INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Design and Implementation of a Distributed Block Storage on RIOT

Robin Lösch

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Design and Implementation of a Distributed Block Storage on RIOT

Robin Lösch

Supervision:	Prof. Dr. Dieter Kranzlmüller
Advisors:	Dr. Nils gentschen Felde Jan Schmidt
Date:	February, 5th 2019

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 5. Februar 2019

.....
(Unterschrift des Kandidaten)

Abstract

IoT and WSN networks gained further relevance in recent years. A common problem in such networks is sharing and storing of data independent of node failures. This thesis examines the requirements on a distributed storage network with a special focus on embedded devices with low computational power. A design for such a storage network on top of the Chord P2P overlay network is proposed and implemented accordingly. As a result of the design, a fully distributed block storage driver is built on top of the RIOT operating system with support for dynamic growth and shrinking of the network and for heterogeneous storage sizes contribution by the members of the network. The requirements are checked against defined acceptance criteria and performance evaluations are done to show the behavior of the implemented modules. The experiments show good results in terms of the implemented distributed hash table and provide an overview of the introduced communication overhead by the implemented features.

Contents

1 Introduction

This chapter provides an introduction, as well as background information about the motivation and the objectives of this thesis. A practical scenario for the objective of this thesis and a problem statement is defined to set the scope for this thesis. Finally, the methodology used in the remainder of this thesis is specified and explained.

1.1 Motivation

In recent years, Internet of Things (IoT) and Smart Home devices have gained a lot more relevance. The reasons for this are besides the decreasing hardware costs, a constant rising in computational power and wide support for radio transmission technologies. Smart home and IoT devices are often deployed in a network like fashion. One example for these networks are Wireless Sensor Networks (WSNs) where multiple devices are collecting sensor information and communicate over a radio network. Another example are Smart Homes, where different participants of the network are responsible for collecting data and controlling different functions in a building.

A limitation in such networks is besides the computational power, the limited transfer rate and communication capabilities with the internet. These limitations raise a necessity for a preprocessing of collected data between the participants of such networks. Another problem is the availability of the data in the network. WSNs are often deployed in an environment with limited controlling possibilities and with non-redundant hardware and power supply, thus making them prone to errors or complete failures of single participants. Without a redundant data storage between the participants of the network, this can result in a loss of all the information stored on a failing node.

These two problems let arise the question of how it is possible to share the data in between the nodes and thus making it possible to pre-aggregate the data and save the data in a redundant fashion. A distributed view of the data also allows us to use an optimal utilization of the storage capacity in the network. In the last years, research projects had focused on this topic with the aim to develop distributed storage systems. Research contributing to these problems includes distributed file system like CFS[DKK+01], Ivy[MMGC02] or IgorFS[HAEF08], cloud-based management systems[AAH+13, AH14] as well as local storage engines.

A common limitation with cloud-based storage is the need for a constant or at least periodic available uplink into the internet as well as an additional storage environment with an internet connection. New trends like „Fog Computing“ addressing the problems of cloud computing for IoT devices by providing computational resources at the edge of a network. Data gets stored and processed at the so-called „fog node“, which does not need a constant

uplink to the internet. However, as already mentioned as a downside for cloud computing, fog computing relies on a cloud-based solution as a final destination for the stored data. Other approaches like MEC or „Cloudlets“ can also act as a standalone service on the edge of a network and thus needs no cloud service provider to work, but instead relies on a resource-rich machine in the local network [MNY+18].

Distributed file systems, in contrast, can act on a local network perspective. There are different approaches for the design and implementation of distributed file systems depending on the target environment. The well-known Network File System (NFS), for example, uses a straight forward approach by exporting a file system from a server to multiple clients [PJS+95]. More recent approaches like Ceph focusing on the scalability of such solutions by decoupling metadata access and direct I/O as well as distributing the workload among a cluster of nodes [WBM+06]. While these two examples are classical client-server architectures there are also a few designs focusing on Peer-to-Peer (P2P) distributed file systems. Common examples are Freenet, CFS, PAST, BitTorrent and Ivy [HAY+05]. A downside of these examples is the fact that they are designed for large-scale data sharing and distribution amongst large and resource-rich systems. Another problem is the limitation to the file system specific behavior and a well-defined use case, which makes them infeasible for some scenarios. In addition, these solutions are mainly designed for the use of file sharing between multiple untrusted participants over the internet and not as a shared storage engine in a distributed network of trusted computers.

Another alternative to the already mentioned cloud-based solutions or distributed file systems is a distributed storage engine, which acts as a block storage device to save data in the network. Such a system could act as a generic storage device to the operating system just like any other hard disk or flash drive and thus making a cooperative data storage independent from a specific file system implementation or the need for a cloud-based solution. This could be even more interesting in respect to the increasing amount of projects and companies that are focusing on the development of operating systems, which are tailored to the use-cases of IoT devices. The main goal of these projects is to provide a framework, which allows the user to write applications without needing to build hardware specific code and thus write portable software that is not limited to a specific environment. Examples are open source projects like Contiki, Mbed OS and RIOT as well as commercial products like „Windows 10 for IOT“ and „WindRiver VxWorks“. Since these projects are normally implementing some sort of file systems and APIs to develop storage drivers, a block storage solution that provides a POSIX compatible interface to store data could be easily integrated into existing projects. From a users point of view, it would be convenient to get a block storage driver which just acts as a local flash or disk storage and could be used either directly or via an additional file system layer.

1.2 Scenario

The ambition behind this thesis is to provide a way to share data in between a network of computational nodes with a special focus on embedded devices with low computational power such as IoT devices and WSNs. There is already an existing range of applications for such networks. Healthcare, home automation, agriculture and industrial automation are a few examples where networks of embedded computers are already in use [MPV11].

In respect to the wide range of deployment scenarios there are multiple requirements on such networks, depending on the needed computational power, used storage capacity, power consumptions and the availability of a networking infrastructure. As already stated such networks also often lack the availability of redundancy features such as a redundant power supply and are often only powered by a single battery with a limited lifetime. Another common property of such networks is the fact that the nodes are organized in a P2P network where nodes are equally privileged and the functioning of the network is independent of the availability of a central authority.

The design of the storage network as described in this thesis should focus on such a scenario where multiple non-uniform and equal powerful devices are deployed in a wide range of environments and need to store data. This could be due to the need to aggregate and share data between nodes or the need to store data in a reliable way in case of device failures. A special focus on the scenario lies in the P2P like organization of such networks and the low computational resources of a single member.

1.3 Problem Statement

Distributed applications can roughly be distinguished into classical client-server architectures and P2P architectures. In a client-server architecture participants act either as clients who request a service from the participants in the server role or as a server that provides the service for the clients. P2P architectures in contrasts do not provide exclusive roles to the participants in the network. Every node in the network can act as an equal eligible member and is free to provide a service or to use services provided by other participants. IoT networks often consist of multiple nodes with equal responsibility. In respect to this „server-less“ property of IoT networks, where every member can act as a service provider, it is convenient to implement a distributed storage as a P2P architecture [SW05, p. 10].

Another reason for this choice is that servers can be also a victim of node failures and thus the availability of the service is then determined by the number of participants in the server role. In IoT networks node failures can be quite common since the devices are often deployed in a non-controlled environment and consist of non-redundant parts. Also, the communication between the nodes is often based on a radio protocol such as IEEE 802.11 and thus is not reliable. A solution to the defined problem has to be resilient against joining, and even more importantly, against simultaneous failures of multiple nodes. In P2P networks the occurrence of simultaneous joining and leaving node is referred to as churn. Since churn can happen anytime, a mechanism needs to be provided which is capable of handling failing nodes without data loss. Furthermore, a joining or failing node can result in a different size of the overall storage. The design needs to cover a way to support the resulting dynamic growing and shrinking of the storage.

Other properties of IoT Networks are the limited energy and space requirements and thus lower computational power and possible data rates compared to traditional computers. The design should focus on a P2P block-level storage with the possibility of high churn and thus take care of failing nodes as well as the dynamic resizing of the storage device. To ensure the availability of resources to the main application running on a device, a special focus on a low overhead in terms of computation and communication needs to be taken.

The solution should act as a block storage driver on top of the RIOT operating system for IoT devices. The design of the solution should be portable and not limited to a specific environment like RIOT. This makes the selection of RIOT as a target system arbitrary and is motivated by the easy to use modular design of RIOT.

The resulting implementation should act as a generic storage driver and behave like any other block orientated device. With a minimal interface to read and write blocks of data, it should be possible to run an arbitrary file system on top of the device. The participants of the system are located in a local network without the need for an uplink into the internet. In a distributed system participants can fail or even join the system anytime or on demand. For this reason, the design should face a dynamic replication scheme that ensures data availability up to a certain level.

1.4 Methodology

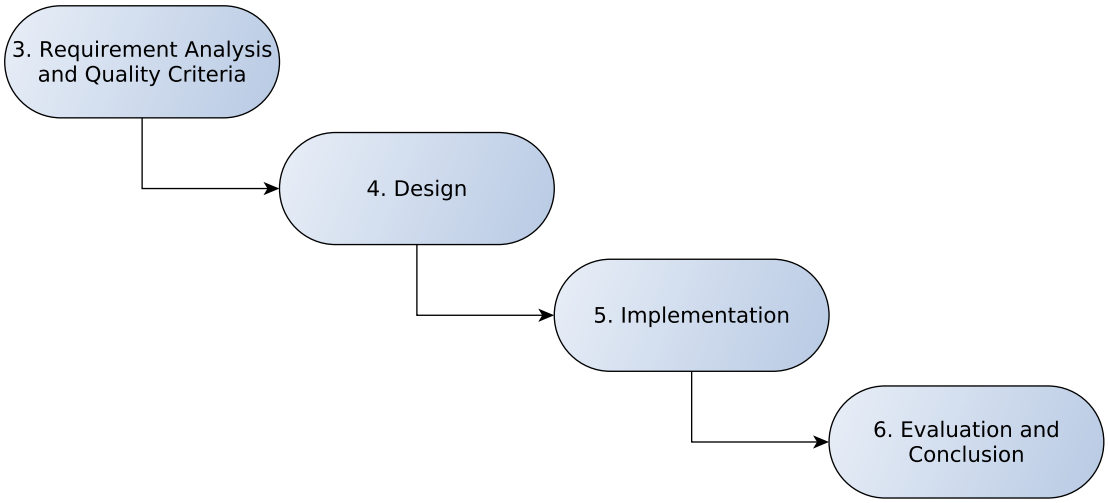


Figure 1.1: Methodology

As shown in Figure 1.1 the methodology follows a linear approach. In Chapter 3, a requirement analysis is performed and the requirements are divided into functional and non-functional requirements. Alongside with the requirements, a set of acceptance criteria for later verification needs to be defined. Afterward, a design proposal for the final solution needs to be defined in Chapter 4. The design should address the requirements defined in the previous chapter and should include a solution to the found problems and pitfalls. The next step involves an implementation of the proposed design with an in-depth explanation in Chapter 5. This also includes the verification of the acceptance criteria that were defined in Chapter 3. The last step is to perform performance evaluations and experiments addressing the correctness of the implementation in Chapter 6. The measurements need to be evaluated and summarized in a conclusion in conjunction with the remaining tasks and findings.

2 Background

In this chapter, an introduction to work related to this thesis is provided as well as the theoretical background of some used technologies.

2.1 Related Work

Researching ways to share data is a key challenge in the design of distributed systems. Early approaches like the well-known NFS[PJS+95], AFS[How88] and Microsofts CIFS[Mic17] focusing on exporting a whole file system to a computer network. In case of CIFS and NFS this is achieved by providing a network Application Programming Interface (API) to allow syscall-like file access on a local file system to a remote user. Newer systems like BeeGFS[Hei14], Ceph[WBM+06], Lustre[S+03] and GlusterFS are extending these ideas by adopting newer features like the decoupling of metadata from the actual content of a file and scaling the content to multiple servers. In addition, they allow using different replication mechanisms like mirroring or striping of the data to provide high availability and high throughput for file system access operations.

A different approach to store data in a distributed manner is to use a database. For structured data like sensor data, a relational database system could be used. To store large amounts of arbitrary unstructured data more modern approaches like NoSQL databases could be used. The usage of NoSQL databases gained further relevance with the constant development and spread of cloud computing applications, which sets a special focus on „large scale and high-concurrency applications“. Googles Bigtable, Redis, Cassandra, MongoDB and CouchDB are well-known examples of NoSQL databases of different types and purposes, ranging from simple key-value-stores over column-orientated design to complex document databases [HELD11].

However, both approaches lack a fundamental primitive to share data and instead use a high-level abstraction in terms of a file system or a Database Management System (DBMS) software. Another downside is that such proposals often do not set a focus on constrained hardware and assume at least a significant amount of computational power, storage capacity or huge amounts of stored data to work properly. Research that is focusing on a pure block or byte-orientated way to share data is rare. Distributed Replicate Block Device (DRBD) is a commonly used software to build a distributed block device that is shared amongst multiple nodes. A downside of DRBD is the fact that it is tied closely to the GNU/Linux operating system making it infeasible to use in some constrained environments. Furthermore, it is limited to an active-standby replication scenario where the storage overhead on all standby replicators is equal to the size of the shared block device [Ell08]. Other interesting approaches to build distributed block storages include VBS-Lustre[GMP+10] and systems that are using erasure codes such as Reed-Solomon-Codes. Virtual Block Store (VBS) is a block storage

device interface for cloud infrastructures. It is intended to work like Amazon EBS with primitives to create, delete and attach storages to virtual machines without the limitation of being tied to a single cloud provider. It can be used as a raw block storage and can be used exclusive amongst multiple virtual machines. VBS-Lustre is a distributed storage system, build on top of VBS, which makes use of the Luster file system to extend the existing VBS volumes to enable a distributed usage between multiple machines.

Erasure codes, in contrast, provide an optimal space efficient way to store blocks of data in a distributed system [FMS+04, GWGR04, AJX05]. The widely used Hadoop file system makes use of erasure codes in the „HDFS RAID“ module, which uses a combination of striping and erasure codes to provide failure resistant storage for a Hadoop distributed file system. In contrast to the proposal of this thesis VBS-Lustre as well as HDFS make use of an additional file system to provide a distributed functionality on a non-distributed block storage.

A fairly new approach on distributed storage engines is Software Defined Storage (SDS), which uses software abstractions on top of multiple independent storage systems to build a uniform logical storage interface. The Storage Networking Industry Association (SNIA) defines SDS as „Virtualized storage with a service management interface. SDS includes pools of storage with data service characteristics that may be applied to meet the requirements specified through the service management interface“. While providing a simple and easy extendable way to build distributed storage devices as well as a feature-rich software ecosystem, SDS gives strong freedom of choice to the user to build a storage system to fit their requirements. A downside of using a SDS solution is the common focus on large scale datacenter applications without a use-case for small scale constrained environments.

Another common way to build distributed data storages are Distributed Hash Tables (DHTs). Implementations like Chord, Content Addressable Network (CAN), Pastry, Tapestry and others are building overlay networks in a P2P environment to map arbitrary data onto keys in a distributed manor [AR06]. Since they are not limited to a specific data layout stored in the DHT and only providing a protocol for a structured organization of nodes in a P2P network the purpose of a DHT is highly implementation specific. The DHash storage engine build on top of the Chord DHT uses a block-orientated design to act as a storage system. Multiple application like the Cooperative File System (CFS) and Ivy file systems are built on top of DHash to prove its practical use as a distributed storage engine [DBK+01, MMGC02].

2.2 Used Technologies

In this section, an overview of technologies and terms used in this thesis is provided. Some important key technologies included in this work are pointed out for further reference. The explained terms are mostly relevant to the implementation provided in Chapter 5 as well as for the further explanation of used acronyms.

2.2.1 Distributed Hash Tables

A Distributed Hash Table applies the data structure of normal hash tables to a distributed system. As in a classical hash table, a distributed hash table consists of a constant number

of buckets in which data items get stored identified by the hash of the data module the number of available buckets. In a distributed hash table a single node acts as a bucket which is able to store data elements.

2.2.2 Consistent Hashing

DHTs often make use of a technology called consistent hashing to distribute the load over the available nodes. In respect to classical hash tables where an additional bucket in the table can result in a complete remapping of all elements, consistent hashing requires only an upper bound of $\log(n)$ data items to be remapped if a bucket is added or removed from the table [KLL+97a]. Consistent hashing as introduces by Karger et. al provides a numerical id to each node and data item by hashing the content of the data or a node's identifier and store the data on the node which provides the hash that is the direct successor of the data items hash.

2.2.3 RIOT OS

RIOT is a self-declared operating system for the internet of things. The aim of RIOT is to provide an operating system for small and constrained IoT devices with a special focus on low computation and memory overhead, as well as low power consumption. RIOT provides many standard APIs like a Portable Operating System Interface (POSIX) compatible socket, networking and file system interface to provide a generic and portable way to run applications on different hardware layouts. It is written in the C programming language and uses a strict modular design to make the system extensible and to provide a convenient way to replace existing parts of the system [BHW+12].

2.2.4 Block Storage

A block storage is a storage device which divides the usable storage into a set of smaller blocks. A block is therefore the smallest fetchable part of the storage. Usually, the used size of a single block is a fixed constant. A common example for block storage devices are hard disk drives and some flash based storage media.

2.2.5 Pseudo Random Hash Functions

A hash function describes a mathematical function which is able to map a large set of possible inputs to an output of fixed size. A pseudo random hash function has the additional property that the outputs also appears to be random. That means that outputs are distributed evenly among the possible output domain and small changes in the input string result in a different output which is not related to previous outputs.

2.2.6 Remote Procedure Calls

A Remote Procedure Call (RPC) is a protocol used by software implementations to execute implemented procedures on remote computers. Arguments, relevant for the remote function are serialized into a network packet and transferred to the remote computer. After the call

to the remote function is executed successfully a return value can be sent back to the calling entity.

2.2.7 POSIX

The Portable Operating System Interface is a set of standards focusing on operating systems. The standards are specified by a collaboration between the IEEE association and „The Open Group“, an industry consortium, focused on the development of standards. POSIX is used to provide a standardized API and behavioral aspects implemented by different operating system.

3 Requirements

In this chapter, the requirements for a distributed block storage are examined and summarized for further use in the system design. The requirements can be divided into functional and non-functional requirements. Functional requirements are often defined as behavioral aspects of a system and its outputs to a set of given inputs. The term non-functional requirement on the other side is not clearly defined. Many of the common definitions explain non-functional requirements as non-behavioral attributes or properties of a system such as „portability, reliability, efficiency, human engineering, testability, understandability, and modifiability“ [Gli07, Dav93]. This discrimination sometimes can be vague, due to the fact that a requirement which results from one aspect of the solution is closely coupled to another aspect and vice versa. While a proposal for a working solution must fulfill all of the functional requirements the non-functional requirements sometimes are implicit and can be optional. A final evaluation of the implementation has to verify if all the functional and non-functional requirements are covered. Furthermore, an acceptance criterion for every requirement is defined in this chapter. The acceptance criteria can be later used to verify whether the design and implementation adapt the requirements. Additional, a set of security relevant requirements which focus on a security concerned implementation of the network protocol and the application itself are provided. The implementation of the security relevant requirements are out of the scope of this thesis and considered as future work.

3.1 Functional Requirements

1. **Raw Device Interface**
   To make use of a block storage device an interface that specifies multiple access operations must be defined and implemented. These actions can range from simple read and write commands to more complex and device-specific functionality. A well-known and widespread interface for device operations is the Small Computer System Interface (SCSI) [Sea18]. A subset of the available SCSI commands should be implemented to provide a common way to access the resulting device.

   Acceptance criteria: At least a command to read and write a single block of data must be defined. A more sophisticated implementation can also implement read and write operation for multiple blocks as well as additional SCSI commands. The user must be able to write a specific block of data and read it afterward.

2. **Concurrency**
   Traditional block devices like optical or magnetic disks are often not intended to be used concurrently by multiple applications. This can lead to multiple problems including data inconsistency. As stated in the scenario the resulting device should consist of multiple distributed nodes in a P2P environment and thus is likely to face concurrent

read and write operations without a central authority. A way to ensure operation of mutual concurrent read and write access on the resulting device must be provided.

Acceptance criteria: Multiple write accesses on one or more data blocks should be executed in the correct order and lead to correct data after the last write access. Multiple users must be able to write multiple blocks concurrent without getting inconsistent data.

3. **Storage Size**
   While some file systems support the usage of storage devices with dynamic size either directly or due to online resize possibilities, the majority of file systems expects at least some information about the size of the underlying storage architecture. Since the resulting system should consist of multiple devices with different storage back-ends and sizes we need a way to communicate the overall storage size of the resulting block device in the network. The resulting system needs to implement a way to communicate the overall storage size to all nodes even if nodes join or leave and thus resize the overall storage in the network.

   Acceptance criteria: Multiple join or leave operation should be communicated across the network and the network should converge to the correct storage size over time. When a node joins or leaves the network any other node in the network must be able to update its information on the overall size in between a given period of time.

4. **Load Distribution**
   To provide an optimal utilization of participating nodes in the network and a minimal need to re-balance items on node failures the network should provide an even load distribution in terms of storage utilization to the participants. Since different nodes can provide different storage sizes the mechanism needs to utilize the load depending on a node-specific weighting factor. Using a P2P network implies that no single node is necessarily aware of all other nodes weights. This makes the weighted distribution a challenging aspect in a distributed storage device.

   Acceptance criteria: The probability for every data item to get stored at a node n_i with positive weight w_i needs to be $\frac{w_i}{W}$ where $W = \sum w_i$.

5. **Data Availability and Consistency**
   Churn events can happen quite often in a P2P network. A mechanism to ensure the availability and correctness of data must be implemented, even when multiple devices fail simultaneously. The used mechanism needs to relocated data in the network in case of node joins and failures and must be able to reconstruct correct data.

   Acceptance criteria: The design of the resulting network must guarantee to deal with a certain amount of churn. After multiple nodes fail a user must be able to look up the correct data in the network.

6. **Locality**
   As already mentioned in Chapter 1 a common problem with cloud-based solutions is the need for a constant or periodic uplink to the internet. To work around this requirement the target network should even work in a Local Area Network (LAN) scope without any uplink into other networks. This means that the routing and the organization of the network can take place locally, without a central authority.

Acceptance criteria: The resulting system must be able to run in a local network without any uplink to other networks, as well as across network boundaries.

7. **Guaranteed Lookups**
   As soon as data is inserted into the network, a guarantee that a lookup for the newly inserted data succeeds, has to be provided. This is due to the fact that simultaneous access on a single block can occur anytime and thus an override on a more recent block should be prohibited.

   Acceptance criteria: After an insert operation into the storage gets acknowledged by the responsible storage location, any other member of the network must be able to query the data immediately.

3.2 Non-Functional Requirements

1. **Awareness on Computational Constraints**
   As stated in Chapter 1 the resulting storage solution should work in an environment of multiple constrained devices like smart homes or sensor networks. When choosing a network the low computational power and the limited storage size of the participating nodes has to be respected. Both resources should be used as little as possible to ensure no disruption of the main application running on the device.

   Acceptance criteria: An example target environment must be defined and the Central Processing Unit (CPU) utilization and the storage requirements of the resulting storage system need to be checked against the available resources of the target environment. The resulting system should not use more than a fraction of the available resources.

2. **Path Length & Network Size**
   The used network should support a good trade-off between network size and hop size to reach an arbitrary subscriber in the network from any other point in the network. Since the number of participants in the network should not be limited to a specific size the network should ensure a good upper bound for maximum hop count in the network. Furthermore, it is important to not overwhelm the limited memory on constrained devices, making a network with static memory usage for pointers and routing tables feasible.

   Acceptance criteria: An upper bound for the number of hops until a certain destination is reached must be provided.

3. **Performance Indicators**
   Depending on the underlying available storage technologies and sizes the performance of the storage must ensure the usability of the system. The SNIA defines multiple performance metrics for storage devices. Important basic performance metrics are I/O operations per second, Throughput per second, response time in milliseconds, retries per store operation, I/O per watt, service time and average store queue depth [SNI13].

