CS 740 Advanced Computer Networks: AS2
Prototype Implementation of Chord

Ethan Brown
University of Wisconsin Madison

1 Overview

For assignment 2, I implemented a simple version of the Chord peer-to-peer lookup protocol [1] including their approach to consistent file hashing and scalable key location using finger tables. The Chord protocol functions as expected for adding and removing nodes from a ring-structured peer-to-peer network, and for scalably storing and retrieving files distributed across the ring. My implementation was primarily concerned with reproducing the correctness and scalability of the Chord algorithm, as the original paper focused more on those components than evaluating and comparing the latency of Chord with other peer-to-peer protocols.

2 Implementation and Correctness

My Chord prototype is implemented in Python (contained in node.py). Each node on the prototype Chord peer-to-peer network runs as a separate process on the same machine for simplicity, exposed over a different port. Communication between nodes is handled by Python socket TCP communication, and serves as a reasonable simplification of real machine-to-machine communication with synchronous messages. My implementation uses a value of m = 8 for Chord, meaning that keys for files and nodes use the lower 8 bits of hash values (with a maximum value of 2^8) and node finger tables contain 8 entries. This value is low for a typical Chord network, where m must be high enough to ensure no collisions between hash keys; however, a lower value of m makes it easier to visibly verify the protocol's correctness, and with manually-assigned node hash keys (discussed below) the collision chance with m = 8 is sufficiently low for basic testing.

The two most significant key ideas in the Chord paper which I focused on were the consistent hashing protocol, which structured nodes in a ring and associated files with them according to hash key, and the key lookup algorithm in which nodes maintained finger tables to lookup other nodes in increasing intervals. These contributions are what enable the Chord protocol's main scalability advantages, which are having nodes store information about a limited subset of other nodes and achieving a logarithmic overhead for querying nodes in file lookups.

Nodes in this prototype maintain only information about their successor, predecessor, and m (8) nodes from their finger table. Performing file stores and lookups take at worst a logarithmic number of jumps between nodes, shown by output on messages received by each node. Nodes easily maintain files that 'belong' to them under the hash rules, and can take over or give up files from their successor and predecessor as nodes are added and removed using operations involving a single request and response with a single neighbor node. No lookup is required and no files are lost during file redistribution after ring structure changes in my implementation.

My implementation prompts all nodes on the ring to update their finger tables upon adding or removing a node. The simplest approach is for each node to re-do lookups for each table entry key. This prototype performs an optimization, in which table entries are replaced by the ID of a new node if the previous entry was the new node's successor and the new node is recognized to be 'taking over' that entry key. A similar optimization is performed for updating tables after node removal, where only table entries containing the old node are replaced with its successor (avoiding the need for lookups in both cases). Adding and removing nodes on the prototype demonstrates at most m table entry update operations per node on the ring, with none of them requiring lookups beyond table initialization for the new node, as described by the paper. The above performance can be demonstrated using as few as 6 nodes with well-spaced IDs (described in README), and the implementation was checked against 16 nodes for scalable correctness.

Initializing the finger table for a new node is unoptimized in my implementation. The paper proposes several methods for initializing tables, including making a lookup query for each of the m entries (m * log(m) time) and copying/selectively updating the finger table of a neighbor node. My implementation uses the first approach, but could be extended using existing functions that query if a node is the successor to a hash key given more time and initialization complexity.

One of the most important functionalities in the core Chord protocol is the ability to locate the "successor" of a hash key, which refers to the node which is responsible for a given hash key. In the case of a node's hash key, the successor is the node itself; for a file, the successor is the node with a hash ID closest following the hash key of the file. The paper describes how this can be done in one of two ways: a naive solution where each node queries its successor resulting in O(n) queries, or a scalable solution in using finger tables resulting in O(log(n)) queries as described in depth in the paper. I initially implemented my Chord prototype using the naive circular search pattern for key lookups while building the other components of the algorithm, then implemented the proper finger table lookup mechanism for the final version.

3 Limitations

Due to time constraints and the complex nature of some of the features and optimizations proposed in the Chord paper, this prototype implementation leaves certain features to future work. The core functionality of the Chord algorithm is implemented and demonstrated by the prototype, and most of the limitations of the prototype are either extensions of basic Chord functions which are not part of the main contribution of the algorithm or features of a fully-implemented peer-to-peer file hashing system which are beyond the scope of demonstrating the core algorithm.

3.1 Manually Starting Nodes

This prototype implementation requires that nodes be given a hash ID, a port number for themselves, and the port number of the existing node they will join onto as command line arguments on creation. It also requires that users creating a new node select the node to join onto based on the appropriate predecessor of the new node ID.

In a fully complete Chord implementation, new nodes are assigned hash IDs by hashing their IP using the same method as hashing files on the Chord network: computing a SHA-1 hash and taking the lower m bits, with m being high enough to avoid collision. A complete Chord implementation also performs node join operations without requiring a user to enforce joining a new node with the correct existing node to preserve node ordering on the ring. Instead, it has nodes periodically notify their successors and predecessors, and reassigns successors and predecessors values in nodes as necessary to restore ordering in the ring. For simplicity of implementation and testing, my prototype has users specify custom hash IDs for new nodes, and it requires new nodes to be joined in correct order so that nodes may notify a single predecessor/successor to insert itself into the ring at the time of the join rather than using concurrent updates. However, although this approach does not support non-concurrent joins, the basic join functionality where predecessors/successors are notified to "stitch" together the ring can be reasonably extended to propagate full circle over the ring (similarly to my implementation's finger table updates).

3.2 Background Stabilization

A complete Chord implementation supports concurrent operation by handling joining nodes, removing nodes, and updating finger tables via background operations which periodically check a node's predecessor, successor, and finger table entries for correctness. These checks are implemented in a non-concurrent design in my implementation; joining and removing nodes triggers messages to the successor and predecessor to update their pointers and hand over/take over files, and triggers a request propagated around the ring for nodes to update their finger tables. The complete concurrent-support version of Chord would have nodes perform these same operations after notification by a periodic check running in the background rather than a direct one-time request from the node causing the change. Extending this prototype to support concurrency would require more complex and higher volume communication between nodes in the background, but can be assumed to work correctly using the same fundamental protocol implemented by the non-concurrent prototype.

3.3 Crash Recovery

One consequence of the non-concurrent prototype described above is that my implementation depends on nodes leaving the Chord ring to send signals to adjacent nodes prompting the necessary updates throughout the ring. Thus, nodes that crash unexpectedly will leave the network nonoperable for any operation that requires contacting that node. The complete Chord implementation would handle crashes using the aforementioned background stabilization behavior, in which missing nodes would be detected by periodic background checks and predecessors/successors/finger tables updated accordingly.

3.4 Mock File Data

My implementation uses string data in place of fully implemented files; nodes prompt users for string text as "file input", hashes the string in place of a file name and path, sends the file string between nodes during storage and retrieval, and stores the file string and associated hash in a dictionary in the node (rather than writing/reading on a disk). This omission was made for simplicity and should have no impact on the correctness of the Chord algorithm, as the file data strings function similarly to file names.

References

[1] Stoica, Ion Morris, Robert Liben-Nowell, David Karger, David Kaashoek, Frans Dabek, Frank Balakrishnan, Hari. (2003). Chord: A scalable peer-to-peer lookup protocol for Internet applications. IEEE Transactions on Networking. 11. 10.1109/TNET.2002.808407.