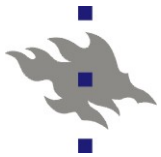
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Overlay and P2P Networks

Structured Networks and DHTs

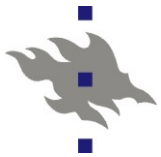
Prof. Sasu Tarkoma

6.10.2010



Contents

- DHTs
  - Plaxton
  - CAN
  - Chord
  - Tapestry
  - Pastry
  - Kademlia
  - Viceroy
- Discussion



Structured Overlays

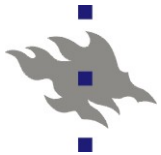
Structured overlays are typically based on the notion of a semantic free index and consistent hashing

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

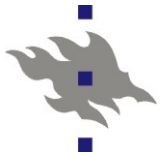


Deployed DHT Applications

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



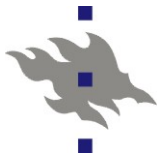
Requirements

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!

DHT Algorithms



Plaxton's algorithm

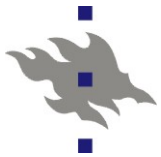
The Plaxton's algorithm realizes an overlay network for **locating** named objects and routing messages to these objects

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

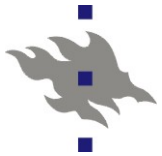


Performance of the Plaxton's algorithm

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

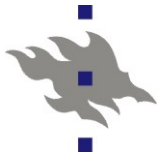
→ Organize into levels and each level into the different possible suffix lengths

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

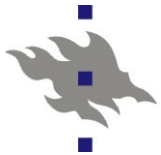
Wildcards are marked with X
Primary neighbour is one digit away

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



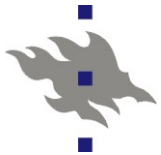
Key limitations of Plaxton

Requirement for global knowledge

Static node set

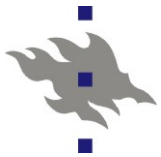
Root nodes are possible points of failure

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*



Content Addressable Network (CAN)

The *Content Addressable Network (CAN)* is a DHT algorithm based on virtual multi-dimensional **Cartesian** coordinate space

In a similar fashion to other DHT algorithms, can is designed to be scalable, self-organizing, and fault tolerant

The algorithm is based on a ***d*-dimensional torus** that realizes a virtual logical addressing space independent of the physical network location

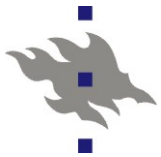
The coordinate space is dynamically partitioned into **zones** in such a way that each node is responsible for at least one distinct zone

CAN performance

For a d dimensional coordinate space partitioned into n zones, the average routing path length is **$O(d * N^{1/d})$** hops and each node needs to maintain 2d neighbours

This means that for a d-dimensional space the number of nodes can grow without increasing per node state

Another beneficial feature of CAN is that there are many paths between two points in the space and thus the system may be able to route around faults



Logarithmic CAN

A logarithmic CAN is a system with d = log n

In this case, CAN exhibits similar properties as Chord and Tapestry, for example O(log n) diameter and degree at each node

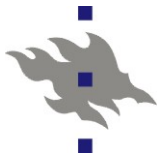
Joining a CAN network

In order for a new node to join the CAN network, the new node must first find a node that is already part of the network, **identify a zone that can be split**, and then **update** routing tables of neighbours to reflect the split introduced by the new node

In the seminal CAN article the bootstrapping mechanism is not defined

One possible scheme is to use a DNS lookup to find the IP address of a bootstrap node (essentially a rendezvous point)

Bootstrapping nodes may be used to inform the new node of IP addresses of nodes currently in the CAN network



Leaving a CAN network

Node departures are handled in a similar fashion than joins.

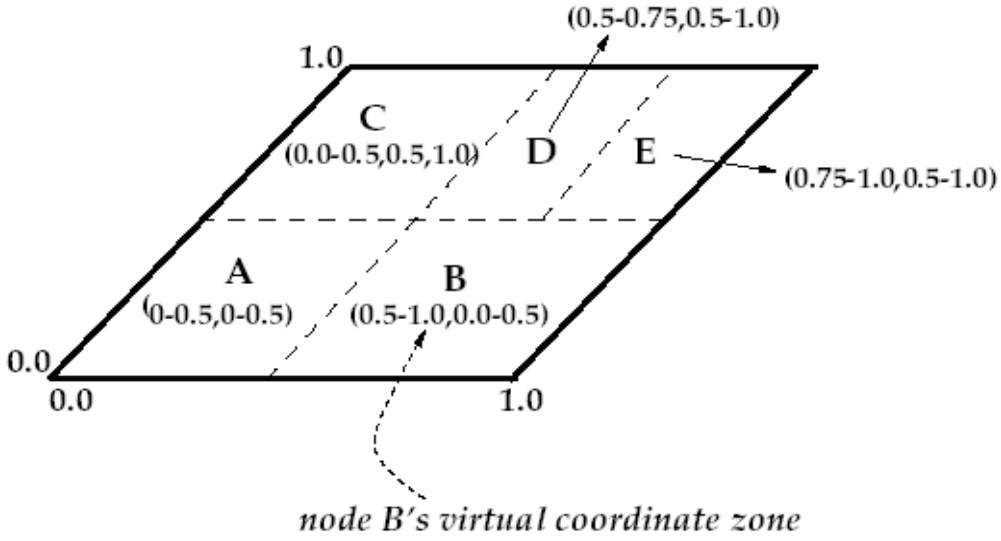
A node that is departing must **give up its zone** and the CAN algorithm needs to **merge** this zone with an existing zone routing tables need to be then updated to reflect this change in zones

A node's departure can be detected using heartbeat messages that are periodically broadcast between neighbours

If a merging candidate cannot be found, the neighbouring node with the smallest zone will take over the departing node's zone

After the process the neighbouring nodes' routing tables are updated to reflect the change in the zone responsibility

CAN



Virtual *d*-dimensional

Cartesian coordinate

system on a *d*-torus

Example: 2-*d* [0,1]x[1,0]

Dynamically partitioned

among all nodes

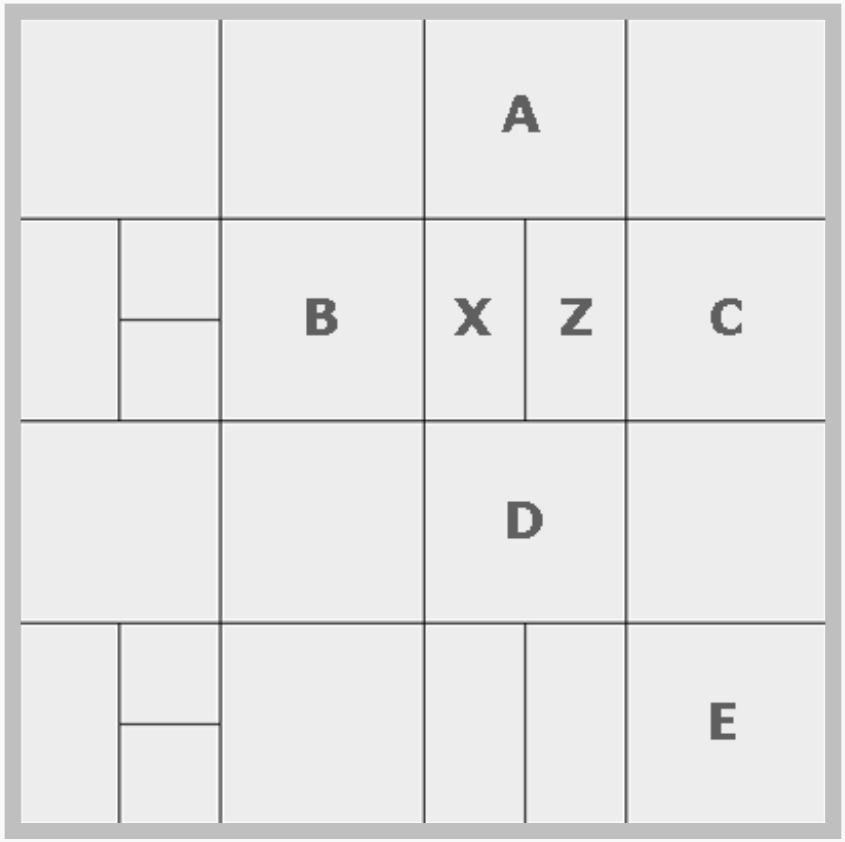
Pair (*K*,*V*) is stored by

mapping key *K* to a point *P* in the space using a uniform hash function and storing (*K*,*V*) at the node in the zone containing *P*

