CS 268: Structured P2P Networks: Pastry and Tapestry

Sean Rhea
April 29, 2003

Domain

- Structured peer-to-peer overlay networks
  - Sometimes called DHTs, DOLRs, CASTs, …
  - Examples: CAN, Chord, Pastry, Tapestry, …
- Contrasted with unstructured P2P networks
  - Gnutella, Freenet, etc.
- Today talking about Pastry and Tapestry

Service Model

- Let I be a set of identifiers
  - Such as all 160-bit unsigned integers
- Let N be a set of nodes in a P2P system
  - Some subset of all possible (IP, port) tuples
- Structured P2P overlays implement a mapping

$$\mathit{owner}_{\mathrm{N}}: \mathrm{I} \rightarrow \mathrm{N}$$

  - Given any identifier, deterministically map to a node
- Properties
  - Should take O(log $|\mathrm{N}|$) time and state per node
  - Should be roughly load balanced

Service Model (con't.)

- Owner mapping exposed in variety of ways
  - In Chord, have function

```
n = find_successor (i)
```

  - In Pastry and Tapestry, have function

```
route_to_root (i,m)
```

- In general, can be iterative or recursive
  - Iterative = directed by querying node
  - Recursive = forwarded through network
- May also expose $\mathit{owner}^{-1}: \mathrm{N} \rightarrow \mathrm{P}(\mathrm{I})$
  - Which identifiers given node is responsible for

Other Service Models

- Other models can be implemented on owner
- Example: Distributed hash table (DHT)

```
void put (key, data) {
   n = owner (key)
   n.hash_table.insert (key, data)
}

data get (key) {
   n = owner (key)
   return n.hash_table.lookup (key)
}
```

5

Lecture Overview

- Introduction
- PRR Trees
  - Overview
  - Locality Properties
- Pastry
  - Routing in Pastry
  - Joining a Pastry network
  - Leaving a Pastry network
- Tapestry
  - Routing in Tapestry
  - Object location in Tapestry
- Multicast in PRR Trees
- Conclusions

6

PRR Trees

- Work by Plaxton, Rajaraman, Richa (SPAA '97)
  - Interesting in a distributed publication system
  - Similar to Napster in interface
  - Only for static networks (set N does not change)
  - No existing implementation (AFAIK)
- Pastry and Tapestry both based on PRR trees
  - Extend to support dynamic node membership
  - Several implementations

7

PRR Trees: The Basic Idea

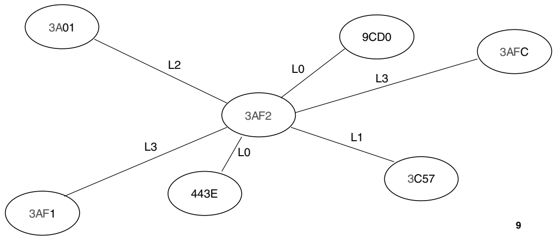
- Basic idea: add injective function

  node_id: $\mathrm{N} \rightarrow \mathrm{I}$
  - Gives each node a name in the identifier space
- *owner* (*i*) = node whose *node_id* is "closest" to *i*
  - Definition of closest varies, but always over I
- To find owner (*i*) from node with identifier *j*
  1. Let *p* = longest matching prefix between *i* and *j*
  2. Find node *k* with longest matching prefix of |*p*|+1 digits
  3. If no such node, *j* is the owner (root)
  4. Otherwise, forward query to node *k*
- Step 2 is the tricky part

8

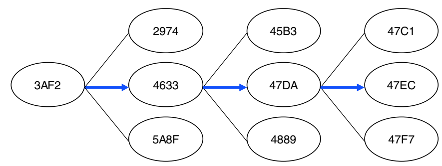
PRR Trees: The Routing Table

- Each node *n* has O(*b* log $_b$ |N|) neighbors
  - Each L*x* neighbor shares *x* digits with *n*
  - Set of neighbors forms a routing table



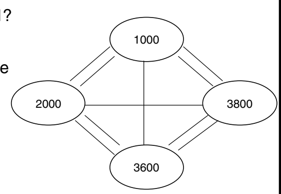
PRR Trees: Routing

- To find owner (47E2)
  - Query starts at node 3AF2
  - Resolve first digit by routing to 4633
  - Resolve second digit by routing to 47DA, etc.



PRR Trees: Routing (con't.)

