Scalable Peer-To-Peer Key Value Store

Roy Langa

Submitted for the Degree of Master of Science in

Distributed and Networked Systems



Department of Computer Science

Royal Holloway University of London

Egham, Surrey TW20 0EX, UK

June 16, 2020

**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

**Word Count**:

**Student Name**: Roy Langa

**Date of Submission**:

**Signature**:

**Abstract**

It has now become a well-established fact that the expansion of the internet, coupled with the adoption of IoT (Internet of Things) technologies, has also led to a rapid increase in the amount of data being generated. A report by IDC [1]⁠ predicted that the ‘global datasphere’, the total amount of data hosted on the internet, would grow from 33 Zettabytes in 2018 to 175 Zettabytes by 2025. Another noticeable trend is the popularity of a small set of platforms that generate vast quantities of data. Data generated and consumed at high velocity. A 2019 edition of an info-graphic titled ‘every minute of the day’ [2] highlights the data quantities in numbers. For examples, on average every minute, 4.5 million videos are watched on YouTube, 694,444 hours of video is streamed on Netflix 55,140 photos are posted on Instagram⁠.

Storage of data at this scale would run into issues if the traditional centralised storage model is used; such as performance bottlenecks with sudden/gradual growths in demand, a single point of failure either through Denial of Service (DoS) attacks or server crashes. Fortunately, several alternative storage mechanisms have been proposed to handle the peer-to-peer nature of devices on the internet and the transient nature of peers.

This report will look at some of the prominent existing peer-to-peer storage protocols and how these handle the transient nature of devices. Additionally, the report will also document the implementation and evaluation of a proof-of-concept program using an actor model framework specifically the Elixir programming language.

**Contents**

[1 Introduction 1](#_Toc48573934)

[2 Background Research 2](#_Toc48573935)

[2.1 Napster 2](#_Toc48573936)

[2.2 Gnutella 2](#_Toc48573937)

[2.3 Content Addressable Network (CAN) 3](#_Toc48573938)

[2.4 Chord 5](#_Toc48573939)

[2.5 Kademlia 7](#_Toc48573940)

[2.6 Skip Graph 8](#_Toc48573941)

[3 Planning 10](#_Toc48573942)

[3.1 Engineering Method 10](#_Toc48573943)

[3.2 Requirements Analysis 10](#_Toc48573944)

[4 Design 11](#_Toc48573945)

[5 Implementation 12](#_Toc48573946)

[5.1 An overview of OTP design 12](#_Toc48573947)

[5.2 Node Logic 13](#_Toc48573948)

[5.2.1 OTP design implementation 13](#_Toc48573949)

[5.2.2 Node State 13](#_Toc48573950)

[5.2.3 Node Object Representation 14](#_Toc48573951)

[5.2.4 Synchronous Callback Example 14](#_Toc48573952)

[5.2.5 Asynchronous Callback Example 14](#_Toc48573953)

[5.2.6 Finger Table 15](#_Toc48573954)

[5.2.7 Load balancing 15](#_Toc48573955)

[5.2.8 Voluntary Node Departure 16](#_Toc48573956)

[5.3 Communication Layer 16](#_Toc48573957)

[5.3.1 gRPC vs. JSON-RPC 16](#_Toc48573958)

[5.3.2 JSON-RPC Server 16](#_Toc48573959)

[5.3.3 JSON-RPC Client 17](#_Toc48573960)

[5.3.4 Simulating Remote Function Calls 17](#_Toc48573961)

[5.4 Storage Layer 18](#_Toc48573962)

[5.5 Library Entry Point 18](#_Toc48573963)

[6 Testing 20](#_Toc48573964)

[7 Future Work 21](#_Toc48573965)

[8 How To Use My Project 22](#_Toc48573966)

[9 Professional Issues 23](#_Toc48573967)

[10 Self-Assessment 24](#_Toc48573968)

# Introduction

In this project, I will implement the Chord protocol using Elixir

# Background Research

## 2.1 Napster

One of the first and most popular systems to try addressing the large-scale data storage issue was the Napster music sharing service in 1999. Rather than store a large collection of MP3 files on a single server/cluster, which would have had significant costs for the Napster founders, they opted for a peer-to-peer model in which content was distributed amongst the platform users. Although file storage was peer-to-peer, file search was done using a centralised server based on lists provided by each user device for what files they were hosting. To retrieve a file, a user would query the file index server using the file’s well-known name; to which the server would respond with the IP address of a user device storing the requested file. Finally, the file can be downloaded directly from that user device [3].

This storage format reaped Napster a lot of success with 80 million registered users at its peak [4]⁠. However, the biggest issue with the Napster model was that the central file index server was a single point of failure either from Denial of Service attacks or overwhelmed from legitimate traffic. What ultimately affected Napster the most was not technical challenges but legal ones due to the large-scale distribution of copyrighted audio which forced them to shut down their file sharing service on 11th July 2001 [5].

