EECS 262a
Advanced Topics in Computer Systems
Lecture 21

Chord/Tapestry
April 11th, 2016

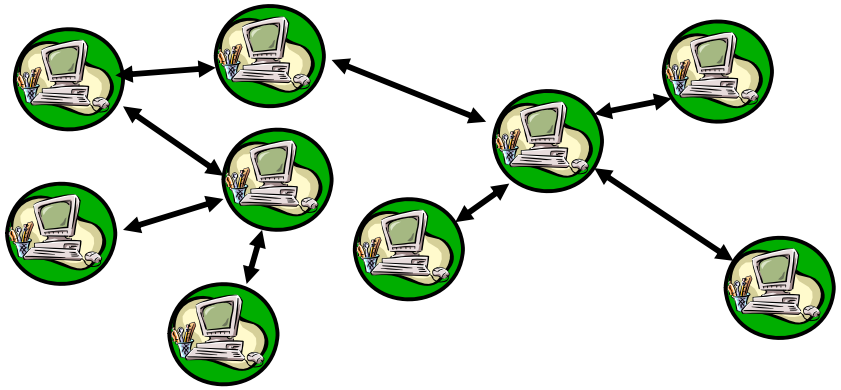
John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

- Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications, Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, Appears in *Proceedings of the IEEE/ACM Transactions on Networking*, Vol. 11, No. 1, pp. 17-32, February 2003
- Tapestry: A Resilient Global-scale Overlay for Service Deployment, Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Appears in *IEEE Journal on Selected Areas in Communications*, Vol 22, No. 1, January 2004
- Today: Peer-to-Peer Networks
- Thoughts?

Peer-to-Peer: Fully equivalent components



- Peer-to-Peer has many interacting components
  - View system as a set of equivalent nodes
    - » "All nodes are created equal"
  - Any structure on system must be self-organizing
    - » Not based on physical characteristics, location, or ownership

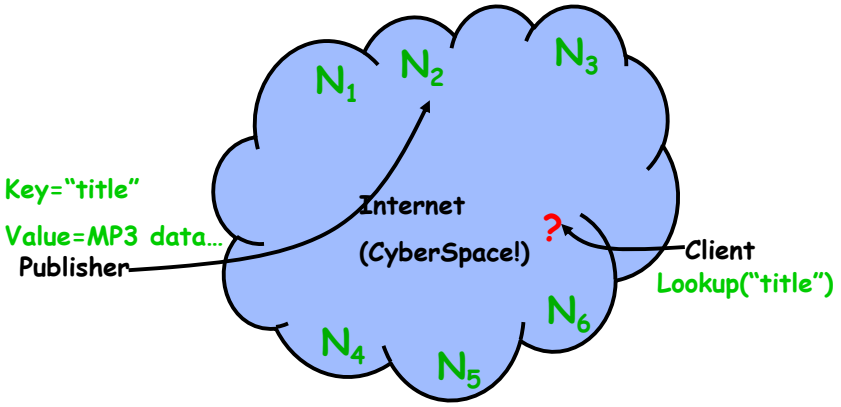
Research Community View of Peer-to-Peer

- Old View:
  - A bunch of flakey high-school students stealing music
- New View:
  - A philosophy of systems design at extreme scale
  - Probabilistic design when it is appropriate
  - New techniques aimed at unreliable components
  - A rethinking (and recasting) of distributed algorithms
  - Use of Physical, Biological, and Game-Theoretic techniques to achieve guarantees

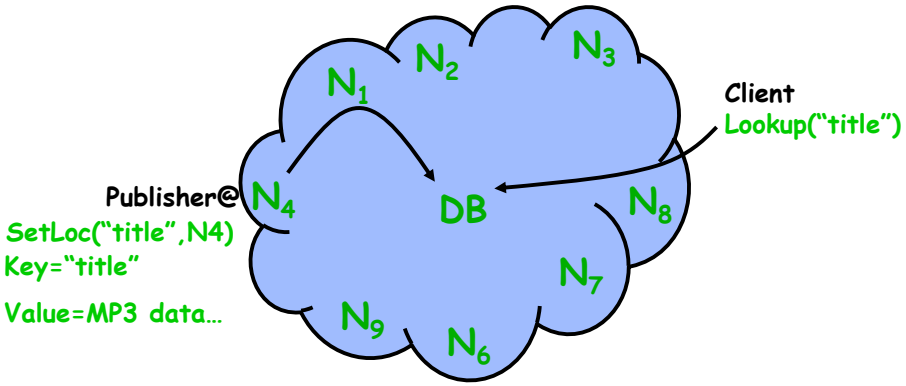
Early 2000: Why the hype???

