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 Where other published and unpublished source materials have been used, these have been acknowledged.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](#_Toc46755211)

[2 Background Research 2](#_Toc46755212)

[2.1 Napster 2](#_Toc46755213)

[2.2 Gnutella 2](#_Toc46755214)

[2.3 Content Addressable Network (CAN) 3](#_Toc46755215)

[2.4 Chord 5](#_Toc46755216)

[2.5 Kademlia 7](#_Toc46755217)

[2.6 Skip Graph 8](#_Toc46755218)

[3 Section three 10](#_Toc46755219)

# Introduction

# 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.

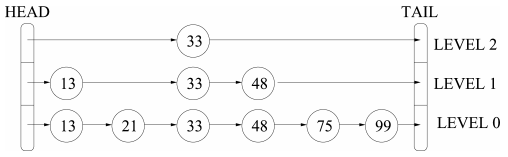


Figure 8: Example of a skip list

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*.

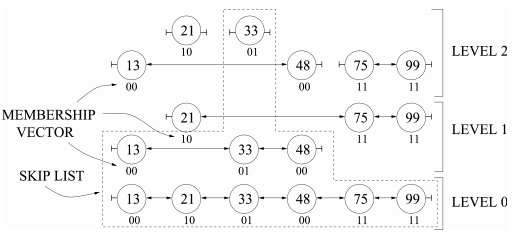


Figure 9: Example of a skip graph

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.

# Section three

**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.