Peer-to-Peer Systems

DHT examples, part 2
(Pastry, Tapestry and Kademlia)

Michael Welzl  michael.welzl@uibk.ac.at

DPS NSG Team http://dps.uibk.ac.at/nsg
Institute of Computer Science
University of Innsbruck, Austria

Plaxton routing

- Plaxton, Rajamaran and Richa: mechanism for efficient dissemination of objects in a network, published in 1997
  - Before P2P systems came about!
- Basic idea: prefix-oriented routing (fixed number of nodes assumed)
  - Object with ID A is stored at the node whose ID has the longest common prefix with A
  - If multiple such nodes exist, node with largest common suffix is chosen
- Goal: uniform data dissemination
- Routing based on pointer list (object - node mapping) and neighbor list (primary + secondary neighbors)
- Generalization of routing on a hypercube
- Basis for well known DHTs Pastry, Tapestry (and follow-up projects)
  - Method adapted to needs of P2P systems + simplified

Pastry: Topology

- Identifier space:
  - 2^l-bit identifiers (typically: l = 128), wrap-around at 2^l - 1 ↔ 0
  - Interpret identifiers to the base of 2^l (typically: b = 4, base 16)
  - Prefix-based tree topology
  - Leaves can be keys and node IDs
  - (key, value) pairs managed by numerically closest node

Pastry: Routing Basics

- Routing in Pastry:
  - In each routing step, query is routed towards "numerically" closest node
  - That is, query is routed to a node with a one character longer prefix (= b Bits)
  - $O(\log_b N)$ routing steps
  - If that is not possible:
    - Route towards node that is numerically closer to ID

Pastry: Routing Basics /2

- Example:
  - Node ID = 0221
  - Base = 3 (one power of 3), because it is easier to draw :)

Pastry: Routing Basics /3

- Example:
  - Node ID = 0221
  - Base = 3 (one power of 3), because it is easier to draw :)

Plaxton:

- Plaxton, Rajamaran and Richa: mechanism for efficient dissemination of objects in a network, published in 1997
  - Before P2P systems came about!
- Basic idea: prefix-oriented routing (fixed number of nodes assumed)
  - Object with ID A is stored at the node whose ID has the longest common prefix with A
  - If multiple such nodes exist, node with largest common suffix is chosen
- Goal: uniform data dissemination
- Routing based on pointer list (object - node mapping) and neighbor list (primary + secondary neighbors)
- Generalization of routing on a hypercube
- Basis for well known DHTs Pastry, Tapestry (and follow-up projects)
  - Method adapted to needs of P2P systems + simplified
Pastry: Routing Basics /4

- Data (key-value-pairs) are managed in numerically closest node
  - keys \( \rightarrow \) nodes:
    - 0002 \( \rightarrow \) 0002, 01**
    - 0002 \( \rightarrow \) 0110

- Linking between Prefix-areas:
  - Nodes within a certain prefix area know IP addresses of each other
  - Each node in a prefix area knows one or more nodes from another prefix area

- From which prefix areas should a node know other nodes?
  - Links to shorter prefix node areas on each prefix level

Pastry: Routing Basics /5

- Example:
  - Node in area 222* knows nodes from prefix areas
    - 220*, 221* & 20**, 21** & 0***, 1***

- Logarithmic number of links:
  - For prefix-length p: \( (\log(N)) \) links to other nodes with prefix length p,
    but with a different digit at position p

- \( \log(N) \)

Pastry: Routing Information

- Challenges
  - Efficiently distribute search tree among nodes
  - Honor network proximity

- Pastry routing data per node
  - Routing table
  - Leaf set
  - Neighborhood set

- Routing table
  - Long distance links to other prefix realms
  - \( l/b \) rows: one per prefix length
  - \( 2^{b-1} \) columns: one per digit different from local node ID

Pastry: Routing Table

- Routing table
  - Long distance links to other prefix realms
  - \( l/b \) rows: one per prefix length
  - \( 2^{b-1} \) columns: one per digit different from local node ID

- Routing table for node 120:

<table>
<thead>
<tr>
<th>Digit at position ( i+1 )</th>
<th>Shared prefix length with Node-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, 3, 4, 6</td>
<td>0, 1, 2, 3, 4, 6</td>
</tr>
</tbody>
</table>

Pastry: Routing Information

- Leaf set
  - Contains numerically closest node
  - Fixed maximum size

- Neighbor set
  - Contains nearby nodes
  - Fixed maximum size

- \( \log(N) \)
Pastry Routing Algorithm

- Routing of packet with destination K at node N:
  1. Is K in Leaf Set, route packet directly to that node
  2. If not, determine common prefix (N, K)
  3. Search entry T in routing table with prefix (T, K) > prefix (N, K), and route packet to T
  4. If not possible, search node T with longest prefix (T, K) out of merged set of routing table, leaf set, and neighborhood set and route to T

- This was shown to be a rare case

- Access to routing table O(1), since row and column are known
- Entry might be empty if corresponding node is unknown

Pastry: Routing Procedure

- Long-range routing
  - if key k not covered by leaf set:
    - forward query for k to
      • node with longer prefix match than self or
      • same prefix length but numerically closer

- Close-range routing
  - k covered by nodes IDs in leaf set
  - pick leaf node nL numerically closest to k
  - nL must be responsible for k → last step in routing procedure
  - return nL as answer to query for k

Another example

Key = 01200
Common prefix:
01200

Key = 32102
Node-ID = 32101
Common prefix:
32101

Key = 32122
Node-ID = 32101
Common prefix:
32101

Another example /2

Key = 32162
Node-ID = 32101
Node is in range of Leaf-Set

Arrival of a new node

- Node X wants to join Pastry DHT
  - Determine NodeID of X
  - Initialize tables at node X
  - Send JOIN message to key 12333 via topologically nearest Pastry node
  - Node currently in charge of this key: 2

Arrival of a new node /2

- Node X wants to join Pastry DHT
  - Node X copies Neighbor-Set from node A0

Copy Neighbor-Set
Arrival of a new node /3
- Node X wants to join Pastry DHT
  - Node A0 routes message to node Z
  - Each node sends row in routing table to X
    - Here A0

Join X
X = 12333
A4 = Z = 12332
A1 = 13231
A2 = 12222
A3 = 12311
A0 = 23231

Arrival of a new node /4
- Node X wants to join Pastry DHT
  - Node A0 routes message to node Z
  - Each node sends row in routing table to X
    - Here A1

Join X
X = 12333
A4 = Z = 12332
A1 = 13231
A2 = 12222
A3 = 12311
A0 = 23231

Arrival of a new node /5
- Node X wants to join Pastry DHT
  - Node A0 routes message to node Z
  - Each node sends row in routing table to X
    - Here A2

Join X
X = 12333
A4 = Z = 12332
A1 = 13231
A2 = 12222
A3 = 12311
A0 = 23231

Arrival of a new node /6
- Node X wants to join Pastry DHT
  - Node A0 routes message to node Z
  - Each node sends row in routing table to X
    - Here A3

Join X
X = 12333
A4 = Z = 12332
A1 = 13231
A2 = 12222
A3 = 12311
A0 = 23231

Arrival of a new node /7
- Node X wants to join Pastry DHT
  - Node A0 routes message to node Z
  - Each node sends row in routing table to X
    - Here A4

Join X
X = 12333
A4 = Z = 12332
A1 = 13231
A2 = 12222
A3 = 12311
A0 = 23231

Arrival of a new node /8
- Node X wants to join Pastry DHT
  - Node A0 routes message to node Z
  - Each node sends row in routing table to X
    - Here A4

Join X
X = 12333
A4 = Z = 12332
A1 = 13231
A2 = 12222
A3 = 12311
A0 = 23231

Copy Leaf-Set to X
Arrival of a new node /9

- Some entries are doubtable
  - Entries pointing to "own-ID-positions" not required

