Insider-Resistant Distributed Storage Systems

Dissertation

In partial fulfillment of the requirements for the academic degree
Doctor rerum naturalium (Dr. rer. nat.)

Faculty of Computer Science,
Electrical Engineering and Mathematics
Department of Computer Science
Research Group Theory of Distributed Systems

Martina Eikel

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Reviewers: Prof. Dr. Christian Scheideler, University of Paderborn
Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn
Contact: Martina Eikel (martina@eikel.org)

Acknowledgments

First and foremost, I thank my supervisor Christian Scheideler for his continuous support and many fruitful discussions. Without his help this thesis would not exist in this form.

I thank Alexander Setzer for the joint work in the field of insider-resistant distributed storage systems resulting in many productive, helpful and last but not least enjoyable discussions.

I thank my "roommates" Tobias Tscheuschner and Robert Gmyr for sharing an office with me. The many interesting technical and especially non-technical conversations with you always made working hours very pleasant. Additionally, I thank Robert for very patiently enduring my continuous finger drumming.

Thanks to Marion Bewermeyer for her recommendations on exciting books and interesting conversations about almost everything.

I also thank Rainer Feldmann for introducing me to the research of theoretical computer science by supervising my student job in that field and also supervising my Bachelor's and Master's thesis.

Furthermore, I thank our whole research group for the enjoyable lunch breaks with a wide variety of discussion topics and in general making my time in this group so pleasant.

For financially supporting my work, I thank the Collaborative Research Center 901 "On-The-Fly Computing".

I thank my family and friends for their moral support and improving my overall well-being. Especially, I thank my parents for paving the way for my education and always providing me a safe home.

Special thanks go to my beloved husband Benjamin for always being there for me.

Paderborn, November 2015 Martina Eikel

Abstract

In this work we present distributed information and storage systems that are provably robust against attacks by an insider adversary, i.e., an adversary with complete knowledge of the system. Our systems break the barrier between allowing significantly more than a polylogarithmic number of servers to be attacked by an insider adversary on the one hand, but still adhering to polylogarithmic efficiency bounds (in the number of servers) and at most logarithmic redundancy on the other hand. First, we present Basic IRIS, a distributed information system that works provably correctly despite the existence of an insider adversary that may use his knowledge about the system in order to crash up to $O(n^{1/\log\log n})$ servers (with n being the number of servers in the system). To be more precise, Basic IRIS provably correctly serves any set of lookup requests (with at most $O(1)$ requests per server not crashed) with polylogarithmic time and work (in the number of servers) and a constant redundancy. By extending Basic IRIS, we end up with Enhanced IRIS, a distributed information system that tolerates even up to a constant fraction of all servers to be crashed by an insider adversary. At the same time, Enhanced IRIS asymptotically achieves the same efficiency bounds except for the redundancy which increases to $O(\log n)$. While both Basic IRIS and Enhanced IRIS are restricted to the handling of lookup requests, with RoBuSt we introduce a distributed storage system with similar efficiency bounds. That is, despite the existence of an insider adversary that crashes up to $O(n^{1/\log\log n})$ servers, RoBuSt provably correctly answers any set of lookup and write requests (with at most $O(1)$ requests per server not crashed) with polylogarithmic time and work. Due to outdated data items that may remain in the system after performing updates on existing data items, the redundancy of RoBuSt increases to $O(\log n)$. Finally, we strengthened the adversary that we considered even further by allowing it to not only crash servers, but to corrupt their storage. By this, we end up with a somehow "semi-Byzantine" insider adversary. That is, in this setting the insider adversary is allowed to corrupt the permanent storage of the servers, but it is not allowed to corrupt the servers' main memory or the protocol itself. With OSIRIS we present a distributed storage system that provably correctly serves any set of lookup and write requests (with at most $O(1)$ requests per server) despite the existence of an insider adversary that may corrupt the permanent storage of at most $O(n^{1/\log\log n})$ servers while asymptotically adhering to the same efficiency and redundancy bounds as RoBuSt.

Zusammenfassung

In dieser Arbeit stellen wir verteilte Informations- und Speichersysteme vor, die gegen Angriffe eines Insiders beweisbar robust sind. Ein Insider ist in diesem Zusammenhang ein Gegner, der Wissen über das gesamte System hat. Unsere Systeme erlauben dabei signifikant mehr als polylogarithmisch viele durch einen Insider angegriffene Server und garantieren gleichzeitig polylogarithmische Schranken für die Effizienz und eine höchstens logarithmische Redundanz. Zuerst stellen wir Basic IRIS vor, ein verteiltes Informationssystem, das trotz der Existenz eines Insiders, der sein internes Wissen über das System nutzen kann, um bis zu $O(n^{1/\log\log n})$ Server abstürzen zu lassen, beweisbar korrekt arbeitet (wobei n die Anzahl der Server im System bezeichnet). Genauer gesagt beantwortet Basic IRIS jede Menge von Suchanfragen (mit höchstens $O(1)$ Anfragen pro nicht abgestürzten Server) in polylogarithmischer Zeit und mit polylogarithmischem Aufwand pro Server bei lediglich konstanter Redundanz. Durch Erweiterung von Basic IRIS erhalten wir Enhanced IRIS, ein verteiltes Informationssystem, welches gegen einen Insider robust ist, der bis zu einem konstanten Anteil aller Server zum Absturz bringen kann. Gleichzeitig erfüllt Enhanced IRIS asymptotisch die gleichen Effizienzschranken wie Basic IRIS, abgesehen von der Redundanz, welche auf $O(\log n)$ ansteigt. Während Basic IRIS und Enhanced IRIS auf die Beantwortung von Suchanfragen beschränkt sind, stellen wir mit RoBuSt ein verteiltes Speichersystem mit ähnlichen Effizienzschranken vor. Trotz der Existenz eines Insiders, der bis zu $O(n^{1/\log\log n})$ Server zum Absturz bringen kann, beantwortet RoBuSt jede Menge von Such- und Schreibanfragen (mit höchstens $O(1)$ Anfragen pro nicht abgestürzten Server) in polylogarithmischer Zeit und Arbeit beweisbar korrekt. Aufgrund von veralteten Daten, die nach Aktualisierungen im System zurückbleiben können, steigt die Redundanz von RoBuSt auf $O(\log n)$ an. Abschließend verstärken wir den betrachteten Gegner noch weiter, indem wir diesem das Korrumpieren des Speichers der Server erlauben. Hierdurch erhalten wir einen sogenannten „semi-Byzantinischen“ Insider. Das heißt, wir erlauben es dem Gegner den permanenten Speicher der Server zu korrumpieren, das Korrumpieren des Arbeitsspeichers der Server und des Protokolls ist hingegen nicht erlaubt. Mit OSIRIS stellen wir schließlich ein verteiltes Speichersystem vor, das jede Menge von Such- und Schreibanfragen (mit höchstens $O(1)$ Anfragen pro Server) trotz der Präsenz eines Insiders, der den permanenten Speicher von bis zu $O(n^{1/\log\log n})$ Servern korrumpieren darf, beweisbar korrekt beantwortet.

Contents

Chapter 1

Introduction

In recent years, the usage of online services has increased significantly. For instance, communicating with friends via Facebook, sharing videos via YouTube, or shopping online via Amazon. This induces the necessity to store large amounts of data online in such a way that they can be managed and accessed efficiently. Distributed storage systems constitute one of the most natural approaches for the implementation of such a storage. Popular examples include storage solutions offered by Google, Apple, and Amazon [Goo; App; Ama]. In this work a distributed storage system is defined as a network consisting of several servers that provide a lookup protocol and a write protocol. Via these protocols users can pose requests for adding data items to the system or for finding data items stored at the servers. If solely a lookup protocol and no write protocol is provided we call the system a distributed information system. Since availability and retrievability of the stored data is a key aspect of distributed storage systems, these systems should have various mechanisms in place to protect them against adversarial attacks. One of the biggest threats distributed storage systems are exposed to are crash failures. A server that experiences a crash failure is not available anymore, meaning that it neither responds to any request nor performs any further operations. Crash failures can be temporary or permanent and can have many causes, such as maintenance work, hardware or software failures, or denial-of-service (DoS) attacks. The basic goal of a DoS attack against a server is to make that server unavailable. There are various ways of achieving that, such as causing computationally expensive operations, exploiting protocol bugs, or overloading the intended server with junk traffic. Especially crash failures caused by DoS attacks can pose a serious threat, since

they usually are unpredictable, hard to prevent, and can cause the unavailability of a server for some time. As a report about network security [Net15] has revealed, distributed DoS attacks still are the number one operational threat to service providers. With the growth of bandwidth available, the size of the DoS attacks likewise significantly increases. In particular, the largest reported DoS attack in 2014 reached 400 Gbps of network junk traffic [Net15].

Besides crash failures, storage failures also constitute a big threat to distributed storage systems. A server that experiences a storage failure may hold arbitrarily corrupted data in its storage without being aware of that. While a crash failure can easily be detected using crash failure detectors, this does not hold for massive storage failures. Instead, the distributed storage system needs to implement techniques and methods in order to work correctly despite the existence of servers with storage failures. Storage failures may not only be caused by malicious adversaries, they may also occur due to technical errors, such as disk faults or physical interconnect failures. For instance, in 2008 Amazon's S3 storage service experienced a multi-hour downtime due to a single bit corruption resulting in monetary loss for Amazon and unavailability of data stored at the S3 storage service [Ama08].

The predominant approach in distributed storage systems to deal with the threat of failures is to use redundancy and information hiding: The idea behind this is that information which is not only stored at a single server, but also replicated on multiple servers, is more likely to remain accessible during an attack, in particular if the adversary does not know the storage locations of the redundant data items. For example, if a logarithmic number of copies of each data item is distributed among the servers in the system, and the adversary is not aware of these locations, then it is easy to see that with high probability a copy of each data item is still accessible if the adversary crashes less than half of the servers. However, the situation is completely different when considering an insider adversary, i.e., someone who has *complete* knowledge of the system and may use this knowledge to crash a large fraction of the servers. Since information cannot be hidden anymore in this case, it seems unavoidable to replicate each data item across more than t servers in order to remain accessible if the system is under an attack that crashes t servers. Unfortunately, in this case the storage overhead becomes very large when considering adversaries that may crash a large fraction of the servers. However, it turns out that this dilemma can be circumvented when using coding, which is one of the key ideas we use in the development of the systems presented in this work.

There are various reasons for an attacker to have access to insider knowledge about a system. For instance, the attacker can be a (former) administrator of the system or someone who illegally obtained access to secure information about the system. As a recent study [IBM15] has shown, attacks performed by an insider pose a big threat to current systems. This result was gained

in the context of the IBM Cyber Security Intelligence Index, which offers an overview of the major threats to business systems worldwide over the past year. For that purpose, IBM continuously monitors billions of security events for clients all over the world. This study showed that more than half of the attacks monitored in 2014 were carried out by insiders.

