ASYNCHRONOUS SYSTEMS COURSE PROJECT, FALL 2013, PROJECT REPORT

Chord DHT

Chidambaram Ramanathan - 109079125
Chidambaram.ramanathan@stonybrook.edu

Table of Contents

Project Description and plan

The main goal of the project is to implement the Chord protocol in several languages and prove the correctness of the same. The implemented protocol should be able to handle insertion of keys, deletion of keys and lookups of keys. All these operations should not be centralized and be completely distributed. Once this structure is implemented, the end result will be measured of how quick a data is retrieved from the key supplied by the driver program. The implemented algorithm should be correct, live and fair.

Approach

To implement all these things, a team of 13 members were chosen and each individual had a separate task. I have chosen to optimize the basic version of Chord protocol and add small additions to it so that the lookup is faster.

Role in the team

I have taken up the role of optimizing the existing algorithm.

Motivation

I chose to implement the optimizations in DistAlgo [10] which is a high level language to implement distributed algorithms. Doing the implementation in DistAlgo allows the programmer to focus more on the logic of the algorithm rather than focusing on the syntax and semantics of the language. Thus the end product would be an improvised algorithm which is very concise and easy to read. Doing in DistAlgo makes the person who reads the code understand better and hence any future additions to the code will be easy. Thus doing the implementation in DistAlgo not only helps us understand the algorithm better but also helps the programmer to come up with an efficient algorithm.

My motivation to choose optimizing the protocol is that, I strongly believe understanding an existing algorithm and then making additions to it will not only be challenging, but also will allow me to appreciate the performance of the algorithm once the implementation has been done. This way, I am understanding the implementation of the algorithm which is by far the best known method to understand anything and I am also able to think ahead of the task instead of doing something which is already done. I am using the papers [1] in order to understand the exact working of the protocol and the optimizations that are supposed to be made.

Previous implementations

Chord is not previously implemented in DistAlgo. The closest implementations were as follows.
Pyrope [19] – An open source and MIT licensed implementation of Chord in python.
OpenChord [20] – An open source implementation of Chord for distributed hash table.
PythonDHT [11] – An open source and GPL licensed implementation of Chord
Donut [12] – A robust Distributed hash table based on Chord

Input and Output

In addition to the distributed nodes, there is a driver program which takes care of monitoring the network. It also drives the nodes by sending a request, waiting for the response and then finally evaluate the performance of the Chord ring. This driver plays a major role in evaluating the performance.

Input
The input for the problem statement would be given by the driver. The driver will determine the size of the ring, how many nodes will be a part of the ring, what keys to insert and what keys to look for. The driver will wait for the ring to stabilize. Once the ring stabilizes, the driver inserts a key and queries for the key. The keys and data are inserted through a consistent hashing mechanism.

Output
Once the driver finds that the query is answered, it calculates certain parameters for performance calculation such as mean path length, the response time to get the query, the number of hops taken and other critical parameters which reflect efficiency.

Project Plan
Week 1
During week 1, the project proposal was written. It was then revised as per comments. In order to arrive at the proposal, a research in the internet about existing work was done in order to analyze what part can be chosen. Once the role was decided, and extensive research of the existing papers [13], [14], [15], [16], [17] for Chord protocol was done. On receiving feedback, another research about backup plane was made and updated in the document. [18]

Week 2
Since my role involved implementing optimizing the existing version, it was imperative to have a thorough understanding of the existing implementation. The paper [1] was used as the main source of learning. Further, searches in the internet yielded other resources from which a firm understanding of the protocol was gained.

Week 3
Effort was spent to come up with the design for the project. The optimizations to be done were decided. Then the classes and methods we are designed. An idea of how the algorithm was going to work was framed. In order to understand the implementation of the original algorithm being done, there was a brainstorming session with the developer of the algorithm. Once the path was clear, the methods which were to be used for the optimization was decided and an overview of the flow of the algorithm was formed.

Week 4
Once the implementation was handed in, special efforts were made to understand the code given. Since my goal was to optimize the algorithm, it was very important that I understood each and every line of the code. This also required that I compared the implementation with those proposed in the papers and make myself clear. Once this was clear, the implementation was started.

