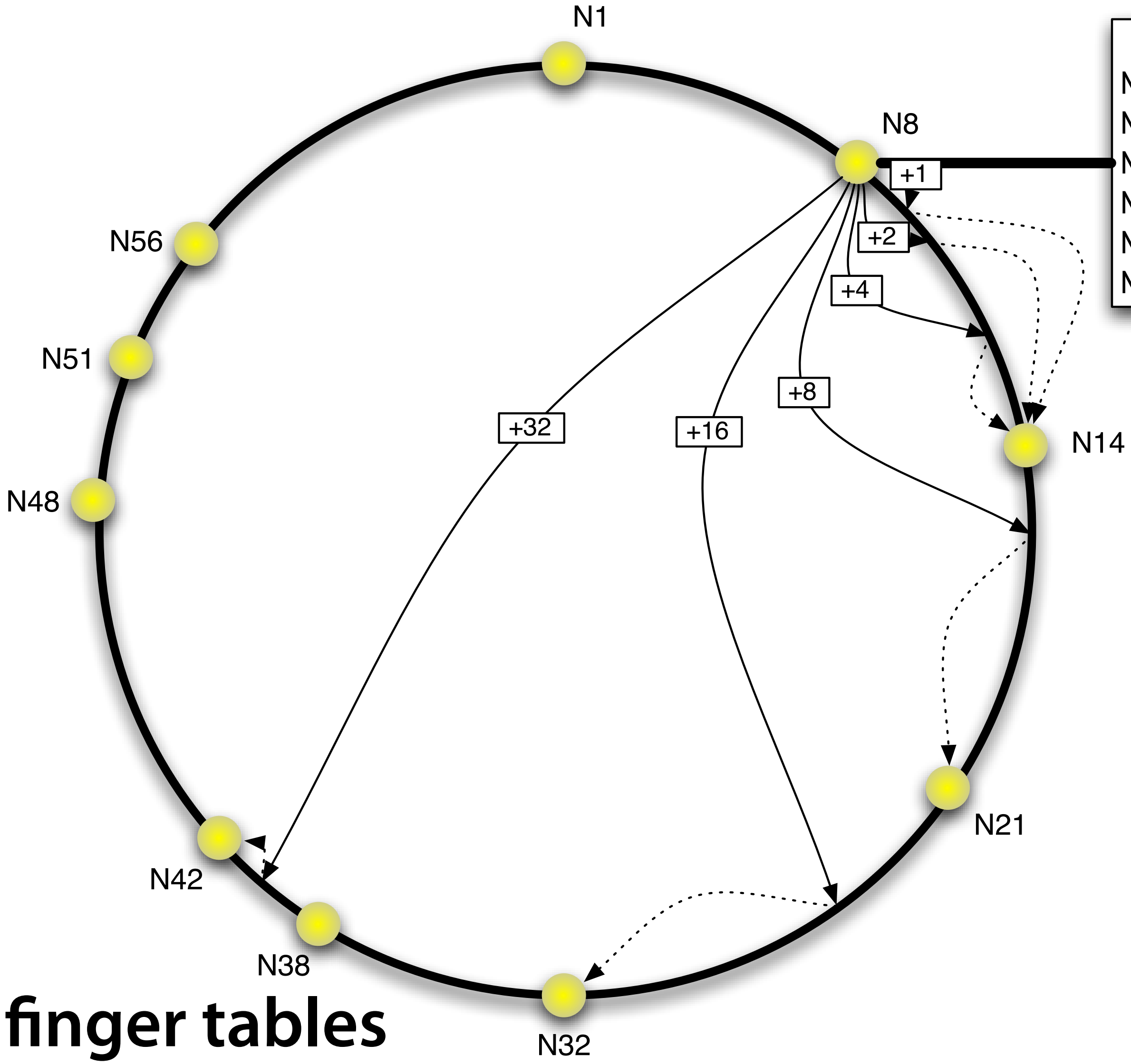
Simple (Linear) Key Location

```
#ask node n to find the successor of id
n.find_successor(id)
    if n < id ≤ successor
        return successor
    else
        #forward query around circle
        return successor.find_successor(id)
```



- **Simple key location can be implemented in time O(N) and space O(1)**
- **Example: Node 8 performs a lookup for Key 54**

Scalable Key Location



Finger table

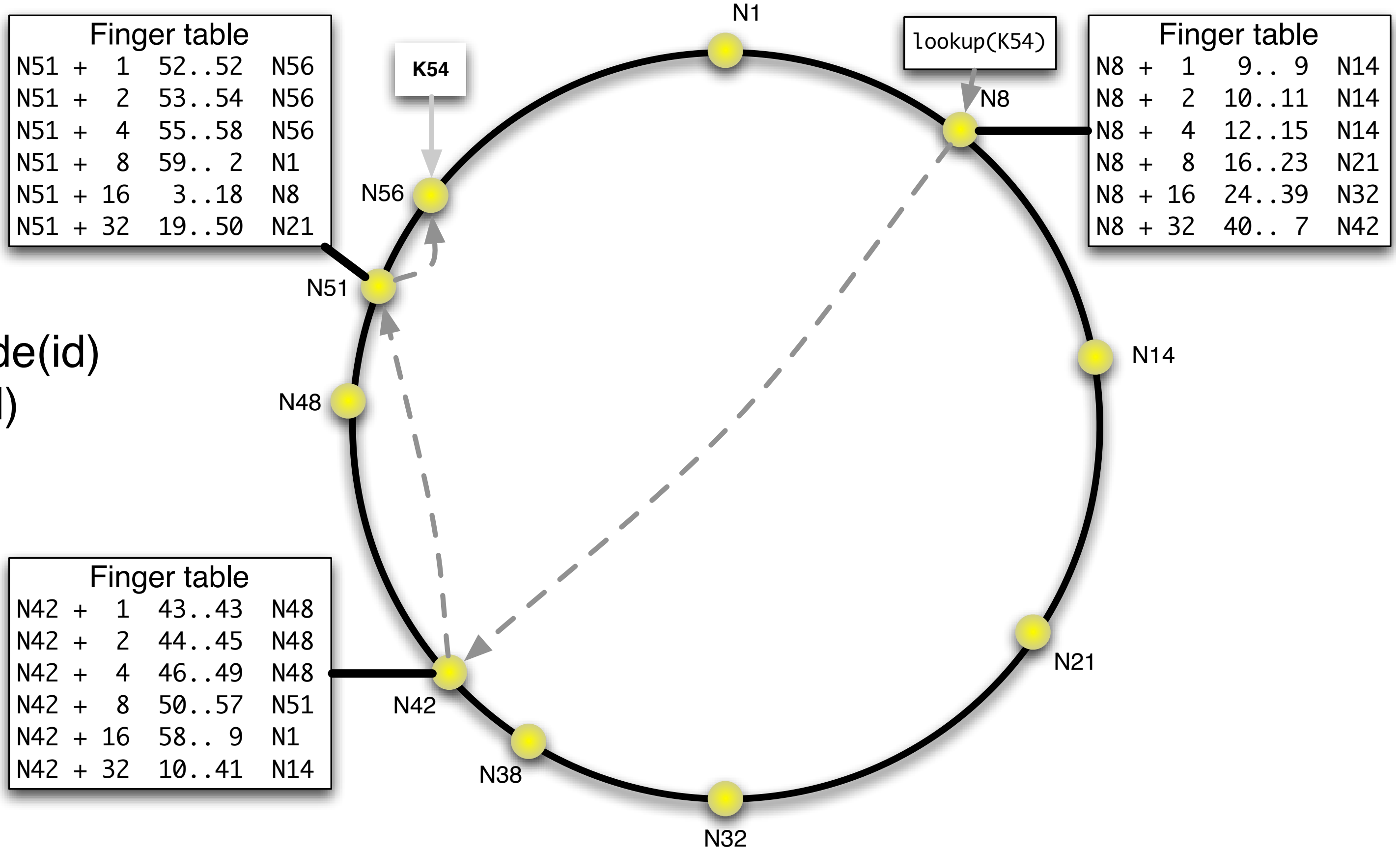
N8 +	1	9.. 9	N14
N8 +	2	10..11	N14
N8 +	4	12..15	N14
N8 +	8	16..23	N21
N8 +	16	24..39	N32
N8 +	32	40.. 7	N42

