Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Amir H. Payberah (amir@sics.se)
Recap
Distributed Hash Tables (DHT)

- An ordinary hash-table, which is ...

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatemeh</td>
<td>Stockholm</td>
</tr>
<tr>
<td>Sarunas</td>
<td>Lausanne</td>
</tr>
<tr>
<td>Tallat</td>
<td>Islamabad</td>
</tr>
<tr>
<td>Cosmin</td>
<td>Bucharest</td>
</tr>
<tr>
<td>Seif</td>
<td>Stockholm</td>
</tr>
<tr>
<td>Amir</td>
<td>Tehran</td>
</tr>
</tbody>
</table>
Distributed Hash Tables (DHT)

- An ordinary hash-table, which is distributed.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fatemeh</td>
<td>Stockholm</td>
</tr>
<tr>
<td>Sarunas</td>
<td>Lausanne</td>
</tr>
<tr>
<td>Tallat</td>
<td>Islamabad</td>
</tr>
<tr>
<td>Cosmin</td>
<td>Bucharest</td>
</tr>
<tr>
<td>Seif</td>
<td>Stockholm</td>
</tr>
<tr>
<td>Amir</td>
<td>Tehran</td>
</tr>
</tbody>
</table>
Distributed Hash Tables (DHT)

1. Decides on common key space for nodes and values
2. Connects the nodes smartly
3. Make a strategy for assigning items to nodes

Set of nodes: 12, 25, 7
Key of nodes: 12, 25, 7

Set of items: 2, 14, 31
Key of items: 2, 14, 31

4th April 2011
Chord an Example of DHT
How to Construct a DHT (Chord)?

- Use a **logical name space**, called the **identifier space**, consisting of identifiers \( \{0,1,2,\ldots, N-1\} \)

- Identifier space is a **logical ring** modulo \( N \).
How to Construct a DHT (Chord)?

- Use a **logical name space**, called the **identifier space**, consisting of identifiers \{0,1,2,..., N-1\}

- Identifier space is a **logical ring** modulo N.

- Every node picks a random identifier though Hash \( H \).

- Example:
  - Space \( N=16 \) \{0,...,15\}
  - Five nodes a, b, c, d, e
  - \( H(a) = 6 \)
  - \( H(b) = 5 \)
  - \( H(c) = 0 \)
  - \( H(d) = 11 \)
  - \( H(e) = 2 \)
The successor of an identifier is the first node met going in clockwise direction starting at the identifier.
The **successor** of an identifier is the first node met going in **clockwise direction** starting at the identifier.

**succ(x)**: is the first node on the ring with id greater than or equal x.
- **Succ(12) = 0**
- **Succ(1) = 2**
- **Succ(6) = 6**
Connect the Nodes

- Each node points to its successor.
  - The successor of a node $n$ is $\text{succ}(n+1)$.
  - 0’s successor is $\text{succ}(1) = 2$
  - 2’s successor is $\text{succ}(3) = 5$
  - 5’s successor is $\text{succ}(6) = 6$
  - 6’s successor is $\text{succ}(7) = 11$
  - 11’s successor is $\text{succ}(12) = 0$
Where to Store Data?

- Use globally known hash function, $H$.

- Each item $<\text{key}, \text{value}>$ gets identifier $H(\text{key}) = k$.
  
  - $H(\text{"Fatemeh"}) = 12$
  - $H(\text{"Cosmin"}) = 2$
  - $H(\text{"Seif"}) = 9$
  - $H(\text{"Sarunas"}) = 14$
  - $H(\text{"Tallat"}) = 4$
Where to Store Data?

- Use globally known hash function, $H$.
- Each item $<\text{key}, \text{value}>$ gets identifier $H(\text{key}) = k$.

- $H("\text{Fatemeh}") = 12$
- $H("\text{Cosmin}") = 2$
- $H("\text{Seif}") = 9$
- $H("\text{Sarunas}") = 14$
- $H("\text{Tallat}") = 4$
Where to Store Data?

- Use globally known hash function, $H$.