The goal of this work is the development of distributed information and storage systems that provide efficient lookup and write protocols that work provably correctly despite the existence of an insider adversary that may attack a large fraction of the servers by causing crash failures or a special type of storage failure. In this context, by efficient we mean at most polylogarithmic in the number of servers, and with a large fraction of attacked servers we mean asymptotically much larger than polylogarithmic, in particular $O(n^{1/\log\log n})$, with n being the number of servers, or even up to a constant fraction of all servers. At the same time, we ensure the additional amount of storage required by each server to be limited by at most a logarithmic factor. Note that the systems presented in this work constitute a proof of concept of distributed information and storage systems with the desired properties. In particular, we prove the correctness and efficiency of these systems, but we do not provide a practical implementation.

1.1 Thesis Overview

Before we present the distributed information and storage systems we develop, we first introduce previous works that are related to the content of this work (Section 1.2). Afterwards, in Chapter 2, we describe the model specifications considered for this work and introduce some required preliminaries.

In the following main chapters of this work we present in total four distributed information and storage systems that are provably robust against insider attacks. The first two systems we present, Basic IRIS (Chapter 3) and Enhanced IRIS (Chapter 4), are based on the following journal article:

> **2015** (with C. Scheideler). "IRIS: A Robust Information System Against Insider DoS-Attacks". In: *ACM Transactions on Parallel Computing*, pp. 18:1–18:33, cf. [ES15].

Furthermore, an extended abstract about these two systems has been published in the proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) [ES13].

Basic IRIS is a distributed information system that is provably robust against an insider adversary that crashes up to $O(n^{1/\log\log n})$ servers while requiring only a constant storage redundancy. The main innovation in this system is the development of a technique for the efficient encoding of the data items stored in

the system with each other using a hierarchical coding strategy that is based on the structure of a k-ary butterfly ($k = \Theta(\log n)$) and a simple parity-based code. This technique allows to specify a lookup protocol that guarantees to correctly serve each lookup request for any data item with polylogarithmic work at each server and polylogarithmic time, although the adversary may crash up to $O(n^{1/\log\log n})$ servers. This coding strategy is also used as a building block in the further distributed information and storage systems presented in this work.

Enhanced IRIS is an extension of Basic IRIS that is able to tolerate even up to a constant fraction of all servers to be crashed. Except for the storage redundancy, which increases to a logarithmic factor, Enhanced IRIS still guarantees the same properties as Basic IRIS. The main idea behind this extension of Basic IRIS is to not only use a k-ary butterfly as the underlying topology for the encoding, but to additionally make use of permutations that fulfill certain expansion properties in order to spread the encoding information even further among the servers.

While Basic IRIS and Enhanced IRIS are distributed *information* systems that provide only a lookup functionality, in Chapter 5 we present RoBuSt, a distributed *storage* system, i.e., a system that besides the lookup functionality also provides a write functionality. This system is based on the following publication:

> **2013** (with C. Scheideler and A. Setzer). "RoBuSt: A Crash-Failure-Resistant Distributed Storage System". In: *Proceedings of the 18th International Conference on the Principles of Distributed Systems (OPODIS)*, pp. 107–122, cf. [ESS14].

More precisely, RoBuSt is a distributed storage system that correctly handles lookup and write requests in polylogarithmic time and with polylogarithmic work despite the existence of an insider adversary that crashes up to $O(n^{1/\log\log n})$ servers. Thereby, RoBuSt requires only a logarithmic storage redundancy. RoBuSt reuses the k-ary butterfly encoding approach introduced with Basic IRIS with the additional ingredient of a clever arrangement of the data items stored in the systems into so-called buckets and an appropriate strategy for traversing the buckets efficiently.

In Chapter 6 we strengthen the adversary considered in such a way that it now may not only crash servers, but instead even corrupt the storage of up to $O(n^{1/\log\log n})$ servers. Here, we confine ourselves to the corruption of the data stored at the servers while assuming the protocols and main memory of the servers to be reliable. As we will discuss later, this kind of attack can also be interpreted as a DNS spoofing attack. The main challenge in this setting is that in contrast to crashed servers there is no way to efficiently detect corrupted

servers. Hence, we need to add techniques for verifying the validity of data. By appropriately interweaving techniques from the field of authenticated data structures, namely Merkle trees, with techniques developed for IRIS and RoBuSt, we develop OSIRIS, a distributed storage system that is provably robust against an insider adversary that may corrupt the storage of up to $O(n^{1/\log\log n})$ servers. At the same time, OSIRIS correctly answers any set of lookup and write requests in polylogarithmic time and with polylogarithmic work per server while requiring a logarithmic redundancy only.

We conclude this work in Chapter 7 with a summary and discussion on interesting open problems in this field.

1.2 Related work

Throughout this work we make use of techniques from several fields of research. Most suitably this work can be classified into the field of distributed hash tables. Therefore, in Section 1.2.1 we provide an overview of the results important to us in that context. Besides this basic concept, this work also makes use of techniques from the field of (erasure) coding, network coding, and authenticated data structures. A short overview on results from these fields important for this work is given in Sections 1.2.2 and 1.2.3.

1.2.1 Distributed Hash Tables

The most prominent approach for the implementation of a distributed storage system is to build on a distributed hash table (DHT), i.e., a distributed system that provides a write and lookup functionality in order to add data to the system or to retrieve data from the system. In order to assign data items to store in the DHT to the servers, most distributed hash tables make use of the approach of consistent hashing [Kar+97]. In this approach, both data items and servers are mapped to the $[0, 1)$ interval. By this, each data item is assigned to the nearest server (to its right) in the $[0, 1)$ interval. Together with the use of Cuckoo hashing [PR04], one can additionally easily achieve an even load balancing among the servers.

While in early DHTs [Kar+97] each server used to maintain a list of all other servers in the network, this is too expensive when considering networks of millions of servers. Thus, the focus moved to scalable DHTs, i.e., DHTs that efficiently provide their basic functionalities (e.g., lookup/write operations, or join/leave operations) even for a very large number of servers in the system. For this purpose, in many scalable information and storage systems each server does not store a list of all servers in the network, but only of a subset of these. For instance in the Chord system [Sto+01] each server stores only a reference to $O(\log n)$ servers (where n is the total number of servers in the system) while

still being able to handle join and leave operations and to answer all lookup requests using $O(\log n)$ hops in the network. The Koorde system [KK03] is based on Chord but only requires 2 neighbors for each node while still requiring only $O(\log n)$ hops for a lookup. When allowing $O(\log n)$ neighbors per node, Koorde even requires only $O(\log n / \log \log n)$ hops for a lookup. SkipNet [Har+02], which is a distributed generalization of Skip Lists [Pug90], achieves the same efficiency results as Chord. But in contrast to Chord, SkipNet enables the possibility of explicitly controlling the location of the data placement. Examples for further scalable information systems with similar results are Pastry [RD01], CAN [Rat+01], P-Grid [Abe01], Viceroy [MNR02], and Tapestry [ZKJ01].

When using a pure DHT design as in the previously mentioned systems, bottlenecks may occur at some servers in the case of flash-crowd attacks. In a flash-crowd attack the affected servers experience a sudden and large increase in their traffic. This could, for instance, be due to too many requests for the same file at a single server. However, there are strategies that are robust against flash-crowd attacks: e.g., in CoopNet [PS02] the problem of web flash crowds is circumvented via the use of cooperation among the servers. Backslash [SMB02] builds on a DHT together with the use of caches. In PROOFS [SRS02] the connectivity in the overlay is randomized to make the system more robust. Naor and Wieder [NW03] provide a DHT which by using dynamic caching techniques maintains low load and storage at each server even under the occurrence of flash crowds.

A disadvantage of the previously mentioned systems is the missing robustness against adaptive lookup attacks, i.e., attacks that crash a server not by flooding the system with a request for the same data item, but by requesting too many (possibly distinct) data items that are located at the same server. In this case, standard combining and caching techniques do not work any more. Strategies that are robust against adaptive lookup attacks date back to the time of deterministic simulation of CRCW PRAMS on complex networks [MV84]. The first usage of these strategies in P2P networks is due to Awerbuch and Scheideler [AS06]. Awerbuch and Scheideler present the first scalable and robust DHT that is provably robust against adaptive adversarial join/leave attacks and lookup/insert attacks [AS06]. In a follow-up work [Awe07] Awerbuch and Scheideler design oblivious join and leave operations that are provably robust against any combination of outsider and insider attacks. Here, outsider attacks are brute-force DoS attacks that may be performed against any server in the network. An insider attack inserts adversarial servers into the system via join/leave attacks.

Unfortunately, these strategies are not robust against an adversary that attacks the existing servers, e.g., via massive DoS attacks. For instance, the system proposed in [AS06] makes use of quorums of servers such that the number

of attacked and non-attacked servers in each quorum is balanced. However, an adversary that is able to attack existing servers could easily unbalance these quorums and cause the system to not work correctly anymore. However, there exist systems whose functionality and efficiency has been shown experimentally, which are robust against DoS attacks such as SOS [KMR02], WebSoS [Sta+05], Mayday [And03], StopIt [LYL08], TVA [YWA05], and NetFence [LYX10]. A disadvantage of theses systems is that they are not provably robust against *insider* DoS attacks, i.e., against adversaries who know everything about the system and who may use this knowledge to attack the servers. By these means, attacks of an insider constitute worst-case attacks. Prior to this work, Awerbuch and Scheideler [AS07] and Awerbuch et al. [BSS09] were the first to present distributed information/storage systems that are provably robust against DoS attacks by *past insider* adversaries. A past insider is an attacker who has knowledge about the complete system up to some point in time t_0. The systems by Awerbuch et al. [AS07; BSS09] guarantee that any lookup or write request for a data item that was inserted or updated after t_0 can be answered correctly in polylogarithmic time (in the number of servers) and with a constant redundancy only.

Worst-case failures are also studied by Kuhn et al. [KSW10] and Augustine et al. [Aug+13]. Kuhn et al. [KSW10] propose a system that is based on a hypercube which tolerates $\Theta(\log n)$ worst-case joins and/or crashes per constant time interval while still being as efficient as previous systems, i.e., having $O(\log n)$ neighbors per server and guaranteeing a lookup time of $O(\log n)$ rounds. Augustine et al. [Aug+13] present a system that despite a high node churn rate of $O(n/\log^{1+\delta} n)$ (for $\delta > 0$ constant) per round guarantees that a large number of nodes in the system can store, retrieve, and maintain a large number of data items.