- **Uses finger tables**
  - n.finger[i] = find_successor($n + 2^{i-1}$), $1 \leq i \leq m$

Scalable Key Location

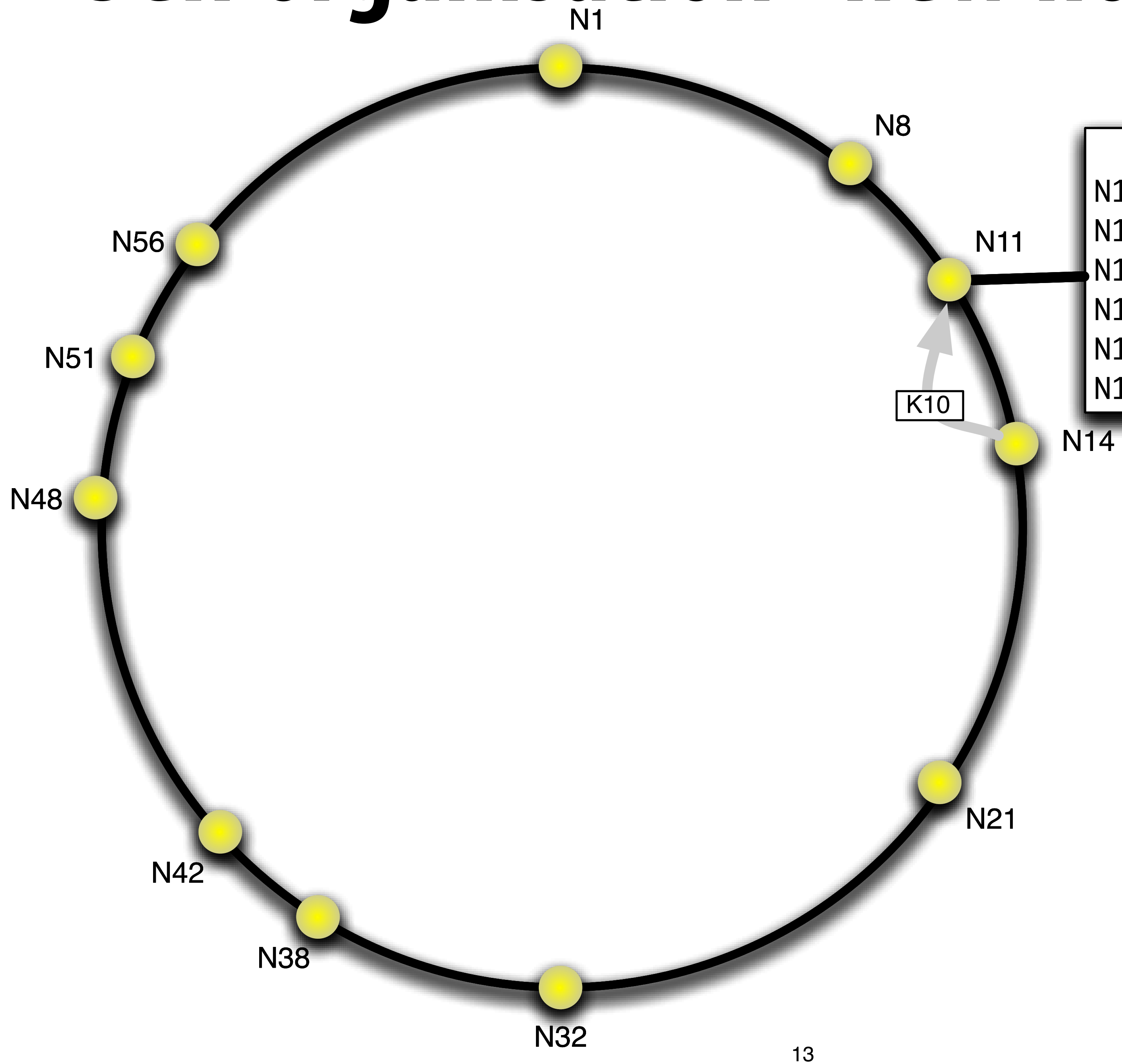
```
n.find_successor(id):
    if n < id ≤ successor
        return successor
    else
        n’ = closest_preceding_node(id)
        return n’.find_successor(id)

n.closest_preceding_node(id):
    for i = m downto 1
        if n < finger[i] < id
            return finger[i]
    return n
```



- **If successor not found, search finger table to find n’ whose ID most immediately precedes id**
- **This node will know the most about n’ of all nodes in the finger table**

Self organisation - new node arrival



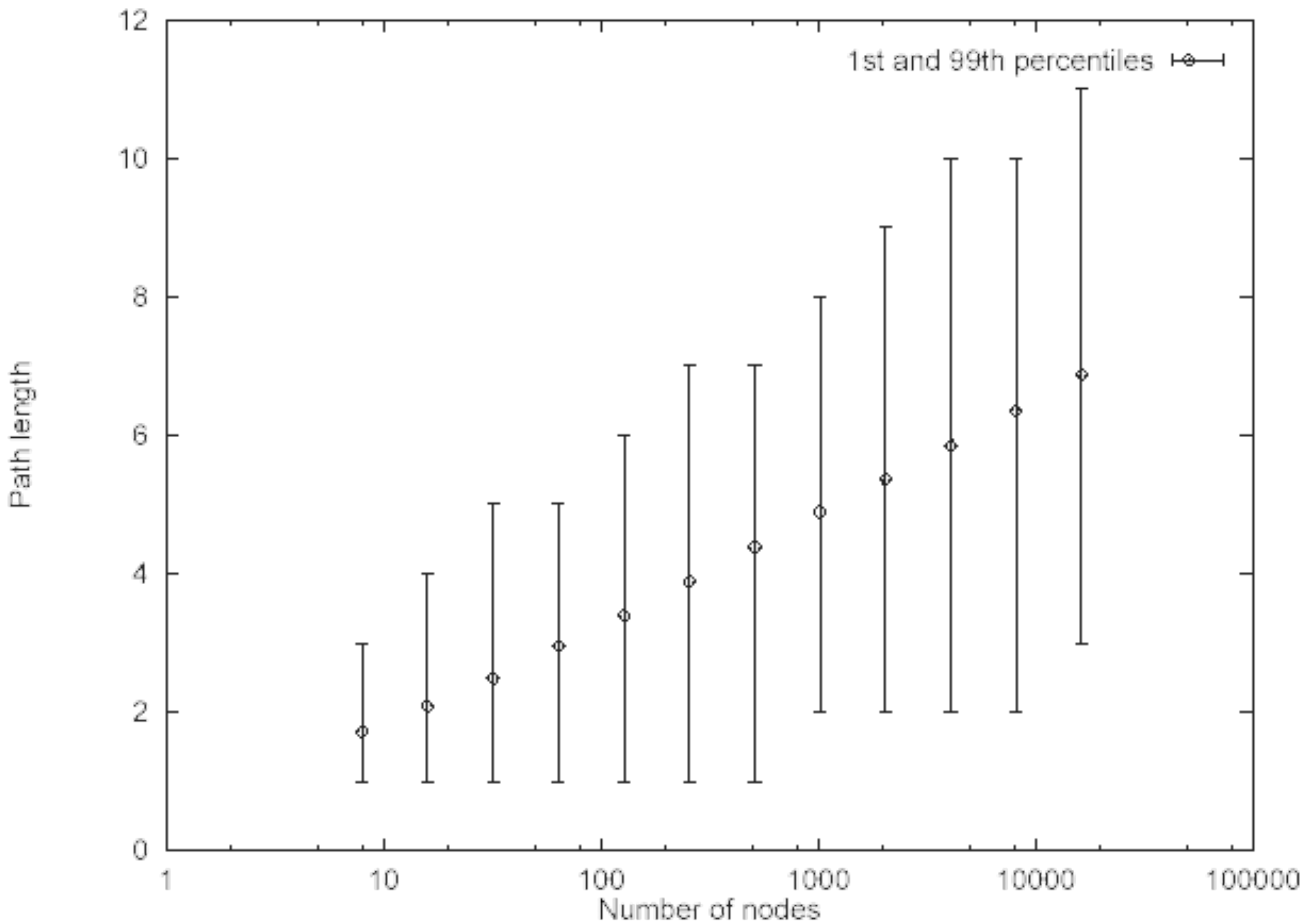
Finger table

N11	+	1	12..12	N14
N11	+	2	13..14	N14
N11	+	4	15..18	N21
N11	+	8	19..26	N21
N11	+	16	27..42	N32
N11	+	32	43..10	N48

Self organisation - node failures

- **Chord maintains successor lists to cope with node failures**
  - node leaving could be viewed as a failure
  - if nodes leaves voluntarily, it may notify its successor and predecessor, allowing them to gracefully update their tables
  - otherwise, Chord can on demand use the successor lists to rebuild the information

