Peer-to-peer systems and Distributed Hash Tables (DHTs)

COS 461: Computer Networks
Spring 2009 (MW 1:30-2:50 in COS 105)

Mike Freedman
Teaching Assistants: Wyatt Lloyd and Jeff Terrace
http://www.cs.princeton.edu/courses/archive/spring09/cos461/
Overlay Networks

• P2P applications need to:
  – Track identities & (IP) addresses of peers
    • May be many and may have significant churn
    • Best not to have $n^2$ ID references
      – Thus, nodes’ “views” << view in consistent hashing
  – Route messages among peers
    • If you don’t keep track of all peers, this is “multi-hop”

• Overlay network
  – Peers doing both naming and routing
  – IP becomes “just” the low-level transport
    • All the IP routing is opaque
    • Assumption that network is fully-connected (← true?)

(Many slides borrowed from Joe Hellerstein’s VLDB keynote)
Many New Challenges

• Relative to other parallel/distributed systems
  – Partial failure
  – Churn
  – Few guarantees on transport, storage, etc.
  – Huge optimization space
  – Network bottlenecks & other resource constraints
  – No administrative organizations
  – Trust issues: security, privacy, incentives

• Relative to IP networking
  – Much higher function, more flexible
  – Much less controllable/predictable
Early P2P
Early P2P I: Client-Server

- Napster
Early P2P I: Client-Server

- Napster
  - Client-search search

xyz.mp3
Early P2P I: Client-Server

• Napster
  – Client-search search

xyz.mp3

xyz.mp3?
Early P2P I: Client-Server

• Napster
  – Client-search search
  – “P2P” file xfer
Early P2P I: Client-Server

- Napster
  - Client-search search
  - “P2P” file xfer

xyz.mp3

xyz.mp3?
Early P2P II: Flooding on Overlays

An “unstructured” overlay network

xyz.mp3

xyz.mp3
Early P2P II: Flooding on Overlays

Flooding
Early P2P II: Flooding on Overlays
Early P2P II: Flooding on Overlays

xyz.mp3
Early P2P II.v: “Ultra/super peers”

• Ultra-peers can be installed (KaZaA) or self-promoted (Gnutella)
  – Also useful for NAT circumvention, e.g., in Skype
Hierarchical Networks (& Queries)

• **IP**
  – Hierarchical name space ([www.vldb.org](http://www.vldb.org), 141.12.12.51)
  – Hierarchical routing: AS’s corr. with name space (not perfectly)

• **DNS**
  – Hierarchical name space (“clients” + hierarchy of servers)
  – Hierarchical routing w/aggressive caching

• **Traditional pros/cons of hierarchical mgmt**
  – Works well for things aligned with the hierarchy
    • E.g., physical or administrative locality
  – Inflexible
    • No data independence!
Lessons and Limitations

- Client-Server performs well
  - But not always feasible: Performance not often key issue!
- Things that flood-based systems do well
  - Organic scaling
  - Decentralization of visibility and liability
  - Finding popular stuff
  - Fancy *local* queries
- Things that flood-based systems do poorly
  - Finding unpopular stuff
  - Fancy *distributed* queries
  - Vulnerabilities: data poisoning, tracking, etc.
  - Guarantees about anything (answer quality, privacy, etc.)
Structured Overlays:
Distributed Hash Tables
DHT Outline

• High-level overview
• Fundamentals of structured network topologies
  – And examples
• One concrete DHT
  – Chord
• Some systems issues
  – Heterogeneity
  – Storage models & soft state
  – Locality
  – Churn management
  – Underlay network issues
High-Level Idea: Indirection

• Indirection in space
  – Logical (content-based) IDs, routing to those IDs
    • “Content-addressable” network
  – Tolerant of *churn*
    • nodes joining and leaving the network
High-Level Idea: Indirection

- Indirection in space
  - Logical (content-based) IDs, routing to those IDs
    - “Content-addressable” network
  - Tolerant of *churn*
    - nodes joining and leaving the network

- Indirection in time
  - Temporally decouple send and receive
  - Persistence required. Hence, typical sol’n: soft state
    - Combo of persistence via *storage* and via *retry*
      - “Publisher” requests TTL on storage
      - Republishes as needed

- Metaphor: Distributed Hash Table
What is a DHT?

• Hash Table
  – Data structure that maps “keys” to “values”
  – Essential building block in software systems

• Distributed Hash Table (DHT)
  – Similar, but spread across the Internet

• Interface
  – insert (key, value) or put (key, value)
  – lookup (key) or get (key)
How?