- File Sharing: Napster (+Gnutella, KaZaa, etc)
  - Is this peer-to-peer? Hard to say.
  - Suddenly people could contribute to active global network
    - » High coolness factor
  - Served a high-demand niche: online jukebox
- Anonymity/Privacy/Anarchy: FreeNet, Publis, etc
  - Libertarian dream of freedom from the man
    - » (ISPs? Other 3-letter agencies)
  - Extremely valid concern of Censorship/Privacy
  - In search of copyright violators, RIAA challenging rights to privacy
- Computing: The Grid
  - Scavenge numerous free cycles of the world to do work
  - Seti@Home most visible version of this
- Management: Businesses
  - Businesses have discovered extreme distributed computing
  - Does P2P mean "self-configuring" from equivalent resources?
  - Bound up in "Autonomic Computing Initiative"?

The lookup problem

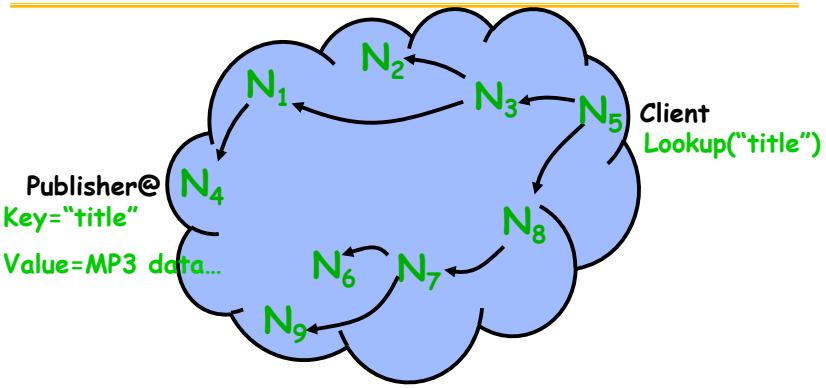


Centralized lookup (Napster)



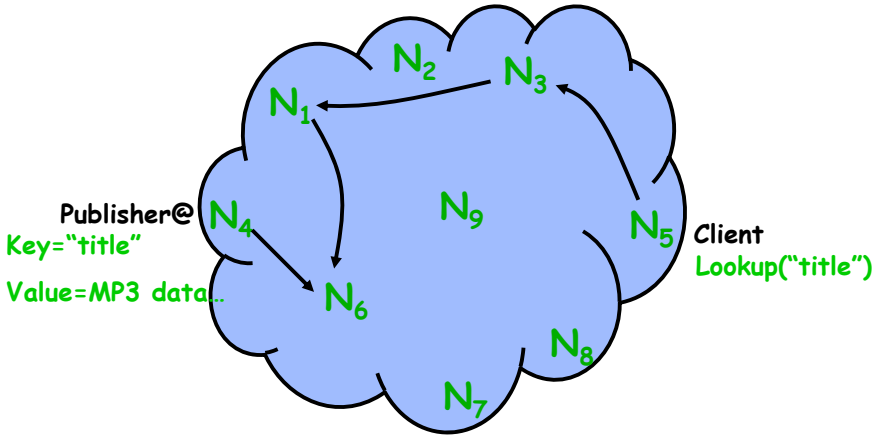
Simple, but $O(N)$ state and a single point of failure

Flooded queries (Gnutella)



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

Routed queries (Freenet, Chord, Tapestry, etc.)



Can be O(log N) messages per lookup (or even O(1))

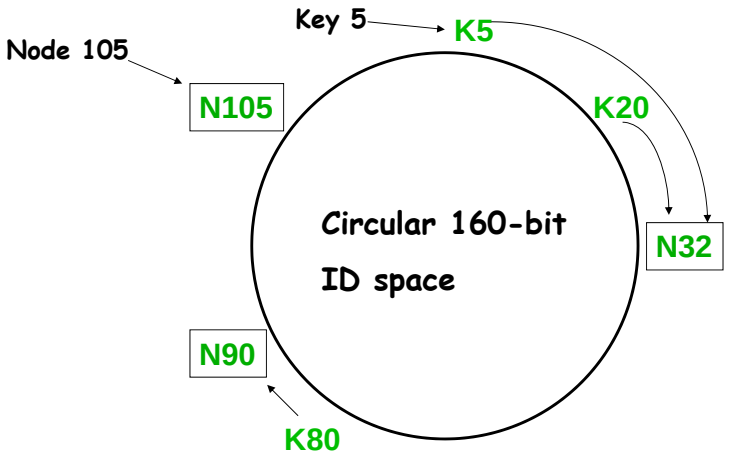
Potentially complex routing state and maintenance.

