THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

The School of Engineering and Applied Science
of The George Washington University

CSCI 6421 Fall 2019

Assignment 2

Chord System Implementation

Group 1	
Alvaro Albero Gran	aalbero @ gwu.edu
Tyler Jones	tdjones @ smcm.edu
Sean McBride	seanmcbride @ gwu.edu
Naim Merheb	naim @ gwu.edu

Abstract

Group 1 implemented a working version of the Chord peer-to-peer lookup service, as described in the seminal papers by Stoica et al. Given the identity of an existing node in the chord, which is an independent process running on an arbitrary host, the implementation will join a new node to the chord as a new process that is logically between the proper predecessor and successor nodes. The knowledge of the existence of the new node subsequently propagates to the other nodes accordingly. The implementation is in Node.js using gRPC. An explanatory video was posted on YouTube.

Table of Contents

Requirements for the technical report

- ☑ *"include Chord algorithm"*
- ☑ *"your reasoning for choosing an implementation method"*
- ☑ *"design"*
- ☑ *"explaining how we can execute your app"*
- ☑ *"conclusion"*

Evaluation criteria pertaining to the technical report

1. Using reliable references and citing them
2. Writing based on a technical report format
3. Proper explanation
4. Having professional discussion and conclusion according to your reasonable thought in the discussion section of the paper

Introduction to Chord

Chord utilizes a distributed hash function to process data into keys and nodes into identifiers (IDs). Each datum is stored at its key's successor - the node with the lowest ID greater than or equal to the key. Additionally, Chord uses modulo logic such that a key of 7 may have a successor ID of 2 if there is no ID greater than or equal to 7 or less than 2 available. Since IDs and keys are assigned by a distributed hash function, the data should be, on average, spread out evenly.