Results–Path Length/#Nodes



Summary

- **Decentralised lookup of nodes responsible for storing keys**
  - based on distributed, consistent hashing
  - performance and space in O(log N) for stable networks
  - simple; provable performance and correctness
  - too simple; does not consider locality or strength of peers
    - though they do outline a solution using nearest (in IP space) nodes for finger tables rather than exact matches (in ID space)

Overview

- Chord
- **Pastry**
- **Kademlia**
- **Conclusions**

Pastry

- **Aim: Effective, distributed object location and routing substrate for P2P networks**
  - **Effective**: O(log N) routing hops
  - **Distributed**: no servers, routing and location distributed to nodes, only limited knowledge at nodes(routing tables size O(log N))
  - **Substrate**: not an application itself, rather it provides Application Program Interface (API) to be used by applications. Runs on all nodes joined in a Pastry network
  - Each node has a unique identifier (nodeId) (128 bits)

Pastry API

- nodeId = pastryInit(Credentials, Application) make the local node join/create a Pastry network. Credentials are used for authorisation. A callback object is passed through Application
- route(msg, key) routes a message to the live node with nodeId numerically closest to the key (at the time of delivery)

- Application interface to be implemented by applications using Pastry
  - deliver(msg, key) called on the application at the destination node for the given id
  - forward(msg, key, nextId) invoked on applications when the underlying node is about to forward the given message to the node with nodeId = nextId.

Node Identifiers

- **Each node is assigned a 128 bit nodeId**
  - nodeIds are assumed to be uniformly distributed in the 128 bit ID space ⇒ numerically close nodeIds belong to diverse nodes
  - nodeId = cryptographic hash of node's IP address

Assumptions and Guarantees

- **Pastry can route to numerically closest node in $\log_{2^b} N$ steps (b is a configuration parameter)**
- **Unless |L|/2 (|L| being a configuration parameter) adjacent nodeIds fail concurrently, eventual delivery is guaranteed**
  - such failure is *very* unlikely
- **Join, leave in O(log N)**
- **Maintains locality based on application-defined scalar proximity metric**

Routing table

NodeId 10233102

Leaf set (SMALLER | LARGER)

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

|L|/2 numerically closest smaller nodeIds

|L|/2 numerically closest larger nodeIds

ceiling($\log_{2^b} N$) rows with 2^b-1 entries

each entry in row n shares a prefix of length n with the present node

nodes chosen according to proximetry metric

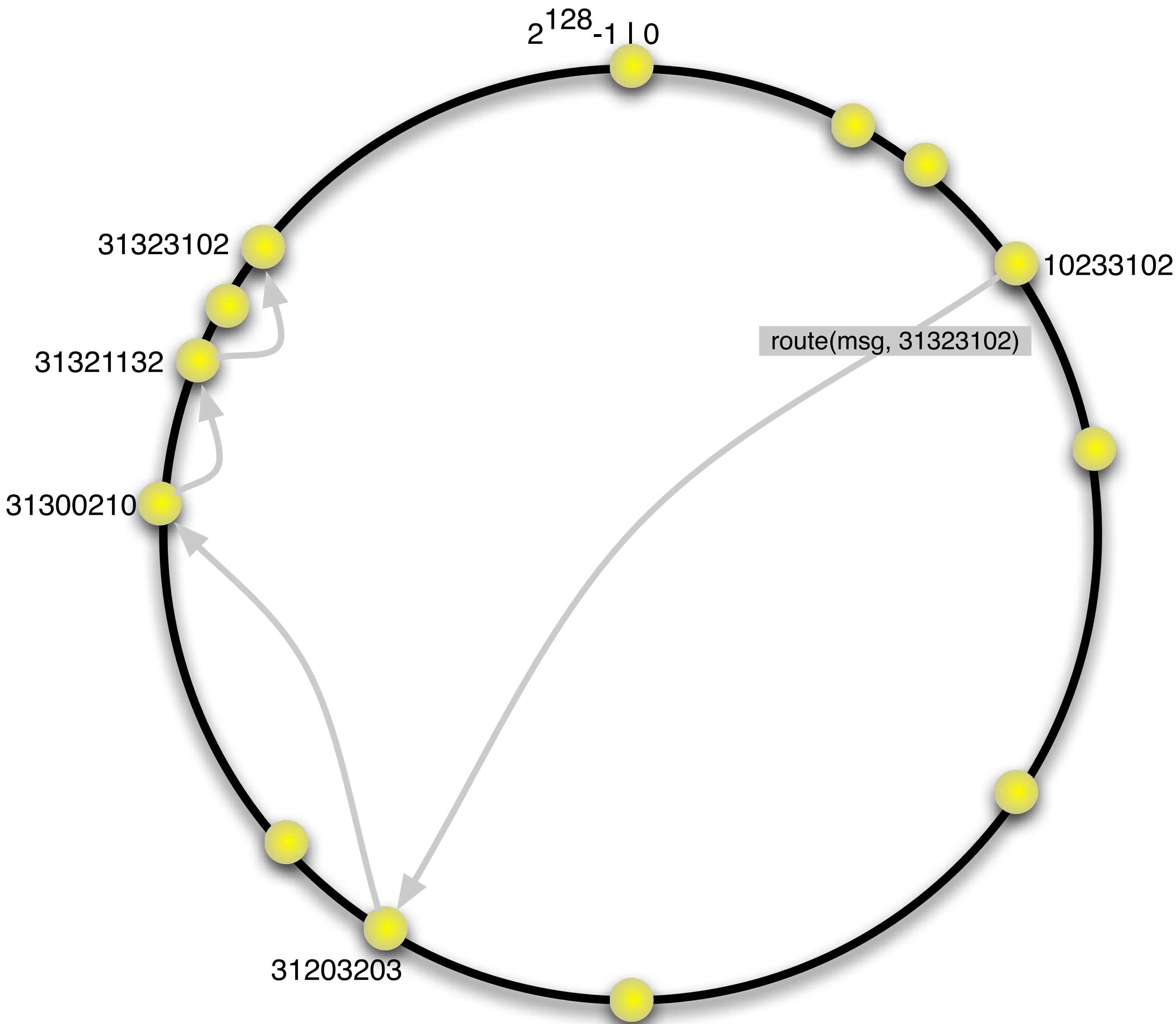
|M| closest nodes (according to proximetry metric)

b = 2; L = 8; M = 8

Pastry routing

- **The node first checks if the key falls within the range of its leaf set. If yes, forward the message to the destination node.**
- **If not, use routing table to forward the message to a node that shares a common prefix with the key by at least one digit.**
- **In some rare cases, the appropriate entry is empty or unreachable, then the message will be forwarded to a known node**
  - that has a common prefix with the key at least as good as the local node (**and is numerically closer**)

Routing in Pastry



(Expected) Performance

1. Either: Destination one hop away
2. Or: The set of possible nodes with a longer prefix match is reduced by 2^b (i.e., one digit)
3. Or: Only one extra routing step is needed (with high probability)
    - given accurate routing tables, the probability for 3) is the probability that a node with the given prefix does not exist and that the key is not covered by the leaf set

(Expected) Performance

- **Thus, expected performance is O(log N)**
  - The worst case routing step may be linear to N. (when many nodes fail simultaneously)
- **Eventual message delivery is guaranteed unless |L|/2 nodes with consecutive nodeIds fail simultaneously**
  - *highly* unlikely, as leafset nodes are widely distributed due to uniform hashing

