A distributed key-value store using the Chord protocol

Daniel Casenove, Emil Dudev, Julius Hohnerlein, Serkan Keskin, Shivam Kapoor
{d.casenove, e.dudev, j.m.hohnerlein, rs.keskin, s.kapoor}@student.vu.nl

Supervisor: Animesh K. Trivedi
animesh.trivedi@vu.nl

December 17, 2019

Abstract

Key-value stores are the backbone of many distributed systems. However, traditional single node key-value stores do not scale in this setting. This report details the requirements, design and evaluation of a distributed key-value store based on the Chord protocol. Chord is a distributed lookup protocol that enables scalable key lookups in a dynamic environment through decentralisation. Additionally, the report also introduces optimisations to the stabilisation part of said protocol. Multiple experiments performed on a cluster show that these optimisation are effective and the developed system is scalable and performant.

1 Introduction

A key-value store is a data storage paradigm that, in its simplest form, allows storing arbitrary data identified by a unique key. Data can be stored, modified and queried using the key. Another way of seeing a key-value store is as a type of NoSQL database with no structure. Popular examples include Redis [11] and Memcached [7]. Key-value stores are popular and are being used in many distributed systems and ecosystems. The performance, scalability and reliability of these systems is therefore impacted by the same attributes of the key-value stores. Achieving several of these requirements while maintaining a practically useful notion of consistency, however, is a classical problem in distributed systems.

This report delineates the design and implementation of a peer-to-peer key-value store based on the Chord protocol [2]. The Chord protocol provides an efficient way to partition a keyspace onto a mutable set of nodes in a peer-to-peer network while supporting one operation: it maps any given key onto a node. Since achieving exactly this mapping from keys to nodes is a crucial part of any distributed key-value store, some of the requirements are inherited from or limited by Chord. However, as the Chord paper does not provide a complete solution for implementing a key-value store, there is still some design space to be explored.

The rest of the paper is structured as follows. The requirements for the project are laid out in section 3. An exploration of the design space, including optimisations and additions to the Chord protocol, are described in section 4, while the necessary prerequisites of the protocol are provided in section 2. The evaluation method used on the proposed system is then explained in section 5 with a series of benchmarks ran on the university's DAS5 cluster. Results of said experiments are then discussed in section 6, and a final conclusion on the work done is drawn in section 7. Additionally, a timetable can be found in the appendix.

2 Background

This section describes the operation of the Chord protocol, which is used in our implementation for node location. The protocol provides a mechanism to map a key onto a node in the system and defines how new nodes can join the system, what happens if nodes fail and planned departure of nodes. According to the Chord paper, it achieves these functions while fulfilling the following requirements:

Load balancing The keyspace is partitioned evenly (with high probability).

Decentralisation All nodes are homogeneous and therefore are equally important.

Scalability The cost of looking up a node is logarithmic in the number of nodes in the network.

Availability Node failure and planned changes in the system are handled without compromising availability.

Flexible naming There are no requirements to adhere to for naming a node.

Technically, the functionality provided by Chord could be described as a peer-to-peer, *consistent* hash table. Consistent hashing is a hashing technique that ensures only $\frac{|K|}{n}$ keys have to be moved to a new slot/node, when a single node/slot is added or removed, if K is the keyspace and n is the number of slots/nodes [1]. This ensures that nodes can join and leave the network without having to redistribute the entire keyspace.

Chord achieves consistent hashing using the following technique: each key and node is hashed to an m-bit identifier using SHA-1. These identifiers form, both figuratively and mathematically, an ordered ring, which is also called the identifier circle or Chord ring. The node responsible for a key is defined as the first node which has the same or following identifier as the key - also referred to as the successor node for the key. When a node leaves the system, the keys it is responsible for are reassigned to its successor. Mirroring this behaviour, when a new node joins the system, it becomes responsible for some of the keys of its successor. Figure 1 visualises a Chord ring and keyspace distribution with 64 identifiers.

Looking up a key in the network only requires each node to know its own name and the name of its successor. Knowing these two names, each node can individually find the partition of the keyspace its successor is responsible for. A key can then be looked up by forwarding the request to the successor node until the responsible node is found.