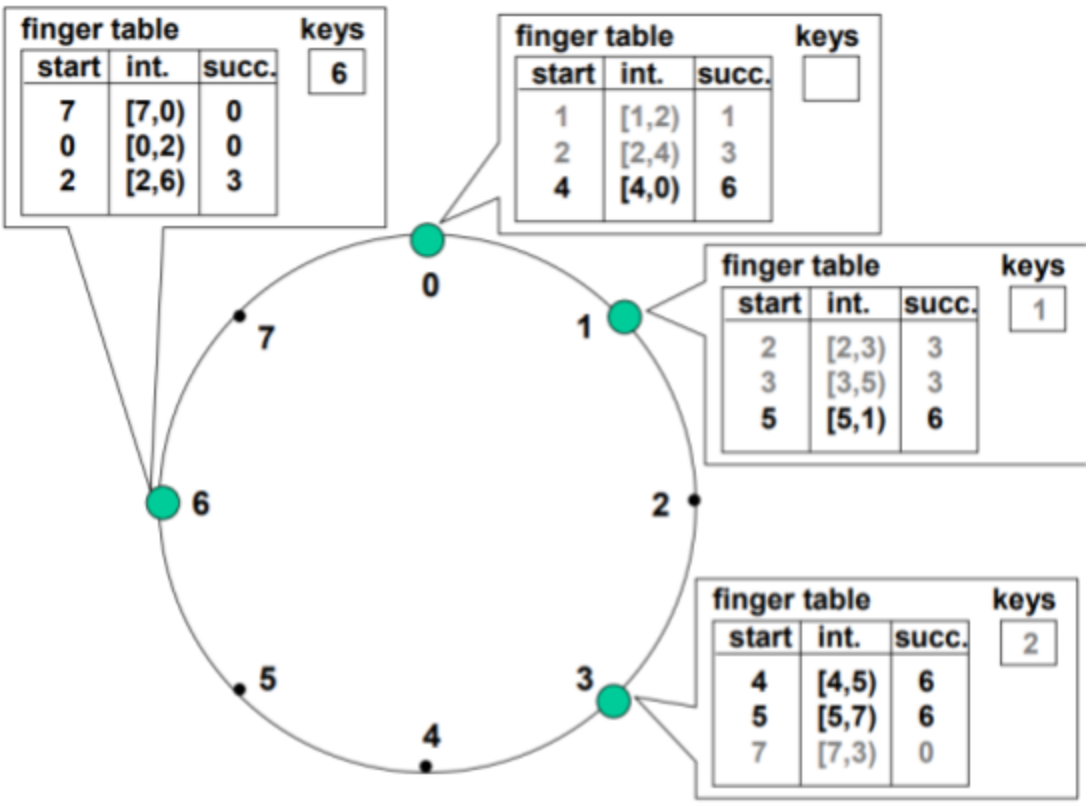


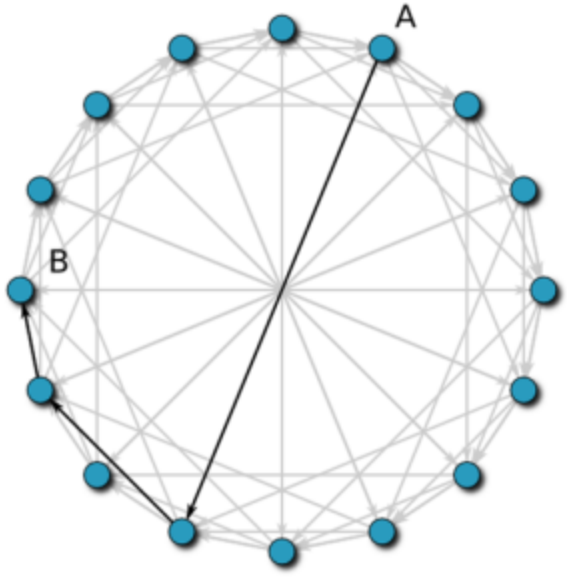
Figure 5.a. from Stoica et al.

Storing a file could be as simple as hashing the file's name to generate its key, finding the key's successor, and passing the file to the appropriate node, which is the node with an ID that is the immediate successor of the key. Similarly, removing a file from the distributed system requires hashing the key based on some known information - such as the file name - and asking the appropriate successor to delete said file.

Adding nodes to the system - known as a "join"operation - and removing nodes from the system - known as a "leave" operation - are only slightly more complicated, at least from the

perspective of pseudocode. Imagine a node *n*, its successor *s*, and its predecessor *p*, which was the predecessor of *s* until *n* joined. When *n* joins the network, it asks a known node of the Chord system for its successor *s* and asks *s* to transfer over all of its data that now has *n* as its successor; i.e., data with keys *k* such that *n* is greater than or equal to *k*. It also coordinates with *s* and *p* to update their predecessors and successors as relevant. When *n* leaves the system, it transfers all of its data to *s* as *s* is now the successor to what was *n*'s data and informs *p* that its successor is now *s*.

The above pseudocode takes for granted that one node can easily find the successor to any node or datum. However, the reality of finding the successor is more complicated, as its impractical for each node to contain a record of every other node. It is also too slow to crawl around the node sequentially. Instead, Chord uses the concept of "finger tables", as explained in the project's accompanying YouTube video, to make jumps that are much further down the Chord. The previous figure shows three fingers for each node because the example was of a Chord with a hash length of three bits. The figure below shows a node "A" in a Chord with hash length of four can quickly forward a query towards "B" by having used its finger tables. Basically, it will forward the query to the furthest of its fingers that still precedes the queried key. That intermediate node does the same, which happens to put the query right at the predecessor of "B". So it took only three steps to get from "A" to "B" instead of eleven. This example of three versus eleven nicely demonstrates the performance improvement that approaches O(log N) steps, where N in this case would be sixteen and three approaches four.



The rest of this document describes the implementation of such a Chord algorithm as part of a group project for the Distributed Systems course at GWU.

