Overlay and P2P Networks
Structured Networks and DHTs

Dr. Samu Varjonen

6.2.2017
Contents

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

• A DHT based on a circular flat identifier space

• 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. 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
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
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.
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 \(65a1x\), \(b = 4\). Digits are in base 16, \(x\) represents an arbitrary suffix.

The IP address associated with each entry is not shown.
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
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 Routing Example
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.

Proximity metric is a **scalar value** that reflects the distance between any pair of nodes, such as the round trip time.
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)` (to exact match instead of closest match)
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 node’s 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, e.g., taking consistently the next higher entry in routing table)
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

Complex and hard to maintain
## 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>$O(\log_B N)$</td>
<td>$O(\log_B N)$</td>
</tr>
<tr>
<td></td>
<td><em>Note proximity metric</em></td>
<td><em>Note surrogate routing</em></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 (similar to join)

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*1+1*2d+2*4d+3*6d...

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

\[
TPL = \sum_{i=1}^{\frac{n^{1/d}}{2}-1} i \cdot 2d + \frac{n^{1/d}}{2} (n^{1/d} - 1) \cdot d + \sum_{i=\frac{n^{1/d}}{2}+1}^{n^{1/d}} i \cdot 2(n^{1/d} - i) \cdot d + n^{1/d} \cdot 1
\]

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

Source: www.mpi-inf.mpg.de/departments/d5/teaching/ws03_04/p2p-data/11-18-paper2.ppt
Peer X's coordinate neighbor set = {A B D Z}
New Peer Z's coordinate neighbor set = {A C D X}
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><strong>Routing function</strong></td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
</tr>
</tbody>
</table>
Kademlia and 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**

- $d(x,x) = 0$
- $d(x,y) > 0$, if $x \neq y$
- For all $x,y : d(x,y) = d(y,x)$ -- symmetry
- $d(x,z) \leq d(x,y) + d(y,z)$ -- triangle inequality

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

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 users
Kademlia

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.

[Diagram of network partition for node 110]
Kademlia Routing Table

For each i (0 ≤ i < 160 (SHA-1)) every node keeps a list of k nodes of distance between $2^i$ and $2^{i+1}$ from itself

[1, 2) [2, 4) [4, 8) [8, 16) [16, 32) [32, 64) [64, 128) [128, 256]

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)
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 α nodes closest to T from the routing table
• Send FIND_NODE(T) to each of the α nodes in parallel

Iteration:
• Select α nodes closest to T from the results of previous RPC
• Send FIND_NODE(T) to each of the α 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.
Kademia

The lookup procedure can be implemented either using recursively or iteratively

The current Kademia 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** |
|------------------------|------------------------|
| **Foundation** | XOR metric |
| **Routing function** | Matching key and nodeID |
| **System parameters** | Number of peers N, base of peer identifier B |
| **Routing performance** | $O(\log_B N) + \text{small constant}$ |
| **Routing state** | $B \log_B N + B$ |
| **Joins/leaves** | $\log_B N + \text{small constant}$ |
Summary

- Overlay networks have been proposed
  - Searching, storing, routing, notification,..
  - Lookup (Chord, Tapestry, Pastry), coordination primitives (i3)
  - Logarithmic scalability, decentralised,..
<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>$\text{Between } O(\log \log N)\text{ and } O(\log N), \text{ depending on state}$</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>$B\log_B N + B$</td>
<td>From constant to log N</td>
<td>$2B\log_B N$</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>
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 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.
Routing table size and network distance

Routing table size

n
log n
<= d
0

Worst-case distance

O(1) O(log n) O(n^{1/d}) O(n)
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.
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.
Criticism

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.