## 2.2 Gnutella

Napster served as inspiration for other peer-to-peer systems that came after it with one example being the Gnutella protocol used by file sharing software like LimeWire; which in turn was referenced in other protocols aimed at being improvements on both it and Napster. The network consisted of nodes connected to a limited number of neighbouring nodes. The node requires a connection to at least one other node on start-up which is done several ways:

* Using a list of existing network nodes that ships with the client
* contacting existing network nodes that serve as web caches

The client will attempt connection to node from these lists until it reaches its connection quota. Addresses that were not tried will be stored while failed connections are discarded.

To query for a file, the client node sends a hash of the file keyword in a search request to all of its neighbours which in earlier versions of the Gnutella protocol was around 5 neighbours with a maximum of 7 hops. When a search result was found, the response was sent back along the search path to the requester.



Figure 1: Gnutella file query process

However, since v0.6 of the protocol, the structure was changed to a leaf and ultra node (peer) structure; with leaf nodes connected to a small set of ultra nodes (around 3) and the ultra nodes having a high degree of connectivity with other ultra nodes (more than 32). When leaf nodes are bootstrapping, they send their keyword hash list (Query Routing Table) to its connected ultra-peers which merge that list with their own and exchange the new list with neighbouring ultra-peers. Search results were sent directly back to the requester using the IP address and port number included in the search request [6].

There were drawbacks however to the Gnutella search the first being it didn’t fully account for the frequent amount of node joins/departures on the internet. Secondly, ⁠the load on each node grew linearly with the network size and number of queries. All of these issues rendered Gnutella unscalable [7][8]. These drawbacks were what gave inspiration for development of Distributed Hash Table (DHT) based protocols which shall be discussed more below.

## 2.3 Content Addressable Network (CAN)

The term, *Content Addressable Network* is a term coined by Ratnasamy et al. to describe a distributed, internet-scale hash-table which they proposed could serve as improvements to peer-to-peer file sharing systems like Napster and Gnutella [7]⁠. The CAN is composed of many nodes each storing a chunk (called a zone) of the entire hash table. Zones are mapped in a virtual d-dimensional Cartesian coordinate space. The node also holds information about a small number of adjacent zones of the table for request routing. Requests will be routed through intermediate nodes towards the node whose zone contains that key.

To join a network, a CAN node contacts a bootstrap node through a DNS lookup. The bootstrap node will contain a list of nodes that it believes are currently online and will give the connecting node a random list of connected nodes. The joining node then picks a random point P and sends a JOIN request for that point via the existing node which will route the request to the node zone where point P lies. The node occupying that zone will split the zone and give one half and its corresponding keys to the joining node. Finally, the new node learns the IP addresses of its neighbours from the node that just assigned it a new half and that node removes neighbours now belonging to the new node. Both nodes will send messages to neighbouring nodes to update them of the changes. The number of neighbours is determined by the dimension size thus this node join affects *O(d)* nodes.

Figure 2: 2-d space with 5 nodes

Figure 3: Node 7 joining the network

To store a key-value (KV) pair <K1, V1>, K1 is deterministically mapped to a point P in the coordinate space using a uniform hash function. The corresponding KV pair is then stored at the node that owns the zone within which point P lies. To retrieve V1, nodes will apply the same hash function against K1 to get point P for data retrieval. A node will utilise its neighbour coordinate set to route messages towards their destinations using greedy forwarding to neighbours with closest to the destination.

When it comes to changes in the network, departures are done by the node handing over its zone and KV records to one of its neighbours. The handover is decided by assigning the neighbour whose zone can be merged with to create a valid single zone; else the neighbour with the smallest zone is handed the departing zone [7]⁠. Node failure is handled using an immediate takeover algorithm that ensures a failed node’s neighbour takes over the zone. The failed node’s data however will be unavailable until the state is refreshed by holders of the data.

Each node monitors its neighbours by listening for periodic messages containing the neighbour’s zone coordinates and its neighbours. Absence of a message for a prolonged time period indicates node failure which triggers a takeover timer. When this timer expires, the node sends a TAKEOVER message to the failed node’s neighbours and on receipt of the TAKEOVER message, a node will cancel its own timer if the zone volume in the message is smaller than its own zone volume; else it will reply with its own zone volume. In a scenario of multiple adjacent node failure and less than half of the neighbours are reachable, a node taking over another zone might lead to the CAN state becoming inconsistent. To mitigate this, the node would perform an expanding ring search for any nodes beyond the failure region prior to running the repair mechanism; thus re-establishing good neighbour state for a correct takeover [7]⁠.

