Πανεπιστήμιο Πατρών

Προχωρημένα Θέματα σε Κατανεμημένα Συστήματα

Distributed Hash Tables
Today’s Agenda

- What are DHTs?
  - Why are they useful?

- What makes a “good” DHT design

- Case studies
  - Chord
  - Pastry

Based on slides by Pascal Felber
P2P challenge: Locating content

- Simple strategy: flood (e.g., expanding ring) until content is found
  - If $R$ of $N$ nodes have a replica, the expected search cost is at least $N/R$, i.e., $O(N)$
  - Need many replicas to keep overhead small

- Other strategy: centralized index (Napster)
  - Single point of failure, high load

- Goal: **Decentralize the index!**
Indexed Search

- **Idea**
  - Store **particular content on particular nodes**
    - alternatively: **pointers to content**
  - When a node wants this content, go to the node that is supposed to hold it (or knows where it is)

- **Challenges**
  - Avoid bottlenecks:
    - *Distribute the responsibilities “evenly”* among the existing nodes
  - Self-organization w.r.t. nodes joining or leaving (or failing)
    - Give responsibilities to joining nodes
    - Redistribute responsibilities from leaving nodes
  - Fault-tolerance and robustness
    - Operate correctly also under failures
In a classic Hash Table:
- Table has $N$ buckets
- Each data item has a key
- Key is hashed to find bucket in hash table
- Each bucket is expected to hold $1/N$ of the items, so storage is balanced

In a Distributed Hash Table (DHT), nodes are the buckets:
- Network has $N$ nodes
- Each data item has a key
- Key is hashed to find peer responsible for it
- Each node is expected to hold $1/N$ of the items, so storage is balanced
- Additional requirement: Also balance routing load!!
DHTs: Problems

- **Problem 1 (dynamicity):** adding or removing nodes
  - With hash mod N, virtually every key will change its location!
    \[ h(k) \mod m \neq h(k) \mod (m+1) \neq h(k) \mod (m-1) \]

- **Solution:** use consistent hashing
  - Define a fixed hash space
  - All hash values fall within that space and do not depend on the number of peers (hash bucket)
  - Peers have a hash value (ID)
  - Data items have hash values (KEY)
  - Each data items goes to peer with ID closest to its KEY
DHT Hashing

- Based on **consistent hashing** (designed for Web caching)
  - Each server is identified by an ID uniformly distributed in range [0, 1]
  - Each web page’s URL is (via some hash function) associated with an ID which is uniformly distributed in [0, 1]
  - A page is stored to the closest server in the ID space
  - A client hashes the desired URL, retrieves page from appropriate server
  - Good load balancing: each server covers roughly equal intervals and stores roughly the same number of pages
  - Adding or removing a server invalidates few keys

![Diagram of DHT Hashing](image)
DHTs: Problems (cont’d)

Problem 2 (size): all nodes must be known to insert or lookup data
- Works with small and static server populations

Solution: each peer has only a few “neighbors”
- Messages are routed through neighbors via multiple hops (overlay routing)
What Makes a Good DHT Design?

- **Small diameter**
  - Should be able to route to any node in a few hops
  - Different DHTs differ fundamentally only in the routing approach

- **Load sharing**
  - DHT routing mechanisms should be decentralized
  - no *single point of failure*
  - no *bottleneck*

- **Small degree**
  - The number of neighbors for each node should remain “reasonable”

- **Low stretch**
  - To achieve good performance, minimize *ratio of DHT vs. IP latency*

- **Should gracefully handle nodes joining and leaving**
  - Reorganize the neighbor sets
  - Bootstrap mechanisms to connect new nodes into the DHT
  - Repartition the affected keys over existing nodes
DHT Interface

- Minimal interface (data-centric)
  \[ \text{Lookup(key)} \rightarrow \text{IP address} \]

- Generality: Supports a wide range of applications
  - Keys have no semantic meaning
  - Values are application dependent

