Previous Topics

Data
- Database design

Queries
- Query processing
- Localization
- Operators
- Optimization

Transactions
- Concurrency control
- Reliability
- Replication

Next Topic

Client-server architecture
- Relational data
- Good understanding of
  - What the data is
  - Where the data is

Client-server architecture
- Relational data
- Good understanding of
  - Where the data is?
Peer-to-Peer Systems

_Distributed applications where nodes are_

- Autonomous
- Very loosely coupled
- Equal in role or functionality
- Sharing & exchanging resources with each other

Peer-to-Peer Systems

**Related concepts**

- File sharing
  - P2P is one option
- Grid computing
  - Focus is on computing
- Autonomic computing
  - Focus is on self-management

Peer-to-Peer Systems

**Search**

_Essential problem to solve_

- Query: Who has X?
### Peer-to-Peer Systems

**Search**

Node 1
- Resources: R_{1,1}, R_{1,2}, ...

Node 2
- Resources: R_{2,1}, R_{2,2}, ...

Node 3
- Resources: R_{3,1}, R_{3,2}, ...

Query: Who has X?

Answers:
- Node 1
- Node 2

### Distributed Lookup

Have \(< k, v >\) pairs, each of \(n\) nodes holds some pairs.

Given key \(k\), find matching values \(\{ v_1, v_2, ... \}\).

<table>
<thead>
<tr>
<th>k</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
</tbody>
</table>

\[\text{lookup}(4) = \{ a, d \}\]

### Communication overlay network

**Structured**
- (+) Efficient distributed lookup
- Unstructured
  - (+) Easy, robust
    - Can handle high churn rates
  - (-) Flood queries

**Hybrid**
- E.g., centralized search, decentralized exchange
Distributed Lookup

Distributed hashing
Most common way to create a *structured* overlay network

\[ H(k) \text{ is an } m \text{-bit number (} k \text{ is a key)} \]
\[ H(X) \text{ is an } m \text{-bit number (} X \text{ is a node identifier)} \]

Hash function \( H \) is “good”

\[ k \text{ stored at } Y \]

Chord

The Chord circle

**Ownership rule**
Consider nodes \( X, Y \) such that \( Y \) follows \( X \) clockwise
Node \( Y \) owns keys \( k \) such that \( H(k) \) in \( (H(X), H(Y)] \)
Chord

Notation

X.function(...) (remote) calls function at X
X.A returns value A at X
If X omitted, refers to current node

Chord

Successor/predecessor links

Search for owner using successor links

X.find_succ(k):
  if k in (pred, X]
    return X
  else if k in (X, succ]
    return succ
  else
    return succ.find_succ(k)

Value lookup

X.lookup(k):
  Y := X.find_succ(k)
  return Y.get_value(k)

X.get_value(k):
  // Return local value v for k, if it exists
Example Searching for $K_{52}$

$N_{51}.\text{find\_succ}(K_{52}) = N_{56}$

$N_{8} + 1 \rightarrow N_{14}$  
$N_{8} + 2 \rightarrow N_{14}$  
$N_{8} + 4 \rightarrow N_{14}$  
$N_{8} + 8 \rightarrow N_{21}$  
$N_{8} + 16 \rightarrow N_{52}$  
$N_{8} + 32 \rightarrow N_{42}$

Node that owns key $8 + 32 = 40$

Search using the finger table

$X.\text{find\_succ}(k)$:
  if $k$ in (pred, $X$) return $X$
  if $k$ in ($X$, succ) return succ
  else
    $Y := \text{closest\_preceeding}(k)$
    return $Y.\text{find\_succ}(k)$

$X.\text{closest\_preceeding}(k)$:
  for $i := m$ downto 1
    if $\text{finger}[i]$ in ($X$, $k$) return $\text{finger}[i]$
  return nil
Example
Looking up K54

Chord

Example
Looking up K54

Adding nodes
Need to
1. Update links
2. Move data

For now, assume nodes never die
Chord

Adding nodes

X.join(Y):
  // Node Y is known to belong to the circle
  pred := nil;
  succ := Y.find_succ(X);

Chord

Periodic stabilization

X.stabilize():
  Y := succ.pred
  if Y in (X, succ)
    succ := Y
    succ.notify(X)
  X.notify(Y):
    if pred = nil or Y in (pred, X)
      pred := Y

Chord

Join example
Before updating links

Chord

Join example
After Nx.join()
Chord

Join example
After $N_x$.stabilize()

Chord

Join example
After $N_p$.stabilize()

Exercise: fix finger table

Chord

Moving data
When?

