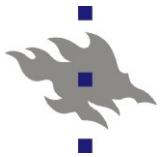
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Overlay and P2P Networks

Structured Networks and DHTs

Prof. Sasu Tarkoma

4.2.2013

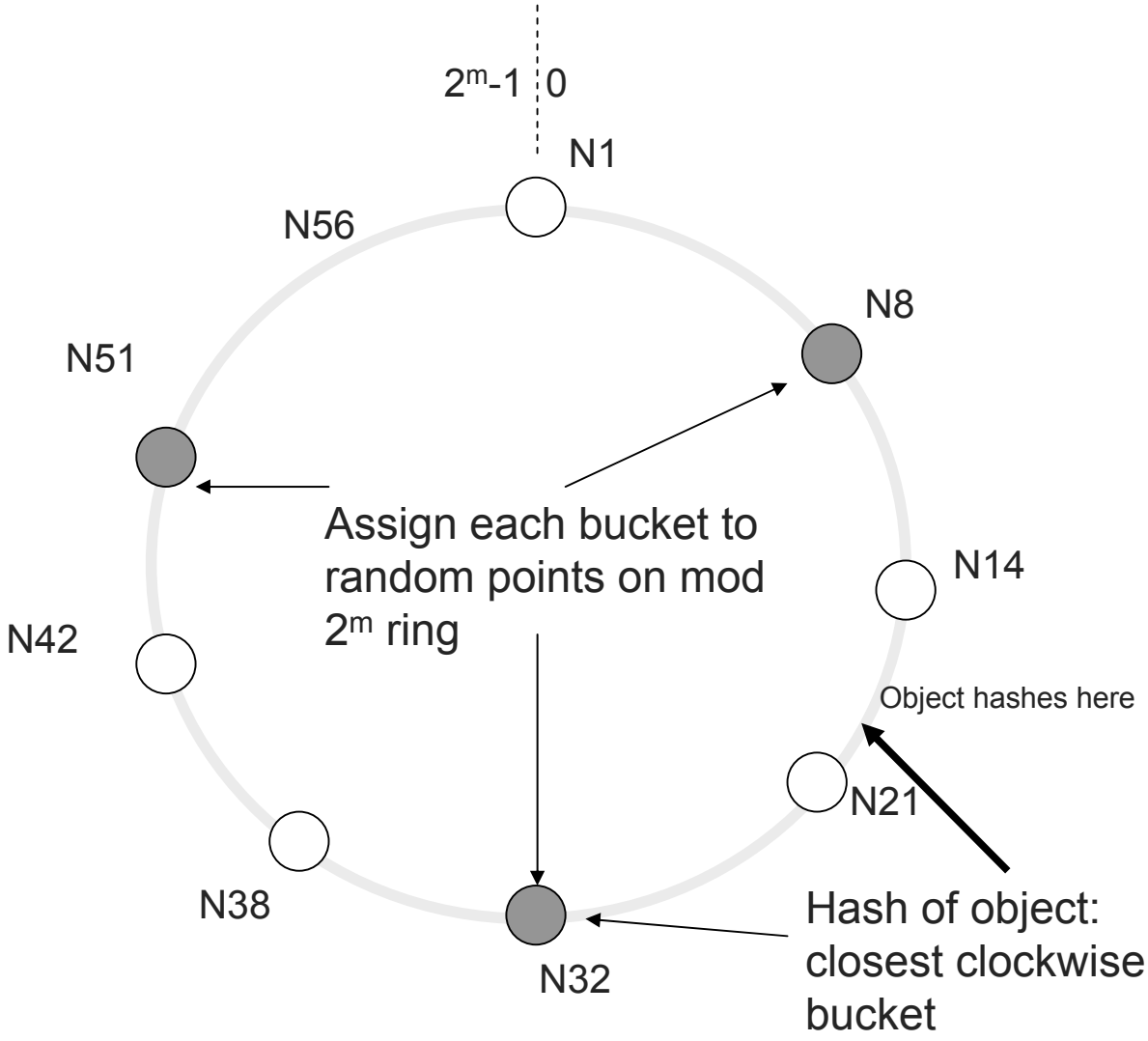


Contents

- Today
  - Consistent Hashing Revisited
  - Semantic free indexing
  - Distributed Hash Tables (DHTs)
- Thursday
  - DHTs continued
  - Discussion on geometries

- Next week
  - Shift to Applications
  - Dr. Samu Varjonen will talk about lookup services based on DHTs

Consistent Hashing Revisited



Global information: Node list

Results in single-hop discovery

View mechanism aims to alleviate this issue (subset of servers visible) → virtual buckets needed

Main point in consistent hashing

The technique requires only *K/n* keys to be remapped on average, where *K* is the number of keys, and *n* is the number of nodes

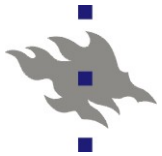
Used in most DHT algorithms

Developed by Karger et al. at MIT

Somewhat involved for example in Chord

Used by CDNs and caches

- Akamai

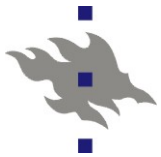


Semantic free indexing I

With semantic free indexing in structured overlays, data objects are given unique identifiers called **keys** that are chosen from the same **identifier space**

Keys are mapped by the overlay network protocol to a node in the overlay network

The overlay network needs to then support scalable **storage** and **retrieval** *(key, value)* pairs



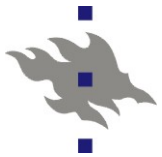
Semantic free indexing II

In order to realize the **insertion**, **lookup**, and **removal** of (key, value) pairs, each peer maintains a **routing table** that consists of its neighbouring peers (their node identifiers and IP addresses)

Lookup queries are then routed across the overlay network using the information contained in the routing tables

Typically each routing step takes the query or message closer to the destination

Requirement to reach the destination in bounded hops!



