Distributed Hash Tables (DHTs)
Tapestry & Pastry

CS 699/IT 818

Sanjeev Setia

1

Acknowledgements

Some of the followings slides are borrowed or adapted from talks by Robert Morris (MIT) and Ben Zhao (UC, Santa Barbara)

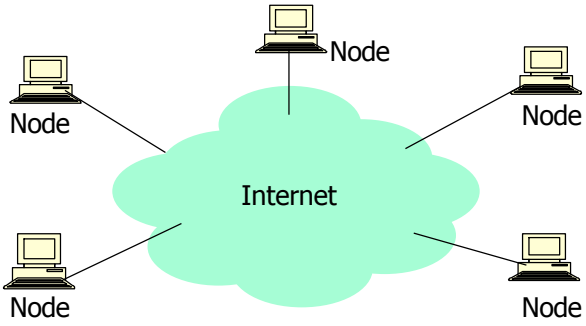
2

DHTs

- Distributed Hash Tables: a building block for P2P applications
- Today:
  - Tapestry (Zhao et al -- UC Berkeley)
  - Pastry (Rowstron et al - Microsoft Research)
- Next class
  - Chord
  - CAN
- Several other DHTs have been proposed
  - Student presentations

3

What Is a P2P System?



- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
- Large number of unreliable nodes
- Enabled by technology improvements

4

The Promise of P2P Computing

- High capacity through parallelism:
  - Many disks
  - Many network connections
  - Many CPUs
- Reliability:
  - Many replicas
  - Geographic distribution
- Automatic configuration
- Useful in public and proprietary settings

5

What Is a DHT?

- Single-node hash table:

  key = Hash(name)

  put(key, value)

  get(key) -> value
  - Service: O(1) storage
- How do I do this across millions of hosts on the Internet?
  - *Distributed* Hash Table

6

What Is a DHT?

Distributed Hash Table:

key = Hash(data)

lookup(key) -> IP address (Chord)

send-RPC(IP address, PUT, key, value)

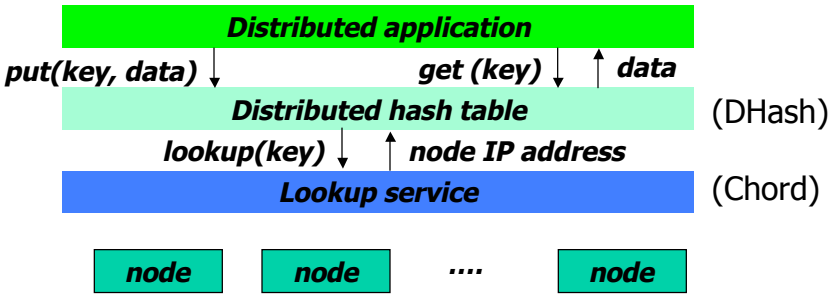
send-RPC(IP address, GET, key) -> value

Possibly a first step towards truly large-scale distributed systems

- a tuple in a global database engine
- a data block in a global file system
- rare.mp3 in a P2P file-sharing system

7

DHTs



- Application may be distributed over many nodes
- DHT distributes data storage over many nodes

8

Why the put()/get() interface?

- API supports a wide range of applications
  - DHT imposes no structure/meaning on keys
- Key/value pairs are persistent and global
  - Can store keys in other DHT values
  - And thus build complex data structures

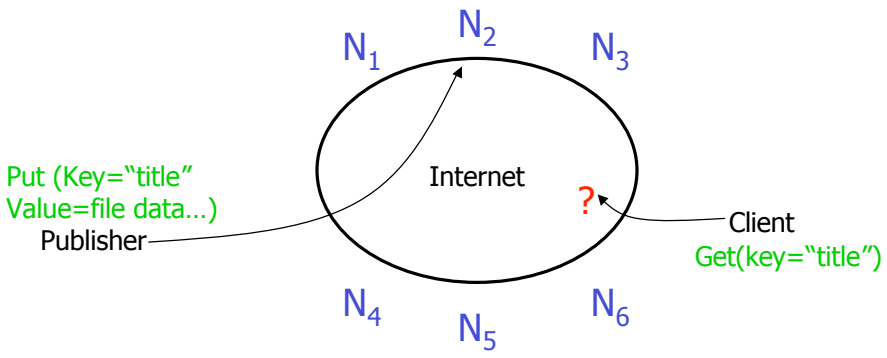
9

Why Might DHT Design Be Hard?