   Acceptance criteria: On a given hardware the performance is bounded by the underlying storage technology. The throughput should be measured and any deviation from the theoretical maximum needs to be reasoned.

4. **Node Bootstrap**
   In a P2P environment, there is no central authority that is well known to new nodes that want to join the network. Ideally, a mechanism needs to be established that allows any node to join the network without any pre-configured entry nodes. Furthermore, one could improve the bootstrap process in a way that allows nodes to join a network independent of its own configuration by negotiating the configuration with the entry node.

   Acceptance criteria: A node should be able to join an existing network without any pre-configured entry node.

3.3 Security Considerations

P2P networks are often built as structured and unstructured networks where multiple untrusted participants provide and request services from and to each other. Famous examples of such protocols are Gnutella, Freenet and the Napster protocol. While many of such publicly available networks require no authorization, there are multiple security considerations to take into account. With the focus on a distributed storage device where the possible participants may be limited to a few trusted computational devices, this becomes even more relevant since the provider of such networks may want to keep the stored data confidential. In the remainder of this chapter, security considerations are divided into protocol security and application security considerations. While an actual design and implementation of the measures suggested in this chapter are not in the scope of this thesis the author acknowledged the need of security considerations for a modern software environment and therefore wants to give an overview on best practices and views a security concerned implementation as future work on the system.

3.3.1 Network Security

To implement a secure distributed system it is important to provide a secure protocol for the communication between the nodes in a network. The Internet Engineering Task Force (IETF) provides RFC3365[Sch02] with an overview about the important definitions and defines security as a „SHOULD IMPLEMENT“ feature of a protocol, meaning that a protocol does not necessarily needs to enforce security actions on the protocol, but instead should implement them and let the user have the freedom of choice wherever to enable them. This property is important for constrained environments where the use of modern cryptography can easily overwhelm a device. While modern processor already provides instruction sets extensions and hardware acceleration for common cryptographic use cases. Microcontroller Units (MCUs) often completely lack this feature. This makes it rather expensive to use modern cryptography on embedded hardware. The authors of [CBG12] have implemented five candidates for the SHA3 hash function contest and evaluated the performance gain on a 16-bit MCU when an additional instruction set extensions are used. The performance gain for contest winner Keccak was as good as 30% in terms of CPU cycles and 20% in terms of data memory access operations. Due to the importance of security and cryptographic operations in modern distributed systems, as well as the significant performance improvements an

extension on the instruction sets of modern MCUs can provide, it is likely that it gets more common for future MCUs to support such features.

The IETF defines three important security services a network protocol must implement. Namely they are the „Authentication service“, the „Data confidentiality service“ and the „Data integrity service“ [Sch02]. It is not necessary for a protocol to implement every security service and rather depends on the use case of the protocol which must be validated in the context of every protocol.

The IETF defines the authentication service as „A security service that verifies an identity claimed by or for an entity, be it a process, computer system, or person. At the internetwork layer, this includes verifying that a datagram came from where it purports to originate. At the application layer, this includes verifying that the entity performing an operation is who it claims to be“. While authentication of participants is not relevant in every possible setup of a distributed storage device the majority of real-world usages are expected to want to control wherever a node in the network is allowed to participate and join or not. This raises the need for the implementation of an authentication scheme in the protocol, making it impossible for non-authorized entities to participate in the network. Since one does not want to lower the guarantees provided by the P2P nature of the system a certificate-based approach can be used to implement an independent distributed validation of any participant in the network. Other solutions could implement central authority based validation of participants as suggested in RFC3365. Depending on the computational capacities in a network it is further possible to use a certificate-based approach to sign all messages or use a message authentication code with pre a exchanged secret key. An additional benefit of this proposal is that a node in the network would be able to detect modifications on the transferred payload and ignore messages which are changed either intentional due to an attacker or unintentional due to a failure in the transmission.

A data confidentiality service protects data against unauthorized access. For a network protocol, this means that data should not be transferred in clear text over an untrusted network. To ensure data confidentiality some type of synchronous encryption protocols like Advanced Encryption Standard (AES) can be used. It is important to provide a secure key exchange protocol to enable all participants to communicate securely without sending the keys over an insecure network. An implementation can make use of digital certificates to enable an encryption protocol like Transport Layer Security (TLS) to provide a secure key exchange as well as the encryption of the messages. Since asynchronous key exchange is very costly compared to the act of synchronous encryption of data blocks the user may want to have the option to use a predefined secure key stored directly on the participants to speed up the communication. A downside of an approach where keys are pre-distributed is the fact that an attacker only needs a single key to gain access to all communication, where a secure key exchange protocol can make use of different keys for different communication channels.

Data integrity refers to the correctness of the data and protects against unauthorized changes to the data transferred over the network. Using a message authentication code or encryption protocol as described in the last paragraphs could already give strong guarantees on data integrity. However, securing the data from unauthorized access may not be enough. In addition, one may want to secure the data from authorized

but unintentional modification. For this purpose, an error checking and correction mechanism can be used. However, since error correction is already enforced on many of the lower levels of a network protocol including the transport protocol an error detection or correction solution may provide no advantages.

3.3.2 Application Security

The application itself needs to ensure a certain kind of security independent of the security guarantees of the network protocol used. While there is nothing as a „secure application“ it is important to evaluate common best practices to avoid common pitfalls and give the best possible guarantee on the security of the application. The Open Web Application Security Project (OWASP) provides a guideline which focuses on developing of secure software to avoid common mistakes [OWA10]. Important examples are the correct validation of user-generated input, correct memory management and system security. Since the OWASP mainly focus on a client to server communication over the web not all suggested features are relevant for a P2P style network and can be omitted. Testing can also help to protect from already discovered security incidents and ensure the absence of the same errors in the future.

4 Design

In this chapter, the final design as a result of the requirements of the last chapter is provided. First, an appropriate P2P protocol is chosen such that it already covers as many requirements on the network as possible. Furthermore, a design for a bootstrap protocol that allows new nodes to join into an existing network and the block storage on top of the P2P protocol is proposed. The protocol is extended to match the remaining requirements. This includes improvements on the even load distribution, the placement and redundancy policy of the storage as well as the communication of storage sizes across the network. At the end of this chapter, an overview of the designed modules is provided.

4.1 Peer-to-Peer Protocol

Optimally the P2P protocol used in this thesis should already cover as many properties from the requirements as possible. In this section, an overview of different protocols should be given and an optimal protocol in terms of the covered requirements should be selected. Furthermore, changes to the chosen protocol should be discussed to match the remaining requirements.

4.1.1 Structured vs. Unstructured Networks

P2P networks are often distinguished into structured and unstructured networks [SW05, p. 15]. While unstructured networks perform well in terms of node joins and failures, the overhead of finding data in an unstructured network can be rather high since data is usually found via a flooding mechanism. Another problem is the fact that there is no correlation between the data and the participant which holds the data and thus leads to cases where lookups result in failures even if the requested data is stored somewhere in the network [SW05, p. 84]. Structured networks, on the other hand, provide an organization of the participants in a defined topology. A commonly used technology for structured P2P networks are DHTs [SW05, p. 15]. In a DHT a unique identifier gets assigned to any data item through hashing the data. This maps data items onto the node which is responsible for the address region which includes the unique identifier [SW05, p. 87].

Structured networks provide less resilience against node joins and failures as unstructured networks, but can guarantee that every node in the network can be reached via the used topology and furthermore, the ability to look up the responsible node for a given data item from any node in the network. The main drawback of using a structured P2P network is the lack of support for so-called fuzzy or complex search queries. To retrieve a data item through a search request the identifier of the data needs to be known [SW05, p. 85]. This drawback is bearable since we only perform access to well-known blocks of the storage device and do

not perform searches on the raw data. Since the content of a single block seldom appears as a coherent entity, searching with complex statements provide no advantages.

Due to the fact that unstructured networks violate the requirement for guaranteed lookups, a structured network needs to be chosen.

4.1.2 Comparison of Structured P2P Networks

Structured P2P networks are mostly implemented as DHTs. Like a traditional hash table, a DHT also consists of multiple buckets where every participant in the DHT represents a single bucket which is responsible for a range of items. To avoid a reordering of every item in the hash table when a churn event happens, consistent hashing algorithms are used to re-balance only a subset of $\log(N)$ items, where N is the number of items in the hash table.

Several well-researched and practically proven DHTs are available, such as Chord, Pastry, CAN, Kademlia, Viceroy and Symphony. The differences between these Protocols are mainly different optimizations in terms of needed routing hops or management overhead.

Table 4.1 shows an detailed overview on the mentioned DHT designs for later reference. In case of CAN the parameter D describes the number of dimensions used. In Pastry the parameter b divides the 128 bit id space into different segments each being b bits long. The parameter k used by Symphony and Viceroy can be chosen freely and is a trade off between routing complexity and the network overhead introduced by additional links to other nodes.

DHT	Routing Complexity	Node State	Arrival	Departure
Chord	$\mathcal{O}(\frac{1}{2}\log(n))$	$\mathcal{O}(2\log_2(n))$	$\mathcal{O}(\log_2^2(n))$	$\mathcal{O}(\log_2^2(n))$
Pastry	$\mathcal{O}(\frac{1}{b}\log(n))$	$\mathcal{O}(\frac{1}{b}(2^b-1)\log(n))$	$\mathcal{O}(\log_{2^b}(n))$	$\mathcal{O}(\log_b(n))$
CAN	$\mathcal{O}(\frac{D}{2}n^{\frac{1}{D}})$	$\mathcal{O}(2D)$	$\mathcal{O}(\frac{D}{2}n^{\frac{1}{D}})$	$\mathcal{O}(2D)$
Symphony	$\mathcal{O}(\frac{1}{k}\log^2(n))$	$\mathcal{O}(2k+2)$	$\mathcal{O}(\log^2(n))$	$\mathcal{O}(\log^2(n))$
Viceroy	$\mathcal{O}(\frac{1}{k}\log^2(n))$	$\mathcal{O}(2k+2)$	$\mathcal{O}(\log_2(n))$	$\mathcal{O}(\log_2(n))$
Kademlia	$\mathcal{O}(\log_b(n))$	$\mathcal{O}(b\log_b(n))$	$\mathcal{O}(\log_b(n))$	$\mathcal{O}(\log_b(n))$

Table 4.1: Performance comparison of DHT systems. The columns show the averages for the number of routing hops during a key lookup, the amount of per-node state, and the number of messages when nodes join or leave the system [SW05, p. 116]. The Parameter n is used for the size of the identifier space.

Chord Distributed Hash Table

In this section, Chord should be introduced as a P2P Network. Chord uses a traditional DHT style ring topology and extends it with a so-called „fingertable“ to optimize the lookup for an item in the ring to $\mathcal{O}(\log(n))$ where n is the size of the ring. Another benefit of using Chord is that it allows a rather strong resilience against simultaneous leaving nodes. „Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance“ [SMK+01a].

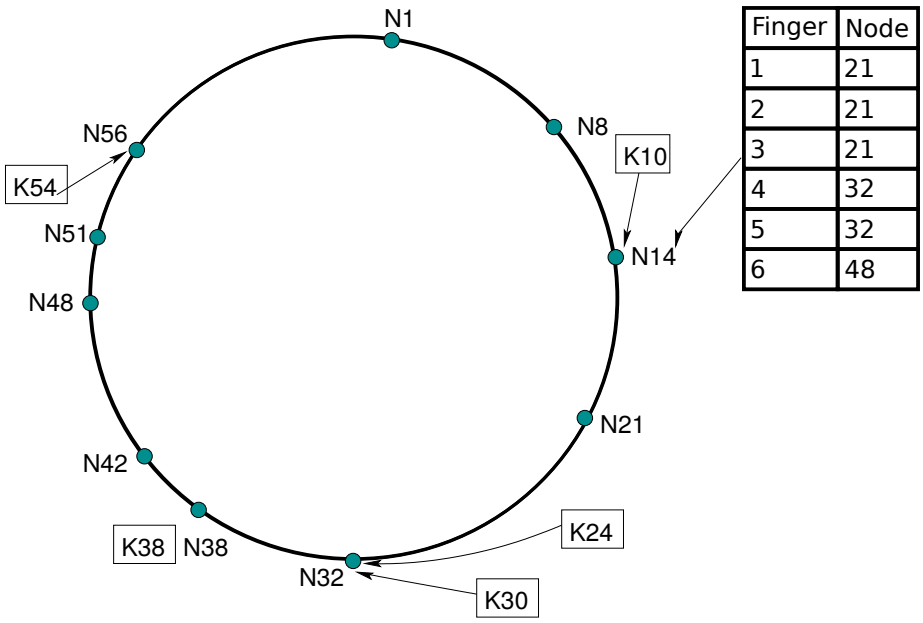


Figure 4.1: A Chord ring with 10 out of 64 possible nodes. Labels starting with „K“ are referring to keys and labels stating with „N“ are referring to nodes. The table on the upper right shows to the fingertable of node N14 [SMLN+03]

Chord uses it's linear ring topology to store items in a network of size N. A participant in the Chord ring is called a „node“. To receive a position in the Chord ring a node must compute a hash value of its own identifier. An example of a node identifier could be a nodes IP address. This can make it impossible for an attacker to gain an arbitrary position in the ring since the identifier of a node can be verified by other nodes. A node $\mathcal{A}$ can either create a new ring or join an existing ring by sending a message to any known member $\mathcal{B}$ of the existing ring, asking for its own immediate successor $\mathcal{C}$. After $\mathcal{C}$ is found a „notify“ message is send to inform $\mathcal{C}$ about its new predecessor $\mathcal{A}$. Any node in the ring periodically sends a message to its successor asking for the successor's predecessor. If the returned predecessor is not the same as the requesting node, the node needs to update its successor and send a notify message to its new successor [SMK+01a]. This ensures that the algorithm convergence to a correct state over time. To provide resilience against failing nodes every member of a Chord ring stores an additional data structure called a „successorlist“. A successorlist is a simple list of the following $\log(N)$ subscriber of a Chord ring and gets constructed by copying the successorlist of the direct successor and removing the furthest node while adding its direct successor. Another important property of a Chord implementation is the already introduced fingertable. A fingertable is an additional table with $\log(N)$ entries, where each entry includes a pointer to different nodes in the networks. The fingertable is constructed such that the i^{th} entry of the fingertable is the node which is the direct successor of the id $n + 2^{i-1}$ where n is the nodes own identifier. Using this additional set of links it is possible to reach every node in the network in a maximum of $\log(N)$ steps.

Content Addressable Network

CAN or Content Addressable Network also provides DHT functionality like Chord. Unlike in Chord, a CAN node does not have a fixed node identifier depending on its IP address and thus provide less resilience against malicious nodes. The used topology for a CAN network is a multidimensional euclidean space with a dimension $d \geq 2$. Nodes joining this network select any random coordinate tuple $(a_1, a_2, \ldots, a_d)$ and ask any known node which node is responsible for this part of the id space. After the responsible node is found the new node ask the existing node to split its space at one axis. A node split its own space onto two new sub-spaces by first splitting the x-axis, then the y-axis up to the $d-1th$ axis. The newly arisen void is then occupied by the new node. A CAN node needs to keep track of $\mathcal{O}(2d)$ neighbors, where a neighbor is any node which „overlaps with another node on $d-1$ dimensions and abut along 1 dimension“. A routing request for a single point in the space is done through a greedy routing algorithm which forwards the request to the neighbor with the minimum euclidean distance to the point. It could be proven that the maximum number of hops needed to reach any target is therefore $\mathcal{O}((\frac{d}{4})(n^{1/d}))$ [RFH+01].

Conclusion

The design of distributed hash tables already covers many of the requirements on the network:

1. **Load Distribution**
   Since DHT like systems relies on a hash function to place items onto nodes, the load gets distributed even by choosing a hash function which provides nearly uniform distribution of its outputs [SMK+01b]. A good example of such hash functions are the members of the Secure Hash Algorithm (SHA) family. In practice, the even load balancing suffers from multiple problems including an uneven balancing when the amount of nodes is rather small as well as no support for weighted distribution.

2. **Computational Requirements**
   A DHT itself has low requirements on computational power. To keep a DHT in a correct state a node must go through a synchronization period which consists of sending multiple messages to its direct neighbors. Therefore the communication done by a DHT to keep the state is limited by the number of pointers to other nodes. With increasing the time between two synchronization periods the requirements on the computational overhead can be lowered, with the cost of increased time for a DHT to get fully synchronized after a node joins or leaves. To receive a data item from another node a single message must be sent to the target node. This message must be routed through the network and requires multiple hops until it reaches the target. The amount of hops is limited in structured P2P networks according to Table 4.1.

3. **Path Length & Network Size**
   A DHT network uses some kind of data structures to keep track of multiple neighboring nodes. The data structures are often used to optimize lookup times to an upper bound. Chord for example uses an upper bound of $\log(n)$ neighbors to route a message with a maximum hop count of $\log(n)$ [SMK+01b]. CAN allows more fine-tuning of these parameters since it strongly depends on the size of the dimensional space. The same

guarantees as in Chord can be achieved using a value of $d = (\log(n)/2)$ [RFH+01]. Since the state a single node must keep is limited to a constant value, the network can theoretically grow infinitely. In reality, the size is limited number of possible node ids.

4. **Locality**
   DHTs as P2P overlay networks have no need for an external authority outside of a local network. Every node in a DHT can be an equal eligible member of the network and even create a new ring.

5. **Guaranteed Lookups**
   Since a data item can be directly mapped onto a single node in the network the data is available as soon as it is inserted into the network. Even when the neighbor tables on the requesting node are not correct it is always reachable via a linear ring search over all participants [SMK+01b, RFH+01].

4.1.3 Bootstrap

Bootstrapping in a P2P network describes how a new node is able to join an existing network as well as creating a new network on demand. DHT networks like Chord and CAN rely on the fact that a joining node is aware of at least a single entry node in the network for a first contact and for retrieving an initial position in the network. Due to the P2P based nature of such systems, a node is also always able to create a new network if no such node is available. The designs of existing DHT implementations do not make any assumption on how the entry node should be determined and simply assume that a list of possible entry nodes is available somewhere. The bootstrap node is sometimes also referred to as „Rendezvous Points“ or „Rendezvous Nodes“. In many distributed networks like TOR and IPFS, this is done via a fixed set of entry nodes provided with the distribution of the software [SDM04, IPF15]. This approach requires the entry nodes to be long lived and not be affected by churn which does not reflect the realities in a P2P network. Since entry nodes are able to leave the network due to unplanned outages this could result in a situation where new nodes are not able to join the network anymore. Furthermore, this results in additional communication overhead at the entry nodes. Multiple approaches focusing on how a list of possible entry nodes can be obtained dynamically as well as how to distribute the load for join operation in the network more evenly. Apart from dynamic obtainment of entry nodes, a good bootstrapping protocol should not make any assumption on the network size in the beginning and should be able to scale with the increasing size of a network [CH07].

One approach focuses on creating an additional P2P network with a special purpose at providing bootstrap information for other P2P networks. This shifts the initial problem of bootstrapping to the bootstrap network which then can act as a generic information base for any kind of P2P network. The initial bootstrap of the bootstrap P2P network is done via „Local Random Address Probing“ a derivation of „Random Address Probing“. In random address probing a node simply tries to contact random IP addresses hoping to find a member of the network behind the address. It is easy to see that this method is not very practical and heavily depends on the number of nodes in the probed address space and the overall size of the address space. Local random address probing only limits the possible address range to a subnet of the available address range [CH07]. This assumes that there is an existing node in the given subnet and furthermore this is also not practical in networks with large

address spaces like IPv6. Even if an unrealistic high amount of 2^{32} nodes in the bootstrap network is assumed the sheer amount of possible IPv6 addresses makes it nearly impossible to random guess the correct address. In addition, this approach is limited to hosts with a reachable IP address and does not work for nodes behind a firewall or Network Address Translation (NAT) and assumes a node runs on a given set of TCP or UDP ports.

Another approach to bootstrap nodes in a P2P environment could make use of network layer mechanisms like broadcast to enable nodes to ask all nodes in a given broadcast domain for bootstrap information. This can be extended with anycast to additionally allow nodes to join P2P networks outside their own broadcast domain with the additional benefit of choosing the best node in terms of neighbor awareness [CKF04]. The Domain Name System (DNS) can be used to further optimize this approach. Through using multicast DNS[SC13] a service could be propagated in a local network while regular DNS A or TXT records could be used to identify multiple bootstrap nodes in an internet-scale setup. The downside of this approach is the need to either set up a multicast DNS service in every local domain which implies that a node already exists in this domain or that we need to set up DNS records for well-known bootstrap nodes. This can reduce the stability of the P2P network since the network could only continue work until either the DNS server or all nodes which are represented by the DNS records affected from an outage. Using more dynamic DNS approaches like DynDNS services could provide better resistance against the later. Through self-registering of the nodes against a DNS server an optimal resistance against failing nodes could be provided. However, there is still a central authority outside the P2P network which can be affected by an outage which shortens the guarantees provided by our P2P network.

Conclusion

Bootstrapping a P2P network outside a local domain is a non-trivial task. Neither a static list nor a random probing of network addresses is feasible as stated in this chapter. One could ensure reachability by combining the methods named above. For a local network, a broadcast, multicast or multicast DNS approach is the most suitable since it can be implemented into the existing P2P network to allow nodes to announce them self in a given domain. This would result in no central authority and no single point of failure weakening the nature of the P2P network. For large scale application which should be able to connect over different networks, a DNS based round robin approach would be possible to obtain the address of one or more nodes. This address could make use of anycast to allow additional load balancing, fault tolerance and neighbor awareness. It can be stated that due to the „hen and egg“ property of this problem no realistic and P2P based way exists by this time.

4.2 Replicated Block Storage

The part of the system used for storing actual data in a DHT ring must solve the remaining requirements and address the already mentioned problems of weighted load balancing. In this chapter, an overview of data storage strategies in a DHT should be given as well an introduction to an existing storage solution for Chord style DHTs called DHash. Limitations on DHash should be examined and possible solutions should be discussed. Finally, the best solution in terms of solved requirements should be picked and a final proposal for a design should be provided.

4.2.1 Consistent Hashing for Data Storage

Data items get mapped to a node in a DHT due to the use of consistent hashing as introduced by Karger et. al [KLL+97b]. To store a set of data items V, a hash function h is used to compute the hash value of every item $v_i \in V$ as $h(v_i) \bmod N = x_i$. In case of Chord like DHTs the data item gets stored at the node which is the direct successor of x_i. Trough the use of consistent hashing data elements gets distributed evenly across the ring of N elements. As soon as an event occurs, where either a node joins or leaves the network, it is only necessary to reassign a small fraction of the keys in the hash table.

4.2.2 DHash

DHash is already utilizing as a storage layer on top of the Chord P2P overlay to store uniquely identified blocks in a network. DHash takes care of replication, caching and distribution of the data blocks [DKK+01].

1. **Distribution**
   The Distribution of data items in DHash is done with the original notion of consistent hashing as explained above.

2. **Replication**
   DHash replicates a block by storing it at the next k successors of the target node in the Chord ring. This can be easily achieved by using the nodes successorlist of size r. The constant k can be chosen to match $1 < k \leq r$, with a larger k to provide better resistance to failing nodes while increasing the needed storage space for a single data item.

3. **Caching**
   DHash uses a caching mechanism to improve the lookup time for popular data. Every node in the ring uses a fixed amount of disk space for caching. If a data element is retrieved via a route of nodes, every node in the path stores a copy of the data due to a Least Recently Used (LRU) cache strategy. Since nodes close to the target node are likely to get contacted by any querying node, caching can reduce the needed lookup time significantly.

4.2.3 Load Balancing

The method of using consistent hashing as implemented in DHash face multiple problems in terms of even load balancing and weighted load balancing. In a storage network, the participants are able to contribute different storage sizes to the network.

