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*
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?

Node 1
Resources $R_{1,1}, R_{1,2}, ...$

Node 2
Resources $R_{2,1}, R_{2,2}, ...$

Node 3
Resources $R_{3,1}, R_{3,2}, ...$
Peer-to-Peer Systems

Search

Query: Who has X?

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}, ...

Answers
Peer-to-Peer Systems

Search

Query
Who has X?

Node 1
Resources $R_{1,1}, R_{1,2}, ...$

Node 2
Resources $R_{2,1}, R_{2,2}, ...$

Node 3
Resources $R_{3,1}, R_{3,2}, ...$

Answers

Request resource

Provide resource
Distributed Lookup

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

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

<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>

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

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

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

Hash function \( H \) is “good”
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

$m = 6$
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
Successor/predecessor links

N1 \text{.pred} \quad N1 \text{.succ}
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 K52

N51.find_succ(K52) = N56

N14.find_succ(K52)
Chord

Finger table

N56
N51
N48
N42
N38
N32
N8
N1
N21
N14

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>

$m = 6$
Chord

Finger table

Finger table for N8

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N8 + 1</td>
<td>N14</td>
</tr>
<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>

Node that owns key 8 + 32 = 40

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

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

m = 6
Chord

Example
Looking up K54

\[ N51.\text{find\_succ}(K54) \]
\[ N42.\text{find\_succ}(K54) \]
Chord

Example
Looking up K54

\[ m = 6 \]

N51.find_succ(K54)
N42.find_succ(K54)
N8.find_succ(K54)
Chord

Adding nodes

Need to
1. Update links
2. Move data

For now, assume nodes never die
Chord

Adding nodes

\texttt{X.join(Y):}
\begin{itemize}
  \item \texttt{// Node Y is known to belong to the circle}
  \item \texttt{pred := nil;}
  \item \texttt{succ := Y.find_succ(X);} 
\end{itemize}
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 $N_x$.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 \((N_p, N_x]\)

\(N_x\)

Keys in \((N_p, N_s]\)

\(N_p\)

\(N_s\)
Chord

Moving data
After \( N_s.notify(N_x) \)

Send all keys in \((N_p, N_x]\) when \( N_s.pred \) gets updated
Moving data

Revised \texttt{notify()}

\texttt{X.notify(Y):}
\begin{itemize}
  \item if \texttt{pred = nil} or \texttt{Y in (pred, X)}
    \begin{itemize}
      \item \texttt{Y.add(data in (pred, Y)}
      \item \texttt{temp := pred}
      \item \texttt{pred := Y}
    \end{itemize}
  \end{itemize}
  \item \texttt{X.remove(data in (temp, Y)}
\end{itemize}

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 $(N_p, N_x]$ 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 $\sim(\log n)/2$

$n = \text{number of live nodes}$
Chord

Node failures

Assume $N_x$ dies

Links become invalid

Data gets lost
Chord

Node failures
Fixing links

\[
X.\text{check}_\text{pred}():
\]
\[
\quad \text{if pred failed}
\]
\[
\quad \quad \text{pred := nil}
\]

Also, keep \textit{backup} links to \(s > 1\) successors
Chord

Node failure example
State before failure

backup succ link (s = 2)
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

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

Data for keys that hash to 0,1

Data copy

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

Data for keys that hash to 1

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

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

Adding nodes
Next, change control

<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
Adding nodes
Next, remove data from N0

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>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
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

N₁
Adding nodes

What about inserts while adding a node (e.g., during copy)? Apply at $N_0$ then copy? Apply at both? Redirect to $N_1$?

Node $N_0$

<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>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 $N_1$

<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

Node $N_2$

<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

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

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)
Neighborhood Search

Example

TTL = 1
N1.lookup(13), TTL = 4, id = 77

Do not reprocess 77
Neighborhood Search

Example

Node N1 [77, 4]

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

Bootstrapping

Bootstrap server

Known nodes
$S = N_1, N_2, \ldots$

Get neighbors
Add to $S$
Neighborhood Search

Problems

Unnecessary messages

High load and traffic

\[ \text{E.g., if nodes have } p \text{ neighbors, each search } \sim p^{\text{TTL}} \text{ messages} \]

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

- Availability
  - Efficiency
  - Load balancing

- Authenticity
  - DoS prevention

- Incentives
  - Anonymity

- Performance

- Correctness

- Participation
Peer-to-Peer Systems Summary

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