Comparison to IP routing

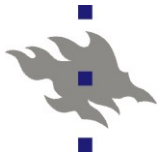
IP routing is based on the longest matching prefix
- Keep a prefix data structure (ternary tree, TCAM)
- Find next hop based on the list (or the destination)

IP addresses are obtained through a local configuration
- process and/or BGP tables, default routes as well

For the DHT case we do not have the IP address semantics
- and mapping to the IP topology

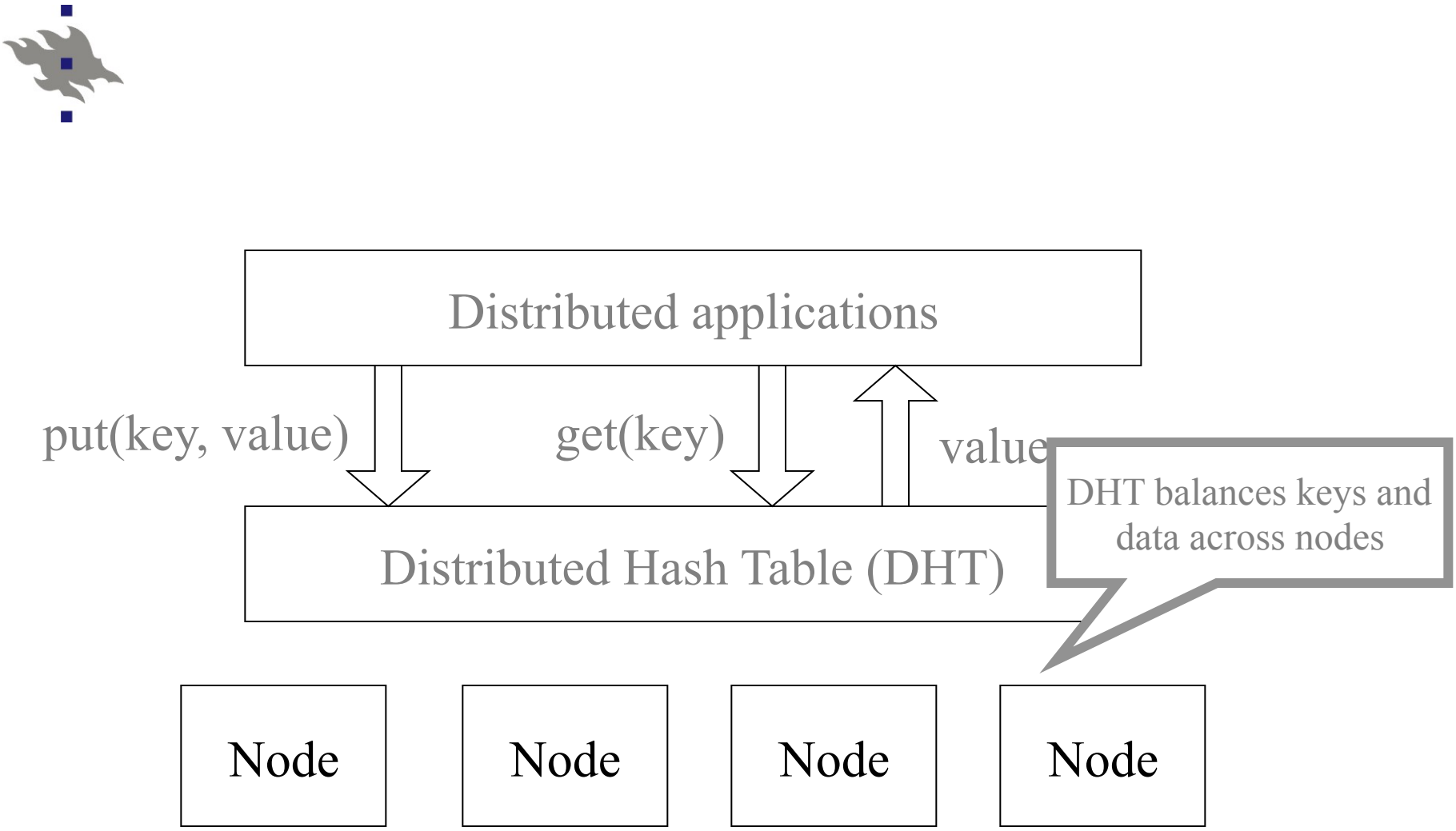
The DHT topology is flat! (typically)

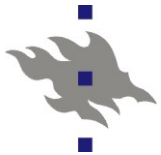
Hence the table structure with suffixes/prefixes.



DHT interfaces

- DHTs offer typically two functions
  - put(key, value)
  - get(key) → value
  - delete(key)
- Supports wide range of applications
  - Similar interface to UDP/IP
    - Send(IP address, data)
    - Receive(IP address) → data
- No restrictions are imposed on the semantics of values and keys
- An arbitrary data blob can be hashed to a key
- Key/value pairs are persistent and global


Distributed applications
put(key, value)
get(key)
value
DHT balances keys and data across nodes
Distributed Hash Table (DHT)
Node
Node
Node
Node

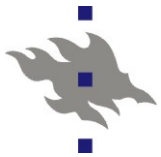


Foundations of Structured Networks

We distinguish between a **routing algorithm** and the **routing geometry**. The algorithm pertains to the exact details of routing table construction and message forwarding

Geometry pertains to the way in which neighbours and routes are chosen. Geometry is the foundation for routing algorithms

The key observation is that the geometry plays a fundamental part in the construction of decentralized overlays



Geometries

The five frequently used overlay topologies are:

- trees
- rings
- tori (k-ary n-cubes)
- butterflies (k-ary n-flies)
- de Bruijn graphs
- XOR geometry

The differences between some of the geometries are **subtle**

For example, it can be seen that the static DHT topology emulated by the DHT algorithms of Pastry and Tapestry are Plaxton trees; however, the dynamic algorithms can be seen as approximation of hypercubes.

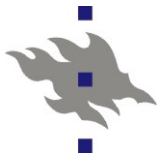
Tree Geometry as an Example

We now investigate the tree geometry

Most geometries follow a similar organizational principle

After trees, we consider rings

Later geometries such as hypercube and the XOR metric



Tree Geometry

The tree's hierarchical organization makes it a suitable choice for efficient routing

In a tree geometry, node identifiers represent the leaf nodes in a **binary tree of depth log n**

The distance between any two nodes is the height of their smallest common subtree

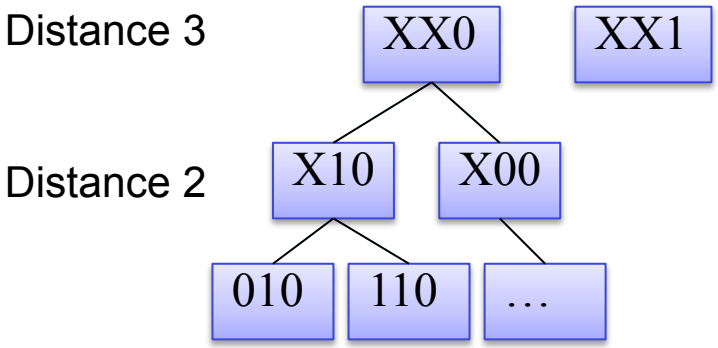
Distance is determined based on suffix / prefix

