# Replacing an External Configuration Service with the Chord Peer-to-Peer Protocol in a Distributed Database

Krisela Skenderi
TUM
Munich, Germany
krisela.skenderi@tum.de

David Silva
TUM
Munich, Germany
UP
Porto, Portugal
ge58dez@tum.de
up201705373@fe.up.pt

Lukas Bernwald
TUM
Munich, Germany
lukas.bernwald@tum.de

## ABSTRACT

With an increased scale of data orientated applications, also in relation to Big Data, there is a need for scalable database systems, that can deliver the required performance. As traditional databases can become bottlenecks in an application, there is a need for distributed databases. Thus, in previous milestones of this course, we implemented a distributed database system. This system, however, introduced a Single Point of Failure, namely the External Configuration Service. We propose the use of the Chord protocol as an alternative to building and maintaining the distributed database. The results of our experiments indicate that Chord performs similarly in terms of latency and throughput compared to the centralized system while overcoming the Single Point of Failure problem, making it more preferable.

## KEYWORDS

Distributed Systems, Cloud, Databases, Chord

## 1 INTRODUCTION

In previous milestones, the External Configuration Service (ECS) represented a Single Point of Failure (SPoF) for the implemented distributed database. It was responsible for coordinating the servers, maintaining metadata, failure detection and recovery, and coordinating the replication of key-value pairs among the servers in the network. In this project, the ECS is removed, and the servers are extended with Chord to be able to support the same system functionalities as the previous system. Chord is a scalable, decentralized Peer-to-Peer (P2P) Distributed Hash Table (DHT) protocol. It enables the efficient lookup of servers responsible for specific keys, even in the face of concurrent node arrivals and departures. This report provides details about our implementation of Chord, the new system architecture, and a comparison with the original system.

The goal is to evaluate and compare the two systems in terms of latency and throughput. This evaluation assess the trade-off between resiliency by eliminating the SPoF and performance caused by the increased complexity of the system. Specifically, we find that Chord is more difficult to work with and there might be more failures during network changes, however the overall performance of Chord is similar to the ECS system while eliminating the SPoF. Furthermore, we determine that replication in the system is also a trade-off between resiliency and performance.

## 2 PROBLEM STATEMENT

In the previous system, all servers maintain information about all other servers and their respective key-range responsibilities in the server ring. The ECS is responsible for maintaining and updating the metadata of all servers whenever a server joins or leaves the ring. Delegating the task of achieving consensus of key responsibility to a single point can also be advantage due to its simplicity. However, in the event of an ECS failure, the copies of metadata stored in each server would become stale, and no new servers would be able to join the ring and share the load of the requests. Replication, failure detection and recovery would not be possible, leading to scenarios where some key ranges would become inaccessible. This dependence makes the ECS an SPoF of the system.

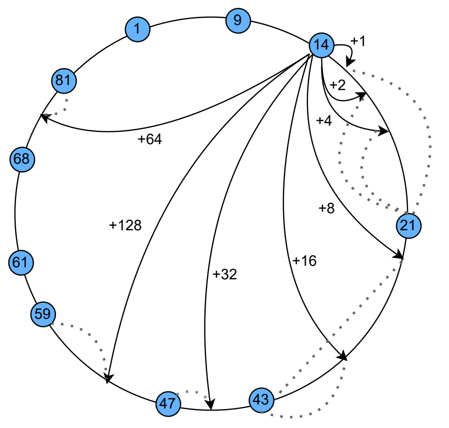
Removing this SPoF is the motivation behind using a decentralized P2P service such as Chord. It avoids the need for a centralized entity to maintain the metadata by having the servers communicate directly with each other and exchange information to update their state and maintain the Consistent Hash Ring structure.

A different approach for overcoming the SPoF problem would be to use a Leader Election (LE) algorithm, such as the Chang & Roberts Algorithm [1], to elect one server in the system to act as an ECS. The disadvantage of that approach is the overhead of electing the new leader and the system potentially not being functional during this election. An additional advantage of Chord over the LE scenario is that the size of the metadata stored in each server is independent of the number of servers in the network, and a server joining or leaving only affects a few other nodes. By contrast, in the ECS and LE systems, the size of the metadata that needs to be stored and updated at each node after each network change grows with each new server joining.