However, this lookup method requires up to n (the amount of nodes in the system) messages to be sent, making it non-ideal from a scalability perspective. Faster lookups are achieved using a per-node $finger$-table: The $finger$ table entry $finger[k]$ contains the successor of each identifier $(p + 2^k) \bmod 2^m$, where $0 \leq k < m$, and p is the position of the node in the ring. Using this table, instead of forwarding each request to its direct successor, a node can forward the request to the nearest known predecessor of the (identifier of the) key. Each time a message is forwarded in this way, the keyspace to search in is cut in half in the worst case. Therefore at most $\log_2(n)$ messages are needed to locate a node, making the method much more scalable, while keeping the local state each node has to maintain small ($\mathcal{O}(m)$).

Chord implements a stabilization protocol that is used by the nodes to keep a list of their successor pointers up to date. Similarly, the finger table entries are periodically being verified and updated. In case of a node joining the network, the protocol aims to guarantee at least the reachability of pre-existing nodes in

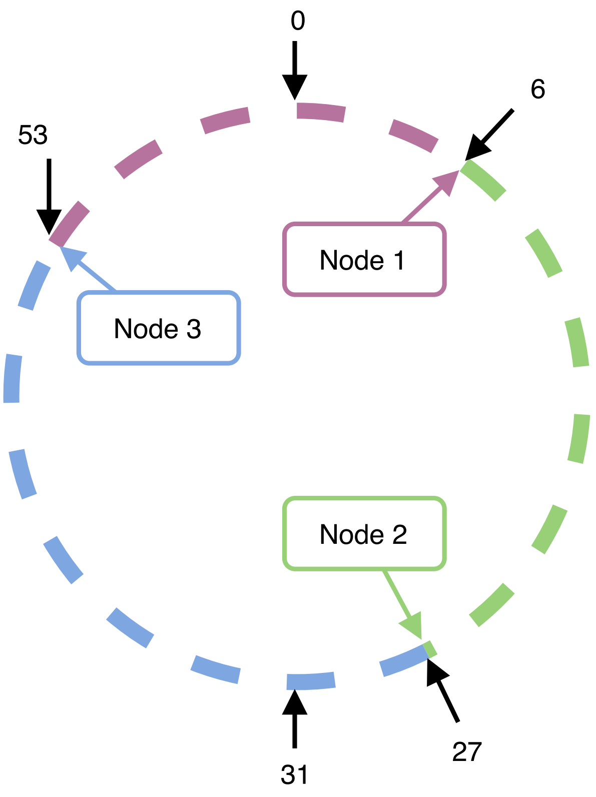

Figure 1: A system of 3 nodes and 64 identifiers ($m = 6$). The keyspace distribution among the nodes is visualised. For example, node 2 is responsible for keys 7 to 27.

the network with a possibility of failing lookups on keys that have yet to be migrated to the node entering the ring. Failure is handled by the network using the successor list: nodes that realise a fail in their successor replace it with the next alive entry in their list. Therefore, the system is always able to heal itself as long as not all successor list entries fail at once. Combining this mechanism with the stabilization protocol results in the eventual correction of wrong finger table and successor list entries. Having a correct finger table enables the protocol to perform fast lookups. On top of being a fundamental aspect in failure recovery, the successor list could be used by higher level applications to implement data replication.

3 Requirements

As the distributed key-value store is based on the Chord protocol, some of the functional and non-functional requirements are inherited from or limited by Chord. In this section, the functional and non-functional requirements of the proposed implementation are laid out.

Scalability The distributed key-value store should be able to scale while maintaining the Chord protocol properties: the amount of hops a lookup request takes should scale logarithmically with the amount of nodes.

Availability The underlying Chord implementation of the system should be available.

Consistency The implementation should achieve strong consistency in the face of nodes joining and leaving voluntarily.

Interoperability The system should be interoperable with the basic functionalities of a memory-caching system, supporting operations of storage, modification and retrieval of data.

Correct lookups Correctness of lookups is required in order to have a functioning Chord ring implementation.

Efficient lookups Following the Chord protocol, in a ring with n nodes, lookups should be resolved via $\log_2 n$ messages.

In-memory storage While data persistence is not required, the system should be at least able to manage key and data storage in-memory.