Week 5
The optimizations which were proposed were attempted. Though all were not possible to be implemented in the time frame, best efforts were made to achieve the targets proposed. Implementing

Week 6

The results obtained were documented and the report was prepared

Benchmarking

In order to compare the results, the initial version handed was used as the reference. Test cases were identified to test if the optimized version was better than the previous version. The main attributes chosen were the mean path length, number of hops taken and the response time obtained by the driver to obtain a value associated with the key.

State of the art

Chord

Chord is one such system for a P2P network which implements a DHT. There are other similar systems like Kademlia [4], Pastry [5], Tapestry [6] and CAN [7]. The Chord protocol [8] consistently maps a key onto a node. Both keys and nodes are assigned an m-bit identifier. For nodes, this identifier is a hash of the node's IP address. For keys, this identifier is a hash of a keyword, such as a file name. Chord is simple and handles concurrent node joins and failures as well. Chord can be used as a lookup service to implement a variety of systems. It uses a distributed hash function which distributes the key and values widely across the network, i.e it is fully distributed. This makes Chord robust. The cost of a lookup is very less in Chord. It is just the logarithm of the number of nodes. This is a huge performance efficiency for large networks. When there are thousands of nodes, Chord efficiently routes the packets in very minimal time. This improves efficiency to a great extent. Chord is also resilient. When a new nodes joins the network or leaves the network, Chord automatically adjusts its routing table.

Value Addition

Even such an efficient algorithm suffers from some disadvantages. An excellent application of Chord would have been implement it in a search engine. But Chord is not favored for implementing a search engine. The version of Chord which is prevalent supports only exact match of the values and does not support any similar matching mechanisms. Chord can be used only when the user knows what he/she is searching for. It is also not suitable because, specific information about desired data is needed to compute the query's hash value. The very efficient lookup algorithm also has one disadvantage. The lookup always happens in a clockwise direction. Thus, if a node with a value is just few hops from the source in the anticlockwise direction, the query has to travel through the entire ring before it reaches the intended destination. A simple optimization such as letting the source node to choose in which direction it must send the query improves the efficiency of basic version of Chord.

Distributed Hash Table

A distributed hash table [2] is a decentralized distributed system that provides a look up service similar to a hash table in which the key value pairs are stored in the DHT pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This

allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. Most DHT's use some form of hashing called consistent hashing. Consistent hashing [3] is a special kind of hashing such that when a hash table is resized and consistent hashing is used, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped.

Applications

Peer to peer systems

Chord is mainly used in peer to peer environments where sharing of resources is the main goal. The resources can be any data which the nodes in the network want to share. The Chord protocol efficiently provides location of such resources which are shared in the network. A key is associated with each data item. Chord maps keys onto values. A value can be an address, a document or any arbitrary data item. The peer which is in need of the data issues a request to find the node which has the key and the corresponding node replies back. The real use is appreciated when the network state is volatile. It is very common for a node to join and exit the network at random times. Since it is a distributed hash table, the information about the keys and the nodes is spread across and hence the protocol becomes robust and handles failure scenarios well.

DNS Ip Look up service

Another excellent application of Chord protocol is the DNS IP lookup service [1]. It hashes each hostname to a key. DNS is a lookup service where a host name is supplied and an IP address is returned. Chord does a similar work to this. It provides the same service by hashing a host name to a key. The added advantage is that, now it does not require any centralized server like a DNS. DNS explicitly requires a management service to store the routing information whereas Chord automatically stores the information in them. Chord imposes no naming structure like DNS. Chord can be used to find data which are not tied to any particular machine unlike a DNS server.

Co-operative mirroring

It is a load balancing mechanism where a product's availability is of importance. This becomes an apt application for Chord which allows many computers to load balance instead of all the work being done by a single centralized server [8].

Time-shared usage

This is a mechanism in which a computer's data is distributed throughout the network so that its data is available even if it exits from the network abruptly.

P2P file transfer

This is a mechanism where a file's data which is needed is spread across the network. It reduces the load on a node as the data is distributed across the networks. This also allows offline file access where the node which was sharing the data exited the network.

