Overlay and P2P Networks
Structured Networks and DHTs

Dr. Samu Varjonen
8.2.2016
Contents

• Today
  • Distributed Hash Tables (DHTs)
• Thursday
  • Power-law networks
Pastry I

- A DHT based on a circular flat identifier space

- Invariant: node with numerically closest id maintains object

- 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 Components

Leaf set
- \( \frac{L}{2} \) smaller and larger numerically closest nodes. \( L \) is a configuration parameter (typically 16 or 32).
- To ensure reliable message delivery
- To store replicas for fault tolerance

Routing table

Neighborhood set
- \( M \) entries for nodes “close” to the present node (typically \( M = 32 \)). Used to construct routing table with good locality properties
Node departure (failure)

Leaf set members exchange heartbeat

Leaf set repair (eager): request set from farthest live node in set

Routing table repair (lazy): get table from peers in the same row, then higher rows
Leaf set is a ring:

If $L/2 = 1$: each node has a pointer to its ring successor and predecessor.

If $L/2 = k$: each node has a pointer to its $k$ ring successors and $k$ predecessors.

Ring breaks if $k$ consecutive nodes fail concurrently.

$k - 1$ concurrent node failures can be tolerated.
Pastry: Routing procedure

if (destination is within range of our leaf set)
    forward to numerically closest member
else
    let $l =$ length of shared prefix
    let $d =$ value of $l$-th digit in $D$’s address
    if ($R_i^d$ exists)
        forward to $R_i^d$
    else
        forward to a known node that
        (a) shares at least as long a prefix
        (b) is numerically closer than this node
Joining the Network

The join consists of the following steps:

• Create NodeID and obtain neighbour set from the topologically (network) nearest node.
• Route message to NodeID.
• Each Pastry node processing the join message will send a row of the routing table to the new node. The Pastry nodes will update their long distance routing table if necessary (if numerically smaller for a given prefix).
• Receive the final row and a candidate leaf set.
• Check table entries for consistency. Send routing table to each neighbour.
Node departure (failure)

Leaf set members exchange heartbeat

Leaf set repair (eager): request set from farthest live node in set
Routing table repair (lazy): get table from peers in the same row, then higher rows
Routing table of a Pastry node with nodeId \textbf{65a1x}, \( b = 4 \). Digits are in base 16, \( x \) represents an arbitrary suffix.

The IP address associated with each entry is not shown.
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.
Proximity

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.

**Proximity invariant:** Each routing table entry refers to a node close to the local node (in the proximity space) among all nodes with the appropriate prefix.
Pastry: Routes in proximity space

Source: Presentation by P. Druschel et al. Scalable peer-to-peer substrates: A new foundation for distributed applications?
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).
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
# Pastry and Tapestry

<table>
<thead>
<tr>
<th></th>
<th>Pastry</th>
<th>Tapestry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
<td>Plaxton-style mesh (hyper-cube)</td>
<td>Plaxton-style mesh (hyper-cube)</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
<td>Matching key and prefix in nodeID</td>
<td>Suffix matching</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
<td>Number of peers N, base of peer identifier B</td>
<td>Number of peers N, base of peer identifier B</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
<td>$O\left(\log_b N\right)$&lt;br&gt;Note proximity metric</td>
<td>$O\left(\log_b N\right)$&lt;br&gt;Note surrogate routing</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
<td>$2B \log_b N$</td>
<td>$\log_b N$</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
<td>$\log_b N$</td>
<td>$\log_b N$</td>
</tr>
</tbody>
</table>
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 $O(d \times N^{1/d})$ 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.
Routing to point P

1. Node checks whether it or its neighbors contain the point P
2. If does not contain then
3. Node orders the neighbors by Cartesian distance between them and the point P
4. Forwards the search request to the closest one
5. Repeat step 1

1. When cannot repeat, return result to user
**CAN: average path length**

Total path length is given by the sum
\[0 \times 1 + 1 \times 2d + 2 \times 4d + 3 \times 6d \ldots\]

Average path length is the total path length divided by the number of nodes

\[\text{Avg. path length} = \frac{TPL}{n} = d \times \frac{n^{1/d}}{4}\]

Virtual $d$-dimensional Cartesian coordinate system on a $d$-torus

Example: 2-$d$ $[0,1] \times [1,0]$

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 $P$
Extensions

Increasing dimensions of the coordinate space reduces path length and latency with small routing table size increase.