Example (adresses base 2)

Entries Level	1 Primary neighbour	2	3
0	010	X00	XX0
1	**110**	X10	XX1

Suffix-based

Table size: base * address length

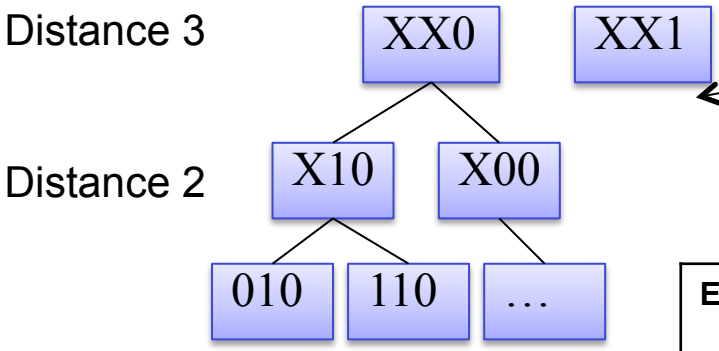


The distance between any two nodes is the height of their smallest common subtree

The table maintains a view to a subset of nodes that allows to route toward destination in log n steps (always taking message closer)

Example (adresses base 2): routing

Entries Level	1 Primary neighbour	2	3
0	010	X00	XX0
1	**110**	X10	XX1



Sending message to 011

Matching suffix XX1 → 101

Entries Level	1 Primary neighbour	2	3
0	001	X01	
1	**101**	X11	XX1

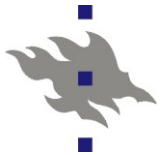
Target is here

Performance of tree based routing

At most $\log_b N$ logical hops are needed for locating nodes, where *N* is the size of the identifier namespace and *b* is the base.

Suffix-routing: Since a node assumes that the preceding digits all match, at each level only a small constant entries are maintained resulting in a total routing table size of $b \log_b N$

Example: for b=2, N=2**3: 2*3=6 entries



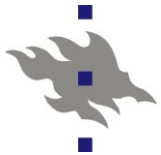
Plaxton Tree Routing

One of the first DHT algorithms, the Plaxton's algorithm, is based on this geometry (object rooted at a node)

Each node maintains a routing table with log *n* neighbours.

The *i*th neighbour is at distance *i* from the current node.

Greedy routing can then be used to forward a message to its destination on the network given the target identifier



Example: Plaxton

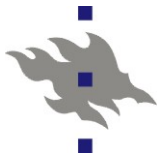
Global ordering of nodes (only one root node possible)

Static configuration

Forest of trees where each server is a **root**

Populate routing table to reflect possible distances

- One suffix digit at a time



Plaxton's algorithm

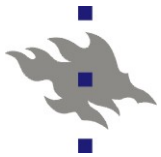
The Plaxton's algorithm realizes an overlay network for **locating** named objects and routing messages to these objects

The algorithm was proposed in 1997 to improve web caching performance by Plaxton, Rajaraman, and Richa

The algorithm guarantees a delivery time within a small factor of the optimal delivery time

The algorithm requires **global knowledge** and does **not** support **additions and removals** of nodes and it is therefore a precursor to the DHT algorithms that tolerate churn, such as Chord, Pastry, and Tapestry

The Plaxton overlay can be seen as a **set of embedded trees** in the network, one rooted in every node, where the destination is the root

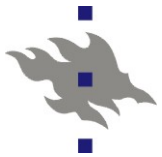


Performance of the Plaxton's algorithm

With consistent routing tables Plaxton's algorithm guarantees that any existing unique node in the system will be found within at most $\log_b N$ logical hops, where N is the size of the identifier namespace and b is the base.

Suffix-routing: Since a node assumes that the preceding digits all match, at each level only a small constant entries are maintained resulting in a total routing table size of $b \log_b N$

It has been proven that the total network distance traveled by messages during both read and write operations proportional to the underlying network distance



Plaxton routing table

The idea in the routing table is to keep track of the suffixes

- More detail about local neighbours
- Less information about far-away nodes
- Sufficient information to do global routing

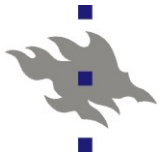
→ Organize into levels and each level into the different possible suffix lengths

Base * address length elements are needed

We already know the longest matching suffix

Use this fact to structure the routing table

Similar table maintained by most DHT algorithms (the details depend on the algorithm)



Plaxton's algorithm: routing table of node 3642

Entries / Levels	1 Primary neighbour	2	3	4
1	0642	X042	XX02	XXX0
2	1642	X142	XX12	XXX1
3	2642	X242	XX22	XXX2
4	**3642**	X342	XX32	XXX3
5	4642	X442	XX42	XXX4
6	5642	X542	XX52	XXX5
7	6642	X642	XX62	XXX6
8	7642	X742	XX72	XXX7

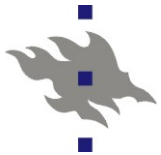
Wildcards are marked with X
Primary neighbour is one digit away

Example lookup

Node **3642** receives message for **2342**
- The common string is **XX42**
- Two shared digits, consult second column and choose the correct digit
- Send to node with one digit closer
- Fourth line with **X342**

Table size: base * address length
In this example octal base (8)
and 4 digit addresses

Each routing table is organized in routing levels and each entry points to a set of nodes closest in network distance to a node which matches the given suffix



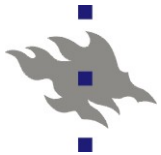
Key limitations of Plaxton

Requirement for global knowledge

Static node set

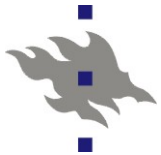
Root nodes are possible points of failure

Lack of ability to adapt to dynamic query patterns



Plaxton

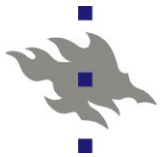
	Plaxton
Foundation	Plaxton-style mesh (hyper-cube)
Routing function	Suffix matching
System parameters	Number of peers N, base of peer identifier B
Routing performance	$O(\log_B N)$
Routing state	$B\log_B N$ *Note: global ordering of nodes*
Joins/leaves	*Not supported*



DHT Algorithms

Plaxton is an early example of a DHT

Next we focus on DHTs that support dynamic operation and do not require global knowledge