## 3 APPROACH

Chord was chosen because its underlying requirements and structure, such as the ring topology, fit with the architecture of the previous system. By using Chord, the same functionalities as the previous milestones can be provided while keeping the internal changes in the system transparent to the user. Other requirements such as eventual consistency and load balancing are also preserved. Our implementation of the Chord protocol closely follows the Chord specification [6].

We adapt the previous system by extending the server with new classes to implement the Chord protocol and reusing or adapting code from the ECS implementation. There are minimal changes in the server-server and server-client messaging libraries to reflect



FINGER TABLE OF N14

| | |
|---|---|
| N14 + 1 | N21 |
| N14 + 2 | N21 |
| N14 + 4 | N21 |
| N14 + 8 | N43 |
| N14 + 16 | N43 |
| N14 + 32 | N47 |
| N14 + 64 | N81 |
| N14 + 128 | N59 |

**Figure 1: Chord Ring and Finger Table [6]**

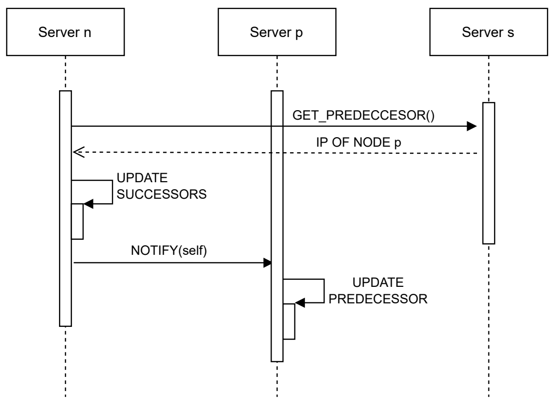


**Figure 2: Sequence Diagram for Stabilization Operation when a node $p$ is joining the ring between nodes $n$ and $s$**

the new message exchange between Chord servers. The server was extended with messages specific to the Chord protocol, and the client messaging was updated to conform to the new server-client protocol. The details of the implementation are explained in section 4. Following the implementation, the two systems were evaluated in terms of latency and throughput. We ran the same test cases with a varying number of parameters for each system and compare the results in detail in section 6.

## 4 ARCHITECTURE

This section contains a description of the implemented architecture, describing the main Chord procedures, as well as extensions that allow the new system to perform the same functionalities as before, e.g. handoff and replication.

### 4.1 Overview of Chord Architecture

Chord is a protocol for a P2P DHT [6]. Similar to the old ECS system, the new architecture is based on the concept of Consistent Hashing. Servers are represented as nodes organized in an identifier circle and keys, which are in the same *id* space as the servers, are assigned to the successor node in the ring.

The main difference is the format of the metadata stored in each node. A Chord Server stores information about only a few other nodes by maintaining a finger table of $m$ entries, where $m$ is the number of bits of the key/node identifier. Each entry $i$ of the finger table contains the address of successor$(n + 2^{i-1}) \mod 2^m$. In this way, the query for a key can easily be handed off to the largest node in the finger table preceding this key, making it possible to complete the lookup in $O(\log(n))$. The ring order is maintained by the predecessor and successor pointers of each Chord Server. An example of a Chord ring and finger table is provided in fig. 1.

### 4.2 Server Join

In contrast to the old system in which each new server had to register through ECS, the new Chord Server joins the ring by using another node to *bootstrap*. The new server $(A)$, queries the bootstrap server $(B)$ for $A$'s successor $(C)$. Server $A$ then notifies $C$, informing it that $A$ might be its new predecessor. Both server $A$ and $C$ update successors and predecessors respectively. Eventually, $C$'s previous predecessor $(D)$ will notice that $A$ is $C$'s new predecessor, and will notify $A$ that $D$ might be its predecessor. By having both the successor and predecessor pointers, $A$ is now in the ring.

### 4.3 Maintenance of the Ring Structure

In the old system, the ECS is responsible for updating the metadata each time a server joins or leaves the ring. Every new server has to join the ring through the ECS, and the ECS also performs the heartbeat operation with each server to detect server failures. Afterwards, it sends a message to each server in the ring to update their copies of the metadata. Instead, Chord maintains correct metadata and hash ring structure in the form of the finger-table, predecessor and successors. This metadata is mostly focused on the local neighborhood of the server in the consistent hash ring. The locality of this metadata might lead to increased scalability of the system. Updates of the metadata are performed through two periodic procedures: Stabilization and Fix-Fingers.