Retrieve entry (*K*,*V*) by applying the same hash function to map *K* to *P* and retrieve entry from node in zone containing *P*

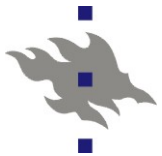
If P is not contained in the zone of the requesting node or its neighboring zones,

route request to neighbor node in zone nearest *P*



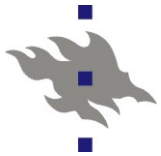
Z joins the system

Peer Xs coordinate neighbor set = {A B D Z}
New Peer Zs coordinate neighbor set = {A C D X}



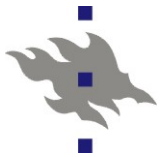
Pythagorean based CAN algorithm

Data: c is current node, P is the target point.
Function: *RouteCAN*(c,P) returns the owner p of point P
if $P \notin c$ **then**
  /* P is in c's neighbors n */
  $p \leftarrow n$
else
  /* P is not in origo node c's zone */
  $p \leftarrow c$ /* current node is set to p */
  while $P \neq p$ **do**
    /* Until an owner is found for P */
    $d \leftarrow \sqrt{(P_x - n_x)^2 + (P_y - n_y)^2}$
    Neighbour n with shortest distance d is the next hop node
    $p \leftarrow n$
  end
end
Point P is in the current node p's zone
return p



Content Addressable Network (CAN)

	CAN
Foundation	Multi-dimensional space (d-dimensional torus)
Routing function	Maps (key,value) pairs to coordinate space
System parameters	Number of peers N, number of dimensions d
Routing performance	$O(dN^{1/d})$
Routing state	*2d*
Joins/leaves	*2d*



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

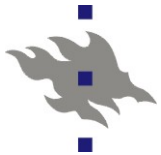
A node has a well determined place within the ring

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

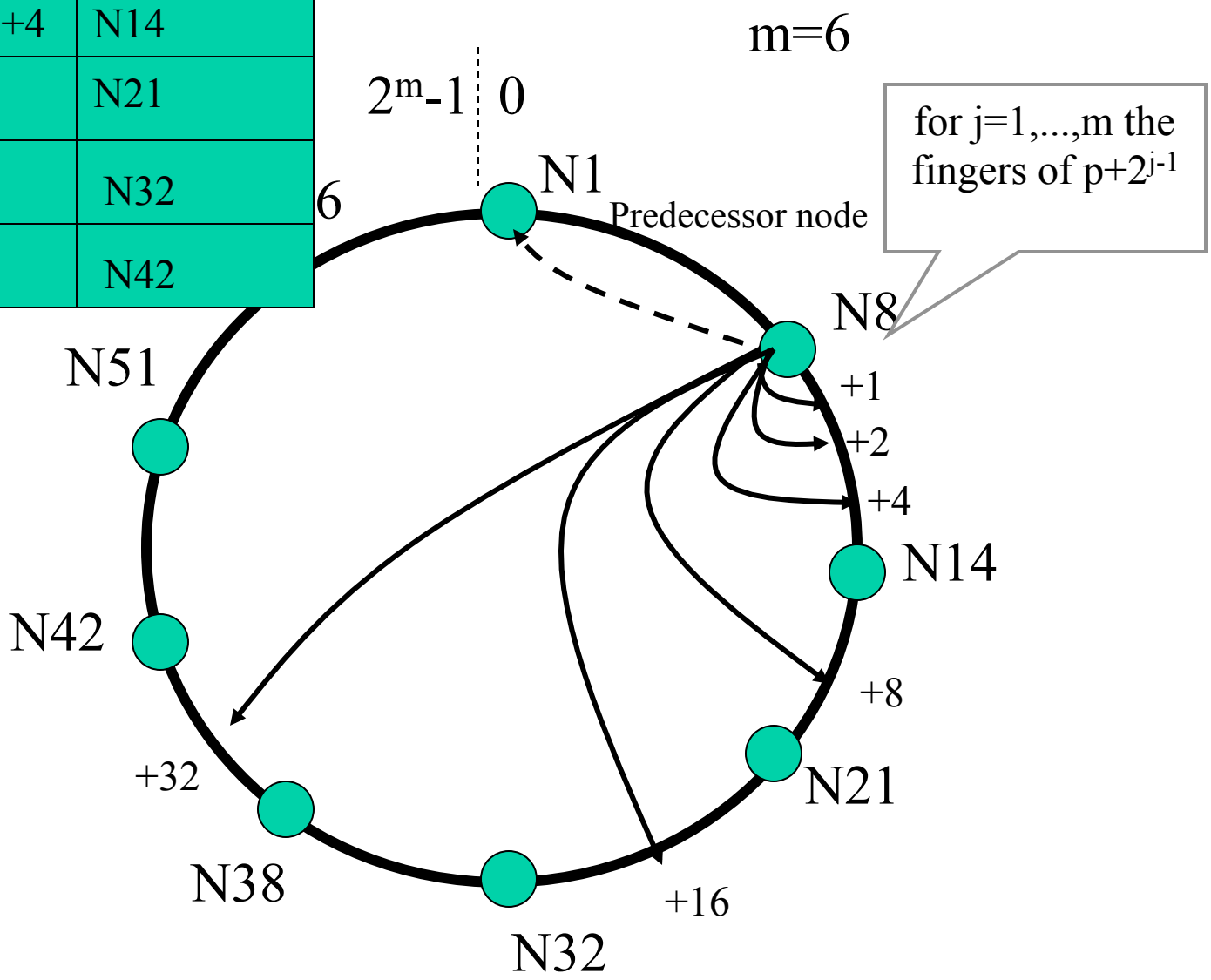
Each node maintains a routing table with at most m entries