With the use of consistent hashing as defined above every node gets an even amount of data items. This ignores the heterogeneity of the nodes where different nodes can contribute different storage capacities. Furthermore, due to the random placement of the nodes in the ring, nodes can be responsible for irregular intervals of the id space. This can lead to nodes with small storage capacity getting a big share of the id space where nodes with large storage capacity getting a small share of the id space. In the remaining thesis, this is referred to as overloading and starvation of participants. Ideally, every node should be responsible for $\frac{K}{N}$ keys in a network with N nodes and K keys. Due to the uneven distribution of few nodes, this can be as bad as an upper bound of $(1+\epsilon)\frac{K}{N}$ keys per node. Without further optimizations the following upper bound can be proved $\epsilon = \mathcal{O}(\log(N))$ [KLL+97b]. To overcome these problems multiple solutions are proposed. As described above a solution need to provide good guarantees on weighted load balancing, overloading and starvation of participants. Further, a good solution should introduce only a little overhead in managing the DHT and migration of data items on churn events.

To select one of the described methods it is necessary to define multiple metrics for a possible solution. The metrics are mostly extracted from the requirements of Chapter 3 and the problems with the naive approach of consistent hashing as mentioned above.

1. **Fair Load Balancing**
   An important property is a fair load balancing that takes node heterogeneity into account. In a weighted load balancing scenario every node with weight w_i and overall weight $W = \sum w_i$ should get a fair share of $\frac{w_i}{W}$ data items. An intuitive approach would just assign a node with a multitude of storage capacity a factor of data items which correspondence with its capacity. This approach is used in Highest Random Weight (HRW) hashing methods. However as already explained in Chapter 3 this is not applicable in a P2P environment since there is no overview over all node weights without drastically decreasing scalability. Node heterogeneity also has an impact on the fair load balancing proposed by consistent hashing. Figure 4.2 shows a simulation of the mean node imbalance in a setup where every node has a capacity $2^6 \leq c \leq 2^{12}$ in contrast to another setup where every node has a fixed capacity of 2^6. As one can see using heterogeneity in node capacities have a negative impact on load distribution. This gets worse as the variance in node capacities grows.

2. **Overshooting of Single Participants**
   Since consistent hashing uses a pseudo-random function to map nodes and keys onto a ring of a fixed size it is possible that single nodes are responsible for a larger share of the id space than available storage capacity on this nodes. A good load balancing scheme must be aware of the utilization of the nodes and should not assign more data elements than available storage capacity. It is important to provide at least an upper bound of over-utilization to reserve a certain amount of storage capacity for overshooting elements. To waste as little storage capacity as possible it is important to keep this value as small as possible.

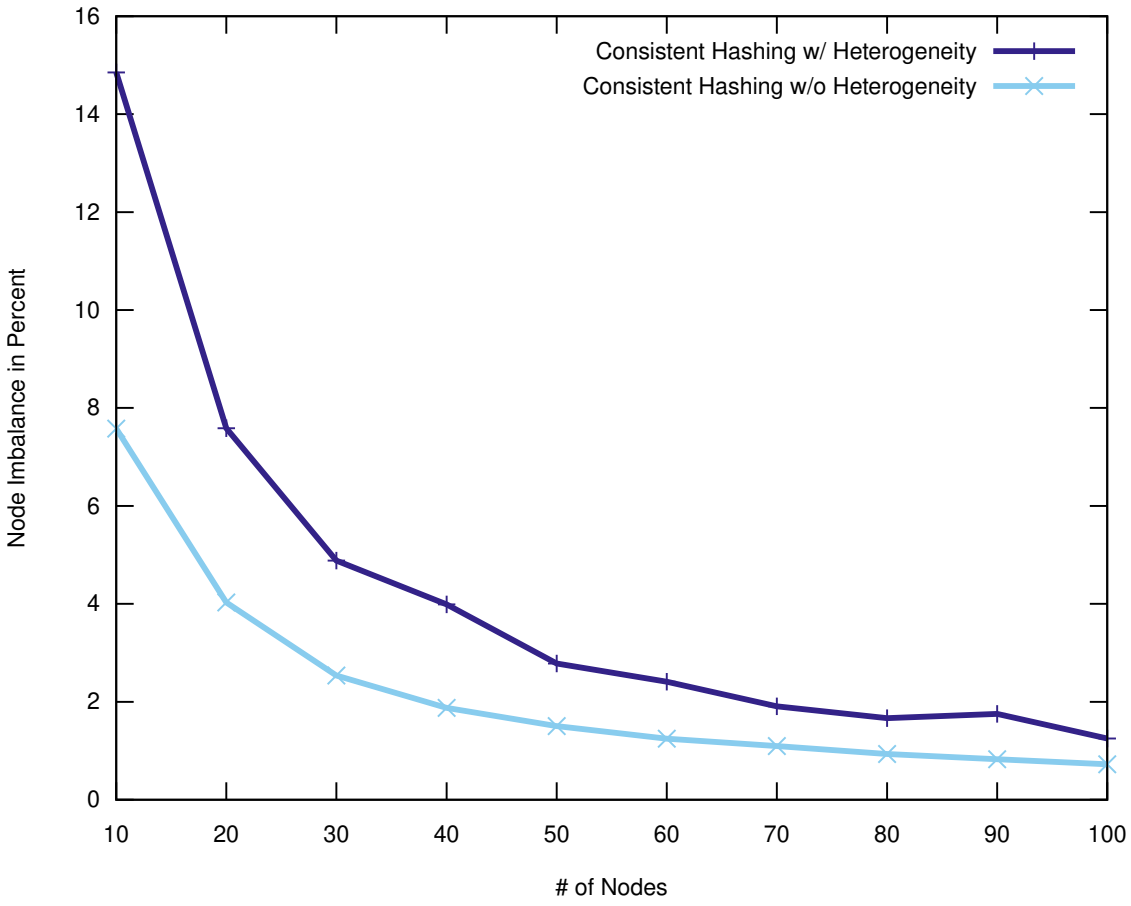


Figure 4.2: Mean difference on fair share in percent as the amount of nodes grows

3. **Consistent Hashing**
   For reliable lookups of data items, it is important to not weaken the guarantees provided by consistent hashing. A fast and guaranteed lookup of data items even if a node fails is crucial for a distributed storage device. If a solution implies a location of data elements depending on a temporary state it could lead to data elements that are not able to be requested even if they are already stored in the network. This would result in false lookups and thus to inconsistent data.

4. **Overhead in Communication**
   It is possible that some of the mentioned properties introduce overhead in regular state-keeping communication. Since this additional communication overhead can easily overwhelm small and constrained device it is important to keep this overhead as small as possible.

5. **Overhead on Churn Events**
   Some load balancing scenarios imply a certain overhead if nodes join or leave the network. This is due to updates of existing data structures and movement of multiple data blocks to a new location. Depending on how often a churn event happens this can have an impact on the performance of the whole storage environment.

Virtual Nodes

An early approach to solving the problem of uneven load balancing in rings with few nodes is to use virtual nodes. Chord suggest to use $\log(N)$ virtual nodes for every real node to reduce ϵ to an arbitrary small constant [SMLN+03]. The downside of this method is the increased overhead that arises for every additional virtual node on the owning physical node. Since every node needs to store at least a fingertable, as well as its successorlist, the communication channels and the memory usage of a single node is equal to $\mathcal{O}(2\log(n))$ partner nodes. Implementing virtual nodes increase this to $\mathcal{O}(2v\log(n))$, where v is the number of virtual nodes per real node, making it a worst of $\mathcal{O}(2\log^2(n))$.

Further methods of using virtual nodes do not enforce a fixed amount of virtual nodes per real node, but instead taking the different storages sizes into account by defining a minimum contribution weight w per node. A node providing the smallest possible contribution size only uses a single virtual node, where a node which provides a multiple $m \cdot w$ of the minimum weight use m virtual nodes. This results to a weighted load balancing in a heterogeneous network. This solution increases the communication overhead by a factor equal to the number of virtual nodes on a single host but does not provide a good guarantee on even load balancing over all nodes since the number of virtual nodes can be lower than the proposed $\log(n)$.

Godfrey et. al combines a method of arbitrary id selection and virtual nodes in a DHT based on Chord called Y_0 [GS05]. In this solution, a scheme for virtual server selection called Low Cost Virtual Server Selection (LC-VSS) is introduced to allow a node to create multiple virtual nodes in a consecutive interval to gain responsibility over a fraction of the id space with respect to the servers weight contribution. The average weight of all nodes c_n is normalized to 1 and every node with capacity c_v picks $\Theta(c_v \log(n))$ virtual nodes. Since the virtual servers are clustered, Y_0 does not imply that every real node needs to take care of $\Theta(c_v \log^2(n))$ network connections. Due to the fact that a node does not need to keep a link between two of its own virtual servers the scheme results in a communication overhead of $\Theta(c_v \log(n))$. The memory overhead of this solution grows by a logarithmic factor [GS05]. The use of LC-VSS let arise two problems. First of all, nodes with a capacity lower than a fraction of the average capacity cannot contribute to the network since this would introduce high communication overhead. The proposed solution to this problem is to allow the nodes to participate as a storage node for a chosen parent node, extending the parent nodes capacity. The other problem arises from the fact that nodes need to estimate the number of nodes in the network and the average capacity of all nodes. The performance of the network and the guarantees on load distribution and communication overhead are dependent on a correct estimation of these values. The paper introducing Y_0 makes suggestions for estimation algorithms without further validating them.

Multiple Hash Functions

Another method for load balancing in distributed hash tables is the usage of multiple hash functions. The power of two choices method uses a certain number of hash functions $n_h \geq 2$ to achieve a better load balancing property. The method originates from the so-called „Balls-and-Bins Problem“. In this experiment n balls are placed randomly into n bins. The upper bound of balls any bin should hold after all balls are placed is $\frac{\log(n)}{\log\log n}$. If we change the

algorithm such that every ball is placed in one out of d randomly chosen bins, where the ball is placed into the bin with the fewest balls, we get an upper bound of $\frac{\log\log n}{\log(d)+\Theta(1)}$ with high probability. This result shows that even if we introduce only a small amount of choices we can reduce the upper bound each bin holds exponentially [ABKU99].

This results in a more balanced allocation of the bins and can easily be applied to distributed hash tables. In this case, the participating nodes act as bins and the data items are the equivalent of the balls. To select multiple nodes we can simply hash a data item multiple times. This results in an overhead on storing and retrieving information in the DHT. Instead of a single hash, we must compute n_h hashes and furthermore need to query every successor of the resulting hashes to gain knowledge about the nodes storage capacities. The data element finally gets stored at the node with the freest capacity. To prevent the necessity of querying n_h nodes on data retrieval it is important to store an additional pointer on the remaining $n_h - 1$ nodes to not break the guarantees on lookup time provided by the DHT. This additional dynamic rise in memory is not feasible since it can overflow the limited memory of constrained devices.

The authors of the CAN DHT also suggest using multiple hash functions to improve path lengths and availability of data [RFH+01]. This can be easily combined with the method of the power of two choices to also improve load balancing. Again this comes with the same drawbacks as mentioned above, but can be optimized to a certain level by combining both methods. One can use error correction codes such as Reed–Solomon codes or a Redundant Array of Independent Disks (RAID) approach. In a naive approach, a data segment can be at n independent locations hashing the data element $2n$ times. The data element gets stored at the n least utilized locations. Assuming the probability of every node to get the data is $\frac{1}{2}$. This results in a message overhead on storing and retrieving data by a factor of two but would add additional data availability by using a redundant storage location for the data.

Working on Intervals

Another rather interesting proposal to spread load even in a DHT is originated from the physical process of Thermal Dissipation (TD). As heat energy spread even in any body, the proposal promises an even spread of data in a DHT like networks. In this method, the nodes in a DHT deliver data to surrounding nodes until an equilibrium state is achieved, where every node it utilized evenly. Similar to the method of arbitrarily selected node id's this is achieved by splitting intervals, moving a part of the nodes responsible interval to its neighboring nodes or moving nodes with few items to overloaded intervals. This implies an overhead on data redistribution to surrounding nodes. The heat dissipation algorithm provides two additional benefits. It includes fault-tolerant storage of items and does not allow nodes to be underutilized. With the original notion of consistent hashing, it is possible for nodes to gain only a very small section of the interval. Sometimes this can be as bad as some nodes only responsible for a single data element. Since the thermal dissipation method uses nodes with low utilization to reduce the load on heavy used nodes by moving them to different intervals this gets resolved over time [RPW04].

A slightly different approach working with intervals is implemented in the design of the distributed hash table CAN. In CAN a multidimensional torus with $d \geq 2$ dimensions is

used instead of a linear ring. New nodes choose a random position at arrival and splitting the biggest interval of all neighboring nodes into half to achieve control over the newly created section of the hypercube. The resulting DHT uses $\mathcal{O}(\frac{d}{4}n^{1/d})$ queries to find a data item using a constant factor of d pointers to neighboring nodes [RFH+01]. Since nodes can choose their position freely the fairness of interval sizes are determined by the number of random guesses a node uses to finds its entry position in the DHT. This can be used to achieve a constant factor load balancing of $\mathcal{O}(1)$ instead of $\Omega(\log(n))$ while only maintain a constant factor of $\mathcal{O}(\log(n))$ pointers per node and $\mathcal{O}(\log(n))$ queries for data location [AHKV03].

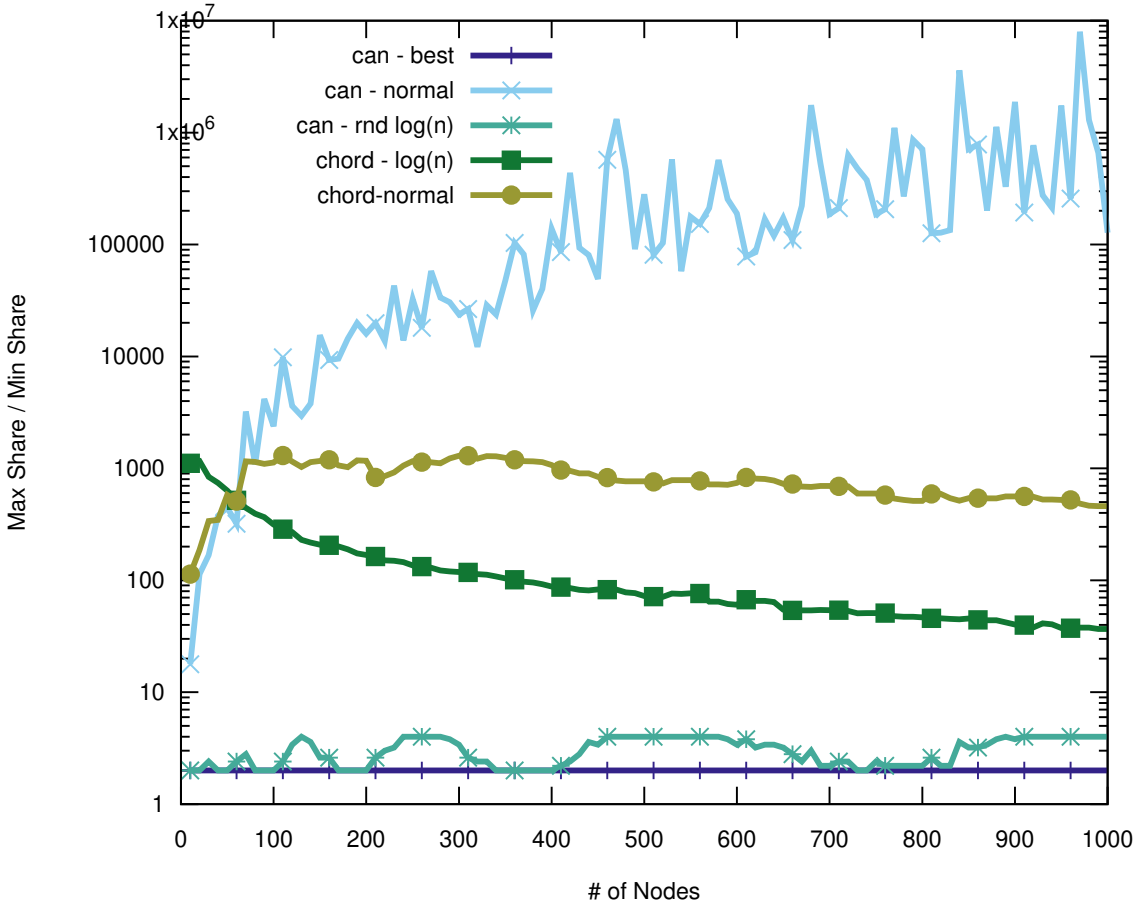


Figure 4.3: Load imbalance of Chord and CAN with $d = 2$ using a logarithmic scale

Figure 4.3 shows the ratio between the node with the maximum and the node with the minimum share in a Chord and CAN DHT after the insertion of a fixed number of nodes. The experiment is done 100 times and the average value is computed. „Chord - normal“ refers to Chord with random id placement, „chord - $\log(n)$“ refers to Chord with $\log(n)$ virtual nodes per real node, „can - normal“ refers to CAN where the split interval is determined by choosing a random node in the network, „can - best“ is CAN where the split interval is the best interval with respect to the size of the interval and „can - rnd $\log(n)$“ use CAN where the best out of $\log(n)$ random chosen intervals is split. The simulation ignores leaving nodes and only take care of joining nodes. Thus the simulation is only valid in cases where the number of leaving and joining nodes is nearly equal. As one can see CAN using random node positions, performs worse as more nodes join. This can be optimized by a huge factor if multiple node positions are taken into account reaching a near optimal value. To maintain a good value an additional re-ordering of the nodes needs to be done in the case of leaving nodes. The ratio in normal Chord and Chord with virtual nodes is rather high but in between expectations. In this simulation, Chord uses a ring of size 2^{16}. With 1000 nodes and an expected imbalance of factor $\log(n)$ we expect the nodes with the highest load to own about $\frac{2^{16}}{1000} * 16$ of the id space. Since its highly possible that nodes have only very small spaces of a few elements this results in a worst-case ratio of about 1000. Using virtual nodes can lower this factor. Through adding more nodes the max share decreases only by a factor of $\log(n)$.

Another approach which uses intervals to distribute load evenly reaches the same guarantees as CAN without making considerations on the underlying topology. The authors interpret the member in a distributed hash table as leaves in a binary tree where the location of each node is the binary representation of its unique identifier. The location of a node gets determined through a search algorithm in the binary tree. First, a random leaf of the tree is determined with a random search down the whole tree. After a leaf is found the new node gets inserted in the subtree of the selected leaf. The size of the subtree is determined by a function $\phi(l) \in [0, \log(N)]$. This function defines a frontier in the binary tree at a given depth which splits the binary tree into multiple subtrees. Choosing $\phi(l) = 0$ results in perfect balanced binary tree where $\phi(l) = \log(N)$ results in a random insertion in the binary tree. The insertion in the subtree is done by choosing the participant with the highest id share in the subtree and split its share into two halves. The new node occupies the lower part of the id space and thus becomes the new predecessor of the selected node. This id selection finds the location for any new node in $\Theta(R + \log(n))$ steps where R is the routing cost of the underlying DHT to find a given node and n is the number of participants in the network. This scheme results in a ratio of the largest to the smallest share of at most $\sigma \leq 4$ with a guarantee that at most a single participant must reassign its id if any node in the network leaves. Furthermore, the authors extend their algorithm such that the ratio can be further reduces to a most of $\sigma \leq 1 + \epsilon$ for any $\epsilon > 0$ with the need to reassign only $\mathcal{O}(\frac{1}{\epsilon})$ existing host on a node departure [Man04].

Highest Random Weight Hashing

Other possible solutions to achieve weighted load distribution are HRW hashing algorithms. In this approach nodes using a hash function $h(x, n) = v$ for a given set of nodes $N = \{n_0, n_1 \dots n_k\}$ and an arbitrary data item x. This needs to be done for all k nodes in

the network. The data item gets assigned to the node which maximizes the hash value v. This results in an even balancing of all data items and allows a simple notion of weighted distribution by allowing a node to participate multiple times in the node list N. On the downside this increase the complexity of the computation of the node responsible for any data item to $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$ compared to the method of consistent hashing. This only works for weights which are integer multiplies. Furthermore, any node needs to be aware of all node capacities in the ring, increasing the communication overhead by a large factor.

To overcome the restriction that node weights must be integer multiplies, different solutions are proposed. An early IETF draft called „Cache Array Routing Protocol“ addresses this problem by multiplying the resulting hashes with a per-node generated load factor to achieve arbitrary granularity in node weights [IET98]. A disadvantage of this approach resides on the fact that node weight factors are scaled relative to the size of the network. When a node arrives or leaves the network the load factors of all nodes are likely to change and the location of all data items must be recalculated. This renders the approach of CARP unusable for high churn P2P networks. Another approach is called RUSH, which uses a map of the storage hierarchy to distribute the keys to a node depending on any weighting value. This results in an approximately optimal redistribution of keys when a new device joins the network. Since CRUSH is designed for large data center application where node failures are unusual and mostly temporary it is prone to failing nodes. CRUSH solves failing nodes by keeping them in place because „failure is typically a temporary condition (failed disks are usually replaced) and because it avoids in-efficient data reorganization.“ This results in a high overhead in the computation of the data locations and makes it unusable for high churn networks [WBMM06].

A newer approach describes a method which combines a HRW system with distributed hash tables. The method called „The Logarithmic Method“ uses unscaled weights and a logarithmic distance function to assign data onto nodes. This allows to only relocate a fraction of the elements a failing node had stored. Furthermore, it proposes a data structure, which allows nodes to locate the responsible node for a given data item in $\mathcal{O}(\log(n))$ time. Again this approach suffers from the fact that every node must be aware of the other nodes and their corresponding weights [SS05].

Simulations

To validate different placement strategies introduced by the power of two choices method and HRW hashing, a simulation of these placement strategies is provided. To simulate the methods a non-distributed simulation of a simple DHT is used, in which a list of n participants with a random weight $x_i \in \{2^7, 2^8 \ldots, 2^{11}\}$ for each node n_i is generated. Afterward, the nodes are inserted randomly into a linear ring with size N and $X = \sum x_i$ data elements are inserted randomly according to the placement strategies. As a baseline value, the original notion of consistent hashing is used. Simulated placement strategies are the power of two choices method with $\log(N)$ choices, as a HRW method the minimum linear distance method as described in [SS05] applied over all nodes and the same method applied only to the $2\log(n)$ successors of the target node determined by consistent hashing and another HRW hashing approach where the elements hash gets multiplies with the weight of the next

$2\log(n)$ successors and placed at the maximized hash, called maximum linear distance. The measured indicators are the mean deviation of all nodes from their fair share of the id space e.g. $\frac{x_i}{X}$. Every test is executed 100 times and the average value of all runs is plotted. We expect all methods expect the original consistent hashing to perform optimal or near optimal when the number of node fall below a certain number. This is the case when the node amount is small enough to fit all nodes in local data structures and thus make it possible to keep every node into account when placing a data element. The power of two choices method with $\log(N)$ iteration should be near optimal if the amount of nodes is below $\log(N)$. The other methods using the next $2\log(n)$ nodes starting from a certain point and thus expect an optimal placement if the number of nodes below this value. In the simulation, 2^{22} is used as the ring size.