*4.3.1 Stabilization Procedure.* Each Chord Server periodically contacts its immediate successor for the purpose of maintaining its successor pointers. The node asks its successor to return the successor's predecessor, and then compares the information to determine whether it needs to update its successor list. The sequence diagram in fig. 2 illustrates the case when a new node $p$ has joined the ring between nodes $n$ and $s$. This is an extremely important procedure, because as long as the successor list of each node is correct, key lookups are guaranteed to be fast and efficient and finger tables can be updated correctly.

*4.3.2 Fix Fingers Procedure.* Each Chord Server periodically checks whether a random entry in its finger table is still correct, by initiating a lookup for the key corresponding to that entry. If the ring topology has changed, the entry is updated.

### 4.4 Failure Detection and Recovery

In addition to the mechanisms mentioned in section 4.3, some operations are also employed to try to maintain a viable ring configuration. First, each server periodically sends a heartbeat to its predecessor. If the predecessor doesn't respond, the server removes it. Eventually, the true predecessor will assign itself, through the stabilization procedure outlined in section 4.3.1. Additionally, during the stabilization, a server sends some messages to its successor. If, for some reason, the successor does not respond, the server shifts its successor list, thus making its successor's successor, its new successor.

## 4.5 Replication

In the previous implementation, replication was maintained by comparison of key ranges upon a metadata update by the ECS. This process is now achieved by listening to successor changes. In practice, this is achieved by extending the Chord implementation to have *listeners* which are notified when predecessor or successors change[1]. Note that despite this change, the replication process is still the same as before: When receiving a *PUT* or *DELETE* operation, if replication is enabled, the request is sent to the replicas respectively successors in the form of a *PUT_SERVER* or *DELETE_SERVER* message.

Replication is started or ended by inspecting the size of the successor list, which must always be larger or equal to the desired replication factor. When the number of servers is equal to the replication factor, each server replicates their keys in the range between its predecessor and itself to its successors. Similarly, when the successor list size falls below the replication factor, each server deletes its replicated keys. When the server $A$ notices that one of its successors changed, replicated keys are deleted from old successors, if they are still alive, and new successors receive copies of said keys.

## 4.6 Handoff

Whereas previously the ECS was notified when a server joined or left and proceeded to order the execution of a handoff to the correct servers, now, this is done independently. When server $A$'s predecessor changes, for example to server $B$ when it joins, server $A$ sends $B$ all keys in range $(\text{pred}_A, B]$, where $\text{pred}_A$ is $A$'s previous predecessor. The keys are sent using the specific message *PUT_SERVER_OWNER*, that signals to $B$ that it should replicate those keys. When a server leaves, in a controlled manner, it starts a handoff procedure, during which it *write-locks* itself and sends its keys to its successor.

## 5 REALIZATION

Following the theoretical approach of the system, we employed different techniques and concepts to realize it into a working system.

## 5.1 Coding Best Practices

During the development of the previous milestones, as well as the project, great care was taken in creating a system that conformed to programming practices that are generally regarded as "good". To this extent, the system was developed using a layered architecture, where responsibilities between the different components were well separated. This strategy allowed for minimal changes on the transition to the Chord protocol. There was also a focus on following the *Dependency Inversion Principle* [4], which reduced the overall coupling of the different components of the implemented solution. This was an asset that not only facilitated testing, but also allowed for a fluent system evolution during the different milestones and the project.

Furthermore, since its inception, unit tests were created in order to enable a certain degree of trust in the correctness of the implemented functionalities. In addition, we made an effort to provide documentation for most public methods, which not only aided the development process but will also allow for future reference of the implemented solution.

## 5.2 Persistent Storage

To further explore database related technologies and data-structures, a B-Tree [2] was implemented to serve as a persistent storage structure for the servers. A B-Tree is a balanced search tree that's designed to be used on storage devices. This type of tree allows $O(\log n)$ *get*, *set* and *delete* operations while providing a structure that enables scalability suited for a computer's typical memory structure. This is achieved by only reading blocks of keys respectively nodes of the tree as they are needed to perform the operations. This way, the tree can grow freely without being limited, since the whole structure is never fully loaded into main memory. The size of each B-Tree block can be configured by setting the tree's minimum degree. This parameter is usually optimized given properties of the storage where the tree will be held, e.g. related to the memory's page size. However, this optimization is out of the scope of this project.