Methodology

The team made several key decisions about the overall methodology and design. First, the team decided on using Node.js as a language with C-like syntax that could easily support modern abstract data types and had strong library support to add subsequent features like remote procedure calls (RPC) and user interfaces in further stages of the project. Second, the team decided to do an actual implementation rather than a simulation, which necessitated the selection of the solution for the current delivery. The team settled on gRPC because it is used broadly, which would expose the team to additional marketable skills. For example, gRPC uses Protocol Buffers (Protobuf) for its interface description language and familiarity with Protobufs is itself a desirable skill for many projects doing interprocess communications with paradigms other than through RPC.

The team leader additionally introduced the team to the Visual Studio Code extensible source-code editor based on his extensive experience mentoring budding coders with the tool. One of the key features of Visual Studio Code that helped the group was its ability to support real-time collaboration when extended with the "Live Share" plugin available from Microsoft. Team members would do focused coding on various aspects of the project's algorithm individually but then come together in pair-programming and mob-programming sessions even though the members are scattered in three different cities.

Visual Studio Code also has plugins to interface directly with Git, which along with GitHub, were the chosen platform and technology for code repository and version control. The project's repo is listed in the references.

Design

With the tools and methodology for collaboration in place, the team embarked on implementing the algorithm as closely as possible as described in the pseudocode. Functions and methods were adopted from both versions of the Stoica paper since the IEEE version has additional features for fault tolerance that will be introduced in future deliverables of the project. The function and method names from the pseudocode were preserved, even though the verb-underscore-noun convention is idiomatic neither for JavaScript nor for Node.js. This design choice, however, proved invaluable in debugging the algorithm's implementation since it reduced the mental burden of the translation, allowing the team to focus on the algorithm instead of on the language. Future deliverables will refactor the code to make it more idiomatic - though that may make it more complicated for future students to use as a template for understanding the implementation of the algorithm.

One peculiarity of Node.js is that a local object does not allow RPC calls to itself. Thus, there are wrapper functions for the methods that are called locally and remotely, such as

"find_successor()" and "closest_preceding_finger()". The wrapper for the latter was implemented later in the process and is more elegant as the calling function doesn't have to decide whether to call one method or the other, as was the case with the implementation of the former. That is, while the RPC version of the "find_successor()" is simply a wrapper that calls the local version such that the algorithm doesn't have to be debugged twice, the calling function does have to implement logic to call the local version or the wrapper, which detracts from readability, aside from idiomatic concerns. On the other hand, "closest_preceding_finger()" hides the complexity from the calling function by internally implementing the necessary logic to issue an RPC to its instantiation on the remote node. This will be normalized during the refactoring for the subsequent milestone release.

In order to prevent race conditions, the implementation leverages the - now native - facilities for the async/await pattern in Node.js. Because of the nature of JavaScript, each node is a single thread within its process so there aren't intranode race conditions. The race condition instead arises internode in cases when two nodes may be stabilizing after the join of a third node. The async/await pattern resolved this issue for the most part.

Since the group decided upon implementing a working system instead of just a simulation, each node is a process that could be run on a different host altogether; e.g., a different IP address. However, for security reasons, the group has not yet attempted to instantiate nodes on different hosts as this would require opening ports to the Internet - in the absence of virtual private networks (VPNs), which would add another layer of complexity to an already complicated timeline. Future versions of the project will solve this by launching nodes on separate hosts within a virtual private cloud (VPC) in a cloud service provider. The VPC would limit traffic from the Internet to SSH from the team members for control and monitoring, and the hosts would be free to exchange data within their internal IP space on the necessary open ports.

The pseudocode defines each "finger" in the "finger_table[]" as an object with three elements: a start id, an interval and a successor. The interval is itself a 2-tuple where the second element is the lesser - in modulo arithmetic - of the "start id" element of the subsequent finger and the id of the node. This can be particularly useful to simplify the logic for data searches. Since the current delivery does not implement data storage, the fingers only contain the start and successor elements. Adding the extra object is trivial in the Node.js and Protobuf framework so the team can defer the decision as to whether to modify the data structure or implement the interval check in logic when data features are subsequently added. This is a small example of the flexibility of the project's methodology and design.

