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

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashock, Frank Dabek, Hari Balakrishnan

March 4, 2013
Problem
In a peer-to-peer network, how does one efficiently locate a node which is storing a desired data item?

Solution
**Chord:** A scalable, distributed protocol which efficiently locates the desired node in such a dynamic network.
Other efforts in the same direction

<table>
<thead>
<tr>
<th>DNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. While DNS requires special root servers, Chord has no such requirement.</td>
</tr>
<tr>
<td>3. DNS works best when hostnames are structured to reflect administrative boundaries. Chord imposes no naming structure.</td>
</tr>
</tbody>
</table>

presented by Durgesh Samant
### DNS

1. While DNS requires special root servers, Chord has no such requirement.
3. DNS works best when hostnames are structured to reflect administrative boundaries. Chord imposes no naming structure.

### Napster, Gnutella, DC++

1. Napster & DC++ use a central index. This leads to a single point of failure.
2. Gnutella floods the entire network with each query.
3. No *keyword* search in Chord. Only unique IDs.
Content Addressable Network (CAN)

Problem Identification

**Scalability Bottleneck** :- Centralized hash table
Content Addressable Network (CAN)

Problem Identification

**Scalability Bottleneck** :- Centralized hash table

**Scheme**

- $d$-dimensional co-ordinate space
- Completely logical. Has **no** bearing with physical co-ordinates.
- Map each Key *deterministically* to a point $P$ using uniform hashing.
Problem Identification

**Scalability Bottleneck :-** Centralized hash table

**Scheme**

- $d$-dimensional co-ordinate space
- Completely logical. Has no bearing with physical co-ordinates.
- Map each Key *deterministically* to a point P using uniform hashing.
- Space creation. Bootstrapping.
Content Addressable Network (CAN)

Problem Identification

**Scalability Bottleneck** :- Centralized hash table

Scheme

- $d$-dimensional co-ordinate space
- Completely logical. Has **no** bearing with physical co-ordinates.
- Map each Key *deterministically* to a point $P$ using uniform hashing.
- Space creation. Bootstrapping.
- Node join/departures.

presented by Durgesh Samant

Chord
Problem Identification

**Scalability Bottleneck :-** Centralized hash table

**Scheme**

- \( d \)-dimensional co-ordinate space
- Completely logical. Has **no** bearing with physical co-ordinates.
- Map each Key *deterministically* to a point \( P \) using uniform hashing.
- Space creation. bootstrapping.
- Node join/departures.
- Message routing.

presented by Durgesh Samant | Chord
### Problem Identification

**Scalability Bottleneck** :- Centralized hash table

### Scheme

- $d$-dimensional co-ordinate space
- Completely logical. Has **no** bearing with physical co-ordinates.
- Map each Key *deterministically* to a point $P$ using uniform hashing.
- Space creation. Bootstrapping.
- Node join/departures.
- Message routing.

### Key Facts

- Info maintained by each node is independent of $N$
- How does one fix $d$?
Network Assumptions

1. **Symmetric:** If $A \rightarrow B$, then $B \rightarrow A$
2. **Trasitive:** If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$
Network Assumptions

1. **Symmetric**:- If \( A \rightarrow B \), then \( B \rightarrow A \)
2. **Transitive**:- If \( A \rightarrow B \) and \( B \rightarrow C \) then \( A \rightarrow C \)

Targets

- **Load Balance**:- Distributed hash function.
- **Decentralization**:- No node is more important than the other.
- **Scalable**:- Achieved without any parameter tuning.
- **Availability**:- Handles most network failures.
- **Flexible Naming**:- Flat and unstructured key space.
Network Assumptions

1. **Symmetric**: If $A \rightarrow B$, then $B \rightarrow A$
2. **Transitive**: If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

Targets

1. **Load Balance**: Distributed hash function.
Network Assumptions

1. **Symmetric** :- If $A \rightarrow B$, then $B \rightarrow A$

2. **Trasitive** :- If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

Targets

1. **Load Balance** :- Distributed hash function.

