Implementing a Distributed Peer to Peer File Sharing System using CHEWBACCA – CHord, Enhanced With Basic Algorithm Corrections and Concurrent Activation

Matthew Baker, Russ Fink, David Trimm, Adam Whisman

In Partial Fulfillment of the Requirements of CMSC 621, Advanced Operating Systems

Fall, 2003

Abstract

We implemented a distributed, peer-to-peer file system in Java using the Chord protocol for efficient location of files stored across a network. Chord is a distributed key lookup technique that scales logarithmically with an increase in the number of storage nodes. Initially, we discovered problems with the Chord algorithms, and prototyped corrections to verify the changes. To implement the file sharing system, a Sockets-based messaging framework was implemented in Java, using command line interfaces and an FTP-like file push/pull semantics. The file system enables distributed peers to present a virtual single directory to users while sharing file load across different platforms. Using the application, we verified the claimed messaging performance of the basic Chord algorithm to be close to Mlog(N) behavior in an average case. The algorithms were modified to do locally-optimal balanced node joins, and experiments showed a 20% improvement in load balancing fairness over the standard, unbalanced algorithms. We designed algorithm modifications to support concurrent joins and leaves of the Chord cloud using a variant on Raymond's tree-based distributed mutual exclusion algorithm, and informally proved some safety and liveness properties of the implemented balance and proposed token-based mutual exclusion algorithms. During the investigation, several improvements were suggested for the Chord-based file system, including distributed mutual exclusion with quorum voting, security, replication, and other enhancements.

Table of Contents

1 Introduction 4
2 Background 4
2.1 Peer-to-peer File Serving 4
2.2 The Chord Protocol 4
3 Design and Implementation 5
3.1 Messaging 5
3.2 Chord Algorithm 6
3.3 Chord Algorithm modifications 7
3.3.1 Corrections to Published Algorithm 7
3.3.2 Algorithm Enhancements 8
3.4 File Sharing Application 10
4 Experimentation Methodology 11
5 Results 12
5.1 Algorithm verification 12
5.2 Baseline results 12
5.2.1 Data 13
5.2.2 Findings 16
5.2.3 Results of Improvements 17
6 Lessons Learned and Future Work 18
6.1 Limitations of the System 18
6.2 Lessons Learned 18
6.3 Suggested Enhancements 19
6.3.1 Security 19
6.3.2 Locking and Concurrent Join/Leave 19
6.3.3 "Gossip" Messaging 21
6.3.4 Other Enhancements 22
7 Conclusions 23
8 References 24
9 Appendix: Corrected Chord Algorithm 25
10 Appendix: Enhanced Chord Algorithm 28
11 Appendix: Informal Enhancement Proofs 34
11.1 Backfilling 34
11.1.1 Safety 34
11.1.2 Locally-Optimal Balancing 34
11.1.3 Performance 35
11.2 Cloud Locking 35
11.2.1 Safety 35
11.2.2 Free of Starvation 36
11.2.3 Performance 37

1 Introduction

This project implements a simple peer-to-peer (P2P) file sharing system using a software library designed and implemented for the use of applications on a P2P system. The underlying P2P software library of this system is an implementation of the algorithms of the Chord protocol, implemented in the Java programming language. Each node in the system uses the Chord protocol and TCP/IP messages to communicate. The P2P system as a whole is referred to as a Cloud. The Chord protocol provides methods for nodes to join and leave the Cloud at anytime. On top of the implementation of the Chord protocol a simple file sharing application was implemented. In the file sharing application, files are stored in the nodes of the Cloud and various clients exist in the system that can communicate with a local Cloud node. The clients are able to perform the basic operations of storing, retrieving, finding, and deleting a file from the Cloud and listing all of the files that are stored in the Cloud. After implementing the system a study was conducted on the performance characteristics of the system and the results are shown in this paper.

2 Background

2.1 Peer-to-peer File Serving

A P2P system is a distributed system that consists of individual clients that connect to each other directly rather than connecting to a central server. All processing and messaging is handled by the clients themselves, so in effect all clients are both clients and servers, and all clients have equivalent software or equivalent functionality. This means that each client is capable of making some type of request to other clients and responding to requests from the other clients as well. The main advantage of such a system is that processing and data can be distributed to each client of the system rather than at any central location. One of the main application areas of P2P systems is file sharing. In a P2P file sharing system each client has the ability to store files that other clients can access and the client can find and access files that are stored at other clients of the system. The clients only communicate with other clients and files are only stored at the clients, eliminating the need for a central file server accessible to all clients. For more information on P2P systems please see [1].

2.2 The Chord Protocol

The Chord protocol is a simple P2P protocol made to simplify the design of P2P systems and applications based on it [2]. The Chord protocol has five main features that are used to solve the difficult problems of a P2P system and distinguish it from other protocols. First, Chord provides a degree of natural load balancing by using a uniform hashing function, in which it spreads keys evenly over the nodes of the Chord system. Second,

Chord is a completely distributed system, meaning no node is more important than any other node, making Chord appropriate for dynamic and loosely-organized P2P applications. Third, Chord automatically scales well to large numbers of nodes and Chord keeps the cost of a Chord lookup down, since it grows as the log of the number of nodes. Fourth, Chord provides a high level of availability meaning that Chord nodes can always be found no matter how much the system changes. Chord provides for availability to newly joined nodes and takes care of node failures as well. Lastly, Chord provides for flexible naming making no constraints on the structure of names and keys meaning applications can be flexible in mapping names to Chord keys. In its basic form the Chord protocol specifies three functions. Those functions are finding keys, joining new nodes, and recovering from the planned departure of existing nodes. In the system proposed in this paper, the Chord software has the form of a library that is linked with the file sharing applications that use it for its P2P features.

3 Design and Implementation

3.1 Messaging

One of the keys to the design was choosing to use sockets instead of RMI. The overhead of RMI, inflexibility of the two-way messaging model, and running a separate RMI server on each node was not desired, so a pure sockets implementation was used.

To support the sockets implementation, a Message Manager was implemented. Simply put, the Message Manager is a dispatch handler similar to RMI in purpose, but implemented differently. The Message Manager handles three types of message patterns: unsolicited inbound messages, one-way outbound messages, and round-trip messages that represent a request-reply pairing.

In the implementation, the Message Manager runs an Acceptor thread to listen to inbound messages, and has public methods to facilitate messaging by the application:

- sendMessage - sends a one-way message
- sendMessageAndWaitForReply - sends a message, suspends the current thread until a reply to that message is received, implementing round-trip semantics
- sendReply - sends a reply to a received message

There are also handlers that get registered to handle unsolicited in-bound messages as well as replies to round-trip messages.

Messages themselves are serialized ChordMessage derivatives. Each ChordMessage contains a method, executeTask(), which is called by the recipient of the message. ExecuteTask() contains code that is run by the recipient, and makes use of state in the

recipient to continue its processing. Each type of message transfer in the system is represented by a different subclass of ChordMessage. For instance, ChordMessageFindPredecessor represents a request of a node to return its predecessor.

In order for messaging to work correctly, the Message Manager was designed to promiscuously accept all received messages for processing. Because of this, security is fairly weak in the system in that the message manager never discriminates on messages received. This choice was made due to time constraints on the project.

The Message Manager has an additional feature called *deferred replies*. This is used to delegate replies to nodes that have the needed information, such that these nodes do not have to be the recipient of the original request. For instance, in round trip messages, sendMessageAndWaitForReply blocks the calling thread until the proper reply message is received. Since the initial message may be forwarded along the chord to a different node, deferred replies can be sent by the final target node to the originator causing it to unblock and process the reply. By utilizing deferred messages, message traffic can be kept to a minimum. Instead of "A sends to B sends to C, C replies to B replies to A," the Message Manager permits "A sends to B sends to C, C replies to A."