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



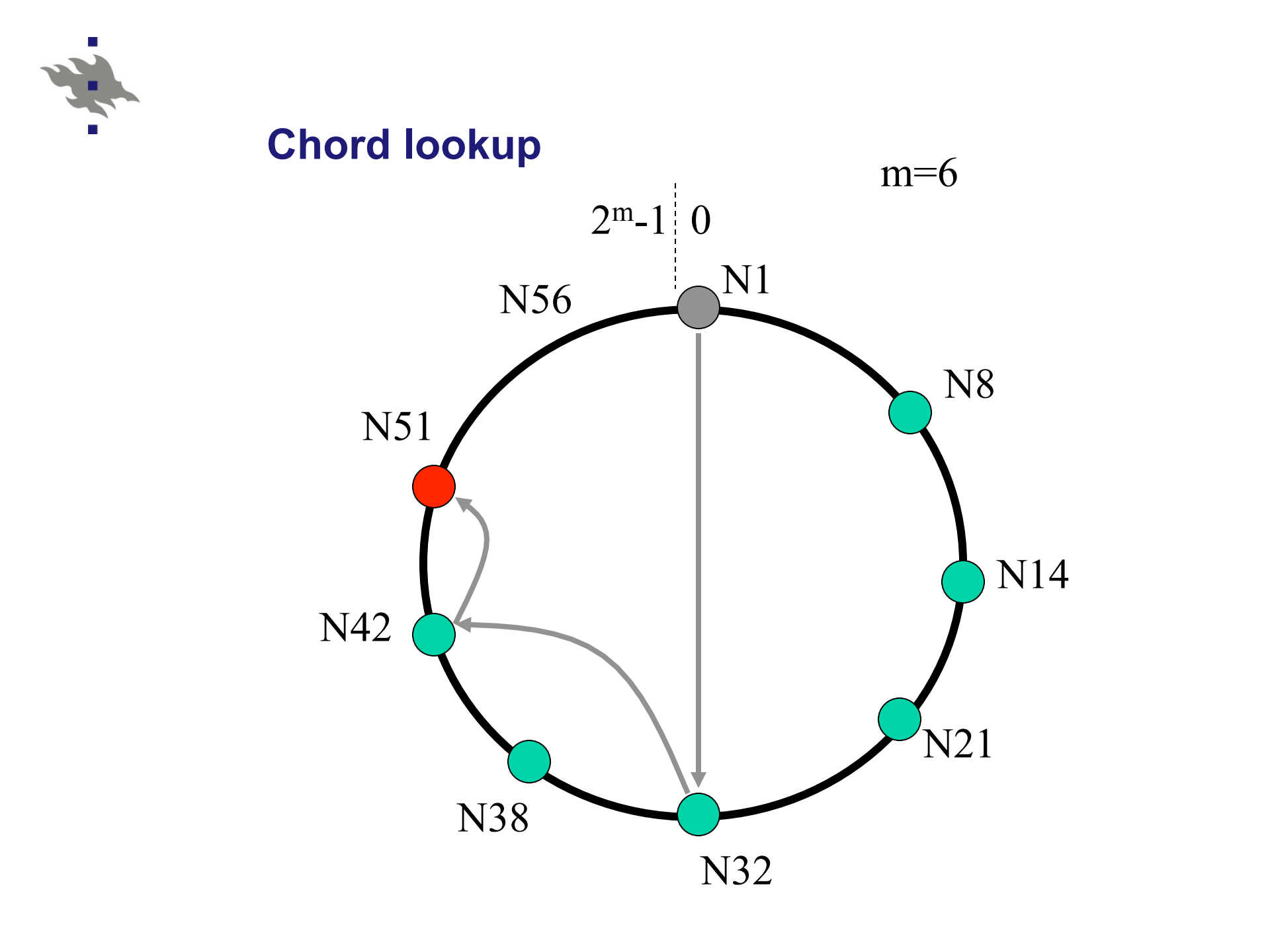
Chord routing II

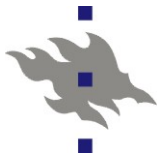
Routing steps

- 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

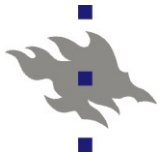
A new node *n* joins

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)

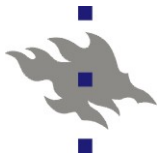


1. Initialize routing information

- 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

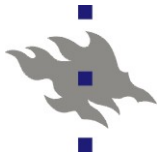
Steps 2. And 3.

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*



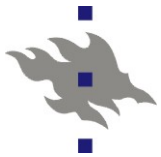
Chord Properties

- 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)

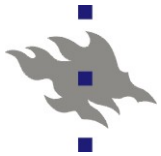
Tapestry Routing

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)

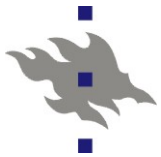


Tapestry Routing II

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



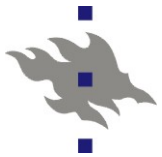
Surrogate Routing

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

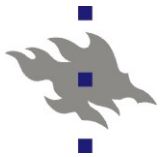
1. Find surrogate by hashing the node id
2. Route toward the node id and at each hop copy the neighbour map of the node (shares a suffix with each hop)
3. Each entry should be a closest neighbour (iterate also neighbour's neighbours until these are found)
    1. Iterative nearest neighbour for routing 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 nextHop algorithm

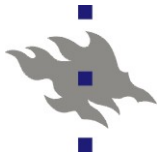
Data: n is the previous hop number, G is the destination GUID, β is the base of the GUIDs (width of the routing table), $R_{i,j}$ is the routing table, in which the i:th entry in the j:th level is the ID and location of the closest node that begins with prefix(N, $j-1$)+i. *MaxHop(R)* is the height of R.
Function: *NextHop* returns the next hop or self if local node is the root

```
if n = MaxHoP(R) then
    /* Destination reached                                   */
    return self
else
    d = G_n  /* d is the n:th digit of G                      */

    e = R_{n,d}  /* e is the d:th entry of the n:th row of R  */

    while e = nil do
        /* Incremental suffix routing                        */

        d = (d + 1) mod β
        e = R_{n,d}
    end
    if e = self then
        return NextHop(n + 1,G)
    else
        return e
    end
end
```



Tapestry Routing Table

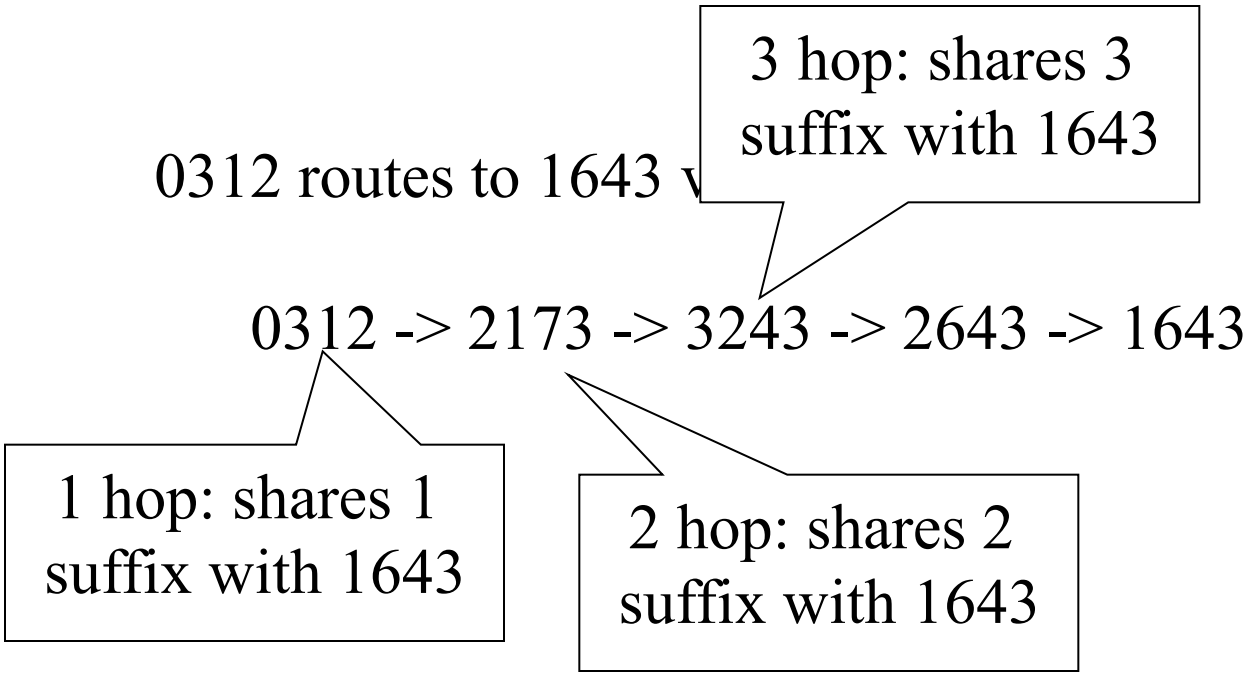
Entries / Levels	1 Primary neighbour	2	3	4
1	0642	X042	XX02	XXX0
2	1642	X142	XX12	XXX1
3	2642	X242	XX22	XXX2
4	3642	X342	XX32	XXX3
5	4642	X442	XX42	XXX4
6	5642	X542	XX52	XXX5
7	6642	X642	XX62	XXX6
8	7642	X742	XX72	XXX7