2. **Decentralization** :- No node is more important than the other.
Network Assumptions

1. **Symmetric:** If $A \rightarrow B$, then $B \rightarrow A$
2. **Transitive:** If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

Targets

1. **Load Balance:** Distributed hash function.
2. **Decentralization:** No node is more important than the other.
3. **Scalable:** Achieved without any parameter tuning.
Network Assumptions

1. **Symmetric**: If $A \rightarrow B$, then $B \rightarrow A$
2. **Transitive**: If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

Targets

1. **Load Balance**: Distributed hash function.
2. **Decentralization**: No node is more important than the other.
3. **Scalable**: Achieved without any parameter tuning.
4. **Availability**: Handles most network failures.
### Network Assumptions

1. **Symmetric:** If $A \rightarrow B$, then $B \rightarrow A$
2. **Transitive:** If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

### Targets

1. **Load Balance:** Distributed hash function.
2. **Decentralization:** No node is more important than the other.
3. **Scalable:** Achieved without any parameter tuning.
4. **Availability:** Handles most network failures.
5. **Flexible naming:** Flat and unstructured key space.
The big picture
Consistent Hashing

1. Assign an m bit identifier to each node and key separately.
2. Use SHA-1 to ensure keys are evenly distributed.
3. Chord ring: a $m \frac{2^m}{\text{identifier}}$ circle.

Theorem 1
- Each node responsible for $(1 + \epsilon) K / N$ keys.
- Only $O(\frac{K}{N})$ keys change hands when $(N + 1)$st node joins/leaves.

presented by Durgesh Samant
Consistent Hashing

How do you do it?

1. Assign an \( m \) bit identifier to each node and key separately.
2. Use SHA-1 to ensure keys are evenly distributed.
3. **Chord ring**: a \( 2^m \) identifier circle.

\[ m=6, \text{ 6 keys, 10 nodes} \]
Consistent Hashing

How do you do it?

1. Assign an \( m \) bit identifier to each node and key separately.

2. Use SHA-1 to ensure keys are evenly distributed.

3. **Chord ring**: a \( 2^m \) identifier circle.

\[
\text{m=6, 6 keys, 10 nodes}
\]

Theorem

1. Each node responsible for \((1 + \epsilon)K/N\) keys

2. Only \(O(K/N)\) keys change hands when \((N + 1)^{st}\) node joins/leaves.
Naive Key Lookup

Algorithm

// ask a node n to find the successor of id
n.find_successor(id)
if (id \belongs \(n, \text{successor}\])
    return successor;
else
    // forward the query around the circle
    return successor.find_successor(id);

Performance

\(O(N)\)

presented by Durgesh Samant
Scalable Key Lookup

- **Finger Table**: \( m \) entries, only \( O(\log(N)) \) are distinct
- \( i^{th} \) entry = *first* node that succeeds the current node by atleast \( 2^{i-1} \) on the identifier circle.
- \( n.\text{finger}[i] \), a.k.a. \( i^{th} \) finger of \( n \)
Scalable Key Lookup

- **Finger Table** :- \( m \) entries, only \( O(\log(N)) \) are distinct
- \( i^{th} \) entry = first node that succeeds the current node by atleast \( 2^{i-1} \) on the identifier circle.
- \( n\.finger[i] \), a.k.a. \( i^{th} \) finger of n
- **Successor** :- next node, \( n\.finger[1] \)
- **Predecessor** :- previous node, \( p\.finger[1]=n \)

presented by Durgesh Samant

Chord
Scalable Key Lookup

- **Finger Table**: \( m \) entries, only \( O(\log(N)) \) are distinct
- \( i^{th} \) entry = *first* node that succeeds the current node by at least \( 2^{i-1} \) on the identifier circle.
- \( n.finger[i] \), a.k.a. \( i^{th} \) finger of \( n \)
- **Successor**: next node, \( n.finger[1] \)
- **Predecessor**: previous node, \( p.finger[1]=n \)

**Important Observations**

1. Each node stores a small amount of info.
2. Each node knows more about closer nodes than far off ones.
3. A node’s finger table does not contain enough info to directly find the successor of any arbitrary node \( k \).
The Lookup Algorithm

N8 looks up K54

Algorithm

// ask a node n to find the successor of id
n.find_successor(id)
if(id \ belongs \ (n,successor) 
  return successor;
else
  n’=closest_preceding_node(id);
  return n’.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i= m down to 1
    if (finger[i] \ belongs \ (n,id))
      return finger[i];
  return n;

Theorem

The no. of nodes which need to be contacted are \(O(\log(N))\)
presented by Durgesh Samant
The Lookup Algorithm

N8 looks up K54

Algorithm

// ask a node n to find the successor of id
n.find_successor(id)
if(id \ belongs (n,successor) )
    return successor;
else
    n’=closest_preceding_node(id);
    return n’.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
for i= m down to 1
    if (finger[i] \ belongs (n,id))
        return finger[i];
return n;

Theorem

The no. of nodes which need to be contacted are $O(\log(N))$
Every node periodically runs the stabilize algo to learn about newly joined nodes.
The algo is, basically ask the successor for its predecessor $p$. Decide if $p$ should be its successor.
Thereby, the successor also gets a chance to check its predecessor.
Each node periodically fixes its finger table by essentially reconstructing it.
Similarly, each node periodically checks if its predecessor is alive. If it is not, then it initializes it to nil.
Every node periodically runs the stabilize algo to learn about newly joined nodes.