3.2 Chord Algorithm

The Chord algorithm is implemented within a ChordNode class. ChordNode has methods corresponding to the primary algorithms of the Chord Protocol:

- findSuccessor
- findPredecessor
- join
- leave
- and other supporting methods such as updateOthers, initFingerTable, and so on.

The ChordNode class was chosen to model the paper algorithms as closely as possible for ease of understanding by someone familiar with the paper.

As a helper for ChordNode, a FingerTable class was developed. ChordNode contains entries in its FingerTable similar to those in the Chord paper. The key difference in the array between the implementation and the paper is that the finger table index starts at 0 in the implementation, while it starts with 1 in the paper.

For efficiency, some parts of the Chord algorithms are implemented in the executeTask() methods of some ChordMessages. While it was desirable to keep the primary portions of the Chord algorithms inside the ChordNode class, some functional coupling was done in the interest of time. The group felt that functional reorganization was not worth the time, as this is not a "production" system.

3.3 Chord Algorithm modifications

Extensive hand traces initiated at the beginning of the project to understand the characteristics of Chord soon led to discovery of fundamental transcription mistakes in the published algorithm. Additionally, further understanding of the algorithm pointed to some deficiencies that were readily corrected with changes to the algorithm.

This group has pursued both types of changes, and has recorded empirical data to show the effects of these changes.

3.3.1 Corrections to Published Algorithm

Hand traces revealed one algorithm error where update_finger_table would not complete under some circumstances. After this discovery, a prototype was written and combinatorial testing was performed that eventually led to discovery of two more errors.

The first error, discovered by a hand-trace, is that update_finger_table would not terminate under the specific case where it was called by the node "s" itself, meaning that "s" should update its finger table with "s." This would result in the predecessor of "s" being called once again, which would begin an *N-walk* around the chord again. This condition was observed by the prototype. The fix was to check if self equals "s" in the call, and return if it did, terminating the node walk at the newly joined node.

The second error, discovered through the prototype, was that update_others would fail to update the immediate predecessor of a newly joined node if that predecessor occupied the slot right behind it. This *shadow bug*, wherein the predecessor can "hide in the shadow" of the newly joined node, was fixed by adding 1 to the find_predecessor call, which would include the immediately previous slot and cases where a predecessor occupied the slot.

The third error was that init_finger_table would sometimes return an incorrect finger entry. This case occurs where the new node is joining between a previous node and the finger[M] entry of that node; in this case, the finger[M] entry of the new node would point to its successor, instead of itself. The problem comes from the fact that the chord node does not have knowledge of the newly joining node when this check is made. The fix was to do an ordering check on returned finger table entries, to ensure that the *i*th finger node is never in front of the newly joining node.

The prototype verified that omission of any of these fixes would result in incorrect finger table entries, or infinite loop conditions. It is speculated that because the original Chord paper discusses a periodic polling variation that tolerates simultaneous joins and leaves, the original authors did not implement the listed algorithm and are unaware of the problems.

The corrected algorithm appears in the appendix 9.

3.3.2 Algorithm Enhancements

A few enhancements were made to Chord to fulfill the needs of the project, and to provide a more efficient implementation. This section gives an algorithmic overview of certain enhancements; refer to the appendices for detailed discussion of correctness conditions, as well as a listing of the algorithms.

3.3.2.1 Leave

The initial enhancement to the protocol was the algorithm for *leave*. Leave is similar to join, except that there is no call to find_successor because the chord position is already known and the leaving node’s fingertable does not have to be filled. Instead, *leave* disconnects itself from the chord, sets its successor's predecessor to its predecessor, and it's predecessor's successor to its successor. It then initiates what the group has been calling a "hand grenade" volley backwards in the chord to update the other nodes whose finger tables might have included the departing node.

In this technique, the leave() algorithm "fires grenades" at each of its preceding finger positions: the predecessor of the slot 1 position back, 2 slots back, 4 back, ... 2^(m-1) slots back. This is similar to what update_others() does during a join. Each recipient of the "grenade" updates its finger table, then propagates "shrapnel" back to its predecessor, hopping from one predecessor to the next updating finger tables, until such a point that a predecessor has no record of the departing node in its finger table. Since there are "m" grenades launched, there are "m" shrapnel chains, and each chain ends with a node that does not have the departing node in its finger table. This feature is important in the termination conditions that were also implemented.

3.3.2.2 Balanced Join

Balanced join is a simple adaptation to the protocol that makes use of one very interesting fact about Chord - the hash value of the node itself has nothing to do with the hash value of the files that it stores. The purpose of the hash of the node during the initial join is merely to ensure some pseudo-random even distribution of the nodes in the chord positions, in order to create a balanced tree.

In practice, the hash functions do not do a perfect job of balancing the tree. The group has noticed clustering of nodes around certain hash positions - not always, but often enough to warrant a change. In balanced join, a joining node will hash itself to a region of the chord, then query its successor and predecessor's slot number and set its own slot (index position on the chord) to be precisely halfway between the two. While non-optimal globally, it is a locally optimal solution and should be "good enough" provided the hash function doesn't cause clustering too often.

3.3.2.3 Backfilling (Balanced Leave)

While the hash function should ensure a roughly balanced cloud during joins, a more significant imbalance can occur when a sequence of adjacent nodes leaves the cloud. Should this condition happen, then eventually there becomes a single node that assumes all of the departed nodes' slots, which can result in a severe cloud imbalance.

To ensure balance during sequential departures, a technique called *backfilling* was implemented. Note that whenever a node leaves, it pushes its files and slot assignments to its successor - after several sequential adjacent departures, there might exist a huge gap behind one of the nodes. Since the gap is behind the node, it is easy to reason that any non-busy nodes will occur in front of the node.

Backfilling proceeds by a node first detecting a threshold condition on the number of slots assigned to it. (The number of slots equals the number of hash positions for files that this node is responsible for, and having many slots implies the possibility of storing many files.) An example threshold that was implemented is ceiling(N/2). If the threshold on number of slots is exceeded, the node *should* look forward for a non-busy node (one with relatively few slots assigned to it) by querying each of its M fingers for their slot span. [The actual implementation, within time constraints, looks backward in the cloud.] A slot span of a node is the number of slots that lie between the node's successor and predecessor.

The busy node collects slot spans from the leaving node's backward fingers. During the leave, log(N) RemoveNode messages are sent backwards at incrementing backward finger positions. Each backward position represents the head of a "grenade path" (as we call it), and performs a predecessor walk to update finger tables. During the predecessor walk, slot count information is *gossiped* between successive nodes, and a minimum slot count is maintained. When the RemoveNode reaches the last node in the "grenade path", the last node sends a SlotCount to the departing node's successor. This SlotCount represents the local minimum of that grenade path.

After receiving the slot counts from all backward finger "grenade paths," the busy node then chooses the node with the smallest slot span that is less than the threshold condition for number of slots - the idea is not to solve one problem by creating another. It then sends a leave-with-rejoin message, and instructs the non-busy node to *backfill the gap* behind it (from whence the name of the technique is derived) by rejoining with a slot equal to that which is half the distance between the busy node and its predecessor. The non-busy node leaves, and transfers perhaps a small number of files to its successor, then rejoins and accepts approximately half of the files from the busy node, its new successor.

The reason for looking forward as opposed to looking backward for a non-busy replacement is "common sense" because the problem is behind the node, not in front of it.

Suppose the gap is M/2; then, if there are more than M/2 nodes in the cloud, they must all be bunched up in front of the busy node for the M/2 gap to exist. (If there isn't this bunching, then each of the fingers in front will report a large gap, and no balancing will take place). Rather than recursively probing forward, a "good enough" approach is for the busy node to just look at its own M fingers and select a minimum among them. That minimum finger might not be the cloud-minimum node, but it finds it using only M (or log(N)) messages. The changes to support this were omitted from the implementation, though backward looking is done.