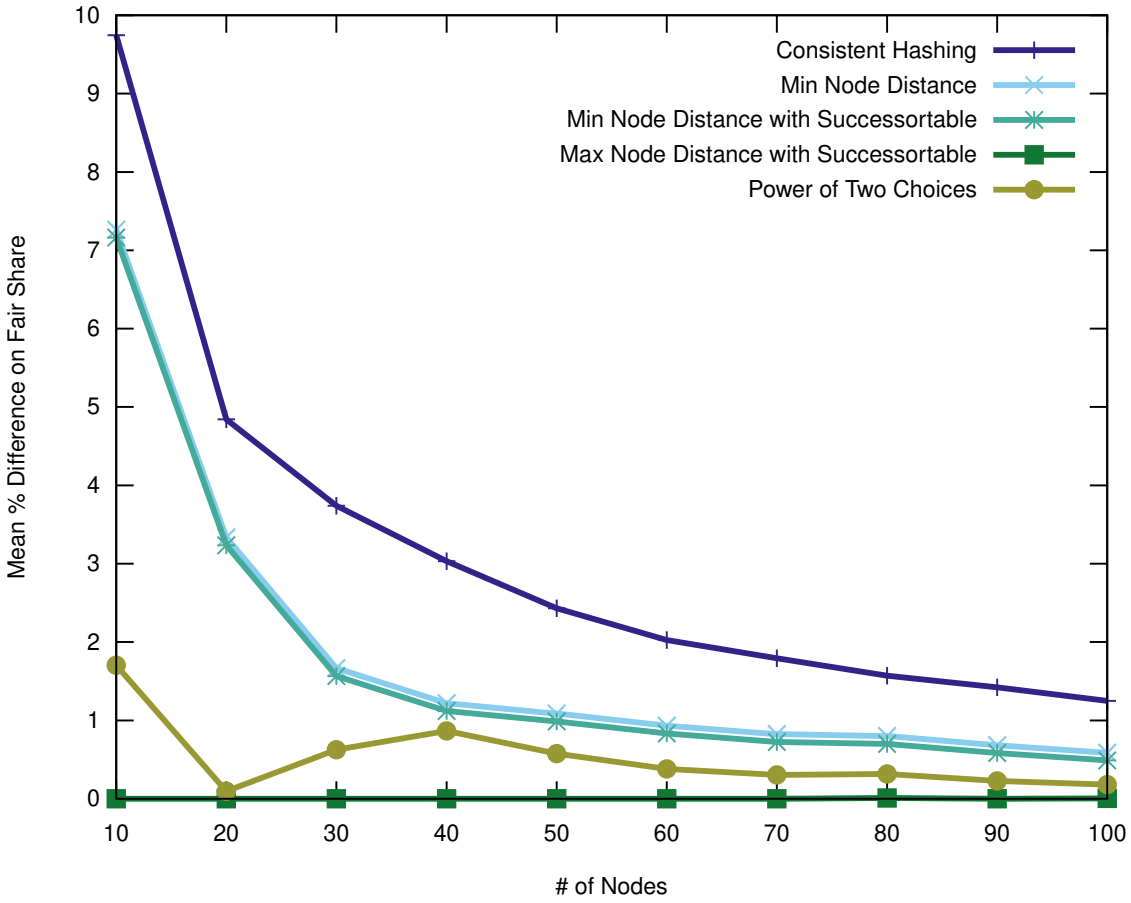


Figure 4.4: Mean difference on fair share in percent as amount of nodes grew

As one can see in Figure 4.4 the mean difference of all nodes on their fair share follows the expectations. The normal method of consistent hashing works as expected to get more fair with an increased number of nodes. Furthermore, we can observe that the distribution is optimal on a small node count for most strategies. The mean difference of the power of two choices method improves as node count rises. This is easily explainable since it is less likely that multiple hashes of the same data item get mapped onto the same node, improving its efficiency. The efficiency of a single data placement with the power of two choices method

is determined by the number of distinct nodes selected as possible targets. As the nodes are determined in a random fashion due to a pseudo-random hash function a single node can be selected multiple times.

It also turns out that using the minimum distance method performs equally good when only a subset of n_s out of all nodes are taken into account. This reduces the computational overhead of the linear method by a factor of $\frac{n}{n_s}$. The best performing method so far is the maximum neighbor method. If the network is smaller than the number of neighbors taken into account it results in an optimal placement of elements reducing only by a small factor as the network increases.

Verification of the Methods

In this section, a solution to address the problems of weighted load balancing in a DHT should be specified and further validated. Table 4.2 shows a matrix of all presented solutions together with the supported features and introduced overhead. This makes clear that none of the mentioned solutions is able to address all problems a good load balancing scheme must solve.

	Virtual Nodes	**Id Selection**	Y_0	**Power of two**	**TD**	**HRW**
Fair Balancing	✓	✗	✓	✗	✗	✓
Overshoot	✗	✗	✗	✗	✓	✓
Consistent	✓	✓	✓	✓	✓	✗
Comm. Overhead	$\mathcal{O}(2v\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(c_v\log(n))$	$\mathcal{O}(n_h)$	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$
Churn Overhead	none	$\mathcal{O}(\log\log(n))$	none	none	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Memory Overhead	$\mathcal{O}(2v\log(n))$	none	$\mathcal{O}(2\log^2(n))$	$\mathcal{O}(n)$	none	$\mathcal{O}(\log(n))$

Table 4.2: Matrix of all possible solutions for distributed load balancing in a DHT with supported features and introduced overhead

Virtual node based approaches suffer from the problem of increased communication overhead. As described in the last section using Chord with $\log(n)$ virtual nodes increases the state keeping communication of every node to $\mathcal{O}(2\log^2(n))$. In addition, using virtual nodes does not improve the underutilization of nodes with a small share of the id space and only focusing on improving the maximum share a node could get. A variation of virtual nodes is used in the Y_0 DHT. In Y_0 there is no constant amount of virtual nodes enforced, but instead, the number of virtual nodes per real node is depending on the weight contribution of a real node. Through using this approach virtual nodes enable a DHT to use methods focusing on the even distribution of elements. Since most methods to balance load in a DHT focus on an even distribution of elements this is easier to achieve than a weighted distribution. With a combination of virtual nodes and methods for even distribution, it is possible to achieve a good weighted balance of all elements in a Chord ring. Furthermore, the communication overhead in this approach is not a fixed constant. Instead, a node with the minimum contribution weight only uses a single virtual node where a node with a factor c of the minimum contribution weight uses c virtual nodes. In a real-world example the contribution weight most likely also correlates with the computational power of a node making the overhead in communication and state keeping bearable.

An arbitrary id selection without any further improvements is contra productive in a heterogeneous network. Since it provides a good guarantee on even load balancing every node with a capacity above or below the mean id share is non-optimal utilized. The thermal dissipation approach suffers from the same problem but instead of relying on large data movements upon node arrival it only needs to delete a certain number of keys in the case of an interval split. In addition, the thermal dissipation approach also focuses on the problem of underutilized nodes by moving nodes with low utilization to intervals with a high load.

Implementing Y_0 in contrast should be beneficial for heterogeneous networks. It promises not only a good load balancing onto nodes with different capacities it also introduce a routing scheme that is aware of different node capacities and prefer larger nodes in proportion to their storage capacity. The Problem with using Y_0 is the necessity for estimating the number of nodes in the network and their average storage capacity. Furthermore, Y_0 discards nodes which have a storage capacity below the average capacity.

The power of two choices method provides additional load balancing properties but increases the amount of memory a single device must use by a dynamic factor. As discussed above we need to store pointers to the location of the data on $h - 1$ nodes when using a number of h hash functions. Since the target environment focuses on constrained devices a method which uses a constant amount of memory is preferable. In contrast, using multiple hash functions to store the load on multiple targets simultaneously can be beneficial for improved load balance and lookup times.

HRW based approaches are promising in terms of weighted load balancing but lack an important feature. It is not possible anymore to get a consistent location of a data item without knowing the whole topology of the network. Using a HRW based solution in a P2P network introduce a limit to the possible size of the network and prevents the network to benefit from its distributed nature.

Conclusion

As discussed above a combination of the provided methods need to be used to solve the load balancing problem in a DHT. Using virtual nodes allows us to focus on even load balancing. For an even load balancing, it is important to use a method which does not underutilize nodes by a larger factor. The thermal dissipation approach or the CAN distributed hash tables are good candidates for providing a near optimal load balancing. Both methods rely on intervals which get split into two halves if nodes join the intervals.

4.2.4 Data Availability

Using a DHT with consistent hashing as described above let arise the question of how to handle events where nodes fail unexpectedly. With the original notion of consistent hashing, a data item is only placed on a single node in the network and thus gets lost as soon as this node fails. Multiple solutions for data availability in a DHT had been proposed and should be explained in this chapter. A good data availability functionality should provide resistance against multiple failing nodes as well as a self-repairing property to restore full data redundancy when failing nodes rejoin the network or get replaced by new joining nodes.

RAID & Erasure Codes

Storing redundant data can be considered as a trade-off between the quality of error correction and the quantity of the stored data. An early approach for redundant data storage is RAID. RAID was designed to bundle multiple hard disk drives into a single logical disk. To provide a certain resistance against disk failures multiple so-called RAID levels are defined to reflect the mentioned trade-off [PGK88]. The first level RAID-0 provide no error correction since data is only striped between multiple disks. If a single disk failed the whole virtual disk is considered as erroneous. RAID-1 provides a very strong resilience against failing disk with high storage overhead. Data is written in a mirrored fashion over multiple disks. This notion doubles the storage overhead while providing resilience against one out of two failing devices. RAID-5 and RAID-6 use a block-level striping with parity bits to distributed the data on multiple disks. Both methods provide resilience against one or respective two failing devices with only an overhead of one or two disks which are needed to store the parity data. In this scheme, a failing disk can be reconstructed from the remaining disk and the parity information. The parity disk can be reconstructed out of the remaining data disks.

A generalization of redundant data storage as used in RAID is called erasure coding. Erasure codes are error correction codes which rather focus on data erasures than on data modification issues. An erasure event is defined as a complete loss of a single data block as in a node failure described above. An erasure code can transform a message of k bits into a longer message of n bits. The original message can be reconstructed from a subset of the longer message with n bits. The code rate is defined as $r = \frac{k}{n}$ and defines the ratio of the size of the original message to the size of the message with redundant information. As r grows the storage overhead gets smaller with the cost of less toleration against failing devices. All RAID level for data redundancy described above can be simulated by the usage of Reed-Solomon-Codes (RS-Codes). Reed-Solomon-Codes can be used to recode a data block of size n into s smaller data blocks of size $\frac{m}{n}$ such that $m < n$. The original file can be reconstructed from any m of the resulting s blocks [Pre89]. RAID-1 with two disks can be described as a RS-Code with $m = 1 \wedge s = 2$, RAID-5 with three disks as $m = 2 \wedge s = 3$ and RAID-6 with four disks as $m = 2 \wedge s = 4$. RS-Codes are so-called Maximum Distance Separable Codes (MDS-Codes) and therefore are able to provide an optimal storage efficiency for a given quality of reliability.

Problems with RS-Codes

While providing an optimal storage efficiency RS-codes suffers from inefficiency when a full data redundancy needs to be repaired. When a failed node gets replaced by a new node the new node needs to restore the original data block from the old node by contacting at least m nodes to restore the original data and recompute the missing block of the RS-coding by recode the information. This results in a large overhead on data transmissions where the stored data block is much smaller than the transferred data. In general, we need to transfer m times more data than we actually need to store on the new node. To improve this value different erasure codes such as regenerating or locally recoverable codes can be used [Kum17]. In [SAP+13] the authors introducing the new family of erasure codes called locally recoverable codes which can improve the repair and bandwidth cost by a factor of two with only increasing the storage overhead by 14%.

In [DGW+10] the authors present an analysis of the repair inefficiency of RS-Codes and introduce the notion of regenerating codes. Regenerating codes can improve repair bandwidth costs up to 50%.

While RS-Codes are able to encode with arbitrary m and s this property comes with the cost of expensive encode and decode operations. Classical RS-Codes provide a complexity of $\mathcal{O}(nm)$ for encoding and $\mathcal{O}(m^3)$ for decoding. This can be further improved by using derivations such as Cauchy instead of Vandermonde matrices as well as „Use neat projection to convert Galois Field multiplications into XORs“ [Pla05]. Different approaches like Parity-array codes and LDPC codes allow a lower encode/decode complexity but only work for a few fixed values of n and m.

Placement of the Data

Independent from the used technology all solution have in common that we need to distribute multiple chunks of data on different nodes. It is important for a single node to compute all locations of the chunks independent from other nodes in case of a node failure. Multiple solutions to this problem had been proposed and should be presented in the following sections.

As a simple straight forward solution, the DHash key-value store uses the existent data structure on a single Chord node of the next $\log(n)$ succeeding nodes to distribute the data in the network. Since this may introduce some unwanted redundancy in networks where the amount of nodes is smaller than $\log(n)$ it can make sense to limit this value to a fixed amount of nodes or omit nodes which are duplicated in the successorlist.

Another approach for distributing data in DHT like networks can be used by hashing the content of the data multiple times. This approach can work on any DHT that uses consistent hashing to place elements in the network and do not rely on a specific data structure. Another benefit is the fact that this approach is not limited to a fixed amount of nodes as arbitrary hashes could be generated. As a downside, this approach may weaken the guarantees on data availability since a single node can be selected multiple times due to the random output of pseudo-random hash functions. Also, it can be quiet expensive to compute multiple hashes for a single data item on constrained hardware.

Another approach used by the already introduced thermal-dissipation inspired method for load balancing in DHT networks is worth to mention. In this approach, multiple nodes are placed in an interval of a fixed size f. This interval gets split into two even sized intervals if the nodes count in a single interval reaches a value of $2f$. The Authors suggest storing a data item which is covered by an interval on any present node in this interval. This allows distributing the data on a minimum of f nodes at a given time [RPW04]. As a downside, this couples the parameter f to the strength of the used data availability solution.

Conclusion

Further studies show that the benefit of using erasure codes over a classical replication based redundancy is heavily depending on multiple factors like the bandwidth of the nodes, the expected availability of the nodes and the available storage size [RL05]. Also, the used

erasure code is depending on whether a fast encoding and decoding or an arbitrary selection of the used parameters is needed. Since we want to store multiple data words on different devices it makes sense to implement the encoding and decoding of the data block we operate on in a modular fashion. This allows using the system depending on a specific use case.

4.2.5 Storage Size

Another requirement yet unsolved is the awareness of the overall storage size in the network on any given node. When using a homogenous network where every node has the same capacity the storage size can be derived by an estimation of the number of nodes n in the network. A problem with estimations as discussed in [Man04] is the fact that the number of nodes can be under- or overestimated by a constant factor s. An overestimation of the capacity can result in a scenario where data can not be stored in the network despite the false fact that there is free capacity left. This can be resolved by reserving a factor of s capacity on every node with the downside of wasting as much as $2s$ units of capacity in the worst case of an underestimation of s. A better solution would be an estimation which guarantees to only underestimate the correct value of n or a solution which guarantees a correct computation of the overall storage size. A naive approach to get a correct storage size could just communicate the size of every node to its direct successor alongside with the information from the last node. This would result in exact storage information at the last node in the network where it can be broadcasted to or queried by other nodes. A clear downside of this approach is the linear complexity to update the information in case of a node join or failure. It would take a worst of $\mathcal{O}(n)$ steps to update the overall capacity on the last node where n is the number of nodes in the network.

Different research topics focusing on data aggregation in distributed P2P networks. Common practices differ mainly in terms of accuracy, communication efficiency, supported networks layouts and support for complex aggregation functions. As already stated a high accuracy would be beneficial, as well as a communication efficiency lower than the stated $\mathcal{O}(n)$ of the naive approach. For the communication of parameters as available storage size in a distributed environment, a simple decomposable aggregation function like sum or count is sufficient. The benefit of decomposable functions is their property that the result of the function can be computed as the result of any partition of the input data set and thus make them highly distributable [JBA15]. In the following section, the focus resides on the optimization of the communication efficiency while using an algorithm which provides perfect accurate aggregations.

For structured P2P networks, many approaches try to take advantage of the existing overlay network such as neighbor knowledge in distributed hash tables to provide better guarantees on communication efficiency. One example is to use additional sets of overlay links to build a tree-like structure in the existing overlay. Using the properties of the emerging tree updates of the parameters can be communicated in $\mathcal{O}(h)$ where h is the size of the tree. Using a perfect balances binary tree-like structure would result in $h = \log(N)$. Li et al. propose an algorithm to build such a tree in a bottom-up approach. In this approach, every node in a distributed hash tables uses a defined parent function P^i to calculate a set of possible parents. After a parent with sufficient free capacities to store an additional child is found, the node registers as a child on the parent node and a state keeping protocol takes place

to guarantee the further completeness of the resulting tree [LSL05]. They further provide a parent function which gives good guarantees on the distribution of the tree and proves that under the assumption of uniform node distribution that the expected hight of the tree is $\mathcal{O}(\log_k(N))$ where k is a fixed branching factor.

With a tree-based approach, the only node which is aware of the output of the aggregation function is the root. It is necessary to communicate this information along the network so that any node is aware of the network size at one point in time. As already described this could be achieved via a pull or push like approach. In a pull approach, every node would periodically contact the root node to gather the required information. This has the benefit that the communication of the parameters is done in a short amount of time. It shouldn't take longer than a single synchronization period to inform all nodes in the network. A clear downside of this approach is the high network bandwidth required as well as an additional load on the root node. Since it is not feasible to increase the load on a single participant in a P2P network a push approach is better suited. In a push approach, the information can either be flooded over the network using the already stated tree structure or the information can simply be broadcasted to all participants. While a typical IP broadcast is limited to the broadcast domain of the root node a thus may not reach all participants in the network we can use the tree structure to broadcast the information to all participants. A broadcast over the tree structure could also be described as a controlled flooding of the network with the guarantee that every participant is only traversed once. This approach needs a relatively high amount of bandwidth in contrast to a traditional IP broadcast since the number of messages must be doubled at every node in the tree but does not limit us to a broadcast domain which only includes a subset of nodes.

4.3 Summary

In Table 4.3 an overview of the requirements is provided and the modules which cover the appropriate requirements are shown. Most of the requirements are covered by the usage of a DHT. Since DHTs are designed to work on distributed systems and the resource usage of the implementation often depends on the chosen synchronization period the resource usage is expected to be rather low. As shown in Table 4.1 all examined DHT implementations provide specific guarantees on the network size and hops needed to reach another node in the network. Through using a structured network the requirements on guaranteed lookups, concurrency and locality are also satisfied. One problem arises from the need for homogenous nodes in the network. To achieve a balanced distribution amongst the members of the network an id selection protocol is chosen to provide even distribution. Using virtual nodes allows simulating a weighted distribution. To provide data availability on node failures a mirroring based approach is chosen. The communication of the storage size is done through the tree construction protocol as described in the last section. This protocol provides logarithmic time for data aggregation and the broadcast of the aggregated data along the network. To bootstrap nodes in the network, a combination of static configuration and multicast is used.

Requirement	**Covered by**
Computational Requirements	DHT
Path Length & Network Size	DHT
Guaranteed Lookups	DHT
Locality	DHT
Concurrency	DHT
Load Distribution	Id Selection
Data Availability	Mirroring
Raw Device Interface	RIOT
Storage Size	Tree
Bootstrap	Multicast

Table 4.3: List of all functional and non functional requirements and their design coverage

5 Implementation

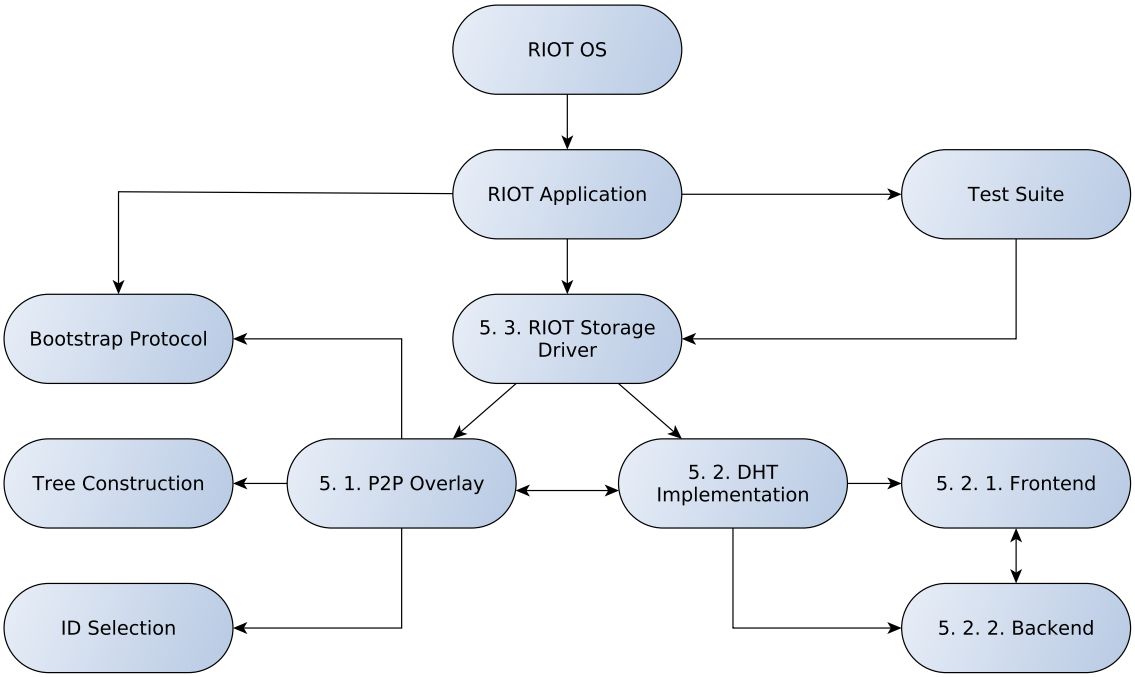


Figure 5.1: Implementation Module Overview

In this chapter, the details of the final implementation are discussed. Implementation details are chosen appropriately to reflect the results from the last chapter. An example implementation can be accessed online[1]. The diagram in Figure 5.1 shows an overview of the implemented modules with their relevant chapter marks and dependencies. As already stated the implementation works on top of the RIOT operating system for IoT and embedded devices. The RIOT implementation can be distinguished into two different parts. First, there is a RIOT storage driver which uses the DHT module to provide storage like functionality. The second part called the application can make use of the storage driver and combine it with other modules like file system implementations. The application further makes use of a shell module and extends it to implement multiple test cases. The DHT implementation used by the storage driver also consists of different modules. These modules are the P2P overlay network and the DHT implementation on top of the overlay network.

[1] `https://github.com/crest42/RIOT`

5.1 Peer-to-Peer Overlay

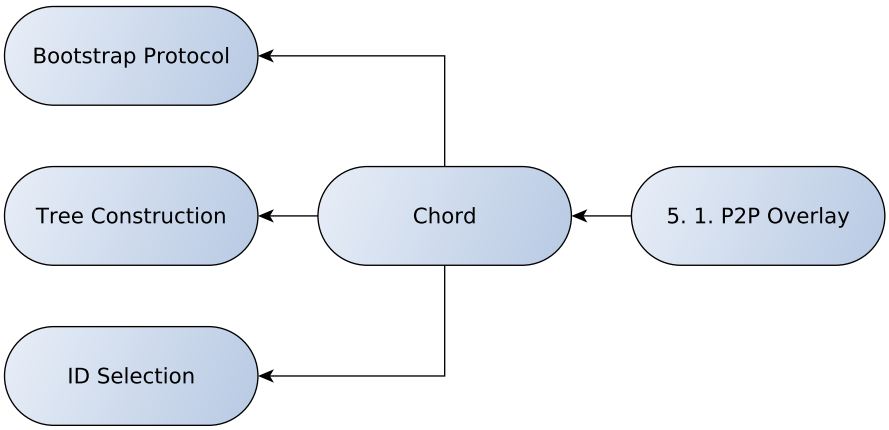


Figure 5.2: Chord Implementation Module Overview

As stated in the last chapter a structured P2P overlay network should be used to implement the distributed storage system. Since all analyzed overlay networks provide a protocol to organize participants in a defined topology, the selected DHT is implemented as a library with a defined interface to allow interchangeability with different implementations. In the last chapter, two common implementations of P2P overlays are further explained. Chord, which is a classical P2P overlay working on a ring structure and CAN, which implements the overlay on a d-dimensional hypercube. Both designs can be used to implement a data storage and mainly differs from their guarantees on load distribution, resistance against churn and routing complexity.