The fact that the Chord protocol is not partition-tolerant has to be noted. In case of network splits, several independent rings may form. To circumvent this, a list of several successors is kept, but it does not provide a 100% guarantee. Taking into consideration the CAP theorem [3], this effectively means that the Chord protocol can, in certain cases, achieve strong consistency and availability.

4 System Design

In this section a description of the architecture of the proposed solution is provided, together with the technologies utilised to implement it. In addition, an elaborate explanation of the optimisations made is also provided.

4.1 Golang

The programming language chosen to implement the Chord protocol and the distributed key-value storage on top of it is Golang [5], a syntactically similar language to C which also provides garbage collection and memory safety. On top of this, Go's concurrency primitives allow developers to easily and efficiently write high performance software in multicore and networked scenarios.

4.2 Memcached

Memcached[7] is an open source memory object caching system and its protocol is used in this distributed key-value store implementation as a way to expose the storage to the end user. In particular, clients can request *set*, *get*, *delete* operations to be performed on the system to a memcached compliant listening server. The choice of implementing a memcached compliant system was of particular help in the testing of the distributed key-value store using existing benchmarking tools as explained in section 5.

4.3 gRPC

gRPC[6] is an open source Remote Procedure Call (RPC) framework developed by Google and is used, together with Protocol Buffers

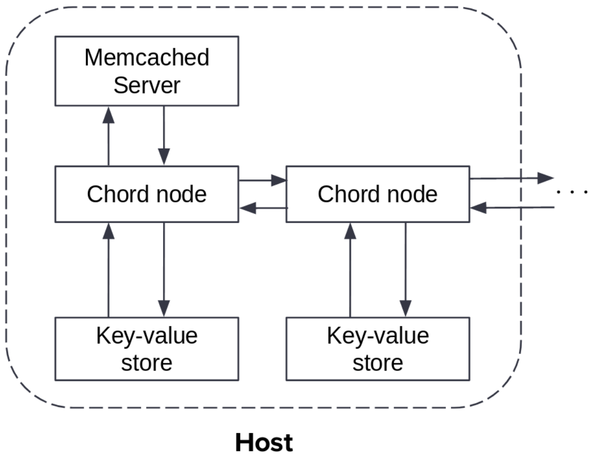

Figure 2: Architectural view of a single host

(protobuf)[10], in our implementation to handle all communication between nodes. The choice of using RPCs as the communication method is derived from the initial Chord paper as this request-response protocol fits the abstractions used in the paper. One of the problems faced using this solution was the possibility of bidirectional communication between peers of the ring. gRPC is limited to an unidirectional request-response model. This effectively means that if one node makes a request to another one, the response will be returned over the same channel. However, if the second node at some point in time wants to make a request to the first node, another channel will be set up, resulting in a total of 4 ports being occupied.

4.4 Architecture

To achieve an implementation that is easy to maintain and improve upon, a clear distinction is drawn between the major elements of the distributed system as depicted in figure 2. For example persistence could be added to the key-value layer without interfering with the ring layer. As previously explained, a memcached backend is responsible for interfacing with clients. Based on the key received from the user, the query is then redirected to the responsible node using the Chord protocol. Once the request has reached the designated node, it is fulfilled by accessing a local map and the result is then propagated to the client.

4.5 Implementation

The architecture just introduced follows the ring topology of the Chord protocol with a major implementation difference: the introduction of so called virtual nodes. This notation is used to identify different ring nodes spawned by the same process, effectively creating a relationship of one to many between processes and virtual Chord nodes. As will be explained in section 5 this implementation choice is mainly used for effectively deploying a considerable amount of ring nodes in the testing environment.

On top of this, the approach used lends itself to future work, like multiplexing connections, and generally provides better performance utilising multiple goroutines in contrast to a single process per node.

4.5.1 Concurrency

To provide correctness and good performance during communication between nodes, each peer implements a fine grained locking mechanism. In particular, the finger tables, predecessor and successor lists are locked independently using readers-writer locks in order to maintain correctness and serve independent requests simultaneously. Additionally, this choice allows the nodes to handle simultaneous RPCs on the same resources if the operation performed is a read. For write operations simultaneous RPCs can only be handled on different resources. The architectural separation between the Chord and storage infrastructures makes it so that operations on the ring, for example maintenance of the topology, do not block storage operations therefore not slowing them down unnecessarily.

