CS740 Assignment 2: Chord

Sophie Stephenson
March 15, 2022

1 My Implementation of Chord

Chord is a distributed lookup protocol that maps keys to nodes that host the data associated with that key. I implemented Chord in Python using Flask.[1] The code can be found at my public github repository: https://github.com/sophiestephenson/cs740_assignment2.

1.1 Implementation Scope

I implemented the most important parts of Chord (Section 4 of [2]): nodes, a join operation, lookup functionality, and all helper functions embedded in this basic functionality.

I *did not* implement the more complex extensions to Chord from Section 5 of [2]: e.g., handlers for concurrent joins or handlers for node leaves and failures. I also did not store any actual data on these nodes, since this would be done by a higher-level application and the design focuses on the lookup of the data's location rather than the data itself.

1.2 Pre-Implementation: IDs Modulo 2^M

A critical feature of Chord is that for a chosen M, each node and data key is mapped to an identifier modulo 2^M. This arranges nodes in a circular fashion and helps to evenly distribute keys (see Figure 2 of [2]). To implement this, as suggested by the paper, I computed the identifier of each node by taking the SHA-1 hash of the node's IP address (mod 2^M).

As I quickly realized, many of the algorithms in this paper involve checking whether a node's ID is in a specific range. This posed a challenge for me because I couldn't just use regular range checking - since we're working modulo 2^M, that does not cover all possible cases. Thus, I implemented my own function (`in_mod_range(item, start, end)`) that checks whether an item is within a specified range (mod 2^M). I also added support for inclusivity and exclusivity for both the start and end of the range.

1.3 Nodes and IDs

Nodes are the foundational building block of Chord. Each node stores data and communicates with other nodes to provide effective lookup. I implemented nodes as Flask apps running locally, each using a different port. The nodes interact by sending requests to each other, either to get data or to tell another node to update its data.

Each running node maintains an instance of my custom `Node` class (see `classes.py` in my code). This class stores the node's IP address and computed ID, as well as its predecessor, its finger table, and whether the node has joined. The `Node` class also contains all basic functions and helper functions needed for a node (essentially, everything mentioned in Section 4 of [2]).

1.4 Inter-Node Communication

In order for Chord to work, nodes must be able to communicate with each other. This is essential for nodes to find out their own predecessors, write their finger tables, and eventually look up the location of data. In my implementation, nodes communicate by sending HTTP requests to each other. When a node receives the request, it either (1) gets the requested data and sends it back to the requesting node or (2) updates its own information as requested by the other node.

Nodes send these requests to each other using the Requests Python library.[2] In the Flask app for each node, each type of request you can receive is implemented as a URL path and connected to an associated function using Flask's `@app.route()` decorators. If the request requires arguments (e.g., `find_successor(ID)`), those arguments are embedded in the request path (e.g., as `127.0.0.1:[port]/findsuccessor/<id>`).

Implementing inter-node communication was one of the trickiest parts of this project. The first challenge was dealing with nodes trying to send requests to themselves. Some node functions, like `find_predecessor()`, send requests across several different nodes until the right one is found; in many cases, this leads to nodes attempting to send a request to themselves. Further, the `update_finger_table()` function recurses over several nodes, and sometimes this can lead to a circular request stalemate. For instance, if a node tells another node to update its finger table, that node may tell its predecessor to update *its* finger table, and the chain of predecessors may lead once again to the original node.

My solution to this problem was twofold. First, in most cases, I ensured that a node did not send a remote request

[1] https://flask.palletsprojects.com/en/2.0.x/

[2] https://docs.python-requests.org/

Algorithm 1 Modified update_finger_table(s, i, orig_sender)

```
if s ∈ (n, finger[i].node] then
    finger[i].node = s
    p = predecessor
    if p ≠ orig_sender then
        return p.update_finger_table(s, i)
    else
        return True
    end if
end if
return False
```

to itself and instead used the current node to do the request. However, this did not deal with the more complex case of `update_finger_table()`, where a request might telescope back to the original requester via multiple other nodes. To take care of this case, I added an *orig_sender* argument and a boolean return value `update_finger_table()`. Algorithm 1 shows the updated version of the function.

The *orig_sender* argument keeps track of the original node which sent the remote request. Before sending another recursive request, the function checks that it is not sending a request back to the original node. If it is, it does not return the request, but instead returns True. This return value then telescopes back to the original sender, which knows it should safely call `update_finger_table()` on itself. If the function does not telescope out to the original sender, then it eventually returns False and the sender knows not to call `update_finger_table()` on itself. This change avoids the circular requests problem.

1.5 Other Choices and Challenges

Algorithm 1 is interesting for another reason: the pseudocode for this algorithm in Figure 6 of [2] is incorrect. Working through examples by hand, I found that the bounds in the first line of this function need to be exclusive and inclusive, rather than the other way around. If you want to verify this problem, comment out line 137 in my `classes.py` and uncomment line 136, then set up the nodes and run my tests. You will find that using the original pseudocode definition of this algorithm yields far worse performance for both correctness and agreement.

Another decision I made was in reference to the `join()` function. This function takes one argument, an arbitrary node that is already in the network. I simplified this in my implementation by designating one node (127.0.0.1:5000) as my starter node. Thus, every node joining the network assumes that the starter node is already part of the network and uses it to perform the necessary joining functions. In real-world applications, this could be easily modified to take in the IP address of any known node in the network.

Finally, to make sure I did not have collisions (while maintaining a fairly small M for simplicity), I hand-picked a list of nodes to include in my network. This would hinder the scalability of the system in the real world, but I also anticipate real implementations would have a much higher M value and therefore collisions would be infrequent. However, in either case, I can't see a good way to catch a collision without asking all nodes what their IDs are - this could be one major drawback of the design.

2 Concluding Remarks

In conclusion, it appears to me that Chord is reproducible. Though there were some small errors in pseudocode, and the algorithms did not explicitly address the intricacies of inter-node communication, the overall design works well.

I would also like to comment that I spent the first half this assignment trying to implement the basic functionality of Hedera [1] using Mininet.[3] I spent around 14 hours fighting with RiplPox code from 2013,[4] trying to make multipath routing work on Mininet, to no avail. In comparison, Chord was much easier to understand and replicate, and I could do it using modern tools—because of that, I actually had fun reproducing Chord. Through this process, I definitely learned the value of easily-replicable research.

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[2] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[3] http://mininet.org/

[4] https://github.com/brandonheller/riplpox