Object Location Pointers
Hotspot Monitor
Object Store

Back Pointers

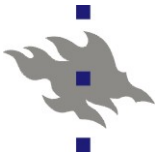
Closest node matching the suffix
Each entry can have multiple pointers for the same object.
Objects can have multiple roots using salt value in hashing.

Suffix routing

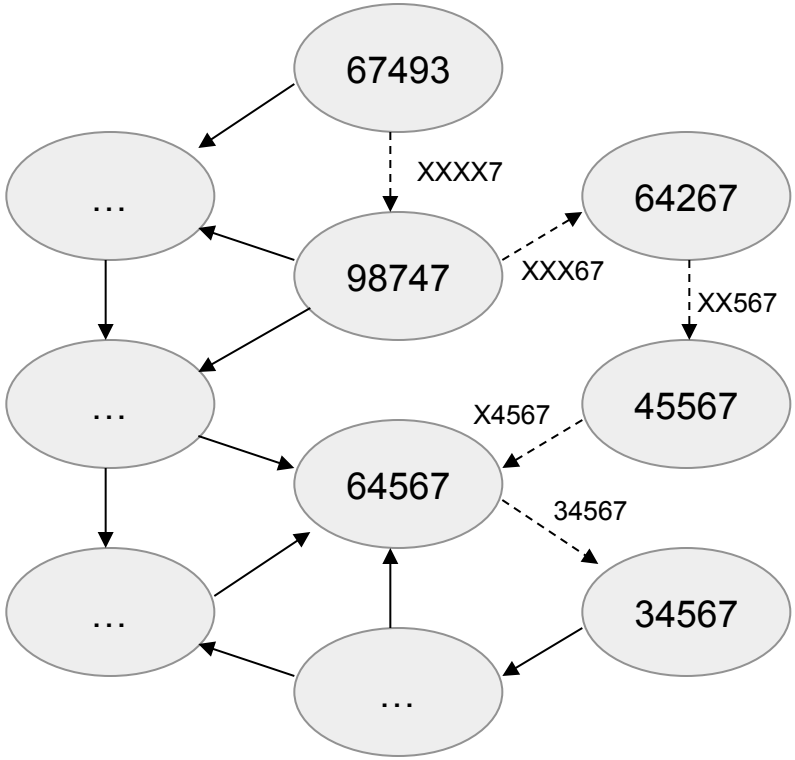


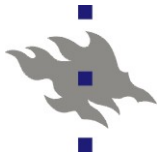
Rounting table with b*logb(N) entries

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 Algorithm in Detail

Data: M is the neighborhood set, D is the destination address, A is the current node, L is the leaf set, L_i denotes the i:th closest node identifier in the leaf set, and $R_{i,j}$ is the entry in the routing table R at column i, $0 \leq i \leq 2^b$ and row l, $0 \leq l \leq \lfloor 128/b \rfloor$

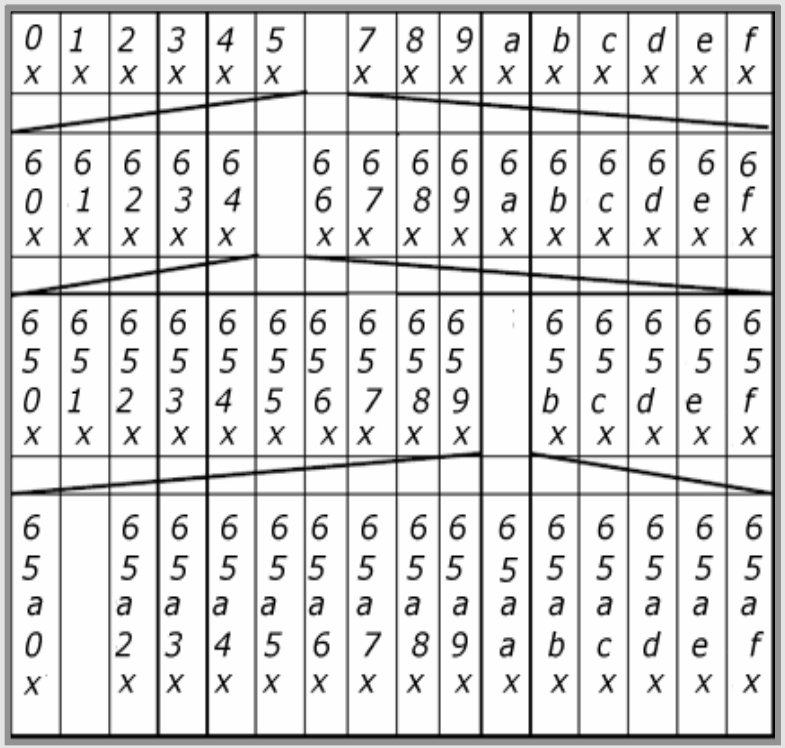
if $L_{-\lfloor |L|/2 \rfloor} \leq D \leq L_{\lfloor |L|/2 \rfloor}$ then
  /* D is within range of the leaf set or is the current node */
  forward to an element L_i of the leaf set with GUID closest to D or the current node
else
  /* Use the routing table */
  Let $l = sbl(D, A)$
  /* l is the longest common prefix of D and A */

  if $R_{l,i} \neq null$ then
    forward to $R_{l,i}$
    /* forward to a node with a longer common prefix */

  else
    /* There is no entry in the routing table */

    forward to any node T in $L \cup R \cup M$ that has a common prefix of length l but is numerically closer
  end
end