- DHTs do **not** store the data
  - Data storage can be built on top of DHTs
    \[ \text{Lookup(key)} \rightarrow \text{data} \]
    \[ \text{Insert(key, data)} \]
DHTs support many applications:

- Network storage [CFS, OceanStore, PAST, ...]
- Web cache [Squirrel, ...]
- E-mail [e-POST, ...]
- Query and indexing [Kademlia, ...]
- Event notification [Scribe]
- Application-layer multicast [SplitStream, ...]
- Naming systems [ChordDNS, INS, ...]
- ...
DHTs in Context

User Application

store_file \rightarrow load_file

File System

store_block \rightarrow load_block

Reliable Block Storage

lookup

DHT

send \rightarrow receive

Transport

Retrieve and store files
Map files to blocks

Storage
Replication
Caching

Lookup
Routing

Communication
Case Studies
- Chord
- Pastry

Questions
- How is the hash space divided evenly among nodes?
- How do we locate a node?
- How do we maintain routing tables?
- How do we cope with (rapid) changes in membership?
CHORD
(MIT)
Circular $m$-bit ID space for both keys and node IDs

Node ID = SHA-1(IP address)

Key ID = SHA-1(key)

Each key is mapped to its successor node
- Node whose ID is equal to or follows the key ID

Key distribution
- Each node responsible for $O(K/N)$ keys
- $O(K/N)$ keys move when a node joins or leaves
Basic Chord: State and Lookup

- Each node knows only two other nodes on the ring:
  - Successor
  - Predecessor (for ring management)

- Lookup is achieved by forwarding requests around the ring through successor pointers
  - Requires $O(N)$ hops
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n, n.successor])
        return n.successor;
    else
        // forward the query around the circle
        return successor.find_successor(id);

(a)
Complete Chord

- Each node knows these two nodes:
  - Successor
  - Predecessor (for ring management)

- But also: Each node has \( m \) fingers
  - \( n \).finger\( (i) \) points to node on or after \( 2^i \) steps ahead
  - \( n \).finger\( (0) \) == \( n \).successor
  - \( O(\log N) \) state per node

- Lookup is achieved by following longest preceding fingers, then the successor
  - \( O(\log N) \) hops

Finger table:

| \( N8+1 \) | \( N14 \) |
| \( N8+2 \) | \( N14 \) |
| \( N8+4 \) | \( N14 \) |
| \( N8+8 \) | \( N21 \) |
| \( N8+16 \) | \( N32 \) |
| \( N8+32 \) | \( N42 \) |

\( m=6 \)
Complete Chord

// ask node n to find the successor of id
n.find_successor(id)
    if (key ∈ (n, n.successor])
        return n.successor;
    else
        n' = closest_preceding_node(id);
        return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
    for i = m downto 1
        if (finger[i] ∈ (n, id))
            return finger[i];
    return n;
Chord Ring Management

- For **correctness**, Chord needs to maintain the following invariants
  - **Successors** are **correctly** maintained
  - For every key $k$, $\text{succ}(k)$ is responsible for $k$

- **Fingers** are for **efficiency**, not necessarily correctness!
  - One can always default to successor-based lookup
  - Finger table can be updated lazily
Joining the Ring

- Three step process:

1. Outgoing links
   - Initialize predecessor and all fingers of new node

2. Incoming links
   - Update predecessors and fingers of existing nodes

3. Transfer some keys to the new node
Joining the Ring — Step 1

- Initialize the new node finger table
  - Locate any node \( n \) in the ring
  - Ask \( n \) to lookup the peers at \( j+2^0, j+2^1, j+2^2 \ldots \)
  - Use results to populate finger table of \( j \)
Joining the Ring — Step 2

- Updating fingers of existing nodes
  - New node \( j \) calls update function on existing nodes that must point to \( j \)
  - Nodes in the ranges \([j-2^i, \text{pred}(j)-2^i+1]\)
  - \( O(\log N) \) nodes need to be updated
Joining the Ring — Step 3

- Transfer key responsibility
  - Connect to successor
  - Copy keys from successor to new node
  - Update successor pointer and remove keys
Leaving the Ring (or Failing)

