Simulation of the Merging of Independent Chord Rings

Andrew Rosen Brendan Benshoof

I. Problem Space

The most important component of a decentralized peer-to-peer network is the ability of nodes to find the various files and data stored in the network [3]. While such a task is simple in a centralized network, the challenge is monstrous in a decentralized network. The lookup protocol must be quick: preferably a $logn$ lookup time. The protocol must be robust; able to deal with the inherent volatility of decentralized system, with multiple nodes joining and leaving at any given time. The protocol must also be straightforward to implement.

Hashing protocols such as Kademlia [2], popularized due to its implementation in BitTorrent [1], address these challenges with great success. Our paper examines the implementation of another, older hashing protocol named Chord [3] and some of the difficulties surrounding the implementation.

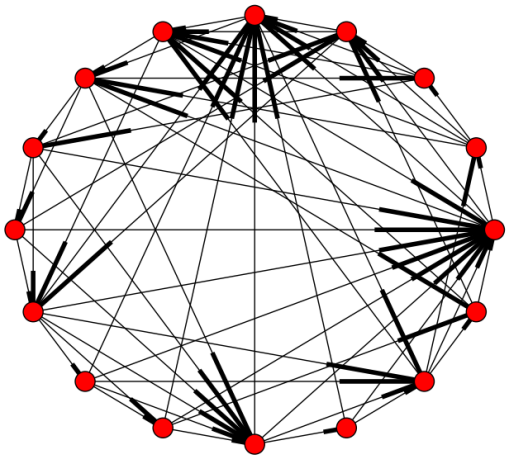

Fig. 1. An example size 16 network produced by the simulation. The lines edges are incoming edges. Note that, unlike in an ideal Chord network, nodes have differing numbers of incoming and outgoing edges.

A. Chord

The Chord protocol [3] takes in some key and returns the identity (ID) of the node responsible for that key. These keys are generated by hashing a value of the node, such as the IP address, or by hashing the filename of a file. The hashing process creates a m-bit hash identifier[1], where 2^m is the maximum number of nodes in the network.

The nodes are then arranged in a ring from the lowest hash-value to highest. Chord then takes the hashed files and places each in the node that has the same hashed identifier as it. If no such node exists, the node with the first identifier that follows this value. This node responsible for the key κ is called the $successor$ of κ, or $successor(\kappa)$. Since we are dealing with a circle, this assignment is done in module 2^m space. For example, if there were some portion of the network with nodes 20, 25, and 27, node 25 could be responsible for the files with the keys (21,22,23,24,25). If node 25 were to decide to leave the network, it would inform node 27, who would then be responsible for all the keys node 25 was covering. An example Chord network is drawn in in Figure 1.

With this scheme, we can reliably find the node responsible for some key by asking the next node in the circle for the information, who would then pass the request through the circle until the successor was found. We can then proceed to directly connect with the successor to retrieve the file. This naive approach is largely inefficient, and is a simplification of the lookup process, but it is the basis of how Chord theoretically works.

To speed up the lookup time, each node stores not just its successor, but also the locations of up to m other nodes in the network a *finger table*. The ith entry of node n's *finger table* will be the

[1]In our simulation, this hash-id is randomly generated for each node.

location of $successor(n + 2^{i-1})\ mod\ 2^{m}$[2]. When a node n is told to find some key, n looks to see if the key is between n and $successor(n)$ and return $successor(n)$'s information to the requester. If not, it looks for the entry in the finger table for the closest preceding node n' it knows and asks n' to find the successor. This allows each step in the to skip up to half the nodes in the network, giving a $\log_2(n)$ lookup time.

Because nodes can constantly join and leave the network, maintenance is essential to keeping the finger tables accurate. To join the network, node n asks n' to find $successor(n)$ for it. Once in the network, n will periodically update entries in his finger tables. In order to adjust for other nodes entering the network, each node n periodically asks $successor(n)$ for its predecessor.

Further details and the specifics of maintenance and protocol can be found in Stioca et al.'s paper on Chord [3].

II. Goals of Modeling and Simulation

We chose to specifically examine the formation period of a Chord Hash. The cited papers discuss how a single node can join a Chord Hash but gloss over the initial state for building a Chord Hash from scratch in a distributed setting. This made general simulation of Chord difficult as at least an initial chord ring had to be "boot-strapped" as a pre-connected initial ring so that other nodes can join it. Because this was an architecture level design problem, we were not interested in parts of the Chord Hash technique like the behavior of the actual network topology, latency, or server capacity. Thus we make a series of simplifying assumptions.