In terms of load distribution, the design of CAN can be easily extended to provide a constant factor load distribution of $\sigma \leq 4$. To get the same guarantee on a Chord network additional improvements on the existing design needs to be implemented. Creating the Chord ring by splitting intervals of nodes and interpreting nodes as leaves in a binary tree gives us the same guarantee as in CAN, with the cost of a single node that needs to change its location when any node leaves the network. This value can be further optimized to a smaller constant by increasing the number of relocation nodes on a churn event [Man04]. From a practical point of view, it is more feasible to use a Chord implementation for its easily understandable design. Furthermore, Chord is a well-established solution for distributed hash tables. Many research focuses on Chord when providing solutions to common problems in DHT like systems. A benefit from using CAN relies on the fact that it is easy to improve the routing performance by increasing the number of used dimensions. Since using more dimension in a CAN also implies storage overhead to store links to neighboring nodes and additional message for state keeping as well as the fact that the target environment consists of very constrained hardware it may provide no real-world advantage for using CAN over Chord. To back this assumption further experiments and analysis on resource usage and routing improvements using different P2P implementations need to be done.

5.1.1 Chord Implementation

```
1  #define CHORD_PERIODIC_SLEEP (2)
2  #define CHORD_PORT           (6667)
3  void *thread_periodic(void *data);
4  void *thread_wait_for_message(void *data);
```

Listing 5.1: DHT Main Threads

The Chord implementation is implemented as a generic library which provides two distinct threads. The interface of the functions reflects the POSIX pthread library[2]

The `thread_periodic` function implements Chords periodic protocol [SMK+01b]. On every run, the routine notifies the nodes current successor, requests the current successors predecessor to updates its own successor and updates an entry in the nodes fingertable. After each run, the threads set itself to sleep for a fixed amount of time defined by the constant value of `CHORD_PERIODIC_SLEEP`.

The `thread_wait_for_message` routine is used to listen on a well-known port defined by `CHORD_PORT` for incoming protocol messages and send an appropriate response. Calls to remote functions are implemented as RPCs in a defined message format. Every Chord message consists of a header section and a data section. The headers are constructed as a 4-byte message type which identifies the needed RPC call, 4 bytes for the calling node id, 4 bytes for the target node id and 4 bytes for the size of the payload. The remainder of the packet is used for arbitrary payload which gets passed to the according RPC function. The maximum message size is chosen to fit at least the size of a single header plus the size of complete successorlist. This is the size of a single node times the size of the successorlist which is $\log(N)$ with N being the size of the networks id space. A future implementation could make use of a RPC library such as rpclib[3] to reduce code complexity and to improve the stability of the overall software.

Load Balancing

As already discussed in the last chapter the guarantees on load balancing provided by Chord are not sufficient. Additional actions need to be taken to improve the guarantees up to a constant value. One approach described in the last chapter let the nodes chose an arbitrary id by interpreting the Chord network as a complete binary tree and split the biggest id space in a subtree of size $\log(n)$ into two halves [Man04]. The id of a node is chosen before a node joins the system by systematically searching for the biggest id space to split. The protocol defines a static function $\phi(l)$ which is used to control the starting level of the binary tree search and thus the quality of the load balancing.

[2] `http://man7.org/linux/man-pages/man3/pthread_create.3.html`
[3] `http://rpclib.net/`

Tree Construction Protocol

```
1  #define CHORD_TREE_BRANCH_FACTOR (2)
2  #define CHORD_TREE_ROOT          (CHORD_RING_SIZE/2)
3  struct child
4  {
5      nodeid_t parent;
6      nodeid_t self;
7      int i;
8      struct node parent_suc;
9      struct aggregate aggregation;
10 };
11 int register_child(struct child *c);
12 int refresh_parent(struct child *c);
```

Listing 5.2: Tree Construction Interface

To communicate the storage size in the Chord network the tree construction protocol as described in the last chapter is used. To implement this feature an additional call to register the child according to the protocol described in [LSL05] is implemented in Chords periodic run. The branching factor of the tree construction algorithm is defined to be 2 to construct a binary tree and distribute the additional load over many participants as possible. Furthermore, this allows us to communicate changes in an expected time of $\log(n)$ where n is the number of nodes currently in the network. After a child is registered at its parent it periodically sends a refresh message including the aggregated information of all its child nodes plus its own capacity information to its parent. The overall capacity information is returned in the acknowledgment message. The root node is defined to be the node which is the successor of `CHORD_RING_SIZE/2`. This node does not register itself as a child but starts sending the aggregated capacity information to both of its children. Using this algorithm any changes in the topology are communicated in $2 \log n$ steps to the whole network. This is due to $\log n$ step it needs in the worst case to update the overall aggregation up to the root node and $\log n$ steps to broadcast the information back to all child nodes.

Bootstrap Protocol

```
1  struct bootstrap_list {
2    uint32_t size;
3    uint32_t current;
4    struct in6_addr list[BSLIST_SIZE];
5  };
6
7  struct bootstrap_list bslist;
8
9  /* Bootstrap List Content Primitives */
10 int add_node_to_bslist(struct in6_addr* addr);
11 int add_node_to_bslist_str(const char* addr);
12 int fill_bslist_mcast(uint32_t max, uint32_t timeout);
13 int fill_bslist_mcast_ll(uint32_t max, uint32_t timeout);
14 /* Bootstrap List Content Primitives */
15
16
17 int
18 chord_start(void);
```

Listing 5.3: Bootstrap Interface

When a node wants to join into an existing Chord network it needs at least a single contact node. As already stated in the last chapter it should be possible to provide these nodes in multiple ways to allow heterogeneity in the used network topologies and in the used deployment scenarios.

To provide a generic way to set up possible contact nodes a data structure called `struct bootstrap_list` was introduced with a maximum size of `BSLIST_SIZE`. When a new Chord node gets started due to an invocation of `chord_start` the function either creates a new network if the global bootstrap list has no entries or it tries to join an existing network by iterating over the list and probing the nodes with a message of type `MSG_TYPE_PING`. As soon as a request is answered with a message of type `MSG_TYPE_PONG` the joining node acquires its id due to the load balancing protocol described in the last section. This continues until a node id is found and Chords regular maintenance protocol can take place.

The source code provided in Listing 5.3 shows multiple primitives that can be used to set up the bootstrap list. As a generic way to add any IPV6 address the functions of the `add_node_to_bslist` family can be invoked. Furthermore, it is possible to make use of IPv6 multicast either on a self-provided multicast address or using link-local multicast addresses to reach all nodes in the multicast group `ff02::1` due to an invocation of the `fill_bslist_mcast` function family. If a multicast-based approach is used a message of type `MSG_TYPE_PING` is sent to the respective multicast address and incoming answers are added to the bootstrap list until the maximum number provided as a function argument is reached, the bootstrap list is full or a timeout is reached.

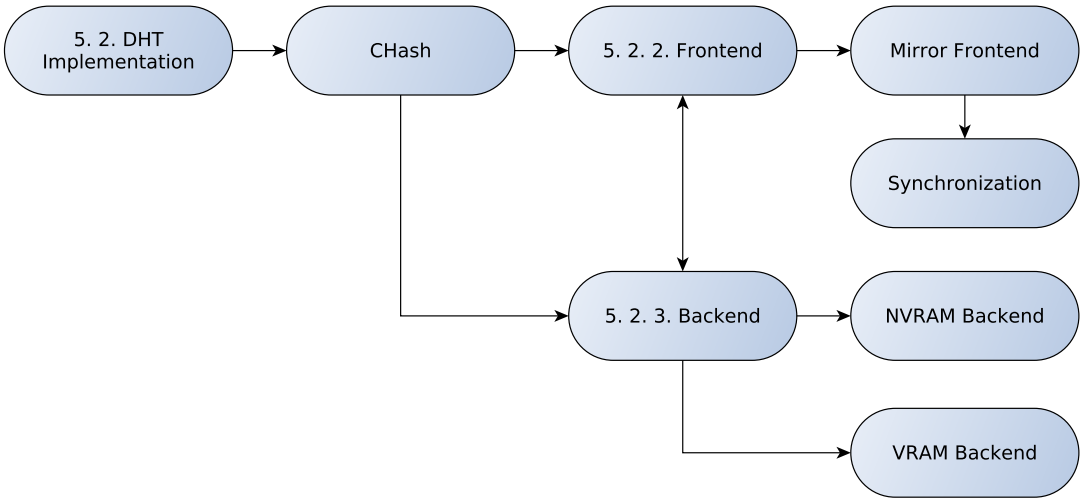


Figure 5.3: CHash Module Overview

5.2 Distributed Hash Table Implementation

In this section, the DHT implementation which is responsible for inserting and retrieving data, as well as distribution and load balancing of data elements, should be described. The DHT is implemented as a library called CHash in respect to the storage layer of the original Chord implementation DHash. The library itself consists of three major parts. The core part is responsible for providing raw network functionality to send and retrieve data to and from other nodes. This part is strongly coupled with the P2P overlay implementation and make use of function pointers to extend Chords base functionality.

The two other components are called the frontend and the backend of the DHT library. The frontend is responsible for looking up the location of data items as well as the data availability concept. The backend is responsible for storing the data in a defined data structure. Both modules are implemented as single compilation units to allow easy interchangeability.

5.2.1 Distributed Hash Table Interface

```
1    nodeid_t id      = find_responsible_node(nodeid_t id);
2    char *data       = get(nodeid_t dht_id):
3    nodeid_t dht_id  = put(char *data);
4    nodeid_t dht_id  = put(nodeid_t data_id,char *data);
```

Listing 5.4: DHT API

Any DHT provide a simple interface to write and retrieve data in the network. In DHTs where consistent hashing is used to compute the location of a data item, a single primitive to return the responsible node needs to be implemented. This primitive can be further used to store arbitrary data in the network using the put and get functions. In the original notion of DHTs with consistent hashing, the put function returns the id of the element as a hash of the content modulo the size of the identifier space. This is not always a practical solution

since one might want to calculate the id as a function on the metadata of a given data object instead of the whole data. In the case of a distributed block storage, the location of a data element is determined by its block number. A block number is a numeric increasing id which holds a fixed amount of data. To reflect this property an extension of the put function is implemented which allows providing the metadata as an additional parameter.

5.2.2 Frontend

```
1  int chash_frontend_put(uint32_t key_size, unsigned char* key,
2                         uint32_t offset, uint32_t data_size,
3                         unsigned char* data);
4
5  int chash_frontend_get(uint32_t key_size, unsigned char* key,
6                         uint32_t buf_size, unsigned char* buf);
7
8  int handle_sync(...);
9  int handle_sync_fetch(...);
10 int chash_frontend_periodic(void* data);
```

Listing 5.5: Frontend Interface

The frontend handles the logic of how and where the data should be stored in the network. Since it is important to provide an arbitrary level of data availability the frontend can create any number of data blocks and place them appropriately. This depends on whether a mirroring based approach or erasure coding is used to provide data availability. Furthermore, the frontend decides on which nodes the data blocks are stored in the network and how they can be retrieved. Listing 5.5 shows the frontend API. The placement and retrieving of data elements is handled in the implementations of `chash_frontend_put` and `chash_frontend_get`. The remaining functions can be used to provide synchronization features as they are invoked automatically in Chords periodic run and on events of `MSG_TYPE_SYNC` or `MSG_TYPE_SYNC_FETCH`.

In this implementation, a simple mirroring based approach is used for data availability. To write data in the network the hash of a blocks numerical identifier is generated and the data is stored on the direct successor of the hashes numerical representation and n of the successors subsequent successors. To provide arbitrary level of fault tolerance any value of n which follows $n \geq 0$ can be used, where $n = 0$ provides no fault tolerance. Alternative implementations can include erasure coding and different placement of data replicas. As an example data items can be placed into the ring by hashing their value multiple times. Since the id of the nodes in the network do not correlate to a geographic location (e.g. neighbors in the ring are not necessarily geographic neighbors) this would provide no advantage over using a linear placement. Furthermore, a simple algorithm to keep a constant level of replication using this linear approach is available and can be applied easily.

Synchronization

To keep a good quality of data availability a synchronization mechanism needs to be deployed to re-synchronize data onto responsible nodes if members of the network are affected

by an outage or a new node joins the system. The authors of the Chord DHT provide a synchronization protocol in their storage implementation DHash. They defined three different properties the system must fulfill for each block to be in an „ideal state“ to provide a given level of data availability. The first property is called multiplicity where any data element exists 14, 15 or 16 times in the network. The second property is called distinctness where „All fragments are distinct with high probability“ and the last property is called location where „Each of the 14 nodes succeeding the block's key store a fragment; the following two nodes optionally store a fragment; and no other nodes store fragment“ [Cat03]. The focus on distinctness and a certain number of fragments result from the usage of erasure coding where the full quality of data availability is determined by the usage of a certain number of distinct fragments. In this implementation, only a mirroring based approach is used which allows the implementation to ignore the second property. Also the number of available elements can be generalized as n elements where n, $n+1$ or $n+2$ elements exist in the network. Since the algorithm also works for mirroring based approaches and an erasure coding based data availability approach may be interesting for future implementations the provided algorithm is implemented in this thesis.

Two possible events can violate one or more of the defined properties. An event where a node joins the network can violate the third property and an event where a node leaves the network can violate the first property. To work around both events two synchronization algorithms take place, called the global- and the local maintenance protocol. The global maintenance protocol pushes items onto a newly joined node and restores the first property whereas the local maintenance protocol handles node failures and restores the third property. Both protocols are able to run distributed on every node.

Global Maintenance Protocol

To provide data availability, data items are stored on the node responsible for the data as defined by consistent hashing and a defined number of x successors. Since all nodes in a Chord network already keep a list of $\log(n)$ successors in their successorlist we can take advantage of this list to detect a node join and the need for re-synchronization of data elements. If a node joins in between the direct successor and one of its x successor the original successor n_x is now the n_{x+1} successor. Every node periodically iterates over its database of keys and request the successorlist of the node responsible for every data element. If the node is now the n_{x+1} successor for any key according to the successorlist the host publishes the key ranges which are not in its scope anymore to the x responsible successors. If any of the successors require any key it simply requests that missing keys. Afterward, the key is deleted from the database of the publishing node.

This algorithm ensures that the first and the third property are not violated at any time. After a key is synchronized there are at most $x + 1$ occurrences of this keys in the network. Since the key gets deleted directly after it is pushed onto another node and before an additional node can request the key we ensure a number of x or $x+1$ keys at any time.

Local Maintenance Protocol

To handle failing nodes accordingly an additional protocol needs to take place to re-sync data elements to nodes newly responsible for a given data item after a node has failed. If any node which holds a data element fails the first property is violated since $x - 1$ elements now exist in the network. To restore a number of x elements, every node in the network periodically sends a list of all keys for which they are directly responsible to all of its x successor according to its successorlist. If a node recognizes it is missing one or more of the keys it needs to request them by sending an answer with the needed key range. The node holding the data now pushes the needed keys onto the requesting node. This restores property one since there are now x elements of a given data item in the ring. This can be also done with a pull-based approach where data elements are not pushed onto missing nodes but rather nodes fetching missing data elements from the network. This is even more interesting in networks using erasure coding instead of replication. In approaches used by certain types of erasure coding, a node needs to fetch a certain number of distinct elements to reconstruct another distinct fragment. Due to a pull-based approach, the load for reconstructing an element can be shifted to the newly joined node and divided amongst the nodes which already hold the data.

5.2.3 Backend

```
1      int chash_backend_put(struct item *i, unsigned char *data);
2      int chash_backend_get(unsigned char *hash, nodeid_t *id,
3                            uint32_t *size);
```

Listing 5.6: Backend Interface

The backend is responsible for handling the local data storage. This depends in which type of memory the data needs to be stored and on the used data structures. As shown in Listing 5.6 any implementation needs to provide at least two functions. One which writes a data block into the storage backend and one which can retrieve any data element from the used structure.

In this implementation two backend implementations are available. A generic volatile memory storage using a linked list and heap memory to keep track of the data items and another implementation which stores data using RIOTs storage driver interface to use any storage backend RIOT provides. Since the complexity to search for existing and to add new data elements in this data structures is $\mathcal{O}(n)$ an additional backend using a more performance concerned implementation would be desirable and is considered as future work.

Volatile Storage Backend

```
1 struct key
2 {
3   uint32_t block;           /*!< Numerical Block id of the key.*/
4   uint32_t size;            /*!< Size of the data. */
5   unsigned char hash[20];   /*!< Hash of the block number. */
6   unsigned char *data;
7   struct key *next;
8 };
```

Listing 5.7: Data Structure for Storing Blocks of Data

The volatile storage backend makes use of heap memory to build up a linked list of data elements stored as a `struct key` as shown in Listing 5.7. The `unsigned char *data` element points to the heap memory holding the actual data. The hash of the numerical block id is stored in the according member of the struct.

Non-Volatile Storage Backend

```
1 typedef struct {
2     const mtd_desc_t *driver;  /**< MTD driver */
3     uint32_t sector_count;     /**< Number of sector in the MTD
          */
4     uint32_t pages_per_sector; /**< Number of pages by sector */
5     uint32_t page_size;        /**< Size of the pages in the MTD
          */
6 } mtd_dev_t;
```

Listing 5.8: Interface for a MTD Device

The non-volatile storage backend for RIOT Memory Technology Devices (MTDs) expects a pointer to a `mtd_dev_t` provided to the `backend_init` function. The MTD device is initialized using the function pointer in the `struct mtd_desc_t` and the according read, write and erase functions are used to store and retrieve data. The key structure as shown in Listing 5.7 is stored on the MTD device in addition to the data blocks. The pointers to the next element and the data pointer are used to point into a region of the MTD device and are relative to the start address.

5.3 RIOT Storage Driver

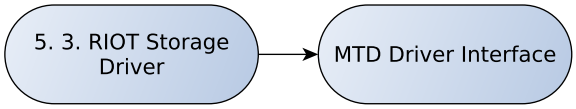


Figure 5.4: RIOT Storage Module Overview

As already stated, the implementation should act as a storage device using the RIOT storage API. This allows the driver to be used just like any other storage backend, such as hard disk and flash memory. In this section, a short introduction in the RIOT storage API should be provided and furthermore, a short overview on how the storage API is mapped onto the storage network should be given.

5.3.1 RIOT Storage API

```
1   int mtd_init (mtd_dev_t *mtd)
2
3   int mtd_read (mtd_dev_t *mtd, void *dest,
4                 uint32_t addr, uint32_t count)
5
6   int mtd_write (mtd_dev_t *mtd, const void *src,
7                  uint32_t addr, uint32_t count)
8
9   int mtd_erase (mtd_dev_t *mtd, uint32_t addr, uint32_t count)
10
11  int mtd_power (mtd_dev_t *mtd, enum mtd_power_state power)
```

Listing 5.9: MTD Driver Interface

The RIOT storage device driver API[4] is divided onto multiple drivers for different use cases. The Disk IO Driver is used to reflect properties of disk drive like CD, DVD or Blueray devices with interchangeable storage media. The Non Volatile Random Access Memory (NVRAM) driver aims for non-volatile memory devices which do not need block wise erase of data elements. Furthermore, multiple specific purpose drivers are available such as a driver for access onto serial NOR flash or a driver for access onto flash cards attached via a Serial Peripheral Interface (SPI). In this thesis, we use the MTD interface. This is due to the fact that it is a generic interface for storing and retrieving data which makes no assumptions on a given hardware or storage backend. Furthermore, it is already used in the RIOT virtual file system layer vfs and thus allows us to use the resulting driver directly to flash a file system onto it.

In Listing 5.9 the interface of the MTD driver is displayed. The `mtd_init` function is used to initialize the Chord network by creating a new network or join into an existing network. The `mtd_read` and `mtd_write` functions translate a given address and amount of data onto one or multiple blocks and fetches or inserts them into the network using the put and get functionality described in Listing 5.4.

[4] `https://riot-os.org/api/group__drivers__storage.html`

mtd_write

The `mtd_write` function needs to translate the provided address to a block number. A fixed definition of the size of a single block is provided via `mtd->page_size` and thus can be computed via integer division of `addr/mtd->page_size`. For every block an update message is send to all responsible nodes according to the frontend implementation which includes the computed block numbers, the actual data, the size of the data and an offset into the first block of size `(addr % mtd->page_size)`.

mtd_read

The `mtd_init` function initialize the values The `mtd_read` function works similar to the `mtd_write` function. The block numbers are computed and fetched using the CHash frontend before they get copied onto the read buffer provided to the function.

5.3.2 Size of the Storage Device

```
1       int backend_periodic(void *data);
```

Listing 5.10: Backend Hook for Periodic Updates

The size of the storage device is stored in the global `struct mtd_dev_t` and needs to be updated when nodes join or leave the system. To provide such functionality and additional hook in the `chord_periodic` function is used. Since every node got the size of the network periodically broadcasted through the tree construction protocol described in the last section, this can be achieved by updating `mtd->sector_count` to an appropriate value.

in Listing 5.10 the interface for the backend hook is defined. A void pointer is used as an argument pointer since we need to invoke it with arbitrary data, depending on the backend implementation.

5.4 Verification of the Acceptance Criteria

In the third chapter, acceptance criteria were defined for each requirement which needed to be evaluated against the implementation defined in the current chapter. Any deviation from the defined criteria must be reasoned and further actions need to be determined.

5.4.1 Functional Requirements

1. **Raw Device Interface**
   Acceptance criteria: At least a command for read and write a single block of data must be defined. A more sophisticated implementation can also implement read and write operation for multiple blocks as well as additional SCSI commands. The user must at least be able to write a specific block of data and read it afterward.

   The backend implementation in conjunction with the RIOT storage device can be viewed as such an implementation. The Interface described in Listing 5.9 provides a

generic read and write implementation which acts as a translation unit from numerical addresses to block numbers. To write a single block of data an address which is a multiple of the block size needs to be provided and the length parameter needs to be any value smaller or equal the block size. Providing a larger value results in reading or writing of multiple blocks. The number of actually processed bytes is returned after the operation is done.

To verify the read and write functionality two different test cases are provided. First, a low-level test is used which writes a single block of data with random content and tries to read it and verify the content afterward. A more high-level test which also writes multiple blocks of data makes use of a simple file system implementation which is supported in RIOT called LittleFS. „LittleFS, the high-integrity embedded file system in Mbed OS is optimized to work with a limited amount of RAM and ROM. It avoids recursion, limits dynamic memory to configurable buffers and at no point stores an entire storage block in RAM[5]“. A LittleFS file system implementation is formatted onto the storage device and can be mounted on multiple devices afterward. The nodes can now write and read files created by other nodes in the network.

2. **Concurrency**
   Acceptance criteria: Multiple write accesses on one or more data blocks should be executed in their correct order and lead to correct data after the last write access. Multiple users must be able to write multiple blocks concurrent without getting inconsistent data.

   In the provided implementation the location of each data block is determined by its numerical increasing identifier. When writing or reading multiple blocks of data it is ensured that the operations are executed in order of their increasing id. When multiple nodes write onto the same block the write operations are executed in order of the incoming UDP messages. Furthermore, the location of a block is determined by hashing its numerical id. This leads to a single node responsible for a single block of data and allows atomic updates of a single data block.

3. **Storage Size**
   Acceptance criteria: Multiple join or leave operation should be communicated across the network and the network should converge to the correct storage size in a given upper bound of time. When a node joins or leaves the network any other node in the network must be able to update its information on the overall size in between a given period of time.

   The tree construction protocol described in a previous section is used to communicate the storage size in the network. Through the usage of the periodic hook the frontend implementation updates this values on every run of the `thread_periodic` implementation.

4. **Load Distribution**
   Acceptance criteria: The probability for every data item to get stored at a node n_i with positive weight w_i needs to be $\frac{w_i}{W}$ where $W = \sum w_i$.

[5] `https://os.mbed.com/blog/entry/littlefs-high-integrity-embedded-fs/`

The load balancing protocol described in the last chapter is used to achieve a constant factor load balancing on multiple devices with the same weight.