- Problem: what if no exact match?
  - Consider the following network
  - Who is the owner of identifier 3701?
- Network is well formed
  - Every routing table spot that can be filled is filled
  - Can route to all node identifiers
- Owner of 3701 not well defined
  - Starting from 1000, it's node 3800
  - Starting from 2000, it's node 3600
- Violation of service model



PRR Trees: Handling Inexact Matches

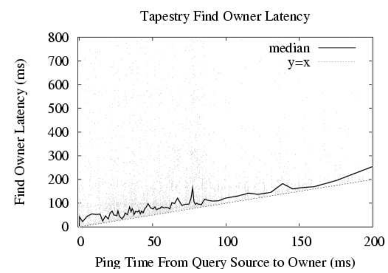
- Want owner function to be deterministic
  - Must have a way to resolve inexact matches
- Solved different ways by each system
  - I have no idea what PRR did
  - Pastry chooses numerically closest node
    - Can break ties high or low
  - Tapestry performs "surrogate routing"
    - Chooses next highest match on per digit basis
- More on this later

Locality in PRR Trees

- Consider a node with id=1000 in a PRR network
  - At lowest level of routing table, node 1000 needs neighbors with prefixes 2-, 3-, 4-, etc.
  - In a large network, there may be several of each
- Idea: chose the “best” neighbor for each prefix
  - Best can mean lowest latency, highest bandwidth, etc.
- Can show that this choice gives good routes
  - For certain networks, routing path from query source to owner no more than a constant worse than routing path in underlying network
  - I’m not going to prove this today, see PRR97 for details

13

Locality in PRR Trees: Experiments



14

Lecture Overview

- Introduction
- PRR Trees
  - Overview
  - Locality Properties
- Pastry
  - Routing in Pastry
  - Joining a Pastry network
  - Leaving a Pastry network
- Tapestry
  - Routing in Tapestry
  - Object location in Tapestry
- Multicast in PRR Trees
- Conclusions

15

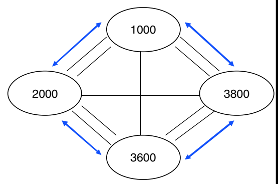
Pastry Introduction

- A PRR tree combined with a Chord-like ring
  - Each node has PRR-style neighbors
  - And each node knows its predecessor and successor
    - Called its leaf set
- To find owner (*i*), node *n* does the following:
  - If *i* is *n*’s leaf set, choose numerically closest node
  - Else, if appropriate PRR-style neighbor, choose that
  - Finally, choose numerically closest from leaf set
- A lot like Chord
  - Only leaf set necessary for correctness
  - PRR-neighbors like finger table, only for performance

16

Pastry Routing Example

- PRR neighbors in black
- Leaf set neighbors in blue
- Owner of 3701 is now well-defined
- From 1000
  - Resolve first digit routing to 3800
  - At 3800, see that we're done
  - (Numerically closer than 3600)
- From 2000
  - Resolve first digit routing to 3600
  - At 3600, 3701 is in leaf set
    - In range 2000-3800
  - Route to 3800 b/c numerically closer



Notes on Pastry Routing

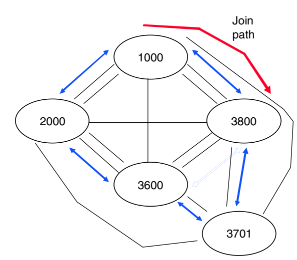
- Leaf set is great for correctness
  - Need not get PRR neighbors correct, only leaf set
  - If you believe the Chord work, this isn't too hard to do
- Leaf set also gives implementation of $owner^{-1}(n)$
  - All identifiers half-way between n and its predecessor to half-way between n and its successor
- Can store k predecessors and successors
  - Gives further robustness as in Chord

Joining a Pastry Network

- Must know of a "gateway" node, *g*
- Pick new node's identifier, *n*, U.A.R. from I
- Ask *g* to find the $m = owner(n)$
  - And ask that it record the path that it takes to do so
- Ask *m* for its leaf set
- Contact *m*'s leaf set and announce *n*'s presence
  - These nodes add *n* to their leaf sets and vice versa
- Build routing table
  - Get level *i* of routing table from node *i* in the join path
  - Use those nodes to make level *i* of our routing table

Pastry Join Example

- Node 3701 wants to join
  - Has 1000 as gateway
- Join path is 1000→3800
  - 3800 is the owner
- 3701 ties itself into leaf set
- 3701 builds routing table
  - L0 neighbors from 1000
    - 1000, 2000, and 3800
  - L1 neighbors from 3800
    - 3600
- Existing nodes on join path consider 3701 as a neighbor



