CS 347
Parallel and Distributed Data Processing
Spring 2016

Notes 9: Peer-to-Peer Systems
Previous Topics

Data
  Database design

Queries
  Query processing
    Localization
  Operators
  Optimization

Transactions
  Concurrency control
  Reliability
  Replication
Previous Topics

Data
  Database design

Queries
  Query processing
    Localization
  Operators
  Optimization

Transactions
  Concurrency control
  Reliability
  Replication

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

- napster
- gnutella
- BitTorrent
- Skype
- Spotify
- bitcoin

Timeline:
- 1999
- 2000
- 2001
- 2003
- 2006
- 2009
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

\textit{Who has X?}

Node 1
Resources \( R_{1,1}, R_{1,2}, \ldots \)

Node 2
Resources \( R_{2,1}, R_{2,2}, \ldots \)

Node 3
Resources \( R_{3,1}, R_{3,2}, \ldots \)
Peer-to-Peer Systems

Search

Query: Who has X?

Node 1: Resources $R_{1,1}, R_{1,2}, \ldots$

Node 2: Resources $R_{2,1}, R_{2,2}, \ldots$

Node 3: Resources $R_{3,1}, R_{3,2}, \ldots$

Answers
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

Request resource

Provide resource
Distributed Lookup

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

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

\[
\begin{array}{c|c}
 k & v \\
1 & a \\
1 & b \\
4 & a \\
7 & c \\
3 & a \\
1 & a \\
4 & d
\end{array}
\]

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

Distributed hashing

Two approaches

  Chord

  Replicated hash table (RHT)
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 $(H(X), H(Y)]$

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
Chord

Successor/predecessor links

N1.pred

N1.succ

m = 6
Chord

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)
Chord

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
Chord

Example
Searching for \( K_{52} \)

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

\[ N_{14}.\text{find\_succ}(K_{52}) \]
Chord

Finger table

Finger table for N8

<table>
<thead>
<tr>
<th>N8 + 1</th>
<th>N14</th>
</tr>
</thead>
<tbody>
<tr>
<td>N8 + 2</td>
<td>N14</td>
</tr>
<tr>
<td>N8 + 4</td>
<td>N14</td>
</tr>
<tr>
<td>N8 + 8</td>
<td>N21</td>
</tr>
<tr>
<td>N8 + 16</td>
<td>N32</td>
</tr>
<tr>
<td>N8 + 32</td>
<td>N42</td>
</tr>
</tbody>
</table>
Chord

Finger table

Finger table for N8

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

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
Chord

Example
Looking up K54

N56
N51
N48
N42
N38
N32
N8
N14
N21

N8.find_succ(K54)
N42.find_succ(K54)

m = 6
Chord

Example
Looking up K54

N51.find_succ(K54)
N42.find Succ(K54)

N56
N51
N48
N42
N38
N32
N1
N8
N14
N21

m = 6
Chord

Example
Looking up K54

N51.find_succ(K54)
N56
N48
N42
N38
N32
N1
N8
N14
N21

N8.find_succ(K54)
N42.find_succ(K54)

m = 6
Chord

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
Join example
Before updating links
Chord

Join example

After $N_x . join()$
Chord

Join example

After $N_x$.stabilize()
Chord

Join example

After $N_p$.stabilize()

Exercise: fix finger table
Chord

Moving data
When?

Keys in \((N_p, N_x]\)

Keys in \((N_p, N_s]\)

\(N_p\)

\(N_x\)

\(N_s\)
Chord

Moving data
After $N_s.notify(N_x)$

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

Moving data
Revised notify()

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

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

Moving data
Revised notify()

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

Moving data
Lookup at wrong node

Lookup for all $k$ in $(N_p, N_x]$ 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

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 $N_s$.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)
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

Node N0

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N3</td>
</tr>
</tbody>
</table>

Data for keys that hash to 0

Node N1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N3</td>
</tr>
</tbody>
</table>

Data for keys that hash to 1

Node N2

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N3</td>
</tr>
</tbody>
</table>

Data for keys that hash to 2

Node N3

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N3</td>
</tr>
</tbody>
</table>

Data for keys that hash to 3
Replicated Hash Table

Adding nodes
E.g., N0 overloaded

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Node N0

Data for keys that hash to 0,1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Node N2

Data for keys that hash to 2,3
Replicated Hash Table

Adding nodes
First, set up N1

Node N0

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 0,1

Node N1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Node N2

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 2,3
Replicated Hash Table

Adding nodes
Next, copy data to N1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 0,1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 2,3
Replicated Hash Table

Adding nodes
Next, change control

Node N0

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 0,1

Node N1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 1

Node N2

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 2,3
### Replicated Hash Table

**Adding nodes**
Next, remove data from N0

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

**Data for keys that hash to 0**

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

**Data for keys that hash to 1**

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

**Data for keys that hash to 2,3**

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>
Replicated Hash Table

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

Node N0

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 0

Node N1

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N1</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 1

Node N2

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 2,3
## Adding nodes

What about inserts while adding a node (e.g., during copy)?

Apply at $N_0$ then copy? Apply at both? Redirect to $N_1$?

### Replicated Hash Table

--

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 0,1

---

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 1

---

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N0</td>
</tr>
<tr>
<td>1</td>
<td>N0</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
</tr>
</tbody>
</table>

Data for keys that hash to 2,3

---

Data copy
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>g</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>14</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
</tr>
<tr>
<td>13</td>
<td>c</td>
</tr>
<tr>
<td>25</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>f</td>
</tr>
<tr>
<td>12</td>
<td>d</td>
</tr>
<tr>
<td>51</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>y</td>
</tr>
</tbody>
</table>
Neighborhood Search

Example

Node N1: N1.lookup(13), TTL = 4
Neighborhood Search

Example

Node N1

TTL = 1

N1.lookup(13), TTL = 4
Answer so far = { c, f, x }
Neighborhood Search

Example

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)
Neighborhood Search

Example

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

Node N1 [77, 4]
Neighborhood Search

Example

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

Do not reprocess 77
Neighborhood Search

Bootstrapping

Bootstrap server

Known nodes
$S = N_1, N_2, ...$

Add to $S$

Get neighbors
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
Neighborhood Search

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

- Performance
  - 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