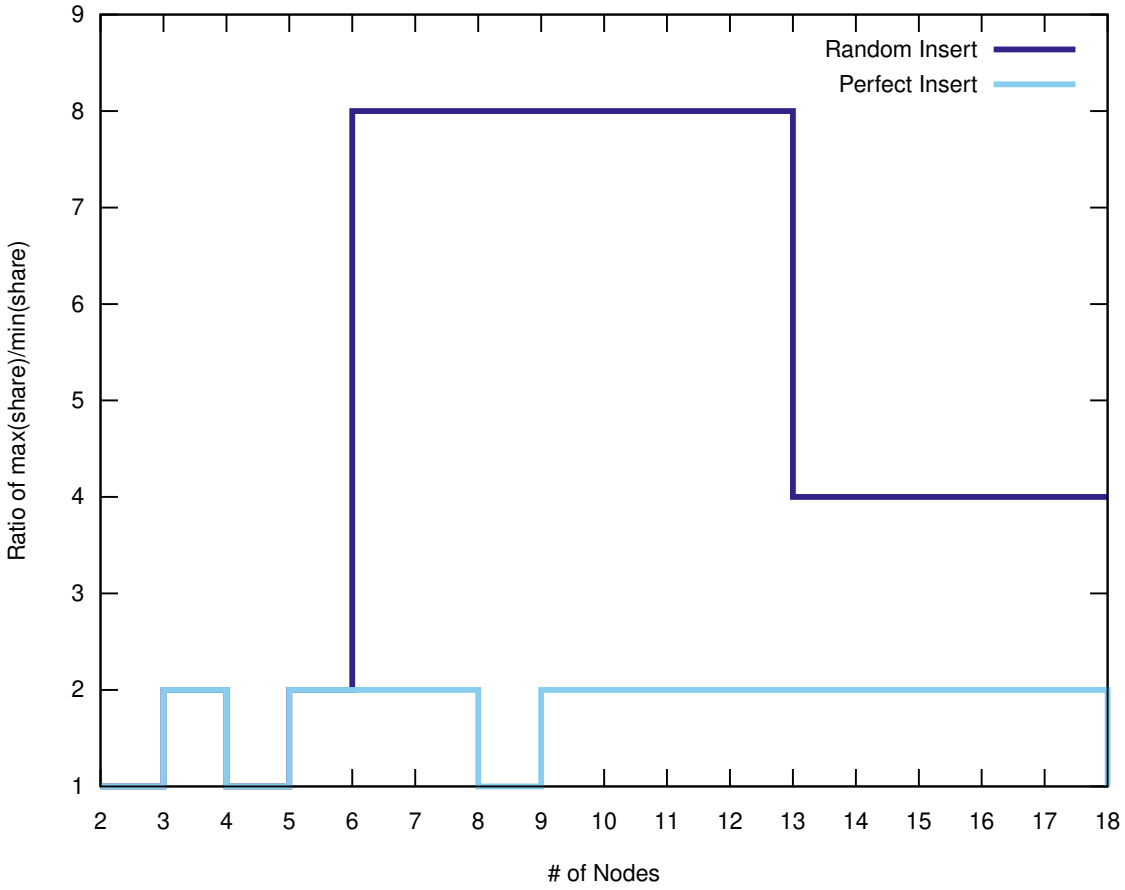


Figure 5.5: Ratio of min (share) and max (share) with random and perfect insertion of new nodes

Figure 5.5 shows the load balancing after the insertion of 18 nodes as implemented in the Chord library using a value of $\phi(l) = 0$ for random insertion and a value of $\phi(l) = \log(N)$ for perfect insertion.

Further studies needs focus on an implementation of correct handling of node failures. The implemented algorithm does not alter the node id after its initial join into the network. This assumes that the number of failing nodes are roughly equal to the number of nodes joining the network to keep a constant factor load balancing. Another part of the acceptance criteria not taken into account yet is the distribution over nodes with heterogeneous capacities. As described in the last chapter this should be achieved by creating multiple virtual nodes per real node which is not implemented yet.

5. **Data Availability and Consistency**
   Acceptance criteria: The design of the resulting network must guarantee to deal with a certain amount of churn. After multiple nodes fail a user must be able to look up the correct data in the network.

The maintenance protocol described as part of the frontend DHT implementation provides data availability up to a certain level. In the existing implementation, a mirroring based approach is used which tolerates the failure of $n - 1$ hosts where n is the number of replicas in the network. After a single synchronization period, the full data availability is restored.

6. **Locality**
   Acceptance criteria: The resulting system must be able to run in a local network without any uplink to other networks, as well as across network boundaries.

   As the existing test cases using the RIOT native implementation on a virtual tap network with link-local IPv6 addresses only, this property is fulfilled. To Bootstrap the device either static addresses or a link-local multicast address can be used. The multicast implementation and the Chord implementation are not limited to a link-local scope and can be used with global IPV6 prefixes to allow connectivity over network boundaries.

7. **Guaranteed Lookups**
   Acceptance criteria: After an insert operation into the storage gets acknowledged by the responsible storage location, any other member of the network must be able to query the data immediately.

   Since the storage of data relies on a request-response protocol implementation and the location of any data item is fixed due to consistent hashing this can be guaranteed. When a node wants to store blocks of data in the network it needs to query the storage location of the target devices and sends the data to the responsible nodes. After the data is stored on the target node a positive response is sent to the storing device. By the time the acknowledgment is sent the data is already stored in the network. Since the location of data is independent of the storage operation itself any device should be able to query the data at this point in time. Furthermore, the data gets stored on multiple devices to provide data availability. This also results in correct lookups if a device is affected by churn as long as not all devices which stores a data element are affected.

   A data lookup is considered a failed (e.g. no data in the network) when all nodes which are considered as storage nodes due to the data availability policy are not able to return any data.

5.4.2 Non-Functional Requirements

1. **Awareness on Computational Constraints**
   Acceptance criteria: An example target environment must be defined and the CPU utilization and the storage requirements of the resulting storage system need to be checked against the available resources of the target environment. The resulting system should not use more than a fraction of the available resources with at most of about 50 percent.

   The implementation was written with respect to computational constraints. The code base and the overhead in terms of external libraries and computational heavy algo-

rithms are kept as low as possible. In the next chapter further analysis are provided to validate this assumption.

2. **Path Length & Network Size**
   Acceptance criteria: An upper bound for the number of hops until a certain destination is reached must be provided

   Chord guarantees a path length of $\mathcal{O}\log(n)$ to find a given node by its id in the network. In cases of churn events, this can be as bad as $\mathcal{O}(n)$ until the fingertable of the remaining nodes are restored [SMLN+03]. Since the node lookups are implemented through an iterative style it is possible to count the number of steps needed to find any node at the requesting node. This value is printed to the debug output after every lookup. This implies that `DEBUG_ENABLED (1)` is defined.

3. **Performance Indicators**
   Acceptance criteria: On a given hardware the performance is bounded by the underlying storage technology. The throughput should be measured and any deviation from the theoretical maximum needs to be reasoned.

   The performance evaluations are done in the next chapter. An experiment to calculate the reached read and write performance is done and the reached values are compared to the possible maximums.

4. **Node Bootstrap**
   Acceptance criteria: A node should be able to join an existing network without any pre-configured entry node.

   Through using the generic bootstrap list interface as described in Listing 5.3 it is possible to use a generic interface to provide node addresses. The `fill_bslist_mcast` is already used in the example implementation to add new nodes into an existing network.

6 Performance Evaluation

Disclaimer: The following chapter is viewed as out of scope for the assessment of the bachelors thesis and is graded as a separate performance. As a part of the thesis and for the sake of completeness it is still included in the final submission.

In this chapter, a detailed analysis of the implementation is provided. Multiple performance evaluations are defined and performed accordingly. For practical reasons and due to the implementation is being aimed to work in constrained IoT devices running the RIOT operating system, two different environments are used for the verification. Tests that are able to run independently of the underlying runtime environment are executed on a GNU/Linux host system. Tests which are expected to be influenced by the runtime environment are executed on top of RIOT. To run on as many different environments as possible any RIOT application is compiled against a specific „board“ target. Usual a board is a specific hardware implementation of a Microcontroller Unit with the appropriate drivers for additional peripheral hardware and configurations concerning the capacities of the boards such as size parameters for static buffers and stack memory.

6.1 RIOT Environment

Figure 6.1 shows an overview of the different sections of the resulting Executable and Linking Format (ELF) binary when the system is compiled against a specific environment. To match the requirements of the different target environments two different binaries are built. The binary labeled with „Native“ is compiled with debugging flags enabled and thus with rather huge static buffers and additional overhead for debugging libraries. The other binary is built with respect to a minimal memory footprint and size. To chose an appropriate board with sufficient memory the minimal executable is used as a baseline for memory consumption. Below two common board environments are specified and used for further verification.

- **RIOT Native Device**
  This board configuration allows RIOT to run native on a operating system host. The resulting binary is provided as a ELF file and can be executed on a host running a GNU/Linux operating system or similar. To simulate MCU, RAM and storage capabilities, the according host hardware is used. To provide networking functionality a simple Linux tap network can be used by the binary.
- **Nucleo STM32 F144 Family**
  The Nucleo board family is a range of development boards providing an evaluation environments for STM32 MCUs. Nucleo boards are available in different configurations with respect to the used MCU, the amount of RAM and the amount of available flash memory. This board family was already used in past projects and has shown broad support by RIOT. Furthermore, the boards are available with an onboard Ethernet

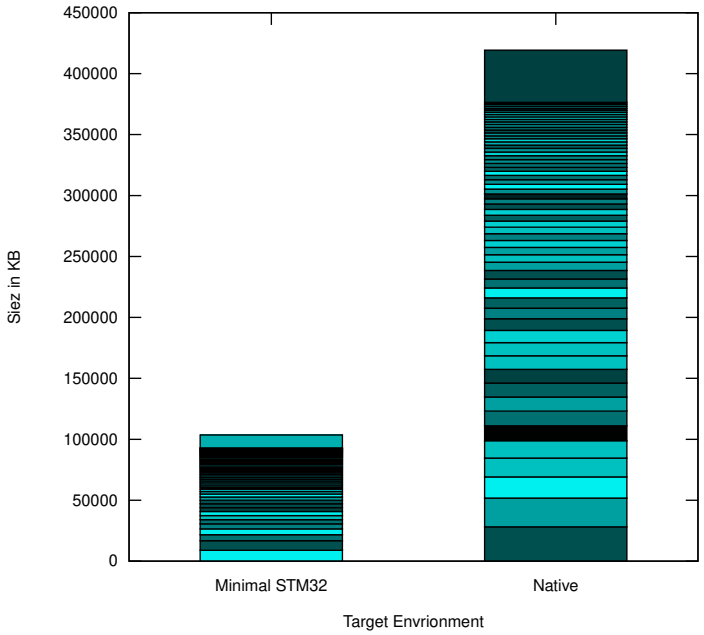


Figure 6.1: Size of the compiled ELF binaries for different target environments. The different colors refer to different ELF section names and are chosen such that sections with the same name resulting in the same color.

controller which makes them an attractive choice for further evaluations. Table 6.1 shows an overview of Nucleo boards with existing support in RIOT. Since the available MCU in a given family does not differ in their computing power the choice for a specific version of a target family is arbitrary and driven by the available storage capacities. Since the Nucleo-F767ZI also provides an onboard Ethernet controller it is used for the respective performance evaluations.

Board	MCU	RAM	Flash	MCU Hz
Nucleo-F767ZI	STM32F767ZIT6	512	2048	216 MHz
Nucleo-F746ZG	STM32F746ZGT6	320	1024	216 MHz
Nucleo-L496ZG	STM32L496ZGT6	320	1024	80 MHz
Nucleo-F413ZH	STM32F413ZHT6	320	512	80 MHz
Nucleo-F412ZG	STM32F412ZGT6	256	1024	80 MHz
Nucleo-F429ZI	STM32F429ZIT6	256	2048	80 MHz
Nucleo-F722ZE	STM32F722ZET6	256	512	216 MHz
Nucleo-F207ZG	STM32F207ZGT6	128	1024	120 MHz
Nucleo-F446ZE	STM32F446ZET6	128	512	80 MHz
Nucleo-F303ZE	STM32F303ZET6	64	512	72 MHz

Table 6.1: Overview on the Nucleo F144 Board Family.

6.2 GNU/Linux Environment

Multiple measurements are independent of RIOT and can be run on any given host system. Since running an ELF binary in a Linux/libc environment provides additional benefit in terms of available data points and the possibility to run large and automated setups an additional executable is built as a standalone ELF binary. Furthermore, a simple Perl script using the `CURSES::UI` module is used to provide an automated testing framework for spawning nodes and aggregation of relevant data points.

6.3 Evaluation

In this section, a model for functional and non-functional evaluations is provided and applied to the existing implementation. Results of the experiments are examined and are discussed further.

6.3.1 Evaluation Model

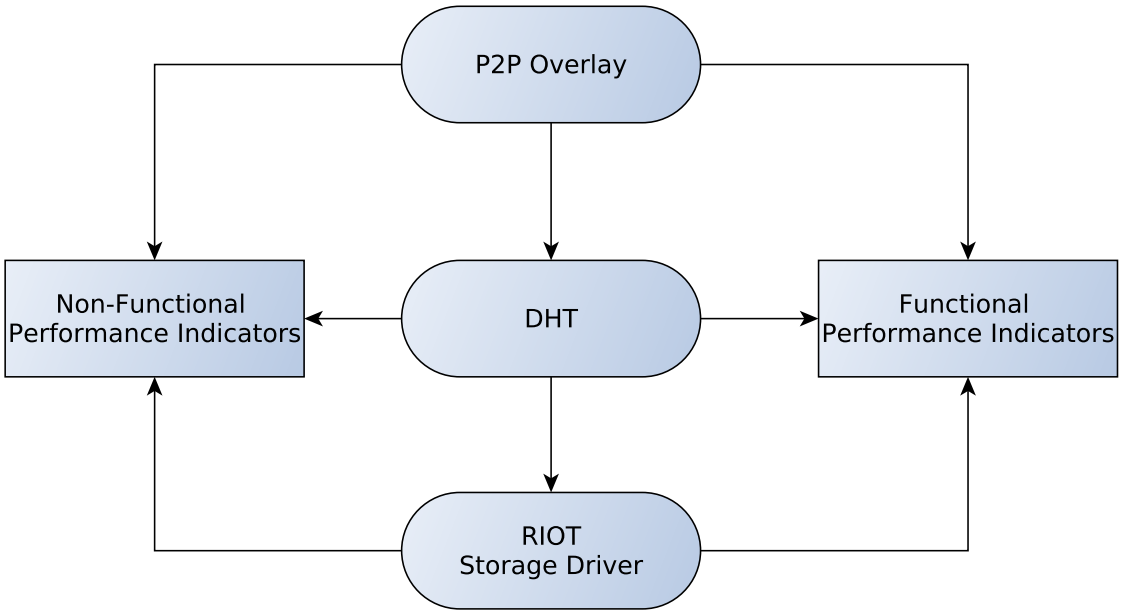


Figure 6.2: Evaluation Model

As described in the last chapter the storage software consists of three major layers which should be evaluated independently of each other. The first part is the P2P overlay network, the second part consists of the DHT implementation and the last part is the storage driver implementation in RIOT. The evaluations need to focus on functional and non-functional indicators. Functional indicators can be viewed as the users perspective on the system i.e. how well the system performs in terms of the service it provides. Non-functional performance indicators describe how the system behaves in terms of resource usage and raw performance such as network, MCU and memory usage.

6.3.2 Functional Performance Indicators

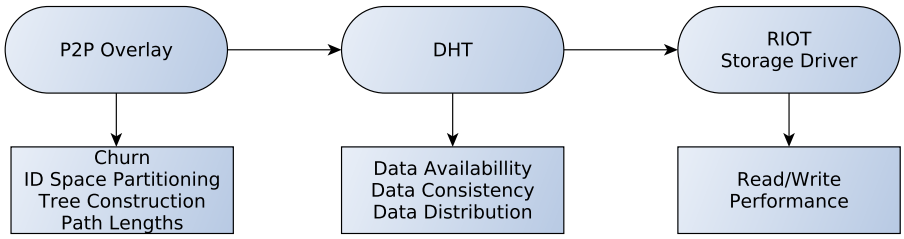


Figure 6.3: Functional Evaluation Model

Peer-to-Peer Overlay Network

As explained above the functional evaluation model should verify that the system behaves correctly in terms of the guarantees given by the implemented modules. As derived from the design of the system the P2P overlay network is responsible for providing four major functionalities:

1. **Resistance against Churn**
   The P2P protocol is responsible for constructing an overlay network which is able to tolerate churn up to a certain point. A single node should be able to recover unless at least one node in its successorlist is still reachable. To verify this property an evaluation also used in the original Chord implementation needs to be done. First, a network of n participants needs to be set up and then a fixed fraction of $p \in [0, 1]$ nodes should fail after the network is synchronized. The remaining fraction of $1 - p$ nodes should be still reachable. This should give confidence on the correctness of stabilization procedure. If the procedure is erroneous we expect only a subset of $(1 - p) \cdot n$ nodes to be reachable [SMLN+03].

2. **Id Space Partitioning**
   The id selection procedure defined in the design should give a certain guarantee on the quality of the id space partitioning. To verify this property a number of n nodes should be spawned and the value σ as the ratio of the maximum and the minimum of the id space of all nodes should be calculated. As described in the design this is expected to be $\sigma \leq 4$.

3. **Tree Construction**
   The tree construction protocol defined in the last chapter is responsible for the aggregation of available nodes capacities and for broadcasting the aggregated values in the network. As described above it should take a maximum $2\log(n)$ synchronization periods to update the overall size of the network on every node. To verify this property an increasing network of nodes should be created and the depth of the tree after each node spawn should be measured.

Distributed Hash Table

Furthermore, we need to verify the implementation of the DHT functionality implemented in CHash as described in the last chapter. The DHT implementation is responsible for providing data availability, data consistency and data distribution in the network.

1. **Data Availability**
   This property is strongly coupled with the correct functionality of the P2P overlay implementation. After a certain amount of nodes had failed every member in the DHT network should be still able to request data items stored on the erroneous node. To verify this property a network of n nodes need to be spawned and a number of data items $d >> n$ should be inserted in the network. Afterward, a fraction of the nodes should be removed from the networks. We expect that all data items are still available in the network up to a certain fraction of failing nodes. Furthermore, it is expected to reach an increased data availability if the number of replications of a single data element is increased.

2. **Data Consistency**
   This property defines the correctness of the data even after node failures and can be verified together with the data availability property. To ensure this property the hash of the numerical block identifier should be additionally stored in every data block inserted into the network. After the node failure scenario described above, it is necessary to test if the block yields the correct content in addition to a successful lookup of the data.

3. **Data distribution**
   Due to the nature of consistent hashing where a hash function is used that guarantees an even distribution of its outputs, we expect an even distribution of data items in the network. To verify this property a network of n nodes should be spawned and $d >> n$ data items should be inserted in the network. Afterward, we should be able to verify the distribution of the data items in the network using statistical tests.

RIOT Storage Driver

Since the RIOT storage driver only acts as a wrapper for the DHT implementation we can safely assume that the experiments from the DHT implementation also holds for the RIOT storage driver. An important additional performance evaluation should target the raw read and write performance of the final implementation. An experiment on a RIOT board as explained in the model needs to be performed and evaluated.

6.3.3 Non-Functional Performance Indicators

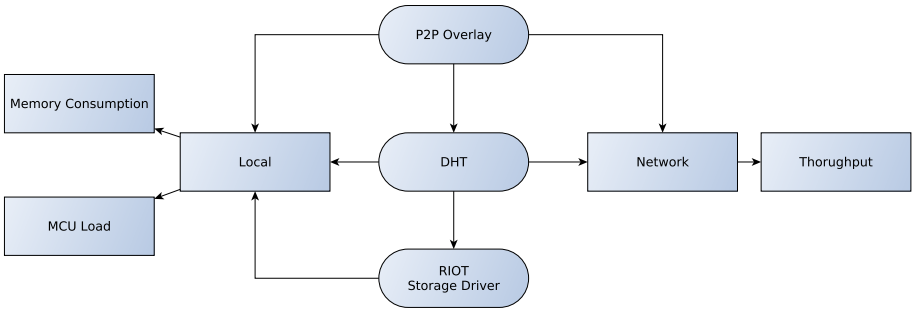


Figure 6.4: Non-Functional Evaluation Model

As shown in Figure 6.4 the used non-functional performance indicators consists of two parts. A „local“ part where the resource usage on a single node needs to be evaluated and a „network“ part where the networking overhead of the different modules needs to be evaluated. This is done within the target environment as described at the beginning of the chapter and should have a special focus on resource usage in a constrained environment. As it is important to attribute the resource usage of the different modules on different functionalities of the modules this test should be implemented in a step-by-step fashion starting with a minimal implementation and enabling single features as described in the implementation at a time to allow an attribution onto different sub-modules.

Peer-to-Peer Overlay Network

The Chord implementation consists of three parts. The initial implementation of the base protocol as described by Stoica et al., the tree construction protocol and the id selection protocol. The last two parts should be evaluated independently of each other. Since both parts depend on a functional overlay network it is not possible to verify the modules independently of the base Chord implementation.

Distributed Hash Table

The base implementation of the DHT protocol adds additional network overhead for the synchronization protocol described in the last chapter. This overhead should depend on the used data availability policy and increase with an increasing number of replications. It is important to measure the effect of the synchronization protocol on the generated load in the network.

RIOT Storage Driver

Since the storage driver itself adds no additional network functionality it is possible to avoid additional network measurements. Anyway, it is important to measure memory consumption and the MCU load on real-world hardware as defined at the beginning of this chapter.

6.4 Results

In this section, the experiments derived from the above structure are explained further and the results are presented accordingly.

6.4.1 Peer-to-Peer Overlay Network

The P2P overlay network is responsible for the coordination and the alignment of subscribers in a distributed ring topology. It can be viewed as the fundament of the distributed block storage and takes care of leaving and newly joined nodes. As shown in Figure 5.2 from the last chapter the Chord implementation consists of three modules. The base Chord implementation, the tree construction protocol and the id selection algorithms. Wherever possible these modules are verified independently. Since the performance of the implementation is independent of a specific target environment all evaluations are done with a static binary on GNU/Linux.

Non-Functional Performance Indicators

As described in Figure 6.4 the non-functional indicators for the P2P overlay are the network throughput, MCU load and memory consumption. It is important to measure those values for all three modules of the overlay network as described above. To verify the impact of the different modules on those values we use the following experimental settings. 10^2 nodes are started simultaneously and after some synchronization period the overall runtime, the send and received bytes, as well as the CPU usage of the two distinct threads, are collected. This experiment is done four times. One time with only the base module enabled, one time with all modules enabled and two times with the base module and one of the extra modules. This should yield the additional performance needed by the extra module as well as any additional overhead if both modules are activated. The base module is used as a baseline for the overhead introduced by the additional modules. The modules are expected to introduce a small overhead as well as an even distribution of the load amongst the nodes due to the P2P nature of the system. The amount of 10^2 nodes are chosen due to the ring size of the network which is set to 2^{16} and thus needs at least 16 nodes to work as intended by the Chord implementation. To cover a realistic scenario the actual node count is chosen as a tradeoff between a multiple of 16 and a value which allows practicable tests within the semi-automated testing environment.