## 2.4 Chord

Chord is another DHT lookup protocol inspired by Napster and Gnutella with the authors suggesting Chord as a potentially good foundation for those earlier protocols [9]. At a top level, Chord maps keys to responsible nodes by applying a *consistent hashing* function to both the data key and the node ID. Consistent hashing uses the SHA-1 algorithm to assign an *m*-bit identifier to a node and a data key hash used by the rest of the protocol for mapping to nodes. Nodes are arranged in an identifier circle (Chord ring)with the range [0,2m-1] where key assignment is modulo 2m.A key K is assigned to the first node in the circle whose hashed ID is equal to K or the next clockwise node, referred to as the successor of K.

Figure 4: Example Chord ring



Each node in Chord keeps record of a set of variables: its own ID, a pointer to its predecessor, the set of keys it is in charge of and a routing table with maximum *m* entries called the finger table; with the node successor stored as the first finger table entry. Eachentry *i* in the table being a node that succeeds by at least 2i-1. When a node wishes to join a network, it will run a few functions that contact an existing node address supplied by the user. Three tasks are executed when joining a network, the first is running functions to initialise the predecessor and finger table entries. Secondly, the finger table entries and predecessors of other nodes in the networks are updated and finally, upper layer software is notified so that the appropriate key transfers can be done.

To query for which node contains a key, a simple but inefficient method would be to pass the query along successors in the ring until the query encounters a node that matches or succeeds that key clockwise; passing back the IP address as a result. The drawback of this is that query might take many hops if the circle is large however, a more scalable method utilises the routing (finger) table mentioned earlier where queries jump around the circle by *N+2i* as shown in figure 6. To maintain correctness of lookups each node must keep its successor pointer and finger table up to date; which is done by running a stabilisation protocol periodically.



Figure 5: Routing with finger table



Figure 6: Simple query routing

Node failures can affect lookup correctness and to mitigate this, each node has a list of successors such that if the immediate successor fails, the next successor in the list is used. This list is maintained in the stabilisation protocol by copying the node successor’s list, removing the last entry and prepending the successor to the list. As Chord was designed for fault tolerance, voluntary node departures are treated as failures with the addition of the node transferring keys to its successor and notifying its predecessor and successor of its departure. The predecessor will then change its successor pointer to the departing node’s successor and the successor will change its predecessor pointer to the departing node’s predecessor.

## 2.5 Kademlia

The third DHT based protocol we shall look at is Kademlia which routes queries and locates nodes with an XOR metric [10]. What differentiates it from other DHT protocols is that configuration information is spread across the network as a side-effect of lookups. Every message is transmitted with the node ID which permits the recipient to learn of the sender’s existence. Additionally, queries are parallel and asynchronous which avoids delays from timeouts if a node fails. Like Chord, Kademlia also uses SHA-1 to create 160-bit node IDs and key hashes. Data is stored in nodes whose ID is closest to that key.

Each node in the network is represented as binary tree leaf with the position determined by the shortest unique prefix of its ID [10]. At each node, the binary tree is sub-divided to subtrees without the node as shown in the diagram below. For each of the subtrees, a node maintains record of at least one node which guarantees that any node can locate another node. To assign values to nodes, Kademlia uses the distance between the key hash and the node using a XOR function.



Figure 7: Example of Kademlia binary tree

Each node keeps a list of <IP address, UDP port, Node ID> triples for nodes in range [2i,2i+1] from itself also known as K-Buckets. Each K-bucket is sorted by time last seen; with least recently seen at the head and most recently seen at the tail. The preference for old nodes comes from research done on Gnutella [11] showed that the longer a node stayed up, the higher it’s chance of staying up for another hour. For large values of i, lists can grow up to size K, K being a system-wide replication value [10]. K’s value is set such that any K nodes are high unlikely to fail within an hour of each other thus allowing for nodes to join and exit while maintaining data availability. On receipt of a message, a node updates the relevant K-bucket for the sender’s ID as follows:

* If the sender ID exists in the K-bucket, it is moved to the tail of the list.
* If the ID is not yet in the correct K-bucket and the bucket has less than K entries, the sender ID is inserted at the tail of the list.
* If the K-bucket is full, the least-recently seen node is pinged and is evicted if no response is received; else the least-recently seen node is moved to the tail and the sender ID is discarded.

## 2.6 Skip Graph

A skip graph is a distributed data structure based on a skip list that functions similarly to tree structures used in distributed systems. One pitfall of DHT systems is that the nature of hashing destroys key ordering properties thus, they lack support for near-match key searches or (efficient) ranged queries. Another issue arises when setting optimal parameters in some protocols like Pastry and Chord for things like replication or stabilization as these require a prior estimation of what the network size or key space. Which is where skip graphs are advantageous as they can be constructed without knowledge of the network size and provide support for key ordering [12].