Another much more significant departure from the pseudocode was in the implementation of the "stabilize()" function. The pseudocode only accounts for the nominal cases of stabilizing the second and subsequent nodes that join a chord so it doesn't account for the corner case of stabilizing the first node in a chord. As such, an additional "stabilize_self()" helper function was implemented that emulates the effect of the call to the successor's "notify()" method.

At this time, the project does not use the chord to store data. However, the facilities for querying the nodes already exist and, in fact, were leveraged to implement the web user interface (UI). A client application was developed that runs in a separate process from all the nodes and crawls by essentially querying each node as if it were searching for data. The application uses the same Protobuf definitions that the nodes themselves use.

The current implementation already includes facilities to modify the maximum size of the chord when the first node is instantiated by having a configurable length in bits for the output of the hash function. However, it should be noted that the design does not support dynamic resizing. If dynamic resizing where necessary, it would have to be implemented by a helper application that launches a new chord of the larger - or smaller - desired size and systematically offloads data from each node in the original chord and terminates the node as it does so.

While the implementation allows for configurable length, the hash function itself is currently stubbed out through the simple use of user-specified IDs for each node from the command line. This was a necessary simplification to enable debugging since the team didn't have to worry about collisions either in data, hence node ID, space and node-address space. Thus, the call to instantiate each node requires the manual specification of an ID and an address, the latter in the form of an address-plus-port tuple, as will be described in the subsequent section.

Execution

There are two essential Node.js applications in the project: one instantiates a node on a chord and the other instantiates the web server necessary for the user interface. In a bit of comp-sci humor, the former is called "node.js" - as in: "the node.js application is written in Node.js" - and the latter is called "client.js".

The web server application includes the "crawler" that navigates through the chord nodes. In the current deliverable it must be started after at least one node exists or the server application will fail. After the first node exists, the web server can be started at any time but the best user experience is obtained if it is started after the first node since that allows the visualization of the subsequent nodes joining.

As shown in the video and in the project's readme on GitHub, the "node" application takes six arguments: `ip`, `port`, `id`, `targetIp`, `targetPort`, `targetId`. These can be thought of as two tuples of three elements each, where the elements correspond to the Internet Protocol (IP) address of the node's host, the Transmission-Control Protocol (TCP) port at which the node application is listening for connections from other nodes and the node's unique identifier. The first 3-tuple corresponds to the characteristics of the node being instantiated and the second 3-tuple corresponds to the characteristics of a node - any node - that is already a member of the chord that the user intends for the new node to join. The application thus supports multiple chords operating in parallel.

The syntax for the call to create the first node at "localhost:8440/tcp" would be:

```
node node --ip localhost --port 8440 --id 0 --targetIp localhost
     --targetPort 8440 --targetId 0
```

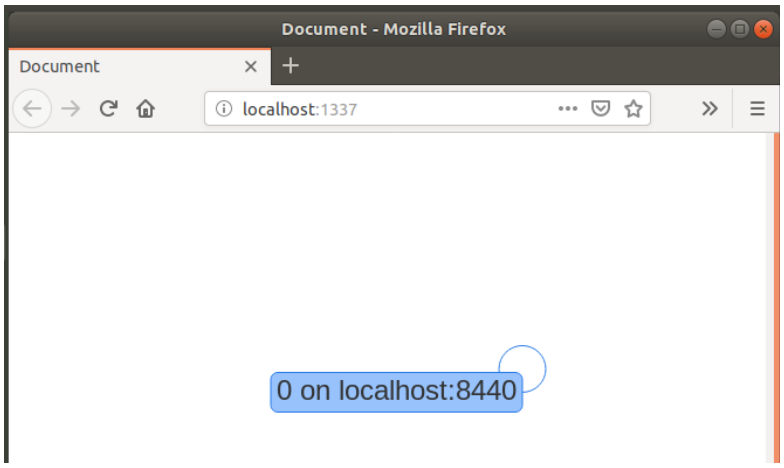
Note that the call must be made from the directory containing the project files or the files must somehow be in the shell's path. Setting the information on both 3-tuples to identical values signals to the chord application that this node is the first one on a new chord, just like in Stoica's pseudocode.