3.4 File Sharing Application

The file sharing application received the least attention because it was the easiest to implement. The application consists of a command-line Java client that connects to a *sponsor node* to perform all the required functionality.

The features provided by the Chord Client approximate some of the primitive features found in FTP, and include:

- "get" a file from the cloud to the local file system
- "put" a file from the local file system to the cloud
- "ls" or list all the files in the cloud
- "delete" a file in the cloud
- "exists" query for the existence of the file in the cloud

These simple operations are by no means a complete file system, but provide sufficient evidence of utilization of Chord for the purposes of the project. The group was constrained by time from implementing a more full-featured set of operations.

The Chord Client was decided to be implemented using a command-line interface, using Java's File class. This is a natural way to envision a file transfer client, similar to FTP. Users can transfer - and rename - any file into the Chord cloud, and likewise retrieve any file and store it as a different local name in the local file system. The Java File class provides features for creating, opening, reading, writing, and deleting files and directories. Further, a command-line mechanism is easy to implement and extend.

The Chord Client performs its tasks by interacting with a Chord sponsor node. The sponsor node is merely a node in the Chord cloud. The sponsor mainly provides findSuccessor used for finding files, whereupon the Chord Client initiates a direct connection with the owner of the file for transfers. Note that "put" involves a findSuccessor using the hash value of the name of the file to create in the cloud, while "get" returns the owner of a file that currently exists, or an error message if the file is not found.

In addition to file system operations, the Chord Client was augmented to provide some Chord management features:

- "leave" instructs a node in the cloud to leave; this is to implement one of the project requirements
- "message_count" reports the number of messages processed by the Message Manager, reset after the last invocation of this call, and is used for statistics data gathering

4 Experimentation Methodology

To verify the performance of our Chord implementation, several additional classes were written that exercised the system to analyze three different metrics. Those metrics were the number of messages required for a node to join, a node to leave, and to execute the find slot operation. The rationality behind choosing those metrics was that they provided coverage for every type of operation the implementation performed. For example, the “file store” operation consists of the find slot operation combined with a constant number of additional messages. Similarly, the “file get” and “file delete” operation are the same. The only operation not covered by these metrics is the “list all files” operation. This operation obviously requires N messages where N is the number of nodes in the system.

The first two metrics, the messages required for join and leave, were measured by a class called ChordRandomTester. This class began with one node and added nodes up to the MAX_NODE variable that was hard-coded into the class. The added nodes were randomized both on their names and on their port numbers (within a specified range). The randomization algorithm used was the SHA-1 Pseudo Random Number Generated supplied with Sun Microsystems' Java Development Kit, version 1.4.2. Once the MAX_NODE limit was reached, the ChordRandomTester randomly chose nodes to leave until only one node remained. This process of joining and leaving was repeated a variable amount of time that was specified by the value SAMPLES. During each join and leave operation, the number of messages sent and received was counted. For correctness, all finger tables of the nodes in the system were also validated for every join and leave operation. A class called Statistics provided analysis of each set of samples and generated the min, max, average, median, and standard deviation for the data.

A class called CountFileFindMessages tested the last metric, the number of messages required for a find slot operation. This class randomly generated a set of N nodes and quickly filled in their finger tables without performing a join operation. The join operation was omitted to decrease the time required to acquire the find message counts. Once all of the nodes were created, each node was asked to find a random filename. This filename was the same across every randomly generated Chord, and the number of messages was counted for each find operation. The CountFileFindMessages class varied

the number of nodes in the system and for each node count repeated the experiment a variable amount of time. Similar to the first two metrics, the Statistics class was used to collate the results.

5 Results

5.1 Algorithm verification

A prototype was constructed to verify the correctness of the augmented algorithm by simulating its main features using a single virtual machine instance. It could perform basic operations on the Chord Nodes (join, leave, find_successor, ...) and check the finger table integrity after each operation.

Further, the prototype was augmented to do exhaustive *combinatorial testing*, where an array of N integers was created and permuted over all possible combinations; each integer represented a node with an ID of that specific value, and each permutation represented all possible clouds formed by orderings of node joins. With each permutation of the integer set, the following tests were done:

- All-join-root: a node was created with an artificial slot ID equal to the first integer in the list, the second node with the second ID in the list joined the first node, a third node with the third ID also joined the first node, and so on. After each join, the finger tables were checked of all existing nodes.
- All-join-previous: similar to All-join-root, except that new nodes join the nodes immediately preceding them in order: second joins first, third joins second, and so on. Again, finger integrity was checked after each join.
- Sequential leave: each identified node leaves in turn, and fingers are checked after each leave operation

The corrected algorithms passed all of the combinatorial tests with perfect finger integrity.

It is believed that such combinatorial testing, further dissected into partial combinations as listed above, represented trials of all possible combinations of node I joining node j for all I and j, and similar for node I leaving node j. For a cloud of 8 nodes, this resulted in over 600,000 separate tests being run.

5.2 Baseline results

5.2.1 Data

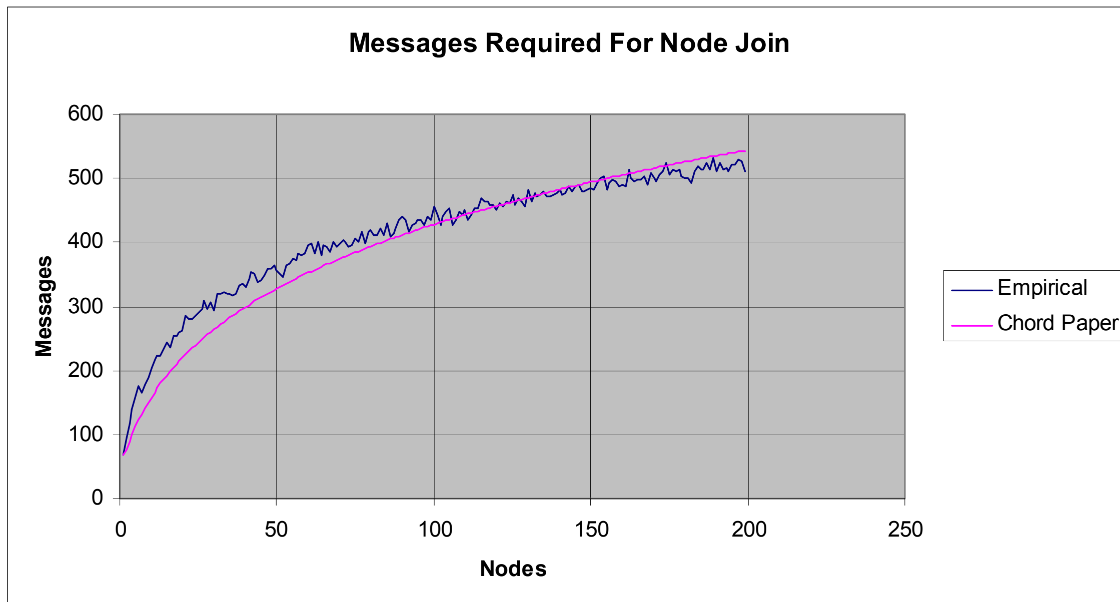

Figure 1. The number of messages required to join compared with the theoretically expected values