At the foundation of skip graphs are skip lists, which are tree-like data structures organized in levels of increasing sparse linked lists [12]. Sitting at the bottom is level 0 which is a linked list containing all the nodes in the network in ascending order by key. Skip graphs utilize doubly linked skip lists where each node a predecessor and successor pointer for each level it resides in; with higher level lists being “express lanes” that enable quick traversal of a node sequence. Searching for a node key starts at the top level and drops down a level when the desired not is not available in that level.

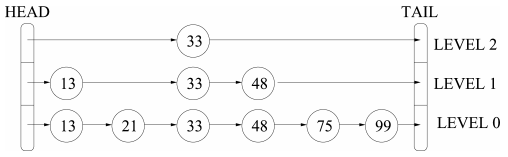


Figure 8: Example of a skip list

The drawback of just using a skip list is that top level nodes are a single point of failure which could partition a significant part of the network and secondly, as nodes are connected on average to *O(1)* other nodes, random node failure can also isolate parts of the network. Skip graphs improve on this by placing multiple linked lists at each level with each node being a member of one list and having *O(log n)* neighbors. This reduces the chance of a node participating in a search in turn eliminating single points of failure and hotspots.

A node’s membership is determined by a membership vector *m(x)* assigned to a linked list. *m(x)* is defined as an infinite random word using a fixed alphabet but it has been observed that on average an *O(log n)* length word is generated. The list at level 0 contains an empty word and for each higher level, a list contains nodes for which a word *w* is a prefix of *m(x)*. A list is said to be part of a level *i* if the length of *w* = *i*.

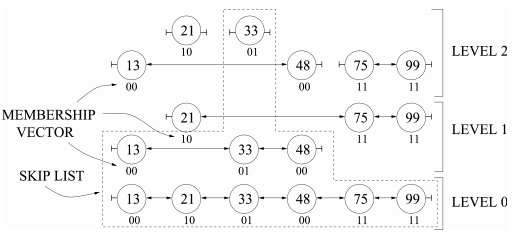


Figure 9: Example of a skip graph

A new node uses an *introducing node* to join the network by inserting itself in one linked list for each level until it is a singleton list at the top level. The first stage of the insert process involves the node running a search operation for itself in the introducing node to discover and link to its neighbors at level 0. Secondly, for each level above level 0, the node locates the closest nodes with the same membership vector for that level and links to them. When a node wishes to leave the network, it simply deletes itself from all lists above level 0 and finally deletes itself from level 0.

Previous work done by the authors of skip graphs describes a graph repair mechanism that they declared inadequate with the worst-case run time being linear in the graph size; in some cases, failing to converge or disconnecting nodes. An alternative mechanism proposed was a generational approach where the skip graph is rebuilt afresh, and existing nodes migrated to the new graph using methods like an initiator node adding the other nodes using the insertion function.

# Planning

This section looks at various parts of the planning process of the project implementation; from a description of the software engineering methodology used to discussing implementation plan formulated during the requirements analysis.

## Engineering Method

The engineering method chosen for this project is an incremental model whereby an initial software model is designed, implemented and tested. With incremental changes being made until the final product is completed [13]. This style reduces the amount of time required to get a working version of the software; making it possible to fulfill the early deliverables deadline of the project. Additionally, this model reduces the overwhelming feeling of implementing the protocol from scratch in a new language as smaller iterations of the software can be tested first, and the project scope can be changed at each milestone if required. The initial model of the software will be an implementation of the base Chord protocol methods and if time permits, implementation of the protocol extensions which improve on the robustness of the system.

## Requirements Analysis

The project requirements were created using descriptions of how the protocol works from the Chord academic paper; first selecting the functions required to create a working version of the protocol with those functions, shown in figures 5 and 6 of the paper being:

* Consistent Hashing of keys and node IDs using SHA-1
* Scalable key lookup functionality using finger tables.
* Functions for node creation, joining a network, network stabilization and lookups.

The paper also mentions extension methods to maintain lookup correctness in the event of node failures or voluntary departures. Methods such as transferring keys during voluntary node departure and maintaining a list of successors in each node to improve lookup robustness in the event of successor failure.

Aside from the technical requirements, it was also important to be mindful of the high-level requirements that the authors set out to solve such as:

* Load balancing of keys across nodes in the network
* Decentralization of responsibility in the network with all nodes being equal
* Scalability with the lookup costs growing as a Log of the network size.
* Availability by maintaining record entries to accurately reflect network changes; ensuring lookup correctness when network failures are not large.
* Flexible key naming with no limits on key structures.

# Design

The Chord authors envisioned their software to be a library that other applications could link to as use which is the direction taken in this implementation. The system takes a layered approach to the different components such as communication for remote procedure calls, the node logic, storage and a top layer as an entry point for external applications to start or stop the network and run lookups.

Library Entry Point

Node Logic

Communication

Storage

Figure 10: Library component layer structure

The advantage of this layered approach gives is that implementations of layers such as communication or storage can be inter-changed while maintaining the same interfaces between the node logic layer.