When not restricting to crash failures but instead considering Byzantine servers, the previously mentioned protocols are not guaranteed to work correctly any further. Fiat et al. [FSY05] propose a variant of Chord which is robust against an adversary who chooses up to a constant fraction of the servers to be Byzantine. However, their system is not able to handle Byzantine servers that were chosen and are controlled by an insider adversary. More information on related work in the field of Byzantine fault-tolerant storage systems is given in Section 1.2.2.

1.2.2 Erasure Codes & Byzantine Fault-Tolerant Storage Systems

The systems proposed in this work make use of techniques from the field of coding in multiple ways: First, we apply erasure codes instead of simple data replication in order to decrease the redundancy required. Next, we introduce a very simple error correcting code that helps us to recover data in case of a

single failure. Finally, we introduce a distributed error detecting and correcting code in order to detect and correct single failures which are due to storage failures.

An erasure code is a forward error correction code that, given a message of k symbols, outputs a code word of $n > k$ symbols such that the original message can be recovered from a subset of the n symbols of the code word. If the erasure code guarantees the recovery of the original message from any k symbols of the code word, then it is called optimal. Examples of optimal erasure codes include Reed-Solomon codes, EVENODD, Star-Code, X-Code, B-Code, and Zigzag-Codes [RS60; Bla+95; HX05; XB99; Xu+06; TWB13].

In our work we not only apply erasure codes for error correction, but we also use them instead of data replication in order to decrease the redundancy required.

Weatherspoon and Kubiatowicz showed in a quantitative comparison [WK02] that a self-repairing distributed storage system based on erasure codes outperforms a self-repairing distributed storage system based on replication in many aspects, e.g., due to huge storage and bandwidth savings. However, as Rodrigues and Liskov [RL05] have shown, the benefits of erasure coding are often rather limited, such that the disadvantages and additional complexity of erasure codes can outweigh their benefits.

In the distributed storage system we present in Chapter 6 we do not solely focus on crash failures but instead consider a special type of storage failures: i.e., the storage of some servers (excluding the main memory and the part of storage that holds the protocol definition) may be arbitrarily corrupted. A protocol that can tolerate arbitrary faulty behavior by the servers is said to be Byzantine fault-tolerant [LSP82]. Besides data replication [MAD02; CDV13; MR97] erasure codes are an important ingredient for the construction of distributed storage systems that tolerate Byzantine failures. Works from this field usually differentiated between (storage) servers and clients where the clients may access the servers for reading and writing data [Goo+04; CT05b; And+14].

By using erasure codes and cryptographic hashes, Goodson et al. [Goo+04] present a protocol that tolerates Byzantine failures of storage servers and clients. In order to tolerate f faults, their protocol requires $4f + 1$ servers. Cachin and Tessaro [CT05b] improve the result of Goodson et al. [Goo+04] (in terms of resilience and storage complexity) by presenting the first distributed erasure-coded storage system that provides atomic semantics and optimal resilience. In particular, their protocol requires only $3f + 1$ servers to tolerate f faults. Their approach is based on AVID, a verifiable information dispersal protocol [CT05a] and erasure coding. Hendricks et al. [HGR07] in turn modify the AVID protocol to utilize homomorphic fingerprinting in order to make it more bandwidth-efficient.

Androulaki et al. [And+14] noticed that much of the overhead of the proto-

cols presented in [CT05b; HGR07] is due to the ability of the protocols to even handle Byzantine clients and servers, instead of only Byzantine servers. For the setting in which only the storage servers may be Byzantine and the clients may crash only, Androulaki et al. [And+14] present an asynchronous Byzantine fault-tolerant erasure-coded storage protocol with optimal resilience. The resilience is reached by separating metadata from erasure-coded fragments.

1.2.3 Authenticated Data Structures

A fundamental problem that arises with the presence of servers whose storage is corrupted, or even Byzantine servers, is the potential corruption of any data item received from a server. Without additional mechanisms it is not possible to guarantee the correctness of data stored in the system. One way to overcome this problem is the usage of authenticated data structures. An authenticated data structure typically involves a structured collection S of objects (e.g., a set of data items to store in the data structure) and three parties: the source, the responder, and the user [Tam03]. The source usually is assumed to be a trusted party that holds the original versions of objects from S. Additionally, it holds authentication information for each object from S (e.g., hashes of the objects) which is supposed to prove the objects' validity. Whenever an operation is performed on any object in S, the source also updates the corresponding authentication information. The responder maintains a copy of S and interacts with the source by receiving updates performed at the set S and their authentication information. Furthermore, the responder answers the user's queries to the set S by providing both the requested object and the authentication information of that object. The user poses requests for an object in S only to the responder, not to the source itself. Since the user does not trust the responder, but the source, it uses the received authentication information to verify the correctness of the received object.

Merkle trees [Mer79] constitute a predominant approach for the efficient implementation of authenticated dictionaries, i.e., authenticated data structures that support the lookup and write functionality on a static set of objects. In Merkle trees (also called hash trees) cryptographic hashes are used as authentication information for the objects in the dictionary. Applications of Merkle trees can, for instance, be found in the IPFS and ZFS file system [Int; Bon+03], BitTorrent [Bit], Git [Tor], and Bitcoin [Nak09]. There are also dynamic authenticated dictionaries, e.g., based on hash trees [NN98], based on skip lists [GT01], or based on RSA one-way accumulators [GTH02]. A survey on authenticated data structures was written by Tamassia and Roberto [Tam03].

The first to study authenticated distributed hash tables are Tamassia and Triandopoulos [TT05; TT07]. Previous DHTs that supported authentication used signatures and cryptographic hash functions applied to each single object.

The authenticated DHT in [TT05; TT07] is based on the design of an efficient distributed Merkle tree, it uses $O(\log n)$ storage per node, and a lookup operation requires $O(\log n)$ network hops. Note that the distributed Merkle tree [TT05; TT07] still requires the existence of a trusted source which is not given in the settings we consider. Papamanthou et al. [PTT09] present a distributed implementation of a Merkle tree with nearby optimal search performance and that, in contrast to [TT05; TT07], also preserves load balancing but also requires the existence of a public key infrastructure.

Chapter 2

Model and Preliminaries

In this chapter we specify the model that we assume for all information/storage systems presented in this work (Section 2.1) and introduce some preliminary tools that we make use of throughout this work (Section 2.2). Since some model details (e.g., the specific type of adversary) differ for the systems presented here, in this chapter we provide a general model specification which holds for all presented systems. Additionally, when presenting a specific information/storage system in the further chapters, we first introduce the necessary model assumptions that are required additionally to the ones provided in this chapter.

2.1 Model

In this work we consider distributed storage systems that consist of a static set $\mathcal{S} = \{s_0, \ldots, s_{n-1}\}$ of n reliable servers of identical type. The servers are responsible for storing the data as well as handling user requests. For all information and storage systems, except for the one that we present in Chapter 6, we assume that the servers form a clique. For the system presented in Chapter 6 we weaken this assumption and only require the servers to be connected via a so-called $\log n$-ary butterfly, which we will define in Section 2.2. In this structure each server is not connected to all other servers anymore but only to $\Theta(\log^2 n / \log\log n)$ many other server but can communicate with any other server in $O(\log n / \log\log n)$ rounds. Note that we do not require the servers to maintain an open connection to each of the servers it is connected to in the overlay. Instead, we only expect the servers to hold the IP addresses of these servers. Besides this, we assume all links between servers to be reliable.

We use the standard **synchronous message passing model** for the communication between the servers. That is, time proceeds in synchronized **communication rounds**, or simply **rounds**, and in each round each server first receives all messages sent to it in the previous round, processes all of them, and then sends out all messages that it wants to send in this round. We assume that the time needed for internal computations is negligible, which is reasonable as the operations in the protocols we describe are simple enough to satisfy this property.

The storage systems we consider are supposed to hold data items, where each data item d is uniquely identified by a key $\mathsf{key}(d)$. The universe of all possible keys is called $\mathcal{U}$, and $m := |\mathcal{U}|$ is assumed to be polynomial in n. For simplicity, we assume all data items to be of the same size, which is assumed to be at most polylogarithmic in n. Besides, bigger data items can still be handled by our system, but in this case the maximum message size is proportional to the size of the biggest data item in the system. Alternatively, huge data items can be split into several chunks such that each of them is at most polylogarithmic in n.

In this work we consider two types of user requests: $\mathsf{lookup}(k)$ for $k \in \mathcal{U}$, and $\mathsf{write}(k, d)$ for $k \in \mathcal{U}$ and a data item d. The user can issue a request by sending it to one of the servers in $\mathcal{S}$. Given a $\mathsf{lookup}(k)$ request, the system is supposed to either return the data item d with $key(d) = k$, or to return NULL if no such data item exists. Given a $\mathsf{write}(k, d)$ request, the system is supposed to store data item d with key k such that subsequent $\mathsf{lookup}(k)$ requests can be answered correctly. Note that via a $\mathsf{write}(\cdot)$ request the user may also update or remove data.

Throughout the whole work we assume the existence of an **insider adversary**. An insider adversary is an adversary that has *complete* knowledge of the system and can use this knowledge to attack arbitrarily chosen servers. In this work two kinds of adaptive failures caused by the insider adversary at the servers are considered: **crash failures** and **storage failures**.

If a server experiences a crash failure, it becomes unavailable to the other servers: i.e., it does not issue or respond to requests any more. In this case we call the attacked server **crashed**. However, we assume the servers to have a crash failure detector that allows them to determine whether a server is crashed such that statements like "if server s is crashed, then..." are possible. Denial-of-service (DoS) attacks constitute a common example for crash failures. DoS attacks overwhelm servers with hacker-generated traffic and make them unavailable for legitimate communications such that to the other servers in the system the overwhelmed servers appear to be crashed. There are various ways of achieving that, like causing computationally expensive operations [Kan+05], downloading large files, exploiting protocol bugs, or just overloading servers with junk requests.

In case of storage failures the affected servers may hold arbitrarily corrupted data without being aware of that. In this case we call the attacked server **corrupted**. In contrast to crash failures, there is no storage failure detector. That is, we cannot rely on any data returned by a server and need to incorporate other mechanisms in order to verify the correctness of data. A server that is not attacked by the adversary is called **intact**.

Throughout this work we assume the adversary to be **batch-based**. For this purpose, rounds are divided into **periods** such that within each single period a set of requests can be answered. At the beginning of each period, the adversary chooses an arbitrary set of servers it attacks. The number of servers to be attacked and the type of the attack (crash failure or storage failure) depends on the storage/information system and will be specified separately in the corresponding chapters. Besides the set of attacked servers, the adversary may also choose any set of requests to be stated at the servers with at most $O(1)$ requests per server. The keys selected for the requests by the adversary may or may not be associated with data items stored in the system, and the attacker is also allowed to issue multiple lookup requests for the same key. That is, besides worst-case failures, we also consider worst-case requests. Furthermore, we assume the adversary not to be able to predict (future) random choices of the system. However, except for the constructions of OSIRIS in Chapter 6, the adversary considered in this work is not restricted to be polynomially bounded.