- Each item $<key,value>$ gets identifier $H(key) = k$.
  - $H(\text{"Fatemeh"}) = 12$
  - $H(\text{"Cosmin"}) = 2$
  - $H(\text{"Seif"}) = 9$
  - $H(\text{"Sarunas"}) = 14$
  - $H(\text{"Tallat"}) = 4$

- Store each item at its successor.
Where to Store Data?

- Use globally known hash function, $H$.
- Each item $<\text{key, value}>$ gets identifier $H(\text{key}) = k$.
  - $H(\text{"Fatemeh"}) = 12$
  - $H(\text{"Cosmin"}) = 2$
  - $H(\text{"Seif"}) = 9$
  - $H(\text{"Sarunas"}) = 14$
  - $H(\text{"Tallat"}) = 4$
- Store each item at its successor.
Lookup?
Look up a key $k$

- Calculate $H(k)$
- Follow succ pointers until item $k$ is found

Diagram:

- Node 0
- Node 1
- Node 2
- Node 3
- Node 4
- Node 5
- Node 6
- Node 7
- Node 8
- Node 9
- Node 10
- Node 11
- Node 12
- Node 13
- Node 14
- Node 15
Lookup?

- To lookup a key k
  - Calculate $H(k)$
  - Follow succ pointers until item k is found

- Example
  - Lookup "Seif" at node 2
  - $H("Seif") = 9$
  - Traverse nodes:
    - 2, 5, 6, 11 (BINGO)
  - Return "Stockholm" to initiator

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seif</td>
<td>Stockholm</td>
</tr>
</tbody>
</table>
// ask node n to find the successor of id

procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else // forward the query around the circle
        return successor.findSuccessor(id)
}

- (a, b] the segment of the ring moving clockwise from but not including a until and including b.
- n.foo(.) denotes an RPC of foo(.) to node n.
- n.bar denotes and RPC to fetch the value of the variable bar in node n.
Put and Get

```java
procedure n.put(id, value) {
    s = findSuccessor(id)
    s.store(id, value)
}
```

```java
procedure n.get(id) {
    s = findSuccessor(id)
    return s.retrieve(id)
}
```

- PUT and GET are nothing but lookups!!
Any improvement?
Improvement

- Any improvement?
  - Speeding up lookups

- If only pointer to $\text{succ}(n+1)$ is used
  - Worst case lookup time is $N$, for $N$ nodes
Speeding up Lookups

- **Finger/routing table:**
  - Point to $\text{succ}(n+1)$
  - Point to $\text{succ}(n+2)$
  - Point to $\text{succ}(n+4)$
  - Point to $\text{succ}(n+8)$
  - ... $\text{Point to } \text{succ}(n+2^m)$

- **Distance always halved to the destination.**
Speeding up Lookups

- Size of routing tables is logarithmic:
  - Routing table size: $M$, where $N = 2^M$.

- Every node $n$ knows $\text{successor}(n + 2^{i-1})$ for $i = 1 \ldots M$

- Routing entries $= \log_2(N)$
  - $\log_2(N)$ hops from any node to any other node

- Example: $\log_2(1000000) \approx 20$
DHT Lookup

// ask node n to find the successor of id

procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else // forward the query around the circle
        return successor.findSuccessor(id)
}
// ask node n to find the successor of id

procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else // forward the query around the circle
        return successor.findSuccessor(id)
}
DHT Lookup

// ask node n to find the successor of id
procedure n.findSuccessor(id) {
  if (predecessor \neq nil and id \in (predecessor, n]) then return n
  else if (id \in (n, successor]) then
    return successor
  else {
    // forward the query around the circle
    m := closestPrecedingNode(id)
    return m.findSuccessor(id)
  }
}