Structured Overlays

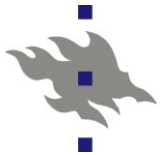
Structured overlays are typically based on the notion of a semantic free index and consistent hashing

They are based on different routing geometries

The decentralized DHTs balance hop count with the size of the routing tables, network diameter, and the ability to cope with changes

Geometries and DHTs

- Tree – Plaxton's algorithm
- Ring – Chord
- Tori – CAN
- Hypercubes – Pastry and Tapestry
- XOR metric – Kademlia
- Butterfly – Viceroy

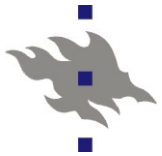


Deployed DHT Applications

Key examples of deployed DHT algorithms include

- Kademlia used in BitTorrent
- Amazon's Dynamo
- The Coral Content Distribution Network
- PlanetLab

We will return to applications later on this course

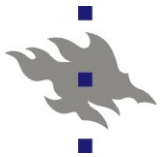


Requirements

An ideal DHT algorithm would meet the following requirements:

- Easy deployment over the Internet.
- Scalability to millions of nodes and billions of data elements
- Availability for the data items so that faults can be tolerated
- Adaptation to changes in the network, including network partitions and churn
- Awareness of the underlying network architecture so that unnecessary communication is avoided
- Secure so that data confidentiality, authenticity, and integrity can be established and that malicious nodes cannot overwhelm the overlay network

It is not easy to meet these requirements simultaneously!



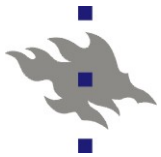
Rings

Rings are a popular geometry for DHTs due to their simplicity. In a ring geometry, nodes are placed on a one-dimensional cyclic identifier space. The distance from an identifier A to B is defined as the clockwise numeric distance from A to B on the circle

Rings are related with **tori** and **hypercubes**, and the 1-dimensional torus is a ring. Moreover, a k-ary 1-cube is a k-node ring

The Chord DHT is a classic example of an overlay based on this geometry.

Each node has a predecessor and a successor on the ring, and an additional routing table for pointers to increasingly far away nodes on the ring



Chord

- Chord is an overlay algorithm from MIT
  - Stoica et. al., SIGCOMM 2001
- Chord is a lookup structure (a directory)
  - Resembles binary search
- Uses consistent hashing to map keys to nodes
  - Keys are hashed to m-bit identifiers
  - Nodes have m-bit identifiers
    - IP-address is hashed
  - SHA-1 is used as the baseline algorithm
- Support for rapid joins and leaves
  - Churn
  - Maintains routing tables

Chord routing I

Identifiers are ordered on an identifier circle modulo 2^m

The Chord ring with m-bit identifiers

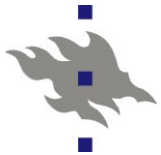
A node has a well determined place within the ring

A node has a predecessor and a successor

A node stores the keys between its **predecessor** and itself

The (key, value) is stored on the **successor** node of key

A routing table (finger table) keeps track of other nodes



Finger Table

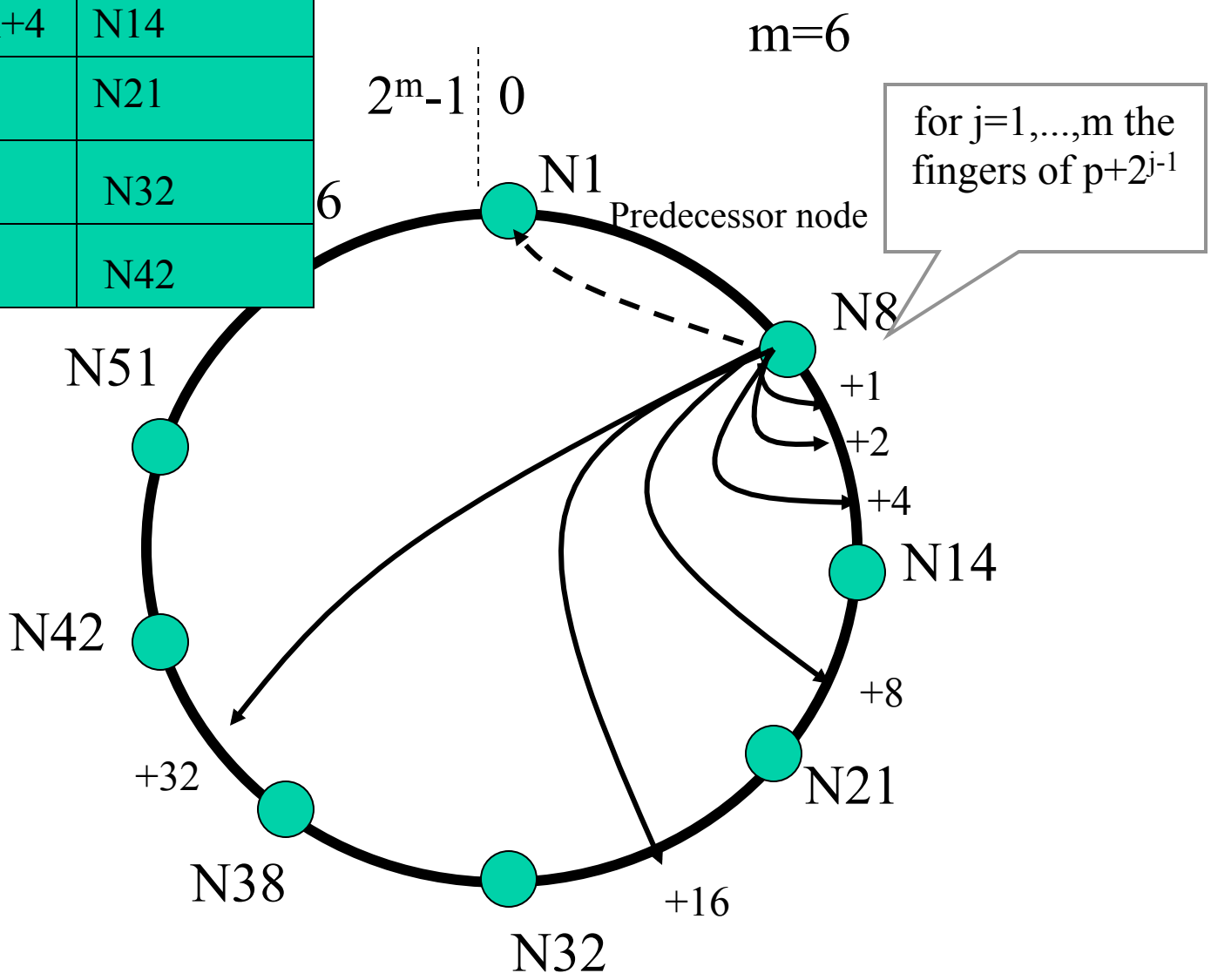
Each node maintains a routing table with at most m entries