Pastry Join Notes

- Join is not “perfect”
  - A node whose routing table needs new node should learn about it
    - Necessary to prove O(log |N|) routing hops
  - Also not guaranteed to find close neighbors
- Can use *routing table maintenance* to fix both
  - Periodically ask neighbors for their neighbors
  - Use to fix routing table holes; replace existing distant neighbors
- Philosophically very similar to Chord
  - Start with minimum needed for correctness (leaf set)
  - Patch up performance later (routing table)

21

Pastry Join Optimization

- Best if gateway node is “close” to joining node
  - Gateway joined earlier, should have close neighbors
  - Recursively, gateway’s neighbors’ neighbors are close
  - Join path intuitively provides good initial routing table
  - Less need to fix up with routing table maintenance
- Pastry’s optimized join algorithm
  - Before joining, find a good gateway, then join normally
- To find a good gateway, refine set of candidates
  - Start with original gateway’s leaf set
  - Keep only a closest few, then add their neighbors
  - Repeat (more or less--see paper for details)

22

Leaving a Pastry Network

- Do not distinguish between leaving and crashing
  - A good design decision, IMHO
- Remaining nodes notice leaving node *n* down
  - Stops responding to keep-alive pings
- Fix leaf sets immediately
  - Easy if 2+ predecessors and successors known
- Fix routing table lazily
  - Wait until needed for a query
  - Or until routing table maintenance
  - Arbitrary decision, IMHO

23

Dealing with Broken Leaf Sets

- What if no nodes left in leaf set due to failures?
- Can use routing table to recover (MCR93)
  - Choose closest nodes in routing table to own identifier
  - Ask them for their leaf sets
  - Choose closest of those, recurse
- Allows use of smaller leaf sets

24

Lecture Overview

- Introduction
- PRR Trees
  - Overview
  - Locality Properties
- Pastry
  - Routing in Pastry
  - Joining a Pastry network
  - Leaving a Pastry network
- Tapestry
  - Routing in Tapestry
  - Object location in Tapestry
- Multicast in PRR Trees
- Conclusions

25

Tapestry Routing

- Only different from Pastry when no exact match
  - Instead of using next numerically closer node, use node with next higher digit at each hop
- Example:
  - Given 3 node network: nodes 0700, 0F00, and FFFF
  - Who owns identifier 0000?
- In Pastry, FFFF does (numerically closest)
- In Tapestry, 0700 does
  - From FFFF to 0700 or 0F00 (doesn't matter)
  - From 0F00 to 0700 (7 is next highest digit after 0)
  - From 0700 to itself (no node with digit between 0 and 7)

26

Notes on Tapestry Routing

- Mostly same locality properties as PRR and Pastry
- But compared to Pastry, very fragile
- Consider previous example: 0700, 0F00, FFFF
  - What if 0F00 doesn't know about 0700?
  - 0F00 will think it is the owner of 0000
  - 0700 will still think it is the owner
  - Mapping won't be deterministic throughout network
- Tapestry join algorithm guarantees won't happen
  - All routing table holes than can be filled will be
  - Provably correct, but tricky to implement
  - Leaf set links are bidirectional, easier to keep consistent

27

Object Location in Tapestry

- Pastry was originally just a DHT
  - Support for multicast added later (RKC+01)
- PRR and Tapestry are DOLRs
  - Distributed Object Location and Routing
- Service model
  - `publish (name)`
  - `route_to_object (name, message)`
- Like Napster, Gnutella, and DNS
  - Service does not store data, only pointers to it
  - Manages a mapping of names to hosts

28

A Simple DOLR Implementation

- Can implement a DOLR on owner service model

```
publish (name) {
   n = owner (name)
   n.add_mapping (name, my_addr, my_port)
}

route_to_object (name, message) {
   n = owner (name)
   m = n.get_mapping (name)
   m.send_msg (message)
}
```


Problems with Simple DOLR Impl.