In order to measure the quality of the information system, we introduce the following notation. A storage strategy is said to have a **redundancy** of r if r times more storage (including any control storage) is used for the data than storing the plain data. Instead of "polynomially logarithmic in n" we also use the short form polylog(n). A server s is said to have a **congestion** of r in some round t if s receives and sends at most r messages of size at most polylog(n) in round t. In this context we assume that if a server receives more than a polylogarithmic number of messages in a round t, then it does not reply to any of these messages. That is, such a server is considered as crashed in round t.

We call an information/storage system

- **scalable** if its redundancy is at most polylog(n),
- **efficient** if any collection of lookup or write requests (depending on the system) specified by the attacker can be processed correctly in at most polylog(n) many communication rounds in which every server sends and receives at most polylog(n) many messages of at most polylog(n) size, and
- **robust** if any collection of lookup or write requests specified by the attacker (with at most $O(1)$ requests per server) can be processed correctly

even if up to an ε-fraction of the servers is attacked by an insider, where the type of the attack and the value for ε depend on the system.

Note that we could also allow the adversary to pose more than $O(1)$ requests per server, but the system's efficiency scales linearly with the number or requests per client. In particular, if there is a client with $\omega(\text{polylog}(n))$ requests, naturally no distributed information/storage system with at most polylogarithmic redundancy can answer all these requests in polylogarithmic time and work in worst case.

In this work we present information/storage systems that are scalable, efficient, and robust for the case of crash failures (Basic IRIS, see Chapter 3, Enhanced IRIS, see Chapter 4 and RoBuSt, see Chapter 5) or storage failures (OSIRIS, see Chapter 6).

2.2 Preliminaries

In this work we frequently make use of the well-known Chernoff bounds [Che52]:

Lemma 2.1 ([Che52])**.** *Let $X_1, \ldots, X_n$ be independent binary random variables. Consider $X = \sum_{i=1}^{n} X_i$ and let $\mu = E[X]$. Then, for all $\delta \geqslant 0$ it holds:*

$$Pr[X \geqslant (1+\delta)\mu] \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \leqslant e^{-\frac{\delta^2\mu}{2(1+\delta/3)}} \tag{2.1}$$

Furthermore, for all $0 \leqslant \delta \leqslant 1$ it holds:

$$Pr[X \leqslant (1-\delta)\mu] \leqslant \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \leqslant e^{-\delta^2\mu/2} \tag{2.2}$$

Inequality (2.1) also holds for any $\mu \geqslant E[X]$, and Inequality (2.2) also holds for any $\mu \leqslant E[X]$.

The following lemma represents a standard example on the application of the Chernoff bounds and is also used at several places throughout this work. In this lemma and throughout this work we make use of the notion "with high probability", or short, "w.h.p." in order to refer to a probability of at least $1 - 1/n^c$ where the constant c can be made arbitrarily large.

Lemma 2.2. *Consider n servers $s_0, \ldots, s_{n-1}$, where εn, for $\varepsilon < 1/2$, of these servers are attacked. When choosing $2 \log n$ servers uniformly at random, then at least $\log n$ of these servers are intact, w.h.p.*

Proof. We introduce a binary random variable X_i for each server s_i, $i \in \{0, \ldots, n-1\}$, with $X_i = 1$ if and only if server s_i is intact. Let $\delta := 1 - \varepsilon > 1/2$ be the fraction of intact servers among all servers $s_0, \ldots, s_{n-1}$. Let $s_{i_1}, \ldots, s_{i_{2 \log n}}$ be the servers chosen uniformly at random. Define $X := X_{i_1} + \ldots + X_{i_{2 \log n}}$. Then, $E[X] > \log n$. Inequality (2.2) implies:

$$Pr[X \leqslant (1-\delta) \log n] \leqslant e^{-\delta^2 \log n/2}$$

Hence, half of the chosen servers are intact, w.h.p. □

Although we assume the servers in the systems presented in Chapters 3–5 to initially form a clique we only require such a strong connectivity at the beginning of the periods, as specified later. After a preprocessing stage, we only require the servers to be connected via a so-called d-dimensional k-ary butterfly, as defined below.

In the following we use the notation $[k] = \{0, \ldots, k-1\}$ for $k \in \mathbb{N}$.

Definition 2.3. *For any $d, k \in \mathbb{N}$, the d-**dimensional** k-**ary butterfly** $BF(k, d)$ is a graph $G = (V_k, E)$ with node set $V_k = [d+1] \times [k]^d$ and edge set E with*

$$E = \{\{(i, x), (i+1, (x_1, \ldots, x_i, b, x_{i+2}, \ldots, x_d))\} \mid x = (x_1, \ldots, x_d) \in [k]^d,\ i \in [d], \text{ and } b \in [k]\}.$$

*A node u of the form (ℓ, x) is said to **be on butterfly level** ℓ of G. Furthermore, $LT(u)$ is the unique k-ary tree of nodes reached from u when going downwards the butterfly (i.e., to nodes on butterfly levels $\ell' > \ell$) and $UT(u)$ is the unique k-ary tree of nodes reached from u when going upwards the butterfly. Moreover, for a node u at level ℓ, let $BF(u)$ be the unique k-ary ℓ-dimensional subbutterfly ranging from butterfly level 0 to ℓ in $BF(k, d)$ that contains u. Finally, we name the unique k-ary subbutterfly of dimension 1 ranging from level ℓ to $\ell + 1$ in $BF(k, d)$ the k-block at level ℓ.*

A visualization of a k-ary butterfly is given in Figure 2.1. Note that each k-block at level $\ell \in \{0, \ldots, \log_k n - 1\}$ is a complete bipartite graph consisting of k nodes on level ℓ and k nodes on level $\ell + 1$. Hence, in each level $\ell \in \{0, \ldots, \log_k n - 1\}$ there are n/k disjoint k-blocks. Furthermore, note that in this and all following figures we depict level 0 as the uppermost and level $\log_k n$ as the lowermost level. Thus, for a level $\ell \in \{0, \ldots, \log_k n\}$ we call all levels $\ell' < \ell$ **higher** levels and all levels $\ell' > \ell$ **lower** levels.

Let $BF(k, d)$ be a k-ary butterfly with $k = \log n$ and $n = k^d$ nodes, i.e., $d = \log_k n$. Define s_i, $i \in \{0, \ldots, n-1\}$, as the server responsible for the $d+1$ butterfly nodes in the i-th column of $BF(k, d)$, i.e., $(0, i), \ldots, (d, i)$. In the following a server responsible for a butterfly node emulates that node. That is, whenever a butterfly node $(0, i), \ldots, (d, i)$ is supposed to perform an action

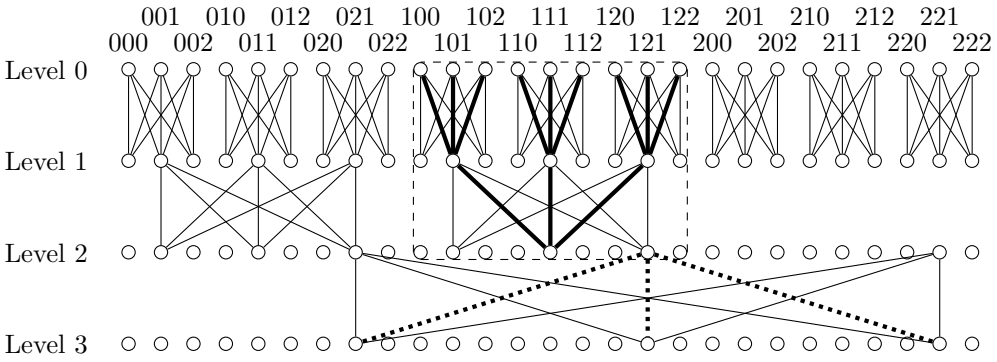

Figure 2.1: Visualization of a k-ary butterfly $BF(k, d)$ for $k = d = 3$. For better readability most of the edges between level one and two and between level two and three are omitted. The dashed box denotes the subbutterfly $BF((2, 111))$. The thick solid lines in the dashed box denote the edges of $UT((2, 111))$. The thick dotted lines denote the edges of $LT((2, 121))$. The edges between level 0 and 1 visualize nine k-blocks at level 0.

(i.e., sending a message or storing data), this action is performed by server s_i. We say a server s is **connected via the k-ary butterfly** to another server s' if there is an edge (u, v) in the butterfly such that u is emulated by s and v is emulated by s'. In order to avoid confusion, only butterfly nodes are called nodes. That is, in particular we never call a server a node. Hence, nodes are artificial entities of the systems we present, while servers are the real entities of the systems that eventually perform the actions.

Even if a server would not be connected to all servers, but only to $(k-1)\log_k n$ other servers (as we assume in Chapter 6), each server can communicate with all other servers within $\log_k n$ communication rounds using the **butterfly routing strategy**. For this purpose, note that for each butterfly node u on level $\log_k n$ and each butterfly node v on level 0 there exists a unique path from u to v of length $\log_k n$ that only uses butterfly edges. Whenever a server s aims at sending a message to a server s' it is not connected to via the k-ary butterfly, it instead routes that message from the butterfly node on level $\log_k n$ which server s emulates along the unique path to the butterfly node on level 0 that is emulated by server s'.

Note that by using this routing approach each butterfly node u at level $\ell \in \{0, \ldots, \log_k n\}$ is able to communicate with each other butterfly node $v = (x, \ell')$ at level $\ell' \in \{0, \ldots, \log_k n\}$ with $x \in \{0, \ldots, n-1\}$ in ℓ rounds by simply routing the desired message along the unique path in the k-ary butterfly

from u to the node $v' = (x, 0)$ at level 0. Since v and v' are emulated by the same server, once the message arrives at v' node v also knows the message.

In the following, we also call the routing of a message from a butterfly node u at level $\ell \in \{0, \ldots, \log_k n\}$ to a butterfly node v at a lower level $\ell' \in \{0, \ldots, \log_k n\}$, i.e., $\ell' < \ell$, **bottom-up routing**. Analogously, if v is at a higher level than u, i.e., $\ell' > \ell$, we call the routing of a message from u to v **top-down routing**.

A key aspect of the storage systems presented in this work is the usage of coding. For that purpose, we will introduce some novel distributed coding techniques (Sections 3.1, 3.2.1, and 6.2.1). However, in order to further decrease redundancy, the storage systems presented in this work will also make use of the well-known Reed-Solomon codes [RS60]. Since the focus of this work is not on coding theory, we use the Reed-Solomon codes as a black box only by making use of the following lemma.