The i:th entry of the table at node n contains the identity of the first node, s, that succeeds n by at least $\mathbf{2^{i-1}}$ on the identifier circle

$s = successor(n + 2^{i-1})$

The i:th finger of node n

Finger	Maps to	Real node
1,2,3	x+1,x+2,x+4	N14
4	x+8	N21
5	x+16	N32
6	x+32	N42



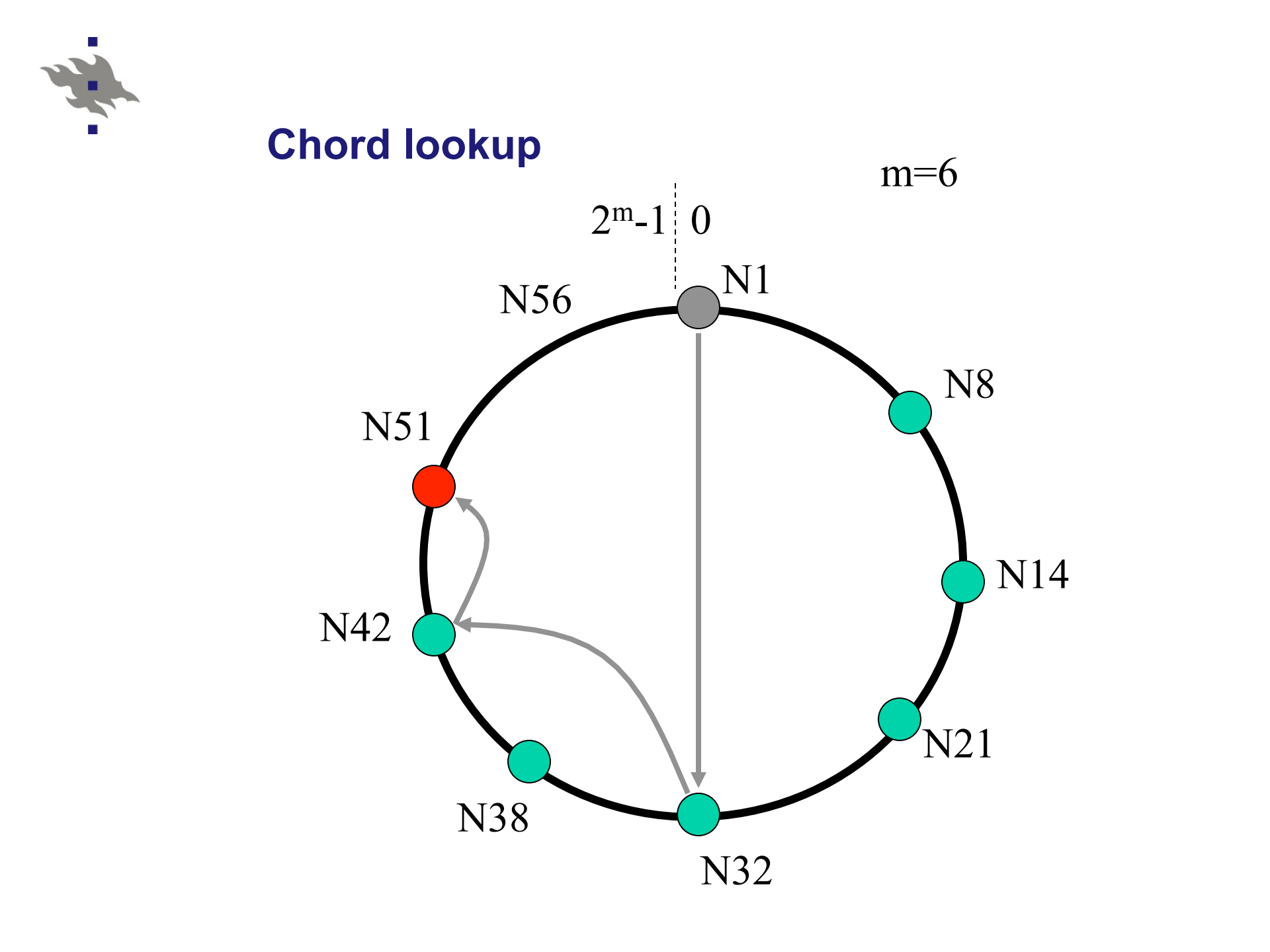
Chord routing II

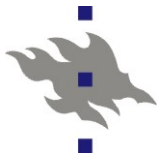
Routing steps

- check whether the key k is found between n and the successor of n
- if not, forward the request to the closest finger preceding k

Each knows a lot about nearby nodes and less about nodes farther away

The target node will be eventually found


Chord lookup
m=6
2m-1
0
N1
N8
N14
N21
N32
N38
N42
N51
N56



Invariants

Two invariants:

- Each node's **successor** is correctly maintained.
- For every key k, node successor(k) is responsible for k.

A node stores the keys between its predecessor and itself

- The (key, value) is stored on the successor node of key

Join

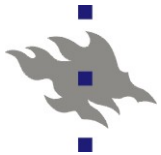
A new node *n* joins

Needs to know an existing node *n'*

Three steps

1. Initialize the predecessor and fingers of node
2. Update the fingers and predecessors of existing nodes to reflect the addition of *n*
3. Notify the higher layer software and transfer keys

Leave uses steps 2. (update removal) and 3. (relocate keys)

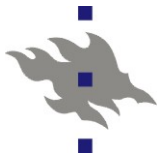


1. Initialize routing information

- Initialize the predecessor and fingers of the new node *n*
- *n* asks *n'* to look predecessor and fingers
  - One predecessor and *m* fingers
- Look up predecessor
  - Requires log (N) time, one lookup
- Look up each finger (at most *m* fingers)
  - log (N), we have Log N * Log N
  - $O(Log^2 N)$ time

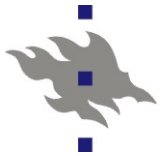
Steps 2. And 3.