- Some are missing
  - Take the node-IDs just visited

\[ X = 12333 \]

\[ A_4 = Z = 12332 \]

\[ A_1 = 13231 \]

\[ A_2 = 12222 \]

\[ A_3 = 12311 \]

\[ A_0 = 23231 \]

\[ 13003130011233012331 \]

\[ 13000123331232212311 \]

\[ > \text{Node-ID} < \]

\[ 12331-12330-12331 \]

\[ 12320-12301-12321 \]

\[ 12111-12033-12112 \]

\[ 13121-01221-13121 \]

\[ 32331-02231-32331 \]

\[ 11213-21021-11001 \]

\[ 00100-32123-01213 \]

\[ 12300-32022-12300 \]

\[ > \text{Row-ID} < \]

\[ 23231-13231-23231 \]

\[ 12222-12311-12222 \]

\[ 12311-12111-12311 \]

\[ 12222-11312-12222 \]

\[ 12331-02231-12331 \]

\[ > \text{Column-ID} < \]

\[ 32123-01213-32123 \]

\[ 00100-32022-00100 \]

\[ 12300-32022-12300 \]

\[ > \text{Node-ID} < \]

\[ 12333-12331-12331 \]

\[ 12332-12320-12332 \]

\[ 12311-12033-12311 \]

\[ 12222-11312-12222 \]

\[ 12331-02231-12331 \]

Failure of Pastry Nodes

- Detection of failure
  - Periodic verification of nodes in Leaf Set
    - "Are you alive" also checks capability of neighbor
  - Route query fails

- Replacement of corrupted entries
  - Leaf-Set
    - Choose alternative node from Leaf (L \cup (L \cup \text{Leaf} \setminus \text{Leaf}))
    - Ask these nodes for their Leaf Sets
  - Entry \( R_{ij} \) in routing table failed:
    - Ask neighbor node \( R_{ij} \) of same row for route to \( R_{ij} \)
    - If not successful, test entry \( R_{++ij} \) in next row

Arrival of a new node /11

- Efficiency of initialization procedure
  - Quality of routing table (b=4, l=16, 5k nodes)

\[ X = 12333 \]

\[ A_4 = Z = 12332 \]

\[ A_1 = 13231 \]

\[ A_2 = 12222 \]

\[ A_3 = 12311 \]

\[ A_0 = 23231 \]

\[ 13003130011233012331 \]

\[ 13000123331232212311 \]

\[ > \text{Node-ID} < \]

\[ 12331-12330-12331 \]

\[ 12320-12301-12321 \]

\[ 12111-12033-12112 \]

\[ 13121-01221-13121 \]

\[ 32331-02231-32331 \]

\[ 11213-21021-11001 \]

\[ 00100-32123-01213 \]

\[ 12300-32022-12300 \]

\[ > \text{Row-ID} < \]

\[ 23231-13231-23231 \]

\[ 12222-12311-12222 \]

\[ 12311-12111-12311 \]

\[ 12222-11312-12222 \]

\[ 12331-02231-12331 \]

\[ > \text{Column-ID} < \]

\[ 32123-01213-32123 \]

\[ 00100-32022-00100 \]

\[ 12300-32022-12300 \]

Performance Evaluation

- Routing Performance
  - Number of Pastry hops (b=4, l=16, 2·10^5 queries)
  - \( O(\log N) \) for number of hops in the overlay

  - Overhead of overlay (in comparison to route between two node in the IP network)
  - But: Routing table has only \( O(\log N) \) entries instead of \( O(N) \)

Failure of Pastry Nodes

- Detection of failure
  - Periodic verification of nodes in Leaf Set
    - "Are you alive" also checks capability of neighbor
  - Route query fails

- Replacement of corrupted entries
  - Leaf-Set
    - Choose alternative node from Leaf (L \cup (L \cup \text{Leaf} \setminus \text{Leaf}))
    - Ask these nodes for their Leaf Sets
  - Entry \( R_{ij} \) in routing table failed:
    - Ask neighbor node \( R_{ij} \) of same row for route to \( R_{ij} \)
    - If not successful, test entry \( R_{++ij} \) in next row

Summary Pastry

- Complexity:
  - \( O(\log h) \) hops to destination
  - Often even better through Leaf- and heighter-Set: \( O(\log P \cdot N) \)
  - \( O(\log N) \) storage overhead per node