One extension of Chord to improve load balance among nodes is by using virtual nodes. These nodes are run on a physical node with their own unique IDs; and the IDs within the range between the physical node and the immediate node successor (next clockwise node in the ID circle) IDs. The application design accommodates for this extension because the node logic is encapsulated in its own Elixir GenServer process (more detail on this later). Virtual nodes can be spun up as their own processes and the number of virtual nodes to spin up passed as an argument at the library entry point when Chord is started. Finally, any inter/intra-node communication will be routed to the appropriate virtual/physical node by the communication layer.

Virtual Node

Communication

Physical Node

Virtual Node

External Network

Figure 11: Communication layers between nodes

# Implementation

## 5.1 An overview of OTP design

Before discussing parts of the implementation in detail, it would be helpful to get a quick understanding of the higher-level design principles that the modules are implementing/following. As mentioned in the design section, the authors envisioned their protocol to be encapsulated in a library to be used by other applications. The best way to do this in Elixir is by implementing ‘Application’ behavior. According to the official documentation [14], applications are the idiomatic way to package software in Erlang/OTP; carrying similar behavior to libraries observed in other languages. OTP stands for ‘Open Telecommunication Platform’ which dates to the telecommunication origins of Erlang and the set of applications to run on these telecoms systems. In this case, the application is build using OTP design principles, defining the structure of modules and processes.

The first aspect of OTP applications is the supervision tree which is a process structuring model split into supervisors and children or workers. To implement a process e.g. a supervisor, a user simply implements the callback functions for that process [15]. These supervised processes also help fulfill Elixir/Erlang’s ‘let it crash’ philosophy while guaranteeing service availability by restarting crashed processes back to their initial ‘good’ state.

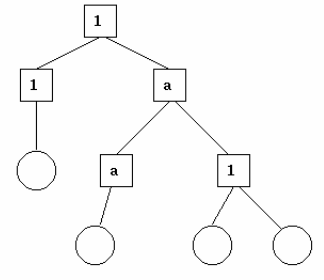


Figure 12: A supervision tree with squares as supervisors and children as circles

In this implementation, the entry point layer would be started as a child under a system process in the Beam VM supervision tree with other modules such as chord nodes, storage management and communication layers running as supervised child processes under the Chord supervisor. In the following sub-sections, we shall look at those layers in more detail while showing how they implement some of those callbacks from OTP modules.

## 5.2 Node Logic

### 5.2.1 OTP design implementation

Method implementations for the node pseudocode in the Chord paper were placed in their own module ‘ChordNode.ex’. This module inherited functionality from another behavior module called GenServer (Generic Server): an Elixir process that can be used to keep state, execute code either synchronously or asynchronously through a set of interface functions and support for being run under a supervision tree [16]. ChordNode implements functions such as *init* and *terminate* to add custom start and cleanup logic; *handle\_call* which is the most common interface implementation used in the module. The *handle\_call* callback synchronously handles messages passed to the GenServer process via the *GenServer.call* method invoked elsewhere in the application. Multiple *handle\_call* implementations can be created to handle different functions through Elixir’s function pattern matching against the input argument signature.

### 5.2.2 Node State

As mentioned previously, GenServer processes store state while running which is advantageous for the case of a Chord node. This state is passed as input to every function callback and is returned as part of the callback’s output. Node state in this implementation is formatted as a custom struct called *NodeState*. As shown in figure 13, a node will start with some default values in its state, but these values can be overridden during the *init* function in the ChordNode module

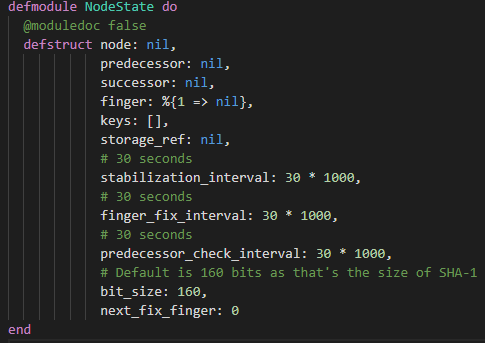


Figure 13: Node State Struct

### 5.2.3 Node Object Representation

As the program must return the corresponding IP address for a node ID, any node references must be <ID, Address> pairs. This is achieved in the program with a struct called CNode. It was not possible to name the struct ‘node’ as this is a special keyword within the Elixir system.

### 5.2.4 Synchronous Callback Example

Earlier on, we looked at the *handle\_call* method callback and how its synchronous run nature. Below is an example callback that implements the *create()* pseudocode from figure 6 of the Chord paper.