Chord IDs

- Key identifier = 160-bit SHA-1(key)
- Node identifier = 160-bit SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space

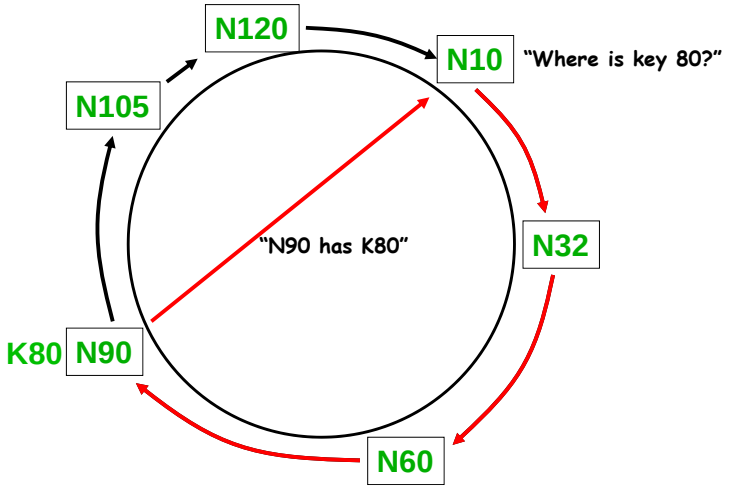
- How to map key IDs to node IDs?

Consistent hashing [Karger 97]



A key is stored at its successor: node with next higher ID

Basic lookup

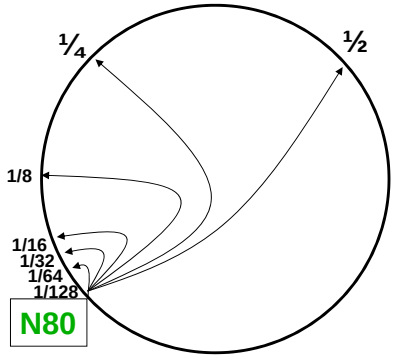


Simple lookup algorithm

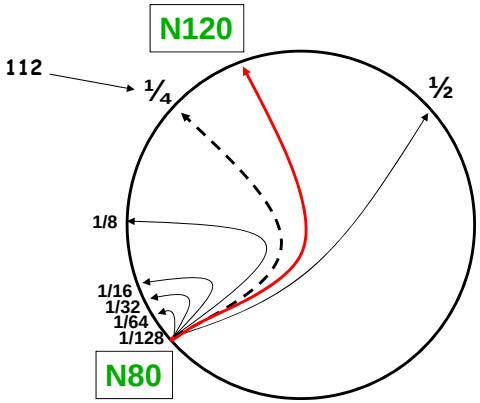
```
Lookup(my-id, key-id)
  n = my successor
  if my-id < n < key-id
      call Lookup(id) on node n   // next hop
  else
      return my successor         // done
```

- Correctness depends only on successors

"Finger table" allows log(N)-time lookups



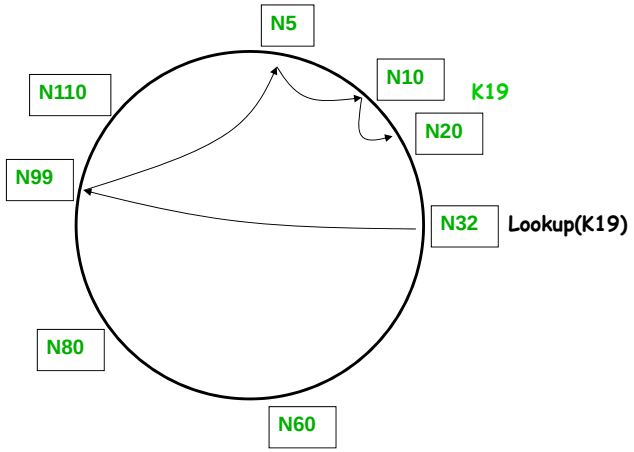
Finger *i* points to successor of $n+2^i$



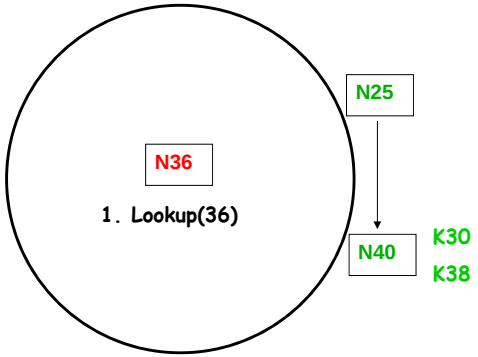
Lookup with fingers

```
Lookup(my-id, key-id)
  look in local finger table for
      highest node n s.t. my-id < n < key-id
  if n exists
      call Lookup(id) on node n   // next hop
  else
      return my successor         // done
```