For this simulation we only consider the overlay topology, not the underlying topology of the network, thus we create links with the assumption that nodes have the potential to link into a clique (and in many small network cases the system does actually form a clique). We also ignore file lookup, as we are not concerned with this network's ability to store files beyond being prepared to do so. Because hash values won't be perfectly distributed, it is perfectly acceptable and expected to have duplicate entries in the finger table.

We focus on what servers are connected to each other and where they think the rest of the chord ring is located. We allow ourselves to change the hash degree of the simulation simply to check that changing it does not change behavior, and in practice a value of 64bit, 128bit or higher should be used, which our simulation platform would not support in both integer handling and scale. We chose a maximum size of 20bit for the dual purpose of limiting the simulation scale and to avoid overflow errors in operations on hashes. If we were careful, it is theoretically possible for us to use as high as 31bit in NetLogo (as it does not support unsigned integer type) but we decided this gave us no benefit and would require a serious amount of effort. We model messages in this network as an agent of it's own. This is because a traveling agent "packet" gives us a model of latency and race conditions, two features that do not necessarily need to relate exactly to how they occur in reality but our technique must defend against in the general sense.

III. Developed Models

Our model uses four types of agents: nodes, links representing connections between the nodes, seekers, and updates. Seekers and updates are turtles that represent the messages being passed along the network.

In our simulation, rather than calling nodes with functions, as would be the case in C or Java, we send messages to the nodes that we want to perform tasks. Specifically, we send *seekers* to find successors of a hash-id, and use emphupdates to respond to them. When a node detects a seeker or update destined for itself, it responds accordingly. Handling a seeker entails either sending an update containing the desired successor back to the original sender, or sending a new seeker to the closest preceding node. To handle an update, a node updates its the corresponding slot in its finger table to the new node[3]. This process is show in Listing 1.

The simulation starts by generating a specified number of nodes, which can be arranged into a ring if desired. While the ring shape is optional, it is highly recommended as it better displays the connections in the network. Most nodes begin disconnected from any ring, with a few (again, the

[2]Because hash values won't be perfectly distributed, it is perfectly acceptable to have duplicate entries in the *finger table*

[3]To update the successor is the same as updating the first entry in the finger table

exact number specified as part of the simulation parameters) creating a ring. If a node n can see another node n' that is a member of a ring, n will join n''s ring[4].

```
to find-successor [msg]  ;; node procedure
  if suc != nobody
  [
    let id [seeking] of msg
    let requestingNode [sender] of msg
    ifelse nodeInRange (hid + 1)  [hid] of suc (id)
    [
      let target suc
      ask msg
      [
        hatch-updates 1
        [
          set color sky
          set connectTo target
          set dest requestingNode
          set in-ring [in-ring] of myself
          face dest
        ]
      ]
    ]
    [
      let target closest-preceding-node id
      if target != self
      [
        ask  msg [hatch-seekers 1
          [
            set label ""
            set color blue
            set dest target
            set in-ring [in-ring] of myself
            face target
          ]
        ]
      ]
    ]
  ]
  ask msg [die]
end
```

Listing 1. Netlogo implementation of finding a key.

The simulation is run by the go procedure, shown in Listing 2. First, the seekers and updates move closer to their destinations.

```
to go
  move-messages
  handle-messages
  maintenance
  tick
end
```

Listing 2. The topmost procedure of the simulation.

Of note is that messages do not always travel along the links. Seekers use the links except when a node is trying to join a ring; in this case, there is no link that exists for the seeker to use. Updates, on the other hand, never have to use the links because it is assumes a direct connection between the node requesting an update and the node sending the update.

After the messages have moved, they are handled in the manner described above. The last step is the ring maintenance (Listing 3).

```
to maintenance
  ask nodes with [in-ring = -1]
  [
    if any? ((nodes in-radius Radius) with
        [in-ring != -1])
    [
      join-closest
    ]
  ]

  every Update-Frequency
  [
    ask nodes with [in-ring != -1]
    [
      stabalize
      fix-fingers
    ]
  ]


  every Absorb-Frequency
  [
    ask nodes with [in-ring != -1]
    [
      fix-rings
    ]
  ]
end
```

Listing 3. Network maintenance.

The $maintenance$ procedure is where the original Chord paper was at its most abstract. Our implementation uses the functions from Chord [3], but the timing is of our own devising and the overall structure was adjusted to account for multiple rings and allowing nodes to have "vision" and only initially interact with nodes they could see within a certain radius.