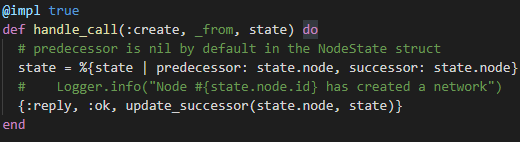


Figure 14: Example of synchronous GenServer callback

The first argument in this function is the request message sent during *GenServer.call* and used in the GenServer to function pattern match. In this case, the request message is simply the atom ‘:create’; atoms being constants whose values are their own name [17]. The second argument contains information about who sent the message and the third argument is the current node (GenServer) state. The output of *handle\_call* callbacks is a tuple with the format {:reply, any\_value, state}. The node state, whether mutated or not, must be passed as part of that output tuple.

### 5.2.5 Asynchronous Callback Example

There might be functions for which results do not have to be awaited on and GenServer provides a callback for these called *handle\_cast.* A good example is the *notify()* method which I decided would be an asynchronous call as there is nothing returned back to the caller.

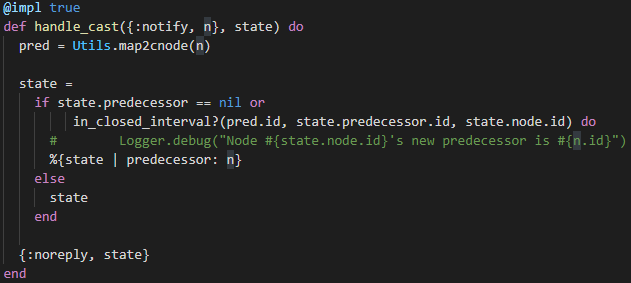


Figure 15: Example of asynchronous GenServer callback

Like *handle\_call,* these asynchronous are accessed in this case by calling *GenServer.cast({:notify, node})*. The output of this callback is a tuple with the :noreply keyword and the node state. Another asynchronous callback provided is *handle\_info* that is used to deal with periodic work such as Chord’s network stabilization functions. An example of this being *check\_predecessor()* which checks if a node’s predecessor has failed. The function below runs *check\_predecessor()* and schedules the next function run by passing the interval period stored in the state to Elixir’s *Process.send\_after()*

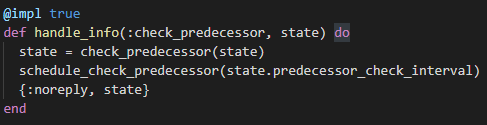


Figure 16: Example of a periodic process function

### 5.2.6 Finger Table

There were some challenges when implementing the finger table, the main one being that the Chord paper seems to portray the finger table as a list of node entries. As the finger table entries are populated and updated periodically, there would be sparse entries. It would be possible to create an empty list then fill this with time; however lists in Elixir are implemented under the hood as linked lists meaning list creation would be *O(m)* m being the SHA-1 bit size of 160. Additionally, accessing a record as we know for linked lists would be *O(n)* making lists inefficient. The solution to this was using Maps instead as these offered *O(1)* access and table entries could be added as need be minimizing overall table size. As shown above in Figure 13, the node state starts with a Map entry for the finger table containing an empty entry for the successor (the first entry in a finger table is the node’s successor).

### 5.2.7 Load balancing

Apart from just providing lookup functionality, the Chord protocol is also meant to alert the application it is running on of changes to the key set for which the node is responsible [9]. For the sake of prototyping, the node in this project implementation also manages keys. In the context of load balancing, when a node discovers a new successor during stabilization, it will transfer any keys in the range between its ID and the successor ID. This means as the network is constantly maintaining lookup correctness, it is also actively balancing the key load.

### 5.2.8 Voluntary Node Departure

Voluntary node departures in Chord are treated as failures as the protocol focuses on robustness in the event of failures. Two enhancements proposed by Stoica et al. to improve performance in this event are having a node transfer its keys to its successor and notifying both the successor and predecessor of its departure. Both of these have been integrated into Elixir’s ‘let it crash’ behavior by using GenServer’s *terminate()* callback; which is called if an exception is raised in the node process or the process supervisor sends an exit signal.

## 5.3 Communication Layer

### 5.3.1 gRPC vs. JSON-RPC

There are cases when a node must run remote function calls to another node such as during a key lookup or joining a network. The two technologies I had in mind for this layer were gRPC and JSON-RPC. gRPC would be advantageous due to its focus on high performance, support for multiple programming languages and efficient data transfer using protocol buffers (protobufs). However, the issue faced was the lack of a mature, stable Elixir gRPC library and for this reason, JSON-RPC was the better choice.

I chose a library called JSONRPC2 for its stability and ease of use. The library implements the transport layer in either HTTP or TCP and I decided to use HTTP because connections can be closed after procedure calls are done. This is advantageous because the number of active connections would scale with the network potentially taking a toll on a node’s resources. Functionality in this library is split into a client and a server with the client being what the node process uses to fire off remote procedure calls and the server being a separate process that listens for connections. The initial intention was to spawn a server process for each node process in the system however, because one of the underlying components of the JSONRPC2 library was using a unique process name, multiple instances of the HTTP server could not be spawned in the same supervision tree. Instead, a singleton HTTP server process would be used and would route requests to the appropriate node in the system.