Landmarks for topology sensitive construction

Nodes measure RTT to landmarks, order landmarks, partition coordinate space into $m!$ equal sizes. Join nearest partition in landmark ordering.

Multiple hash functions

Realities. Multiple independent coordinate spaces.
## Content Addressable Network (CAN)

<table>
<thead>
<tr>
<th>CAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
</tr>
<tr>
<td>Multi-dimensional space (d-dimensional torus)</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
</tr>
<tr>
<td>Maps (key,value) pairs to coordinate space</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
</tr>
<tr>
<td>Number of peers N, number of dimensions d</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
</tr>
<tr>
<td>( O(dN^{1/d}) )</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
</tr>
<tr>
<td>( 2d )</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
</tr>
<tr>
<td>( 2d )</td>
</tr>
</tbody>
</table>
The XOR Geometry

The Kademlia P2P system defines a routing metric in which the distance between two nodes is the numeric value of the exclusive OR (XOR) of their identifiers.

The idea is to take messages closer to the destination by using the XOR distance $d(x,y) = \text{XOR}(x,y)$ (taken as an integer).

The routing therefore "fixes" high order bits in the current address to take it closer to the destination.

Satisfies triangle property, symmetric, unidirectional.
XOR Metric and Triangle Property

Triangle inequality property
\[ d(x,z) \leq d(x,y) + d(y,z) \]

Easy to see that XOR satisfies this

Useful for determining distances between nodes

Unidirectional:
For any given point x and a distance D > 0, there is exactly one point y such that \( d(x,y) = D \). This means that lookups converge.
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 Routing Table

For each $i$ $(0 \leq i < 160)$ every node keeps a list of nodes of distance between $2^i$ and $2^{(i+1)}$ from itself.

Call each list a **k-bucket**. The list is sorted by time last seen.

The value of $k$ is chosen so that any given set of $k$ nodes is unlikely to fail within an hour.

The list is **updated** whenever a node receives a message.
K-buckets

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 160 bits, every node in the network will classify other nodes in one of 160 different distances (first $n$-1 bits need to match for the $n$th list)
Each node knows more about close nodes than distant nodes.
Key space of each bucket grows with the power of 2 with the distance.
Querying for an ID will on average halve the distance to the target in each step.
Joining the network

Find one active node
Insert the bootstrap node into one of the k-buckets

Lookup new node id to populate the other nodes’ k-buckets with the new node id and the joining node’s k-buckets

FIND_NODE(key) : The recipient of the request will return the k nodes in his own buckets that are the closest ones to the requested key

Refresh k-buckets further away than the k-bucket with the bootstrap node. Refresh is a random lookup for a key within the range of a k-bucket
Node Lookup

Goal: Find k nodes closest to ID T

*Initial Phase:*
- Select $\alpha$ nodes closest to $T$ from the routing table
- Send FIND_NODE($T$) to each of the $\alpha$ nodes in parallel

*Iteration:*
- Select $\alpha$ nodes closest to $T$ from the results of previous RPC
- Send FIND_NODE($T$) to each of the $\alpha$ nodes in parallel
- Terminate when a round of FIND_NODE($T$) fails to return any closer nodes

*Final Phase:*
- Send FIND_NODE($T$) to all of k closest nodes not already queried
- Return when have results from all the k-closest nodes.
Joining Node (u):
- Borrow an alive node’s ID (w) offline
- Initial routing table has a single k-bucket containing u and w.
- u performs FIND_NODE(u) to learn about other nodes

Inserting new entry (v)

Find bucket B with longest common prefix as v

Is B full?
- no
  - insert
- yes
  - B has u?
    - no
      - Don’t insert
    - yes
      - Split B, redistribute contacts & insert v
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:
- Graceful → leave message to closest neighbours
- Ungraceful → the k other nodes have replicas

Periodical republishing of key-value pairs to ensure the availability on k nodes closest to the key.