- No locality
  - Even if object stored nearby, owner might be far away
  - Bad for performance
  - Bad for availability (owner might be behind partition)
- No redundancy
  - Easy to fix if underlying network has leaf/successor sets
  - Just store pointers on owner's whole leaf set
    - If owner fails, replacement already has pointers
  - But Tapestry doesn't have leaf/successor sets

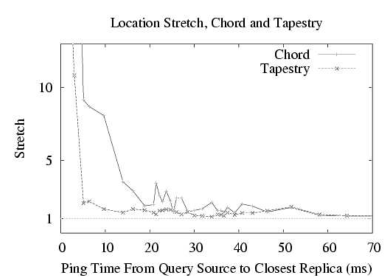
Tapestry DOLR Implementation

- Insight: leave "bread crumbs" along publish path
  - Not just at owner

```
publish (name) {
   foreach n in path_to_owner (name)
      n.add_mapping (name, my_addr, my_port)
}

route_to_object (name, message) {
   foreach n in path_to_owner (name)
      if ((m = n.get_mapping (name)) != null) {
         m.send_msg (message); break;
      }
}
```

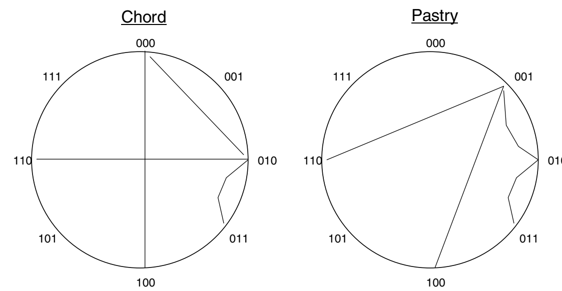

Tapestry DOLR Impl.: Experiments



Tapestry DOLR Impl. Notes

- Bread crumbs called “object pointers”
- PRR show that overlay path from query source to object is no worse than a constant longer than underlying network path
  - Just like in routing in a PRR tree
- True for two reasons:
  1. Hops early in path are short (in network latency)
  2. Paths converge early
- Path convergence is a little subtle
  - Two nearby nodes often have same early hops
  - Because next hop based on destination, not source
  - And because neighbor choice weighted on latency

Path Convergence Examples



(nodes 001, 100, and 110 are “close”)

Lecture Overview

- Introduction
- PRR Trees
  - Overview
  - Locality Properties
- Pastry
  - Routing in Pastry
  - Joining a Pastry network
  - Leaving a Pastry network
- Tapestry
  - Routing in Tapestry
  - Object location in Tapestry
- Multicast in PRR Trees
- Conclusions

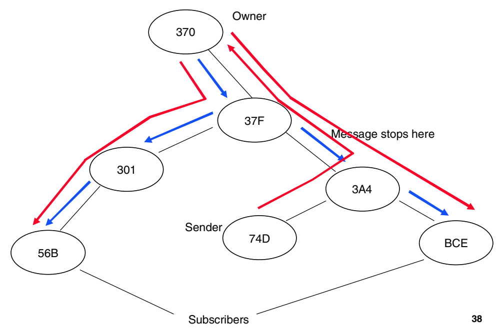
Multicast in PRR Trees

- PRR Tree gives efficient paths between all nodes
  - Uses application nodes as routers
- Since control routers, can implement multicast
  - Seems to have been thought of simultaneously by:
    - Pastry group, with SCRIBE protocol (RKC+01)
    - Tapestry group, with Bayeux protocol (ZZJ+01)
- I’ll talk about SCRIBE

SCRIBE Protocol

- Like Tapestry object location, use bread crumbs
- To subscribe to multicast group *i*
  - Walk up tree towards *owner* (*i*), leaving bread crumbs
  - If find pre-existing crumb, leave one more and stop
- To send a message to group *i*
  - Send message to *owner* (*i*)
  - Owner sends message to each bread crumb it has
    - Its children who are subscribers or their parents
  - Each child recursively does the same

SCRIBE Example



SCRIBE Notes

- Multicast evaluation points
  - Stretch, also called Relative Delay Penalty (RDP)
  - Network Stress
- SCRIBE has constant RDP
  - Assuming Pastry is a good PRR tree
- SCRIBE has low network stress
  - Harder to see, but due to choosing neighbors by latency
  - Demonstrated in simulations (CJK03)

Conclusions

- PRR trees are a powerful P2P primitive
  - Can be used as a DHT
    - Path to owner has low RDP
    - Chord can be “hacked” to do the same
  - Can be used as a DOLR
    - Finds close objects when available
    - No clear way to get this from Chord
  - Can be used for application-level multicast
    - Also no clear way to get this from Chord
- More work to support dynamic membership
  - Pastry uses leaf sets like Chord
  - Tapestry has own algorithm

For more information

- Pastry
  http://research.microsoft.com/~antr/pastry/pubs.htm
- Tapestry
  http://oceanstore.cs.berkeley.edu/publications/index.html

41