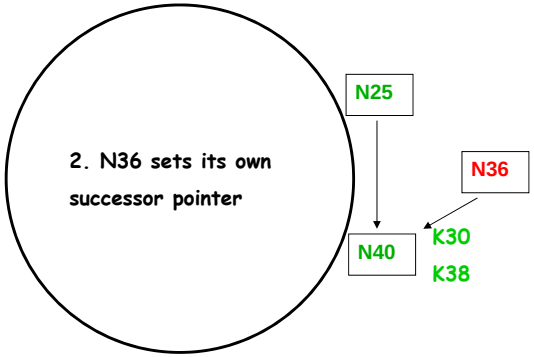
Lookups take O(*log(N)*) hops



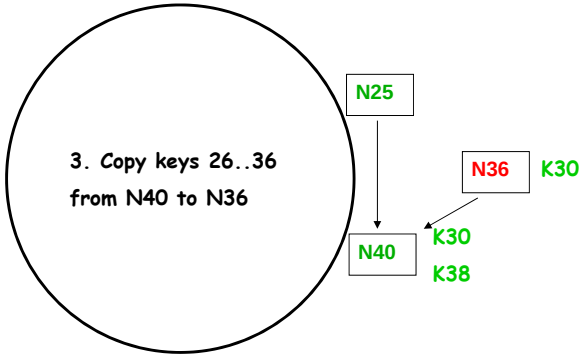
Joining: linked list insert



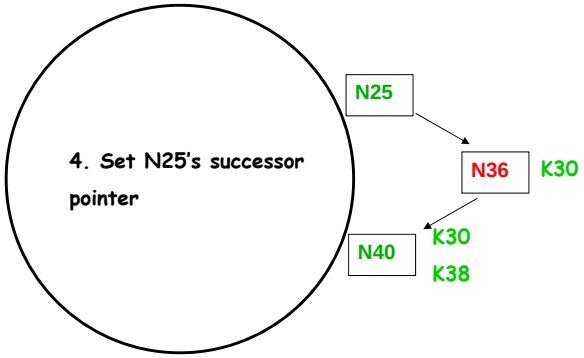
Join (2)



Join (3)



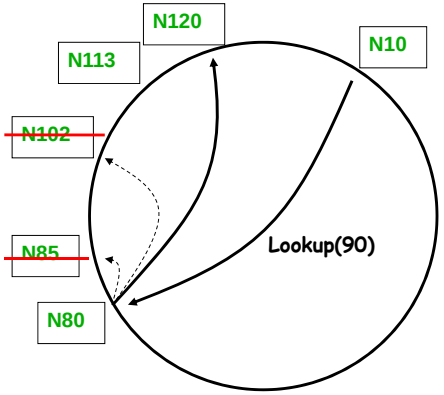
Join (4)



Update finger pointers in the background

Correct successors produce correct lookups

Failures might cause incorrect lookup



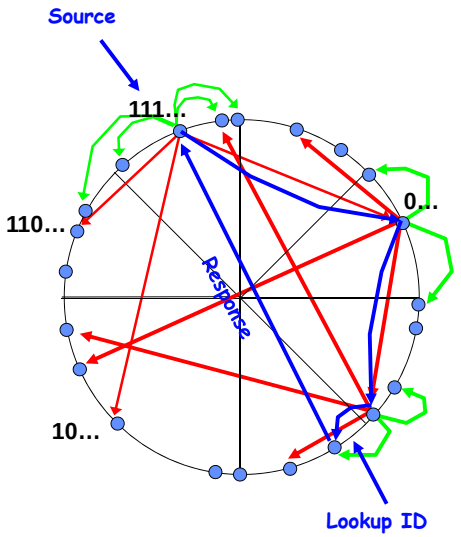
N80 doesn't know correct successor, so incorrect lookup

Solution: successor lists

- Each node knows *r* immediate successors
  - After failure, will know first live successor
  - Correct successors guarantee correct lookups
  - Guarantee is with some probability
- For many systems, talk about "leaf set"
  - The leaf set is a set of nodes around the "root" node that can handle all of the data/queries that the root nodes might handle
- When node fails:
  - Leaf set can handle queries for dead node
  - Leaf set queried to retreat missing data
  - Leaf set used to reconstruct new leaf set

