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

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan

COEN317 2005
What is Chord?

• A peer-to-peer lookup service

• Solves problem of locating a data item in a collection of distributed nodes, considering frequent node arrivals and departures

• Core operation in most p2p systems is efficient location of data items

• Chord protocol supports just one operation: given a key, it maps the key onto a node

  – A peer-to-peer hash lookup:
    • Lookup(key) → IP address
    • Chord does not store the data
Characteristics

• Simplicity, \textit{provable} correctness, and \textit{provable} performance

• Each Chord node needs routing information about only a few other nodes

• Resolves lookups via messages to other nodes (iteratively or recursively)

• Maintains routing information as nodes join and leave the system
Mapping onto Nodes

• Traditional name and location services provide a direct mapping between keys and values

• What are examples of values? A value can be an address, a document, or an arbitrary data item

• Chord can easily implement a mapping onto values by storing each key/value pair at node to which that key maps
Napster, Gnutella vs. Chord

• Compared to Napster and its centralized servers, Chord avoids single points of control or failure by a decentralized technology

• Compared to Gnutella and its widespread use of broadcasts, Chord avoids the lack of scalability through a small number of important information for rounting
# DNS vs. Chord

<table>
<thead>
<tr>
<th>DNS</th>
<th>Chord</th>
</tr>
</thead>
<tbody>
<tr>
<td>provides a host name to IP address mapping</td>
<td>can provide same service: Name = key, IP = value</td>
</tr>
<tr>
<td>relies on a set of special root servers</td>
<td>requires no special servers</td>
</tr>
<tr>
<td>names reflect administrative boundaries</td>
<td>imposes no naming structure</td>
</tr>
<tr>
<td>is specialized to finding named hosts or services</td>
<td>can also be used to find data objects that are not tied to certain machines</td>
</tr>
</tbody>
</table>
Freenet vs. Chord

- both decentralized

- both automatically adapt when hosts leave and join

- Freenet
  - does not assign responsibility for documents to specific servers, instead lookups are searches for cached copies
  + allows Freenet to provide anonymity
  - prevents guaranteed retrieval of existing documents

- Chord
  - does not provide anonymity
  + but its lookup operation runs in predictable time and always results in success or definitive failure
Design Goals for Chord

• **Load balance**: distributed hash function, spreading keys evenly over nodes

• **Decentralization**: chord is fully distributed, no node more important than other, improves robustness

• **Scalability**: logarithmic growth of lookup costs with number of nodes in network, even very large systems are feasible
Design Goals for Chord (cont.)

- **Availability**: chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found.

- **Flexible naming**: no constraints on the structure of the keys and key-space is flat.
  - Namespace of Chord is a fixed length bit string ($m$ bit)
  - Each object is identified by a unique ID.
Example Application using Chord: Cooperative Mirroring

- Highest layer provides a file-like interface to user including user-friendly naming and authentication
- This file systems maps operations to lower-level block operations
- Block storage uses Chord to identify responsible node for storing a block and then talk to the block storage server on that node
Example Application using Chord: Chord-based DNS

- DNS provides a lookup service
  keys: host names values: IP addresses
  Chord could hash each host name to a key

- Chord-based DNS:
  - no special root servers
  - no manual management of routing information
  - no naming structure
  - can find objects not tied to particular machines
The Base Chord Protocol

- Specifies how to find the locations of keys
- How new nodes join the system
- How to recover from the failure or planned departure of existing nodes
Consistent Hashing

• Consistent Hash Functions are designed to allow nodes to enter and leave a network with little disruption to the rest of the network

• Hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1 (Secure Hash Algorithm)
  – ID(node) = hash(IP Address)
  – ID(key) = hash(key)
Consistent Hashing

- An identifier length $m$, must be large enough to make the probability of 2 nodes or keys mapping to the same identifier negligible.

- Properties of consistent hashing:
  - Function balances load: all nodes receive roughly the same number of keys.
  - When an Nth node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location.
How the Hashing Function actually works

• Assigning keys to nodes
  – Identifiers are arranged in an *identifier circle* modulo $2^m$
  – Key $k$ is assigned to the first node whose identifier is equal to or follows (the identifier of $k$ in the identifier space)
  – This node is called the *successor node* of key $k$
  – We denote this node as $\text{successor}(k)$
  – If we represent the identifiers as a circle of numbers from 0 to $(2^m)-1$, then $\text{successor}(k)$ is the first node clockwise from $k$
Successor Nodes

An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.
Node Joins and Departures

• When a node $n$ joins the network, certain keys previously assigned to $n$’s successor, now become assigned to $n$

• When node $n$ leaves the network, all of its assigned keys are reassigned to $n$’s successor
Node Joins and Departures

successor(6) = 7

successor(1) = 3
Scalable Key Location

- A very small amount of routing information suffices to implement consistent hashing in a distributed environment.