  - Good support of locality
    - Explicit search of close nodes (following some metric)

  - Used in many applications
    - PAST (file system), Squirrel (Web-Cache), ...
    - Many publications available, open source implementation: FreePastry
Tapestry

- Tapestry developed at UC Berkeley
  - Different group from CAN developers
- Tapestry developed in 2000, but published in 2004
  - Originally only as technical report, 2004 as journal article
- Many follow-up projects on Tapestry
  - Example: OceanStore
- Like Pastry, based on work by Plaxton et al.
- Pastry was developed at Microsoft Research and Rice University
  - Difference between Pastry and Tapestry minimal
  - Tapestry and Pastry add dynamics and fault tolerance to Plaxton network

Tapestry: Routing Mesh

- (Partial) routing mesh for a single node 4227
  - Neighbors on higher levels match more digits

Tapestry: Neighbor Map for 4227

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1076</td>
<td>27AB</td>
<td>51E5</td>
<td>6F43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>43C9</td>
<td>44AF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>42A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>4228</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- There are actually 16 columns in the map (base 16)
- Normally more entries would be filled (limited by a constant)
- Tapestry has multiple neighbor maps

Tapestry: Routing Example

- Route message from 5230 to 42AD
  - Always route to node closer to target
    - At nth hop, look at (n+1)th level in neighbor map --> "always" one digit more
    - Not all nodes and links are shown

Tapestry: Properties

- Node responsible for objects which have the same ID
  - Unlikely to find such node for every object
  - Node also responsible for "nearby" objects (surrogate routing, see below)
- Object publishing
  - Responsible nodes only store pointers
    - Multiple copies of object possible
    - Each copy must publish itself
  - Pointers cached along the publish path
  - Queries routed towards responsible node
  - Queries "sift through" hit cached pointers
  - Queries for same object go (soon) to same nodes
- Note: Tapestry focuses on storing objects
  - Chord and CAN focus on values, but in practice no difference

Tapestry: Publishing Example

- Two copies of object "DOC" with ID 4377 created at AA93 and 4228
  - AA93 and 4228 publish object DOC, messages routed to 4377
  - Publish messages create location pointers on the way
  - Any subsequent query can use location pointers
**Tapestry: Querying Example**
- Requests initially route towards 4377
- When they encounter the publish path, use location pointers to find object
- Often, no need to go to responsible node
- Downside: Must keep location pointers up-to-date

**Tapestry: Making It Work**
- Previous examples show a Plaxton network
  - Requires global knowledge at creation time
  - No fault tolerance, no dynamics
- Tapestry adds fault tolerance and dynamics
  - Nodes join and leave the network
  - Nodes may crash
  - Global knowledge is impossible to achieve
- Tapestry picks closest nodes for neighbor table
  - Closest in IP network sense (= shortest RTT)
  - Network distance (usually) transitive
    - If A is close to B, then B is also close to A
  - Idea: Gives best performance

**Tapestry: Fault-Tolerant Routing**
- Tapestry keeps mesh connected with keep-alives
  - Both TCP timeouts and UDP "hello" messages
  - Requires extra state information at each node
- Neighbor table has backup neighbors
  - For each entry, Tapestry keeps 2 backup neighbors
  - If primary fails, use secondary
    - Works well for uncorrelated failures
- When node notices a failed node, it marks it as invalid
  - Most link/connection failures short-lived
  - Second chance period (e.g., day) during which failed node can come back and old route is valid again
  - If node does not come back, one backup neighbor is promoted and a new backup is chosen

**Tapestry: Fault-Tolerant Location**
- Responsible node is a single point of failure
- Solution: Assign multiple roots per object
  - Add "salt" to object name and hash as usual
  - Salt = globally constant sequence of values (e.g., 1, 2, 3, ...)
- Same idea as CAN’s multiple realities
- This process makes data more available, even if the network is partitioned
  - With s roots, availability is \( P = 1 - (1/2)^s \)
  - Depends on partition
- These two mechanisms “guarantee” fault-tolerance
  - In most cases :-)
  - Problem: If the only out-going link fails...