Chord

Moving data
After $N_x$.notify($N_x$)

Send all keys in $(N_p, N_x]$ when $N_x$.pred gets updated
Chord

Moving data
Revised notify()

X.notify(Y):
   if pred = nil or Y in (pred, X)
      Y.add(data in (pred, Y])
      temp := pred
      pred := Y
      X.remove(data in (temp, Y])

Glossing over concurrency issues
   E.g., what happens to lookups while moving data?

Exercise: when pred = nil, what data gets moved?

Chord

Moving data
Lookup at wrong node

Lookup for all k in (N_p, N_s] directed to N_s

Chord

Moving data
Revised lookup()

X.lookup(k):
   ok := false
   while not ok
      Y := X.find_succ(k)
      [ok, v] := Y.get_value(k)
   return v

X.get_value(k):
   if k in (pred, X]
      return [true, value for k]
   else
      return [false, nil]
Chord

Moving data
Revised lookup() works
pred, succ links eventually correct
Data ends up at correct node
Finger pointers speed up searches, but do not cause problems

Chord

Performance

With high probability, the number of nodes that must be contacted to find a successor is $O(\log n)$
Although finger table contains room for $m$ entries, only $O(\log n)$ need to be stored
Experimental results show average lookup time is $\sim (\log n)/2$

Chord

Node failures

Fixing links

Assume $N_x$ dies
Links become invalid
Data gets lost

Node failures

Also, keep backup links to $s > 1$ successors
Chord

Node failure example
State before failure

Chord

Node failure example
After failure
After $N_6$.check_pred()

Chord

Node failure example
After $N_p$ discovers that $N_x$ is down

Chord

Node failure example
After stabilization
Chord

Node failure
Protection from data loss: e.g., robust nodes (see replication notes)

Robust node X

Chord

Summary
Finding owner nodes
  • Successor and predecessor links
  • Finger table
Adding nodes
  • Updating links
  • Moving data
Coping with node failures
  • Fixing links
  • Protecting from data loss
Performance

Replicated Hash Table

Adding nodes
E.g., N0 overloaded
Adding nodes

First, set up N1

Replicated Hash Table

Data for keys that hash to 0,1

Node N0

Data for keys that hash to 2,3

Node N2

Adding nodes

Next, copy data to N1

Replicated Hash Table

Data copy

Adding nodes

Next, change control

Replicated Hash Table

Adding nodes

Next, remove data from N0
**Replicated Hash Table**

**Adding nodes**
Finally, update other nodes
Eagerly by N0 or N1? Lazily during future lookups?

**Chord vs. Replicated Hash Table**

Which is simpler to implement?
Cost of operations
  - Looking up values: $O(\log n)$ vs. $O(1)$
  - Adding nodes
  - Recovering from node failures
Storage cost
  - Routing table size: $\log n$ vs. $n$

**Distributed Lookup**

**Communication overlay network**
Structured ✔
  - Chord
  - Replicated hash table
Unstructured
  - Neighborhood search
Neighborhood Search

Each node stores its own data
Searches nearby nodes
E.g., Gnutella

Example

Node N1

N1.lookup(13), TTL=4

Answer so far = { c, d, f, x }

Incorrect/incomplete
Neighborhood Search

Optimization

Queries have unique identifiers

Nodes keep cache of recent queries (query identifier and TTL)

Example

TTL = 1

TTL = 2

TTL = 3

Node N1.lookup(13), TTL = 4, id = 77

Do not process 77

Bootstrap server

Known nodes: S = N1, N2, ...

Get neighbors

Add to S
Neighborhood Search

Problems
Unnecessary messages
High load and traffic
E.g., if nodes have $p$ neighbors, each search $\sim p^{TTL}$ messages
Low capacity nodes are a bottleneck
May not find all answers

Advantages
Can handle complex queries
Simple, robust algorithm
Works well if data is highly replicated

Open Problems

Peer-to-Peer Systems Summary

Structured networks
  - Chord
  - Replicated hash table
Unstructured networks
  - Neighborhood search
Open problems