TOTAL CHORD: THE COMPLETE SEARCH OVER CHORD SOLUTION

CSCI-GA.2620: NETWORKS AND MOBILE SYSTEMS

Chris Pac
cpp270@nyu.edu

Single Keyword Search Over Chord:
A Scalable Peer-To-Peer Lookup Service

Overview

- In a Chord peer-to-peer lookup service a user can only search via specific predetermined indexes (keys) omitting information embedded within a file and/or other content
- Creating an unrestricted content search option over Chord service allows users freedom to:
  - Review information embedded within any file
  - Search full P2P content using any keyword
  - Supply keywords without imposed constraints

Defining the Problem

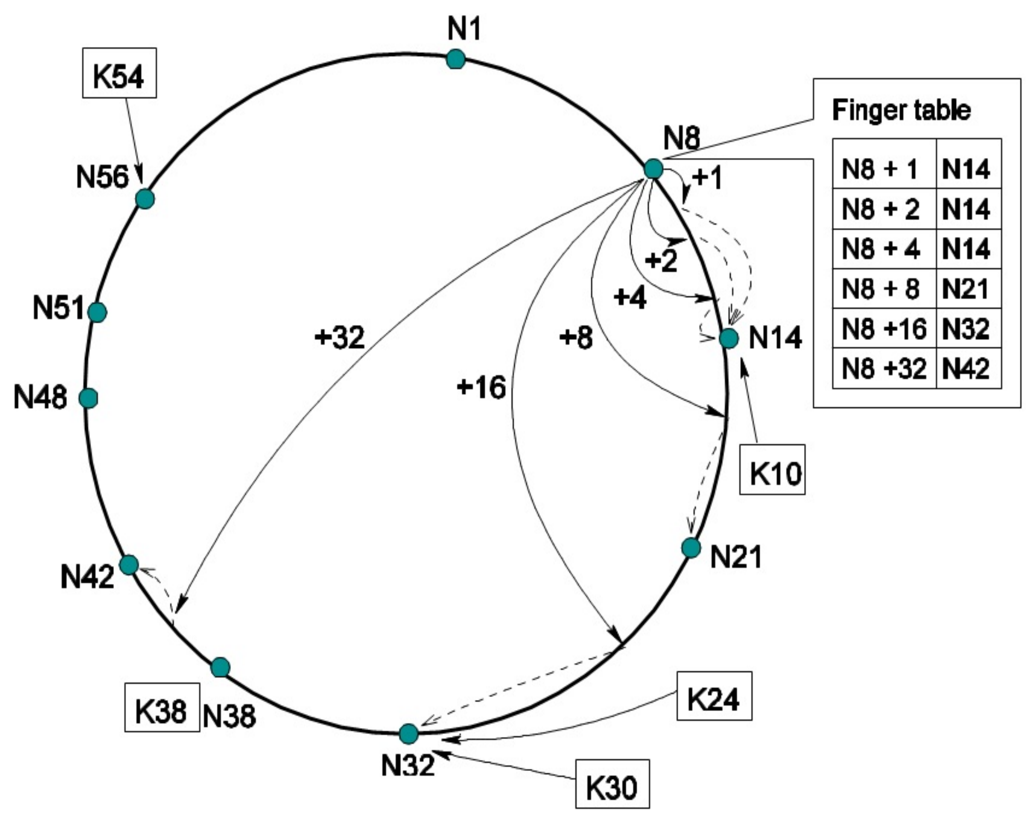
- A Chord P2P service limits a users ability to search for information within files
- Nodes only store (key, value) pairs
  - Key generated from filename, contents, or any arbitrary index.
  - The value can be file contents, pointers, simple strings, etc.
  - In current form Chord service does not provide a method to search the values' content

The Solution

- Solution is a distributed and parallel search
  - The search occurs across every node
  - Each node initiates additional remote searches
  - Results are cached
  - Chord strengths (simplicity, scalability, flexibility, availability, load balance) are leveraged for caching and searching
- Implementing this solution will:
  - Reduce search time for users
  - Balance load across the network
  - Allows users to perform any search(es)

Grasping Chord

- Chord is a distributed lookup service
  - Supports one operation: given key, it maps the key onto a note
  - Maps both keys and nodes (node IPSs) to the same ID space
  - Uses consistent hashing:
    - Spreading keys evenly over nodes
    - Node responsible for K/N keys
  - Lookups needs O(log N) messages (finger table)
  - Allows flexibility of how to map names to Chord keys
  - Fully distributed



* Modified Chord figure from "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications", Ion Stoica et al., IEEE/ACM Transactions on Networking, Feb. 2003.

