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

*Where the data is?*
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

Distributed applications where nodes are
- Autonomous
- Very loosely coupled
- Equal in role or functionality
- Sharing & exchanging resources with each other

Search

Essential problem to solve
- Query: Who has X?
Peer-to-Peer Systems

Search

Node 1 Resources R1, R2, ...

Node 2 Resources R1, R2, ...

Node 3 Resources R1, R2, ...

Query Who has X?

Answers

Node 1

Node 2

Node 3

Distributed Lookup

Have <k, v> pairs, each of n nodes holds some pairs
Given key k, find matching values \{v1, v2, ...\}

<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>

lookup(4) = \{a, d\}

Distributed Lookup

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) is an m-bit number (k is a key)
H(X) is an m-bit number (X is a node identifier)
Hash function H is “good”

\[ H(k) \]

\[ H(X) \]

\[ H(Y) \]

\[ 2^{m-1} \]

\[ k \]

\[ v \]

stored at Y

Chord

The Chord circle

Using hashed values
E.g., N56 is node with id hashing to 56

Chord

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)]

N54
N3
Stores K55, K56, ..., K3
Chord

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

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 K52

N51.find_succ(K52) = N56
N14.find_succ(K52)

Chord

Finger table

Node that owns key 8 + 32 = 40

Chord

Search using the finger table
X.find_succ(k):
  if k in (pred, X) return X
  if k in (X, succ) return succ
  else
    Y := closest_preceeding(k)
    return Y.find_succ(k)

X.closest_preceeding(k):
  for i := m downto 1
    if finger[i] in (X, k) return finger[i]
  return nil
Example
Looking up K54

Chord

Example
Looking up K54

Chord

Adding nodes

Need to
1. Update links
2. Move data

Chord

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 Nₖ.join()
**Chord**

**Join example**

After $N_x$.stabilize()

**Exercise: fix finger table**

**Chord**

**Moving data**

When?

After $N_x$.notify($N_x$)

Send all keys in $(N_p, N_x]$ when $N_x$.pred gets updated
Moving data
Revised notify()

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

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

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

Moving data
Lookup at wrong node

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

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

Assume $N_x$ dies
Links become invalid
Data gets lost

Chord

Node failures
Fixing links

X.check_pred():
  if pred failed
  pred := nil

Also, keep backup links to $s > 1$ successors
Chord
Node failure example
State before failure

Chord
Node failure example
After failure
After Nₖ.check_pred()

Chord
Node failure example
After Nₚ discovers that Nₖ 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
Takeover protocols

Backup protocols

Replicated Hash Table

Adding nodes
E.g., N0 overloaded
Adding nodes
First, set up N1

Replicated Hash Table

Next, copy data to N1

Replicated Hash Table

Next, change control

Replicated Hash Table

Next, remove data from N0
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. \( n \)
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, f, x}

Incorrect/incomplete
Neighborhood Search

Optimization
Queries have unique identifiers
Nodes keep cache of recent queries (query identifier and TTL)

Neighborhood Search

Example

Node N[77.4]  
N1.lookup(13), TTL = 4, id = 77

Do not process 77

Bootstrap server

Get neighbors

Add to S

Known nodes S = N1, N2, ...

Node N[77.4]  
N1.lookup(13), TTL = 4, id = 77

Do not process 77

Node N[77.1]  
TTL = 1

Node N[77.2]  
TTL = 2

Node N[77.3]  
TTL = 3

Node N[77.4]  
TTL = 4, id = 77
**Neighborhood Search**

**Problems**
- Unnecessary messages
- High load and traffic
  - E.g., if nodes have \( p \) neighbors, each search \( \sim p^{\text{TTL}} \) 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**
- Availability
- Efficiency
- Load balancing
- Authenticity
- DoS prevention
- Incentives
- Anonymity
- Correctness
- Participation

**Peer-to-Peer Systems Summary**
- Structured networks
  - Chord
- Replicated hash table
- Unstructured networks
  - Neighborhood search
- Open problems