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

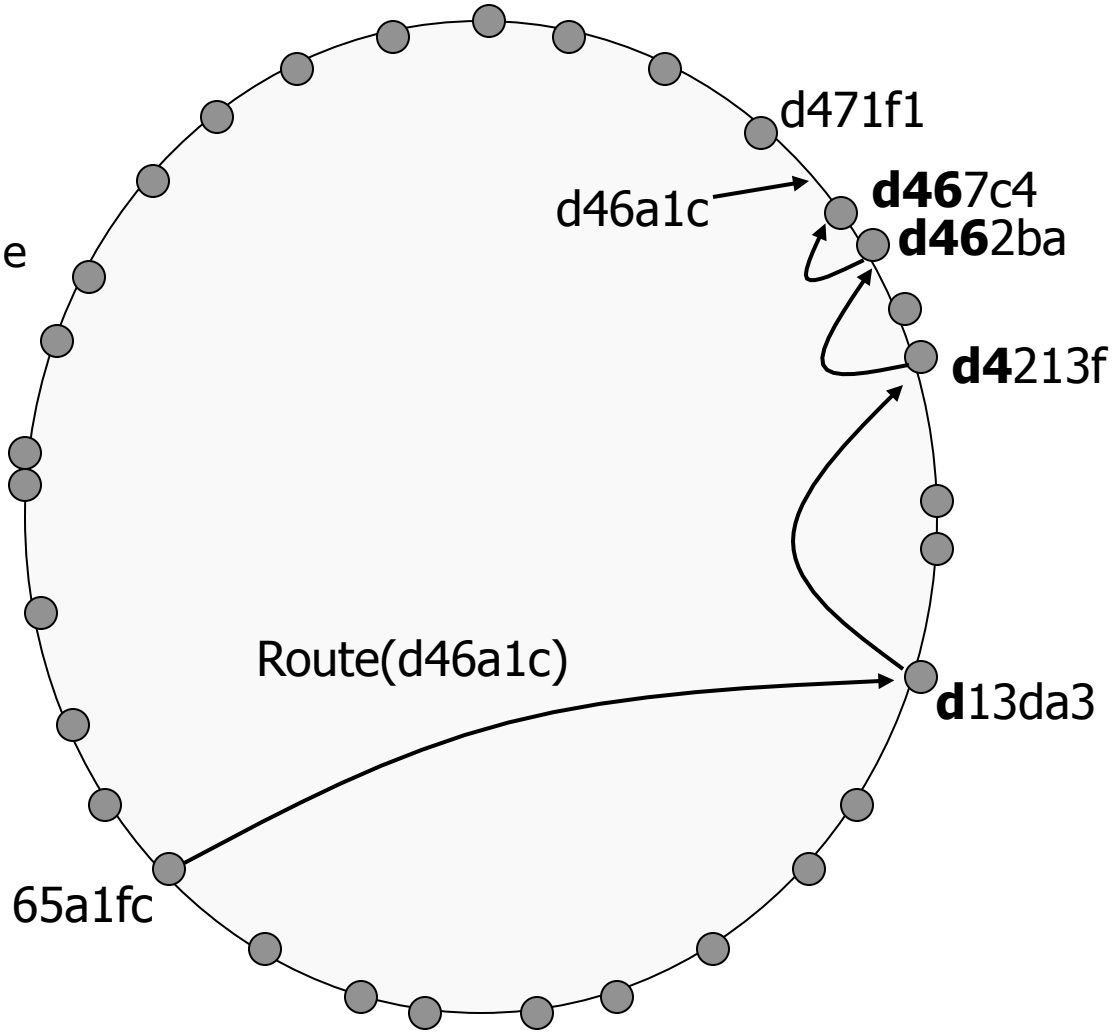
Each node knows something about the global reachability and then more about local nodes

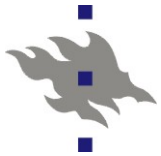
Pastry Routing Example

Prefix-based

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

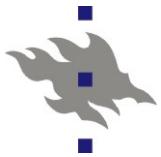
Neighbor set, leaf set and routing table.





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	$\log_B N$	$\log_B N$



Kademlia

Kademlia is a scalable decentralized P2P system based on the **XOR geometry**

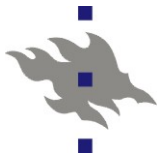
The algorithm is used by the BitTorrent DHT MainLine implementation, and therefore it is widely deployed

Kademlia is also used in kad, which is part of the eDonkey P2P file sharing system that hosts several million simultaneous users

Relying on the XOR geometry makes Kademlia unique compared to other proposals

Kademlia's routing table results in the same routing entries as for **tree geometries** when failures do not occur, such as Plaxton's algorithm

When failures occur, Kademlia can route around failures due to its geometry



Kademlia Overview

The initiating node maintains a shortlist of ***k* closest nodes**

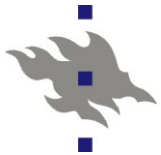
These are probed to determine if they are active

The replies of the probes are used to improve the shortlist

Closer nodes replace more distant nodes in the shortlist.

This iteration continues until *k* nodes have been successfully probed and there subsequent probes do not reveal improvements

This process is called a **node lookup** and it is used in most operations offered by Kademlia

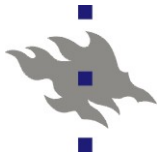


Kademlia

The lookup procedure can be implemented either using recursively or iteratively

The current Kademlia implementation uses the iterative process where the control of the lookup is with the initiating node

Leaving the network is straightforward and consistency is achieved by using leases

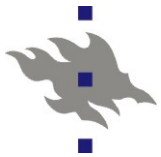


Kademlia performance

The routing tables of all Kademlia nodes can be seen to collectively maintain one large binary tree

Each peer maintains a fraction **O(log(n)/n)** of this tree

During a lookup, each routing step takes the message closer to the destination requiring at most ***O*(log *n*)** steps



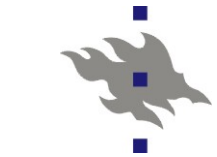
Kademlia Routing

In Kademlia, a node's neighbours are called **contacts**. They are stored in buckets, each of which holds a maximum of *k* contacts. These *k* contacts are used to improve redundancy

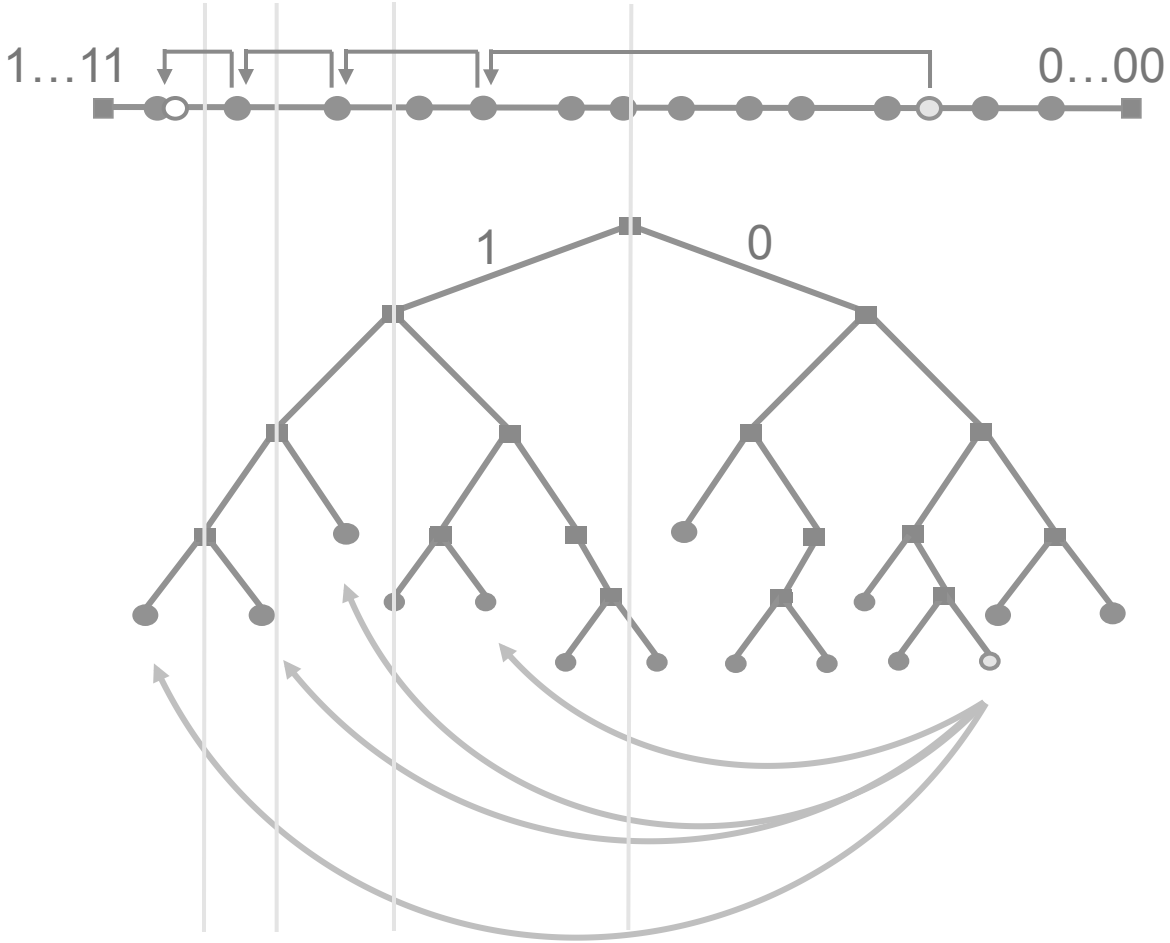
The routing table can be viewed as **a binary tree**, in which each node in the tree is a *k*-bucket