- Decentralized: no central authority
- Scalable: low network traffic overhead
- Efficient: find items quickly (latency)
- Dynamic: nodes fail, new nodes join
- General-purpose: flexible naming

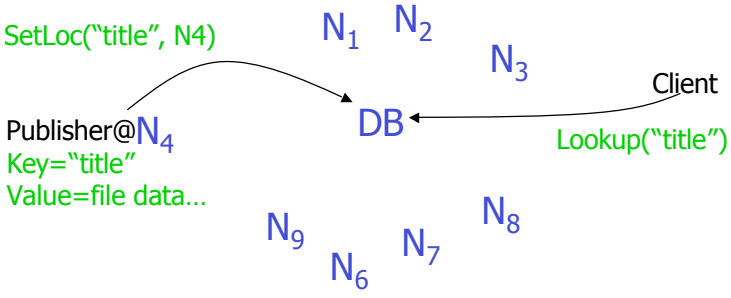
10

The Lookup Problem



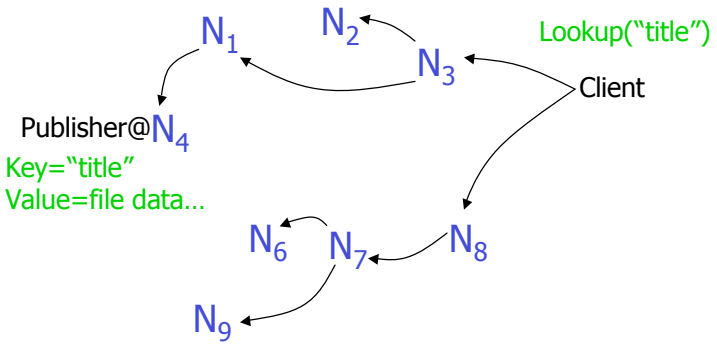
- At the heart of all DHTs

Motivation: Centralized Lookup (Napster)



Simple, but O(N) state and a single point of failure

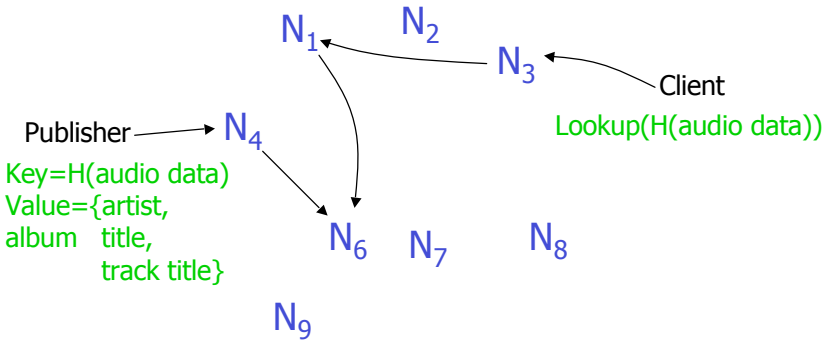
Motivation: Flooded Queries (Gnutella)



Robust, but worst case O(N) messages per lookup

13

Motivation: Routed DHT Queries (Tapestry, Pastry, Chord, CAN, etc)



14

DHT Applications

- global file systems [OceanStore, CFS, PAST, Pastiche, UsenetDHT]
- naming services [Chord-DNS, Twine, SFR]
- DB query processing [PIER, Wisc]
- Internet-scale data structures [PHT, Cone, SkipGraphs]
- communication services [i3, MCAN, Bayeux]
- event notification [Scribe, Herald]
- File sharing [OverNet]

15

Tapestry: Scalable and Fault-tolerant Routing and Location

16

What is Tapestry?