Self organisation – node arrival

- New node, X, needs to know existing, nearby node, A, (can be achieved using, e.g., multicast on local network)
- X asks A to route a "join" message with key equal to X
- Pastry routes this message to node Z with nodeId numerically closest to X
- All nodes en-route to Z returns their state to X

Self organisation – node arrival

- **X updates its state based on returned state:**
  - neighbourhood set = neighbourhood set of A
  - leaf set is based on leaf set of Z (since Z has nodeId closest to nodeId of X)
  - rows of routing table are initialised based on rows of routing tables of nodes visited en-route to Z (since these share increasing common prefixes with X)
- **X calibrates routing table and neighbourhood set based on data from the nodes referenced therein**
- **X sends its state to all the nodes mentioned in its leaf set, routing table, and neighbour list**
- **$O(\log_{2^b} N)$ messages exchanged**

Locality

- **Routing performance is based on small number of routing hops – and "good" locality of routing with respect to underlying network**
  - Pastry relies on a scalar proximity metric (e.g., number of IP routing hops, geographical distance, or available bandwidth)
- **Applications are responsible for providing proximity metrics**
- **Pastry assumes the triangle inequality holds**
- **Join protocol maintains locality invariant**

Locality – Upon node arrival

- **Assume the system holds locality property before the new node arrivals**
- **Assume A is actually near X, so the state updated from A should also hold the locality property**
- **The states updated from the routing path also tend to be close to X – at least in the beginning**
  - as we progress, there will be fewer and fewer candidate nodes to choose from
- **A second stage which updates node X's routing table with closer nodes is used to improve the locality property**

Locality

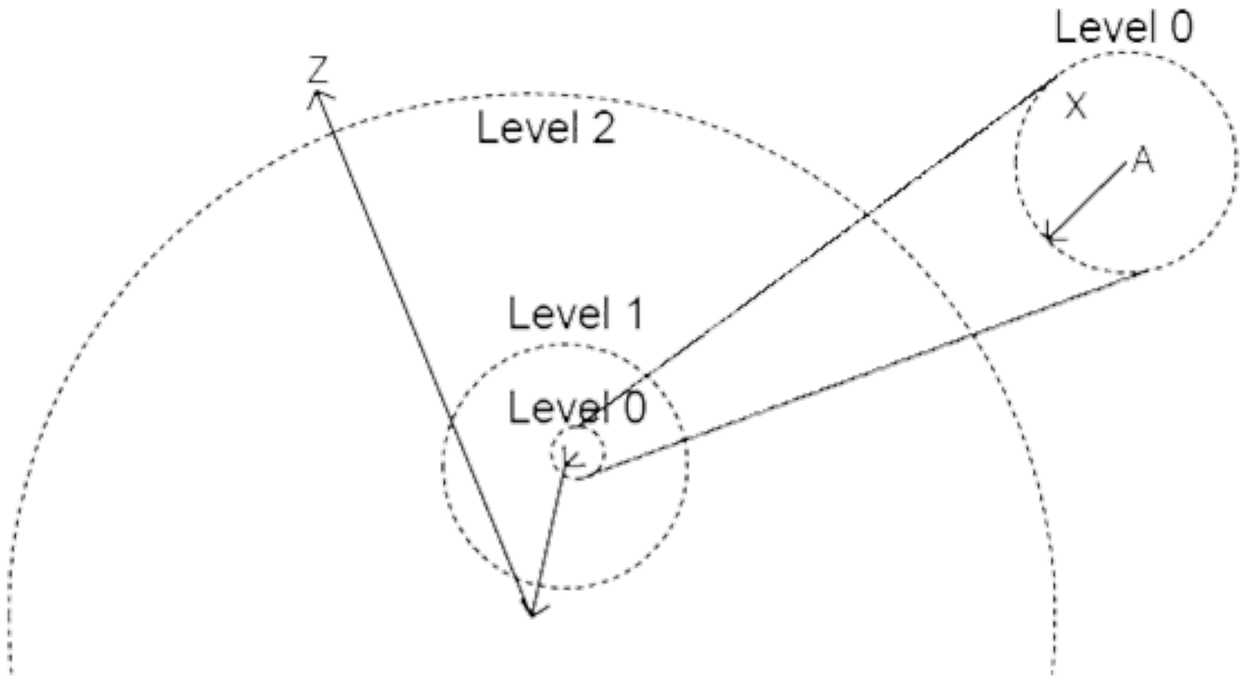


Fig. 2. Routing step distance versus distance of the representatives at each level (based on experimental data). The circles around the n-th node along the route from A to Z indicate the average distance of the node's representatives at level n. Note that X lies within each circle.

Self Organisation – Node Failure

- **Repair of leaf set**
  - contact the live node with the largest index on the side of the failed node and get leaf set from that node
  - returned leaf set will contain an appropriate node to insert
  - this works unless |L|/2 nodes with adjacent nodeIds have failed

Self Organisation – Node Failure

- **Repair of routing table**
  - contact other node on the same row to check if this node has a replacement node (the contacted node may have a replacement node on the same row of its routing table)
  - if not, contact node on next row of routing table

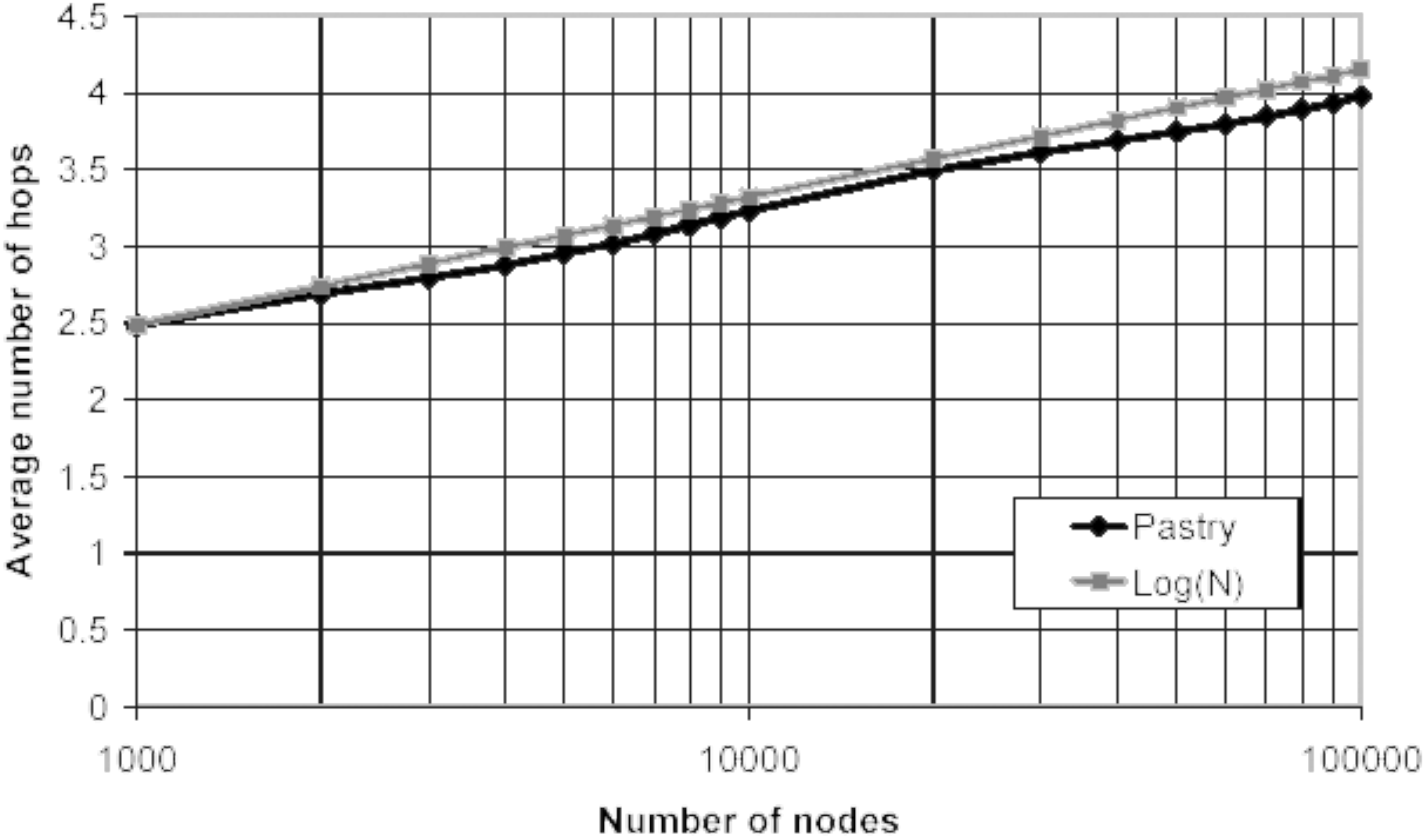
Self Organisation – Node Failure

- **Repair of neighbourhood set**
  - neighbourhood set is normally not used in routing ⇒ contact periodically to check for liveness
  - if a neighbour is not responding, check with live neighbours for other close nodes