The procedure starts by asking all the nodes not in a ring to join a ring. A node can join a ring only if it can see a node in another ring. Nodes already in a ring first ensure the their successor has the proper predecessor, then fix their finger table. This routine is done every few seconds.

The final part is $fix-rings$, where nodes invite another node it can see to their ring. Vision is a required component here; without it, rings would be unable to join together. Nodes accept this invitation

[4]In the event detecting of multiple nodes n', the closest is chosen

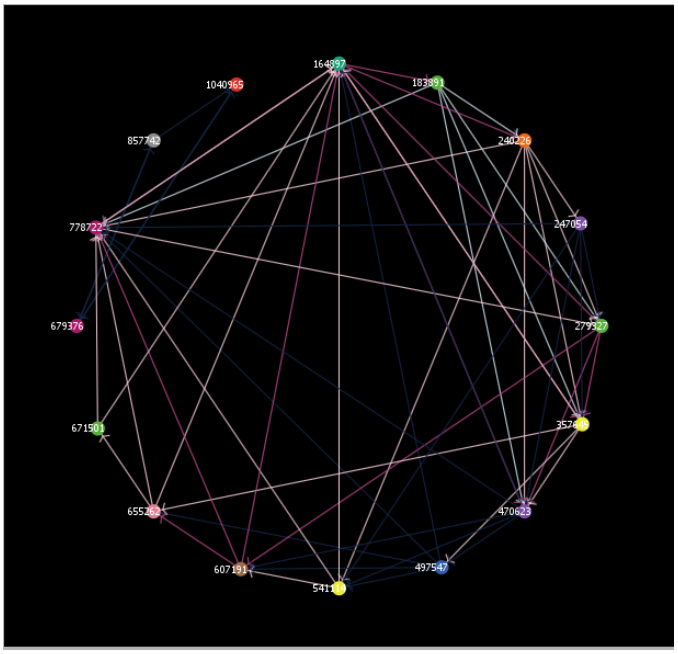

Fig. 2. A chord network with multiple rings. Rings are differentiated by color.

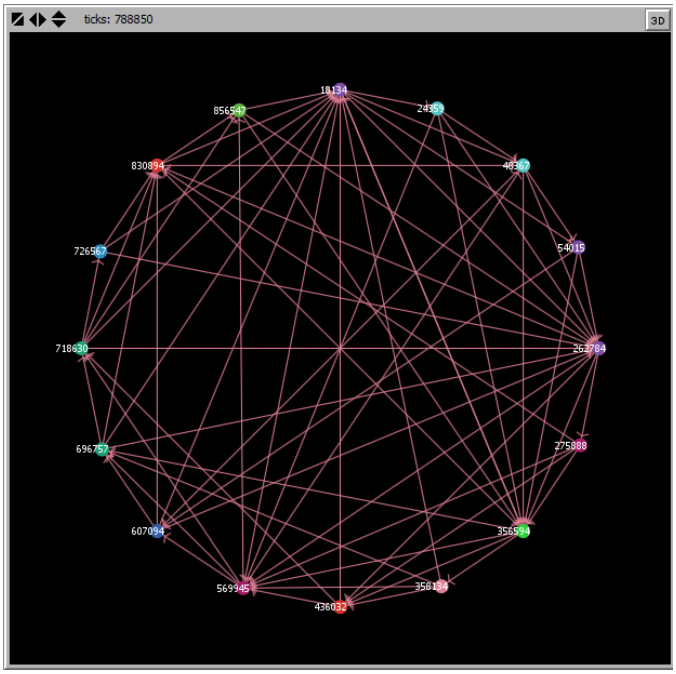

Fig. 3. A chord network where all the nodes have joined a single ring.

if the asker's ring is better. When a node accepts an invitation, it deletes its old finger tables and joins the ring like any new node would. These invitations also happen every few seconds. Figure 2 shows a Chord network with multiple rings, and Figure 3 shows a Chord network after all the separate rings have merged.

In our simulation, a better ring is one with a higher id. This metric is simple and provides a high guarantee of forcing the network to converge on a single ring[5].

IV. Experiments and Results

Our experiment consisted of 200 simulation runs, 100 with our modified technique and 100 with the original algorithm. The number of links at every point in time for all 200 simulations was recorded, and above we graph the mean and variance of each algorithm over time.