**Tapestry: Surrogate Routing**
- Responsible node is node with same ID as object
  - Such a node is unlikely to exist
- Solution: surrogate routing
- What happens when there is no matching entry in neighbor map for forwarding a message?
  - Node (deterministically) picks next entry in neighbor map
    - If that one also doesn’t exist, next of next ... and so on
- Idea: If "missing links" are deterministically picked, any message for that ID will end up at same node
  - This node is the surrogate
- If new nodes join, surrogate may change
  - New node is neighbor of surrogate

**Surrogate Routing Example**
- Peer 2716 searches for 4666:
  - Level 1, current digit j = 4
  - Level 2, j+6 doesn’t exist, next link: j=8
  - Level 3, j=8
  - Level 4 doesn’t have any level 4 neighbors => done
Tapestry: Performance

- Messages routed in $O(\log_b N)$ hops
  - At each step, we resolve one more digit in ID
  - $N$ is the size of the namespace (e.g., SHA-1 = 40 digits)
  - Surrogate routing adds a bit to this, but not significantly

- State required at a node is $O(b \log_b N)$
  - Tapestry has $c$ backup links per neighbor, $O(c \log_b N)$
  - Additionally, same number of backpointers

Complexity comparison of DHTs so far

<table>
<thead>
<tr>
<th></th>
<th>CAN</th>
<th>Chord</th>
<th>Pastry</th>
<th>Tapestry</th>
</tr>
</thead>
<tbody>
<tr>
<td>States per node</td>
<td>$O(d)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Pathlength (Routing)</td>
<td>$O(\sqrt{\log N})$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Join of node</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Leave of node</td>
<td>?</td>
<td>$O(\log N)$</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Kademlia

- From New York University, 2002; used in eMule, Overnet, Azureus, ...

- Routing idea similar to Plaxton’s mesh: improve closeness one bit at a time
  - Nodes and Keys are mapped to m-bit binary strings
  - Distance between two identifiers: the XOR string, as a binary number

- If $x$ and $y$ agree in the first $i$ digits and disagree in the $(i+1)$ then $2^i \leq d(x,y) \leq 2^{i+1}-1$

- Example:
  - $x = 0 1 0 1 1 0$
  - $y = 0 1 1 0 1 1$
  - $d(x,y) = 13$

- $x = 0 1 0 1 1 0$
  - $y = 0 1 1 1 0 1$
  - $d(x,y) = 8$

- $x = 0 1 0 1 1 0$
  - $y = 0 1 1 0 0 1$
  - $d(x,y) = 15$

Kademlia - Routing table

- Each node with ID $x$ stores $m$ k-buckets
  - A k-bucket stores k nodes that are at distance $[2^i,2^{i+1}-1]$ to the node
  - Default $k = 20$
  - Empty bucket if no nodes are known

- Tables (k-buckets) are updated when lookups are performed
  - Query comes from node already in k-bucket: move entry to the end
  - Query comes from new node and k-bucket not full: add node at the end
  - Query comes from new node and k-bucket full: LRU node is removed

- Due to XOR symmetry a node receives lookups from the nodes that are in its own table

- Node Joins
  - Contact a participating node and insert it in the appropriate bucket
  - Perform a query for your own ID
  - Refresh all buckets

Kademlia - Lookups

- Process is iterative:
  - Everything is controlled by the initiator node
  - Query is parallel: the $i$ nodes closest to the query ID
  - Nodes return the $i$ nodes closest to the query ID
  - Node and key lookups are done in a similar fashion

- Underlying invariant:
  - If there exists some node with ID within a specific range then k-bucket is not empty
  - If the invariant is true, then the time is logarithmic
  - If the invariant is false, then we move one bit closer each time

Kademlia vs. Chord and Pastry

- Comparing with Chord
  - Like Chord: achieves similar performance
  - Deterministic
  - (Almost) contacts routing table always
  - Lower node join/leave cost

- Comparing with Pastry
  - Both have flexible routing table
  - Kademlia has better analysis properties (simpler)