2. Updating fingers of existing nodes
   - Existing nodes must be updated to reflect the new node
   - Performed counter clock-wise on the circle
     - Algorithm takes i:th finger of *n* and walks in the counter-clock-wise direction until it encounters a node whose i:th finger precedes *n*
     - Node *n* will become the i:th finger of this node
   - $O(Log^2 N)$ time
3. Transfer keys
   - Keys are transferred only from the node immediately following *n*



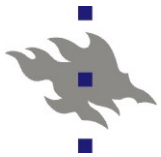
Chord Properties

- Each node is responsible for K/N keys (K is the number of keys, N is the number of nodes). This is the consistent hashing result.
- When a node joins or leaves the network only O(K/N) keys will be relocated (the relocation is local to the node)
- Lookups take O(log N) messages
- To re-establish routing invariants after join/leave $O(\log^2 N)$ messages are needed



Chord

	Chord
Foundation	Circular space (hyper-cube)
Routing function	Matching key and nodeID
System parameters	Number of peers N
Routing performance	*O(log N)*
Routing state	*log N*
Joins/leaves	$(log\ N)^2$



Tapestry

- DHT developed at UCB
  - Zhao et. al., UC Berkeley TR 2001
- Used in OceanStore
  - Secure, wide-area storage service
- Tree-like geometry
- Suffix-based hypercube
  - 160 bits identifiers
- Suffix routing from A to B
  - hop(h) shares suffix with B of length digits
- Tapestry Core API:
  - publishObject(ObjectID,[serverID])
  - routeMsgToObject(ObjectID)
  - routeMsgToNode(NodeID)

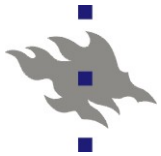
Tapestry Routing

In a similar fashion to Plaxton and Pastry, each routing table is organized in routing levels and each entry points to a set of nodes closest in network distance to a node which matches the suffix

In addition, a node keeps also **back-pointers** to each node referring to it (shortcut links, also useful for reverse path)

While Plaxton's algorithm keeps a mapping (pointer) to the closest copy of an object, Tapestry keeps pointers to **all copies**

This allows the definition of application specific selectors what object should be chosen (or what path)

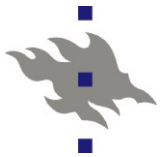


Tapestry Routing II

Each identifier (object) is mapped to an active node called the root

A server *S* publishes that it has an object *O* by routing a message to the root of *O* using the overlay system in similar fashion to the Plaxton's algorithm using **incremental suffix routing**

The original Plaxton scheme used the greatest number of trailing bit positions to map an object to a node



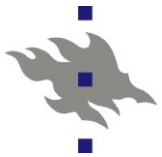
Surrogate Routing

In a distributed decentralized system there may be potentially many candidate nodes for an object's root

Plaxton solved this using global ordering of nodes. Tapestry solves this by using a technique called *surrogate routing*

Surrogate routing tentatively assumes that an object's identifier is also the nodes identifier and routes a message using a deterministic selection towards that destination

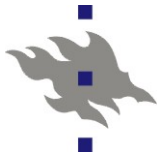
The destination then becomes a surrogate root for the object (in other words, a deterministic function is used to choose among possible routes the best route towards the root)



Tapestry Node Joins and Leaves

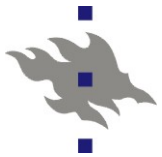
Operations use acknowledged multicast that builds a tree towards a given suffix

1. Find surrogate by hashing the node id
2. Route toward the node id and at each hop copy the neighbour map of the node (shares a suffix with each hop)
3. Each entry should be a closest neighbour (iterate also neighbour's neighbours until these are found)
    1. Iterative nearest neighbour for routing table levels.
4. New node might become the root for existing objects (object refs need to be moved to the new node)
5. Create routing tables & notify other nodes



```
H = G;
For (i=0; H != NULL; i++) {
  Grab ith level NeighborMap_i from H;
  For (j=0; j<baseofID; j++) {
    //Fill in jth level of neighbor map
    While (Dist(N, NM_i(j, neigh)) >
          min(eachDist(N, NM_i(j, sec.neigh)))) {
      neigh=sec.neighbor;
      sec.neighbors=neigh−>sec.neighbors(i,j);
    }
   }
  H = LookupNextHopinNM(i+1, new_id);
} //terminate when null entry found
Route to current surrogate via new_id;
Move relevant pointers off current surrogate;
Use surrogate(new_id) backptrs to notify nodes
by flooding back levels to where
surrogate routing first became necessary.
```

Pseudocode for dynamic node insertion
http://oceanstore.cs.berkeley.edu/publications/papers/pdf/tapestry_sigcomm_tr.pdf



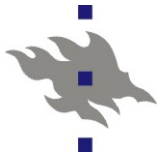
Planned Delete in Tapestry

Leaving node updates its neighbors ($O(\log^2 n)$)

- To out-neighbors: inform that pointers are gone
- To in-neighbors: Exiting node says it is leaving and proposes at least one replacement.
  - Use backpointers to find in-neighbors.
- Exiting node republishes all objects pointers it stores
- Use republish-on-delete