### 5.3.2 JSON-RPC Server

As Elixir uses message passing or process communication, each process is assigned a unique process ID (PID). There is also the option of assigning a custom name to a process using the built-in *Process.register()* function. This is useful as node processes can be identified using their node IDs and, in this system,, the naming convention being “Node\_<node-id>”.

The server module contains request handlers for node function calls and routes them to the appropriate node; as one physical node can have multiple virtual nodes. An example of this process is the *find\_successor()* method shown below.

With the use of a helper function, the destination node ID is mapped to a process PID and a call is made to the destination node.

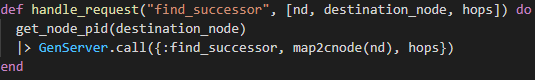


Figure 17: Example JSON-RPC request handler

### 5.3.3 JSON-RPC Client

The client part of the communication layer is implemented as a module containing RPC methods which node processes provide a destination CNode struct and relevant parameters. In the example below, if the module was named ‘RemoteNode’, then the remote call would be invoked by calling *RemoteNode.find\_successor(<destination>, node\_id)* and inside that method, the HTTP call is made to the remote node.

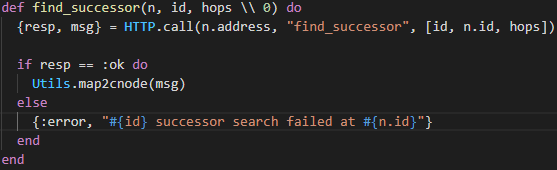


Figure 18: Example JSON-RPC method

### 5.3.4 Simulating Remote Function Calls

When testing the program, the Chord network is simulated with just Elixir processes and no remote calls are made outside Elixir. It made sense then to make an implementation of the communication layer for inter-process communication. Using *find\_successor()* as an example again, invoking this method, assuming invocation by calling *RemoteNode.find\_successor()*, the function in the transport simulation module would have the same logic as figure 17 of the JSON-RPC server.

## 5.4 Storage Layer

Implementation of record storage was done using Elixir’s internal record stores as it was quick and easy to integrate with the Elixir program. The two options for this were Mnesia: a real-time distributed database and Erlang Term Storage (ETS) which is an in-memory store for Elixir/Erlang objects. ETS was the preferred choice as it can be used on a per-process basis; processes representing a node and Mnesia being tailored for large distributed applications where processes are sharing data.

When a node process creates its ETS table, a reference to this table for invoking function calls is returned which the node process stores in its state struct. The storage module implements the normal get, delete, put methods alongside get and delete operations for a key range. These should not be confused for record range queries rather; they are used in node stabilization and departure functions when handing over keys to another node.

## 5.5 Library Entry Point

When another application wishes to use the Chord library, this is the layer responsible for initializing other layers; given application environment variables. Library variables can either be set at compile-time in a file located at *config/config.exs* or at run-time using the built in function *Application.get\_env(Application\_name, variable).*

The application can be run in either a normal mode or a simulation mode. The normal mode looking for environment variables like the address and port number to start the transport layer on and node ID. If no ID is provided, this will be generated by hashing the node’s address and port number. In the initial application iteration without virtual nodes, only one node is started under the Chord application supervisor by passing a child specification as shown below. The specification consists of a random child ID and the module with a start function, given a list of parameters, to start.

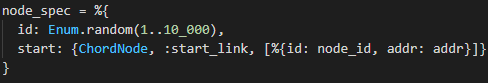


Figure 19: Supervisor child specification

Alongside the child specification above, a specification for the transport layer will be passed to the *Supervisor.start\_link()* function. Given the application name, children and supervision strategy, a supervision tree will be started for all the children. The supervision strategy tells the supervisor how to handle restarting crashed processes with the default being ‘one-for-one’ where if a child process crashes, only that process is restarted. Finally, if parameters to join a network are present in the application parameters, the *join()* function will be invoked.

Other functions in this layer are a *stop()* function which calls *Supervisor.stop()*; halting all child processes. And the *lookup()* function that applications utilizing the Chord library use to map a key to a node. One interesting challenge faced implementing the lookup function was getting the reference to the node in the system. The application’s *start()* method returned only a supervisor ID and kept no state so a way to keep track of the node reference had to be found. The first possible solution was using Elixir’s registry module: a key-value store mapping names to process IDs. What limited using this was the nature of GenServer processes that if they restart, the process will have a different PID rendering that registry entry inaccurate. An alternative to registries is agents which are an abstraction around state; to be accessed either by the same process at various points or by multiple processes. A custom Agent called StateAgent, with *get()* and *put()* functions, was created and gets passed alongside the node and communication layer to the supervisor.