- A prototype of a *decentralized, scalable, fault-tolerant, adaptive* location and routing infrastructure
  (Zhao, Kubiatowicz, Joseph et al. U.C. Berkeley)
- Network layer of OceanStore
- Routing: Suffix-based hypercube
  - Similar to Plaxton, Rajamaran, Richa (SPAA97)
- Decentralized location:
  - Virtual hierarchy per object with cached location references
- Core API:
  - *publishObject(ObjectID, [serverID])*
  - *routeMsgToObject(ObjectID)*
  - *routeMsgToNode(NodeID)*

17

Routing and Location

- Namespace (nodes and objects)
  - 160 bits → 2^{80} names before name collision
  - Each object has its own hierarchy rooted at *Root*
    f(ObjectID) = RootID, via a dynamic mapping function
- Suffix routing from A to B
  - At h^{th} hop, arrive at nearest node hop(h) s.t.
    hop(h) shares suffix with B of length *h* digits
  - Example: 5324 routes to 0629 via
    5324 → 2349 → 1429 → 7629 → 0629
- Object location:
  - Root responsible for storing object's location
  - Publish / search both route incrementally to root

18

Publish / Lookup

- Publish object with ObjectID:

  // route towards "virtual root," ID=ObjectID

  For (i=0, i<Log_2(N), i+=j) { //Define hierarchy

  - j is # of bits in digit size, (i.e. for hex digits, j = 4)
  - Insert entry into nearest node that matches on last i bits
  - If no matches found, deterministically choose alternative
  - Found real root node, when no external routes left

- Lookup object

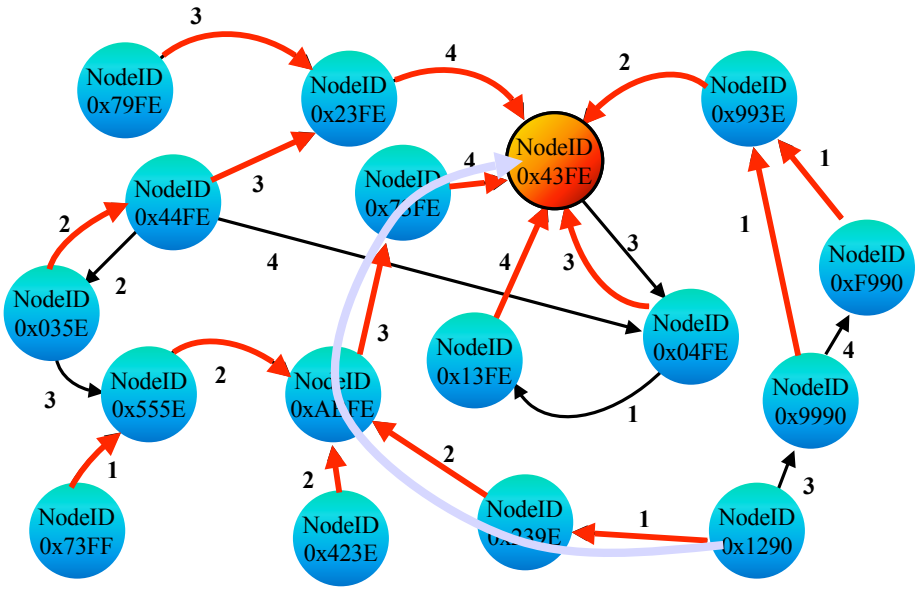
  Traverse same path to root as publish, except search for entry at each node

  For (i=0, i<Log_2(N), i+=j) {

  - Search for cached object location
  - Once found, route via IP or Tapestry to object