Objects rooted at leaving node obtain new roots

- Either proactive pointer copying, or
- wait for republishes



Tapestry Routing Table

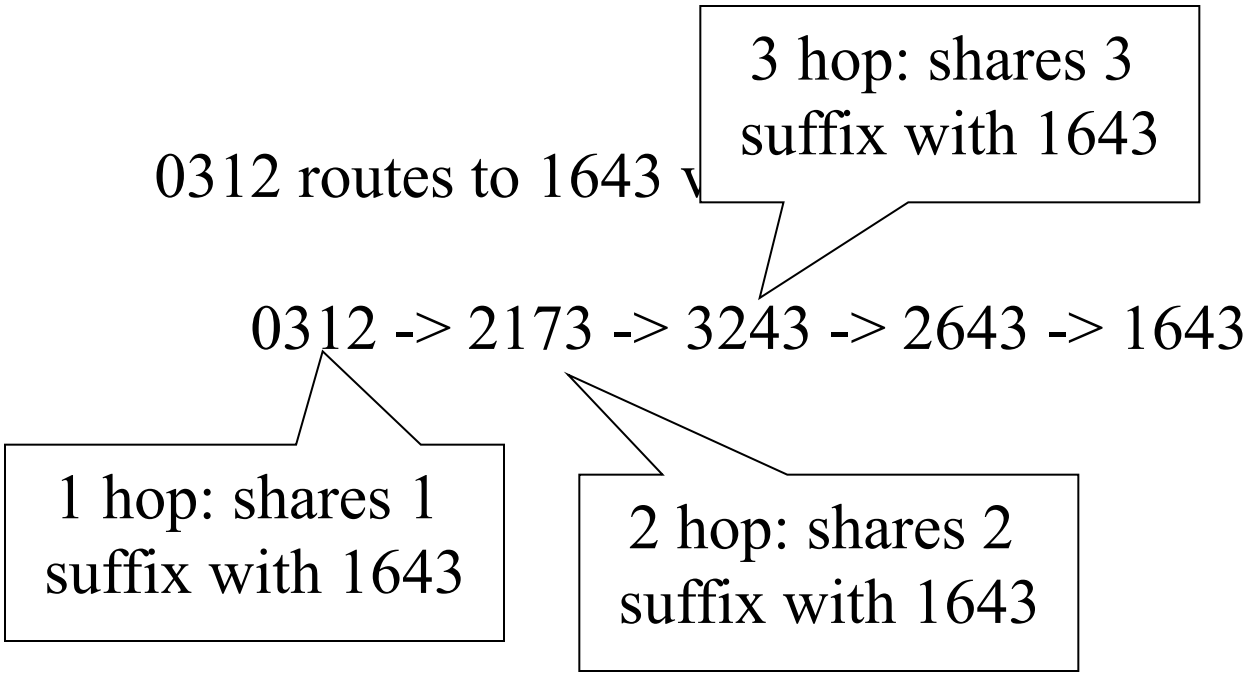
Entries / Levels	1 Primary neighbour	2	3	4
1	0642	X042	XX02	XXX0
2	1642	X142	XX12	XXX1
3	2642	X242	XX22	XXX2
4	3642	X342	XX32	XXX3
5	4642	X442	XX42	XXX4
6	5642	X542	XX52	XXX5
7	6642	X642	XX62	XXX6
8	7642	X742	XX72	XXX7

Object Location Pointers
Hotspot Monitor
Object Store

Back Pointers

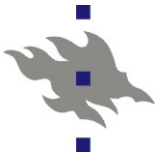
Closest node matching the suffix
Each entry can have multiple pointers for the same object.
Objects can have multiple roots using salt value in hashing.

Suffix routing

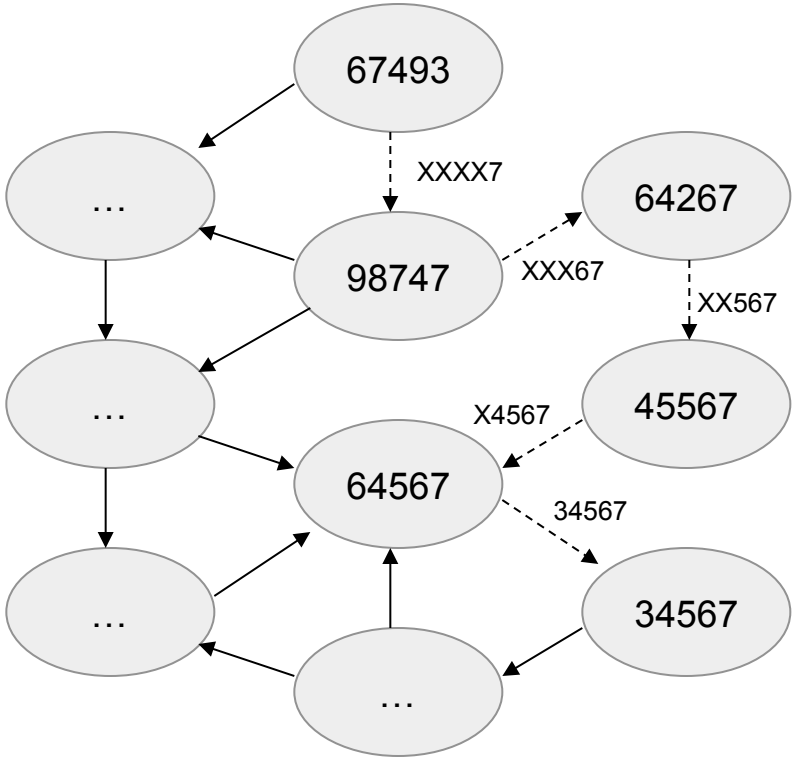


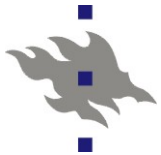
Rounting table with b*logb(N) entries

Entry(i,j) – pointer to the neighbour j+(i-1) suffix



Incremental suffix routing from **67493** to **34567**





Pastry I

- A DHT based on a circular flat identifier space
- Prefix-routing
  - Message is sent towards a node which is numerically closest to the target node
  - Procedure is repeated until the node is found
  - Prefix match: number of identical digits before the first differing digit
  - Prefix match increases by every hop
- Similar performance to Chord

Pastry Routing

Pastry builds on consistent hashing and the Plaxton's algorithm. It provides an object location and routing scheme and routes messages to nodes

It is a prefix based routing system, in contrast to suffix based routing systems such as Plaxton and Tapestry, that supports proximity and network locality awareness

At each routing hop, a message is forwarded to a numerically closer node. As with many other similar algorithms, Pastry uses an expected average of log(N) hops until a message reaches its destination

Similarly to the Plaxton's algorithm, Pastry routes a message to the node with the nodeId that is numerically closest to the given key

Pastry Routing

Each Pastry node maintains a set of neighboring nodes in the nodeId space (called the **leaf set)**, both to ensure reliable message delivery, and to store replicas of objects for fault tolerance

The Pastry overlay construction observes **proximity** in the underlying Internet. Each routing table entry is chosen to refer to a node with low network delay, among all nodes with an appropriate nodeId prefix

As a result, one can show that Pastry routes have a low delay penalty: the average delay of Pastry messages is less than twice the IP delay between source and destination