Using metrics to discuss our success is difficult as our desired change is difficult. An obvious metric like the number of distinct sub-rings proves our technique works, but does not give us any information beyond that. In the unaltered simulation the number of rings stays constant and in the modified behavior it linearly reduces to one. The major metrics of number of links and average travel time both fall into normal ranges when the undesired behavior occurs. Each node has a constant number of links and the expected number of links is $nodes * number\_of\_links * density$. This number of links is unaffected by our undesired behavior as the total density of the ideal ring is equal to the sum of the densities of the sub-rings. Travel time for messages is unaffected as each sub-ring has sections that maintain the entire hash space, thus a message will be replied to, just by the wrong node.

The metrics we found to be of interest is the rate of link creation and variance in the number of links over time. We found after running 100 simulations for each algorithm the the "end behavior" of the number of links metrics was identical, finding a ceiling at around 300 links, but the behavior of each line differed greatly. The unaltered algorithm gave us a behavior which asymptotically approaches the expected 300 links. The altered algorithm however gave us a linear growth which ceiling-ed at the expected 300 links maximum as shown in Figure 4. This is caused by by a component of the new algorithm that when a node changed rings it had to destroy all of it's current links. This loss appears to cancel out the curve-behavior of the unaltered

[5]This could fail when the vision of the nodes is set to a low value and rings are initialized sparsely.

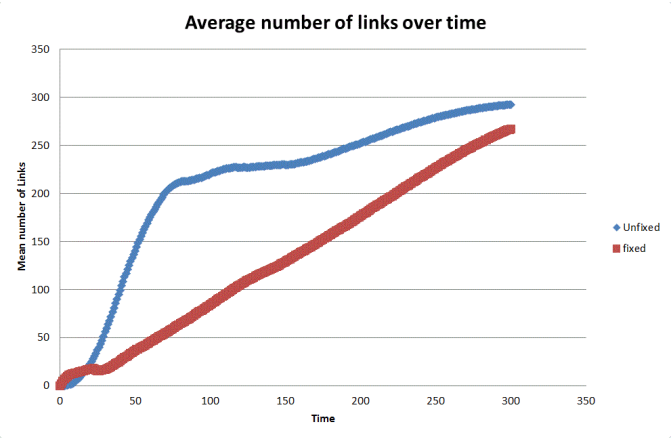

Fig. 4. The unfixed simulation causes nodes to form links more rapidly. This is because nodes are not joining other rings, which would cut the number of links even as more nodes were discovered. Eventually, all nodes are in exclusive rings, which accounts for the slower section of growth.

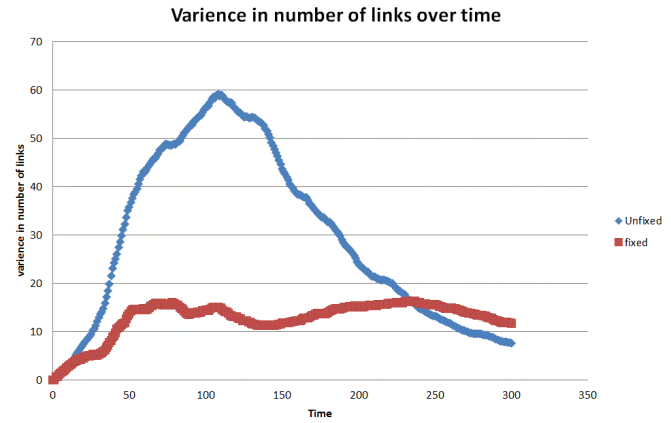

Fig. 5. The number of links is heavily dependent on the initial setup. A lower variance in the fixed simulation implies the values are more predictable.

algorithm.

The variance of the unaltered-algorithm was greater at every point in time then the altered algorithm in increased over time. Figure 5 indicates the unaltered algorithm's behavior was heavily influenced by the random starting conditions of the simulation where the constant variance of the altered algorithm indicated it is more stable in respect to starting topology.

V. Conclusion and Future Work

Our intent with this project was to use our simulation from homework three to solve a problem with the Chord distributed hash. While the final changes to the algorithm may seem simple, the core difficulty with the Chord system is that only its best and worst case behaviors are well defined mathematically. The theoretical model offers minimal aid when dealing with the majority of behaviors.

The simulation proved a very good tool for this process. It allowed us to test and show how small changes to the algorithm would have an impact on the formation of Chord rings. This improvement is an important step to implementing a Chord hash as the basis for a distributed service.

References

[1] R. Jimenez, F. Osmani, and B. Knutsson. Connectivity properties of mainline bittorrent dht nodes. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 262 –270, sept. 2009.

[2] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.