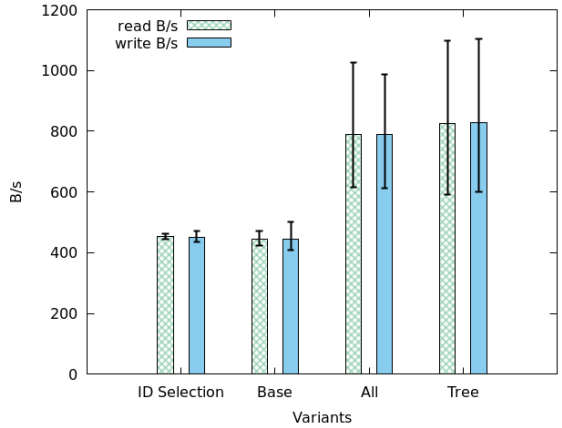
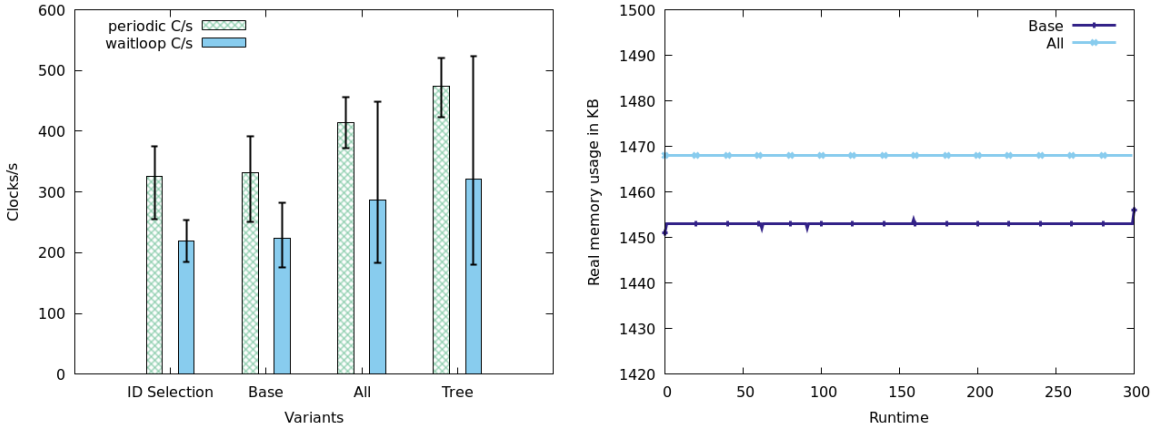


Figure 6.5: Average Read and Write B/s with different Modules enabled and 90% Percentiles

As shown in Figure 6.5 The id selection has little to none impact on the network traffic. Since the algorithm only takes place on node startups and due to the long duration of the experiment which results in a rather high sample size, the overhead introduced by these modules gets compensated over time. Another interesting property is the 90% percentiles of the base algorithm. As explained above we expect the load to be evenly distributed amongst the nodes.

Since the percentile is close to the mean value we expect this property to be fulfilled. Furthermore, we can see a rather big overhead introduced by the tree construction algorithm. Since the mean values of both experiments differ about a factor of two a further experiment is needed to determine the exact reasons for this overhead. The tree construction algorithm involves sending two messages with a size of 80 bytes on every synchronization period. Furthermore, these messages get answered with a message of 72 bytes each. This yields an expected overhead of 152 bytes on every synchronization period. Since a synchronization period takes two seconds we expect an overhead of 76 bytes per second. Therefore the remainder of the overhead is probably introduced by the node lookup and the redirection of full nodes to their neighboring nodes. To verify this hypothesis a further experiment is launched which the number of redirects and successor lookups per node. Due to this experiment, a node sends an average of 3 register messages and 3 successor lookups on each synchronization period dedicated to tree construction. This results in an expected average overhead of $(3 \cdot 76) + (\frac{3 \cdot 60}{2}) = 318$ bytes/s. Assuming the average value for sent bytes from the last experiment this yields an expected value of $452 + 318 = 770 b/s$ per node. This matches the average values of $788 b/s$ from the last experiment. This also explains the increased variance of the measured data. Nodes farther away from the root are more likely to send multiple register messages due to redirects as explained in the implementation chapter.



(a) CPU Usage

(b) Memory Usage

Figure 6.6: Average Memory Usage by node and Clock Cycles/s used by the two threads with different Modules enabled and with the 90% Percentiles

Figure 6.6a shows the average CPU usage by 10^2 nodes measured in total clocks and divided by the POSIX definition of `CLOCKS_PER_SEC` which is normalized to a constant value of 10^5. The usage is distinguished by the two main threads as described in the implementation. As expected the results are similar to the network usage and only increases by a small factor if the tree construction protocol takes place. Also, the increased variance in the wait loop can be explained due to the increased load on nodes near the root of the tree. The values of the relative CPU usage vary in between 0.1% and 0.5% of the total available `CLOCKS_PER_SEC`. Figure 6.6b shows the average memory usage of a single node over time. Since the implementation is aimed to use static memory the constant behavior of the measured data is expected. Activating all features adds little memory overhead due to additional buffers. The values for both graphs are collected with the `getrusage` POSIX extension implemented in libc.

Functional Performance Indicators

In this section, the functional performance indicators of the P2P Protocol should be measured. As explained in the model this includes the resistance against churn, the id space partitioning ratio and the depth of the tree resulting from the construction protocol. To verify this parameter multiple experiments as described in the model are done.

The first experiment launches 10^3 nodes with disabled replication and inserts 10^4 keys into the network. We expect every node to hold about 10 keys. After the keys are inserted a fraction of the node gets killed. Due to the even distribution of keys in the network, we expect about the same fraction of keys not be reachable anymore. Figure 6.7 reflects these expectations pretty well. The network could compensate an outage of about 70% of all nodes until the synchronization procedure failed to reconstruct a valid ring.

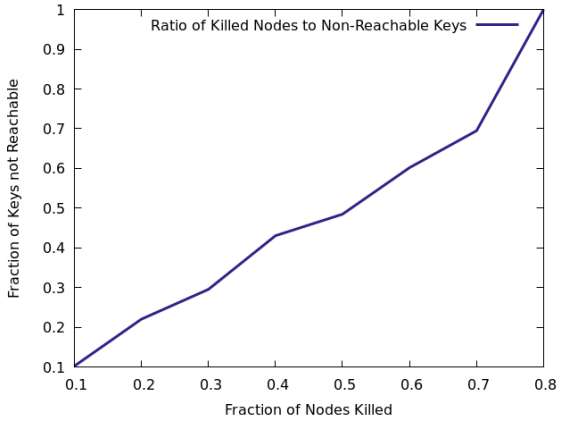


Figure 6.7: Ratio of Killed nodes to Non-reachable keys

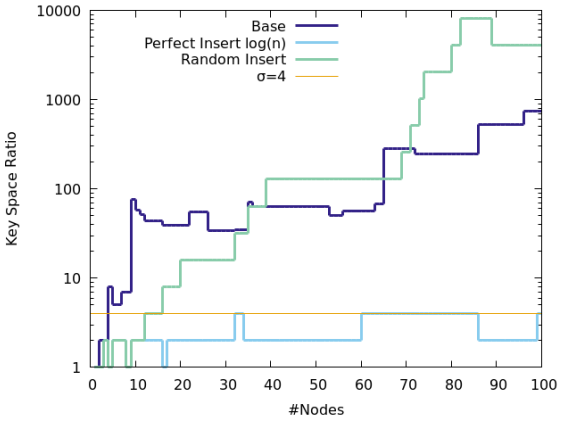


Figure 6.8: Id Space partitioning with multiple strategies using a logarithmic scale for the y axis

The next experiment is used to verify the guarantees given by the id space partitioning protocol. Figure 6.8 shows the ratio of the maximum id share to the minimum id share of all nodes for multiple id selection strategies. Base refers to a random id selection as described by the original Chord implementation. Perfect Insert $\log(n)$ is a insertion strategy, where a random node in the network is selected and the biggest id share in a vicinity of $\log(N)$ successor of this node is split into two halves, where N is the size of the Chord network. For random insert, a random node in the network is selected and its id space is split into two halves. It is easy to see that the implemented id selection scheme provides the given guarantees while both other schemes tend to a rapid grow towards an unfair partitioning.

In the last experiment, the guarantees on the tree construction algorithm are tested. Multiple nodes are spawned and after every additional node, the depth of the tree on every single node is collected. In Figure 6.9 the maximum depth of the tree in contrast to the number of nodes is shown. It is easy to see that the depth of the tree does not grow perfect logarithmic. This is due to the nature of the tree construction protocol which does not give any guarantees on a balanced tree. However, it is possible for the tree to experience a balancing which can result in a decreasing depth as seen in the graph. Furthermore, it is possible to see that the

depth tends to grow slower on increasing node count and does not exceed a level of 10. To calculate the depth of the tree the broadcast and aggregation algorithm provided by the tree construction protocol itself is used. Since every node in the experiment with exception of the root node reached a certain level $l > 1$ it is possible to conclude that every node takes place in the tree construction and gets a position in the tree. Furthermore, we can conclude that there are no cycles and moving nodes after a certain synchronization period. This is due to the fact that after every joining node the ring reached an equilibrium state where the depth information on the node is not affected by a change anymore.

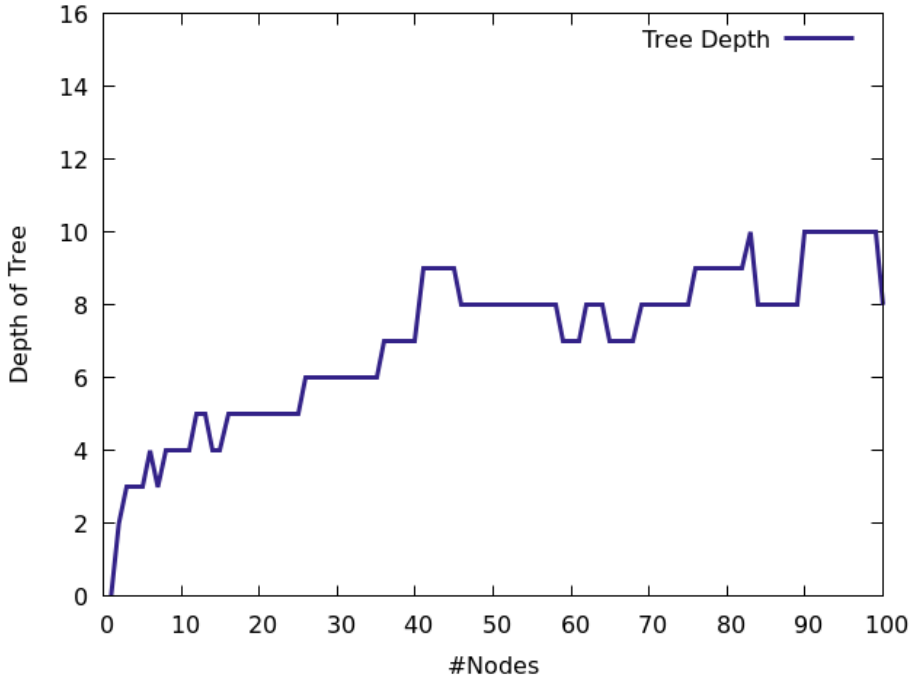


Figure 6.9: Depth of the resulting tree with increasing node count

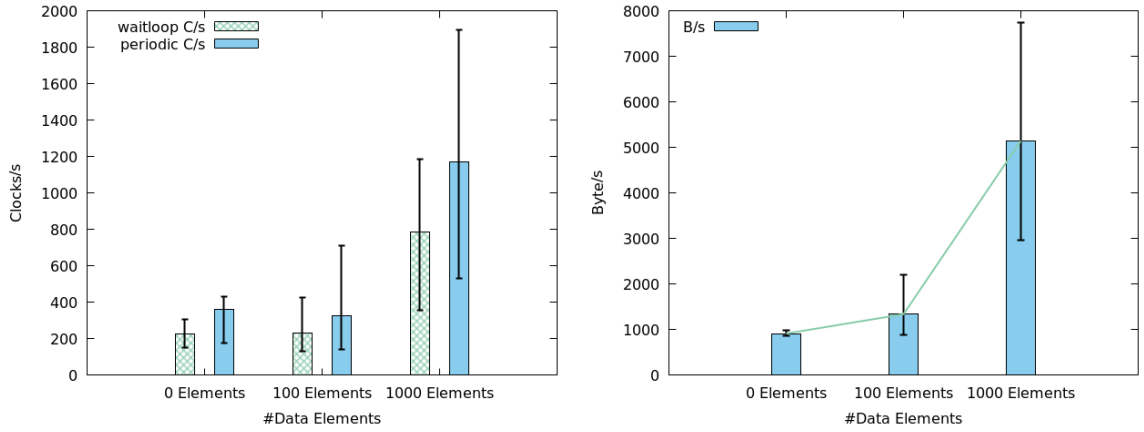
6.4.2 Distributed Hash Table Implementation

In this section, the results of the experiments related to the DHT Implementation are described and further explained. As in the last section, the environment for generating the data consists of a static binary on a GNU/Linux environment.

Non-Functional Performance Indicators

To verify the non-functional performance indicators we perform the experiments as already done for the P2P overlay network. In addition to the already explained experiments, we insert an increasing number of elements into the network. This should yield the overhead introduced by the data availability protocol. Furthermore, different replication policies should be used for a fixed number of elements. We expect the network overhead to increase linearly with the chosen replication level. To get a good sense of the introduced overhead from the state keeping protocol this experiment is designed such that after starting a fixed amount

of nodes and inserting a certain number of elements the statistics on the nodes are reset to zero. This is done because otherwise the network statistics would be dominated by the load for the initial storage of the elements. Due to this reset, a baseline value is included where no elements get stored into the network with a replication level of one. Figure 6.10a shows the CPU usage for different amount of stored data elements and Figure 6.10b shows the increases in network throughput with an increasing amount of elements. The CPU usage is computed per thread as CPU clocks/s through `getrusage` as described in the last section.



(a) CPU Usage

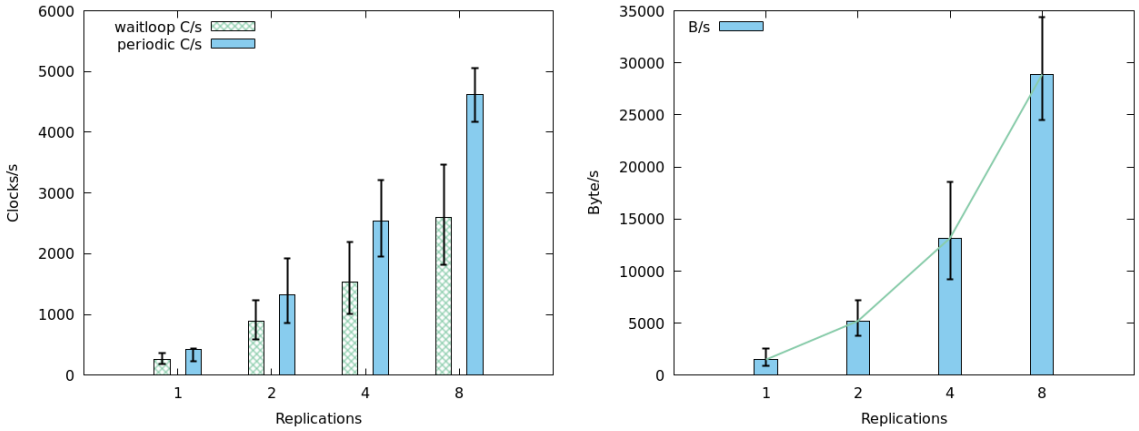
(b) Network Usage

Figure 6.10: Average network usage by node and clock cycles/s used by the two threads with different numbers of elements in the ring and the 99% percentiles

Figure 6.11a and Figure 6.11b shows the CPU and network utilization with an fixed size of 10^2 nodes and 10^3 elements in the ring. Instead of altering the number of nodes or elements the replication level is increased in each run. Since the algorithm for the replication of elements performs linear with the number of replications the results show a linear growth in the network usage as well as a near linear growth in the CPU usage.

Functional Performance Indicators

First, the data availability property in conjunction with the correctness of the data is verified. To implement this, a fixed number of 10^3 nodes are started and 10^4 data elements are inserted into the network. Afterward, an increasing number of nodes are deleted from the network and the fraction of available data item is measured. In every step, 10 nodes are deleted. In Figure 6.12 the fractions of available items for three different replication levels are displayed. Replication Level 1 means no replication where replication level 2 and 4 means that the data is additionally stored on one or three succeeding nodes. As expected the fraction of lost data without replication grows linearly with the fraction of failed nodes since every data item on every failing node is lost. The fraction of lost data decreases very fast as replication takes place. This is due to the fact that even if multiple neighboring nodes fail in one step only the number of data items stored on the first node in a row is lost. Since every node holds about 10 data items, we expect this number of data items to be lost if two neighboring nodes fail simultaneously. Since we kill a fixed number of nodes on each round the chance to kill two



(a) CPU Usage (b) Network Usage

Figure 6.11: Average network usage by node and clock cycles/s used by the two threads with increasing replication level and the according 99% percentiles

neighboring nodes increases over time as the population gets smaller. This also shows that the data distribution is working correctly.

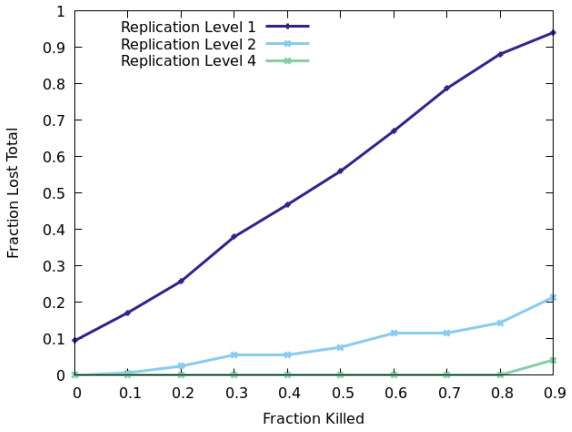


Figure 6.12: Fraction of Lost items in correlation to the fraction of lost nodes

To verify the data distribution further, we start the experiment as described in the model. Again a fixed number of 10^3 nodes are spawned and 10^4 data items are inserted into the network. After the data is inserted the number of data elements n_e for each node is collected. To make a statistical test on the collected data we first need to take the share of the id space of each node s_n into account. As an example, if a single node got two times of the id share of another node we expect two times more data elements on this node. To normalize the values we compute the expected fair share s_{fair}. If the nodes distribution would follow an uniform distribution we would expect an fair share of $s_{fair} = \frac{2^{16}}{10^3}$ with an id space of the size 2^{16}. The fraction of the fair share a node uses is then computed as $s_{frac} = \frac{s_n}{s_{fair}}$ and multiplied with with amount of elements to get the normalized amount of elements $s_n = s_e \cdot s_{frac}$. Furthermore, we compute the normalized expected number of elements for each node. Assuming a uniform distribution of the nodes we would expect $s_{uexp} = \frac{10^4}{10^3}$ elements per node. The get the normalized value in respect to the real id share we compute the real expected amount of values per node as $s_{exp} = s_{uexp} \cdot s_{frac}$. With this data we can perform a chi-squared goodness of fit test[1] on the calculated data, assuming a uniform distribution. To verify this we calculate the chi-squared test statistic using the observed and the expected values after the normalization and compare the result

[1] `https://www.openintro.org/stat/textbook.php?stat_book=os` p. 286

with the critical value from the chi-squared distribution using 98 degrees of freedom and a confidence level of 0.01. This results in a value of 73.4835 for the test statistic and a value of $\mathcal{X}_{[98,0.01]} = 133.47$. Since $73.4835 << 133.47$ we can assume strong confidence in the uniformity of the distribution. To get an additional visual representation Figure 6.13 shows a QQ-Plot of the observed normalized elements per node against an uniform distribution. Since the percentiles of the measured data are closely aligned on the hypothetical perfect distribution we gain additional confidence in the uniform distribution of the data.

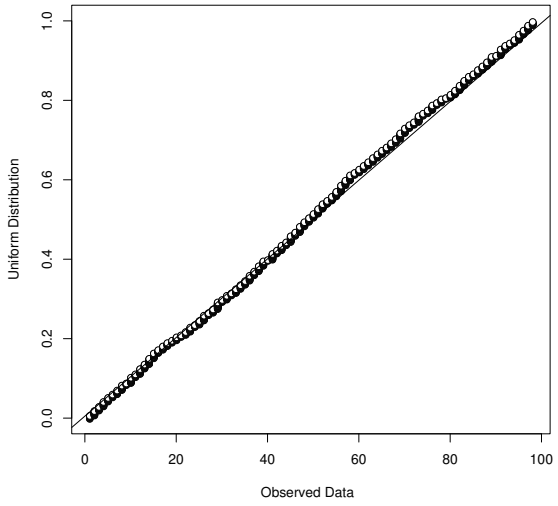


Figure 6.13: QQ-Plot of the data against a uniform distribution

6.4.3 RIOT Driver

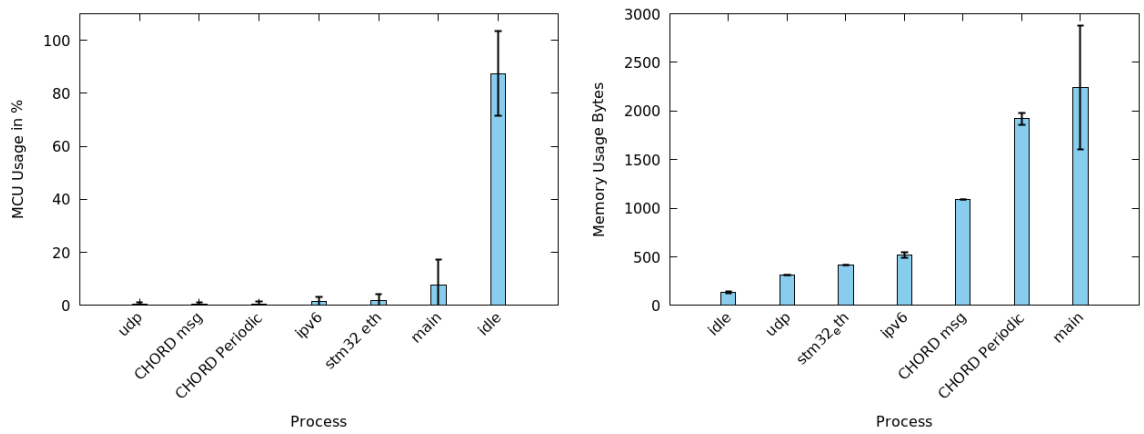
In this section, the RIOT storage driver is evaluated in different setups. Since the storage driver only acts as a wrapper for the DHT implementation as verified in the last section, only one additional non-functional evaluations are done. The experimental setup is used to verify the results on a real MCU as described at the beginning of this chapter. The access to real hardware is limited to seven available boards and thus the test setup is smaller then it is experienced from the last sections.

Non-Functional Performance Indicators

In this section an experiment is launched to verify the MCU utilization on a Nucleo-F767ZI board. This board provides an ARM Cortex M7 MCU with a clock rate of 216 MHz. As stated in the requirements the MCU utilization should not exceed more than 50% of the available capacity. This value is chosen somehow arbitrary it should be in fact only amount to a fraction of the available computational power. Calculating the utilization of the used MCU is a non-trivial task and can be done through two different methods. The first method could use the build in Data Watchpoint and Trace (DWT)[2] module of the STM32 MCU

[2] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0489d/BIIFBHIF.html`

which provides two counters to measure the cycle count and the cycle count in sleep mode respectively. As a downside, this method would imply that RIOT would use the sleep functionality of the MCU unit as much as possible. Furthermore, it would be impossible to clearly measure the overhead of MCU usage introduced by the storage driver independent of the MCU power used by RIOT. Another method makes use of the implemented debugging features in RIOT. Enabling the `MODULE_SCHEDSTATISTICS` flag in the preprocessor leads to a measurement of the time each thread is scheduled by the cooperative multitasking scheduler implemented in RIOT. The following experiment is done in a setup where each of the seven available Nucleo boards run the current implementation of the storage driver with each feature enabled. The used CPU utilization of each thread is computed every second for a total time of 100 seconds on every board. The data is collected and the mean of the data points is computed accordingly. Figure 6.14a shows the mean amount of time in percent each thread was scheduled. The error bars show the standard deviation of the data. It is easy to see that the MCU spends the most time in the idle thread which is scheduled more than 80% of the available time. The idle thread gets scheduled every time no other thread is available for scheduling due to being in sleep mode or blocking wait for an event. The main thread also has a rather high resource usage. This thread was used for the data computation and collection which runs in a computational heavy loop that leads to the high MCU usage and thus can be attributed to the idle thread. The remaining threads are created by the storage driver directly or indirectly in case of the UDP, ipv6 and network interface threads. All these remaining threads are exclusively used by the storage driver implementation and thus can be aggregated as the overhead introduces by the implemented software.