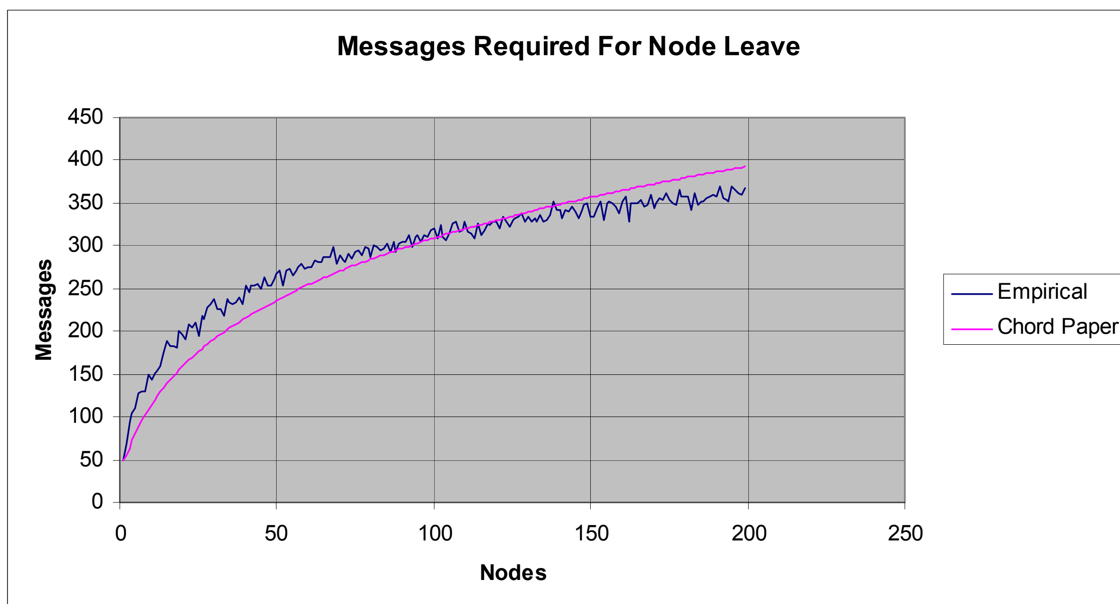

Figure 2. The number of messages required to leave compared with the theoretically expected values