Lemma 2.4. *Let d be a data item of length $z \in \mathbb{N}$ and let $c \in \mathbb{N}$, $q \in \mathbb{Q}$. Using Reed-Solomon codes we can encode d into a data item d' of length γz, for some constant $\gamma \geqslant 1$, such that for any partition of d' into c pieces any q-fraction of these pieces suffices to recover d.*

Proof. Using Reed-Solomon codes we can append $2t$ parity bits to d such that $t \in \mathbb{N}$ errors in the resulting code word d' can successfully be corrected [RS60]. Since we require a redundancy of γ, for $\gamma \geqslant 1$ constant, t and γ have to be chosen such that it holds: $z + 2t = \gamma z$. Furthermore, we require $t = (1 - q)c$. Hence, for γ it holds: $\gamma = 1 + 2(1 - q)/z$, which is upper bounded by a constant and therefore proofs the lemma. □

Chapter 3

Basic IRIS

In this chapter we present Basic IRIS, an information system, i.e., a system that provides a lookup functionality, but does not provide a write functionality. Hence, we assume that at the beginning the data items are already stored in the system by following the storage strategy we present in Section 3.2.

Basic IRIS is based on two articles by Eikel and Scheideler: One article is published in the the proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) [ES13]. The other one is an article in the Journal of ACM Transactions on Parallel Computing [ES15].

As already specified in Chapter 2, time is divided into periods, where each period consists of a polylogarithmic number of rounds. More specifically in this chapter we require a period to consist of $O(\log^2 n)$ rounds such that within a single period a set of lookup requests can be answered correctly. In this chapter we assume the existence of an insider adversary that may, at the beginning of each period, select at most $\gamma n^{1/\log\log n}$ servers, with $\gamma = 1/9$, for being crashed during the period. Besides the crashed servers, the adversary additionally chooses an arbitrary set of lookup requests (with at most $O(\text{polylog}(n))$ requests per non-crashed server) sent to the non-crashed servers.

Basic IRIS is scalable, efficient and robust despite the existence of an insider adversary that crashes up to $\gamma n^{1/\log\log n}$ servers. More specifically, we show the following theorem.

Theorem 3.1 (Basic IRIS Main Theorem)**.** *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/9$. Then, using only a constant redundancy, Basic IRIS correctly serves any set of lookup requests (at most $O(1)$ per intact server)*

after at most $O(\log^2 n)$ communication rounds with a congestion of at most $O(\log^3 n)$ at every server in each round, w.h.p.

Note that we present a modification of Basic IRIS in Chapter 4 with different bounds for the maximum number of crashed servers tolerated and on the redundancy needed. This system will be called Enhanced IRIS. However, we sometimes omit the term "Basic" in Basic IRIS and in that case refer to Basic IRIS, not to Enhanced IRIS.

In order to achieve a constant, redundancy we require the data items to be of size $\Omega(\log n)$, as we explain later.

The remainder of this chapter is organized as follows: First, we introduce the distributed coding strategy, called Butterfly Coding Strategy, whose framework is reused in each information/storage system presented in this work (Section 3.1). The Butterfly Coding Strategy internally requires a distributed error correcting code, which we present in Section 3.2. Furthermore, in Section 3.2 we present the overall storage strategy of Basic IRIS. In Section 3.3 we present the rather complex lookup protocol of Basic IRIS. While the running time and congestion analysis of the lookup protocol immediately follow from the protocol, the correctness analysis is fairly involved and presented in Section 3.4.

Table 3.1 provides an overview of variables and their bounds that are commonly used in this chapter.

Term	Bound	Description
γ	$= 1/9$	Constant in fraction of crashed servers from $n^{1/\log\log n}$ servers
ε	$< \gamma n^{1/\log\log n - 1}$	Fraction of crashed servers
α	$> 2(1-\varepsilon)/\gamma$, e.g., $\geqslant 9$	Constant in congestion bound in Probing Stage
β	$> 3/2$	Constant in congestion bound in Decoding Stage
c	$\geqslant 8 \log m$	Number of pieces created for each data item
m	$\geqslant n$	Size of the universe

Table 3.1: Variables commonly used in the presentation of IRIS.

3.1 Butterfly Coding Strategy

In the following we present the Butterfly Coding Strategy which is used by all information and storage systems presented in this work. This strategy describes how to encode n data blocks with each other that have already been assigned to n servers. Internally, a distributed code is used which can be

exchanged by another distributed code in order to achieve different properties for the overall Butterfly Coding Strategy. However, Basic IRIS requires the use of a distributed code that guarantees the correction of a single data block if the faulty data block is known in advance. This distributed code appends some parity bits to the input data blocks only in order to build the result. Section 3.2.1 provides a description of this code. For the rest of this section we use the internal code as a black box.

The Butterfly Coding Strategy is a block-based distributed coding strategy that hierarchically encodes sets of $k \in \mathbb{N}$ data blocks with each other. Here, the topology of the k-ary butterfly will define how to choose these sets.

In the following let $BF(k, d)$ be a k-ary butterfly with $n = k^d$ nodes, i.e., $d = \log_k n$ and $k = \log n$. Recall that in Section 2.2 we defined the server s_i, $i \in \{0, \dots, n-1\}$ as the server responsible for the $d+1$ butterfly nodes in the i-th column of $BF(k, d)$, i.e., $(0, i), \dots, (d, i)$. Furthermore, server s_i was defined to emulate the nodes it is responsible for, i.e., whenever a butterfly node $(0, i), \dots, (d, i)$ is supposed to perform an action (i.e., sending a message or storing data), this action is performed by server s_i.

In the following let $\{b_0, \dots, b_{n-1}\}$ be a set of data blocks that are supposed to be encoded with each other. We assume there exists a mapping that assigns each data block b_i to a server s_i. In Section 3.2 we describe how to implement this mapping from the data blocks to the servers. In order to encode the data blocks $b_0, \dots, b_{n-1}$ assigned to the servers $s_0, \dots, s_{n-1}$ with each other, initially, b_i is assigned to butterfly node $(0, i)$ for each $i \in \{0, \dots, n-1\}$. For a butterfly node u let $d(u)$ denote the data that is assigned to node u.

Assume we already assigned data blocks $d(u_i)$ to the butterfly nodes $u_i := (\ell, i)$, $i \in \{0, \dots, n-1\}$, at level ℓ. For each k-block at level ℓ with nodes $(\ell, i_1), \dots, (\ell, i_k), i_1, \dots, i_k \in \{0, \dots, n-1\}$ at level ℓ, encode the data blocks $d((\ell, i_1)), \dots, d((\ell, i_k))$ with each other using the internal distributed code. By this, for each server s_{i_j}, $j \in \{1, \dots, k\}$ a parity bit string p_{i_j} is computed which will be appended to $d((\ell, i_j))$. Thus, the resulting k data blocks are $d((\ell, i_1)) \circ p_{i_1}, \dots, d((\ell, i_k)) \circ p_{i_k}$. Finally, assign $d((\ell, i_1)) \circ p_{i_1}, \dots, d((\ell, i_k)) \circ p_{i_k}$ to the nodes $(\ell+1, i_1), \dots, (\ell+1, i_k)$ by setting $d((\ell+1, i_j)) = d((\ell, i_j)) \circ p_{i_j}$ for $j \in \{1, \dots, k\}$.

At the end, server s_i, $i \in \{0, \dots, n-1\}$, holds the data block b_i and all parity information that has been assigned to a butterfly node (ℓ, i), $\ell \in \{0, \dots, \log_k n\}$.

In a distributed fashion the encoding of the given n data blocks can simply be computed by a top-down approach. That is, first, for each k-block B at level 0, the nodes from B at level 0 send the data blocks assigned to them to the nodes at level 1 from B such that each node $(1, i)$, $i \in \{0, \dots, n-1\}$, at level 1 can compute $d((1, i)) = d((0, i)) \circ p_i$. Next and analogously to level 0, the nodes from level 1 send their data to the according nodes from their k-block at level 2 such that all level 2 nodes can compute the data they are supposed

to store. This approach proceeds level by level until all nodes at the last level, level $\log_k n$, have computed the data they are supposed to hold.

Note that by this approach, at the end of the encoding process at least some parity information of each data block is stored at any server.

3.2 Storage Strategy

Let $\mathcal{K} \subseteq \mathcal{U}$ be the set of all keys that have a data item in the system and define $c = 8 \log m$. The idea of storing the $|\mathcal{K}|$ data items in IRIS is to divide these data items into several n-tuples that are separately encoded with each other using the Butterfly Coding Strategy presented in Section 3.1. For this purpose we require the use of several hash functions: First, we require a hash function $L : \mathcal{U} \to \{0, \ldots, \rho \cdot |\mathcal{K}|/n\}$ for some constant γ which we call the **layer hash function**. Second, we require c further hash functions $h_1, \ldots, h_c : \mathcal{U} \to \mathcal{S}$ chosen uniformly at random.

In detail, storing a set of $|\mathcal{K}|$ data items in IRIS works as follows:

Step 1 (Reed-Solomon Coding): Use Reed-Solomon codes [RS60] to partition each data item d into $c = \Theta(\log m)$ pieces $d_1, \ldots, d_c$ such that any $c/4$ of these pieces of d suffices to recover d.

Step 2 (Mapping to Servers): Use the layer hash function $L : \mathcal{U} \to \{1, \ldots, \rho \cdot |\mathcal{K}|/n\}$ in order to assign each data item to a **layer**. Afterwards, for each data item d, assign each piece d_i, $i \in \{1, \ldots, c\}$, of d to a server using hash function $h_i : \mathcal{U} \to \mathcal{S}$. We denote the concatenation of data pieces that are assigned to server s_j and layer α as $b_j(\alpha)$. If it is clear from the context we omit the α and write b_j instead.

Step 3 (Encoding Layers): For each layer $j \in \{1, \ldots, \rho \cdot |\mathcal{K}|/n\}$ encode the data blocks $b_0(j), \ldots, b_{n-1}(j)$ with each other using the Butterfly Coding Strategy presented in Section 3.1. Recall that the Butterfly Coding Strategy requires an additional internal code which was used as a black box in Section 3.1. In Section 3.2.1 we present an according code used in Basic IRIS.

For simplicity, we assume that the data blocks $b_0(j), \ldots, b_{n-1}(j)$ are of the same size z. If this is not the case, we could simply append dummy bits to the data blocks such that all data blocks are finally of the same size.

The layer hash function L has to be implemented in such a way that $\Theta(n)$ data items are mapped to each of the $\rho \cdot |\mathcal{K}|/n$ layers. For the hash functions $h_1, \ldots, h_c$ we require that for each layer $\Theta(c)$ data pieces are mapped to each server. The simplest way of realizing L and $h_1, \ldots, h_c$ with these requirements is to use cuckoo hashing [PR04] where $\rho = 2$ suffices: i.e., each data item has

two optional layers where it is supposed to be stored and the data items are distributed among these positions such that each of the $\rho \cdot |\mathcal{K}|/n$ layers holds $\Theta(n)$ data items in total. Analogously, each data piece has two optional servers that are supposed to hold it and the data pieces are distributed among the servers such that each server holds $\Theta(c)$ data pieces in total. Of course, in this case a lookup request for some data item d would involve looking at the two optional layers and for each piece of d at the two optional servers. However, this would just double the work spent for the lookup operation described in Section 3.3, so in the following we just assume that L and $h_1, \ldots, h_c$ are injective hash functions that can be directly evaluated in order to determine the unique layer and server for a data piece.