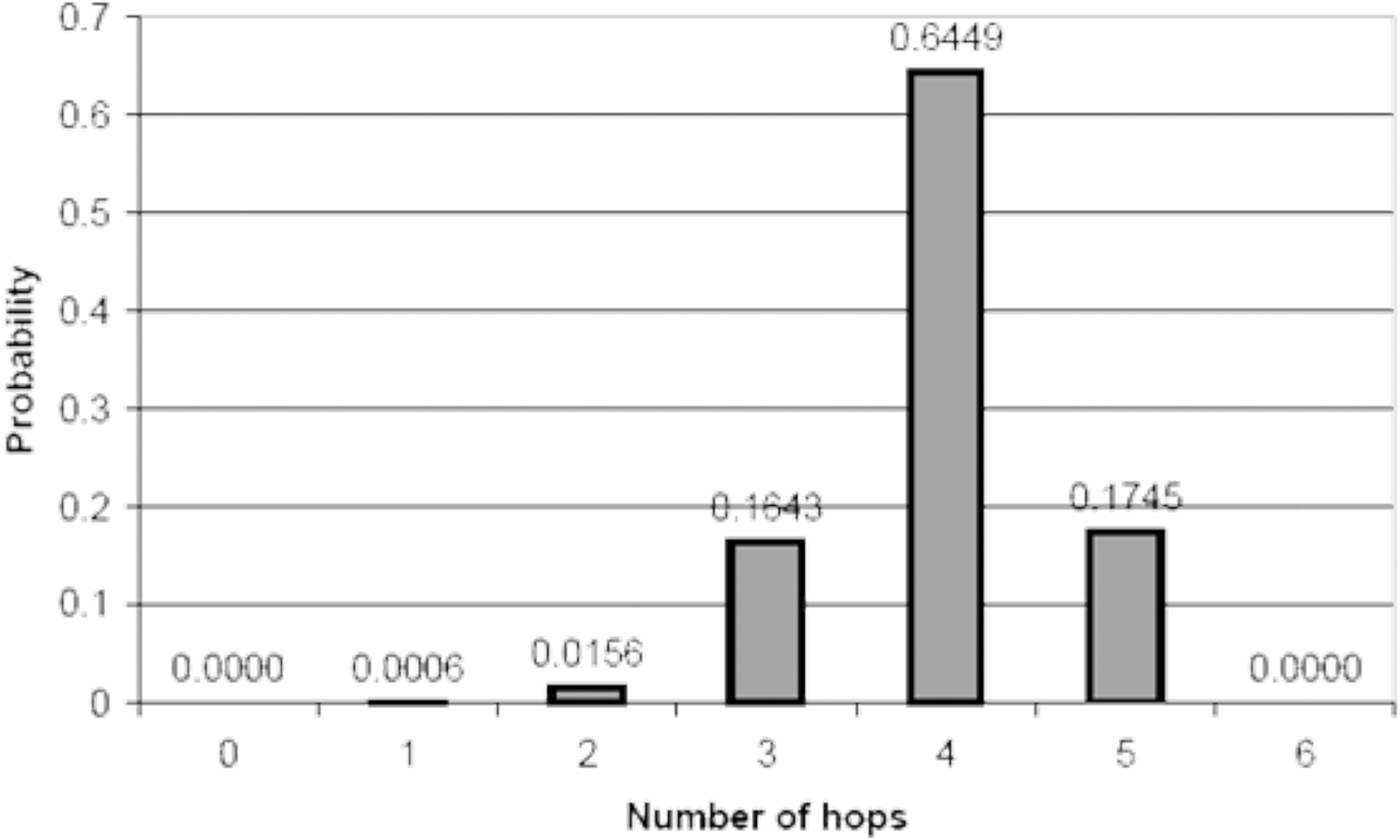
Fault tolerance and malicious peers?

- **Choose *randomly* between nodes satisfying the criteria of the routing protocol**
- **A message can be forwarded to a node with longer common prefix or same common prefix but numerically closer**
  - randomly select a node from the nodes that satisfy the criterion described above
  - thus the routing is not deterministic, and it is possible to avoid bad nodes