Design

Algorithm

Overview

Chord is a scalable peer to peer look up protocol. Each node in Chord requires only very few routing information. The algorithm specifies how to find the location of keys. It creates a consistent hashing and yields the IP address of the node responsible for the key. Each node maintains a finger table which specifies a link from itself to the next desirable node where the lookup has to be passed on to. The important aspect about this is that, each node maintains only at most 'm' entries in the finger table where 'm' is just the logarithm of total number of nodes. It has to be noted that the base of the logarithm is 2. Once a query is received, the node scans its finger tables and gets the node to which the request has to be passed on to. In case the value is contained by the node itself, then it returns the value to the node which made the request. This happens throughout the ring till the value is found. To achieve this, each node maintains a list of successors. In the event of a node failure, the nodes adjacent to the failed nodes are notified. On such a notification, the corresponding nodes adjusts its finger tables so that any new request is now passed on to the node which is alive.

In depth explanation

The chord protocol is a solution for connecting the P2P network. The main advantage of Chord is its simplicity and its robustness. Also, Chord is famous for its provable correctness [9]. A ring is formed logically between the nodes which is a part of the network. The nodes are numbered from 0 to 2^(m-1) where m is the logarithm of the total number of nodes. The keys are distributed among the ring. Each node knows who its successor is and also knows what key is associated with the successor. The keys are assigned to the nodes in the following manner. A key is assigned to a node 'n' when the id of the node 'n' is greater than or equal to the id of the key 'k'. This method is followed for all the nodes in the ring. This method sufficiently improves the lookup efficiency of the protocol. If it were to store a normal routing table, then almost liner time would be needed to search a key. But this method searches the key in just log(n) time complexity. Chord achieves this by maintaining a finger table which contains maximum 'm' entries. The 'I'th entry of this table will contain the address of the successor ((n+2^{i-1}) mod 2^m) [8]. Thus the number of entries in the finger table becomes just log(N) where N is the total number of nodes. We can see how efficient this becomes when there are thousands of nodes. Even the storage complexity is log(N). This makes Chord a much attractive algorithm for peer to peer systems.

Node Join

Each node when it newly comes, sets up the initial parameters and tries to join the ring. This is handled by the join_ring() function. Once the nodes the join the ring, it need to communicate with the other nodes and need to stabilize so that the finger tables of each node

maintains consistent information. To stabilize, each node sends a request to its successor to find the predecessor of that node. Once the predecessor is identified, it notifies that node about the new node's existence. The predecessor node on receiving the notification message, updates its successor as the new node which has just joined the ring. This is crucial part of the algorithm as this is backbone step of populating the finger table.

Updating Finger tables

Once the ring has stabilized, the nodes populate their individual finger tables. It finds a successor and predecessor for each entry in the finger table. Next time when a request arrives, it is only from this table, the nodes search for data. Once updating the finger tables, the nodes periodically check if they are alive and if it finds that any of the node is dead, it dynamically updates its finger
tables to record the loss of a neighboring node.

Node exit

Similar to when a node joins, the case of a node exit is dynamically handled by Chord. Since it periodically tries to synchronize its finger tables , it will not receive a response and hence consider the node dead.

Find Key

This is the main part where my optimization resides. In the initial algorithm, the algorithm does not make any intuitive decision and forwards to the entries in the finger tables. In the case, where the destination node is close to the source, but in the anti-clockwise direction, it will better for the source node to send the request in the anti-clockwise direction so that the full traversal of the ring can be saved.