To run the application in simulation mode, the variable *simulation: true* is added to either the config.exs file or *Application.put\_env()* at run-time (this function is used later in testing). The difference with normal mode is that only node processes are passed to the supervisor; the number of nodes being the *network\_size* environment variable. The JSON\_RPC server is not started as nodes will use the simulation communication module instead which uses built-in Elixir inter-process communication.

# Testing

# Future Work

# How To Use This Project

## 8.1 Pre-requisites

Running the project code requires Elixir v1.10 or later installed on the computer with installation instructions available at <https://elixir-lang.org/install.html>.

## 8.2 Downloading dependencies

It is important to download third-party library dependencies that project relies on. First, open a terminal window and ensure it is in the root directory of the project. The next step is to run the *mix deps.get* command which will download all necessary dependencies. Once complete, the application can now be run.

## 8.3 Setting environment variables

Before running the program, it is useful to set some variables like the address and port number for the communication layer or what mode to run the application in. Setting these can be done in the file under directory *config/config.exs* under the section *config :chord.*

## 8.4 Running in interactive shell

Elixir comes with an interactive shell to run commands or access documentation for built-in functions; accessible by typing *iex* into a terminal session. The documentation was very useful in implementing this project can be accessed by typing *h <function>* in the interactive shell. A shell session, with the project application and dependencies loaded into its runtime, can be started by opening a terminal window in the project root directory and typing *iex -S mix.*

### 8.4.1 Running in normal mode

Chord can be started by running *Chord.start()* and if you wish to join an existing network on start, include the existing node’s ID and address in *config.exs* before starting the iex shell. Finally, stopping Chord is done by running *Chord.stop().*

### 8.4.2 Running in simulation mode

Simulation mode is intended for testing and is run by setting *simulation: true* in *config.exs* before starting iex. Test functions will dynamically set application variables using their respective function arguments. //Test functions

# Professional Issues

# 10 Self-Assessment

**References**

[1] D. Reinsel, J. Gantz, and J. Rydning, “The Digitization of the World - From Edge to Core,” 2018. [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[2] Domo, “Data Never Sleeps 7.0,” 2019. https://www.domo.com/learn/data-never-sleeps-7 (accessed Jul. 20, 2020).

[3] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, “Making Gnutella-like P2P Systems Scalable,” *Comput. Commun. Rev.*, vol. 33, no. 4, pp. 407–418.

[4] M. Gowan, “Requiem for Napster,” *PC World*, 2002. https://www.pcworld.idg.com.au/article/22380/requiem\_napster/ (accessed Jul. 20, 2020).

[5] M. Richtel, “Napster Is Told to Remain Shut,” *The New York Times*, 2001. https://www.nytimes.com/2001/07/12/technology/ebusiness/napster-is-told-to-remain-shut.html.

[6] Wikipedia, “Gnutella,” *Wikipedia*. https://en.wikipedia.org/wiki/Gnutella (accessed Jul. 20, 2020).

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, “A scalable content-addressable network,” *Comput. Commun. Rev.*, vol. 31, no. 4, pp. 161–172, 2001, doi: 10.1145/964723.383072.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, 2001, doi: 10.1145/964723.383071.

[9] I. Stoica *et al.*, “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications,” 2001.

[10] P. Maymounkov and D. Mazières, “Kademlia: A peer-to-peer information system based on the XOR metric,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 2429, pp. 53–65, 2002.

[11] P. K. Gummadi, S. Saroiu, and S. D. Gribble, “A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems,” vol. 32, no. 1, pp. 82–82, doi: 10.1145/510726.510756.

[12] J. Aspnes and G. Shah, “Skip graphs,” *ACM Trans. Algorithms*, vol. 3, no. 4, pp. 37-es, doi: 10.1145/1290672.1290674.

[13] A. Ghahrai, “Software Development Methodologies,” 2016. https://devqa.io/software-development-methodologies/#:~:text=The incremental build model is,involves both development and maintenance.&text=This allows partial utilisation of product and avoids a long development time. (accessed Aug. 13, 2020).

[14] Elixir, “Application - Elixir v1.10.4.” https://hexdocs.pm/elixir/1.10.4/Application.html (accessed Aug. 14, 2020).

[15] Erlang, “OTP Design Principles - Erlang.” https://erlang.org/doc/design\_principles/des\_princ.html#applications (accessed Aug. 14, 2020).

[16] Elixir, “GenServer - Elixir v1.10.4.” https://hexdocs.pm/elixir/1.10.4/GenServer.html (accessed Aug. 14, 2020).

[17] Elixir, “Atom - Elixir v1.10.4.” https://hexdocs.pm/elixir/Atom.html (accessed Aug. 14, 2020).