3.2.1 Internal Distributed Error Correcting Code

In the following we present a simple error correcting code that is able to recover one out of $k \in \mathbb{N}$ data blocks by only using some simple parity computations and appending the resulting parity bits to the original data blocks. In other words, assume k servers holding one data block each while exactly one of these servers is crashed. Then, the code we present guarantees that the data block held by the crashed server can be recovered using only the information of the remaining $k-1$ servers. In detail, the encoding of k data blocks $b_1, \ldots, b_k$ works as follows:

1. Compute $P = b_1 \oplus b_2 \oplus \ldots \oplus b_k$ where "$\oplus$" is the bit-wise parity operation.
2. Cut P into $k-1$ pieces $p_1, \ldots, p_{k-1}$ of equal size (up to an additive 1).
3. Set $b'_i = b_i \circ p_i$, $i \in \{1, \ldots, k\}$, where "$\circ$" is the concatenation operator.
4. Set $b'_k = b_k \circ p_k$ with $p_k = p_1 \oplus p_2 \oplus \cdots \oplus p_{k-1}$.

See Figure 3.1 for a visualization of the encoding of k data blocks.

For this code the following lemma can be shown.

Lemma 3.2. *Let the data blocks $b_1, \ldots, b_k$ be encoded with each other using the previously described coding strategy resulting in $b'_1, \ldots, b'_k$. Then, if one b'_j, $j \in \{1, \ldots, k\}$, is inaccessible (e.g., the server holding it is crashed), the information in $b'_1, \ldots, b'_{j-1}, b'_{j+1}, \ldots, b'_k$ suffices to recover b'_j.*

Proof. Suppose data block b'_j is inaccessible. By definition of the error correcting coding strategy we can compute $p_j = \bigoplus_{i \neq j} p_i$. This allows us to recover P by computing $P = p_1 \circ \ldots \circ p_{k-1}$. With this we can recover b_j by computing $b_j = P \oplus (\bigoplus_{i \neq j} b_i)$. □

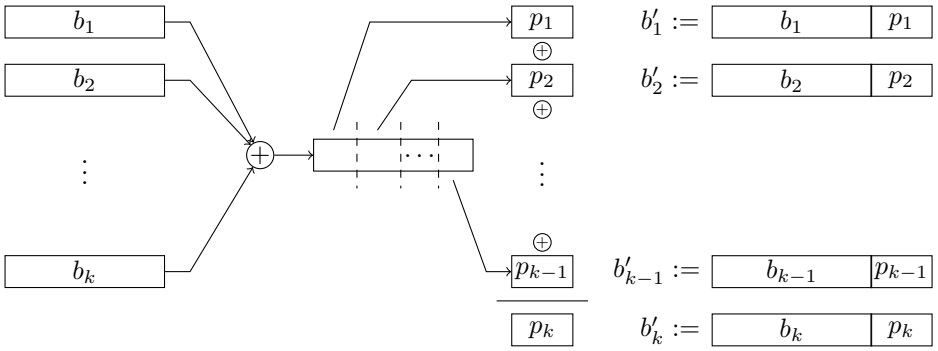

Figure 3.1: Visualization of the encoding of k data blocks $b_1, \ldots, b_k$.

3.2.2 Redundancy Analysis

In this section we analyze the redundancy required by the storage strategy of Basic IRIS. For a data block b let $|b|$ denote the size of the data block b, i.e., the number of bits in the binary representation of b. For a node v in $BF(k, d)$, let $|d(v)|$ denote the size of the data stored in node v. Since to each node $(\ell + 1, x_i)$ a bit string of size $|d(\ell, x_i)|/(k-1) \pm 1$ is appended, the following lemma holds.

Lemma 3.3. *For any k-block B with nodes $(\ell, x_1), \ldots, (\ell, x_k)$ and $(\ell+1, x_1), \ldots, (\ell+1, x_k)$, it holds $|d(\ell + 1, x_i)| \leqslant (1 + 1/(k-1))|d(\ell, x_i)|$ up to an additive 1.*

For simplicity, in the following we ignore the additive 1 due to the fact that $|d(\ell, x_i)|$ may not be perfectly divisible by $k-1$. This will only cause a constant factor deviation from the bounds below as long as the original data items have a size of z with $z \geqslant k$.

In the following we examine the redundancy caused by the storage strategy described in Section 3.2. For this purpose we first consider the redundancy that occurs if at most n data items are stored in the system.

With $d(s_i) = d_i \circ p_1(i) \circ \ldots \circ p_{\log_k n}(i)$ we get the following lemma.

Lemma 3.4. *For any $k > d$ and $|d_i| \geqslant k$ it holds that $|d(s_i)| \leqslant (1 + e)|d_i|$ for every server s_i.*

Proof. Notice that for the ℓ-th parity bit string $p_\ell(i)$ appended to d_i it holds: $|p_\ell(i)| \leqslant (1 + 1/(k-1))^{\ell-1}|d_i|/(k-1)$ for all $\ell \in \{1, \ldots, \log_k n\}$, which can be shown by induction on ℓ.

By definition of $d(s_i)$ for each server s_i it holds:

$$|d(s_i)| = |d_i| + \sum_{j=1}^{d} |p_j(i)| \leqslant |d_i| + \sum_{\ell=1}^{d} \left(1 + \frac{1}{k-1}\right)^{\ell-1} \cdot \frac{|d_i|}{k-1}$$

With $(1+x) \leqslant e^x$ for all $x \geqslant 0$ we get:

$$|d(s_i)| \leqslant |d_i| + \sum_{\ell=1}^{d} e^{(\ell-1)/(k-1)} \cdot \frac{|d_i|}{k-1} \leqslant |d_i| \left(1 + \frac{d \cdot e^{(d-1)/(k-1)}}{k-1}\right)$$

Since $k > d = \log_k n$ this term is upper bounded by $(1+e)|d_i|$, which proves the lemma. □

Notice that $d = \log_k n = \log n / \log\log n$. Hence, in order to ensure $k > d$, it must hold $k > \log n / \log\log n$. For $n > 4$ it holds $\log n > \log n / \log\log n$. Thus, with $k = \log n$ and data items of a size of at least k, we achieve a storage strategy with a constant redundancy only.

Since we assume the size of the universe $\mathcal{U}$ to be polynomial in the number of servers, the number of layers required is constant. Furthermore by Lemma 2.4, the redundancy required for partitioning a data item d into c pieces via Reed-Solomon codes such that any $c/4$ pieces suffice to recover d is also constant.

Hence, in order to store $O(n)$ data items each of size z into a layer, we first create for each of these data item $c = \Theta(m)$ pieces, that are distributed among the servers such that at the end each server holds c pieces, i.e., a data block of size $\Theta(z)$. Then, the n-tuple of these data blocks is encoded with each other resulting in $\Theta(n)$ data blocks each of size $\Theta(z)$ if $z > \log n$ (Lemma 3.4).

Hence, for the redundancy of Basic IRIS it follows:

Corollary 3.5. *Storing poly(n) data items, each of size $\Omega(\log n)$, in Basic IRIS requires a constant redundancy.*

3.3 Lookup Protocol

The lookup protocol describes which actions the intact servers perform in order to serve the lookup requests they received. Recall that, due to the Reed-Solomon coding, for each intact server with a request for some data item d it suffices to retrieve $c/4$ pieces of d in order to recover d and thereby correctly answer the request.

The naïve approach in which each server with a request for a data item d simply asks the servers $s_1, \ldots, s_c$ that hold the c pieces of d does not work for the following reasons: First of all, all the servers $s_1, \ldots, s_c$ could be crashed, which disables them from answering any requests. In another scenario the adversary could have sent a lookup request for the same data item d to all intact servers. In this case each intact server would contact the same servers $s_1, \ldots, s_c$, causing these servers to become congested, i.e., receiving more than polylog(n) many messages.

Hence, we need a cleverer strategy to serve the lookup requests. For that purpose, we divide the lookup protocol into three stages: a **Preprocessing Stage**, a **Probing Stage**, and a **Decoding Stage**. In the Preprocessing Stage, the intact servers determine a unique representative for each crashed server so that we can route in the k-ary butterfly as if all servers are still intact (but, of course, the data in the crashed servers is unavailable). Also, information is collected that allows us to bound the work of decoding specific pieces of data items. In the Probing Stage, we issue lookup requests for the c pieces of each requested data item d. If we retrieve sufficiently many of these pieces without causing an excessive congestion at any node we can answer the request for d. Otherwise, the request for d will further be handled in the Decoding Stage in which sufficiently many pieces of d will be recovered in order to finally recover d itself.

Note that stages similar to the Probing Stage and to the Decoding Stage are also used in the Lookup Protocol of the storage and information systems by Scheideler et al. [AS07; BSS09]. However, the determination of the representatives and the decoding depth computation in the Preprocessing Stage cannot be found in [AS07; BSS09]. At the end of the presentation of the Lookup Protocol, in Section 3.3.4, we point out the differences and similarities between the Probing Stage and the Decoding Stage of this work and the according stages in [AS07; BSS09].

Since the correctness analysis of the Probing Stage and the Decoding Stage is rather involved and we want to keep the description of the lookup protocol as clear as possible, we postpone this analysis to Section 3.4.

3.3.1 Preprocessing Stage

The Preprocessing Stage is divided into two further substages: the **Butterfly Completion Stage** and the **Decoding Depth Computation Stage**.

3.3.1.1 Butterfly Completion Stage

The goal of the Butterfly Completion Stage is to determine for each crashed server a unique representative chosen from the intact servers. The task of the representatives is to take over the role of the crashed servers, e.g., by forwarding messages or performing computations that were initially supposed to be sent or performed by the according crashed servers. That is, via the representatives the servers can route messages through the k-ary butterfly as if no server was crashed. Note that a representative can only act as the crashed server it represents, but it does not have access to the crashed server's data. Once an intact server s' becomes the representative of a crashed server s, it has still to be ensured that each other intact server that is connected to s via the

underlying k-ary butterfly knows the representative s' of s. In the rest of the lookup protocol, whenever a server s is supposed to contact a crashed server s', s contacts the representative of s' instead. Once, the representatives have been determined and each server is aware of the representative of any crashed server it is connected to via the k-ary butterfly, we do not require the servers to form a clique any more. Instead, the connections formed for the k-ary butterfly suffice for the rest of the protocol.