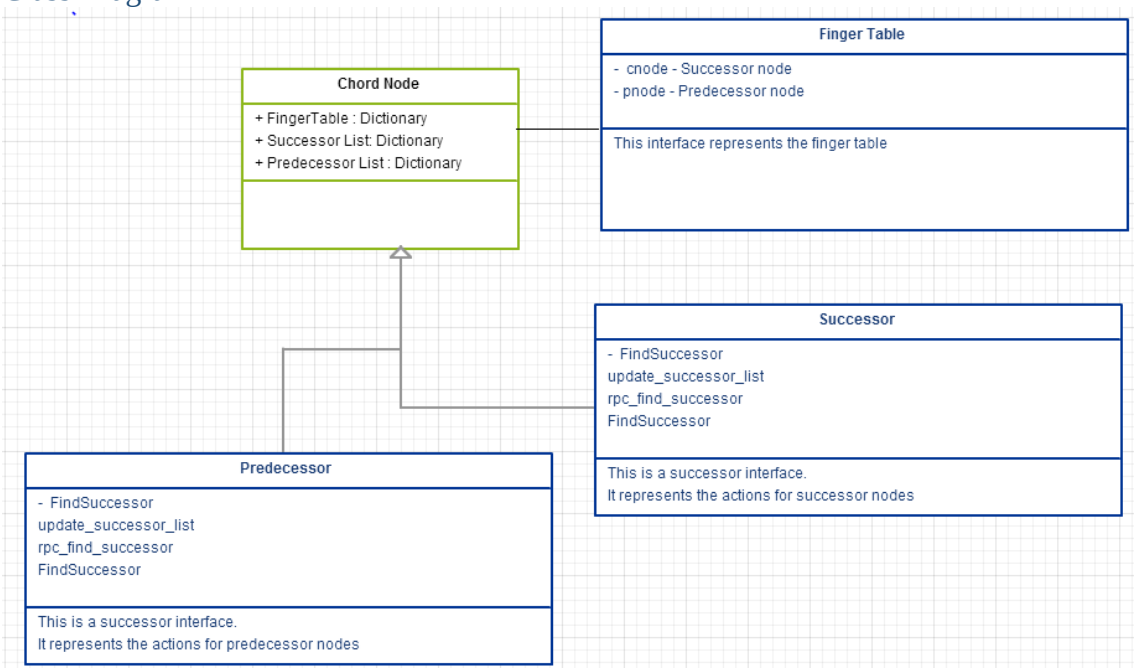
Design Details

The design document [21] has been uploaded in the course website. It contains the main classes and methods used for optimizations. It is to be noted that, the base code given by Sourabh Yerfule [25] has been extended to achieve this. The pydoc [24] also has been uploaded in the course website.

Brief summary

The design mainly includes ideas to implement the optimizations. One of the main idea is to add an optimization for the lookup done by the original Chord algorithm. Initially, when a node receives a request, it checks its finger tables to find the intended destination. Once a match is found, it sends in the clockwise direction. The proposed idea tries to reduce the time taken for this lookup. If the node which has the key is close to the source but in the anti-clockwise direction, then it is intuitive to send the request in the closes possible direction. This will greatly improve performance in a real time network. This was also improve the mean path length. Thus the load in the network is reduced greatly. Also each node can now make a decision of sending the request in the closest direction.

Class Diagram



Core Modules

Class ChordNode

This is the main module which represents a Chord node. Optimizing the protocol means to optimize this class. The finger tables created are a part of this class. The class has a method called FindKey which handles the message sent by other nodes in order to find a key. This method enables the ChordNode to make decisions on sending the request to the closest node. This applies to all the nodes in the chord ring so that everyone make a decision consistent with each other.

Finger Table

This is the central part of the chord node which maintains information about the successors and the predecessors. There is a separate table called the reverse finger table which stores information about the predecessors. This table also helps the nodes to make decisions based on the distance from the source.

Implementation

Overview

As mentioned in the design, the implementation was done using DistAlgo. It has been tested and run in Linux. One of the main goals in the implementation is to understand the given code in DistAlgo and then perform the optimizations. The code was studied with the help of debugging prints in places of the code and then understanding the flow of the algorithm. Once a grasp was got, the optimizations were implemented. The classes and methods used for the code has been documented using pydoc.

Tools used

DistAlgo – A high level language for implementing distributed algorithms. (DistAlgo 0.6a) and Linux 2.6.18

Development Effort

The main effort involved in developing the code was to understand the existing code handed over by Sourabh Yerfule. It involved few brain-storming techniques with him to understand the exact nature of the code written. Once the code was understood from him, separate efforts were put to thoroughly understand the code. Since optimizing a code requires that the developer needs to understand the entire code beforehand, substantial efforts was involved for that. Once the given code was understood and ready to be optimized, the major efforts were implementing those optimizations. Once the optimizations were implemented, it required some time to fix bugs and make changes to the implementation. Once everything was done, the implemented code was tested for correctness, safety and fairness. When abnormal behaviors were found, efforts were taken to fix it.