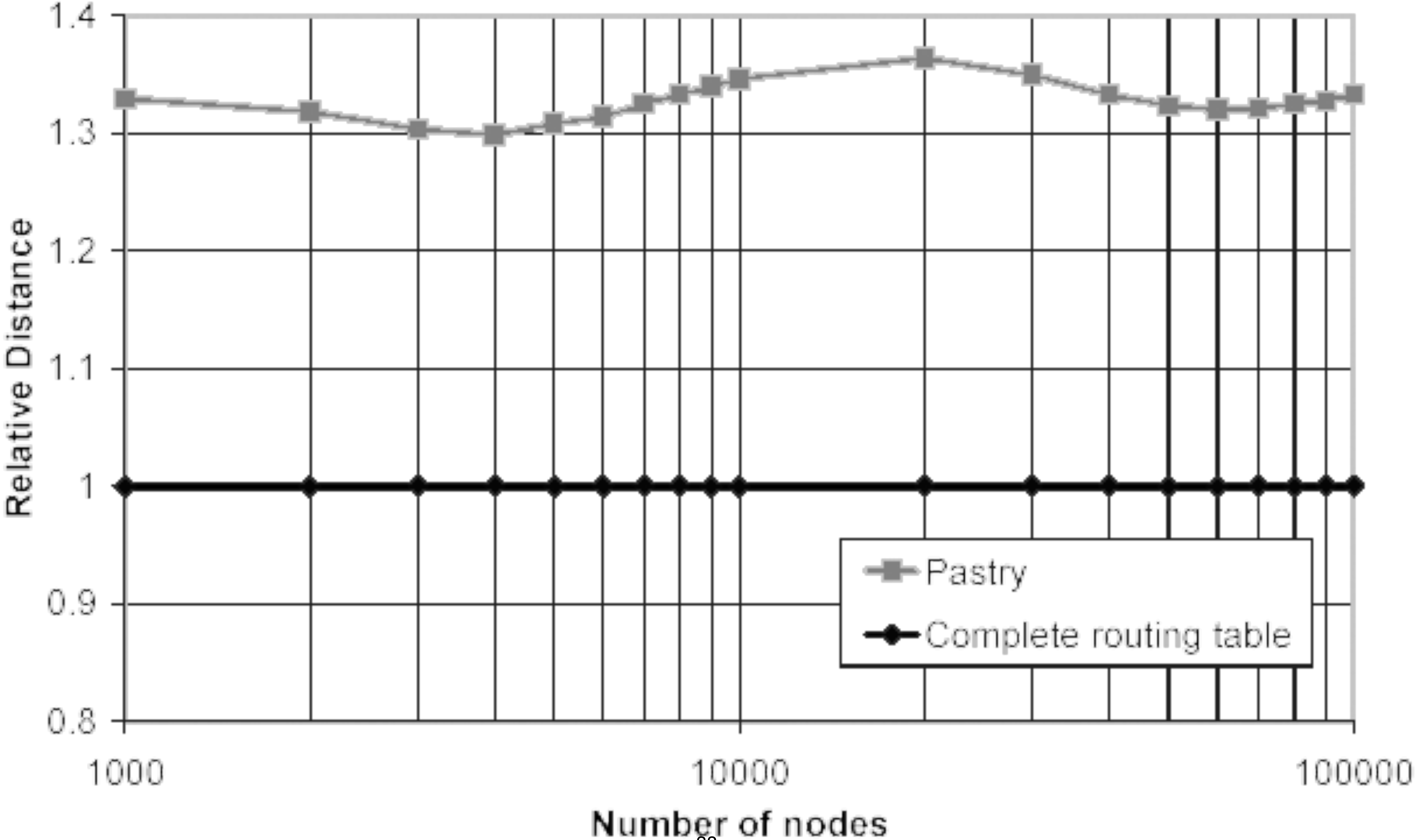
Routing Performance



Routing Distribution



Routing Distance compared to Optimal Routing



Summary

- **Pastry is a P2P content location and routing substrate**
  - structured overlay network
  - usable for building various P2P application
- **Applications built on top of Pastry**
  - SCRIBE: group communication/event notification
  - PAST: archival storage
  - SQUIRREL: co-operative Web caching
- **Space and time requirements (expected) in O(log N), N = number of nodes in network**
- **Takes locality into account**

Overview

- Chord
- Pastry
- **Kademlia**
- **Conclusions**

Kademlia: A Peer-to-Peer Information System Based on the XOR Metric

- **Distributed Hash Table**
  - NodeIDs and keys based on SHA-1 (160 bits)
- **Routing done by halving the ID-space distance in each routing step**
  - Similar to Pastry's routing table routing (prior to leaf node)
- **Routing done in O(log N), space used O(log N)**

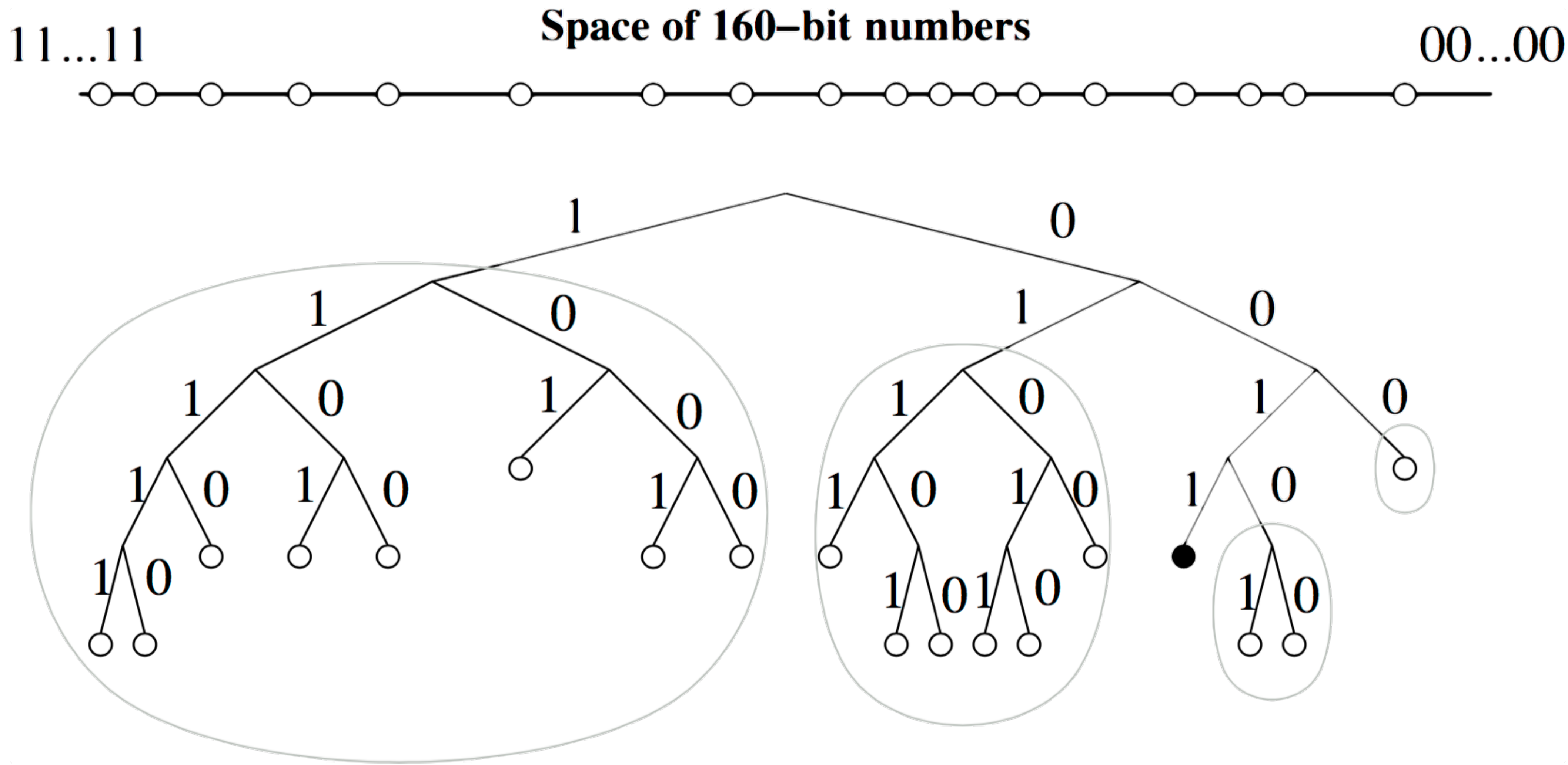
Critique of other systems

- **Chord**
  - Finger tables only forward looking
  - I.e., messages arriving at a peer tell it nothing useful – knowledge must be gained explicitly
  - Separate track of control message exchanges
  - Rigid routing structure
  - Locality difficult to establish
- **Pastry**
  - Complex routing algorithm
  - First routing table, then leaf set
  - Maintains three different tables: leaf, routing and neighbour