In short, the Butterfly Completion Stage consists of the following phases:

Step 1: Build a tree T of depth $O(\log n / \log\log n)$ over all intact servers.

Step 2: Transform the constructed tree T into a doubly-linked list L consisting of n intact servers in which each intact server is contained either once or twice.

Step 3: Rearrange the created list L such that each intact server with identifier i is at position i in L and for each crashed server with identifier j there is an intact server at position j in L that is declared the **representative** of the crashed server.

Step 4: Transform the resulting list into an auxiliary k-ary butterfly that only consists of intact servers and where each intact server s that is connected to a crashed server s' in the k-ary butterfly consisting of all servers is in the auxiliary butterfly connected to the representative of s'.

We now give a detailed description of the previously mentioned phases.

Phase 1: Build a tree over the intact servers In order to build a tree of depth $O(\log n / \log\log n)$, we first build a graph of degree $O(\log n)$ and diameter $O(\log n / \log\log n)$, w.h.p., consisting of all intact servers. Afterwards, this graph is transformed into a tree of depth $O(\log n / \log\log n)$ using the technique of a breadth-first search. Recall that $c = O(\log m)$ and $m \geqslant n$.

The graph is constructed as follows: First, each intact server s chooses $\Theta(c)$ other servers uniformly at random. By Lemma 2.2, each server s chooses c intact servers $s_1, \ldots, s_c$, w.h.p. Afterwards, s creates an edge to each server $s_1, \ldots, s_c$ and additionally informs each of those servers about that, implying the servers $s_1, \ldots, s_c$ to also create an edge to s.

Note that the process of randomly choosing $\Theta(c)$ servers for each intact server is the only process in the Lookup Protocol in which we require the servers to initially form a clique. If we would not require the servers to initially form build a clique but only to be connected via a k-ary butterfly, it may happen that an intact server s is only connected to crashed servers via the k-ary butterfly.

But i n that case server s can not participate in the construction of the initial tree over the intact servers.

Lemma 3.6. *After $O(1)$ rounds the intact servers have built a graph with degree $O(\log n)$ and diameter $O(\log n / \log\log n)$, w.h.p.*

Proof. Let G be the graph created by the intact servers via the described procedure. From Lemma 2.2 it follows that G has a degree of $\Theta(\log n)$.

In order to show that the maximum distance between any two nodes is at most $O(\log n/\log\log n)$, we first show that each set U of at most $n/(2d)$ nodes from G (with $d = \Theta(\log n)$ being the degree of U) has a good expansion. To be more precise, we show that for all sets U of at most $n/(2d)$ nodes from G for the set of neighboring nodes $\Gamma(U)$ of U it holds:

$$|\Gamma(U)| \geqslant \delta \cdot d \cdot |U| \tag{3.1}$$

with $0 < \delta \leqslant 1/2$ constant. In order to prove this, we first show that the probability that a given set U of k nodes does not have a good expansion is small. That is, we need to show that the probability that less than a δ-fraction of all edges of the nodes from U are incident to a node that is not in U is small. This probability again equals the probability that at least a $(1-\delta)$-fraction of all edges from U are incident to a node from U, which is

$$\sum_{i=(1-\delta)kd}^{kd} \binom{kd}{i} \left(\frac{k}{n}\right)^i \leqslant kd \cdot \binom{kd}{(1-\delta)kd} \left(\frac{k}{n}\right)^{(1-\delta)kd} = kd \cdot 2^{kd} \cdot \left(\frac{k}{n}\right)^{(1-\delta)kd}$$

Hence, the probability that there exists any set U of at most $n/(2d)$ nodes from G that does not fulfill equation (3.1) is upper bounded by

$$\sum_{k=1}^{n/(2d)} \binom{n}{k} \cdot kd \cdot 2^{kd} \cdot \left(\frac{k}{n}\right)^{(1-\delta)kd} \leqslant \sum_{k=1}^{n/(2d)} kd \cdot \left(\frac{en}{k} \cdot 2^d \cdot \left(\frac{k}{n}\right)^{(1-\delta)d}\right)^k$$

Using $1 \leqslant k \leqslant n/(2d)$ we can upper bound this term by

$$\frac{n^2}{4d} \cdot \left(en \cdot 2^d \cdot \left(\frac{1}{2d}\right)^{(1-\delta)d}\right)^{n/(2d)}$$

Since $d = \Theta(\log n)$, for n sufficiently large this term is upper bounded by

$$\frac{n^2}{4\log n} \cdot \left(en^2 \cdot \left(\frac{1}{2n^{\log\log n+1}}\right)^{1-\delta}\right)^{n/(2d)} \tag{3.2}$$

For any constant $x > 0$ and n sufficiently large equation (3.2) is upper bounded by $\Theta(1/n^x)$ implying equation (3.1).

Hence, as long as U consists of at most $n/(2d)$ nodes, its neighborhood grows by a factor of at least δd. In particular, for n sufficiently large, $n/(2d)$ distinct nodes can be reached from a node u in at most h hops with h satisfying

$$\begin{aligned}&(\delta \log n)^h = n/(2\log n)\\ \Leftrightarrow\ & h = \log_{\delta \log n}(n/(2\log n)) = \frac{\log n - \log\log n - \log 2}{\log \delta + \log\log n}\end{aligned}$$

Thus, for n sufficiently large, from any node u we can reach $n/(2d)$ many nodes in $\Theta(\log n/\log\log n)$ hops. Since for any set of nodes U with $|U| = n/(2d)$ it holds $\Gamma(U) = (\delta/2)n$, we get that $\Theta(n)$ nodes are at distance $\Theta(\log n/\log\log n)$ from any node u.

It remains to show that $|\Gamma(U)| = n$ for any set U of $\delta' n$ nodes from G with $0 < \delta' \leqslant 1$ and degree $d = \Theta(\log n)$, w.h.p. Since the nodes from U are incident to at most $d\delta' n$ edges in total, the probability that none of these edges is incident to one fixed node from G is upper bounded by $(1-1/n)^{d\delta' n}$. Hence, the probability that there exists a node that is neither in U nor it is adjacent to a node from U, is upper bounded by

$$n \cdot \left(1 - \frac{1}{n}\right)^{\delta d n} \leqslant n \cdot \frac{1}{e^{\delta d}} \leqslant \frac{1}{n^{\delta x - 1}}$$

where we used $d = x \log n$ for $x > 2$ constant, n sufficiently large and $(1 - 1/n)^n \leqslant 1/e$. Hence, for any set U of $\Theta(n)$ nodes from G the neighborhood of U consists of all nodes from G, w.h.p.

□

In order to transform the constructed graph G consisting only of intact servers into a tree of depth $O(\log n/\log\log n)$, each intact server initiates a breadth-first search (BFS). The idea is to let the servers create a tree that is rooted at the intact server with minimum ID among all intact servers. Since the intact servers cannot determine this ID in advance in at most polylogarithmic time, each intact server initiates a BFS. In the following each intact server s holds three variables $\mathsf{minDist}(s)$, $\mathsf{minDistSource}(s)$, and $\mathsf{parent}(s)$ that are initialized with 0, $\mathsf{id}(s)$ and NIL, respectively. $\mathsf{minDistSource}(s)$ will hold the minimum server ID from which s has received a message so far. $\mathsf{minDist}(s)$ will hold the minimum distance to $\mathsf{minDistSource}(s)$ that server s has stored so far. $\mathsf{parent}(s)$ will hold the server ID from which it has received the last message that initiated an update of $\mathsf{minDistSource}(s)$. In the first round, each intact server s sends the message $(\mathsf{id}(s), \mathsf{minDist}(s), \mathsf{minDistSource}(s))$ to each

of its neighbors in G. Algorithm 1 describes the actions performed by each intact server s in each of the following rounds as soon as s has received all messages from its neighbors.

Algorithm 1 GRAPHTOTREE performed by intact server s

```
1: for all messages (id, minDist, minDistSource) received at the beginning of
   this round do
2:     if minDistSource < minDistSource(s)
         ↳ or minDistSource = minDistSource(s)
         ↳ and minDist < minDist(s) then
          ▷ update internal variables
3:         minDistSource(s) ← minDistSource
4:         minDist(s) ← minDist + 1
5:         parent(s) ← id
6: for all neighbors s' of s in G do
7:     Send message (id(s), minDistSource(s), minDist(s)) to s'.
```

Together with Lemma 3.6 and for $\varepsilon < \gamma n^{1/\log\log n - 1} < 1/2$ it follows:

Lemma 3.7. *After $O(\log n / \log\log n)$ rounds it holds, w.h.p.:*

1. *The parent(s)-values induce a tree T of depth $O(\log n / \log\log n)$ over all $(1-\varepsilon)n$ intact servers rooted at the intact server with minimum identifier.*
2. *The degree of each node in T is at most $O(\log n)$.*

Phase 2: Transform list to tree Next we show how to transform T into a doubly-linked list L of n intact servers in which each intact server is contained at most twice. First, using a bottom-up approach, each intact server s determines for each of its children s' in T the size size(s') of the subtree of T rooted at s' and the identifier of the rightmost server rightmost(s') in this subtree and reports it to its parent server. Obviously, this is possible in $O(depth(T))$ rounds. Using this information, in a top-down approach, each intact server then determines its position and its neighbors in a doubly-linked list of n intact servers as follows: First, the root r of T (i.e., the server r with $p(r) = NULL$) initiates a pre-order walk of T by performing Algorithm 2 with parameters $1, NULL$. Whenever a server receives a message, it also performs Algorithm 2. Clearly, after at most $depth(T)$ rounds each intact server knows its position and its left neighbor in L (see Figure 3.2 for a visualization). In order to transform L into a *doubly*-linked list, each intact server s sends its ID to its left neighbor and sets right(s) to the ID it receives, or to $NULL$ if it does not receive a message. In parallel to the transformation of T into a doubly-linked list of

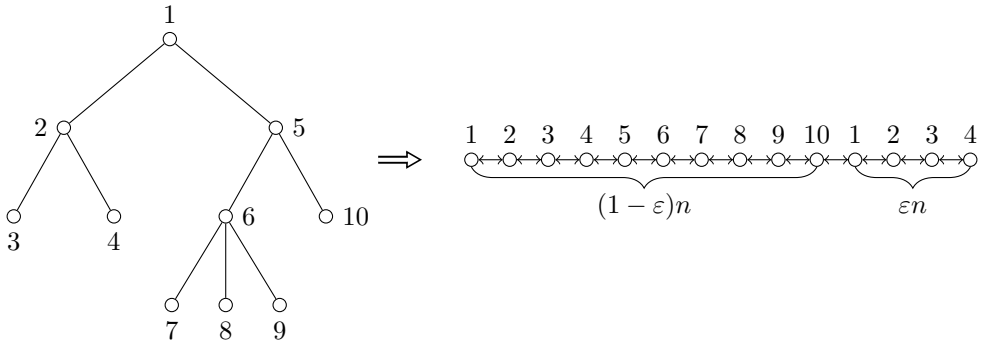

Figure 3.2: Transformation of T into a sorted list of n intact servers. The numbers next to the tree nodes denote the order of their appearance in the tree traversal.

the $(1-\varepsilon)n$ intact servers, s initiates an additional pre-order traversal of T by additionally performing Algorithm 2 with parameters $(1-\varepsilon)n+1$, $\mathsf{rightmost}(r)$. In case ε is not known to the servers, r can simply determine it by computing $\varepsilon = n - \sum_{i=1}^{p} \mathsf{size}(\mathsf{child}_i(r))$, where $\mathsf{child}_1(r), \ldots, \mathsf{child}_p(r)$ denote the children of r in T. In contrast to the first traversal of T, the values $\mathsf{left}(s)$ and $\mathsf{pos}(s)$ in Algorithm 2 are now substituted by $\mathsf{left}_2(s)$ and $\mathsf{pos}_2(s)$. Additionally, we use the modification that as soon as an intact server sets its pos_2 value to n, the algorithm terminates. Then, analogously to the right values, each server sets its right_2 value. By this additional tree traversal the first εn servers of the traversal are appended to L. Notice that this pre-order traversal of T guarantees that the first εn servers visited form a connected subtree of T.