// search locally for the highest predecessor of id
procedure closestPrecedingNode(id) {
  for i = m downto 1 do {
    if (finger[i] \in (n, id)) then
      return finger[i]
  }
  return n
}
procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else { // forward the query around the circle
        m := closestPrecedingNode(id)
        return m.findSuccessor(id)
    }
}
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
    return successor
  else {
    // forward the query around the circle
    m := closestPrecedingNode(id)
    return m.findSuccessor(id)
  }
}
procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else { // forward the query around the circle
        m := closestPrecedingNode(id)
        return m.findSuccessor(id)
    }
}
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
    return successor
  else { // forward the query around the circle
    m := closestPrecedingNode(id)
    return m.findSuccessor(id)
  }
}
procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else { // forward the query around the circle
        m := closestPrecedingNode(id)
        return m.findSuccessor(id)
    }
}
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
    return successor
  else {
    // forward the query around the circle
    m := closestPrecedingNode(id)
    return m.findSuccessor(id)
  }
}
procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
    else if (id ∈ (n, successor]) then
        return successor
    else {
      // forward the query around the circle
      m := closestPrecedingNode(id)
      return m.findSuccessor(id)
    }
}
Discussion

- We are basically done.

- But …

- What about joins and failures/leaves?
  - Nodes come and go as they wish.

- What about data?
  - Should I lose my doc because some kid decided to shut down his machine and he happened to store my file?

- So actually we just started …
Handling Dynamism?
Ring Maintenance?
Handling Dynamism - Ring Maintenance

• Everything depends on successor pointers.

• In Chord, in addition to the successor pointer, every node has a predecessor pointer as well for ring maintenance.
  - Predecessor of node $n$ is the first node met in anti-clockwise direction starting at $n-1$. 
Handling Dynamism - Ring Maintenance

- **Periodic stabilization** is used to make pointers eventually correct.
  - Try pointing `succ` to closest alive successor.
  - Try pointing `pred` to closest alive predecessor.
**Handling Dynamism - Ring Maintenance**

- **Periodic stabilization** is used to make pointers eventually correct.
  - Try pointing `succ` to closest alive successor.
  - Try pointing `pred` to closest alive predecessor.

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
  set succ := v
send a notify(n) to succ
```
**Handling Dynamism - Ring Maintenance**

- **Periodic stabilization** is used to make pointers eventually correct.
  - Try pointing `succ` to closest alive successor.
  - Try pointing `pred` to closest alive predecessor.

```plaintext
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
  set succ := v
send a notify(n) to succ
```