4.5.2 Local storage

As every node of the ring is responsible for a part of the overall key space, a local storage mechanism is needed in every peer. The implementation utilised is a simple Go built-in associative data type called Map which, as the name suggests, simply uniquely maps data to keys provided. Due to the possibility of a node receiving multiple requests for the same key, a simple implementation like the one just proposed would fail due to the readers-writers problem. While Golang recently added support for a concurrent map structure, due to its performance in what would be the use case[13], concurrency is handled in this implementation utilising a simple readers-writer locking mechanism on the typical Map data type.

4.5.3 Data Migration

When a node joins the ring, it becomes responsible for a set of keys, which, before the join, belonged to its successor. To achieve consistency, the keys (and the data associated with them) need to be moved to the new node. The data migration algorithm must ensure consistency (as much as possible) in case multiple nodes join at the same time, and in case any of the participating nodes fails during the migration.

The proposed approach for data migration is as follows. When a node joins the ring, it switches to a 'proxying' state in which it forwards all upper-layer (non-lookup, non-Chord related) operations to its successor. This is needed as the new node does not have the required state to successfully execute key-value store operations. On the other hand, when the successor gets notified that it has a new predecessor, it transitions to a data migration state and starts to do two things. First, it sends its current state to the newly joined node, its predecessor. Second, it keeps a log of all high-level operations that are executed with the keys that are being migrated. When all the state has been migrated, the node starts to transfer the log. When the log has been exhausted, the successor notifies the newly joined node that everything has been transferred. The newly joined node, at this point, stops proxying requests to its successor and this marks the end of the data migration process.

The above algorithm mandates two clarifications. The first one being that ending the data migration process does not happen atomically on both nodes. The successor node is the one that knows when the process has to end, but its predecessor is the one that proxies requests to it. Without careful consideration, a race condition may occur. To circumvent this issue, a monotonically increasing counter for the proxied operations, and a two-phase commit step when bringing the migration process to a halt, can be used. The successor notifies the newly joined node that everything has been transferred up to operation T, and, if the last proxied operation had an index greater than T, the commit is unsuccessful and the data migration process continues.

The second clarification that has to be made is that nodes can join at any time. This effectively means that a data migration chain can be formed, and that a node can be both in the proxying state and in the data migration state. The chain is then contracted sequentially in reverse order (based on the position of the nodes in the ring).

Lastly, this algorithm can be extended to accommodate for data replication. Having data replication in place, the implementation can achieve consistency in case of node failures and be fault-tolerant. It should also be noted that the ring itself is already fault-tolerant.

4.5.4 Stabilisation improvements

A considerable amount of the implementation effort of this solution has been spent in what is referred to, in the Chord paper, as stabilisation. This operation is routinely executed by Chord nodes to maintain a correct ring topology in case of churning. Additionally, it is used to achieve fast lookups by maintaining a correct finger table. The implemented improvements can be broken down into three major parts which concern the maintenance of the finger table and are described next.

These optimisations make use of the finding that the finger table will often contain duplicate nodes. This can be seen from the following thought experiment: An identifier has a size of m bits, hence the whole ring consists of 2^m identifiers. Without loss of generality, the number of nodes in the ring n shall be represented as 2^k. The *expected* inter-node distance is $\frac{2^m}{2^k} = 2^{m-k}$. Then any given node n_i will be at distance of 2^{m-k} to its successor. Since the finger table at slot l shall be occupied by the successor of 2^l (relative to the node position), the first $m - k - 1$ finger table entries shall be occupied by the node's immediate successor. In concrete terms: If SHA-1 is used as a hash function, $m = 160$ and given a ring size of $n = 512$ ($k = 9$), the first $160 - 9 - 1 = 150$ finger table entries will be occupied by the same

```
learnNode($n$):
    for $k$ in range of $fingerTable$:
        if $k == 0$:
            continue
        if $n \in (fingerTable[k].start,$
                $fingerTable[k].node)$:
            $fingerTable[k].node = n$
```

Figure 3: Pseudcode for node learning. 'fingerTable[k].start' references the minimal position for the k-th finger table entry. 'fingerTable[k].node' references the node stored at the k-th entry in the finger table.

node. This observation shall be denoted as `O1`.