Lookup with Leaf Set

- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups



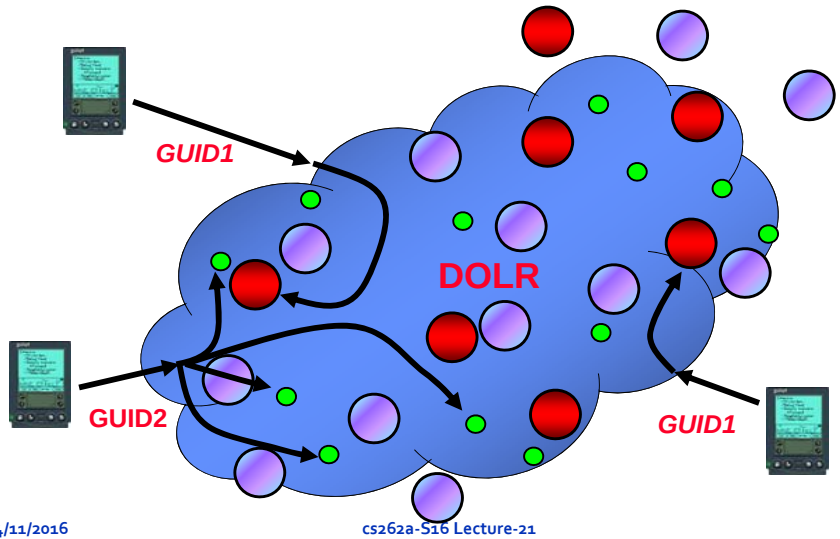
Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Decentralized Object Location and Routing: (DOLR)

- The core of Tapestry
- Routes messages to endpoints
  - Both Nodes and Objects
- Virtualizes resources
  - objects are known by name, not location

Routing to Data, not endpoints! Decentralized Object Location and Routing



DOLR Identifiers

- ID Space for both nodes and endpoints (objects) : 160-bit values with a globally defined radix (e.g. hexadecimal to give 40-digit IDs)
- Each node is randomly assigned a nodeID
- Each endpoint is assigned a *Globally Unique IDentifier* (GUID) from the same ID space
- Typically done using SHA-1
- Applications can also have IDs (application specific), which are used to select an appropriate process on each node for delivery

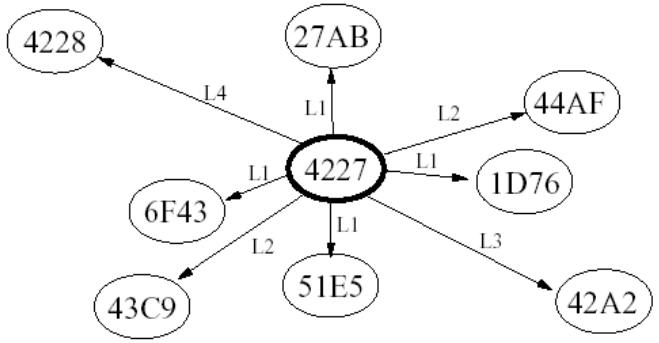
DOLR API

- PublishObject(O_G, A_{id})
- UnpublishObject(O_G, A_{id})
- RouteToObject(O_G, A_{id})
- RouteToNode(N, A_{id}, Exact)

Node State

- Each node stores a neighbor map similar to Pastry
  - Each level stores neighbors that match a prefix up to a certain position in the ID
  - Invariant: If there is a hole in the routing table, there is no such node in the network
- For redundancy, backup neighbor links are stored
  - Currently 2
- Each node also stores backpointers that point to nodes that point to it
- Creates a *routing mesh* of neighbors