The syntax for the call to start the web server application for the user interface to watch the new chord above would be:

```
node client crawl --ip localhost --port 8440 --webPort 1337
```

This shows that the application takes three arguments: `ip`, `port`, `webPort`. The latter is the port to which a web browser application would connect to provide the user interface, and the former are a tuple with the IP address and TCP port of any node on the chord to display. Since at this point there was only one node at port 8440, that is the port that was specified.

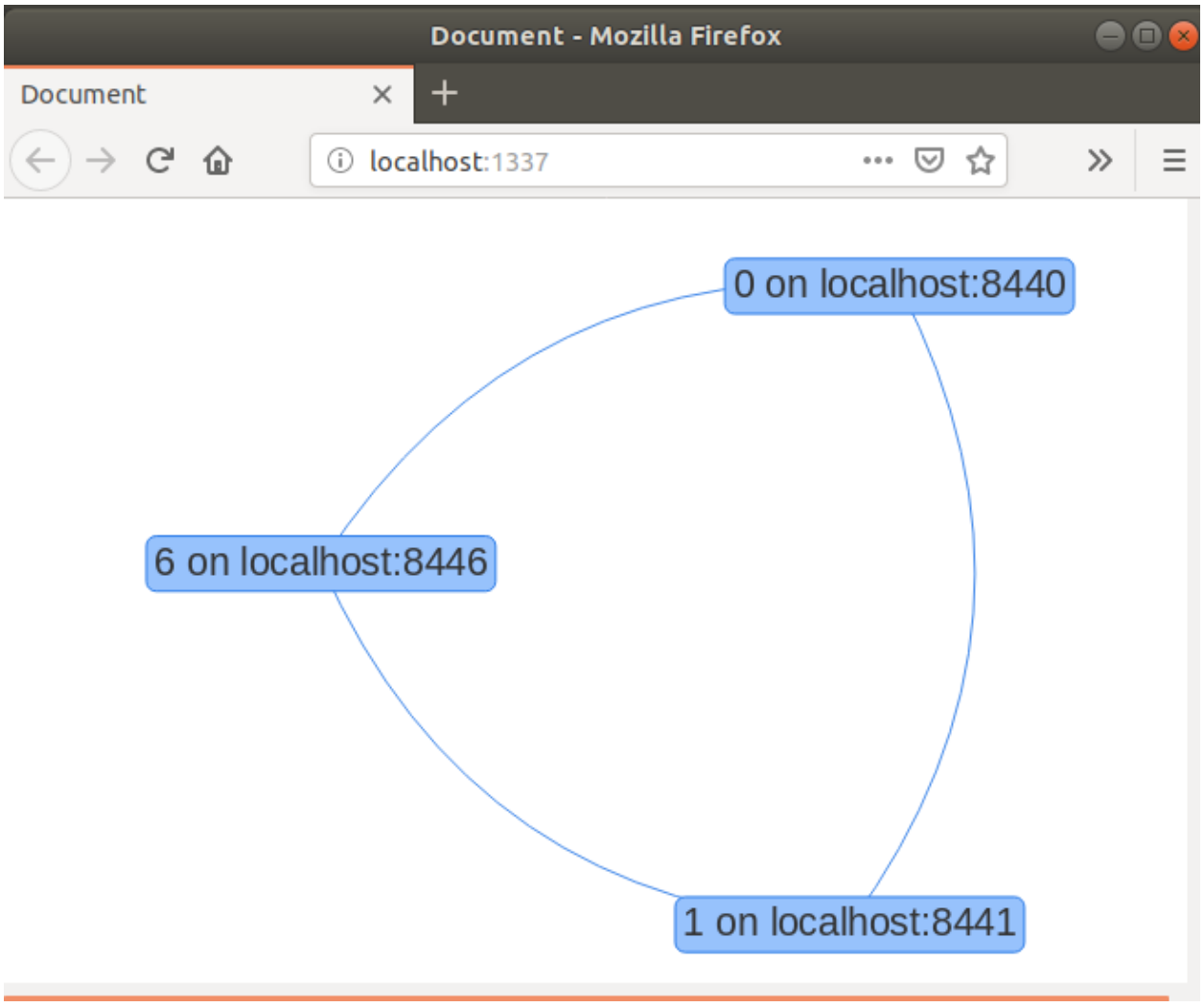
The user should now launch a web browser and navigate to "localhost:1337", as in the following figure:



Additional nodes can now be added to the chord in a visually meaningful way. For example, to add nodes 6 and 1, respectively, the user would issue the following commands on separate terminal windows:

```
node node --ip localhost --port 8441 --id 6 --targetIp localhost
     --targetPort 8440 --targetId 0
node node --ip localhost --port 8442 --id 1 --targetIp localhost
     --targetPort 8440 --targetId 0
```

The user interface now shows three nodes on the chord, as follows:



Helper shell script to to launch a full chord

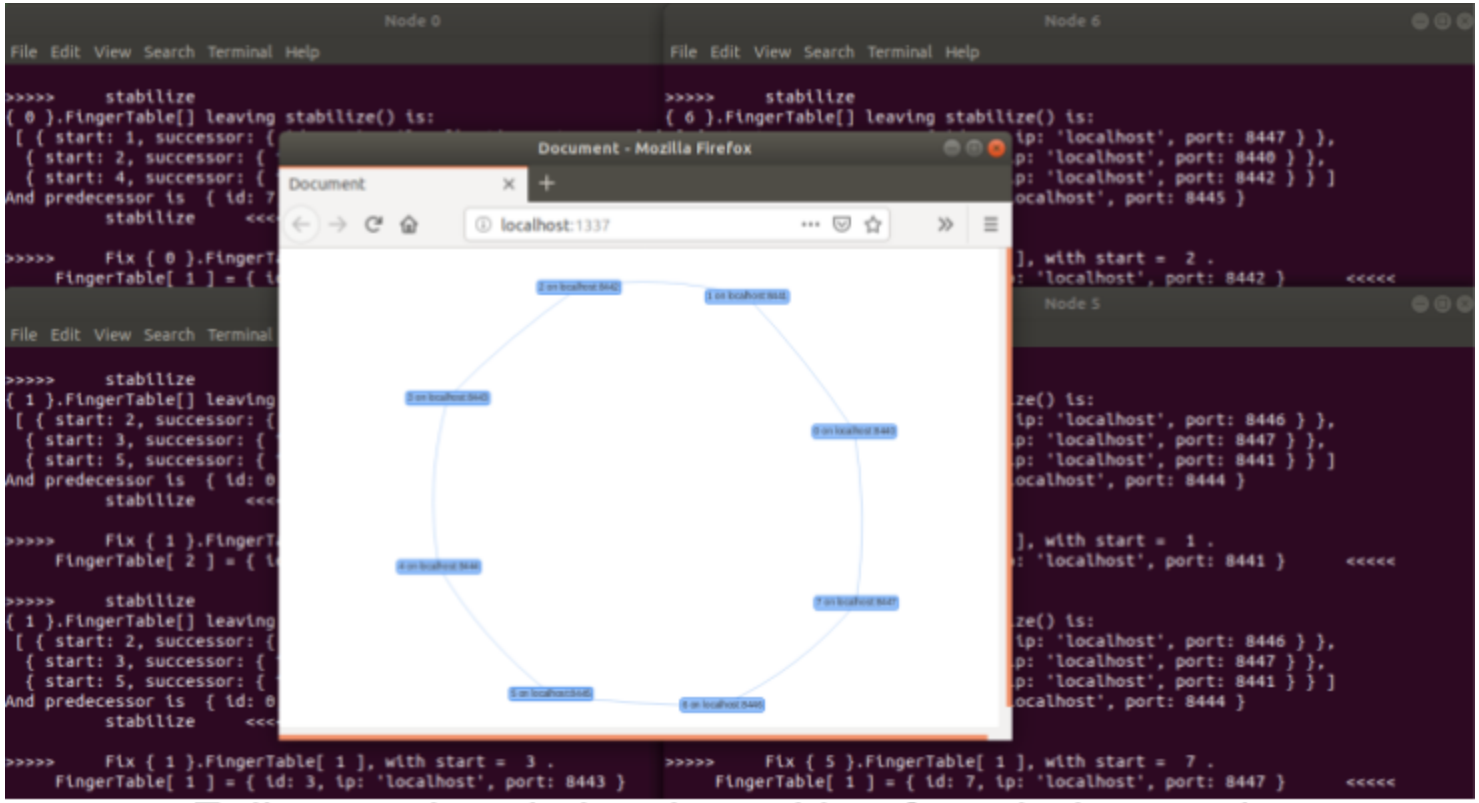
The shell script below conveniently instantiates the nodes on a chord, kicks off the UI server, and launches a browser to show the UI. It assumes that the user is running the GNOME desktop, which is nowadays a reasonable assumption for all except the nerdiest of Linux users, and has the Firefox browser installed. The user must also set the "WORKING_DIR" constant to the value of her directory containing node.js and client.js.