While the nodes are unlikely to be perfectly evenly distributed, they should approach an even distribution with an increasing ring size. Uneven distributions will result in the number of duplicate entries in the finger tables of the nodes to vary. However, for the most part, duplicates will be present, as long as the ring size is order of magnitudes less than 2^m.

Intelligent Node Learning A simple function which ensures that a given node takes up all of the expected entries in the finger table has been implemented, as shown in figure 3. Whether a node n is supposed to occupy the k-th finger table slot can be determined with certainty for every k and n by checking whether n precedes the previous occupant of the table and succeeds the minimal position of that slot.

The `learnNode` method is called in various locations in the code where a node is known to be alive. For example, during stabilisation, according to the Chord protocol, each node n will contact its successor s to let s know that n thinks n is the predecessor of s. If this RPC succeeds, n knows that s is alive. These kinds of opportunities are available for any RPC call. Using the information that a node is alive, a check is made to make sure that it has been entered into the finger table. While nodes that are not proven to be alive could be learned, this optimisation was applied carefully in order to not learn stale entries and achieve a definitive improvement.

Note that, while no rules are broken regarding the finger table, even if they were, the finger table is only responsible for *fast* lookups, but not *correct* lookups. In fact, none of the proposed optimisations touch any data that is critical for correctness.

Dynamic Stabilisation Interval While the previous optimisation is passive in the sense that it makes better use of the *existing* information, this optimisation actively seeks out new information. Concretely, the ability to schedule stabilisation for the finger table and successor list immediately, instead of waiting for the next interval, was added. For example, if it is realised that a node in the network died, it is removed from the successor list and finger table. While doing that, a new stabilisation for each of them is scheduled to run immediately if the node that died was part of it. In case of the finger table a request for immediate stabilisation also carries a specific index to stabilise, which allows guiding the stabilisation routine to where entries need to be fixed.

Note that this optimisation again, does not change any aspects regarding correctness, as only the interval for stabilisation is being affected. The stabilisation routines themselves are unchanged and the stabilisation interval is not relevant for correctness.

Skipping Finger table Updates The previous optimisations unlock a third one: the ability to skip slots during finger table stabilisation. To see why certain slots in the finger table can be skipped, consider the following: apart from the finger table, the immediate successor, the successor list and the predecessor are periodically checked and fixed due to stabilisation. It is easy to see, that there is overlap between these nodes and the nodes in the finger table. For example, the first slot in the finger table is by definition occupied by the immediate successor.

Since during these stabilisation procedures, the finger table is always being updated with the newly learned information due to the optimisations, slots that are occupied by these nodes can therefore be skipped when updating the finger table. In particular, just by not iterating over the slots occupied by the immediate successor, which based on `O1` are $m - k - 1$, the required number of iterations to traverse the

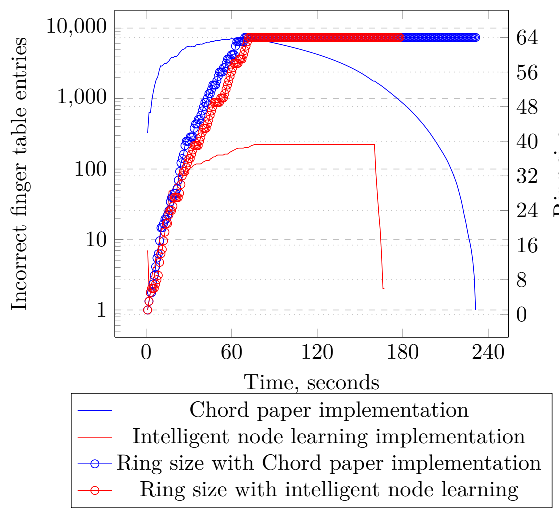

Figure 4: Finger table entry stabilisation
Over time comparison of incorrect table entries per implementation. Number of nodes = 64