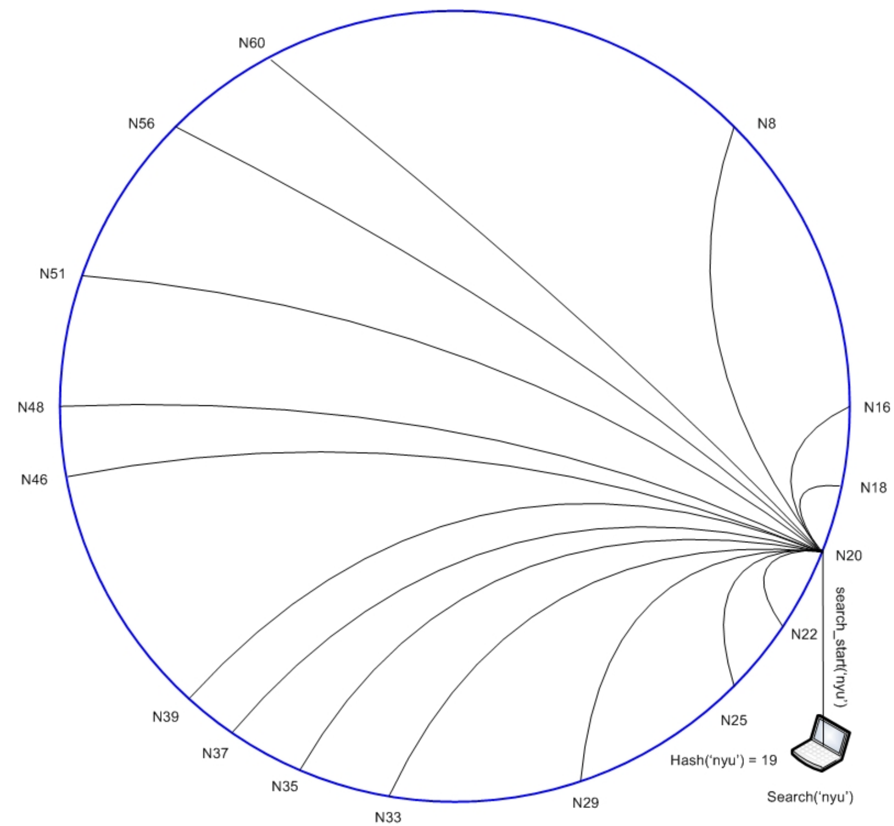
Implementation

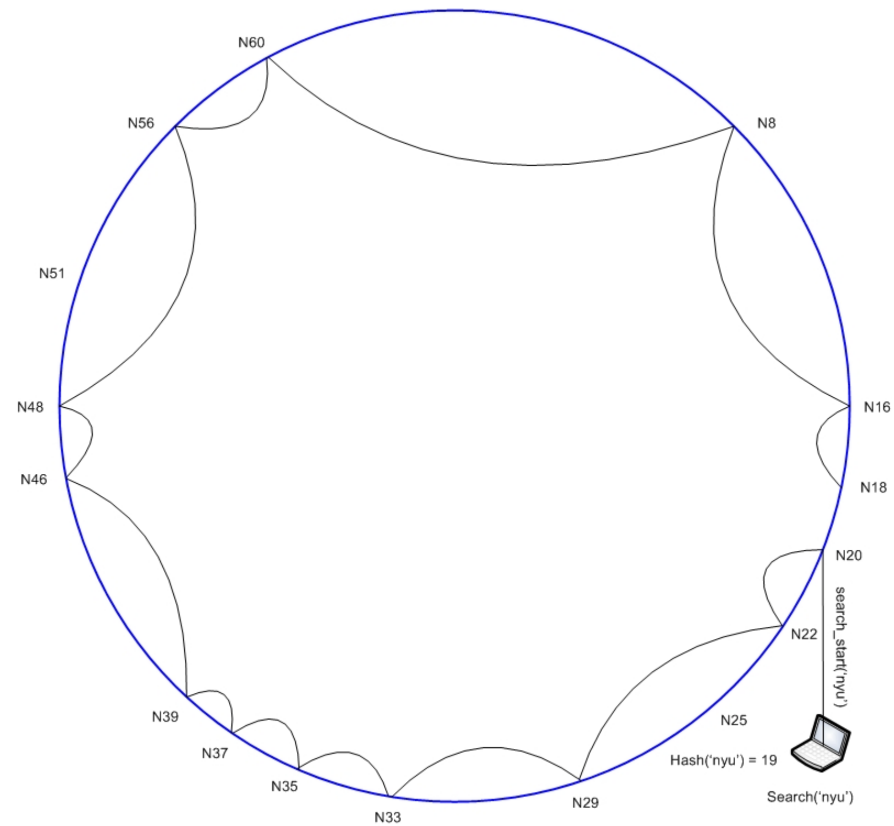
- Build on top of Chord solution

```
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (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 downto 1
        if (finger[i] ∈ (n, id))
            return finger[i];
    return n;

// create a new Chord ring.
n.create()
    predecessor = nil;
    successor = n;

// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find_successor(n + 2^(next-1));

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;

// called to perform search using the finger table over the entire ring
n.search(wallID, keyword)
    for i = m downto 1
        if (finger[i] ∈ (n, wallID))
            finger[i].search(wallID, keyword)
            wallID = finger[i]
    // perform search over data in n and return results

// called to initiate search
n.search_start(keyword)
    n.search(n, keyword)

// called to save keys for particular keyword search
n.cache_put_keys(keyword, keys)
    // hash keyword and save the keys at that location

// called to retrieve keys for particular keyword
n.cache_get_keys(keyword)
    // hash keyword and return keys at that location for that keyword
```