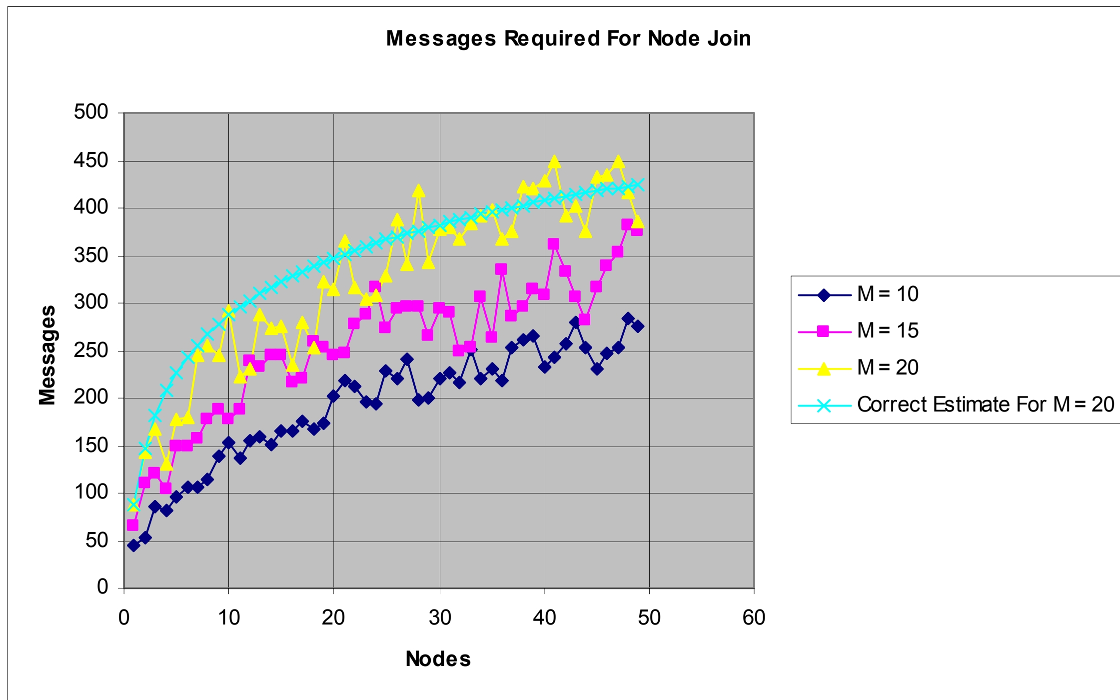

Figure 3. The number of messages required to join with differing cloud sizes

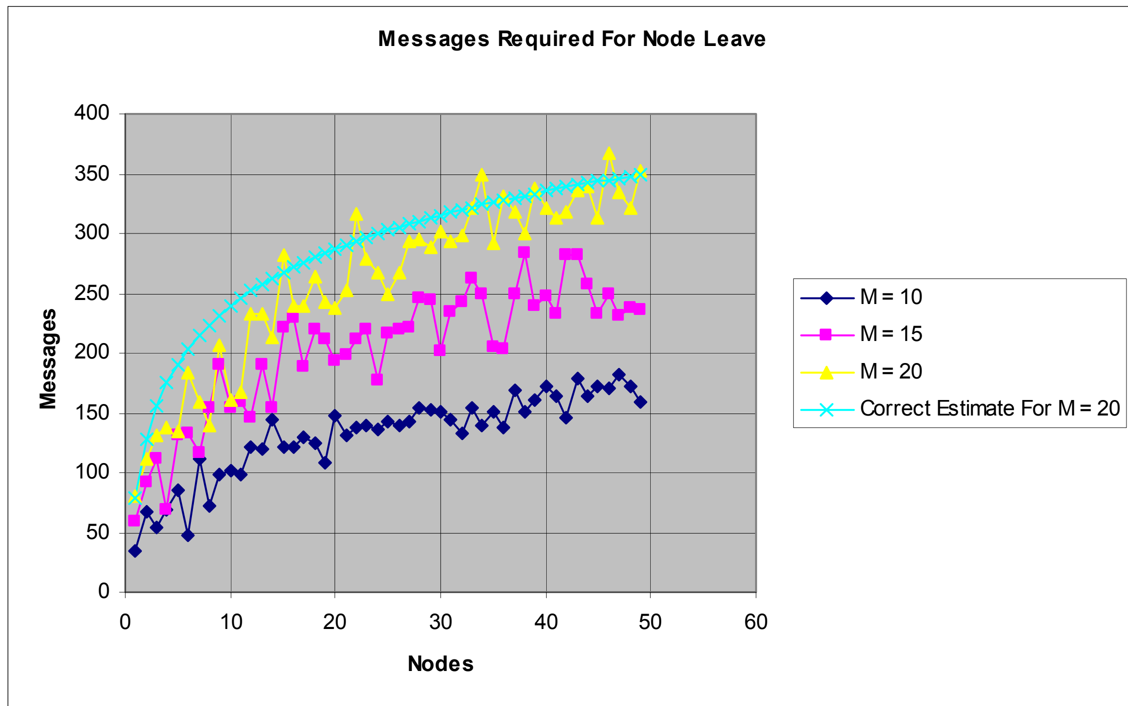

Figure 4. The number of messages required to leave with differing cloud sizes

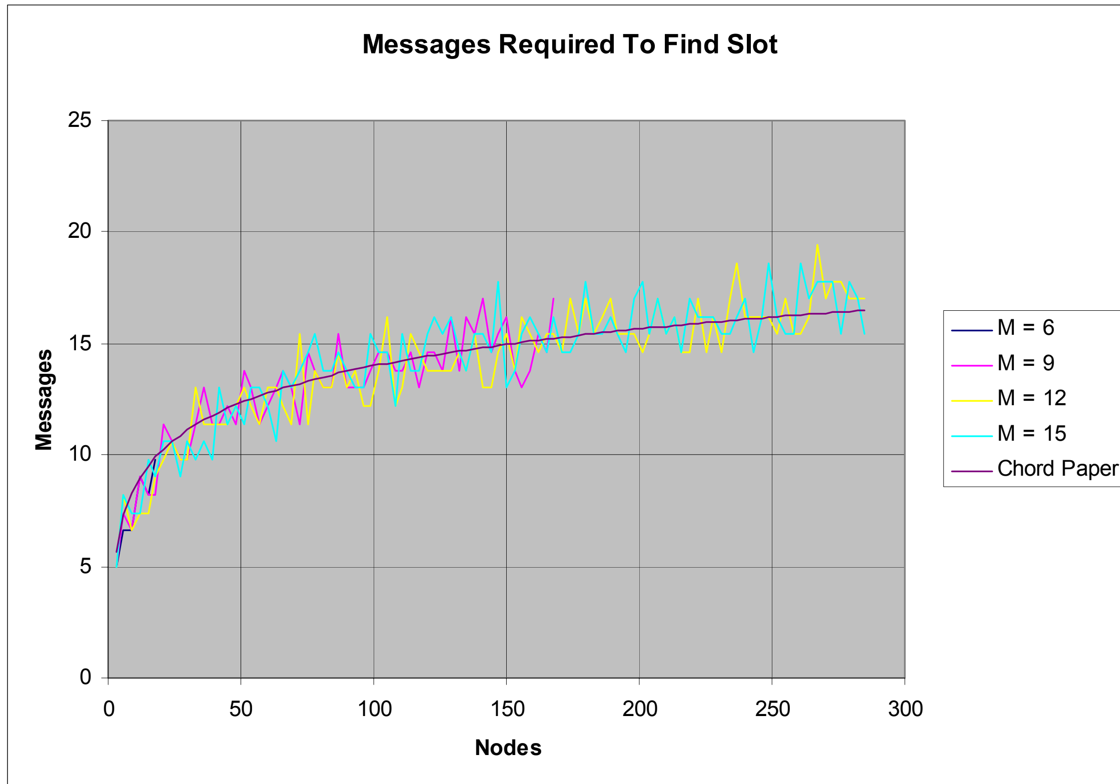

Figure 5. The number of messages required to find a file with differing cloud sizes

5.2.2 Findings

Fig. 1 and Fig. 2 depict the results for Join and Leave, respectively, in our implementation of Chord. Both figures were generated using the ChordRandomTester class that was described previously. Fig. 1 shows the empirical number of messages required for a node to join the system. Also included is the Chord Paper's running time with appropriate constants added to approximate our findings. Similarly, Fig. 2 provides the same data for a node leaving the system. As one can see, the Chord Paper's running time only loosely matches the number of messages required for Join and Leave. An analysis of these running times will be provided in the next paragraph.

After analyzing the ChordNode implementation, it is clear that a more accurate running time for the join operation should include the value M that determines the number of slots on the Chord virtual ring. Recall that there are 2^M slots in the virtual ring. The first step of the join process is to fill all of the entries in the joining nodes finger table. Since there are M entries, the algorithm should take M*Log(N) since each find requires Log(N) messages where N is the number of nodes currently in the system. Once the entries are

filled in, the joining node must update every node that should have it in their finger tables. Initially, M "update your finger table with me" messages are sent out to the existing nodes. Since the recipients of this message must first be found, the number of messages is M*Log(N). Additionally, each node that receives one of these messages may pass it to their predecessor if they, themselves, updated their table. If the nodes are perfectly distributed around the virtual ring, then one node, at most, is updated with each of these messages. Therefore, the number of messages for join is M*Log(N) + M*Log(N) or O(M*Log(N)). The correct estimate with varying M's is shown in Fig. 3.

The leave operation performs in a similar manner to the join operation except that it doesn't fill in its own finger table. However the "update others" method is very similar to the joins with the modification that you are telling other nodes to replace entries of the leaving node with its successor. Therefore, the number of messages for leave is M*Log(N). This estimate with varying M's is shown in Fig. 4.

The find operation performs as specified by the Chord Paper. This is intuitively obvious because the Chord finger table implements a distributed binary search. For every hop that a find message takes, it roughly halves its distance to the searched for slot. The metrics for the find operation are shown in Fig. 5.

5.2.3 Results of Improvements

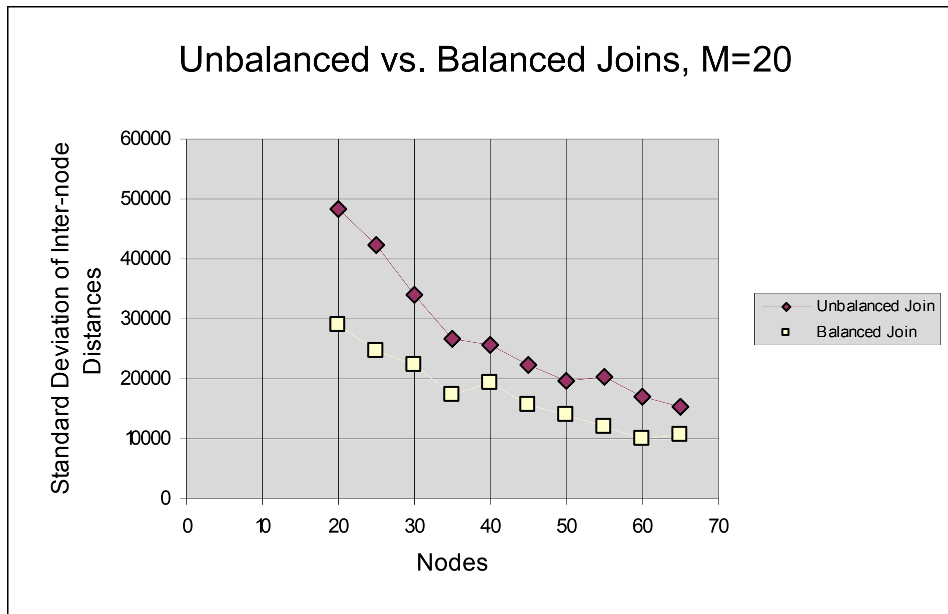

Figure 6. The number of messages required using balanced-join compared with those required using join

Figure 6 shows the results of balanced joins in the system. To measure the "goodness" of this feature, it was decided to collect and measure the standard deviation of inter-node *distances* in terms of slots ahead of and behind each of the nodes.

Two trial sets were run with varying numbers of nodes, one using standard hashing to position the nodes in the chord during the join, and the other using explicit placement of nodes at midpoints between adjacent nodes, close to their original hash value. This placement is *locally optimal* in that placement is optimal within the triple of nodes (new node, successor, and predecessor), but that there might exist larger *gaps* in the chord that could better be served by an extra node.

As can be seen in the graph, a significant improvement was gained with this technique, approximately a 20% improvement in the standard deviation of inter-node distances was found.

The benefit of a balanced chord is that files will be more evenly allocated across the nodes in the chord, reducing the likelihood of any single node becoming significantly busier than others.

Due to implementation difficulties, improvements of leave-based balancing were not collected for this report.

6 Lessons Learned and Future Work

6.1 Limitations of the System

In our implementation the size of files is limited to the size of a java byte array. Our implementation does not support arbitrary failures of nodes, nor does it allow concurrent joining and or leaving of nodes. Similarly, our implementation does not allow network partitions. Due to the synchronous nature of implementation, our implementation is only as fast as the slowest link between two of its nodes.

6.2 Lessons Learned

While working on this project we learned not to trust protocols as written without testing them first. While implementing the Chord protocol we found visualization of Chord nodes and their finger tables helpful in debugging the implementation. Similarly, implementing the Chord protocol without messaging helped us to debug our implementation.

6.3 Suggested Enhancements

6.3.1 Security

The Chord protocol as envisioned by its authors is inherently insecure. Applications that implement Chord need to implement some sort of security either application level security or maintenance level security. Application level security enforces file access, user permissions, etc. Maintenance level security deals with the integrity of the cloud topology.

We believe some sort of maintenance level security should be implemented. In its current state, our implementation allows any client to tell any Chord node in the cloud to leave. An adversary could use this flaw in order to force a Chord node to become overburdened. A suggestion to address this flaw would be to allow only the client attached to the Chord node to tell it to leave.