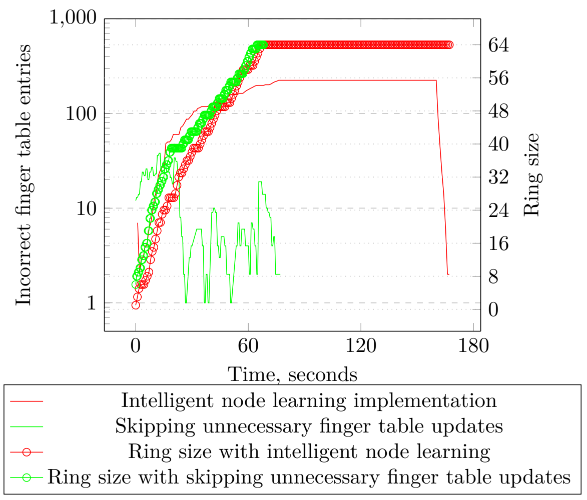

Figure 5: Finger table entry stabilisation time reduction
Over time comparison of incorrect table entries per implementation. Number of nodes = 64

finger table can be reduced to just $m - (m - k - 1) = k + 1 = \mathcal{O}(log(n))$.

This last optimisation is particularly interesting, in that it reduces the time to full stabilisation significantly. The recommendation from the Chord paper is to update finger table entries sequentially, which means it will take m stabilisation runs in the worst case to achieve a correct finger table in a steady network state. After this optimisation, a correct finger table is achieved after just $\mathcal{O}(log(n))$ stabilisation runs, as per the finding described earlier (`O1`).

5 Results

While Chord was initially designed for global internet usage, the testing has been conducted on the VU's DAS5 cluster [4], a distributed homogeneous supercomputer. The reasoning behind this decision is the ease of deployment of the implementation compared to solutions like AWS or Azure. Each node of the cluster provides two 2.4GHz 8-core CPUs coupled with 64GB of RAM and 128TB of storage.

Due to the high reservation time for large amounts of cluster nodes, the implementation has been tested by assigning more than a single virtual Chord node per cluster node. In addition, having more virtual nodes helps in statistically achieving a more even key distribution. On top of many Chord virtual nodes, a DAS5 node also exposes a Memcached server listening for requests to be dispatched to the ring.

The deployment on the DAS5 is automated using bash scripts which run the implementation on multiple nodes. Benchmarking is carried out, using the memcache-compliance of the implementation, thanks to tools like Memtier[9] and Memslap[8] automated by scripts with different parameters launched on a separate set of cluster nodes. In addition to these benchmarking tools, a program was written to test the correctness of the Chord protocol implementation: upon learning a node it traverses the whole ring using successor nodes and calculates the amount of incorrect finger table entries.

The first experiments conducted with this utility are represented in figures 4 and 5. They aim to demonstrate the results of the stabilisation optimisations. The former experiment compares the basic Chord protocol to the proposed modified algorithm, having enabled intelligent node learning. The latter experiment demonstrates the effectiveness of skipping unnecessary finger table updates.

Depicted is a bootstrapping scenario of the Chord ring with a total of 64 nodes and, shown over time, is the number of incorrect finger table entries present in the network. The first optimisation keeps the overall number of incorrect finger table entries an order of magnitude less than the non-optimised version of the pro-

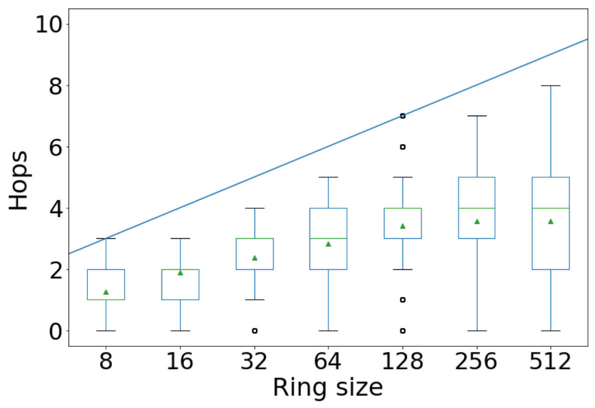

Figure 6: Hop count per ring size
The boxes go from the Q1 to Q3, with the triangles representing the mean and the green line the median (Q2). The top whisker is the nearest value below Q3 + 1.5*(Q3-Q1) while the bottom whisker is the nearest value above Q1 - 1.5*(Q3-Q1), dots are outliers. The blue line represents $\log_2 n$.

