Chord Design Document

Distributed systems

Stephanos, Koi

# Chord Overview

Chord is a peer-to-peer network structure that distributes nodes and resources around a circular structure, in which nodes and resources are assigned by hashing them into a position in the ring. Each node is responsible for resources mapped to their own ID and preceding IDs for which there is no node currently present. Each node maintains a link to its immediate predecessor and successor, as well as a finger table containing shortcuts around the chord. These shortcuts are based on offset of powers of 2, enabling a O (log(n)) look up time for any resource present in the chord.

In order to maintain this structure, policies for nodes joining and leaving the node, as well as looking up a resource key and constructing the finger table of shortcuts must be implemented. The remainder of this document contains diagrams and pseudocode indicating the high-level operations of such policies.

# Chord Diagram

Here is an example chord structure, with finger table and lookup sequence:

Diagram

Description automatically generated[[1]](#footnote-1)

# Node Joining Chord

## Pseudocode

|  |
| --- |
| **requestToJoin(joiningID):** *// Sent to some node already present in the chord*  *// Member node who receives request checks to see if it is predecessor/successor for node*  **successor(joiningID):**  *// If joining node fits here, update predecessor and report back with resources*  **If** (joiningID > self.predecessorID **AND** joiningID < self.successorID):  updatePredecessor(joiningID)  **foreach** (resource **in self.**resources):  **If** (resource.key <= joiningID) resources.add(resource)  alertJoiningNode(successorNode, predecessorNode, resources)  **Else** forwardRequest(joiningID) *// Then successor is repeated on nodes until success*  *// Once joining node is alerted, set its predecessor and successor, and alert its new successor*  **setPredecessorSuccessor(successorNode, predecessorNode)**  **alertNewPredecessor(joiningNodeID)**  **updatePredecessor(joiningNodeID)** *// Set joining node to predecessor, completing insert* |

## Sequence Diagram

Diagram

Description automatically generated

# Node Leaving Chord

## Pseudocode

|  |
| --- |
| // To initiate leave, notify successor node of its new predecessor and transfer your resources  **alertNewPredecessor (successorID):**  sendMsg(predecessorNode, successorID)  // Then the successor node updates its predecessor, and updates its resources  **updatePredecessor(predecessorID)**  **updateResources(resources):**  self.resources += resources  // Once we update ourselves, alert the leaving node’s predecessor we’re its new successor  **alertNewSuccessor(successorID)**  // Then the leaving node’s predecessor updates its successor and sends a confirmation  **updateSuccessor(successorID)**  **confirmLeave():**  sendMsg(leavingNode, “All set to disconnect”)  // Finally, leaving node disconnects all its connections  **disconnect()** |

## Sequence Diagram

Diagram

Description automatically generated

# Looking up Resource

## Pseudocode

|  |
| --- |
| *// Assuming we aren’t responsible, first check fingerTable for responsible node*  **checkFingerTable(resourceKey):**  **for** (nodeID **in** fingerTable):  **If** (nodeID > resourceKey) **requestResource(resourceKey, nodeID)**  *// If responsible node not found in fingerTable, forward request to successor*  **requestResource(resourceKey, nodeID, requestingNode):**  **If** (resourceKey in resources) **returnRequestedResource(requestingNode, resource)**  **Else** r**equestResource(resourceKey, successorID, requestingNode)**  *// Successor node then checks finger table, forwarding if unsuccessful until resource found*  *// Once the node responsible for the resource is found, return it to the requesting node*  **returnRequestedResource(requestingNode, resource):**  connect(requestingNode)  sendMsg(requestingNode, resource) |

## Sequence Diagram

Diagram

Description automatically generated

# Constructing Finger Table

## Pseudocode

|  |
| --- |
| *// Shortcut keys are to offsets into chord (based on powers of 2 to decrease search time)*  *// Shortcut values are the chord member node responsible for the resource at that position*  *// In order to determine which nodes belong in the table, we need the chord structure*  **pollChordStructure(chordStrucDict):**  *// First check if we initiated the poll, if so, create the fingerTable*  **If** (chordStrucDict[0][nodeID] == self.ID) **createFingerTable(chordStrucDict)**  *// Otherwise, add own nodeID and address to dict, and forward to successor*  **Else** chordStrucDict.append({self.ID, self.Address}) AND forwardPoll(successorNode)  *// When the dict gets back to the initial node, we have all the data for the fingerTable*  **createFingerTable(chordStrucDict):**  **for** (shortcut **in** fingerTable):  **for** (nodeID **in** chordStrucDict):  **If** (nodeID >= shortcut)  fingerTable[shortcut] = nodeID **AND** connect(nodeID) |

## Sequence Diagram

Diagram

Description automatically generated

1. Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." *ACM SIGCOMM Computer Communication Review* 31.4 (2001): 149-160. [↑](#footnote-ref-1)