The algo is, basically ask the successor for its predecessor $p$. Decide if $p$ should be its successor.

Thereby, the successor also gets a chance to check its predecessor.

Each node periodically fixes its finger table by essentially reconstructing it.

Similarly, each node periodically checks if its predecessor is alive. If it is not, then it initializes it to $nil$.

**Theorem**

If any sequence of join operations are interleaved with stabilize, eventually, the successor pointers will form a cycle on all nodes in the network.
Impact of Node Joins on Lookups

Case 1: Finger table entries are reasonably correct: Theorem
The node is correctly located in $O(\log N)$ time.

Case 2: Successor pointers are correct, finger table inaccurate
Lookups will be correct. Just slower.

Case 3: Successor pointers incorrect
Lookup will fail. The high level application can try again after a small pause. It will not take time for the successor pointers to get fixed.

presented by Durgesh Samant Chord
Case 1: Finger table entries are reasonably correct: Theorem
The node is correctly located in $O(\log(N))$ time.
Case 1: Finger table entries are \textit{reasonably} correct: Theorem
The node is correctly located in $O(\log(N))$ time.

Case 2: Successor pointers are correct, finger table inaccurate
Lookups will be correct. Just slower.
Impact of Node Joins on Lookups

Case 1: Finger table entries are **reasonably** correct: Theorem
The node is correctly located in $O(\log(N))$ time.

Case 2: Successor pointers are correct, finger table inaccurate
Lookups will be correct. Just slower.

Case 3: Successor pointers incorrect
Lookup will fail. The high level application can try again after a small pause. It will not take time for the successor pointers to get fixed.
Invariant Assumed so far :- Each node knows its successor.

To increase Robustness, maintain a *successor list* containing $r$ successors.

Probability of all $r$ nodes concurrently failing $= p^r$
Failure and Replication

- Invariant Assumed so far: Each node knows its successor.
- To increase Robustness, maintain a *successor list* containing $r$ successors.
- Probability of all $r$ nodes concurrently failing = $p^r$

### Modified stabilize algorithm

- Copy successors list, remove the last entry and *prepend* the successor.
- If the successor has failed, do the above with the first *live* successor in own list.
Failure and Replication

- Invariant Assumed so far :- Each node knows its successor.
- To increase Robustness, maintain a successor list containing \( r \) successors.
- Probability of all \( r \) nodes concurrently failing = \( p^r \)

**Modified stabilize algorithm**
- Copy successors list, remove the last entry and *prepend* the successor.
- If the successor has failed, do the above with the first *live* successor in own list.

**Modified closest preceding node**
Search not just the finger table, but also the successor list for the most immediate successor of \( id \)
Robustness Guarantee

Theorem

If we use a successor list of length $r = \Omega(\log(N))$, in a network which is initially stable, and every node fails with probability 0.5, then with high probability \textit{find successor} returns the closest living successor to the query key.

Theorem

In a network which is initially stable, if every node fails with probability 0.5, then the expected time to execute \textit{find successor} is $O(\log(N))$.

presented by Durgesh Samant
Robustness Guarantee

**Theorem**

If we use a successor list of length $r=\Omega(\log(N))$, in a network which is initially stable, and every node fails with probability 0.5, then with high probability \textit{find\_successor} returns the closes living successor to the query key.
Robustness Guarantee

Theorem

If we use a successor list of length $r = \Omega(\log(N))$, in a network which is initially stable, and every node fails with probability 0.5, then with high probability `find_successor` returns the closest living successor to the query key.

Theorem

In a network which is initially stable, if every node fails with probability 0.5, then the expected time to execute `find_successor` is $O(\log(N))$. 
Voluntary Node Departures

- Treating a departure as a node failure is rather wasteful.
- A node which is about to leave may transfer its keys to its successor as it departs.
- It can also notify its predecessor and successor before departing.
- The predecessor can remove the node from its successor list and add the last node in the new successor list to its own successor list.
- Similarly, the departing node’s successor can update its predecessor to reflect the departure.
successor list size = 1
when the predecessor of a node changes, it notifies its old predecessor about its new predecessor
packet delay modelled with exponential distribution with mean 50ms.
node declared dead if it does not respond within 500ms.
not concerned with actual data. Lookup is considered successful if current successor has the desired key.
Load Balance

Without virtual nodes

With virtual nodes

Parameter Settings

- No. of nodes = $10^4$
- $10^5 \leq \text{No. of keys} \leq 10^6$
- Increments of $10^5$
- 20 runs per No. of keys

presented by Durgesh Samant

Chord
Path Length

- Node count = $2^k$
- Key count = $100 \times 2^k$
- $3 \leq k \leq 14$
- Picked a random set of keys
- Find query length
Improving Routing Latency

Nodes closer in identifier ring can be quite far in underlying network. Actual latency can be large although avg. path length is small. Maintain alternative nodes for each finger. Route the query to the one which is closest.