tocol. In addition to that, the second optimisation achieves a faster stabilisation time. In fact, the optimised version stabilises almost immediately after the last node joins, while the non-optimised version takes almost 4 times as long, as it has to stabilise the finger table one entry at a time. While this scenario is unlikely to happen often in normal operation, as massive changes to the ring are unlikely, it was chosen to clearly show the effect of the optimisation. In other situations the effect may be less pronounced however as described in section 4 the stabilisation optimisation should still achieve better or equivalent performance than the original implementation.

To validate the claim of achieving efficient lookups in $\log_2 n$ hops, experiments with varying ring sizes were run and are summarized in figure 6. For each run of the experiment 10000 get and set operations were executed.

The graph shows the growth of $\log_2 n$, with n being the number of nodes of the ring, and the percentile of hops for communication. The implementation adheres to the claims made by the Chord paper in terms of communication necessary for lookups in the ring.

The last experiment aims to provide some insight on how performant the implementation is. A cluster of Chord virtual nodes and an equal number of benchmarking threads are run. Each benchmarking test manages 50 clients executing simultaneous requests to the key-value store

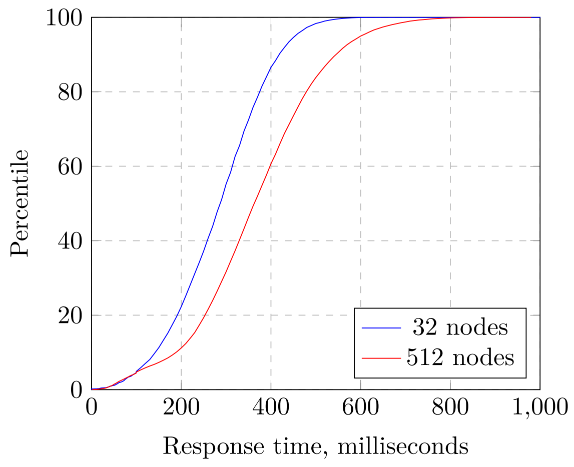

Figure 7: Key-value store responsiveness, CDF graph

running on top of the Chord ring. The data for each query is 1024 bytes. This test is conducted against two rings, one with a size of 32 nodes and one with a size of 512 nodes. The results are shown in figure 7. It is shown that when the size of the ring is increased 16 times, a latency penalty of about 100 milliseconds is achieved, a fraction of the original response time. This latency hike is also supported by the increase in the amount of required hops, as shown in figure 6.

6 Discussion

In this section an in-depth explanation of the results reported in section 5 is presented. On top of this, challenges that arose during the implementation of the solution are explained, together with trade-offs of the designed system.

In-memory Storage The data is kept in memory and the system, as is, does not provide persistent storage through disk writes. However, the system has been designed to be extended with persistent storage.

Interoperability Basic compliance with text-based memcached is provided and the system has been tested using the tools presented in section 5 with the supported get, set, delete operations.

Reliability Reliability is obtained in the Chord protocol implementation, as demonstrated by the tests done and the correctness of the stabilisation function. Even though a successor list is used to correct the ring in case of failures, it is not yet utilised to achieve data replication. For these reasons the proposed system does not achieve fault tolerance for the key-value storage. Furthermore, data migration is yet to be implemented.

Lookups and Scalability The remaining requirements related to Chord, correctness of lookups and scalability, are also satisfied by the implementation and closely follow the results obtained by the paper. On top of performing correct lookups, the efficiency requirement of $\mathcal{O}(\log n)$ communication in the given system is satisfied, as seen in figure 6. The scalability of the proposed implementation has been tested on the DAS5 supercomputer and, as the number of nodes present in the ring grows linearly, the amount of hops grows by a logarithmic factor.

Virtualisation of Nodes The usage virtualised nodes, as presented in section 4, does not come without its drawbacks and, even if mainly implemented for testing purposes, shaped a lot of the implementation with its advantages and disadvantages. A major drawback is the possibility of a node having its successor list filled with virtual nodes from the same, but different physical node. In case of failure, the system can not ensure that the ring will not split.