Please be patient while the chord stabilizes after each new node joins since Fault Tolerance hasn't been implemented. The script includes a timed call to the OS "sleep" facility between nodes.

```
#!/bin/bash

# directory containing node.js
WORKING_DIR="/home/test/Documents/chord/chord-grpc"

# port for node 0
#    port for each other node would be BASE_PORT + node's ID
LOWEST_PORT=8440
# number of seconds to wait for a chord node to stabilize
#    intended to be >2 times the period at which stabilize() is called in node.js
STABILIZATION_INTERVAL=21

# first node; e.g., 0 --> 6 --> 1 --> 2 --> 3 --> 7
node_id=0
known_node_id=0
gnome-terminal --geometry=+0+0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep 3

# start the crawler on port 1337
gnome-terminal --geometry=-0+0 --title="Crawler" --working-directory="${WORKING_DIR}" -- node client
crawl --ip localhost --port $(($LOWEST_PORT + $node_id)) --webPort 1337
sleep 3

# browse to the crawler
firefox localhost:1337 &
sleep 1

# next node
node_id=6
known_node_id=0
gnome-terminal --geometry=-0+0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL

# next node
node_id=1
known_node_id=0
gnome-terminal --geometry=+0-0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL
```

```
# next node
node_id=2
known_node_id=0
gnome-terminal --geometry=-0-0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL

# next node
node_id=3
known_node_id=0
gnome-terminal --geometry=-0-0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL

# next node
node_id=7
known_node_id=0
gnome-terminal --geometry=-0-0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL

# next node
node_id=4
known_node_id=3
gnome-terminal --geometry=-0-0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL

# next node
node_id=5
known_node_id=3
gnome-terminal --geometry=-0-0 --title="Node ${node_id}" --working-directory="${WORKING_DIR}" -- node
node --ip localhost --port $(($LOWEST_PORT + $node_id)) --id "${node_id}" --targetIp localhost
--targetPort $(($LOWEST_PORT + $known_node_id)) --targetId "${known_node_id}"
sleep $STABILIZATION_INTERVAL
```

The script launches each node of the fully-populated 3-bit-hash-length chord in its own terminal window, as shown in the figure that follows.