Also, the nodes consider all encountered nodes to be added to the k-buckets.
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

<table>
<thead>
<tr>
<th>Kademlia</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
</tr>
<tr>
<td>XOR metric</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
</tr>
<tr>
<td>Matching key and nodeID</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
</tr>
<tr>
<td>Number of peers $N$, base of peer identifier $B$</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
</tr>
<tr>
<td>$O(\log_B N) + \text{small constant}$</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
</tr>
<tr>
<td>$B\log_B N + B$</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
</tr>
<tr>
<td>$\log_B N + \text{small constant}$</td>
</tr>
</tbody>
</table>
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.
DHT: A General Approach

What is an address?

Base b with n digits

How to route efficiently?

Fix at least one digit per hop or take to the numerically closest destination based on routing table

How efficient is this?

Log N steps gives $O(\log N)$ state and $O(\log N)$ hops!
DHT: A General Approach

How to populate routing table?

Iterative nearest neighbour search to fill the routing table. Get enough information to be able to populate the routing table.
<table>
<thead>
<tr>
<th>CAN</th>
<th>Chord</th>
<th>Kademlia</th>
<th>Koorde</th>
<th>Pastry</th>
<th>Tapestry</th>
<th>Viceroy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
<td>Multi-dimensional space (d-dimensional torus)</td>
<td>Circular space (hyper-cube)</td>
<td>XOR metric</td>
<td>de Bruijn graph</td>
<td>Plaxton-style mesh (hyper-cube)</td>
<td>Plaxton-style mesh (hyper-cube)</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
<td>Maps (key,value) pairs to coordinate space</td>
<td>Matching key and nodeID</td>
<td>Matching key and nodeID</td>
<td>Matching key and nodeID</td>
<td>Matching key and prefix in nodeID</td>
<td>Suffix matching</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
<td>Number of peers N, number of dimensions d</td>
<td>Number of peers N</td>
<td>Number of peers N, base of peer identifier B</td>
<td>Number of peers N</td>
<td>Number of peers N, base of peer identifier B</td>
<td>Number of peers N</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
<td>$O(dN^{1/d})$</td>
<td>$O(\log N)$</td>
<td>$O(\log B N) + \text{small constant}$</td>
<td>Between $O(\log \log N)$ and $O(\log N)$, depending on state</td>
<td>$O(\log B N)$</td>
<td>$O(\log B N)$</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
<td>$2d$</td>
<td>$\log N$</td>
<td>$\log B N + B$</td>
<td>From constant to $\log N$</td>
<td>$2\log B N$</td>
<td>$\log B N$</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
<td>$2d$</td>
<td>$(\log N)^2$</td>
<td>$\log B N + \text{small constant}$</td>
<td>$\log N$</td>
<td>$\log B N$</td>
<td>$\log B N$</td>
</tr>
</tbody>
</table>
Comparing geometries

Gummadi et al. compared the different geometries, including the tree, hypercube, butterfly, ring, and XOR geometries.

Loguinov et al. complemented this list with de Bruijn graphs.

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

The ring and XOR geometries were also found to be the most flexible in terms of choosing neighbours and routes.

Only de Bruijn graphs allow alternate paths that are independent of each other.
Comparison

Can you choose neighbours?

Can you choose routes?

Are there alternative routes?

Are there alternative routes without overlap?
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Hypercube</th>
<th>Ring</th>
<th>Butterfly</th>
<th>XOR</th>
<th>De Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbour selection</td>
<td>Yes*</td>
<td>1</td>
<td>Yes**</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Route selection</td>
<td>1</td>
<td>Yes</td>
<td>Yes**</td>
<td>1</td>
<td>Some</td>
<td>Yes</td>
</tr>
<tr>
<td>Sequential neighbours</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Independent paths</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* For a tree there can be multiple choices for neighbours.