```plaintext
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
  set pred := p
```

![Diagram of ring maintenance](image)
Handling Join?
• When $n$ joins:
  - Find $n$’s successor with $\text{lookup}(n)$
  - Set $\text{succ}$ to $n$’s successor
  - Stabilization fixes the rest

// Periodically at $n$:
$v := \text{succ}.\text{pred}$
if ($v \neq \text{nil}$ and $v \in (n,\text{succ}]$) then
  set $\text{succ} := v$
send a notify($n$) to $\text{succ}$

// When receiving notify($p$) at $n$:
if ($\text{pred} = \text{nil}$ or $p \in (\text{pred}, n]$) then
  set $\text{pred} := p$
• When \( n \) joins:
  - Find \( n \)'s successor with \( \text{lookup}(n) \)
  - Set \( \text{succ} \) to \( n \)'s successor
  - Stabilization fixes the rest

// Periodically at \( n \):
\[
v := \text{succ}.\text{pred}
\]
if \((v \neq \text{nil} \text{ and } v \in \langle n, \text{succ} \rangle)\) then
  set \( \text{succ} := v \)
send a notify\((n)\) to \( \text{succ} \)

// When receiving notify\((p)\) at \( n \):
if \((\text{pred} = \text{nil} \text{ or } p \in \langle \text{pred}, n \rangle)\) then
  set \( \text{pred} := p \)
When \( n \) joins:
- Find \( n \)'s successor with \( \text{lookup}(n) \)
- Set \( \text{succ} \) to \( n \)'s successor
- Stabilization fixes the rest

// Periodically at \( n \):
\[ v := \text{succ}.\text{pred} \]
\[ \text{if } (v \neq \text{nil} \text{ and } v \in (n, \text{succ}]) \text{ then} \]
\[ \text{set } \text{succ} := v \]
\[ \text{send a notify}(n) \text{ to succ} \]

// When receiving notify\((p)\) at \( n \):
\[ \text{if } (\text{pred} = \text{nil} \text{ or } p \in (\text{pred}, n]) \text{ then} \]
\[ \text{set } \text{pred} := p \]
When $n$ joins:
- Find $n$'s successor with $\text{lookup}(n)$
- Set $\text{succ}$ to $n$'s successor
- Stabilization fixes the rest

// Periodically at $n$:
$v := \text{succ}.\text{pred}$
if ($v \neq \text{nil}$ and $v \in (n, \text{succ})$) then
    set $\text{succ} := v$
send a $\text{notify}(n)$ to $\text{succ}$

// When receiving $\text{notify}(p)$ at $n$:
if ($\text{pred} = \text{nil}$ or $p \in (\text{pred}, n)$) then
    set $\text{pred} := p$
When $n$ joins:
- Find $n$'s successor with $\text{lookup}(n)$
- Set $\text{succ}$ to $n$'s successor
- Stabilization fixes the rest

// Periodically at $n$:
$v := \text{succ}.\text{pred}$
if ($v \neq \text{nil}$ and $v \in (n, \text{succ}]$) then
set $\text{succ} := v$
send a notify($n$) to $\text{succ}$

// When receiving notify($p$) at $n$:
if ($\text{pred} = \text{nil}$ or $p \in (\text{pred}, n]$) then
set $\text{pred} := p$
Fix Fingers?
Chord – Fix Fingers

• Periodically refresh finger table entries, and store the index of the next finger to fix.

• Local variable next initially is 0.

// When receiving notify(p) at n:
procedure n.fixFingers() {
    next := next+1
    if (next > m) then
        next := 1
    finger[next] := findSuccessor(n ⊕ 2^(next - 1))
}
Current situation: succ(N48) is N60.

\[ \text{Succ}(21 \oplus 2^{(6-1)}) = \text{Succ}(53) = N60. \]
Chord – Fix Fingers (2/4)

- \( \text{Succ}(21 \oplus 2^{(6-1)}) = \text{Succ}(53) = ? \)
- New node N56 joins and stabilizes successor pointer.
- Finger 6 of node N21 is wrong now.
- N21 eventually try to fix finger 6 by looking up 53 which stops at N48, however and nothing changes.
Chord – Fix Fingers (3/4)

- $\text{Succ}(21 \oplus 2^{(6-1)}) = \text{Succ}(53) = ?$
- N48 will eventually stabilize its successor.
- This means the ring is correct now.
Chord – Fix Fingers (4/4)

- $\text{Succ}(21 \oplus 2^{(6-1)}) = \text{Succ}(53) = N_{56}$
- When N21 tries to fix Finger 6 again, this time the response from N48 will be correct and N21 corrects the finger.
Handling Failure?
A node has a **successors list** of size $r$ containing the immediate $r$ successors:

- $\text{succ}(n+1)$
- $\text{succ}(\text{succ}(n+1)+1)$
- $\text{succ}(\text{succ}(\text{succ}(n+1)+1)+1)$

**How big should** $r$ **be?**

- $\log(N)$
// join a Chord ring containing node m
procedure n.join(m) {
    pred := nil
    Succ := m.findSuccessor(n)
    updateSuccessorList(succ.successorList)
}

// Periodically at n
procedure n.stabilize() {
    succ := find first alive node in successor list
    v := succ.pred
    if (v ≠ nil and v ∈ (n,succ)) then
        set succ := v
        send a notify(n) to succ
        updateSuccessorList(succ.successorList)
    }
Dealing with Failures

• Periodic stabilization

• If successor fails
  ▪ Replace with closest alive successor

• If predecessor fails
  ▪ Set pred to nil
Chord – Handling Failure (1/5)

- When \( n \) leaves, just disappear (like failure).
- When \( \text{pred} \) detected failed, set \( \text{pred} \) to \( \text{nil} \).
- When \( \text{succ} \) detected failed, set \( \text{succ} \) to closest alive in successor list.