Implementation Details

The main part of the optimization is to improve the lookup efficiency. The central idea is to make the node think about the direction in which the request can be sent. In order to achieve this, the implementation uses a reverse finger table. For example, if node A is the finger of node B, then B maintains a separate table for storing the finger table related information for A. Thus when a request is destined to some node in the key range of A and B, then the request can be directly forwarded to A instead of traversing the entire circle. This is done by looking at the finger table for B, for the destination information of A, as well as looking at the reverse finger table for the destination information of A. This comparison gives a huge performance improvement. Certain parameters like mean path length, lookup efficiency time are highly reduced. Further, I maintain a successor list and a predecessor list. This is heavily used for maintenance purposes as, whenever a node leaves the ring, instead of querying the network to find the successor information or waiting to be notified from the predecessors, the nodes by themselves maintains a predecessor list. This list is useful to store information such as the previous nodes in the ring. Thus whenever a node leaves or joins the network, the predecessor information can be easily obtained from the predecessor list. The node makes a decision based on the hop count for the predecessors. It calculates the successor difference and the predecessor difference and when it finds either of them is lesser, then the node sends it to a corresponding location. This ensures that the node closest to the destination is chosen. This also enhances the fairness of the system. Earlier in chord, lookup in a clockwise direction allowed a request to take a long path where it could have been done in few paths.

The successor list and predecessor list [1] are optimizations used to fetch the next live successor and next live predecessor accordingly. The correctness of chord relies mainly on the members of the node. If there are three nodes 10, 14 and 18 in the ring, it is important that node 14 knows about node 18. In case the node 18 fails, then node 14 will be pointing to the wrong node. To handle this situation and increase the robustness there is a successor list and predecessor list maintained. The list contains first ‘r’ successors and predecessors. If a node in the successor list does not respond, then that corresponding entry can be removed from the successor list and replaced. In other words,

node n talks to its successors in order to find its successor's successors and add that information into node n's successor list.
The check_predecessor function periodically checks if the predecessors are still alive. If not then it accepts a new predecessor and updates the list.
Similarly the check_successor function periodically checks if the successors are still alive. If it finds that any one of the successor had failed, then it updates the successor list.
The source code of the implementation [23] incorporates all these changes.
Another parameter while considering the decision making in the involved utilizing the predecessor list and the successor list. The idea of finding the closest node should also consider the nodes in a sparse network. It will be helpful if we find the length of the successor list and the length of the predecessor list and then make a comparison. In this case, it will produce more accurate results
In the case of node failure, we have helper functions which finds the next live successor and the next live predecessor. This ensures that the function does not chooses a dead successor or a dead predecessor thus ensuring the correctness of Chord.

Testing and Verification

In addition to implementing the optimization, I also tested the implementation of the algorithm to ensure that it is correct, live and fair.

Test setup

All testing was done using python3.2.2 and distalgo runtime. It was run on linux 2.6.18 machine and the program was expected to produce results.

Simple Insert Lookup test

The driver program waits for all the nodes in the ring to stabilize and let others know who are the successors and predecessors. Once this event happens, the driver inserts a value into the ring. The driver dictates how to insert the nodes. Once the nodes are inserted, we check if the key

dictionary data structure in that node has this value. Once this is done, it means that the driver has successfully inserted the data.

Simple key Lookup test

The driver program inserts the data into the ring and then supplies keys in order to be looked up. The nodes on receiving the lookup request, scans its finger tables and then makes a decision whether to send the request in the clockwise direction or anticlockwise direction. Once the decision has been made, it sends the request in the appropriate direction and the other nodes also work in the similar direction.

Node deletion

The driver program can kill a node and observe if the chord ring is still correct. Once the node has been killed, the other nodes immediately come to know of it because it periodically scans the ring to check if its predecessors and successors are still alive. If they are not alive, the node would not receive a response. In this case, the predecessor and successor list would be updated correspondingly. Further requests will exclude the nodes which are dead.

Node addition

The driver program can add a node and observe if the chord ring is still correct. On addition of a new node, the other nodes become aware of the new node. Thus the node previous to the new now in the ring now modifies its finger tables to keep the new node as the successor. Once this stabilizes, the driver can send a request to find a key so that the lookup correctly happens.

Reversed Lookup

Test case was set up so that, a ring with 64 members were formed. Once the nodes were joined and then stabilized, the driver program issues a request to search a data. The helper node receives the request and immediately identifies that the request is to be forwarded in the reverse direction. It just reaches for the data in very few steps. It then returns the data to the driver. The driver immediately gets the requests and prints the data.