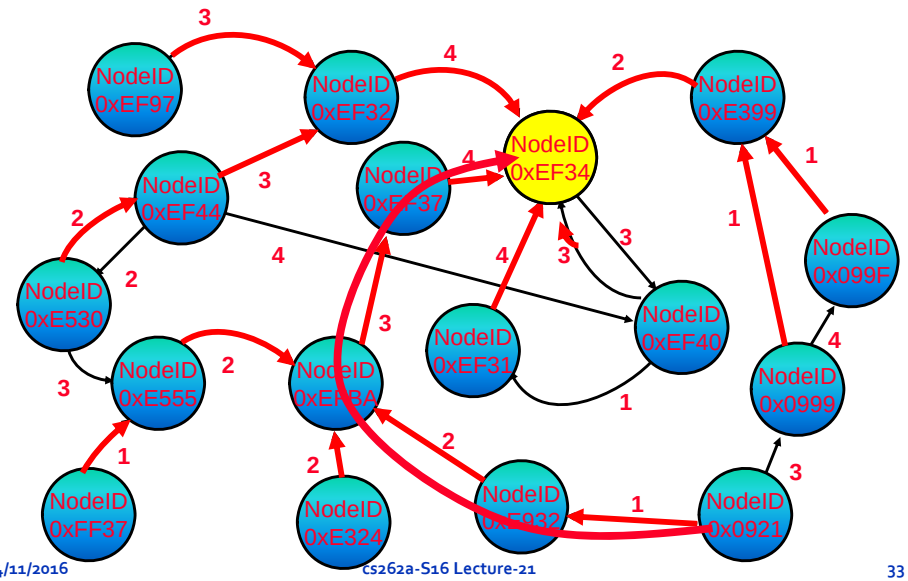
Routing Mesh



Routing

- Every ID is mapped to a *root*
- An ID's root is either the node where nodeID = ID or the "closest" node to which that ID routes
- Uses prefix routing (like Pastry)
  - Lookup for 42AD: 4*** => 42** => 42A* => 42AD
- If there is an empty neighbor entry, then use *surrogate routing*
  - Route to the next highest (if no entry for 42**, try 43**)

Basic Tapestry Mesh
Incremental Prefix-based Routing



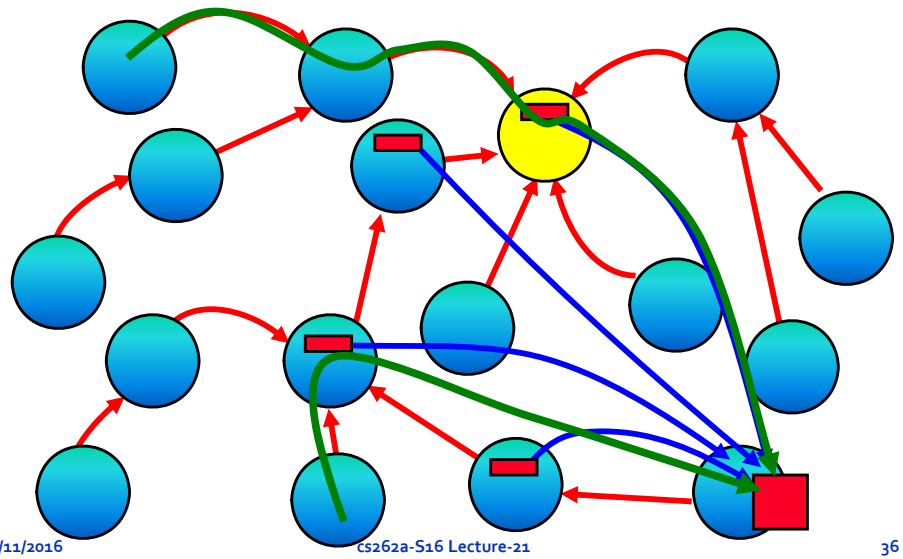
Object Publication

- A node sends a publish message towards the root of the object
- At each hop, nodes store pointers to the source node
  - Data remains at source. Exploit locality without replication (such as in Pastry, Freenet)
  - With replicas, the pointers are stored in sorted order of network latency
- Soft State – must periodically republish

Object Location

- Client sends message towards object's root
- Each hop checks its list of pointers
  - If there is a match, the message is forwarded directly to the object's location
  - Else, the message is routed towards the object's root
- Because pointers are sorted by proximity, each object lookup is directed to the closest copy of the data

Use of Mesh for Object Location



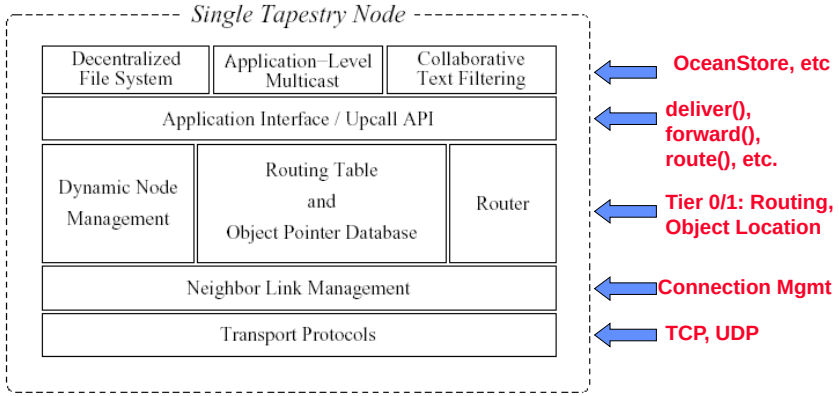
Node Insertions

- A insertion for new node N must accomplish the following:
  - All nodes that have null entries for N need to be alerted of N's presence
    - » *Acknowledged muliticast* from the "root" node of N's ID to visit all nodes with the common prefix
  - N may become the new root for some objects. Move those pointers during the muliticast
  - N must build its routing table
    - » All nodes contacted during muliticast contact N and become its neighbor set
    - » Iterative nearest neighbor search based on neighbor set
  - Nodes near N might want to use N in their routing tables as an optimization
    - » Also done during iterative search

Node Deletions

- Voluntary
  - Backpointer nodes are notified, which fix their routing tables and republish objects
- Involuntary
  - Periodic heartbeats: detection of failed link initiates mesh repair (to clean up routing tables)
  - Soft state publishing: object pointers go away if not republished (to clean up object pointers)
- Discussion Point: Node insertions/deletions + heartbeats + soft state republishing = network overhead. Is it acceptable? What are the tradeoffs?