- Node departures are treated as node failures

- Failure of nodes might cause incorrect lookup:
  - N8 doesn’t know correct successor, so lookup of K19 fails

- Solution: successor list:
  - Each node n knows r immediate successors
  - After failure, n contacts first alive successor and updates successor list
  - Correct successors guarantee correct lookups
Stabilization

- Case 1: finger tables are reasonably fresh

- Case 2: successor pointers are correct, not fingers

- Case 3: successor pointers are inaccurate or key migration is incomplete — **MUST BE AVOIDED!**

- Stabilization algorithm periodically verifies and refreshes node pointers (including fingers)
  - Eventually stabilizes the system when no node joins or fails
Chord and Network Topology

Nodes numerically-close are not topologically-close (1M nodes = 10+ hops)
Cost of Lookup

- Cost is $O(\log N)$, constant is 0.5
Chord Discussion

- **Search types**
  - Only equality

- **Scalability**
  - Diameter (search and update) in $O(\log N)$ w.h.p.
  - Degree in $O(\log N)$
  - Construction: $O(\log^2 N)$ if a new node joins

- **Robustness**
  - Replication might be used by storing replicas at successor nodes
PASTRY
(MSR + Rice)
Pastry

- Circular $m$-bit ID space for both keys and nodes
- Addresses in base $2^b$ with $m/b$ digits
  - Address: $m$ bits
  - Digit: $b$ bits
  - $\Rightarrow$ Address: $m/b$ digits
- Node ID = SHA-1(IP address)
- Key ID = SHA-1(key)
- A key is mapped to the node whose ID is $\text{numerically-closest}$ to the key ID
Pastry Routing

\[ \text{lookup}(K0202) \]

\[ m = 8 \]
\[ b = 2 \]

\[ 2^{m-1} : 0 \]
Pastry Routing

\[ \text{lookup}(K0202) \]
Pastry Routing

$m=8$
$b=2$

$2^{m-1} \cdot 0$

lookup($K_{0202}$)
Pastry Routing

\[ m = 8 \]
\[ b = 2 \]

\[ 2^{m-1}; 0 \]

lookup(K0202)
Pastry Routing

\[
m = 8 \\
b = 2
\]

lookup(\text{K0202})

\[O(\log N)\] hops
Pastry Routing

- $O(\log N)$ hops

- Route to 0202:
  \[2221 \rightarrow 0002 \rightarrow 0221 \rightarrow 0201 \rightarrow 0202\]

- If chain not complete, forward to numerically closest neighbor (successor)
  \[2221 \rightarrow 0002 \rightarrow 0221 \rightarrow 0210 \rightarrow 0201 \rightarrow 0202\]
Pastry State and Lookup

- For each prefix, a node knows some other node (if any) with same prefix and different next digit

- For instance, N0201:
  - N-: N1???, N2???, N3???
  - N0: N00??, N01??, N03??
  - N02: N021?, N022?, N023?
  - N020: N0200, N0202, N0203

- When multiple nodes, choose topologically-closest
  - Maintain good locality properties (more on that later)
### A Pastry Routing Table

Contains the $L$ nodes that are **numerically closest to local node**

**MUST BE UP TO DATE**

<table>
<thead>
<tr>
<th>Leaf set</th>
<th>&lt; SMALLER</th>
<th>LARGER &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>10233033</td>
<td>10233021</td>
<td>10233120</td>
</tr>
<tr>
<td>10233001</td>
<td>10233000</td>
<td>10233230</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routing Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>02212102</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>10031203</td>
</tr>
<tr>
<td>10200230</td>
</tr>
<tr>
<td>10230322</td>
</tr>
<tr>
<td>10233001</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Contains the nodes that are closest to local node according to proximity metric

$b=2$, so node ID is base 4 (16 bits)

Entries in the $m^{th}$ column have $m$ as next digit

$n^{th}$ digit of current node

Entries in the $n^{th}$ row share the first $n$ digits with current node

Entries with no suitable node ID are left empty
The routing procedure is executed whenever a message arrives at a node

1. **IF** *(key in Leaf Set)*
   1. If key is in leaf set, destination is 1 hop away, forward directly to destination.