Aspects of Kademlia

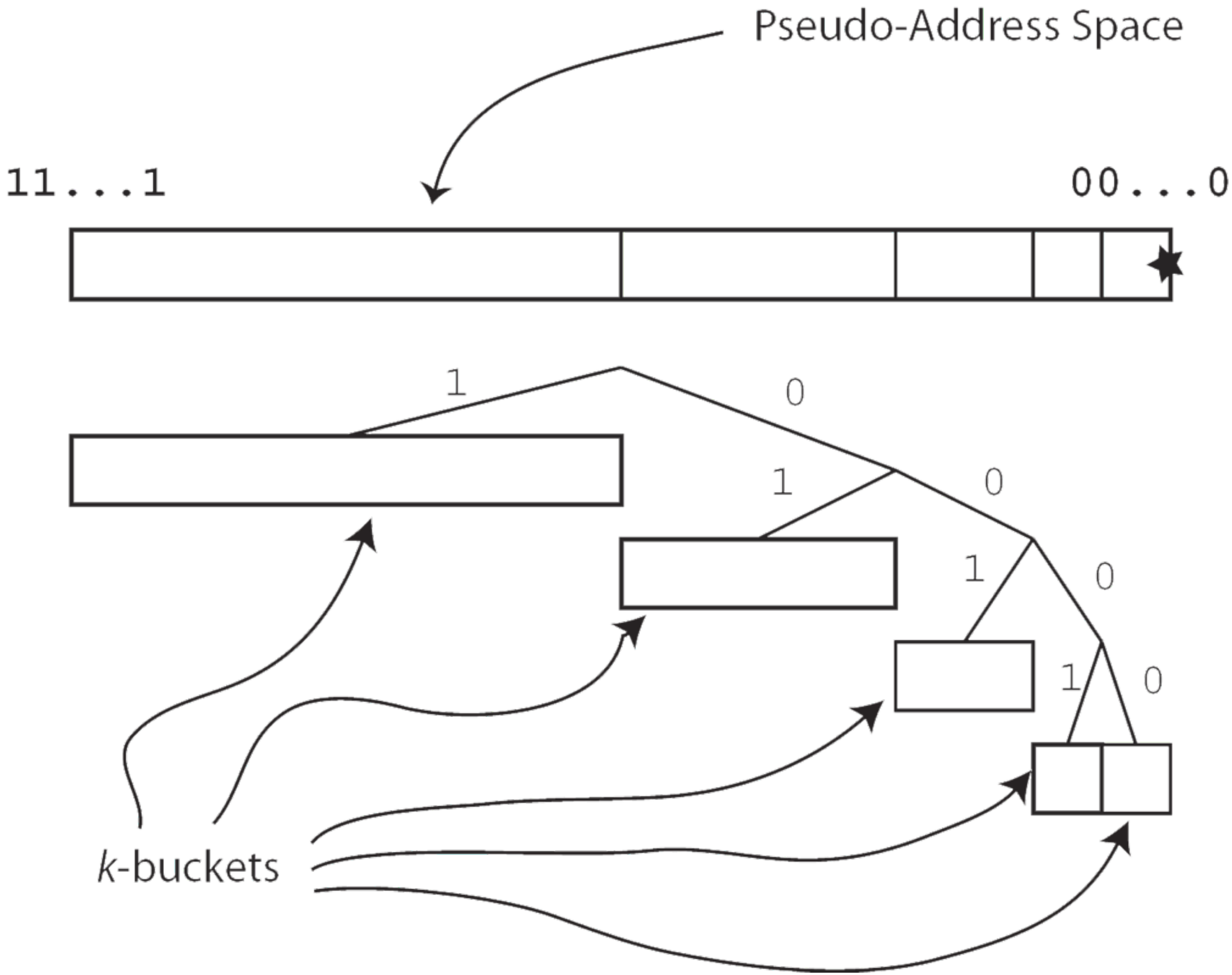
- **All IDs are 160 bits long, found with SHA-1**
  - i.e., uniform distribution, etc
- **To navigate this key space, Kademlia uses XOR**
  - d(X, Y) = X XOR Y; d(X, Y) = d(Y, X)
  - intuition: higher order difference = longer distance
- **A Kademlia routing table stores 160 k-buckets**
  - the i^{th} k-bucket contains nodes within a XOR distance of 2^i to 2^{i+1} from itself (so the i^{th} bit is significant)
  - up to k nodes in each bucket, ordered by liveness (most recently seen at tail)
    - thus, once again, more complete knowledge of 'close' peers, but still knowledge about the rest of the world

Kademlia routing table

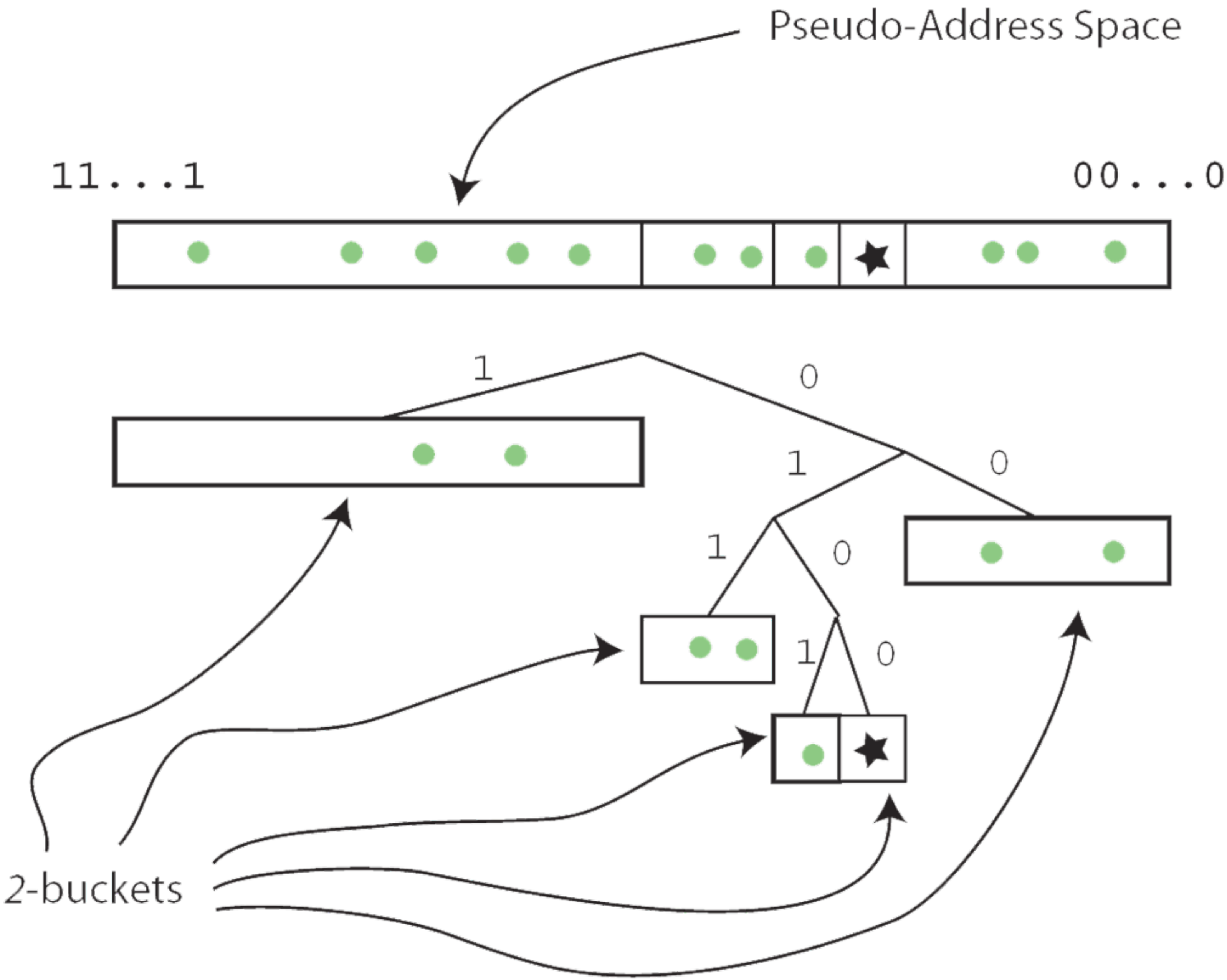


- **Peer 0011 (•) must know some peers in the highlighted groups — all different prefixes to itself**