19

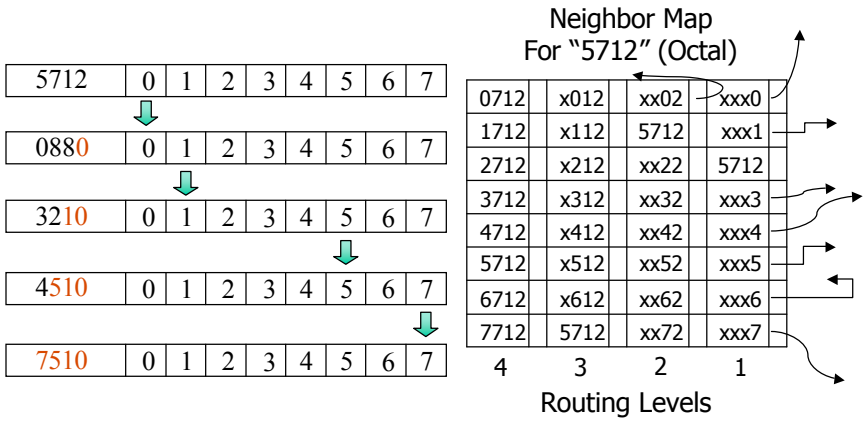
Tapestry Mesh: Incremental suffix-based routing



20

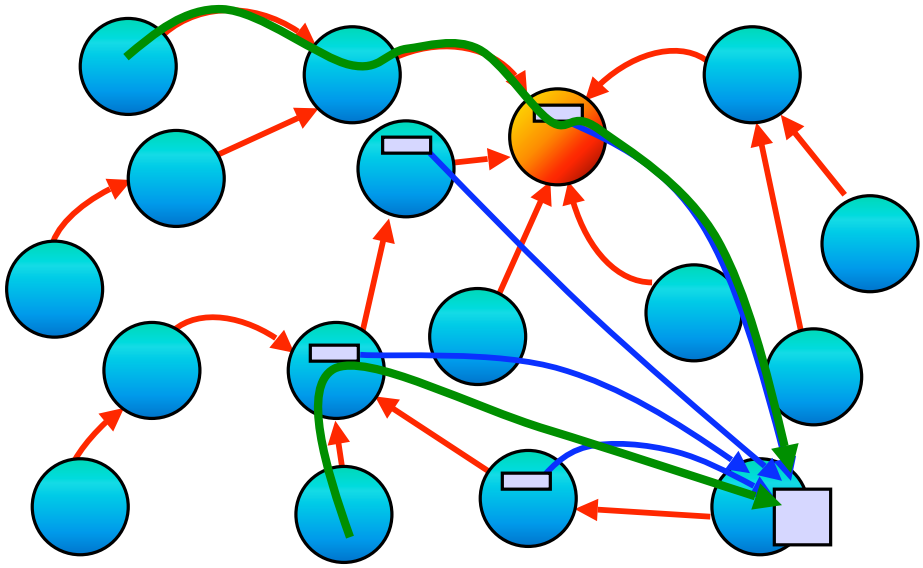
Routing in Detail

Example: Octal digits, 2^{12} namespace, 5712 → 7510



21

Object Location -- Randomization *and* Locality



22

Fault-tolerant Location

- Minimized soft-state vs. explicit fault-recovery
- Redundant roots
  - Object names hashed w/ small salts → multiple names/roots
  - Queries and publishing utilize all roots in parallel
  - P(finding reference w/ partition) = 1 – $(1/2)^n$ where n = # of roots
- Soft-state periodic republish
  - 50 million files/node, daily republish, b = 16, $N = 2^{160}$, 40B/msg, worst case update traffic: 156 kb/s,
  - expected traffic w/ 2^{40} real nodes: 39 kb/s

23

Surrogate Routing

- An object's root or surrogate node is the node which matches object id in the greatest number of trailing bits
  - Plaxton's algorithm relies on global knowledge (total ordering) to route queries when it encounters an empty neighbor entry
- Tapestry uses a deterministic algorithm to incrementally compute a unique root node
  - May involve additional hops in comparison to Plaxton's approach
  - Expected number of additional hops is 2

24

Fault-tolerant Routing

- Strategy:
  - Detect failures via soft-state probe packets
  - Route around problematic hop via backup pointers
- Handling:
  - 3 forward pointers per outgoing route (2 backups)
  - 2nd chance algorithm for intermittent failures
  - Upgrade backup pointers and replace

