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

[Slides by Amir H. Payberah (amir@sics.se), Jim Dowling]
Consistent Hashing

- Imagine we want to store information about books on 4 nodes (servers).
  - Use the ISBN to identify each book.
- We could use one of the nodes as a central directory server.
- But, with the hash of the ISBN, we don't need a central server:
  ```
  switch (SHA-1(ISBN) mod 4) {
    case 0: // store on node1
    case 1: // store on node2
    case 2: // store on node3
    case 3: // store on node4
  }
  ```
- Our store gets bigger......we need to add more 2 nodes.
  - We now have to re-calculate where all the books are stored.
- Do the books stay on the same nodes?
- The only books stored on the same node as before are those where
  ```
  ```
Consistent Hashing

• Consistent hashing allows you to add more nodes and only a small minority of books will have to move to new nodes.

• **Key property:** *low cost hash-table expansion.* That is, a book's hash key is independent of the number of books and independent of the number of nodes.
  - If you add or remove nodes or books, a book's hash key remains the same.

• **Mechanism:** hash something constant at each node
  - E.g., a node’s MAC address

See: Karger et. Al, "Consistent Hashing and Random Trees...”
Consistent Hashing

- Each node is responsible for all books with hash keys between its own hash key and the hash key of the next node (going upwards).

- Imagine we have books with SHA-1(ISBN) in a range 0..16
  - For node1..node4, the nodes' hash keys are:
    - \{node1→0, node2→6, node3→11, node4→16\}

- So, a book with SHA-1(ISBN) → 1 would be stored at node1.
  - (node1) 0 < 1 (book) < 6 (node2)

- Now if we add new nodes positions 4 and 8, respectively:
  - Nodes have hash keys: \{0, 4, 6, 8, 11, 16\}

- Fewer books need to be moved
  - Books with hash keys (6..7) get moved from node2 to the first new node
  - Books with hash keys (8..10) get moved from node3 to the second new node
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. Decide on a common key space for nodes and values

   - Set of nodes
   - Key of nodes

2. Connect nodes using a small, bounded number of links s.t. max hop count is minimized

   - Set of items
   - Key of items

3. Define a strategy for assigning items to nodes
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,..., 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.
  - $\text{Succ}(12) = 0$
  - $\text{Succ}(1) = 2$
  - $\text{Succ}(6) = 6$
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 `<key, value>` gets identifier $H(key) = k$.

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

• Use globally known hash function, $H$.

• Each item $\langle \text{key}, \text{value} \rangle$ 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 $H(\text{key}) = k$.
  - $H(\text{"Fatemeh"}) = 12$
  - $H(\text{"Cosmin"}) = 2$
  - $H(\text{"Seif"}) = 9$
  - $H(\text{"Sarunas"}) = 14$
  - $H(\text{"Tallat"}) = 4$
- Store each item at its successor.
Where to Store Data?

- Use globally known hash function, $H$.

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

- Store each item at its successor.
Lookup?
Lookup?

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

• (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)
}

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

- PUT and GET are nothing but lookups!!
How can we improve this?
Cost of Lookup Operations

- If only the pointer to $\text{succ}(n+1)$ is used
  - Worst case lookup time is $O(N)$, 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 $\text{succ}(n+8)$
  - ...
  - Point to $\text{succ}(n+2^{M-1})$

• 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 \( \text{successor}(n + 2^{\text{(i-1)}}) \) 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)
}
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)
}
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
        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] ∈ (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.
• **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:
\[
v := \text{succ}.\text{pred}
\]
if \((v \neq \text{nil} \text{ and } v \in (n, \text{succ}])\) then
  set \(\text{succ} := v\)
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
```
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 $n$:
$v := \text{succ.ppred}$
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 $\text{pred} := p$
Chord – Handling Join (2/5)

- **When n joins:**
  - Find n’s successor with `lookup(n)`
  - Set succ to n’s successor
  - Stabilization fixes the rest

// Periodically at n:

```plaintext
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:

```plaintext
if (pred = nil or p ∈ (pred, n]) then
  set pred := p
```
• When n joins:
  - Find n’s successor with \(\text{lookup}(n)\)
  - Set succ to n’s successor
  - Stabilization fixes the rest

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

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

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

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

// Periodically at \( n \):
\[
\begin{align*}
& v := \text{succ.prec} \\
& \text{if } (v \neq \text{nil} \text{ and } v \in (n, \text{succ}]) \text{ then} \\
& \hspace{1cm} \text{set succ} := v \\
& \hspace{1cm} \text{send a notify}(n) \text{ to succ}
\end{align*}
\]

// When receiving notify\((p)\) at \( n \):
\[
\begin{align*}
& \text{if } (\text{pred} = \text{nil} \text{ or } p \in (\text{pred}, n]) \text{ then} \\
& \hspace{1cm} \text{set pred} := p
\end{align*}
\]
Fixing Fingers
Chord – Fixing 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 – Fixing 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.
• $\text{Succ}(21 \oplus 2^{(6-1)}) = \text{Succ}(53) = ?$
• N48 will eventually stabilize its successor.
• This means the ring is correct now.
• $\text{Succ}(21 \oplus 2^{(6-1)}) = \text{Succ}(53) = N56$
• When N21 tries to fix Finger 6 again, this time the response from N48 will be correct and N21 corrects the finger.
Handling Failure?
Successor List

- 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?
  - \( \log(N) \)

![Diagram showing the successor list concept](image-url)
// 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.successorListList)
}
Dealing with Failures

- Periodic stabilization

- If successor fails
  - Replace with closest alive successor

- If predecessor fails
  - Set pred to nil
• 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 (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 \( n \):
\[
\begin{align*}
v &:= \text{succ.pred} \\
\text{if } (v \neq \text{nil} \text{ and } v \in (n, \text{succ}]) \text{ then} \\
&\quad \text{set succ := } v \\
&\quad \text{send a notify}(n) \text{ to succ}
\end{align*}
\]

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

procedure n.checkPredecessor() {
    if predecessor has failed then
        predecessor := nil
}
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:
ν := succ.pred
if (ν ≠ nil and ν ∈ (n,succ]) then
    set succ := ν
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 (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:
\[
v := \text{succ.prec}
\]
if \((v \neq \text{nil} \text{ and } v \in (n, \text{succ}])\) then
  set succ := v
send a notify(n) to succ

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

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

\[
\text{pointer}_i = \begin{cases} 
\text{successor} & : i = 0 \\
\text{pointer}_{i-1} \cdot \text{pointer}_{i-1} & : i \neq 0
\end{cases}
\]
Chord vs. Chord#

Chord

0 → 1 → 2
7 → 6 → 5
13 → 12

Chord#

0 → 1
14 → 2
3 → 4
6 → 5
9 → 8
10 → 7
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>
Summary
Summary

• 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