```plaintext
// Periodically at \( n \):
\( v := \text{succ.pre}d \)
if \( (v \neq \text{nil} \land v \in (n,\text{succ})] \) then
  set \( \text{succ} := v \)
send a notify(\( n \)) to \( \text{succ} \)
```

```plaintext
// When receiving notify(\( p \)) at \( n \):
if \( (\text{pred} = \text{nil} \lor p \in (\text{pred}, n]) \) then
  set \( \text{pred} := p \)
```

```plaintext
procedure n.checkPredecessor() {
  if predecessor has failed then
    predecessor := \( \text{nil} \)
}
```
Chord – Handling Failure (2/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

// Periodically at n:
\[
\text{v := succ.pred}
\]
\[
\text{if (v \neq \text{nil} \text{ and } v \in (n, \text{succ}]) then}
\]
\[
\text{set succ := v}
\]
\[
\text{send a notify(n) to succ}
\]

// When receiving notify(p) at n:
\[
\text{if (pred = \text{nil} \text{ or } p \in (\text{pred}, n]) then}
\]
\[
\text{set pred := p}
\]

procedure n.checkPredecessor() {
  if predecessor has failed then
    predecessor := nil
}

15
11
Chord – Handling Failure (3/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

// Periodically at n:

```plaintext
v := succ.pred
if (v \neq nil and v \in (n, succ]) then
    set succ := v
send a notify(n) to succ
```

// When receiving notify(p) at n:

```plaintext
if (pred = nil or p \in (pred, n]) then
    set pred := p
```

```
procedure n.checkPredecessor() {
    if predecessor has failed then
        predecessor := nil
}
```
Chord – Handling Failure (4/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

// Periodically at n:
  v := succ.pred
  if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
  send a notify(n) to succ

// When receiving notify(p) at n:
  if (pred = nil or p ∈ (pred, n]) then
    set pred := p

procedure n.checkPredecessor() {
  if predecessor has failed then
    predecessor := nil
}
Chord – Handling Failure (5/5)

- When n leaves Just disappear (like failure).
- When pred detected failed Set pred to nil.
- When succ detected failed Set succ to closest alive in successor list.

// Periodically at n:
\[v := \text{succ}.\text{pred}\]
if (\(v \neq \text{nil}\) and \(v \in (n,\text{succ}]\)) then
\[\text{set succ} := v\]
send a notify(n) to succ

// When receiving notify(p) at n:
\[\text{if (pred = nil or p} \in (\text{pred, n}]\) then\]
\[\text{set pred} := p\]

procedure \(n.\text{checkPredecessor}()\) {
    if \(\text{predecessor has failed}\) then
    \(\text{predecessor} := \text{nil}\)
}
Variations of Chord
Variations of Chord

- Chord#
- DKS
Chord#

- The routing table has exponentially increasing pointers on the ring (node space) and **NOT** the identifier space.

\[
p_{i} = \begin{cases} 
    \text{successor} & : i = 0 \\
    p_{i-1} \cdot p_{i-1} & : i \neq 0 
\end{cases}
\]
DKS

- Generalization of Chord to provide arbitrary arity

- Provide $\log_k(n)$ hops per lookup
  - $k$ being a configurable parameter
  - $n$ being the number of nodes

- Instead of only $\log_2(n)$
DKS – Lookup

- Achieving $\log_k(n)$ lookup
- Each node contains $\log_k(N) = L$ levels, $N = k^L$
- Each level contains $k$ intervals,
- Example, $k=4$, $N=16$ ($4^2$), node 0

<table>
<thead>
<tr>
<th>Node 0</th>
<th>I0</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>0 ... 3</td>
<td>4 ... 7</td>
<td>8 ... 11</td>
<td>12 ... 15</td>
</tr>
</tbody>
</table>
DKS – Lookup

- Achieving $\log_k(n)$ lookup
- Each node contains $\log_k(N)=L$ levels, $N=k^L$
- Each level contains $k$ intervals,
- Example, $k=4$, $N=16$ ($4^2$), node 0

<table>
<thead>
<tr>
<th>Node 0</th>
<th>I0</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>0 ... 3</td>
<td>4 ... 7</td>
<td>8 ... 11</td>
<td>12 ... 15</td>
</tr>
<tr>
<td>Level 2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
A Page to Remember
A Page to Remember

- **Pointer of the nodes:**
  - **Successor:** first clockwise node
  - **Predecessor:** first anti-clockwise node
  - **Finger list:** successor(n + 2^(i-1)) for i = 1... M (N = 2^M).

- **Handling dynamism**
  - **Periodic stabilization**

- **Handling failure**
  - **Successor list**
  - **Periodic stabilization**
Question?