## 6 EVALUATION

To evaluate the system's performance, we created a *Java* application[2], that aims at simulating real-life scenarios. The tool can *create a configured amount of servers* that are instantiated with defined parameters e.g. cache strategy, cache size and B-Tree node minimum-degree. The servers may be created at the start of the run, or late through the use of *timed events*. These events can also be used to shut down servers after a given time delay. If the experiment is not using Chord, the ECS server is started first and server's receive its address as parameter. However, in the case of a Chord experiment, each server receives the address of another server as bootstrap.

The application also *starts a configured number of clients* that connect to a random server from the list of created ones and that execute *get*, *put* and *delete* operations from the *Enron email dataset* [3]. Clients continuously execute these operations in a cycle of *put-get-delete*. This is done in order to try to balance the number of operations of each request type. However, *deletes* only occur after 30 % of the client's emails have been sent, in order to allow for the growth of the amount of data in the servers. The clients may also be created using *timed events*. Currently, we measure the sum of *get*/*put*/*delete* operations and respective fails in one second time-steps. We also track the time needed for each operation, i.e. the latency of that operation.

[1] See `KVChordListener.java`

[2] See `src/main/java/de/tum/i13/simulator`

### 6.1 Overview of the Experiments

We performed two experiments, each containing multiple runs of the system. From our previous experiments for milestone 3, we concluded that 500 is a good cache size and thus went with it for these experiments. Furthermore, we are using the Least Frequently Used (LFU) cache eviction strategy. The minimum degree of the B-Tree is 200 and the initial delay after the initial servers and clients are created until the experiment actually starts is 10 s. First the additional clients are initiated and after that the additional servers are started. Both experiments are run with both, the ECS and Chord. The experiments were performed on a computer equipped with an Intel Core i5-6500 and 8 GB of RAM, using Ubuntu 20.0.1 and running on Java version 17.0.1.

*6.1.1 Behavior.* In the first experiment, we evaluate the behavior of the system when new servers are joining. Data is replicated to 1 other server from each coordinator server. Before the experiment starts, 0 clients and 3 servers are started. During the experiment, 20 additional clients, for a total of 20 clients, and 3 additional servers, for a total of 6 servers, are started. The delay between starting each server is 120 s and the delay between starting each client is 5 s. Between starting the additional clients and starting the additional servers, the experiment lets the system run for 120 s. After the additional servers are started, the experiment concludes after 120 s.

*6.1.2 Replication.* In the second experiment, we evaluate the performance of the system, when there is a stable number of clients and servers, but the number of servers data is replicated to changes. We test with a replication factor of 0, 1, and 2. Before the experiment starts, 20 clients and 6 servers are started. During the experiment, 0 additional clients and 0 additional servers are started. Thus, the delays between starting additional components are not employed and irrelevant. After the initial delay, the experiment concludes after an additional 180 s, for a total runtime of the experiment of 190 s.

### 6.2 Results

The results of the experiments can be seen in figs. 3 to 5 and tables 1 and 2. The successful operations in the figs. are displayed with a 10 s rolling window average, while the unsuccessful operations are shown per second as is. Note that the figs. differ in scaling across the y-axis and, thus, might not be directly comparable.

*6.2.1 Behavior.* Figure 3 shows the results of the behavior experiment for the ECS and Chord. From the graph we can see that at the start the servers perform quite well, because they do not contain any data yet and still have enough free resources. After a while, the servers resources are used up under the increased load. Eventually the behavior seems to stabilize around the 100 s mark. Furthermore, we can see that once a new server joins, because handoff and replication have to be performed, the successful operations decrease and the errors spike, probably because the servers are write-locked due to data handoff between servers. Although in general, the ECS approach and Chord seem to be performing similarly on average, the error spikes seem to be higher using Chord. This might be due to its more distributed nature and the difficulty of handoffs.

When the third additional server joins, the successful operations of Chord suddenly increase rapidly. At first, this seemed to be an anomaly, but the same behavior was confirmed in a second run of the experiment. It is possible that this behavior is shown, because at this point, some clients that might be waiting for answers from the servers are receiving their responses in rapid succession. However, since this is unexpected behavior, it should be investigated further in future work.