The buckets are organized by the distance between the current node and the contacts in the bucket

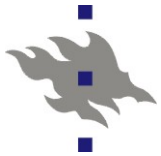
Every *k*-bucket corresponds to a specific distance from the node. Nodes that are in the *n*th bucket must have a **differing *n*th bit from the node's identifier**. With an identifier of 128 bits, every node in the network will classify other nodes in one of 128 different distances (first n-1 bits need to match for the nth list)



Simple iterative lookup

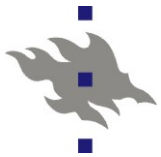


Consult the k-bucket that has the smallest distance to destination



Kademlia

	Kademlia
Foundation	XOR metric
Routing function	Matching key and nodeID
System parameters	Number of peers N, base of peer identifier B
Routing performance	*$O(\log_B N)$ + small constant*
Routing state	*$B \log_B N + B$*
Joins/leaves	*$\log_B N$ + small constant*



Viceroy

The key point in Viceroy is the emphasis on **constant degrees**. The primary motivation was to develop an algorithm that has constant linkage cost, logarithmic path length, and best achievable congestion under the constraints

It generally has constant degree such as CAN. Its degree is smaller than in Chord, Tapestry, and Pastry

Viceroy assumes **a global ordering** on all the nodes in the system, which may make practical deployments in decentralized environments challenging

Viceroy network

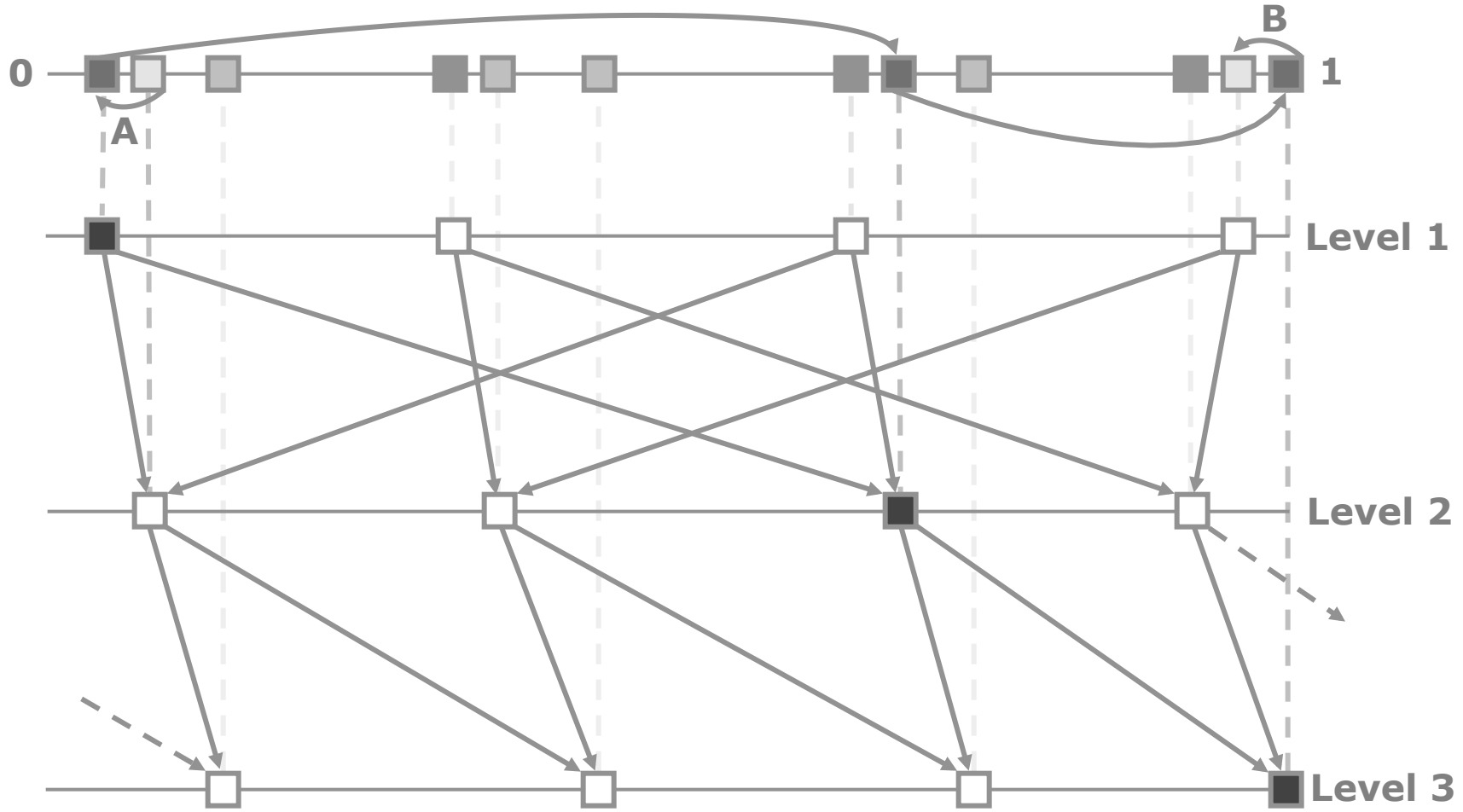
The idea is to approximate a butterfly network

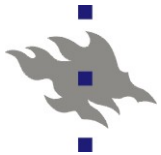
The butterfly network results in constant node degree and thus state

The algorithm is rather involved

Idea is to use the butterfly levels for routing and then vicinity search

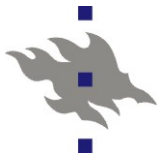
Message is routed upwards to the butterfly network root, and then downwards towards the correct destination, a shortcut may be used to reduce the routing cost


B
0
1
A
Level 1
Level 2
Level 3



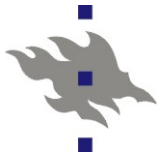
Viceroy

	Viceroy
Foundation	Butterfly network
Routing function	Routing using levels of tree, vicinity search
System parameters	Number of peers N
Routing performance	*O(log N)*
Routing state	*Constant*
Joins/leaves	*log N* *Note: assumes global ordering of nodes*

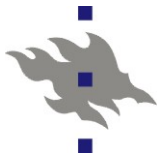


Summary

- Overlay networks have been proposed
  - Searching, storing, routing, notification,..
  - Lookup (Chord, Tapestry, Pastry), coordination primitives (i3), middlebox support (DOA)
  - Logarithmic scalability, decentralised,…
- Many applications for overlays
  - Lookup, rendezvous, data distribution and dissemination, coordination, service composition, general indirection support
- Deployment open. PlanetLab.