We also believe some sort of application level security should be implemented. In its current state, our implementation allows any client to delete any file in the cloud. Similarly, any client can retrieve any file in the cloud. This leads to issues of privacy. However, as a purely P2P system our implementation does not deal with these issues. However, some higher-level encryption could be used. Any client can share any number of files with the cloud. This leads to a flaw in which an adversary could force the cloud to become overburdened.

We implemented none of the discussed security features.

6.3.2 Locking and Concurrent Join/Leave

Another potential enhancement seeks to overcome one of the main deficiencies outlined by the authors in the original Chord paper, that of simultaneous joins and leaves. Using the standard algorithms, it would be disastrous for two nodes to leave at the same time, because there would be an interruption in the processing of finger table updates, which would leave the cloud in an inconsistent state.

To overcome the problems, the authors suggest a stabilization technique that involves periodic probes of nodes in the cloud to ensure finger table integrity. While the technique promises to reduce messages and ensure integrity, the periodic technique is potentially flawed should things change relatively quickly in a high-transaction environment, and incurs the overhead of additional periodic messages.

A better technique would be the ability to *lock* the cloud during join and leave attempts using a distributed mutual exclusion algorithm. Such an algorithm could take advantage of certain invariants of the Chord protocol:

- Availability - the Chord protocol is intolerant of arbitrary node failures, and therefore full availability is assumed
- Correctness of fingers - the correctness of finger tables is critical to Chord working properly, so any change in the cloud is designed to ensure integrity of the finger tables
- Tree-like nature of fingers - the finger tables approximate a binary tree, where the node is at some intermediate position in the tree and its fingers are its descendants

These characteristics of Chord suggested to the group that the Raymond Tree token-based algorithm might be a good fit for implementing distributed mutual exclusion during joins and leaves [3].

When joining or leaving, the responsible node locks the cloud. The act of locking the cloud may result in the synchronized transfer of a *token*, an arbitrary data structure, from some token holder to the responsible node, such that the operation may continue once the locking node holds the token. Calls to locking the cloud on a node are synchronized, such that only one entity may ask a node to lock the cloud at a time.

To implement this, each node maintains a token_finger (tf) that refers to its finger table entry that most closely describes the holder of the token. This is similar to Raymond's Tree algorithm where the token pointer refers to the correct path along which the token is known to exist.

When a node tries to lock the cloud, it checks if it has the token. If not, it begins a forward search starting at the token_finger entry, and proceeds recursively along until a point where the token holder is found. This takes log(N) messages.

When the token holder receives the request and verifies that it holds the token, it (synchronously) sends the token to the requestor. The former token holder then finds the closest preceding finger table entry that points (forward) toward the token holder, and adjusts its token_finger to point to that finger entry. It then recursively updates its token_finger entry by sending an update message to it to adjust its token_finger entry, and so on, until the immediate predecessor of the new token holder is updated. This takes log(N) messages.

Note that not all nodes are informed of the new token holder. It is possible to create two disjoint sets of nodes, one of correct nodes (CN) that consist of nodes that know the correct finger path to the token holder, and one of incorrect nodes (ICN) that don't know the correct path. Despite this, it can be argued that the token will always be found.

To see how this works, initially acknowledge the variant that at any time, the finger tables are correct and that the token can be found. Observe that immediately after the

token has been relocated, the set of CN all points to the new token holder, and includes the former token holder; also, observe that the set ICN are incorrect because they point to the former token holder. In this sense, the former token holder acts as a "bridge" between the incorrect nodes and the correct nodes; therefore, any query initiated by any node may start out following the wrong path, but will then eventually land on the "bridge" node, which will continue the query along the set CN towards the correct token holder.

Though normally an expected average number of inquiry messages to find the token is approximately 2*log(N) - log(N) to find the bridge node, and another log(N) to find the token holder - a worst-case may degenerate into approximately N messages. The worst case is where the token has been passed among every node, such that no correct node is in the finger table of the query originator. The average case metric approximates the number of messages in Raymond's Tree algorithm [3].

By implementing distributed mutual exclusion, simultaneous join and leave requests can be serialized and handled one at a time, and thus rigorous finger table integrity can be maintained without reverting to periodic probes as suggested by the Chord authors.

Distributed mutex represents the first step involved with making the protocol tolerant to network partitions. While Raymond's Tree algorithm (like many token algorithms) fails to perform during network partitions, quorum-based voting techniques might eventually promise to maintain control during partitions. A quorum-based distributed mutual exclusion technique, combined with replication (not done for this project), may produce a protocol that ensures finger integrity while maintaining partial operability during network partitions.

Refer to the appendix 11 for discussions of the enhancement properties regarding safety, performance, and other claims.

6.3.3 “Gossip” Messaging

"Gossiping" is a term that refers to passive information transfer in a distributed system. In a distributed system, oftentimes the information that a local node needs for operation is held by a remote node; in some situations, that remote node regularly communicates with the local node on other matters. System efficiency could be gained by piggyback miscellaneous operational data on the remote node's messages to the local node, providing the local node a copy of perhaps newer operational data than what it has. The act of piggybacking unrelated operational data is known as "gossiping."

As mentioned previously, an example of gossiping in the implemented system is the conveyance of minimal busy nodes in a “grenade path” during a node leave and rebalancing operation.

Certain other operational information in Chord could be effectively gossiped among nodes. Finger table information is a prime candidate for such a gossip; for instance, the balancing operation seeks to find the least busy node for leaves, or the largest gap for joins, in the cloud - yet it only has access to M other nodes by way of its fingers. While it is possible to send recursive node queries to the rest of the cloud to find the ideal position, this would incur log^2(M) messages which is an undesirable overhead.

On the other hand, if a remote node had the ability to gossip the existence of non-busy nodes or large gaps to the local node, the local node could retain a cache of this gap data as "hints" for balancing. To make use of this during balancing, it would first want to check on the freshness of the gossip by querying the cloud at the place in question; if not much has changed since the remote node heard the gossip, then this could be a big win in that an ideal place was found without having to explicitly query for it. In fact, the ideal balance slot would be found for free.

Other examples of gossip might be owners of frequently-accessed files, so that clients requesting this file of the given node could get the file in less than log(M) time.

The drawback of gossiping is a little extra overhead in message size in order to convey the gossip data, but also the fact that gossip is, by nature, not guaranteed to be timely. Such passive data might have been correct at one time, but conditions could have changed in the cloud between the first emission of the gossip and the receipt of the gossip by the local node.

6.3.4 Other Enhancements

Other enhancements were discussed during the project. One enhancement discussed was quorum-based voting mechanisms for distributed mutual exclusion. The problem with the current Chord implementation is that the system is intolerant of node failures, network partitions, and asynchronous topology changes; to be truly useful in a distributed, non-homogeneous network and operational environment, a protocol such as Chord should tolerate such conditions. Two enhancements to Chord would be to implement distributed mutual exclusion, which would enable simultaneous join/leave requests (and omit the original paper's concept of periodic probing), and the use of a quorum-based voting protocol to provide a robust mutex capability in the face of failures.

Additionally, an application built for operations in distributed nonhomogeneous environments should employ data replication to tolerate network partitions. This work was not done, because replication involves much more complexity than there was time for. Replication should be combined with distributed mutual exclusion to enable operation during network partitions, and safety of application data.

Such changes might make Chord extensible to ad-hoc mobile networks.

7 Conclusions

Using Chord provided an efficient, distributed naming service for our peer-to-peer system. Although problems were discovered in the algorithm's published paper, the fundamental algorithm works and scales logarithmically as claimed. We found that by implementing balanced joins, we could achieve about a 20% increase in load balancing fairness based on the fact that files will be more evenly allocated in the chord. We suggest several improvements for the application and Chord, including security and distributed mutual exclusion, and offer various informal proofs for some of the suggested enhancements.

8 References

[1] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z, Xu, "Peer-to-Peer Computing".