** Gummadi et al. observe that Chord can be extended to allow nodes to pick the ith neighbour \([(a+2^i),(a+2^{i+1})]\) instead of the closest node.

Route selection offers ability to route around faults and increased resilience.

It is possible to combine geometries: Pastry is a tree/ring hybrid.
Discussion

Based on previous table the ring looks pretty good

But this is partly due to the sequential neighbours property (predecessor and successor on the ring)

If sequential neighbours is added to other geometries, XOR and de Bruijn are also good
Comparison: Geometries

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

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.

Note: it is possible to combine these. Example: Pastry that combines the tree and ring geometries.
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 viewpoint 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 (Kademlia is an example)

- **Parallel lookups** result in reduced latency due to timeouts, which provide information about the network conditions (but they may poison routing tables, more on this later)
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 $\Omega(\log n)$ 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

Routing table size

- $\log n$
- $\leq d$
- 0
- $n$

Worst-case distance

- $O(1)$
- $O(\log n)$
- $O(n^{1/d})$
- $O(n)$
Criticisms

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.
Additional material (not part of exam)

Butterfly networks and Viceroy

Skip graph

CANON: merging Chord rings

De Bruijn graph
Butterfly Geometry

A \textit{k-ary n-fly} network consists of $k^n$ source nodes, $n$ stages of $k^{n-1}$ switches, and $k^n$ destination nodes. The network is unidirectional and the degree of each switching node is $2k$. The diameter of the network is logarithmic to the number of source nodes. At each level $l$, a switching node is connected to the identically numbered element at level $l + 1$ and to a switching node whose number differs from the current node only at the $l$th most significant bit. The main drawback of this structure is that there is \textbf{only one path from a source to a destination}, in other words, there is no path diversity. In addition, butterfly networks do not have as good locality properties as tori.
Butterfly network (with a tree)
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.
# Viceroy

<table>
<thead>
<tr>
<th>Viceroy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
</tr>
<tr>
<td>Butterfly network</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
</tr>
<tr>
<td>Routing using levels of tree, vicinity search</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
</tr>
<tr>
<td>Number of peers N</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
</tr>
<tr>
<td>$O(\log N)$</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
</tr>
<tr>
<td>Constant</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
</tr>
<tr>
<td>$\log N$</td>
</tr>
<tr>
<td>Note: assumes global ordering of nodes</td>
</tr>
</tbody>
</table>
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.
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.
Most DHTs that have been proposed are flat and non-hierarchical structures. They thus contrast 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.
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 $m$
De Bruijn Graph

An $n$-dimensional de Bruijn graph of $k$ symbols is a directed graph representing overlaps between sequences of symbols. It has $k^n$ vertices that represent all possible sequences of length $n$ of the given symbols.

In a $n$-dimensional de Bruijn graph with 2 symbols, there are $2^n$ nodes, each of which has a unique $n$-bit identifier.
Creating a de Bruijn graph

The node with identifier $i$ is connected to
nodes $2i \mod 2^n$ and $2i + 1 \mod 2^n$

A routing algorithm can route to any destination in $n$ hops by
successively shifting in the bits of the destination
identifier.

Routing a message from node $m$ to node $k$ is accomplished
by taking the number $m$ and shifting in the bits of $k$ one at
a time until the number has been replaced by $k$
De Bruijn Graph

Consider a node $n$ with identifier $b_1 b_2 \ldots b_k$, $b_i \in \{0, 1\}$

$n$ has an out-edge to the nodes with identifier $b_2 \ldots b_k 0$ and $b_2 \ldots b_k 1$.

Node 00: out edge to 00 and 01
Node 01: out edge to 10 and 11
Node 10: out edge to 00 and 01
Node 11: out edge to 10 and 11

This adjacency scheme, based on shifting the identifier strings associated with a node yields a simple prefix based routing policy.
Constructing de Bruijn Graphs

De Bruijn graph for $2^m$ node network can be constructed in a recursive fashion from a $2^{m-1}$ node network.

Take the edge of the $2^{m-1}$ node network

Add a node in the middle

Details:
Example: Adding a digit

Example: Adding a digit