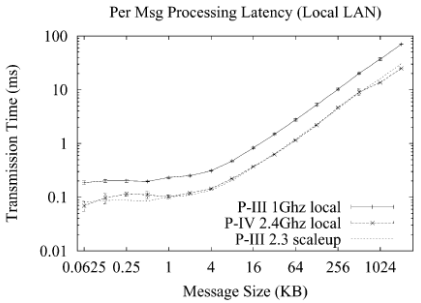
Tapestry Architecture



- **Prototype implemented using Java**

Experimental Results (I)

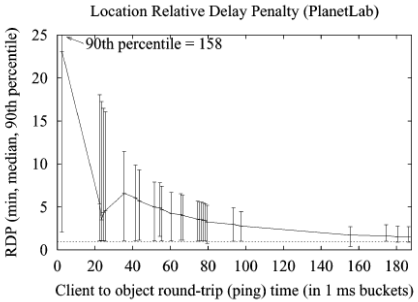
- 3 environments
  - Local cluster, PlanetLab, Simulator
- Micro-benchmarks on local cluster
  - Message processing overhead
    - » Proportional to processor speed - Can utilize Moore's Law
  - Message throughput
    - » Optimal size is 4KB

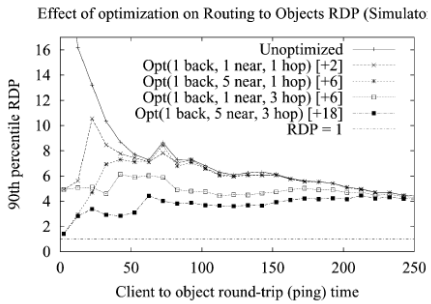


Experimental Results (II)

- Routing/Object location tests
  - Routing overhead (PlanetLab)
    - » About twice as long to route through overlay vs IP
  - Object location/optimization (PlanetLab/Simulator)
    - » Object pointers significantly help routing to close objects
- Network Dynamics
  - Node insertion overhead (PlanetLab)
    - » Sublinear latency to stabilization
    - » O(LogN) bandwidth consumption
  - Node failures, joins, churn (PlanetLab/Simulator)
    - » Brief dip in lookup success rate followed by quick return to near 100% success rate
    - » Churn lookup rate near 100%

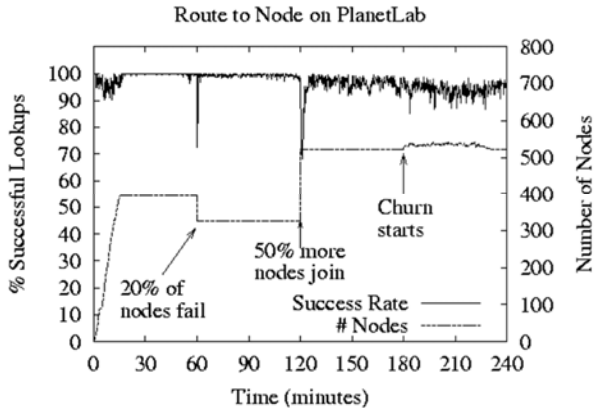
Object Location with Tapestry





- RDP (Relative Delay Penalty)
  - Under 2 in the wide area
  - More trouble in local area – (why?)
- Optimizations:
  - More pointers (in neighbors, etc)
  - Detect wide-area links and make sure that pointers on exit nodes to wide area

Stability under extreme circumstances



(May 2003: 1.5 TB over 4 hours)

DOLR Model generalizes to many simultaneous apps

Possibilities for DOLR?