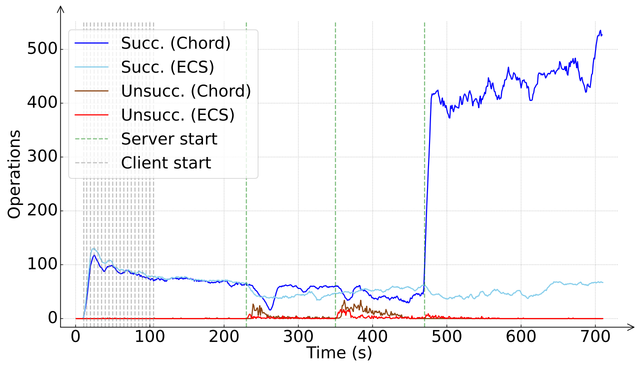


**Figure 3: Throughput, measured by successful and unsuccessful operations over time, using the ECS system and Chord during the behavior experiment**

**Table 1: Average request latency in ms using the ECS and Chord during the behavior experiment**

| System | Get | Put | Delete |
|---|---|---|---|
| ECS | 272.17 | 329.94 | 359.77 |
| Chord | 30.68 | 266.35 | 268.83 |

Table 1 shows the average request latency in the system. According to the data, *get* requests are the fastest in both systems. This is probably due to no writing to persistent storage being necessary for this request type. The latency of the ECS system and Chord are located in the same proximity around the 300 ms mark in general, however Chord seems to be slightly faster on average with a 70 ms to 100 ms shorter response time for *put* and *delete* operations. This can be seen as a trend, but it could also vary due to other external factors. There is also an outlier: For the *get* operations, the latency of Chord with 30 ms is significantly lower than for the ECS system with 270 ms. This behavior might be correlated with the spike in throughput seen in fig. 3, but it has to be further investigated to be finally evaluated. Moreover, the latencies overall seem rather high for locally run experiments, but that might be due to the consistent stress the clients put on the servers and the lack of powerful hardware for the experiments.

*6.2.2 Replication.* Figures 4 and 5 show the replication experiment using ECS and Chord. We can see that the performance of the ECS system and Chord are similar for the same replication factor. It can also be observed, that the error rate is almost 0 since no servers are joining or leaving. The main point that the figs. show, however, is that using replication hurts the performance of the servers. Going from 0 to 1 replicating server almost halves the performance of the system. Although going from 1 to 2 replicating servers the

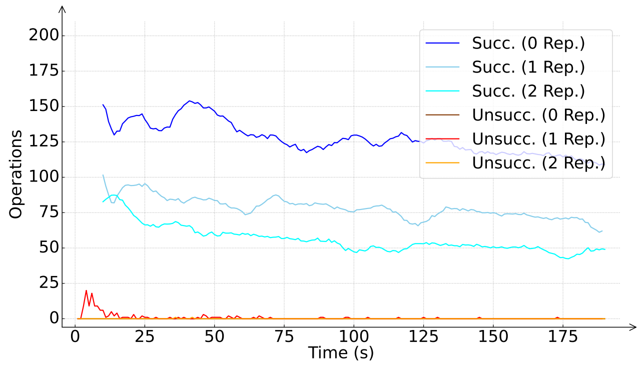


**Figure 4: Throughput, measured by successful and unsuccessful operations over time, using Chord with 0, 1, and 2 replication servers during the replication experiment**

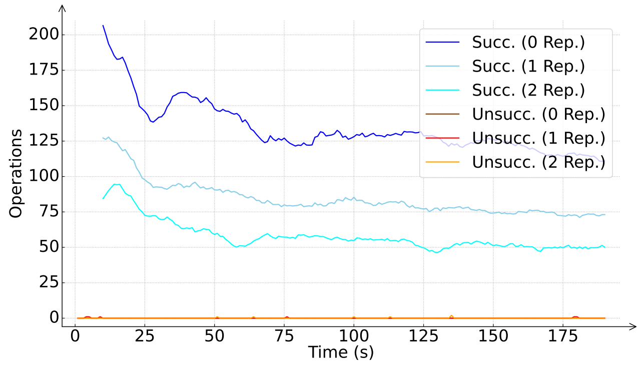


**Figure 5: Throughput, measured by successful and unsuccessful operations over time, using the ECS with 0, 1, and 2 replication servers during the replication experiment**

difference is not as noticeable, there still is a discernible decrease in throughput. Since replication is employed to ensure that no data is lost in the event of a server unexpectedly leaving, while deciding if and how replication is to be performed, a tradeoff between system performance and system resiliency has to be made.