- Each node need only be aware of its successor node on the circle.

- Queries for a given identifier can be passed around the circle via these successor pointers.

- Resolution scheme correct, BUT inefficient: it may require traversing all N nodes!
Modification of Lookups

- Lookups are accelerated by maintaining additional routing information.

- Each node maintains a routing table with (at most) \( m \) entries (where \( N=2^m \)) called the **finger table**.

- The \( i^{th} \) entry in the table at node \( n \) contains the identity of the first node, \( s \), that succeeds \( n \) by at least \( 2^{i-1} \) on the identifier circle.

- \( s = \) successor\((n + 2^{i-1})\)

- \( s \) is called the \( i^{th} \) finger of node \( n \).
Finger Tables

```
<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[2,3)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>[3,5)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>[5,1)</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[4,5)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>[5,7)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Keys:
- 6
- 1
- 2
Characteristics of Finger Tables

• Each node stores information about only a small number of other nodes, and knows more about nodes closely following it than about nodes farther away.

• A node’s finger table generally does not contain enough information to determine the successor of an arbitrary key $k$.

• Repetitive queries to nodes that immediately precede the given key will lead to the key’s successor eventually.
Node Joins with Finger Tables

finger table

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
<td>6</td>
</tr>
</tbody>
</table>

keys = 6

finger table

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[2,3)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>[3,5)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>[5,1)</td>
<td>6</td>
</tr>
</tbody>
</table>

keys = 1

finger table

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[4,5)</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>[5,7)</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
<td>0</td>
</tr>
</tbody>
</table>

keys = 2
Node Departures with Finger Tables

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,2)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[2,3)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>[3,5)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>[5,1)</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[4,5)</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>[5,7)</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start</th>
<th>int.</th>
<th>succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>[7,0)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,2)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>[2,6)</td>
<td>3</td>
</tr>
</tbody>
</table>
Concurrent Operations and Failures

- Basic “stabilization” protocol is used to keep nodes’ successor pointers up to date, which is sufficient to guarantee correctness of lookups.

- Those successor pointers can then be used to verify the finger table entries.

- Every node runs *stabilize* periodically to find newly joined nodes.

- Updates finger tables and successor pointers.
Node joins and stabilization

Stabilization protocol:

- **Stabilize()**: n asks its successor for its predecessor p and decides whether p should be n‘s successor instead (this is the case if p recently joined the system).
- **Notify()**: notifies n‘s successor of its existence, so it can change its predecessor to n
- **Fix_fingers()**: updates finger tables
Node joins and stabilization
Node joins and stabilization

- N26 joins the system
- N26 acquires N32 as its successor
- N26 notifies N32
- N32 acquires N26 as its predecessor
Node joins and stabilization

- N26 copies keys
- N21 runs stabilize() and asks its successor N32 for its predecessor which is N26.
Node joins and stabilization

- N21 acquires N26 as its successor
- N21 notifies N26 of its existence
- N26 acquires N21 as its predecessor
Failure Recovery

• The key step in failure recovery is maintaining correct successor pointers

• To help achieve this, each node maintains a successor-list of its $r$ nearest successors on the ring

• If node $n$ notices that its successor has failed, it replaces it with the first live entry in the list

• $stabilize$ will correct finger table entries and successor-list entries pointing to failed node

• Performance is sensitive to the frequency of node joins and leaves versus the frequency at which the stabilization protocol is invoked
The Mathematical Analysis of Chord

• Every node is responsible for about $K/N$ keys (N nodes, K keys)

• When a node joins or leaves an N-node network, only $O(K/N)$ keys change hands (and only to and from joining or leaving node)

• Lookups need $O(\log N)$ messages

• To reestablish routing invariants and finger tables after node joining or leaving, only $O(\log^2 N)$ messages are required
Experimental Results

• Load Balance
• Path Length
• Simultaneous Node Failures
• Failed Lookups During Stabilization
• Lookup latency
Conclusion

• Simple, powerful protocol
• Only one operation: map a key to the responsible node
• Each node maintains information about $O(\log N)$ other nodes
• Lookups via $O(\log N)$ messages
• Updates to the routing information for nodes leaving and joining require only $O(\log^2 N)$ messages.
• Scales well with number of nodes
More Information

• For more information you can go to

http://pdos.csail.mit.edu/chord/
The End

Thanks