Every DHT node supports a single operation:

– Given *key* as input; route messages toward node holding *key*
DHT in action
DHT in action
DHT in action

Operation: take *key* as input; route msgs to node holding *key*
DHT in action: put()

put($K_1,V_1$)

Operation: take *key* as input; route msgs to node holding *key*
DHT in action: put()

Operation: take *key* as input; route msgs to node holding *key*
DHT in action: put()

Operation: take key as input; route msgs to node holding key
DHT in action: get()

Operation: take key as input; route msgs to node holding key
Iterative vs. Recursive Routing

Previously showed recursive. Another option: iterative

Operation: take key as input; route msgs to node holding key
DHT Design Goals

• An “overlay” network with:
  – Flexible mapping of keys to physical nodes
  – Small network diameter
  – Small degree (fanout)
  – Local routing decisions
  – Robustness to churn
  – Routing flexibility
  – Decent locality (low “stretch”)

• Different “storage” mechanisms considered:
  – Persistence w/ additional mechanisms for fault recovery
  – Best effort caching and maintenance via soft state
DHT Outline

- High-level overview
- Fundamentals of structured network topologies
  - And examples
- One concrete DHT
  - Chord
- Some systems issues
  - Heterogeneity
  - Storage models & soft state
  - Locality
  - Churn management
  - Underlay network issues
An Example DHT: Chord

- Assume $n = 2^m$ nodes for a moment
  - A “complete” Chord ring
  - We’ll generalize shortly
An Example DHT: Chord

• Each node has particular view of network
  – Set of known neighbors
An Example DHT: Chord

- Each node has particular view of network
  - Set of known neighbors
An Example DHT: Chord

• Each node has particular view of network
  – Set of known neighbors
Cayley Graphs

• The *Cayley Graph* \((S, E)\) of a group:
  – Vertices corresponding to the underlying set \(S\)
  – Edges corresponding to the *actions of the generators*

• (Complete) Chord is a Cayley graph for \((\mathbb{Z}_n, +)\)
  – \(S = \mathbb{Z} \mod n\) (\(n = 2^k\)).
  – Generators \(\{1, 2, 4, \ldots, 2^{k-1}\}\)
  – That’s what the polygons are all about!

• Fact: Most (complete) DHTs are Cayley graphs
  – And they didn’t even know it!
  – Follows from parallel InterConnect Networks (ICNs)
How Hairy met Cayley

• What do you want in a structured network?
  – Uniformity of routing logic
  – Efficiency/load-balance of routing and maintenance
  – Generality at different scales

• Theorem: All Cayley graphs are *vertex symmetric*.
  – I.e. isomorphic under swaps of nodes
  – So routing from \( y \) to \( x \) looks just like routing from \( (y-x) \) to 0
    • The routing code at each node is the same
    • Moreover, under a random workload the routing responsibilities (congestion) at each node are the same!

• Cayley graphs tend to have good degree/diameter tradeoffs
  – Efficient routing with few neighbors to maintain

• Many Cayley graphs are *hierarchical*
  – Made of smaller Cayley graphs connected by a new generator
    • E.g. a Chord graph on \( 2^{m+1} \) nodes looks like 2 interleaved (half-notch rotated) Chord graphs of \( 2^m \) nodes with half-notch edges
Pastry/Bamboo

- Based on Plaxton Mesh
- Names are fixed bit strings
- Topology: Prefix Hypercube
  - For each bit from left to right, pick neighbor ID with common flipped bit and common prefix
  - \( \log n \) degree & diameter
- Plus a ring
  - For reliability (with \( k \) pred/succ)
- Suffix Routing from A to B
  - “Fix” bits from left to right
  - E.g. 1010 to 0001: 1010 \( \rightarrow \) 0101 \( \rightarrow \) 0010 \( \rightarrow \) 0000 \( \rightarrow \) 0001
CAN: Content Addressable Network

- Exploit multiple dimensions
- Each node is assigned a zone
- Nodes ID’d by zone boundaries
- Join: chose random point, split its zones
Routing in 2-dimensions

- Routing is navigating a $d$-dimensional ID space
  - Route to closest neighbor in direction of destination
  - Routing table contains $O(d)$ neighbors
- Number of hops is $O(dN^{1/d})$
Koorde

• DeBruijn graphs
  – Link from node $x$ to nodes $2x$ and $2x+1$
  – Degree 2, diameter $\log n$
    • Optimal!