Table 2 shows the average request latency in the system for the replication experiment. Here the latency of Chord seems to be slightly worse than that of the ECS system with an approximately 11 ms average difference overall. Similarly to figs. 4 and 5 we can see that with an increased replication factor, the latency of the system also increases. Going from 0 to 1 replicating server, the latency of the *get* and *delete* operations increases by approximately 110 ms on average. The latency of *get* requests only increase by an average of 60 ms. Going from 1 to 2 replicating servers, the latency overall increased by approximately 100 ms on average.

It seems as though the latency of the *get* requests is impacted the least, going from no replication to 1 replicating server. When increasing the replication factor from an already replicating system, the impact on the latency is similar for all request types. In further work, it has to be checked why this difference in increases of the latency between the requests types is shown by the system. In general, we can see that replication seems to be a tradeoff between system performance in regard to the throughput and request latency and system resiliency in regard to potential data loss caused by unexpectedly shut down servers.

**Table 2: Average request latency in ms using the ECS and Chord in relation to the replication factor during the replication experiment**

| | ECS | | | Chord | | |
|---|---|---|---|---|---|---|
| Rep. | Get | Put | Delete | Get | Put | Delete |
| 0 | 133.27 | 159.38 | 167.87 | 143.76 | 166.16 | 174.82 |
| 1 | 205.94 | 260.50 | 275.16 | 217.38 | 285.86 | 281.17 |
| 2 | 326.16 | 351.80 | 379.51 | 332.55 | 369.23 | 388.64 |

## 7 LIMITATIONS

Although we see great potential in using Chord and our experiments suggest that it is a valid approach for eliminating the ECS as an SPoF, Chord also comes with its own problems and certain factors should be taken in consideration when examining the experiment results.

### 7.1 Network Latency and Data Distribution

A major drawback of Chord is potentially high latency due to network locality and round trip times for server communication. It is worth mentioning that while our tests were run on a single machine, in practice it is possible for servers in the network to be far away from each other, e.g. in different continents altogether. Proximity to a node on the identifier ring does not reflect geographic proximity, resulting in delayed communication and skewed results in our experiments. However, one can argue that, as a system scales, the centralized ECS approach is also not suitable for geographically distributed servers. Furthermore, as the number of servers increases, the amount of information about the ring that each server has to hold and that the ECS has to keep consistent will also increase. By using Chord, the information held by each server is kept the same, independently of the number of server's that exist in the full ring.

### 7.2 Eventual Consistency

Another limitation from the current implementation is that, by not having the central orchestration controlling the handoff procedure, both in the event of a node joining and leaving, there are time windows, in which the system might be inconsistent, i.e. there would be nodes that are responsible for keys which they have yet to receive from the old responsible servers. This happens because, with Chord, the change in topology instantly alters the node's responsibility range, whereas before, this range only changed upon instruction from the ECS, after all the keys have been successfully transferred.

### 7.3 Complexity of Lookup Operation

In theory, the number of nodes that must be contacted to find a successor, i.e. to perform a key lookup, in a $n$-node Chord network is $O(\log(n))$, i.e. logarithmic in the number of servers. As rows in the finger table of the Chord Server increase, we make larger steps around the identifier circle, with each subsequent hop at least halving the distance to the destination. Intuitively, this search procedure is similar to Binary Search, implying a $O(\log(n))$ bound for the lookup of a key. In comparison, the complexity of the "perfect knowledge" model of the ECS is $O(1)$, i.e. constant, since it stores information about the responsibility range of each server. We can already see that even though lookup in Chord is guaranteed to be fast and efficient, for a sufficiently large number of servers $n$ the lookup latency can be rather high. In this case, however, the lookup operation using the ECS would also encompass a large amount of data, since it would contain information about all servers.

### 7.4 Hardware Limitations

The size of the experiments isn't adapted to the available hardware. In a real world scenario, it would be expected that each server would be running on a separate machine. By running several instances of servers in the same hardware, we are not getting a true display of the system's performance, both at the network and processing levels. This also has the effect of diminishing returns of using multi-threading to increase throughput, because it multiplies the number of total threads existent on the system, i.e. the number of servers times the number of threads of each server. These threads may cause an overall performance hit due to the number of context switching that needs to be performed by the operating system. These hardware limitations, however, also are the reason why distribution of data needs to happen and is beneficial to the overall system and why distributed databases are needed in the first place.

