P2P Systems and Distributed Hash Tables

Section 9.4.2

COS 461: Computer Networks
Spring 2011

Mike Freedman
http://www.cs.princeton.edu/courses/archive/spring11/cos461/
P2P as Overlay Networking

• P2P applications need to:
  – Track identities & IP addresses of peers
    • May be many and may have significant churn
  – 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
Early P2P
Early P2P I: Client-Server

• Napster
  – Client-server search
  – “P2P” file xfer

1. insert
2. search
3. transfer

xyz.mp3
xyz.mp3?
Early P2P II: Flooding on Overlays

Flooding
Early P2P II: Flooding on Overlays

Flooding

xyz.mp3

search
Early P2P II: Flooding on Overlays
Early P2P II: “Ultra/super peers”

- Ultra-peers can be installed (KaZaA) or self-promoted (Gnutella)
  - Also useful for NAT circumvention, e.g., in Skype
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
Basic Hashing for Partitioning?

• Consider problem of data partition:
  – Given document X, choose one of k servers to use

• Suppose we use modulo hashing
  – Number servers 1..k
  – Place X on server \( i = (X \mod k) \)
    • Problem? Data may not be uniformly distributed
  – Place X on server \( i = \text{hash}(X) \mod k \)
    • Problem?
      – What happens if a server fails or joins (\( k \to k\pm1 \))?
      – What is different clients has different estimate of k?
      – Answer: All entries get remapped to new nodes!
Consistent Hashing

- Consistent hashing partitions key-space among nodes
- Contact appropriate node to lookup/store key
  - Blue node determines red node is responsible for $key_1$
  - Blue node sends lookup or insert to red node
• Partitioning key-space among nodes
  – Nodes choose random identifiers: e.g., \text{hash(IP)}
  – Keys randomly distributed in ID-space: e.g., \text{hash(URL)}
  – Keys assigned to node “nearest” in ID-space
  – Spreads ownership of keys evenly across nodes
Consistent Hashing

• Construction
  – Assign $n$ hash buckets to random points on mod $2^k$ circle; hash key size = $k$
  – Map object to random position on circle
  – Hash of object = closest clockwise bucket
    – successor (key) $\to$ bucket

• Desired features
  – Balanced: No bucket has disproportionate number of objects
  – Smoothness: Addition/removal of bucket does not cause movement among existing buckets (only immediate buckets)
  – Spread and load: Small set of buckets that lie near object
Consistent hashing and failures

• Consider network of $n$ nodes

• If each node has 1 bucket
  – Owns $1/n^{th}$ of keyspace in expectation
  – Says nothing of request load per bucket

• If a node fails:
  – Its successor takes over bucket
  – Achieves smoothness goal: Only localized shift, not $O(n)$
  – But now successor owns 2 buckets: keyspace of size $2/n$

• Instead, if each node maintains $v$ random node IDs, not 1
  – “Virtual” nodes spread over ID space, each of size $1/vn$
  – Upon failure, $v$ successors take over, each now stores $(v+1)/vn$
## Consistent hashing vs. DHTs

<table>
<thead>
<tr>
<th></th>
<th>Consistent Hashing</th>
<th>Distributed Hash Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing table size</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Lookup / Routing</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Join/leave: Routing</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Join/leave: Routing</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Key Movement</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Distributed Hash Table

- Nodes’ neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node
• Nodes’ neighbors selected from particular distribution
  – Visual keyspace as a tree in distance from a node
  – At least one neighbor known per subtree of increasing size / distance from node
• Nodes’ neighbors selected from particular distribution
  – Visual keyspace as a tree in distance from a node
  – At least one neighbor known per subtree of increasing size / distance from node
• Route greedily towards desired key via overlay hops
The Chord DHT

• Chord ring: ID space mod $2^{160}$
  – $\text{nodeid} = \text{SHA1 (IP address, i)}$
    for $i=1..v$ virtual IDs
  – $\text{keyid} = \text{SHA1 (name)}$

• Routing correctness:
  – Each node knows successor and predecessor on ring

• Routing efficiency:
  – Each node knows $O(\log n)$ well-distributed neighbors
Basic lookup in Chord

```
lookup (id):
    if ( id > pred.id && id <= my.id )
        return my.id;
    else
        return succ.lookup(id);
```

- Route hop by hop via successors
  - $O(n)$ hops to find destination id
Efficient lookup in Chord

`lookup (id):`
```
if ( id > pred.id &&
    id <= my.id )
    return my.id;
else
    // fingers() by decreasing distance
    for finger in fingers():
        if id <= finger.id
            return finger.lookup(id);
    return succ.lookup(id);
```

- Route greedily via distant “finger” nodes
  - $O(\log n)$ hops to find destination id
Building routing tables

Routing Tables

Routing

For $i$ in $1...\log n$:

$$\text{finger}[i] = \text{successor} \left( (\text{my.id} + 2^i) \mod 2^{160} \right)$$
Joining and managing routing

• Join:
  – Choose nodeid
  – *Lookup (my.id)* to find place on ring
  – During lookup, discover future successor
  – Learn predecessor from successor
  – Update succ and pred that you joined
  – Find fingers by *lookup ((my.id + 2^i) mod 2^{160})*

• Monitor:
  – If doesn’t respond for some time, find new

• Leave: Just go, already!
  – *(Warn your neighbors if you feel like it)*
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
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
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 deployed in wild: Vuze (Kademlia) has 1M+ active users