(a) CPU Usage

(b) Memory Usage

Figure 6.14: The mean MCU usage of each thread in percent and the maximum stack memory used in bytes. The error bars indicate the standard deviation.

Table 6.2 shows the total numbers in percent of the distinct threads and an aggregated value for the idle thread and the driver implementation. The aggregated values result in a MCU utilization of the implementation of about 5 %. Given the constrained computational power of the used MCUs, this value is rather low and can even be optimized by tweaking implementation specific parameters. Another measured value is stack size used by each thread to verify the total memory usage. Since RIOT and the driver implementation only uses static memory this value can be used to verify the total memory usage of the implementation.

Figure 6.14b shows the maximum stack usage for each thread over a time of 100 seconds measured every second. As Table 6.2 shows, the total memory usage of the driver implementation is about 4 kB. Since the board provides a total amount of memory of 512 kB the implementation uses under one percent of the available memory.

Thread	MCU Usage	Stack Usage
idle	87.44 %	135 B
main	7.55 %	2240 B
Idle Aggregated	**94.99 %**	**2375 B**
Chord Periodic	0.65 %	1920 B
Chord Waitloop	0.52 %	1088 B
ipv6	1.40 %	517 B
udp	0.50 %	308 B
stm32_eth	1.94 %	420 B
Chord Aggregated	**5.01 %**	**4253 B**

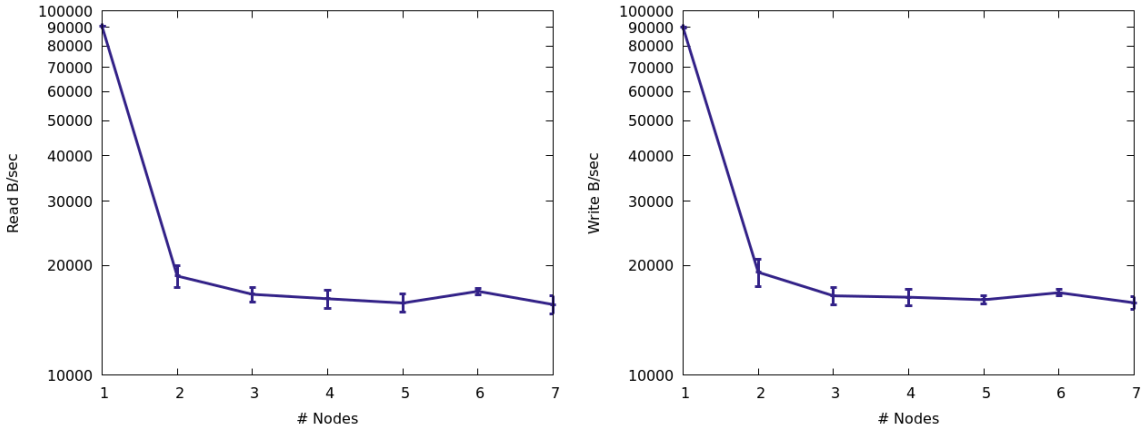
Table 6.2: Aggregated MCU and Stack usage

Functional Performance Indicators

As a functional performance indicator for the implemented storage driver, the main point of interest is the possible read and write performance. As stated in the requirements the read and write performance should be limited by the storage backend and every deviation of the theoretical maximum needs to be reasoned. By the time this experiment is done, there is no RIOT mtd driver available that uses the flash storage provided by the MCU. For this reason, the dynamic memory backend needs to be used which is limited by the read and write speed of the provided SRAM modules. A quick test using a POSIX compatible `memcpy` implementation provided by RIOT shows a raw write performance of about 5 MB/s.

The first experiment measures the read and write speed of the driver as the number of node increases from a single node to all available seven Nucleo nodes. Figure 6.15a shows the measured read performance and Figure 6.15b shows the measured write performance. As expected a setup with only a single node performance exceptionally better than a setup with multiple nodes. This can be easily explained by the introduces network overhead.

Two interesting findings of the experiments is the constant read and write speed as node count increases as well as the low performance of about 20 kB/s. Since the performance drops significantly as soon as multiple nodes are used, this leads to the assumption that the read and write speed is directly limited by the performance of the used Ethernet network. To back this assumption another experiment is done to verify the performance of an optimal case in comparison to the performance of a raw UDP transmission. To get an optimal case only two nodes are used in this experiments and replication of data items is disabled. This is due to the fact that a single store or retrieve operation consists of multiple UDP messages. First the responsible node for a data item needs to be found and afterward, the data item needs to be stored on the responsible node. This is done as often as needed by the defined replication mechanism and thus the replication acts as a multiplier on the network overhead. With only two nodes participating the node lookup can be omitted and with disabled replication, only



(a) Read B/s (b) Write B/s

Figure 6.15: Read and Write speed in bytes per second with increasing node count and standard deviation error bars. The graph uses a logarithmic scale for the y-axis.

a single message needs to be sent for each store and retrieve operation. As a benchmark baseline another performance evaluation is done which only sends a single UDP message with a payload as big as the block size of the storage driver to a modern desktop computer where the message gets echoed back to the sending Nucleo board. This is done to reflect an optimal case in terms of a request-response communication over UDP. Figure 6.16 shows the results of the experiment.

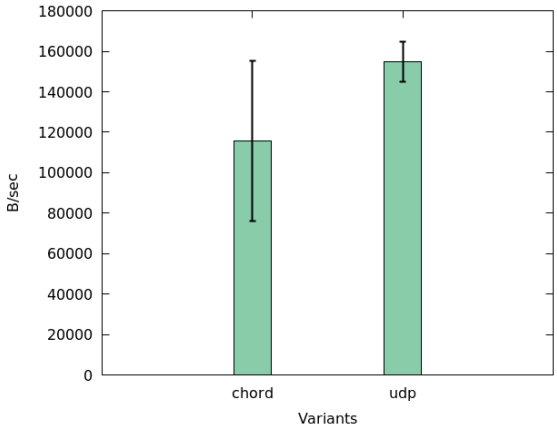
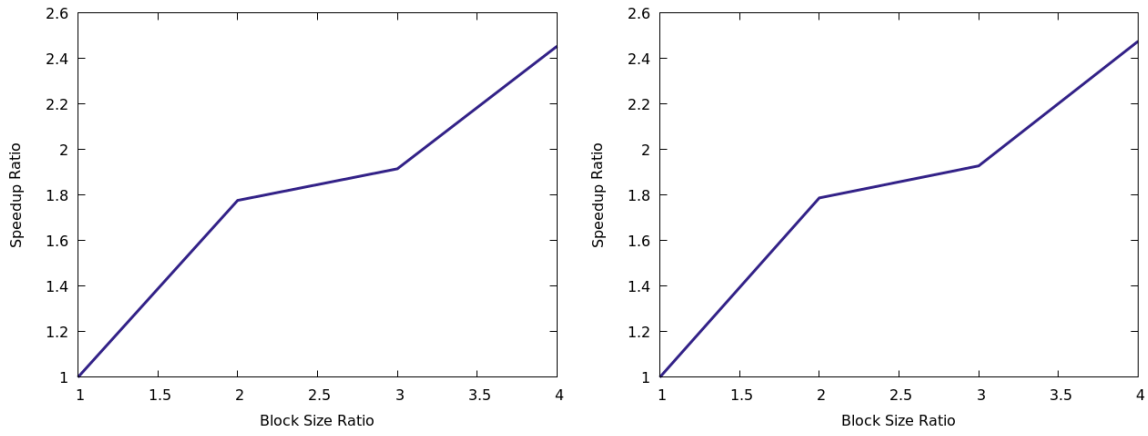


Figure 6.16: Raw Write speed of two nodes vs. simple udp echo benchmark

As the graph shows the read and write performance gets significantly better with the enabled optimizations. The raw UDP benchmark shows a mean performance of about 160 kB. This shows that the performance of the driver is not limited by the underlying storage technology but instead by the used network setup. The small gap between the average write performance in this test and the theoretical maximum of the UDP benchmark shows only a small overhead introduced by the storage driver. Also, the second experiment shows a way better write performance in contrast to the first experiment even if only two nodes are used. This can

be explained due to the fact that replication was disabled in the second experiment and furthermore the block size of the driver was increased by a factor of two. This leads to the assumption that the network performance is not limited by the size of the transmitted data but rather by the number of network packets sent. To verify this assumption another experiment is done where the block size of the storage driver and thus the amount of data transmitted in a single package is increased stepwise. If the assumption is correct we expect an increasing performance of a constant factor if the block size is increased by the same factor. As a baseline, the experiment uses a block size of 128 bytes.

Figure 6.17a and Figure 6.17b shows the result of the experiment. On the y-axis the speedup factor of the read and write performance is shown in relation to the initial performance with a block size of 128 bytes. The x-axis shows the factor by which the block size was increased. The graph shows a nearly linear correlation between the used block sized and the write performance. Additional overhead in the storage driver introduced by the increased block sized leads to a non-perfect linear correlation. As shown in the graph a factor 4 on the used block size (e. g. 512 bytes) leads to a speedup factor of 2.4.



(a) Read Speedup

(b) Write Speedup

Figure 6.17: Read and write speedup factor when the block size is increased by a linear factor.

6.5 Summary

The outcome of the P2P overlay protocol shows mixed results. For the non-functional performance indicators, it is noticeable that that tree construction protocol introduces a rather high overhead in terms of network and CPU usage. The results show a doubling in the number of bytes sent on every node when the tree construction protocol takes place. A future implementation should focus on minimizing this effect, either through optimization on the protocol level or through implementing a different feature for aggregations. In contrast, the load introduced by the Chord implementation as well as the id selection protocol is rather low in terms of both, network and CPU usage. It could be shown that the mean CPU utilization is below 1%. Furthermore, the memory used by the implementation follows the expected constant utilization. The functional indicators on the other side performed within the expectations. The resistance against churn performs as promised by the original

design of Chord, the id selection protocol provides a constant ratio of $\sigma \leq 4$ and the tree construction protocol generates a valid tree with a depth smaller than the expected maximum depth.

The DHT implementation shows promising results in terms of both functional and non-functional indicators. The data availability features show a linear correlation in terms of CPU and network usage if the number of replications or the number of elements in the ring is increased. The used replication level show a significant impact on the quality of the data availability and it was possible to show that the elements are distributed evenly along the Chord ring.

The storage driver itself, verified on real-world hardware confirms the results from the P2P protocol. Given a Nucleo-F767ZI board with an ARM M7 Microcontroller Unit shows a utilization of about 5% and a memory consumption of about 1%. The possible read and write speeds are smaller than expected given the SRAM backend. However, it was possible to show that the limiting factor resides in the network communication. A side-by-side evaluation of a raw UDP network performance test shows only a little difference in terms of the possible write speed the storage driver can achieve. The reasons for the limited network performance are most likely located in the RIOT implementation of the used network stack and the implemented Ethernet driver.

7 Conclusion and Future Work

This chapter provides a final conclusion about the contents of this thesis. The claims made in the problem statement are reviewed and compared with the design and the results of the evaluation. Multiple open questions and features out of the scope of this thesis where identified in the last chapters. A dedicated section focusing on this future work is provided for further reference.

7.1 Conclusion

In this thesis, we discussed the design and implementation of a P2P based distributed storage engine, with a special focus on devices with limited computing requirements. To achieve this existing designs and algorithms were adapted and combined to provide multiple features like resistance against device failures, an even distribution of the data over multiple nodes and a dynamic growth and shrinking of the storage. The claim of the design to follow a strict P2P architecture is only slightly affected by the tree construction protocol, where a single node is chosen to be the root of the tree depending on its position in the ring. Since this selection itself is fully distributed and the functioning of the algorithm is not affected by an outage of the root node we can argue that the system follows a strict P2P nature as addressed in the problem statement. We were able to gain a load distribution, orders of magnitudes better than the original design of the used P2P overlay provides and support for a dynamic variation of the network size due to the tree construction protocol. As shown in the performance measurements we were able to provide a uniform library for a Distributed Hash Table implementation, with a certain independence of the used host system, since we were able to run the library in a standalone GNU/Linux ELF binary or as a block storage driver on top of the RIOT operating system. Another important requirement was the ability of the resulting implementation to run on constrained environments. We were able to run the system on Nucleo-F767ZI boards with only a fraction of the computing power a modern consumer CPU provides.

On the downside, we were not able to implement all features accordingly to their specification. Furthermore, some features turned out to introduce a rather high overhead on computational power and network communication. As acknowledged in the last section there is a lot of potential for future work to improve existing features and for the implementation of missing features. The author considers the improvements on the tree construction protocol and the id space partitioning as mentioned above to be the most critical components since the impact on a productive usage is expected to be rather high.

We designed, implemented and verified all needed features as explained in the problem statement. The measurements of the performance indicators were mostly within the expectations and yield some interesting results. The most promising results are the high resistance against

churn which reflects the expectations raised by the original design of the used DHT implementation and the possibility to run the software in constrained environments due to the low resource usage shown by the verification on real hardware.

7.2 Future Work

As already stated multiple times in the last chapters and due to time and scope constraints there are multiple open or not yet addressed parts of the system which are considered as future work. First, there are multiple, already implemented, features which need optimizations and further evaluations.

- **Improvements on the Id Selection Protocol**
  It was not possible to implement the whole id selection protocol. An extensive evaluation of the protocol including an implementation which is able to handle leaving nodes as well would be interesting. Especially the impact of a final implementation on the resource usage and a verification of the guarantees on the id space partitioning could yield useful data.
- **Improvements on the Tree Construction Protocol**
  As seen in the verification the tree construction protocol works well but introduces a rather high communication overhead. Further studies with the aim to reduce this overhead are necessary. One may use the already existing fingertable of a Chord implementation to gain the same guarantees on broadcasting information with only adding little to none overhead.
- **Additional Storage Backends with Efficient Data Structures**
  The existing storage backends where build in respect to a minimal working implementation. For both implemented backends a linked list is used as a data structure to keep track of the stored blocks. This limits the read and write performance significantly. A full list search with a complexity of $\mathcal{O}(n)$ needs to be performed on every read or write operation. A future operation should implement a better-suited data structure like a binary tree or a static array.
- **Heterogeneity**
  As stated in the design, an implementation with multiple virtual nodes per real node is needed to support heterogeneity. The existing implementation does not support multiple virtual nodes per real node yet. This could yet only be simulated by starting multiple instances of the binary. A future implementation which supports an arbitrary amount of virtual nodes without increasing the number of needed threads would be feasible.
- **Improvements on the Bootstrap Protocol**
  As stated in the requirements an ideal bootstrap protocol also supports the automatic configuration of new devices. In the current implementation, new nodes only send a `MSG_TYPE_PING` to check the availability of bootstrap nodes. A more sophisticated implementation could include multiple network specific parameters in the response.
- **Further Optimizations for Constrained Environments**
  As the evaluation shows, the implementation is already able to work on constrained

environments. However, a further focus on reducing unneeded overhead could be valuable. As an example one could reconsider the usage of a hash function originating from the SHA family and choosing a more lightweight alternative. A detailed profile of the resource usage of the implementation could lead to further improvements.

Another group of future work considerations are features which could not be examined due to scope limitations. These features are missing a complete design and implementation.

- **Alternative P2P Overlay Networks**
  As stated in the design chapter a broad range of P2P overlay protocols with different target environments and optimizations is available. A further look into other protocols would be of interest and could help mitigate the issues with the id space partitioning and the tree construction protocol. Also, an approach which compares the impact of the different protocols on the resource usage and performance of the system could add further value.

- **Security Concerned Implementation**
  In the requirements chapter, the need for a security concerned implementation is already acknowledged. The current implementation does not take care of any authentication, encryption or protection against protocol specific attacks.

- **Neighbor Proximity**
  In the current state of the implementation, the physical proximity of neighboring nodes is not taken into account. It is possible that nodes within a single hop distance to another node need to take multiple hops through the network to reach their destination. Especially environments like WSNs can benefit from an implementation where neighboring nodes are also in a close distance in terms of the used P2P overlay implementation.

- **Partitioning**
  In the performance evaluation that concerns the stability of the P2P overlay network, only nodes failures without a closer look into the underlying network topology where taken into account. In a scenario where multiple nodes are connected through a single interconnection link, a failure of this link could easily lead to a partitioning of the ring topology into multiple distinct rings. This could result in a split brain scenario where it is not possible anymore to merge the partitions into a single ring without losing data. A feature which supports or mitigates such a scenario would be feasible.

List of Figures

List of Acronyms

AES Advanced Encryption Standard

AFS Andrew File System

API Application Programming Interface

CAN Content Addressable Network

CFS Cooperative File System

CIFS Common Internet File System

CPU Central Processing Unit

DBMS Database Management System

DHT Distributed Hash Table

DNS Domain Name System

DRBD Distributed Replicate Block Device

DWT Data Watchpoint and Trace

EBS Elastic Block Storage

ELF Executable and Linking Format

HDFS Hadoop File System

HRW Highest Random Weight

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

IPFS InterPlanetary File System

LAN Local Area Network

LC-VSS Low Cost Virtual Server Selection

LittleFS Little File System

LDPC Low-Density-Parity-Check-Codes

LRU Least Recently Used

NAT Network Address Translation

NFS Network File System

NVRAM Non Volatile Random Access Memory

MCU Microcontroller Unit

MTD Memory Technology Device

OWASP Open Web Application Security Project

P2P Peer-to-Peer

POSIX Portable Operating System Interface

RAID Redundant Array of Independent Disks

RAM Random Access Memory

RIOT Realtime Operating System for Internet of Things

ROM Read Only Memory

RPC Remote Procedure Call

SCSI Small Computer System Interface

SDS Software Defined Storage

SHA Secure Hash Algorithm

SNIA Storage Networking Industry Association

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

TD Thermal Dissipation

TLS Transport Layer Security

UDP User Datagram Protocol

VBS Virtual Block Store

WSN Wireless Sensor Network

Bibliography

[AAH+13] Atif Alamri, Wasai Shadab Ansari, Mohammad Mehedi Hassan, M. Shamim Hossain, Abdulhameed Alelaiwi, and M. Anwar Hossain. A survey on sensor-cloud: Architecture, applications, and approaches. *International Journal of Distributed Sensor Networks*, 9(2):917923, 2013.

[ABKU99] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[AH14] Mohammad Aazam and Eui-Nam Huh. Fog computing and smart gateway based communication for cloud of things. In *Proceedings of the 2014 International Conference on Future Internet of Things and Cloud*, FICLOUD '14, pages 464–470, Washington, DC, USA, 2014. IEEE Computer Society.

[AHKV03] Micah Adler, Eran Halperin, Richard M Karp, and Vijay V Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 575–584. ACM, 2003.

[AJX05] Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 336–345. IEEE, 2005.

[AR06] Filipe Araújo and Luís Rodrigues. Survey on distributed hash tables. In *Survey on Distributed Hash Tables*. University of Lisbon, 2006.

[BHW+12] Emmanuel Baccelli, Oliver Hahm, Matthias Wählisch, Mesut Guünes, and Thomas Schmidt. RIOT: One OS to Rule Them All in the IoT. Research Report RR-8176, INRIA, December 2012.

[Cat03] Josh Cates. *Robust and efficient data management for a distributed hash table.* PhD thesis, Massachusetts Institute of Technology, 2003.

[CBG12] Jeremy Hugues-Felix Constantin, Andreas Peter Burg, and Frank K Gurkaynak. Investigating the potential of custom instruction set extensions for sha-3 candidates on a 16-bit microcontroller architecture. Technical report, Cryptology ePrint Archive, 2012.

[CH07] Michael Conrad and Hans-Joachim Hof. A generic, self-organizing, and distributed bootstrap service for peer-to-peer networks. In *International Workshop on Self-Organizing Systems*, pages 59–72. Springer, 2007.

[CKF04] Curt Cramer, Kendy Kutzner, and Thomas Fuhrmann. Bootstrapping locality-aware p2p networks. In *ICON*, pages 357–361, 2004.

[Dav93] Alan M. Davis. *Software Requirements: Objects, Functions, and States.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[DBK+01] Frank Dabek, Emma Brunskill, M Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 81–86. IEEE, 2001.

[DGW+10] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE transactions on information theory*, 56(9):4539–4551, 2010.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[Ell08] Lars Ellenberg. Drbd 9 and device-mapper: Linux block level storage replication. In *Proceedings of the 15th International Linux System Technology Conference*, 2008.

[FMS+04] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Dependable Systems and Networks, 2004 International Conference on*, pages 125–134. IEEE, 2004.

[Gli07] Martin Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 21–26. IEEE, 2007.

[GMP+10] X. Gao, Y. Ma, M. Pierce, M. Lowe, and G. Fox. Building a distributed block storage system for cloud infrastructure. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 312–318, Nov 2010.

[GS05] P Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 596–606. IEEE, 2005.

[GWGR04] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Dependable Systems and Networks, 2004 International Conference on*, pages 135–144. IEEE, 2004.

[HAEF08] Y. Houri, B. Amann, B. Elser, and T. Fuhrmann. Igorfs: A distributed p2p file system. In *2008 Eighth International Conference on Peer-to-Peer Computing (P2P)*, volume 00, pages 77–78, 09 2008.

[HAY+05] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, volume 2, pages 205–213 Vol. 2, April 2005.

[Hei14] Jan Heichler. An introduction to BeeGFS. Research report, thinkparq, November 2014.

[HELD11] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[How88] John H Howard. An overview of the andrew file system. In *in Winter 1988 USENIX Conference Proceedings*, pages 23–26, 1988.

[IET98] IETF. Cache array routing protocol v1.0, 1998.

[IPF15] IPFS. Bootstrapping, 2015.

[JBA15] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials*, 17(1):381–404, 2015.

[KLL+97a] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[KLL+97b] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[Kum17] P. Vijay Kumar. Codes for big data: Erasure coding for distributed storage, 2017.

[LSL05] Ji Li, Karen Sollins, and Dah-Yoh Lim. Implementing aggregation and broadcast over distributed hash tables. *ACM SIGCOMM Computer Communication Review*, 35(1):81–92, 2005.

[Man04] Gurmeet Singh Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 197–205. ACM, 2004.

[Mic17] Microsoft. [ms-cifs]: Common internet file system (cifs) protocol, 2017.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, December 2002.

[MNY+18] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 20(1):416–464, Firstquarter 2018.

[MPV11] L. Mainetti, L. Patrono, and A. Vilei. Evolution of wireless sensor networks towards the internet of things: A survey. In *SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks*, pages 1–6, Sept 2011.

[OWA10] OWASP. Owasp secure coding practices quick reference guide, 2010.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[PJS+95] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. Nfs version 3 design and implementation. 02 1995.

[Pla05] James S. Plank. T1: Erasure codes for storage applications, 2005.

[Pre89] Franco P Preparata. Holographic dispersal and recovery of information. *IEEE Transactions on Information Theory*, 35(5):1123–1124, 1989.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[RL05] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *International Workshop on Peer-to-Peer Systems*, pages 226–239. Springer, 2005.

[RPW04] Simon Rieche, Leo Petrak, and Klaus Wehrle. A thermal-dissipation-based approach for balancing data load in distributed hash tables. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 15–23. IEEE, 2004.

[S+03] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[SAP+13] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[SC13] M. Krochmal S. Cheshire. Multicast dns, 2013.

[Sch02] Jeffrey I. Schiller. Strong security requirements for internet engineering task force standard protocols, 2002.

[SDM04] Paul Syverson, R Dingledine, and N Mathewson. Tor: The secondgeneration onion router. In *Usenix Security*, 2004.

[Sea18] Seagate. Scsi commands reference manual, 2018.

[SMK+01a] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[SMK+01b] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[SNI13] SNIA. Storage performance and io load basics, 2013.

[SS05] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hash tables. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 218–227, New York, NY, USA, 2005. ACM.

[SW05] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications (Lecture Notes in Computer Science)*. Springer, 2005.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[WBMM06] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.