• Koorde is Chord-based
  – Basically Chord, but with DeBruijn fingers
Topologies of Other Oft-cited DHTs

• **Tapestry**
  – Very similar to Pastry/Bamboo topology
  – No ring

• **Kademlia**
  – Also similar to Pastry/Bamboo
  – But the “ring” is ordered by the XOR metric: “bidirectional”
  – Used by the eMule / BitTorrent / Azureus (Vuze) systems

• **Viceroy**
  – An emulated Butterfly network

• **Symphony**
  – A randomized “small-world” network
Incomplete Graphs: Emulation

- For Chord, we assumed exactly $2^m$ nodes. What if not?
  - Need to “emulate” a complete graph even when incomplete.

- DHT-specific schemes used
  - In Chord, node $x$ is responsible for the range $[x, \text{succ}(x))$
  - The “holes” on the ring should be randomly distributed due to hashing
Handle node heterogeneity

• **Sources of unbalanced load**
  – Unequal portion of keyspace
  – Unequal load per key

• **Balancing keyspace**
  – Consistent hashing: Region owned by single node is $O(1/n (1 + \log n))$
  – What about node heterogeneity?
    • Nodes create “virtual nodes” of # proportional to capacity

• **Load per key**
  – Assumes many keys per region
Chord in Flux

- Essentially never a “complete” graph
  - Maintain a “ring” of successor nodes
  - For redundancy, point to \( k \) successors
  - Point to nodes responsible for \( ID \)s at powers of 2
    - Called “fingers” in Chord
    - 1st finger is the successor
Joining the Chord Ring

• Need IP of some node
• Pick a random ID
  – e.g. SHA-1(IP)
• Send msg to current owner of that ID
  – That’s your predecessor in Chord routing
Joining the Chord Ring

• Update pred/succ links
  – Once ring is in place, all well!
• Inform application to move data appropriately
• Search to find “fingers” of varying powers of 2
  – Or just copy from pred /succ and check!
• Inbound fingers fixed lazily

Theorem: If consistency is reached before network doubles, lookups remain log n
Fingers must be constrained?

- No: Proximity Neighbor Selection (PNS)
Handling Churn

• Churn
  – Session time? Life time?
    • For system resilience, session time is what matters

• Three main issues
  – Determining timeouts
    • Significant component of lookup latency under churn
  – Recovering from a lost neighbor in “leaf set”
    • Periodic, not reactive!
    • Reactive causes feedback cycles
      – Esp. when a neighbor is stressed and timing in and out
  – Neighbor selection again
Timeouts

• Recall Iterative vs. Recursive Routing
  – Iterative: Originator requests IP address of each hop
  – Recursive: Message transferred hop-by-hop

• Effect on timeout mechanism
  – Need to track latency of communication channels
  – Iterative results in direct $n \times n$ communication
    • Can’t keep timeout stats at that scale
    • Solution: virtual coordinate schemes [Vivaldi, etc.]
  – With recursive can do TCP-like tracking of latency
    • Exponentially weighted mean and variance

• Upshot: Both work OK up to a point
  – TCP-style does somewhat better than virtual coords at modest churn rates (23 min. or more mean session time)
  – Virtual coords begins to fail at higher churn rates
Recursive vs. Iterative

Left: Simulation of 20,000 lkps for random keys
Recursive lookup takes 0.6 times as long as iterative

Right Figure: 1,000 lookups in test-bed; confirms simulation
Recursive vs. Iterative

- **Recursive**
  - Faster under many conditions
    - Fewer round-trip-times
    - Better proximity neighbor selection
    - Can timeout individual RPCs more tightly
  - Better tolerance to network failure
    - Path between neighbors already known

- **Iterative**
  - Tighter control over entire lookup
    - Easily support windowed RPCs for parallelism
    - Easier to timeout entire lookup as failed
  - Faster to return data directly than use recursive path
Storage Models for DHTs

• Up to now we focused on routing
  – DHTs as “content-addressable network”

• Implicit in “DHT” name is some kind of storage
  – Or perhaps a better word is “memory”
  – Enables indirection in time
  – But also can be viewed as a place to store things
Storage models

• Store *only* on key’s immediate successor
  – Churn, routing issues, packet loss make lookup failure more likely

• Store on *k* successors
  – When nodes detect succ/pred fail, re-replicate

• Cache along reverse lookup path
  – Provided data is immutable
  – ...and performing recursive responses
Storage on successors?