Kademlia routing table



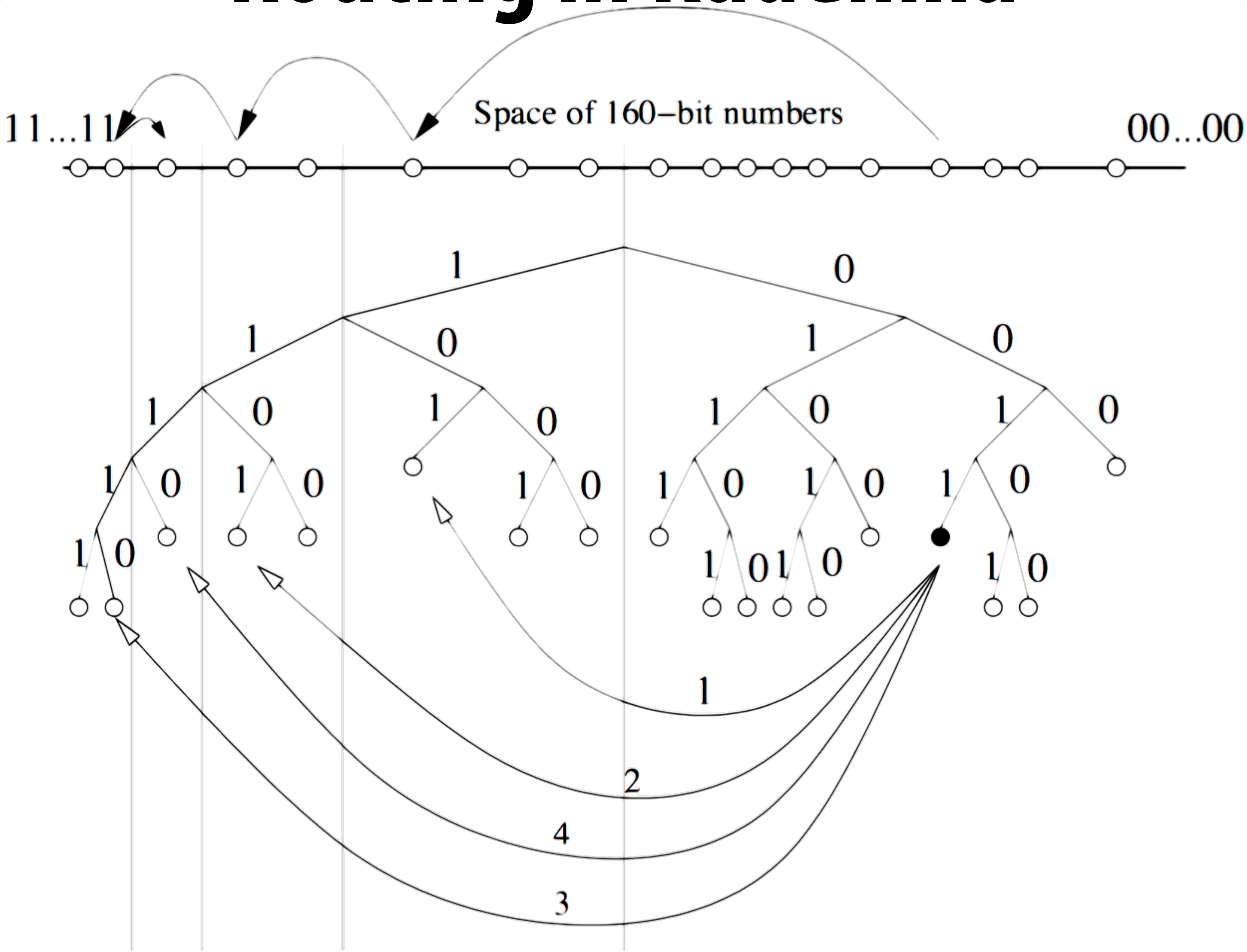
Kademlia routing table



Locating a destination

- **Given a destination, use the (XOR) distance from ourselves to find the matching k-bucket**
- **Contact nodes in that k-bucket to get even closer nodes**
  - if there are not enough nodes in the bucket, use the nearest
- **Repeat until the k closest nodes have been found**

Routing in Kademlia



- **Reaching 1110 from 0011. 0011 knows initially 101**

Operations in Kademlia

- PING
- STORE
- FIND_NODE
- FIND_VALUE

FIND_NODE

- **$FIND\_NODE_n(id)$**
  - returns the k closest nodes to an ID that n knows
- **Iterative process:**
  - $n_0 = origin$
  - $N_1 = FIND\_NODE_{n_0}(ID)$
  - $N_2 = FIND\_NODE_{n_1}(ID)$
  - …
  - $N_m = FIND\_NODE_{n_{m-1}}(ID)$
- **The node can choose any peer among the returned k nodes for the next step**
- **Lookup terminates when k closest nodes have responded**

FIND_VALUE

- **$FIND\_VALUE_n$(key)**
  - works like FIND_NODE, unless n knows the value in which case the value is returned
  - if one of the k closest nodes does not have the value, the requester will store it there

Maintaining routing tables

- **Upon communication with another node**
  - Check the appropriate k-bucket
    - if already there, move to tail
    - if there is room, insert at tail
    - if unknown, and least recently seen node is unresponsive, replace with new node (and move to tail)
    - else: ignore node
  - Thus, the routing tables are populated, and old, active nodes are given preferential treatment
  - Implementation optimization: keep new peers in cache replacement list; replace only member of k-bucket if unresponsive during normal operations

Maintaining routing tables

- **Why prefer old nodes?**
  - Studies show that the longer a peer *stays* online, the higher the probability is that it will *remain* online
  - Makes it difficult to flood the network with bogus peers
- **As SHA-1 is uniform, a Kademlia node will receive messages from nodes with IDs uniformly distributed across the key space**
  - Thus, all traffic is valuable and increase knowledge

Parallelism in Kademlia

- At each step in the lookup process, **FIND_NODE/FIND_VALUE** queries α nodes in parallel
- The node can then choose the quickest peer and move on
- Ensures locality and takes advantages of the strongest peers
- The system does not have to wait until a node times out as with other systems

Redundancy in Kademlia

- Each (key, value) pair is republished every hour and stored at k locations close to the key
- (key, value) expires after 24 hours, so old data is flushed
- But, original publisher republishes (key, value) every 24 hour, so valuable information is maintained
- Whenever a peer A observes a new peer B with ID closer to some of A's keys, A will replicate these keys to B

Joining the network

- Compute an ID
- (Somehow) locate a peer in the network
- Add that peer to the appropriate k-bucket
- Find neighbours by doing FIND_NODE on own ID
- Populate the other k-buckets by performing FIND_NODE
- This process (due to the reflected nature of Kademlia) ensures that the new peer is known across the network

Failure in Kademlia

- Unlikely: Routing tables are continually refreshed due to ordinary traffic
- As SHA-1 is uniform, the k-buckets will be evenly updated
- If there is no traffic, a peer will regularly explicitly refresh oldest k-bucket
- Parallelism in queries ensures that a failing peer is
  - detected
  - routed around

Use of Kademlia

- **Kademlia is fairly widespread for file sharing purposes**
  - eDonkey2000, Overnet, eXeem, Kad
  - a number of BitTorrent clients use Kademlia to locate peers if the original tracker fails
- **Files are stored using a hash of their contents**
- **File names**
  - are divided into keywords
  - the network stores (SHA-1(keyword), (file name, file hash)) for each keyword

Summary

- Built on the experiences from earlier structured networks
- Ensures high performance through parallelism
- All traffic contributes to routing table upkeep
- In widest use of all structured networks

Overview

- Chord
- Pastry
- Kademlia
- **Conclusions**

Structured P2P: A Summary

- **"First generation"**
  - Largely application-specific
  - Few guarantees – worst case O(N)
  - Well suited for "fuzzy" searches
  - No particular overhead
- **"Second generation"**
  - Based on structured network overlays
  - Typically expected O(log N) time and space requirements
    - *...at the cost of overhead for maintaining network*
  - Usually, no "fuzzy" searches – this is exact matches only
    - *. . . but sometimes exact is good enough*
  - . . . *unless* we create an appropriate ID space for keyword matching!

Conclusions

- **Scalability**
  - Much more scalable than unstructured P2P networks measured in number of hops for routing
  - However, churn results in control traffic; slow peers can slowdown entire system (especially in Chord); weak peers may be overwhelmed by control traffic
- **Fairness**
  - The load is evenly distributed across the network, based on the uniformness of the ID space
  - More powerful peers can choose to host several virtual peers

Conclusions

- **Integrity and security**
  - Most systems have various provisions for maintaining proper routing and defending against malicious peers
  - A backhoe is unlikely to take out a major part of the system – at least if we store at k closest nodes
- **Anonymity, deniability, censorship resistance**
  - If we have the key, it is trivial to locate the matching hosts

Milestones!

- **To be presented in Week 37**
  - Kademlia: Implement **FIND_NODE** and **PING**
- **To be presented in Week 38**
  - IoT: Hook up sensors, create web interface to read sensors and set actuators (LEDs)
- **To be presented in Week 39**
  - Kademlia: Implement **STORE** and **FIND_VALUE**
- **To be presented in Week 40**
  - IoT/Kademlia: Store IoT generated data in Kademlia. Ensure resilient data collection and storage. Provide interface to inspect collected data

Milestone 1

- You must implement basic Kademlia. Peers should be able to join and leave in an orderly manner. Implement **PING** and **FIND_NODE**, so k-buckets can be populated
- Requirements: All communication between peers should be RESTful. The individual peer should to a Web browser present a simple page, where the peer's state (such as id and buckets (the latter ideally presented as links to the respective peers)) can be inspected, and where actions, such as searching for an id, can be performed
- You must document your REST API
- You may assume that one Kademlia peer is initially known and available for bootstrapping purposes
- Bonus: Make your system more robust against churn by periodic PINGs