	CAN	Chord	Kademlia	Koorde	Pastry	Tapestry	Viceroy
Foundation	Multi-dimensional space (d-dimensional torus)	Circular space (hyper-cube)	XOR metric	de Bruijn graph	Plaxton-style mesh (hyper-cube)	Plaxton-style mesh (hyper-cube)	Butterfly network
Routing function	Maps (key,value) pairs to coordinate space	Matching key and nodeID	Matching key and nodeID	Matching key and nodeID	Matching key and prefix in nodeID	Suffix matching	Routing using levels of tree, vicinity search
System parameters	Number of peers N, number of dimensions d	Number of peers N	Number of peers N, base of peer identifier B	Number of peers N	Number of peers N, base of peer identifier B	Number of peers N, base of peer identifier B	Number of peers N
Routing performance	$O(dN^{1/d})$	$O(\log N)$	$O(\log_B N)$ + *small constant*	*Between* $O(\log \log N)$ *and* $O(\log N)$, *depending on state*	$O(\log_B N)$	$O(\log_B N)$	$O(\log N)$
Routing state	$2d$	$\log N$	$B\log_B N + B$	*From constant to* $\log N$	$2B\log_B N$	$\log_B N$	*Constant*
Joins/leaves	$2d$	$(\log N)^2$	$\log_B N$ + *small constant*	$\log N$	$\log_B N$	$\log_B N$	$\log N$

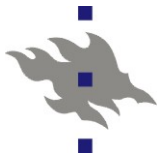


Comparison: Geometries

We observe that the foundations differ across the algorithms, but result in similar scalability properties

The foundations were considered earlier in the previous chapter and for the considered systems they are tori, ring, XOR metric, de Bruijn graph, hypercube, and butterfly network (note Koorde uses de Brujn over Chord, not presented in the slides)

The conclusions of several comparisons of the geometries are that the ring, XOR, and de Bruijn geometries are more flexible than the others and permit the choice of neighbours and alternative routes

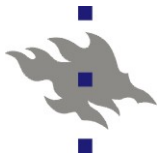


Comparison: Routing

The routing tables of DHTs can vary from size $O(1)$ to $O(n)$.
- The algorithms need to balance between maintenance cost and lookup cost
- From the view point of routing state Chord, Pastry, and Tapestry offer logarithmic routing table sizes, whereas Koorde and Viceroy and support constant or near-constant sizes

Churn and dynamic peers can also be supported with
- logarithmic cost in some of the systems, such as Koorde, Pastry, Tapestry, and Viceroy

Recent analysis indicates that large routing tables actually
- lead to both low traffic and low lookup hops. These good design points translate into one-hop routing for systems of medium size and two-hop routing for large systems



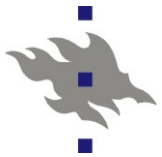
Comparison: Churn

Li et al. provide a comparison of different DHTs under churn
They examine the fundamental design choices of systems including Tapestry, Chord, and Kademlia. The insights based on this work include the following:

- Larger routing tables are more cost-effective than more frequent periodic stabilization

- Knowledge about new nodes during lookups may allow to eliminate the need for stabilization

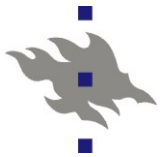
- Parallel lookups result in reduced latency due to timeouts, which provide information about the network conditions



Comparison: Network Proximity

Support for network proximity is one key feature of overlay algorithms. The three basic models for proximity awareness in DHTs are:

- **Geographic Layout.** Node identifiers are created in such a way that nodes that are close in the network topology are close in the nodeId space
- **Proximity Routing.** The routing tables do not take network proximity into account; however, the routing algorithm can choose a node from the routing table that is closest in terms of network proximity
- **Proximity Neighbour Selection.** In this model, the routing table construction takes network proximity into account. Routing table entries are chosen in such a way that at least some of them are close in the network topology to the current node



Asymptotic Tradeoffs

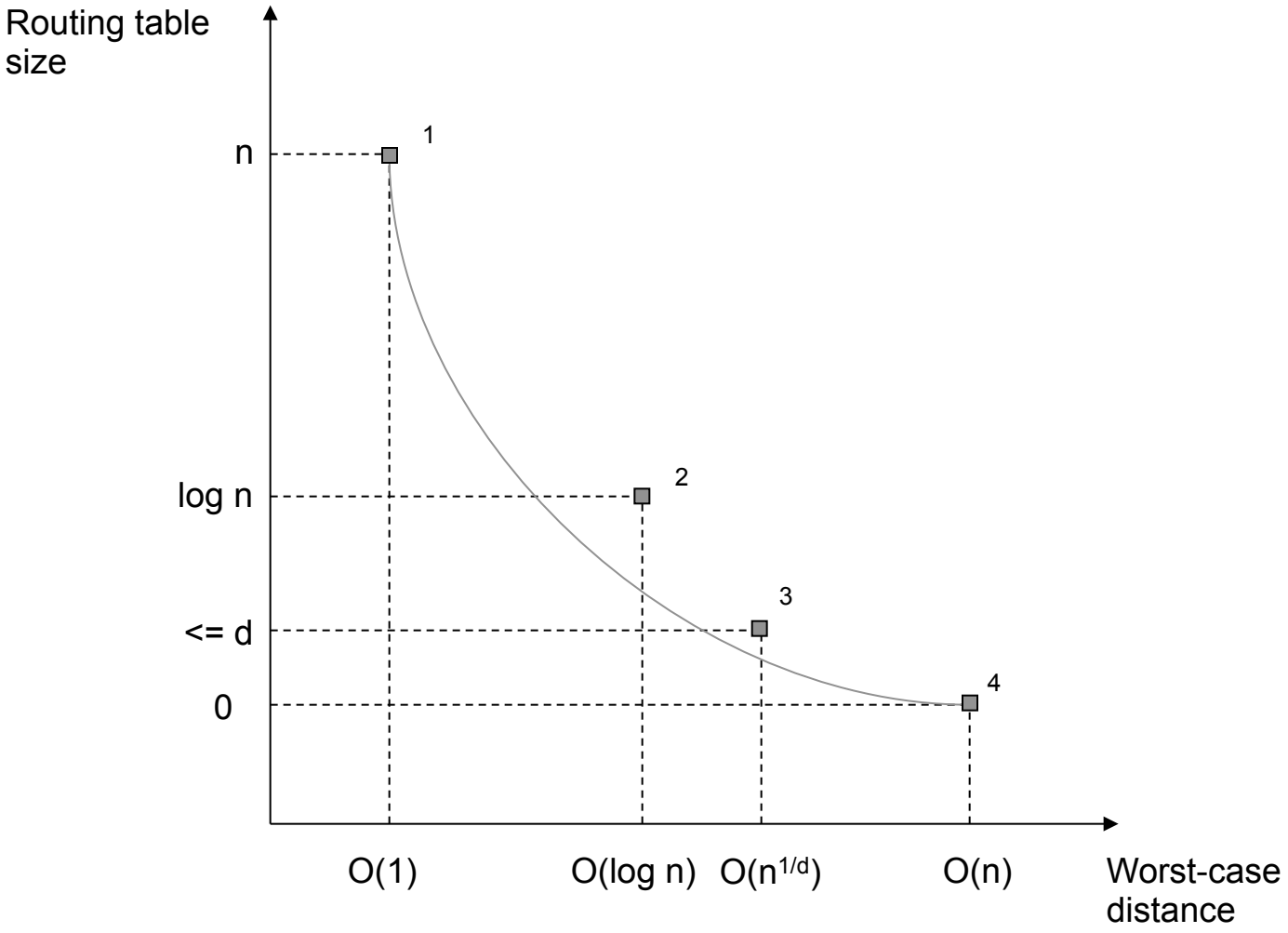
We analyze the asymptotic tradeoff curve between the routing table size and the network diameter