Correctness

The implemented program was found to be correct by inserting a (key, value) pair, then searching for the value by supplying the previous key. It was found that in all cases the driver got the same value as expected. The main advantage of Chord over other DHT applications is that, Chord works correct even if some of the nodes in the ring fail. The only case when Chord can fail is when all the maximum successors of a node fails. In this case, it is inevitable that the protocol would fail. So this case is not considered for the measuring correctness of Chord. Similarly Chord maintains additional routing information to increase efficiency. Those information should not also be considered for measuring correctness of Chord as long as each node maintains its correct successor.

Credits and Acknowledgements

My implementation is derived from the basic code handed by Sourabh Yerfule [25].

References

1. Stoica, I.; Morris, R.; Karger, D.; Kaashoek, M. F.; Balakrishnan, H. (August 2001). "*Chord: A scalable peer-to-peer lookup service for internet applications*". In ACM SIGCOMM Computer Communication http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

2. Distributed Hash Table (Last modified on 5th December 2013), http://en.wikipedia.org/wiki/Distributed_hash_table

3. Consistent Hashing (Last modified on 2nd December 2013)

http://en.wikipedia.org/wiki/Consistent_hashing

4. Kademlia (Last modified on 5th December 2013)
   http://en.wikipedia.org/wiki/Kademlia

5. Pastry (DHT) (Last modified on 15th March 2013)
   http://en.wikipedia.org/wiki/Pastry_(DHT)

6. Tapestry (DHT) (Last modified on 15th March 2013)
   http://en.wikipedia.org/wiki/Tapestry_(DHT)

7. Content Addressible Network (CAN) (Last modified on May 2013)
   http://en.wikipedia.org/wiki/Content_addressable_network

8. Chord (Peer-to-Peer) (Last modified on 1 October 2013)
   http://en.wikipedia.org/wiki/Chord_(peer-to-peer)

9. Pamela Zave, A correct version of Chord
   http://www2.research.att.com/~pamela/aCorrectChordTalk.pdf

10. DistAlgo (Last modified on 22 November 2013)
    https://github.com/sadboy/DistAlgo

11. PythonDHT, An open source Python implementation of Chord,
    https://code.google.com/p/pythondht/

12. Donut, A Robust Distributed Hash Table based on Chord
    http://alevy.github.io/donut/donut.pdf

13. Xianghan Zheng, Vladimir Oleshchuk (2009), "*Improving Chord Lookup protocol for P2PSIP base communication systems*"
    http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05260610

14. Jie Wang, Zhijun Yu, "*A New Variation of Chord with Novel Improvement on Lookup Locality*"
    http://www.cs.uml.edu/~wang/WangYu.pdf

15. Jaber Karimpour, Majid Moghaddam, Ali A. Nozori (July 25, 2012), "*CLTChord: Improving lookup at the Chord protocol using cache location table*"
    http://www.jscse.com/papers/vol2.no7/vol2.no7.5.pdf

16. "Chord A Scalable Peer-to-peer Lookup Protocol for Internet (April 18 2011)"
    http://www.docstoc.com/docs/76945485/Chord-A-Scalable-Peer-to-peer-Lookup-Protocol-for-Internet

17. Chunling Cheng, Yu Xu, Xialong Xu, "*Advanced Chord Routing Algorithm Based on Redundant Information Replaced and Objective Resource Table*" http://www.meeting.edu.cn/meeting/UploadPapers/1281669590875.pdf

18. Chidambaram Ramanathan, "Project proposal Document" https://docs.google.com/a/stonybrook.edu/document/d/1cA_J1xA7utJcfpqfjgxMaOTTPi7-372nYr80pULQMis/

19. Open source Python implementation of Chord. https://code.google.com/p/pyrope/

20. Open source Java implementation of Chord http://open-chord.sourceforge.net/

21. Chidambaram Ramanathan, "Project Design Document" https://docs.google.com/a/stonybrook.edu/document/d/1gz42qgwpdESgkrSZofEX99hwLMZkGp7HoQBY38ASvn4/

22. Chord Implementation http://www.dcs.ed.ac.uk/teaching/cs3/ipcs/chord-desc.html

23. Chidambaram Ramanathan, "Source code for implementation" https://github.com/ChidambaramR/Asynchronous-Systems

24. Chidambaram Ramanathan, "*Design document created by PyDoc*" https://sites.google.com/a/stonybrook.edu/sbcs535/students/chidambaram-ramanathan/Classes_and_methods.html

25. Sourabh Yerfule, https://sites.google.com/a/stonybrook.edu/sbcs535/students/sourabh-yerfule