Topologies

1. 3-d space: The network distance is modeled as geometric distance in a 3-d space

2. Transit stub: A transit-stub topology with 5000 nodes. 50ms link latency for intra-transit domain links. 20ms, for transit-stub links and 1ms for intra-stub links

presented by Durgesh Samant
• Nodes closer in identifier ring can be quite far in underlying network.
• Nodes closer in identifier ring can be quite far in underlying network.
• Actual latency can be large although avg. path length is small.
Nodes closer in identifier ring can be quite far in underlying network.

Actual latency can be large although avg. path length is small.

Maintain alternative nodes for each finger
Nodes closer in identifier ring can be quite far in underlying network.

Actual latency can be large although avg. path length is small.

Maintain alternative nodes for each finger

Route the query to the one which is closest.
- Nodes closer in identifier ring can be quite far in underlying network.
- Actual latency can be large although avg. path length is small.
- Maintain alternative nodes for each finger.
- Route the query to the one which is closest.

**Topologies**

1. **3-d space**: The network distance is modeled as geometric distance in a 3-d space.
2. **Transit stub**: A transit-stub topology with 5000 nodes. 50ms link latency for intra-transit domain links. 20ms, for transit-stub links and 1ms for intra-stub links.
Summary

Major Contributions

Load Balance: Consistent hashing.

Decentralization: Each node knows about only $O(\log N)$ nodes for efficient lookup.

Scalability: Handles large number of nodes, joining and leaving the system.

Availability: Graceful performance degradation. Single correct info is enough.

Efficiency: Each node resolves lookups via $O(\log N)$ messages.

Possible extensions:
- Deal with network partitions
- Deal with adversarial/faulty nodes
## Summary

### Major Contributions

- **Load Balance** :- Consistent hashing.
## Summary

### Major Contributions

- **Load Balance**: Consistent hashing.
- **Decentralization**: Each node knows about only $O(\log(N))$ nodes for efficient lookup.

Possible extensions:
- Deal with network partitions
- Deal with adversarial/faulty nodes

Presented by Durgesh Samant
## Summary

### Major Contributions

- **Load Balance** :- Consistent hashing.
- **Decentralization** :- Each node knows about only $O(\log(N))$ nodes for *efficient* lookup
- **Scalability** :- Handles large number of nodes, joining and leaving the system.

### Possible extensions

- Deal with network partitions
- Deal with adversarial/faulty nodes

---

*Presented by Durgesh Samant*
Summary

Major Contributions

- **Load Balance** :- Consistent hashing.
- **Decentralization** :- Each node knows about only $O(\log(N))$ nodes for *efficient* lookup
- **Scalability** :- Handles large number of nodes, joining and leaving the system.
- **Availability** :- Graceful performance degradation : Single correct info is enough

Possible extensions:
- Deal with network partitions
- Deal with adversarial/faulty nodes

Presented by Durgesh Samant
Summary

Major Contributions

- **Load Balance** :- Consistent hashing.
- **Decentralization** :- Each node knows about only $O(\log(N))$ nodes for *efficient* lookup
- **Scalability** :- Handles large number of nodes, joining and leaving the system.
- **Availability** :- Graceful performance degradation : Single correct info is enough
- **Efficiency** :- Each node resolves lookups via $O(\log(N))$ messages

Possible extensions
- Deal with network partitions
- Deal with adversarial/faulty nodes
Summary

Major Contributions

- **Load Balance**: Consistent hashing.
- **Decentralization**: Each node knows about only $O(\log(N))$ nodes for *efficient* lookup.
- **Scalability**: Handles large number of nodes, joining and leaving the system.
- **Availability**: Graceful performance degradation: Single correct info is enough.
- **Efficiency**: Each node resolves lookups via $O(\log(N))$ messages.

Possible extensions

- Deal with network partitions
- Deal with adverserial/faulty nodes

presented by Durgesh Samant  |  Chord