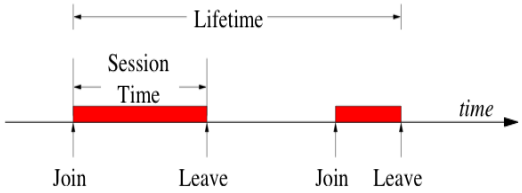
- Original Tapestry
  - Could be used to route to data or endpoints with locality (not routing to IP addresses)
  - Self adapting to changes in underlying system
- Pastry
  - Similarities to Tapestry, now in n^{th} generation release
  - Need to build locality layer for true DOLR
- Bamboo
  - Similar to Pastry – very stable under churn
- Other peer-to-peer options
  - Coral: nice stable system with course-grained locality
  - Chord: very simple system with locality optimizations

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Final topic: Churn (Optional Bamboo paper)

Chord is a "scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures"
-- Stoica et al., 2001



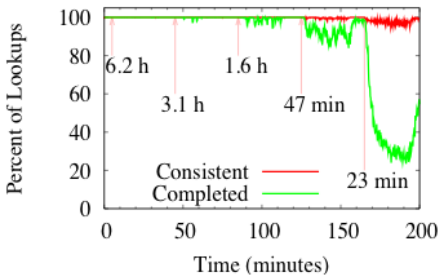
Authors	Systems Observed	Session Time
SGG02	Gnutella, Napster	50% < 60 minutes
CLL02	Gnutella, Napster	31% < 10 minutes
SW02	FastTrack	50% < 1 minute
BSV03	Overnet	50% < 60 minutes
GDS03	Kazaa	50% < 2.4 minutes

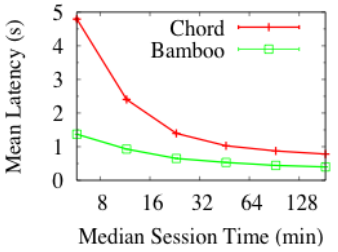
A Simple *lookup* Test

- Start up 1,000 DHT nodes on ModelNet network
  - Emulates a 10,000-node, AS-level topology
  - Unlike simulations, models cross traffic and packet loss
  - Unlike PlanetLab, gives reproducible results
- Churn nodes at some rate
  - Poisson arrival of new nodes
  - Random node departs on every new arrival
  - Exponentially distributed session times
- Each node does 1 lookup every 10 seconds
  - Log results, process them after test

Early Test Results

- Tapestry had trouble under this level of stress
  - Worked great in simulations, but not as well on more realistic network
  - Despite sharing almost all code between the two!
- Problem was not limited to Tapestry consider Chord:





Handling Churn in a DHT

- Forget about comparing different impls.
  - Too many differing factors
  - Hard to isolate effects of any one feature
- Implement all relevant features in one DHT
  - Using Bamboo (similar to Pastry)
- Isolate important issues in handling churn
  1. Recovering from failures
  2. Routing around suspected failures
  3. Proximity neighbor selection

Reactive Recovery: The obvious technique

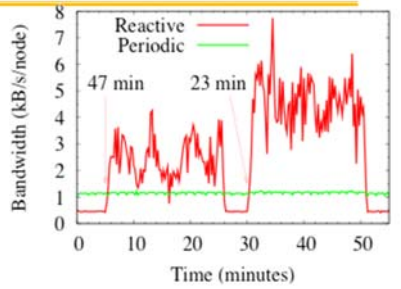
- For correctness, maintain leaf set during churn
  - Also routing table, but not needed for correctness
- The Basics
  - Ping new nodes before adding them
  - Periodically ping neighbors
  - Remove nodes that don't respond
- Simple algorithm
  - After every change in leaf set, send to all neighbors
  - Called *reactive* recovery

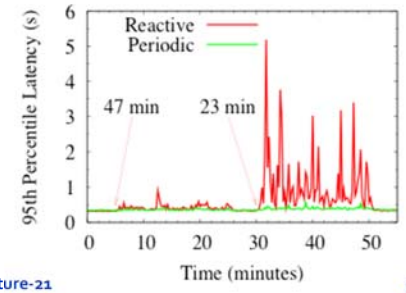
The Problem With Reactive Recovery

- Under churn, many pings and change messages
  - If bandwidth limited, interfere with each other
  - Lots of dropped pings looks like a failure
- Respond to failure by sending more messages
  - Probability of drop goes up
  - We have a positive feedback cycle (squelch)
- Can break cycle two ways
  1. Limit probability of "false suspicions of failure"
  2. Recovery periodically

Periodic Recovery

- Periodically send whole leaf set to a random member
  - Breaks feedback loop
  - Converges in O(log N)
- Back off period on message loss
  - Makes a negative feedback cycle (damping)





Conclusions/Recommendations

- Avoid positive feedback cycles in recovery
  - Beware of "false suspicions of failure"
  - Recover periodically rather than reactively
- Route around potential failures early
  - Don't wait to conclude definite failure
  - TCP-style timeouts quickest for recursive routing
  - Virtual-coordinate-based timeouts not prohibitive
- PNS can be cheap and effective
  - Only need simple random sampling