25

Dynamic Operation

- Node Insertion
  - Populating the Neighbor Map
    - Starting with a gateway node, route to new_id, copying a neighbor map at each hop, then optimize neighbor map
  - Neighbor Notification
    - For surrogate routing links, traverse backpointers and change entries
      - Delete object-to-location entries affected by new routing entry
    - Send hello messages to all neighbors and secondary neighbors in each level

26

Summary

- Decentralized location and routing infrastructure
  - Core routing similar to PRR97
  - Distributed algorithms for object-root mapping, node insertion / deletion
  - Fault-handling with redundancy, soft-state beacons, self-repair
  - Decentralized and scalable, with locality
- Analytical properties
  - Per node routing table size: $bLog_b(N)$
    - N = size of namespace, n = # of physical nodes
  - Find object in $Log_b(n)$ overlay hops

27

Evaluation

- Tapestry Evaluation
  - Implementation, Planetlab
  - Simulation
- Metrics
  - Routing overhead - ratio of routing delay using the overlay to the shortest IP delay
  - Success in routing a request (% successful lookups)
  - Network Bandwidth
  - Node insertion latency
- Experiments
  - Routing latency
  - Impact of caching optimizations
  - Impact of Network Dynamics
    - Single node insertion, parallel node insertion
    - Massive failures, constant churn

28

PASTRY: Scalable, decentralized object location and routing for large P2P systems

Rowstron & Druschel

29

PASTRY

- Based on Plaxton mesh like Tapestry with some differences
- Prefix routing with one digit resolved at each step
  - NodeIds and fileIds are sequences of digits with base 2^b
  - $O(\log_{2^b} N)$ steps
  - Routing table
    - $\log_{2^b} N$ levels each with 2^b-1 entries
    - Leaf set – L numerically closest nodeids (L/2 larger, L/2 smaller)
    - Neighborhood set – L closest nodes based on proximity metric - (this set is valuable in obtaining routing info from nodes that are closeby)

30

PASTRY routing

NodeId 10233102 b = 2, L = 8

Leaf Set	10233033	10233021	10233120	10233122
	10233001	10233000	10233230	10233232
Routing Table	-0-2212102	1	-2-2302332	-3-1220110
	0	1-1-301220	1-2-230010	1-3-210022
	10-0-21022	10-1-32102	2	10-3-21011
	102-0-2101	102-1-2100	102-2-1101	3
	1023-0-110	1023-1-101	1023-2-100	3
	10233-0-11	1	10233-2-01	10233-3-01
	0		102331-2-0	
			2	
Neighborhood Set	13021022	10203001	11312201	31202001
	02201200	22301110	12231101	10203102

31

Pastry routing

- Prefix routing similar to Plaxton, Tapestry
- Differences
  - If key falls within range of leaf set, message forwarded directly to node in leaf set closest to key
  - If routing entry empty or if associated node is unreachable, then forward to a node in leaf set (or neighborhood set) that shares a prefix with the key at least as long as the local node, and whose id is numerically closer to the key than the local node's id.

32

Pastry cont'd

- Node addition
  - Similar to Tapestry, I.e. node routes a message to itself resulting in message being routed to current surrogate (Z)
    - Contacts a nearby node A, asking it to route a message to itself
    - Obtains i-th row of its routing map from i-th node encountered from A to Z
    - Obtains neighborhood set from A
    - Obtains leaf set from Z
- Node deletion
  - Nodes exchange keep-alive messages with nodes in their leaf set, if failure detected in leaf set, obtain leaf sets from a live node in the leaf set with largest index (on the side of the failed node); select a node from this leaf set
  - To repair failed routing entries, get corresponding entry from another node in the same row of routing table; if that fails, try another node in the next row, and so on…

33

Discussion Items

- Differences between Tapestry & Pastry
  - Routing algorithm - surrogate routing vs leaf set
  - Approaches for exploiting locality
- DHTs vs unstructured P2P systems
  - Do you typically know the exact name of the item or file you're looking for?
  - How would DHTs support keyword search?
- How good are the locality heuristics used in Tapestry/Pastry?
- No notion of hierarchy for DHTs
  - Is this good?

34