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications",In the Proceedings of the 2001 ACM SIGCOMM Conference.

[3] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," ACM Transacions on Computer Systems, vol 7, Feb 1989, pp.61-77.

[4] The Java SDK API, version 1.4.2. Available at http://java.sun.com, obtained December, 2003.

9 Appendix: Corrected Chord Algorithm

The baseline corrected Chord algorithm with leave() functionality is given below. It is actual Java code; while no attempts were made to replace the "Java" flavor with the standard Algol syntax prevalent in most published algorithms, the Java has been painstakingly constructed to closely match the terminology in the Chord paper. This algorithm only contains basic corrections plus leave() - it contains no other described enhancements.

```
public ChordNode successor() {
  return finger[1].node;
}

public ChordNode find_successor(int id) {
  ChordNode np = find_predecessor(id);
  return np.successor();
}

public ChordNode find_predecessor(int id) {
  ChordNode np = this;
  while (!on_range(id, np._id+1, np.successor()._id)) {
    np = np.closest_preceding_finger(id);
  }
  return np;
}

public ChordNode closest_preceding_finger(int id) {
  for (int i=_m; i>=1; i--) {
    if (on_range(finger[i].node._id, this._id+1, id-1))
      return finger[i].node;
  }
  return this;
}

public void join(ChordNode np) {
```

```
    if (np != null) {
      init_finger_table(np);
      update_others();
    } else { // n is the only node in the network
      for (int i=1; i<=_m; i++) {
        finger[i].node = this;
      }
      predecessor = this;
    }
  }

  public void init_finger_table(ChordNode np) {
    finger[1].node = np.find_successor(finger[1].start());
    predecessor = successor().predecessor;
    successor().predecessor = this;
    for (int i=1; i<=_m-1; i++) {
      if (on_range(finger[i+1].start(), this._id,
                   finger[i].node._id-1)) {
        finger[i+1].node = finger[i].node;
      } else {
        finger[i+1].node = np.find_successor(finger[i+1].start());
        if (!on_range(finger[i+1].node._id,
                      finger[i+1].start(),
                      this._id)) {
              finger[i+1].node = this;
        }
      }
    }
  }

  public void update_others() {
    for (int i=1; i<=_m; i++) {
      ChordNode p = find_predecessor(wrap(this._id-pow2(i-1)+1));
      p.update_finger_table(this, i);
```

```
      }
    }

    public void update_finger_table(ChordNode s, int i) {
      if (s._id == this._id) {
        return;
      }
      if (on_range(s._id, this._id+1, finger[i].node._id)) {
        finger[i].node = s;
        ChordNode p = predecessor;
        p.update_finger_table(s, i);
      }
    }

    public void leave() {
      if (this == successor()) {
            return; // it is the last node in the cloud
        }
        successor().predecessor = predecessor;
      for (int i=1; i<=_m; i++) {
        ChordNode p = find_predecessor(wrap(this._id-pow2(i-1)+1));
            p.remove_node(this, i, successor());
      }
    }

    public void remove_node(ChordNode n, int i, ChordNode repl) {
        if (finger[i].node == n) {
          finger[i].node = repl;
            predecessor.remove_node(n, i, repl);
        }
      }
```

10 Appendix: Enhanced Chord Algorithm

The following algorithm includes enhancements for cloud locking as discussed elsewhere. This is actual Java, augmented to resemble the algorithmic notations in the original paper.

```
/**
 * Augmented Protocol Methods - Token-based Mutex
 */


    /**
     * This method returns an index of a finger, not the finger
     * itself.
     */
public int closest_preceding_finger_index(int id) {
  for (int i=_m; i>=1; i--) {
    if (on_range(finger[i].node._id, this._id+1, id-1))
     return i;
  }
  return 0; // 0 means "this" is closest prec finger
}

public ChordNode closest_preceding_finger(int id) {
      int cpf = closest_preceding_finger_index(id);
      if (cpf>0)
        return finger[cpf].node;
      else
        return this;
}

    /**
     * Creates a token on the local node
     */
    public void put_token(int sn) {
      token = new Integer(sn);
```

```
        }

        /**
         * Lock the cloud - if local node has the token, then
         * locking proceeds; else it is queried for
         */
        public void lock_cloud() {
          if (token==null)
            pass_token(this);
          is_locked=true;
        }

        public void unlock_cloud() {
          is_locked=false;
          is_locked.notify // wake any waiting threads
        }

        /**
         * Updates forward fingers to notify them that "nhid"
         * is the new holder of the token.  Forward fingers are
         * updated to maintain the set CN of correct nodes;
         * by fixing the forward fingers, it ensures that any
         * node querying for a token will eventually land on the
         * proper path toward the token.
         */
        public void update_forward_token_fingers(int nhid) {
      // recursively update forward fingers
      token_finger = closest_preceding_finger_index(nhid);
          if (token_finger>0)

finger[token_finger].node.update_forward_token_fingers(nhid);
        }

        /**
```

```
 * Causes a token holder to pass a token to the requestor,
 * or if it does not have the token, causes the node to
 * search along its token finger for the token.
 */
public void pass_token(ChordNode nh) {
  synchronize; // While is_locked, Synchronize threads here
  if (token!=null) {
// delete my copy
    int sn=token.intValue();
        sn++;
        token = null;
          // "Pass" the token (recreate token on recipient)
          nh.put_token(sn++); // approximates a message
          // update myself and fwd fingers on path to new token
          update_forward_token_fingers(nh._id);
  } else {
    ChordNode next_node;
          if (token_finger==0) {
      next_node = successor(); // I'm sure succ has the token
        } else {
      next_node = finger[token_finger].node; // ask finger
        }
        next_node.pass_token(nh);
  }
is_locked.notify; - wake waiting threads
}

/**
 * Changes to join - lock the cloud during join to
 * allow asynchronous join requests
 */
public void join(ChordNode np) {
  if (np != null) {
    init_finger_table(np);
```

```
            predecessor.lock_cloud();  // lock during a join
        update_others();
      } else { // n is the only node in the network
        for (int i=1; i<=_m; i++) {
          finger[i].node = this;
        }
        predecessor = this;
            put_token(0);   // if first node in, then make the token
                token_finger=0; // token finger points toward self
      }
    }

        /**
         * Changes to init_finger_table - get token finger "hint"
         * from successor.  It is okay for it to be wrong, it
         * will eventually be corrected should the node request
         * the token.
         */
    public void init_finger_table(ChordNode np) {
      finger[1].node = np.find_successor(finger[1].start());
      predecessor = successor().predecessor;
      token_finger = successor().token_finger;
      for (int i=1; i<=_m-1; i++) {
        if (on_range(finger[i+1].start(),
            this._id,finger[i].node._id-1)) {
          finger[i+1].node = finger[i].node;
        } else {
          finger[i+1].node = np.find_successor(finger[i+1].start());
          if (!on_range(finger[i+1].node._id,
                        finger[i+1].start(),
                        this._id)) {
                         finger[i+1].node = this;
          }
        }
```

```
    }
  }

      /**
       * Change to update_others - move successor.predecessor out of
       * init_finger_table.
       */
  public void update_others() {
    successor().predecessor = this;
    for (int i=1; i<=_m; i++) {
      ChordNode p = find_predecessor(wrap(this._id-pow2(i-1)+1));
      p.update_finger_table(this, i);
    }
  }

      /**
       * Change to update_finger_table - relocation of successor.pred
       */
  public void update_finger_table(ChordNode s, int i) {
    if (s._id == this._id) {
      return;
    }
    if (on_range(s._id, this._id+1, finger[i].node._id)) {
      finger[i].node = s;
      ChordNode p = predecessor;
      p.update_finger_table(s, i);
    }
  }

      /**
       * Changes to leave to lock the cloud
       */
  public void leave() {
    if (this == successor()) {
```

```
                return; // it is the last node in the cloud
        }
        lock(); // lock the cloud
        successor().predecessor = predecessor;
    for (int i=1; i<=_m; i++) {
      ChordNode p = find_predecessor(wrap(this._id-pow2(i-1)+1));
              p.remove_node(this, i, successor());
    }
        if (token!=null) { // give token to successor
           successor().put_token(sn++); // approximates a message
        }
        notify; // wait waiting threads
  }
```

11 Appendix: Informal Enhancement Proofs

Theoretical improvements to the Chord algorithm by the enhancements are presented by highly informal "proofs" - discussions, really - of the enhancements and comparisons to the original algorithm. More formal proof technique should be applied, but was not done so due to time constraints of the project.