Fully-populated chord resulting from helper script.
Each node is spawned in its own terminal window.

Future work

As discussed in the Methodology and Design sections and in the accompanying YouTube video, the intention is to use this assignment as the basis for the final project and other milestone assignments. As such, the group has already established a roadmap of features to be inserted into the flexible design.

Fault tolerance

The current implementation allows for nodes to be added but not gracefully removed. It is expected that the "stabilize()" and "fix_fingers()" functions from the ACM version of the Stoica paper, coupled with the incorporation of the "check_predecessor()" function from the IEEE version of the paper, already provide a solid foundation for both the graceful and uncooperative removal of nodes. However, the logic in the calls to the remote procedures of the current implementation have to be strengthened through the use of the try/catch pattern. The pattern is already part of the design but the catch statements are mostly displaying the errors in the current implementation as the team was familiarizing with the gRPC framework.

Data storage

An overlay to store and retrieve data will be added on top of the working chord. As part of incorporating this feature, a schema for hashing will be designed and implemented on the chord itself that will replace the user-selected node IDs and addresses.

Security

There is currently no security in the server function that each node essentially implements. As such, one could for example, connect to a node on port 8445 as "`telnet localhost 8445`" or send data to it as "`dd if=/dev/zero bs=10M count=1 | nc localhost 8445`". More practically, the crawler for the current UI makes benign use of this feature to query the finger tables from each node. To improve security, the next release will sanitize all inputs from the various RPCs, which are currently "mainlined" into their expecting variables without any error checking.

Aside from error checking, a future version will attempt to switch from the current model with no authentication using the gRPC ".credentials.createInsecure()" method to one using TLS, such as with the ".credentials.createSsl()" method.

The nodes will also be hosted in a VPC that will use ACLs to protect the nodes such that only authenticated ports are available to the Internet. It should be possible to use SSH port forwarding to test a local node joining a chord mostly hosted in the VPC.

Conclusion

The group successfully implemented a working release of the Chord algorithm from Stoica et al in Node.js. The implementation launches each node into a separate process that can be hosted independently from each other node. Internode communications are implemented using gRPC. An accompanying video, listed in the References, demonstrates the implementation, along the lines of the figures included in this document.

The current implementation lends itself well to the addition of features in upcoming group assignments, as described in the Future Work section.

References

chord-grpc, commit 4b660a1, October 25, 2019. GitHub, https://github.com/bushidocodes/chord-grpc/tree/mobbing/.

McBride, Sean, director. Distributed Hash Table using Node.js, gRPC, and the Chord Algorithm. YouTube, uploaded by Sean McBride, Oct 25, 2019, https://www.youtube.com/watch?v=rhch2dZFcdM.

Stoica, Ion, et al. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications." Applications, Technologies, Architectures, and Protocols for Computer Communication: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2001, pp. 149-60, doi:10.1145/383059.383071. http://delivery.acm.org.proxygw.wrlc.org/10.1145/390000/383071/p149-stoica.pdf (accessed October 09, 2019).

Stoica, Ion, et al., "Chord: a scalable peer-to-peer lookup protocol for Internet applications." IEEE/ACM Transactions on Networking, vol. 11, no. 1, pp. 17-32, Feb. 2003, doi: 10.1109/TNET.2002.808407. http://ieeexplore.ieee.org.proxygw.wrlc.org/stamp/stamp.jsp?tp=&arnumber=1180543&isnumber=26510 (accessed October 19, 2019).

Wikipedia contributors. "Chord (peer-to-peer)," Wikipedia, The Free Encyclopedia, 26 April 2019. https://en.wikipedia.org/w/index.php?title=Chord_(peer-to-peer)&oldid=894264835 (accessed October 26, 2019).

Wikipedia contributors. "Visual Studio Code," Wikipedia, The Free Encyclopedia, 15 October 2019. https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=921352329 (accessed October 26, 2019).