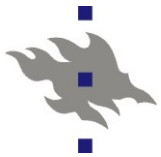
Analysis of the tradeoffs between the two metrics indicate that the routing table size of **Ω(log *n*)** is a **threshold point** that separates two distinct state-efficiency regions

One can observe that this point is in the middle of the symbolic asymptotic curve. If the routing table size is asymptotically smaller or equal, the requirement for congestion-free operation prevents it from achieving the smaller asymptotic diameter

When the routing table size is larger, the requirement for congestion-free operation does not limit the system anymore

Routing table size and network distance





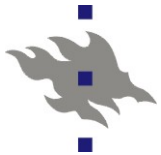
Criticism

There have been two main criticisms of structured systems

The first pertains to peer transience, which is an important factor in maintaining robustness. Transient peers result in churn, which is a current concern with DHTs.

The second criticism of structured systems stems from their foundation in consistent hashing, which makes it more challenging to implement scalable query processing than for unstructured systems. Given that the popular file-sharing applications rely extensively on metadata based queries, simple exact-match key searches are not sufficient for them and additional solutions are needed on top of the basic DHT API

It is also possible to combine structured and unstructured algorithms in so called hybrid models



Key security issues

Security is a weak point of many overlays and DHTs

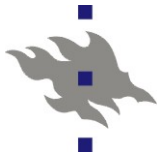
Not an issue with centrally managed system, but a
    significant concern for decentralized systems

Freenet was our key example of a secure P2P system

Overlays are vulnerable to the sybil attack
    One entity presents multiple identities for malicious intent.

Security Considerations

- Malicious nodes
  - Attacker floods DHT with data
  - Attacker returns incorrect data
    - self-authenticating data
  - Attacker denies data exists or supplies incorrect routing info
- Basic solution: using redundancy
  - k-redundant networks
- What if attackers have quorum?
  - Need a way to control creation of node Ids
  - Solution: secure node identifiers
    - Use public keys



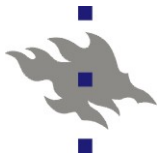
Sybil and Eclipse attacks

Sybil attack

- Malicious nodes overwhelm the network with identities
- Without a central authority that certifies identifies (binding real-world person to nodeID) no realistic approach exists to completely stop the sybil attack

Eclipse attack

- A group of malicious nodes tries to dominate the neighbor set
  - Start with Sybil and then neighbour discovery
- Network partitions



Skip graph I

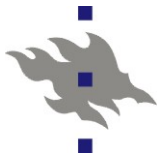
A **skip graph** is a probabilistic structure based on the **skip list** data structure

The skip list has simple and easy insert and delete operations that do not require tree rearrangements. Thus the operations are fast

The skip list is a set of **layered ordered linked lists**. All nodes are part of the bottom layer 0 list. Part of the nodes take part in the layer 1 with some fixed probability. For each layer there is a probability for a node to be part of that layer

As a result the upper layers of a skip list are **sparse**. This means that a lookup can quickly go through the list by traversing the sparse upper layer until it is close to the target

Additional material

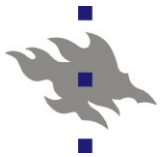


Skip graph II

The downside of this approach is that the sparse upper layer nodes are potential hotspots and single points of failure.

Skip graphs address this limitation and introduce multiple lists at each level to improve redundancy. Every node participates in one of the lists at each level

On average $O(\log n)$ levels are needed in the structure, where n is the number of nodes



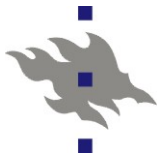
Skip Graph III

The skip graph is a distributed version of the skip list and its performance is comparable to the other DHTs

Each node in a skip graph has average of log *n* neighbours

The main benefit of the structure comes from its ability to support **prefix** and **proximity** search operations. DHTs guarantee that a data can be located, but they do not typically guarantee where the data will be located

Skip graphs are able to support location-sensitive name searches, because they use ordered lists



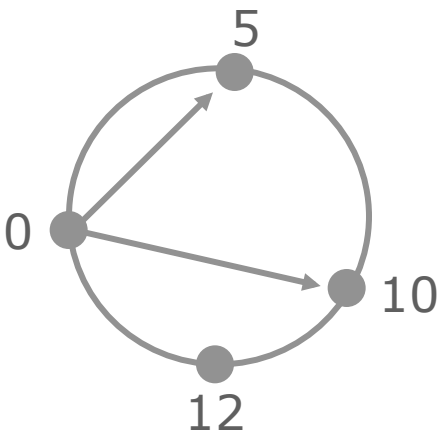
CANON: Adding Hierarchy to DHTs

Most DHTs that have been proposed are **flat** and **non-hierarchical** structures. They thus contrasts the traditional distributed systems, which have employed hierarchy to achieve scalability

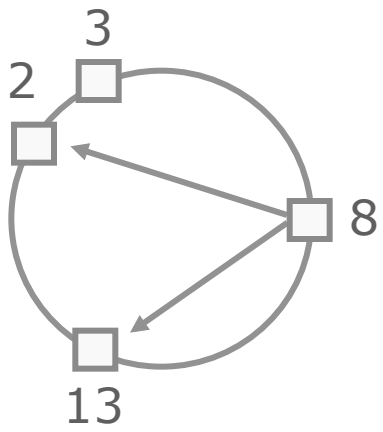
A hierarchical DHT can be constructed that retains the homogeneity of load and functionality of the flat DHTs. A generic construction called Canon has been shown to offer the same routing state and routing hops trade-off found in the flat DHT designs

The benefits of this approach include fault isolation, adaptation to the underlying physical network and its organizational boundaries, and hierarchical storage of content and access control

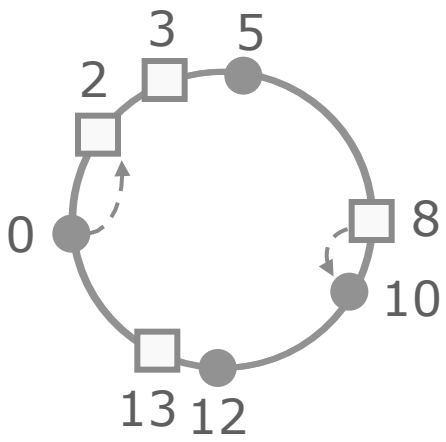
CANON Example



Ring A



Ring B



The Merged Ring

The nodes keep their original links
Each node m in one ring creates a link to a node m′ in the other ring if and only if:
• m′ is the closest node that is at least distance 2k away for some $0 \leq k \leq N$
• m′ is closer to m than any node in the ring of m