Optimisations The proposed optimisations were shown to be effective, which is unsurprising given the simplistic recommendations in the Chord paper. Perhaps more surprisingly, the gap between the optimised and non-optimised versions is significant. Given that these optimisations require just a few additions to the implementation, it seems that there is still considerable potential for further optimisation.

Technologies used Golang proved to be an easy to learn programming language, as implementing non-trivial code was done in a matter of weeks with only one team member being proficient with this language. On top of this, coupling it with gRPC was straightforward due to abstractions fitting well. The static, and cross system, compilation made it also very easy to test the implementation on DAS5.

While Go is a concurrent programming language by nature, it falls short in helping prevent concurrency related bugs such as data races, something that other languages, such as Rust [12], can statically prevent. Achieving concurrency, as claimed by the developers, is indeed easy but there are no mechanisms in play to help make said implementation correct. In the same fashion, the simplistic type system, while being easy to learn, leads to repetitive code which not only is a chore to write, but also a potential source of bugs.

Difficulties in Chord The implementation of the Chord protocol proved to be difficult not only due to the nature of distributed systems, but also due to ambiguous notation and unspecified methods. In particular, ranges of nodes are specified while taking into account the circular nature of the network: notations like (p, p), with p being a node position, include the whole ring space (except p). This kind of usage is not explicitly specified in the paper and could lead to a faulty implementation of the protocol. In the same way, critical methods, such as how to fix the successor list, are also not specified.

7 Conclusion

This report presents a distributed key-value store implementation based on the Chord protocol. The architecture is made to be easily maintainable and amenable to future improvements and includes a backend memcached server for interoperability. Using gRPC for the communication implementation made it easy to translate the Chord's paper ideas into code and Golang helped achieve concurrency without significantly complicating the code base. The Chord protocol has been improved upon by implementing an optimised version of the stabilisation method. This result has been validated on the DAS5 environment. Fast lookups in $log(n)$ hops have also been achieved, in line with the Chord paper claims. While consistency claims have been made, the implementation presented still lacks data migration thus making the testing of this requirement not possible.

7.1 Ongoing work

To fulfill all the requirements previously laid out, data migration is a focus of ongoing work. In fact, since nodes failing is already supported on the ring layer, achieving data migration would also lead the way for data replication, achieving fault-tolerance. On top of this, a more in depth approach to optimising the stabilisation mechanism could be explored. Communication optimisations are also currently being investigated: in particular, inter-node communication of physical nodes could be achieved on a single socket and then dispatched to the final virtual node without RPC usage. The source code of the project can be found at https://github.com/edudev/chord/.

References

[1] Karger et al. "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web". In: *STOC '97*. 1997.

[2] Stoica et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *SIGCOMM '01*. 2001.

[3] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Vol. 7. 2000.

[4] *DAS-5 Clusters*. URL: https://www.cs.vu.nl/das5/clusters.shtml (visited on 12/16/2019).

[5] *Golang Doc*. URL: https://golang.org/doc (visited on 12/16/2019).

[6] *gRPC*. URL: https://grpc.io (visited on 12/16/2019).

[7] *Memcached*. URL: https://memcached.org/ (visited on 12/16/2019).

[8] *Memslap*. URL: http://docs.libmemcached.org/bin/memslap.html (visited on 12/16/2019).

[9] *Memtier*. URL: https://github.com/RedisLabs/memtier_benchmark (visited on 12/16/2019).

[10] *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers (visited on 12/16/2019).

[11] *Redis*. URL: https://redis.io/ (visited on 12/16/2019).

[12] *Rust*. URL: https://www.rust-lang.org/ (visited on 12/16/2019).

[13] *The new kid in town — Go's sync.Map*. URL: https://medium.com/@deckarep/the-new-kid-in-town-gos-sync-map-de24a6bf7c2c (visited on 12/16/2019).

Appendix

A Timetable

Type of activity	Hours spent
think-time	22
dev-time	47.4
xp-time	16
analysis-time	6
write-time	35
wasted-time	7.3
total-time	133.7

Table 1: Aggregate hours spent per category as defined in the lab description.