11.1 Backfilling

The *backfilling* algorithm is safe, achieves locally-optimal balancing, and incurs f+F/2+log(N) messages.

11.1.1 Safety

The algorithm is *safe* in that it satisfies two conditions:

- termination
- avoids deadlock/livelock

To guarantee termination, the backfilling algorithm only queries its M forward fingers during the search for a non-busy node. Once the node is found it is told to leave() and join(), two algorithms that terminate. Since the search and the resulting operation are deterministic, the backfill algorithm will terminate.

Backfilling avoids deadlock and livelock conditions. Deadlock is trivial - there is, at no point, two nodes performing a backfill at the same time. Livelock is more involved. Define the term *busy node* to mean one where the threshold for the number of assigned slots is exceeded, such that it wishes to find a backfill node.

Livelock represents the case where a busy node chooses another busy node as its backfill, which causes the busy node's successor to search for a backfill, and so on. The backfill algorithm uses node spans to ask a "what if" question of its proposed backfill candidate: a backfill candidate is elected for backfilling if and only if its removal from the Chord will not create a busy condition on its successor.

In the case where no such candidate can be found, the algorithm refuses to do backfilling. Furthermore, the threshold check is only initiated at node departure, and since a departing candidate is assured to not trigger another backfill operation, no livelock occurs. Thus, backfilling avoids livelock as well as deadlock, and is therefore safe.

11.1.2 Locally-Optimal Balancing

Backfilling achieves optimal balancing because it chooses the ID of the backfill node to be halfway between the busy node and its predecessor. Since Chord joins with such an

ID represents approximately half the slots assigned to the busy node, and since the hashing function is claimed to have pseudo-random properties resulting in equal probabilities of a file hashing to one of various slots, the halfway mark represents an expected best place for the backfill node.

Further, backfilling achieves only locally optimal balancing because it is triggered by a leave, and there could exist greater gaps in the cloud due to a non-impartial hashing algorithm used during join. The backfilled node is plugging the greatest leave-created gap in the Chord, but there could be greater join-created gaps that are unknown. Thus, backfilling is locally optimal.

11.1.3 Performance

The number of messages incurred in backfilling is f+F/2+log(N), where:

- F is the number of files held by the *busy* node
- f is the number of files held by the backfilled node initially
- N is the number of nodes in the cloud

During backfilling, three steps take place:

1. The busy node sends queries to each of its M fingers, where M=log(N), N being the number of slots in the ring.
2. The non-busy node leaves the cloud temporarily, and transfers f files to its successor.
3. The non-busy node rejoins halfway back from the busy node, and thus receives F/2 files from the busy node.

Backfilling works best when f is much smaller than F; in this case, the number of messages approaches F/2 when F is large and f is small, since log(N) has not much of an impact.

11.2 Cloud Locking

The locking algorithm is safe, free of starvation, and requires 2*log(N) messages to acquire the token in the average case.

11.2.1 Safety

The locking algorithm is *safe* in that it terminates and avoids deadlocks.

Avoidance of deadlocks is easy to accept when dealing with token-based systems. Only the possessor of the token is allowed to lock the cloud; all others must request the token from the possessor, who only releases it when done. Competing requests for the token result in serialization at the requested node; any waiting threads are notified once the

token is either received, or released, from the requested node, and the algorithm ensures correct processing for the freed threads.

The locking algorithm terminates because the token is guaranteed to be found in the cloud in a finite amount of time. Consider two cases, one in which all the token_fingers are correct, and one in which some of the token_fingers are incorrect.

For the correct token_finger case, the token is found in log(N) time, because it follows the same logic as the find_successor call. Eventually, some node will be reached whose token_finger is null, indicating that the successor holds the token. The successor is notified, and the token is transferred, and the algorithm terminates.

In the partially-correct token_finger case, there are two classes of nodes, two disjoint sets of nodes exist, one of correct nodes (CN) that consist of nodes that know the correct finger path to the token holder, and one of incorrect nodes (ICN) that don't know the correct path. The token can be found because the nodes in set ICN will all point to the previous holder of the token; this is assured because at one point in time, all of the node's fingers pointed to the previous token holder. (In fact, the initial token holder is the first node to join the cloud, and all subsequent nodes are updated of that token holder). Since all nodes in the ICN point to the former token holder, the former token holder acts as a "bridge" between the incorrect nodes and the correct nodes. A query that starts in any node in the ICN eventually ends at the bridge who continues the query to the CN set which contains the correct token holder.

There can be the case that the token has moved several times and that there exist several disjoint ICN subsets. For example, consider a token history of A, B, C, where the token started at node A, moved to node B, then moved to node C. There might exist a set of nodes a1, a2, ... ai that point to node A, a different set of nodes b1, b2, ... bj that point to node B, and a third set of nodes c1, c2, ... ck that point to node C. Note that because node A passed the token to node B, node A thinks that node B has the token. This, in effect, makes node A *the same node* as node b1. Similarly, since B has passed the token to C, B is really *the same node as* node c1. This means that any path that ends in ai will point to b1, and any path that ends in bj will point to c1, and any path in c1 will point to C, the true token holder; but this implies completeness (closed mapping) among all possible paths in the cloud that end at the token holder, therefore the token is found in finite time (finite numbers of messages) even though numerous nodes have incorrect token finger information.

11.2.2 Free of Starvation

The cloud locking algorithm can be made fundamentally free of starvation in its implementation. The term *starvation free* means that every node requesting access to the token will eventually get it.

By itself, the algorithm is not free of starvation - for instance, a Java implementation might use java.notify() to wake waiting threads, and this call makes no guarantee of FIFO processing. In Java, is possible for a greedy node to continually rerequest the token and end up first in the notify queue by chance.

Any token scheme can be made to serve competing accesses to the token in first-come, first-served order. In the implementation of this particular algorithm, the limitations of Java can be overcome through the use of an ordered queue of thread ID's, and using notifyAll() to wake all waiting threads once some thread completes. When woken, each of the woken threads can check the queue (synchronously) to see if its thread ID is at the head of the queue; if it is, then that thread executes and the rest continue to wait [4].

Provided that requests are served in FIFO order, the only other problem is ensuring that requesting nodes will eventually receive the token in finite time. Since this was justified above, all requesting nodes will eventually receive the token, and therefore will not starve.

Making a FIFO condition on the implementation will ensure the algorithm is free from starvation.

11.2.3 Performance

Since it is log(N) calls to find the bridge within the ICN set, and log(N) calls to find the token holder in the CN set, this is 2*log(N) in an average case. In the case where there are multiple disjoint ICN subsets, this performance gets a bit worse. The worst case is one in which there are N total nodes, one bridge node, and N-2 nodes in the ICN, each in their own subset (subsets consisting of one node each). In this case, the number of messages reverts to N; however, this case is statistically unlikely because it would require that the token finger of each node in the ICN set would avoid all nodes in the CN set.

Furthermore, a simple extension whereby a node notifies its predecessor when it receives the token would potentially reduce the set of ICN by half, resulting in closer to the average number of messages 2*log(N).

The average case performance is close to that of the classic Raymond Tree algorithm.

Unfortunately, due to time constraints, further probabilistic worst-case performance times were not analyzed.