Implementation

Eliminating Alternate Solutions

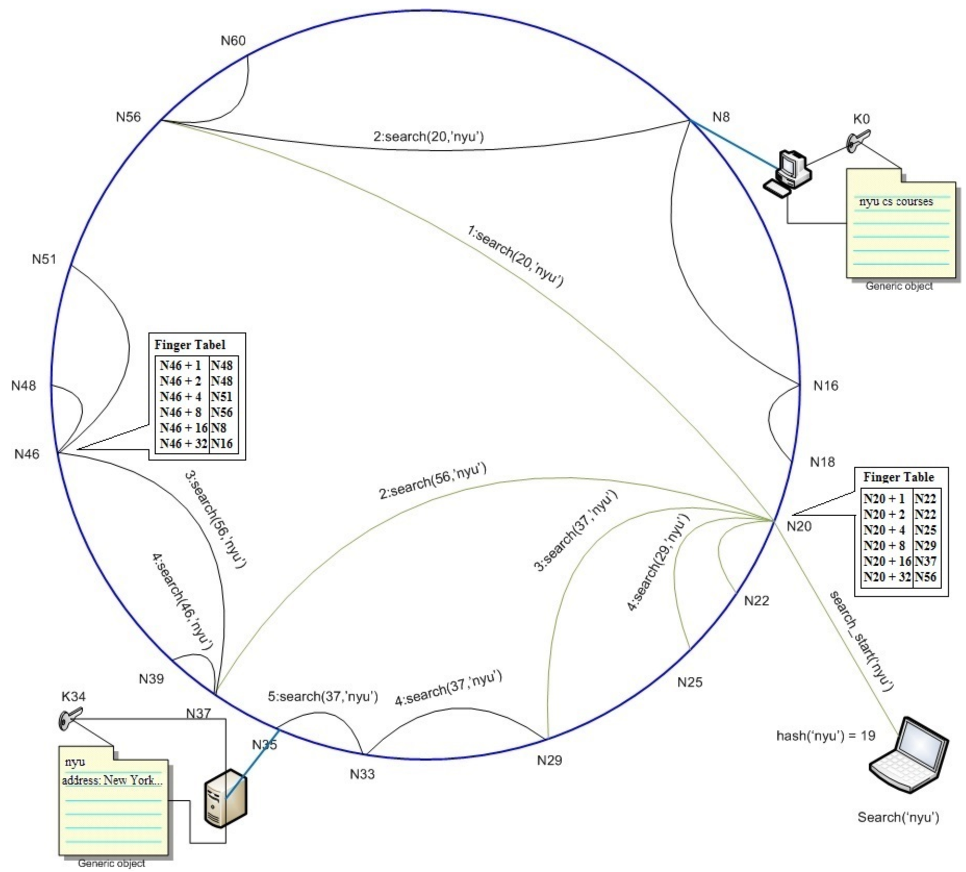


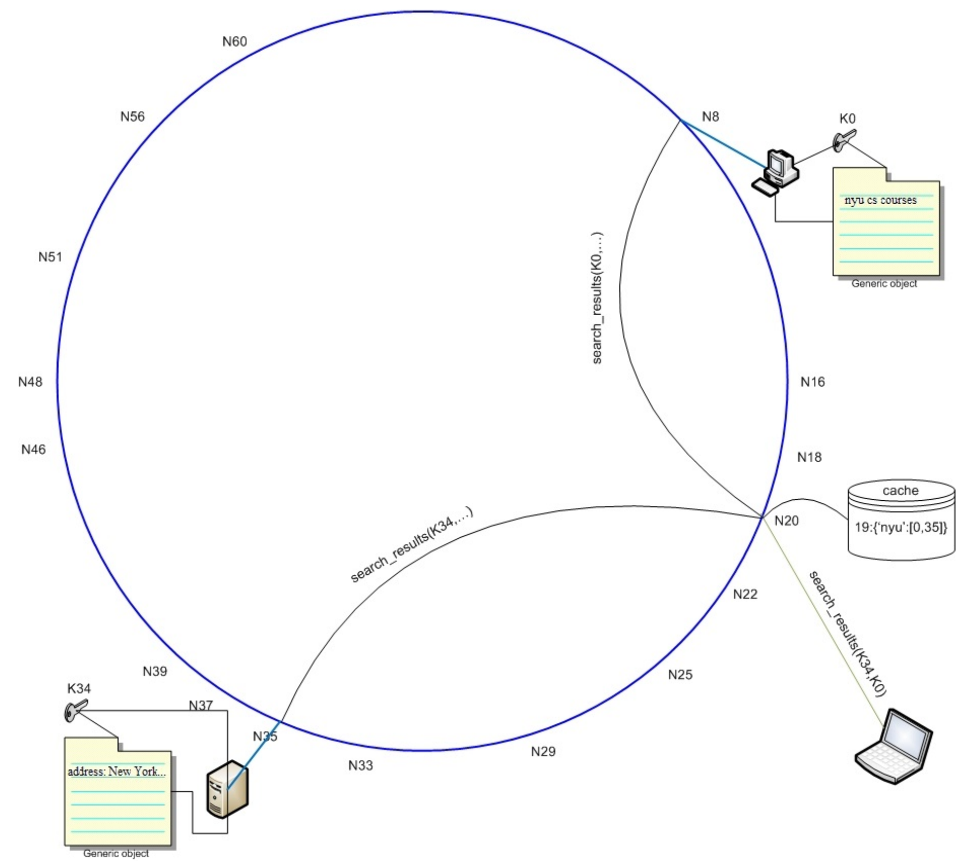


- **Impractical**. This approach does not scale well with large numbers of nodes.
- **Slow**. Search happens one node at a time.

Implementation

Final Proposed Solution





- 1: Search Start.
- 2: Search Results.

Summary

- Implemented Total Chord project using python with 2000 lines of code
- Tested on local machine and over Seattle peer-to-peer computing testbed
- Conclusion is that program works!
  - Initial top line test results indicated that Total Chord solves the problem of limited user search capabilities
  - Total Chord delivers on outlined solution providing flexibility of searching and retrieving more content
  - Next steps include further more-in-depth testing with increased number of machines required to understand full potential and measure reliable performance over continued usage