• **Erasure-coding**
  – Data block split into $l$ fragments
  – $m$ diff. fragments necessary to reconstruct the block
  – Redundant storage of data

• **Replication**
  – Node stores entire block
  – Special case: $m = 1$ and $l$ is number of replicas
  – Redundant information spread over fewer nodes

• **Comparison of both methods**
  – $r = l / m$ amount of redundancy

• **Prob. block available:**

\[
p_{\text{avail}} = \sum_{i=m}^{l} \binom{l}{i} p_0^i (1 - p_0)^{l-i}
\]
Latency: Erasure-coding vs. replication

- **Replication**: slightly lower latency
- **Erasure-coding**: higher availability
  - DHash++ uses erasure-coding with $m = 7$ and $l = 14$
What about mutable data?

• Ugh!

• Different views
  – Ivy: Create version trees [Muthitacharoen, OSDI ‘02]
    • Think “distributed version control” system

• Global agreement?
  – Reach consensus among all nodes belonging to a successor groups: “distributed agreement”
    • Difficult, especially at scale
An oft overlooked assumption: The underlay isn’t perfect!

- All have implicit assumption: full connectivity
- Non-transitive connectivity (NTC) not uncommon

\[ B \leftrightarrow C , \ C \leftrightarrow A , \ A \leftrightarrow B \]

- A thinks C is its successor!
Does non-transitivity exist?

- Gerding/Stribling PlanetLab study
  - 9% of all node triples exhibit NTC
  - Attributed high extent to Internet-2

- Yet NTC is also transient
  - One 3 hour PlanetLab all-pair-pings trace
  - 2.9% have persistent NTC
  - 2.3% have intermittent NTC
  - 1.3% fail only for a single 15-minute snapshot

- Level3 ↔ Cogent, but Level3 ↔ X ↔ Cogent

- NTC motivates RON and other overlay routing!
NTC problem fundamental?

Traditional routing

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S → R</strong></td>
<td>A</td>
</tr>
<tr>
<td><strong>A → R</strong></td>
<td>B</td>
</tr>
<tr>
<td><strong>B → R</strong></td>
<td>R</td>
</tr>
</tbody>
</table>
NTC problem fundamental?

- DHTs implement greedy routing for scalability
- Sender might not use path, even though exists: finds local minima when id-distance routing

<table>
<thead>
<tr>
<th>Traditional routing</th>
<th>Greedy routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → R</td>
<td>S → R</td>
</tr>
<tr>
<td>A → R</td>
<td>A → R</td>
</tr>
<tr>
<td>B → R</td>
<td>C → R</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Potential problems?

- Invisible nodes
- Routing loops
- Broken return paths
- Inconsistent roots
Iterative routing: Invisible nodes

- Invisible nodes cause lookup to halt
Iterative routing: Invisible nodes

- Invisible nodes cause lookup to halt
- Enable lookup to continue
  - Tighter timeouts via network coordinates
  - Lookup RPCs in parallel
  - Unreachable node cache
Inconsistent roots

- Nodes do not agree where key is assigned: inconsistent views of root
  - Can be caused by membership changes
  - Also due to non-transitive connectivity: may persist!
Inconsistent roots

- Root replicates (key,value) among leaf set
  - Leafs periodically synchronize
  - Get gathers results from multiple leafs

- Not applicable when require fast update
Longer term solution?

- Route around local minima when possible
- Have nodes maintain link-state of neighbors
  - Perform one-hop forwarding if necessary (think RON!)

<table>
<thead>
<tr>
<th></th>
<th>Traditional routing</th>
<th>Greedy routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → R</td>
<td>A</td>
<td>S → R</td>
</tr>
<tr>
<td>A → R</td>
<td>B</td>
<td>A → R</td>
</tr>
<tr>
<td>B → R</td>
<td>R</td>
<td>C → R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Summary

• Peer-to-peer systems
  – Unstructured systems
    • Finding hay, performing keyword search
  – Structured systems (DHTs)
    • Finding needles, exact match

• Distributed hash tables
  – Based around consistent hashing with views of $O(\log n)$
  – Chord, Pastry, CAN, Koorde, Kademlia, Tapestry, Viceroy, ...

• Lots of systems issues
  – Heterogeneity, storage models, locality, churn management, underlay issues, ...
  – DHTs (Kademlia) deployed in wild: Vuze is 1M+ active users