Overlay and P2P Networks

Structured Networks and DHTs

Prof. Sasu Tarkoma

6.2.2014
Contents

• Today
  • Semantic free indexing
  • Consistent Hashing
  • Distributed Hash Tables (DHTs)
• Thursday (Dr. Samu Varjonen)
  • DHTs continued
  • Discussion on geometries
Rings

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

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

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

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

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

Identifiers are ordered on an identifier circle modulo $2^m$

The Chord ring with m-bit identifiers

A node has a well determined place within the ring

A node has a predecessor and a successor

A node stores the keys between its predecessor and itself

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

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

Each node maintains a routing table with at most $m$ entries.

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

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

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

$m=6$

for $j=1, \ldots, m$ the fingers of $p+2^{j-1}$
Chord routing II

Routing steps

1. check whether the key \( k \) is found between \( n \) and the successor of \( n \)
2. 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

$2^m - 1$  0

$m = 6$
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

<table>
<thead>
<tr>
<th><strong>Foundation</strong></th>
<th>Circular space (hyper-cube)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Routing function</strong></td>
<td>Matching key and nodeID</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
<td>Number of peers N</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
<td>$\log N$</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
<td>$(\log N)^2$</td>
</tr>
</tbody>
</table>
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 \textbf{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
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
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 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.
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.

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.

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

NodeID space

Route(d46a1c)

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 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).
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>
</tbody>
</table>
| **Routing performance** | $O(\log_B N)$  
*Note proximity metric* | $O(\log_B N)$  
*Note surrogate routing* |
| **Routing state** | $2\log_B N$ | $\log_B N$ |
| **Joins/leaves** | $\log_B N$ | $\log_B N$ |
CAN and Hypercubes
Hypercubes

The distance between two nodes in the hypercube geometry is the number of bits by which their identifier differ. At each step a greedy forwarding mechanism corrects (or fixes) one bit to reduce the distance between the current message address and the destination. Hypercubes are related to tori. In one dimension a line bends into a circle (a ring) resulting in a 1-torus. In two dimensions, a rectangle wraps into the two-dimensional torus, 2-torus. An n dimensional hypercube can be transformed into an n-torus by connecting the opposite faces together. The Content Addressable Network (CAN) is an example of a DHT based on a d-dimensional torus.
Differences

The main difference between hypercube routing and tree routing is that the former allows bits to be fixed in any order.

Tree routing requires that the bits are corrected in a strict order (digit by digit, still can be redundancy in the table).

Thus hypercube is more restricted in selecting neighbours in the routing table but offers more possibilities for route selection!
Hypercubes

\[ d = 0 \quad N = 1 \]

\[ d = 1 \quad N = 2 \]

\[ d = 2 \quad N = 4 \]

\[ d = 3 \quad N = 8 \]

\[ d = 4 \quad N = 16 \]
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

6. 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

\[
P = 0 \cdot 1 + \sum_{i=1}^{\frac{n^{1/d}}{2} - 1} i \cdot 2id + \frac{n^{1/d}}{2} \cdot (n^{1/d} - 1)d + \sum_{i=\frac{n^{1/d}}{2} + 1}^{n^{1/d}} i \cdot 2(n^{1/d} - i)d + n^{1/d} \cdot 1
\]

\[
\text{Avg. path length} = \frac{\text{TPL (Total path length)}}{n \ (# \ of \ Nodes)} = d \cdot \frac{n^{1/d}}{4}
\]

Source: www.mpi-inf.mpg.de/departments/d5/teaching/ws03_04/p2p-data/11-18-paper2.ppt
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 $P$
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
<td>Multi-dimensional space (d-dimensional torus)</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
<td>Maps (key,value) pairs to coordinate space</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
<td>Number of peers N, number of dimensions d</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
<td>$O(dN^{1/d})$</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
<td>$2d$</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
<td>$2d$</td>
</tr>
</tbody>
</table>
Kademlia and the XOR geometry
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.
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.
Kademlia tree

Every node keeps touch with at least one node from each of its subtrees. Corresponding to each subtree, there is a k-bucket.
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.
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)
Details

For each $i \leq i < 160$ 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 give set of k nodes is unlikely to fail within an hour.

The list is updated whenever a node receives a message.
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.
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

Lookup operation returns k-closest nodes of the receiver

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
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.
Simple iterative lookup

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

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