### 7.5 Non-deterministic Behavior

Although not directly related to Chord, but to distributed and parallel systems in general, is the presence of non-deterministic behavior. The experiments that were run on these systems were, similarly to real life workload, nondeterministic by design. Furthermore, the performance of the system is nondeterministic in that sense, that it depends on external factors. Those factors may include the incidental computational load of the computer used for testing during the experiments, communication network load in general, and even factors, such as whether logging is enabled or not. All these factors lead to the results of the experiments sometimes varying not insignificantly. Thus, although on multiple runs of the experiments, there were trends that were mentioned in this paper, it is difficult to give actual definite conclusion just using the experiments. I.e. further work has to be performed.

## 8 CONCLUSION

Following the evaluation of the system, we conclude with a short summary and give an outlook of potential future work.

### 8.1 Summary

In conclusion, while Chord is more complex than the ECS system, the advantages mostly outweigh the disadvantages. Chord's higher complexity stems from its need for increased network communication necessary for data consistency. Furthermore, the data in Chord is also only eventually consistent. However, the performance of Chord is similar and sometimes even better compared to the ECS system. Additionally, the main advantage is not having a SPoF and potentially better scalability due only local knowledge of the system necessary on one server. Thus, we can cautiously recommend Chord over the ECS for managing servers in a distributed cloud database.

### 8.2 Future Work

Although, as explained, we eliminated the SPoF, there are still some areas that the application can be improved in and where further work is needed.

*8.2.1 Additional Evaluation.* The first point is to evaluate the system more rigorously and in scenarios that are closer to real world scenarios. Here, also the scalability of the system in regard to the number of servers should be evaluated. Furthermore, the unusual behavior that Chord showed in the behavior experiment shown in fig. 3 and table 1 should be investigated.

*8.2.2 Data Serialization.* At the moment, the data sent over the network is sent in a clear text format. While this makes testing the application certainly easier by enabling the developer to send simple commands to the application by hand, these messages have a higher data footprint than serialized messages. Thus, to improve network performance, the data sent over the network should be serialized.

Even though serialization is possible using standard Java features, to improve interoperability with potential other software components of the application that might be written in a different programming language, a mechanism such as Protocol Buffers (ProtoBuf)[3] should be used. While the messages using ProtoBuf might be larger in terms of memory than other serialized formats [5], a major advantage is its interoperability, mainly being able to generate programming language specific classes for type safe data serialization. Handing over the network message formatting to a framework and being able to access the data in the network messages via defined interfaces, also makes network communication less error-prone than trying to extract information out from string messages that might evolve over time. With, ProtoBuf it is also possible to extend the message format, while still being able to process messages formatted in an old way.

*8.2.3 Reduced Network Communication.* The implementation of some additional messages could improve the performance of the system. However, in order to reduce the complexity of the proposed solution, and to conform to the project's time constraints, these were left for future work. Namely, some operations could have reduced network footprints by bulk messaging. When performing a handoff, currently each key is sent individually, where an optimized version could send several keys at the same time. Similarly, when deleting keys from old replicas, a bulk delete could be sent instead of an individual delete for each key.

## ACRONYMS

| | |
|---|---|
| **DHT** | Distributed Hash Table. |

[3]See https://developers.google.com/protocol-buffers/docs/javatutorial

| | |
|---|---|
| **ECS** | External Configuration Service. |
| **LE** | Leader Election. |
| **LFU** | Least Frequently Used. |
| **P2P** | Peer-to-Peer. |
| **ProtoBuf** | Protocol Buffers. |
| **SPoF** | Single Point of Failure. |

## REFERENCES

[1] Ernest Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (may 1979), 281–283. https://doi.org/10.1145/359104.359108

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[3] Bryan Klimt and Yiming Yang. 2004. Introducing the Enron Corpus. In *CEAS 2004 - First Conference on Email and Anti-Spam, July 30-31, 2004, Mountain View, California, USA.* http://www.ceas.cc/papers-2004/168.pdf

[4] Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, USA.

[5] Srđan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. 2016. Performance evaluation of using Protocol Buffers in the Internet of Things communication. In *2016 International Conference on Smart Systems and Technologies (SST).* IEEE, 261–265.

[6] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.