Pastry Scalar Distance Metric

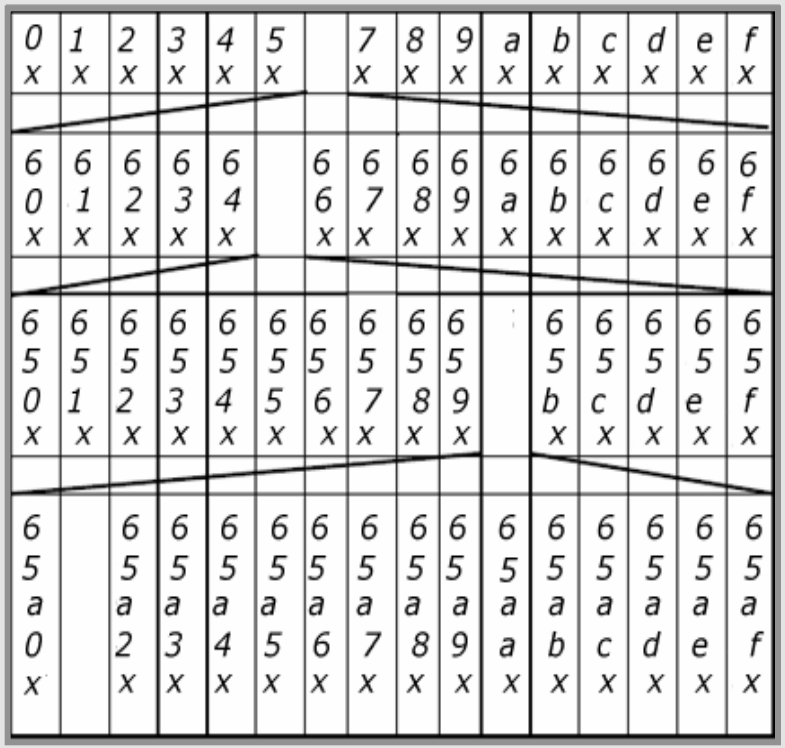
The Pastry proximity metric is a scalar value that reflects the distance between any pair of nodes, such as the round trip time

It is assumed that a function exists that allows each Pastry node to determine the distance between itself and a node with a given IP address

Pastry Message Routing

- If leaf set has the prefix → send to local
- else send to the identifier in the routing table with the longest common prefix (longer than the current node)
- else query leaf set for a numerically closer node with the same prefix match as the current node

Pastry routing table



0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

Routing table of a Pastry node with nodeId **65a1x**, b = 4. Digits are in base 16, x represents an arbitrary suffix.

The IP address associated with each entry is not shown.

Each level adds detail

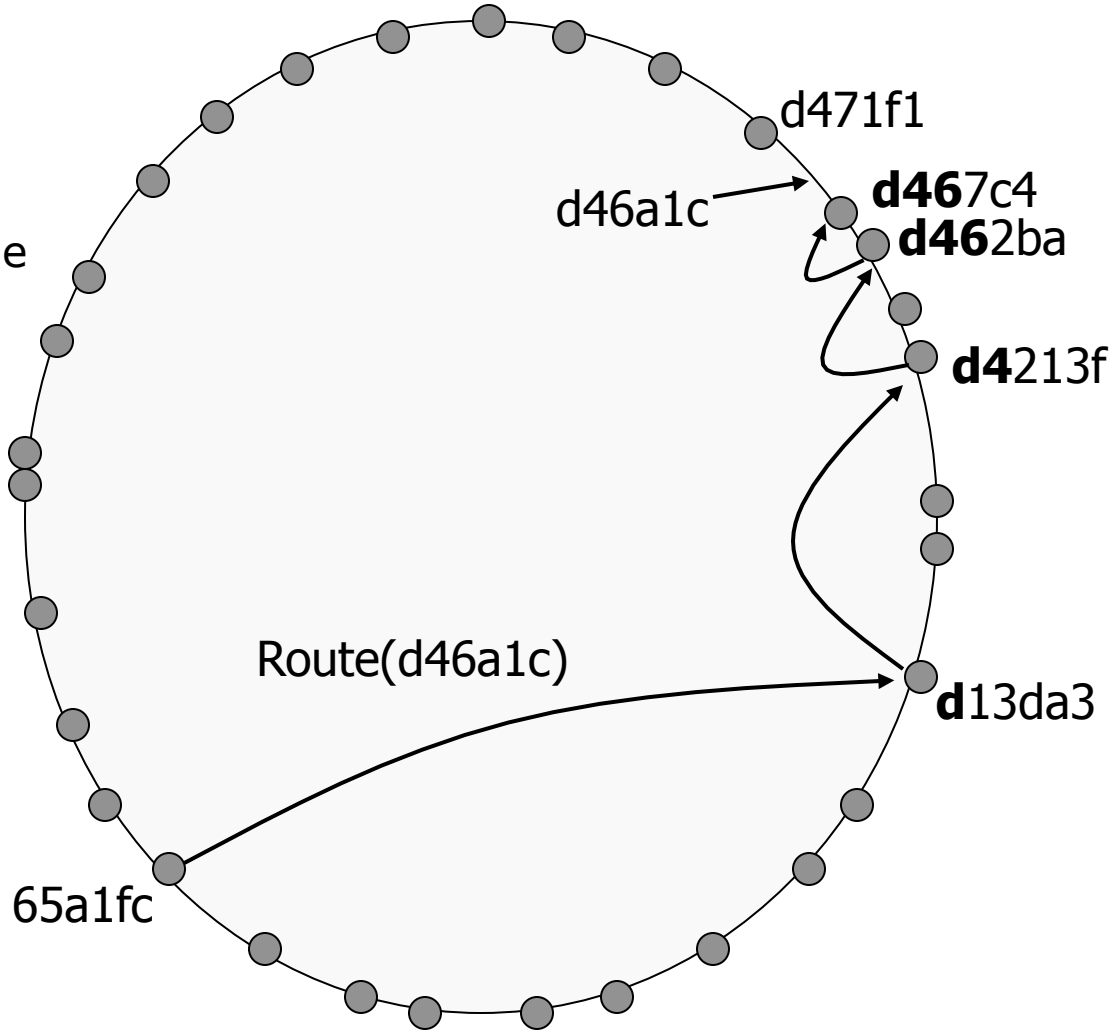
Each node knows something about the global reachability and then more about local nodes

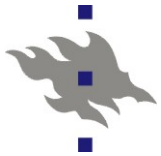
Pastry Routing Example

Prefix-based

Route to node with shared prefix (with the key) of ID at least one digit more than this node.

Neighbor set, leaf set and routing table.





Pastry and Tapestry

	Pastry	Tapestry
Foundation	Plaxton-style mesh (hyper-cube)	Plaxton-style mesh (hyper-cube)
Routing function	Matching key and prefix in nodeID	Suffix matching
System parameters	Number of peers N, base of peer identifier B	Number of peers N, base of peer identifier B
Routing performance	$O(\log_B N)$ *Note proximity metric*	$O(\log_B N)$ *Note surrogate routing*
Routing state	$2B\log_B N$	$\log_B N$
Joins/leaves	$\log_B N$	$\log_B N$