2. **ELSE IF** *(key in Routing Table)*
   1. Forward to node that matches one more digit

3. **ELSE**
   1. Forward to a node numerically closer, from Leaf Set

- **The procedure always converges!**

```plaintext
(1) if \((L_{\left\lceil \frac{|L|}{2} \right\rceil} \leq D \leq L_{\lceil \frac{|L|}{2} \rceil})\) {
(2) // \(D\) is within range of our leaf set
(3) forward to \(L_i\), s.th. \(|D - L_i|\) is minimal;
(4) } else {
(5) // use the routing table
(6) Let \(l = \text{shl}(D, A)\);
(7) if \(R_{Dl}^i \neq \text{null}\) {
(8) forward to \(R_{Dl}^i\);
(9) }
(10) else {
(11) // rare case
(12) forward to \(T \in L \cup R \cup M\), s.th.
(13) \(\text{shl}(T, D) \geq l\),
(14) \(|T - D| < |A - D|\)
(15) }
(16) }
```
Joining

X joins

X knows A (A is “close” to X)

Join message

Route message to node numerically closest to X’s ID

0629’s routing table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>—</td>
</tr>
<tr>
<td>C</td>
<td>—</td>
</tr>
<tr>
<td>D</td>
<td>—</td>
</tr>
</tbody>
</table>

A’s neighborhood set

D’s leaf set

A — ???
B — 0??
C — 06??
D — 062?
Desired **proximity invariant**: all routing table entries refer to a node that is near the present node, according to proximity metric, among all live nodes with prefix appropriate for the entry.

**Assumptions**
- Scalar proximity metric (e.g., ping delay, # IP hops)
- A node can probe distance to any other node

We assume this property holds prior to node X joining the system.

Show how we can maintain the property after X joins the system.
Pastry and Network Topology

Expected node distance increases with row number in routing table.

Smaller and smaller numerical jumps
Bigger and bigger topological jumps
Node Departure

- To replace a failed node in the leaf set, the node contacts the live node with the largest index on the side of the failed node, and asks for its leaf set.

- To repair a failed routing table entry $R^d_i$, node contacts first the node referred to by another entry $R^i_j$, $i \neq d$ of the same row, and asks for that node’s entry for $R^d_i$.

- If a member in the Neighbors table is not responding, the node asks other members for their Neighbors table, check the distance of each of the newly discovered nodes, and updates its own Neighbors table.
Average # Routing Hops

- 2 nodes selected randomly; a message is routed between
- Maximum route length is $\left\lceil \log_2 b \cdot N \right\rceil$ (5 for $N=100,000$)

**Fig. 4.** Average number of routing hops versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$ and 200,000 lookups.
Fig. 5. Probability versus number of routing hops. $b = 4$, $|L| = 16$, $|M| = 32$, $N = 100,000$ and 200,000 lookups.
# Routing Hops vs. Node Failures

![Graph showing routing hops vs. node failures]

**Fig. 10.** Number of routing hops versus node failures, $b = 4$, $|L| = 16$, $|M| = 32$, 200,000 lookups and 5,000 nodes with 500 failing.
Pastry Discussion

- **Search types**
  - Only equality

- **Scalability**
  - Search $O(\log_2 N)$ w.h.p.

- **Robustness**
  - Can route around failures via nodes of leaf set
  - Replication might be used
Conclusion

- DHTs are a simple, yet powerful abstraction
  - Building block of many distributed services (file systems, application-layer multicast, distributed caches, etc.)

- Many DHT designs, with various pros and cons
  - Balance between state (degree), speed of lookup (diameter), and ease of management

- System must support rapid changes in membership
  - Dealing with joins/leaves/failures is not trivial
  - Dynamics of P2P networks are difficult to analyze

- Many open issues worth exploring