Algorithm 2 BUILDLISTFROMTREE(x, l) performed by intact server s

```
1: left(s) ← l, pos(s) ← x
2: for all children child_i(s) of s, i ∈ {1,...,p} do
3:     if i = 1 then
4:         left ← child_i(s)                    ▷ child_i(s) is the leftmost child of s
5:     else
6:         left ← rightmost(child_{i-1}(s))
7:     pos ← pos(s) + 1 + Σ_{j=1}^{i-1} size(child_j(s))
8:     Send message (pos, left) to child_i(s).
```

With ε denoting the maximum fraction of crashed servers allowed, the following lemma holds.

Lemma 3.8. *After $2 \cdot depth(T)$ rounds, T is transformed into a doubly-linked list L of size n over the $(1-\varepsilon)n$ intact servers such that each intact server is contained at most twice in L.*

Phase 3: Rearrange list The goal of this phase is to rearrange L into a doubly-linked list with the properties specified in Lemma 3.9.

Lemma 3.9. *After $O(1)$ rounds it holds:*

1. *Each intact server s_i is at position i in L.*
2. *For each crashed server s_j, the server s' with $\mathsf{pos}(s') = j$ or $\mathsf{pos}_2(s') = j$ is the unique representative of s_j.*

Initially, the **owner** of a position j in L is the server s' with $\mathsf{pos}(s') = j$ or $\mathsf{pos}_2(s') = j$.

First, each intact server s' at position j contacts server s_j. If s_j is not crashed, then s' considers s_j as the new owner of j. s' then asks its direct neighbors in L for the owners of the positions $j-1$ and $j+1$ and forwards that information to s_j so that s_j can take over the position j in L. If s_j is crashed, s' remains to be the owner of j and therefore becomes the representative of s_j.

Phase 4: Build extended k-ary butterfly In this phase L is transformed into a k-ary butterfly using an extended k-ary butterfly.

Definition 3.10 (Extended k-ary Butterfly)**.** *For any $d, k \in \mathbb{N}$, the d-**dimensional extended k-ary butterfly** $EBF(k,d)$ is a graph (V,E) with $V = \bigcup_{\ell=0}^{d} V_\ell$ and $E = \bigcup_{\ell=1}^{d} E_\ell$ where*

$$V_\ell = \{(\ell, i) \mid i \in \{0, \dots, n-1\}\} \text{ and}$$
$$E_\ell = \{\{(\ell-1, i), (\ell, j)\} \in V_{\ell-1} \times V_\ell \mid \exists x \in \{0, \dots, k-1\} : |i-j| = x \cdot k^{\ell-1}\}.$$

*Furthermore, we define $G(\ell) = (V_{\ell-1} \cup V_\ell, E_\ell)$, $\ell \in \{1, \dots, d\}$, and call each node (ℓ, i) a **level ℓ node**.*

See Figure 3.3 for a (partial) visualization of an extended k-ary butterfly.

Recall that in the standard k-ary butterfly (see Definition 2.3) the nodes from level $\ell-1$ and ℓ, $\ell \in \{1, \dots, \log_k n\}$, form only n/k disjoint complete k-bipartite subgraphs which are also contained in $EBF(k,d)$. To be more precise, each block of k consecutive nodes between level 0 and level 1 (k nodes on each level) forms a complete k-bipartite subgraph. Each block of k nodes between level $\ell-1$ and ℓ with a distance of $k^{\ell-1}$ between each two nodes forms a complete k-bipartite graph.

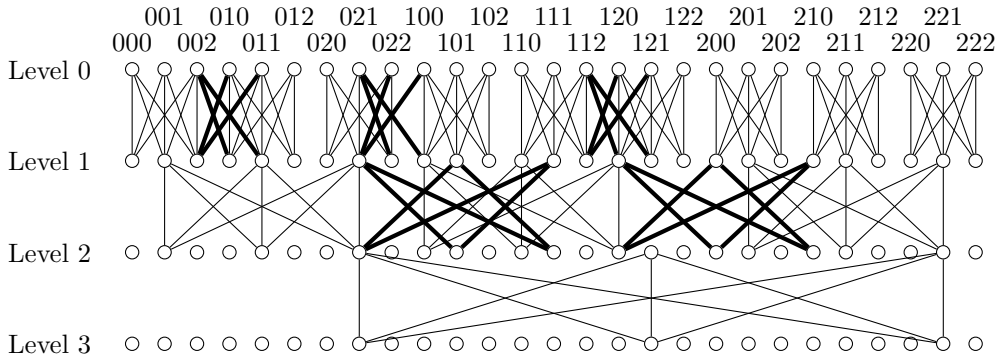

Figure 3.3: Visualization of $EBF(k, d)$ for $k = 3$ and $d = 2$. The thin edges denote edges that are contained in both the standard k-ary butterfly and the extended k-ary butterfly. The solid edges denote edges that are exclusively contained in the extended k-ary butterfly. In order to keep the visualization clear, most of the edges from the standard and extended k-ary butterfly are omitted.

The idea of this phase is to add for each server at position i in L exactly $\log_k n + 1$ virtual nodes $(0, i), \ldots, (\log_k n, i)$ and to successively build the graphs $G(\ell)$, $\ell = 1, \ldots, \log_k n$, beginning with $G(1)$.

In $G(1)$ each node from level 0 needs to connect to all nodes on level 1 that are at distance at most $k - 1$. For this, in the first round, each intact server s asks its two direct neighbors in L for their neighbors in L which introduces s to its neighbors at distance 2 in L. In round $r \in \{2, \ldots, \lceil \log(k-1) \rceil\}$ each intact server s asks its two neighbors at distance 2^{r-1} for their neighbors (at distances $1, 2, \ldots, 2^{r-1}$) in L. This introduces s to all servers at a distance of at most $2 \cdot 2^{r-1} = 2^r$ in L. Hence, after $\lceil \log(k-1) \rceil$ rounds, each intact server knows all servers at a distance of at most $2^{\lceil \log(k-1) \rceil} = k - 1$ in L.

The construction of $G(\ell)$, $\ell = 2, \ldots, \log_k n$, proceeds in $\lceil \log(k-1) \rceil + 1$ rounds and assumes $G(\ell - 1)$ has already been built. That is, each intact server knows all servers at distance $x \cdot k^{\ell-2}$, $x \in \{1, \ldots, k-1\}$. In order to build $G(\ell)$, each intact server needs to be introduced to all servers at distance $x \cdot k^{\ell-1}$, $x \in \{1, \ldots, k-1\}$. In the first round each intact server s asks its neighbors at distance $(k-1)k^{\ell-2}$ (i.e., the servers farthest away in $G(\ell - 1)$) for their closest neighbors in $G(\ell - 1)$ (i.e., their neighbors at distance $k^{\ell-2}$). By this, s is introduced to the servers at distance $(k-1)k^{\ell-2} + k^{\ell-2} = k^{\ell-1}$. In round $r \in \{2, \ldots, \lceil \log(k-1) \rceil + 1\}$ each intact server s already knows all servers at distance $x \cdot k^{\ell-1}$, $x \in \{1, \ldots, 2^{r-2}\}$ and asks its neighbors at maximum distance (i.e., at distance $2^{r-2} \cdot k^{\ell-1}$) for their neighbors at distance $x \cdot k^{\ell-1}$,

$x \in \{1, \ldots, 2^{r-2}\}$. This introduces s to all servers at the following distances:

$$\begin{aligned}
2^{r-2} \cdot k^{\ell-1} + k^{\ell-1} &= (2^{r-2} + 1) \cdot k^{\ell-1} \\
2^{r-2} \cdot k^{\ell-1} + 2k^{\ell-1} &= (2^{r-2} + 2) \cdot k^{\ell-1} \\
2^{r-2} \cdot k^{\ell-1} + 3k^{\ell-1} &= (2^{r-2} + 3) \cdot k^{\ell-1} \\
&\vdots \\
2^{r-2} \cdot k^{\ell-1} + 2^{r-2} \cdot k^{\ell-1} &= (2^{r-2} + 2^{r-2}) \cdot k^{\ell-1} = 2^{r-1} \cdot k^{\ell-1}
\end{aligned}$$

Hence, after round r, s knows all neighbors at distance $x \cdot k^{\ell-1}$ for all $x \in \{1, \ldots, 2^{r-1}\}$. See Figure 3.4 for a visualization. Thus, after $\lceil \log(k-1) \rceil + 1$ rounds, each intact server knows all servers at distance $x \cdot k^{\ell-1}$ for all $x \in \{1, \ldots, (k-1)\}$.

Lemma 3.11. *In the fourth phase of the Butterfly Completion Stage, a sorted list of n intact servers is correctly transformed into an extended k-ary butterfly in time $(2 + o(1)) \log n$ and at any time the congestion at every intact server is at most $O(\log n)$.*

Proof. By induction on ℓ it is easy to show that $G(\ell)$, $\ell \in \{1, \ldots, \log_k n\}$, is built correctly. Since the construction of each $G(\ell)$ takes $2(\lceil \log k \rceil + 1)$ rounds (each round described above actually consists of two rounds) the extended k-ary butterfly is built after $2(\lceil \log k \rceil + 1) \log_k n$ rounds. By Lemma 3.8, each intact server is contained at most twice in L, implying that in each round each intact server contacts (and is contacted by) at most four intact servers and asks for their neighbors in $G(\ell - 1)$ and $G(\ell)$, respectively. Since each intact server has at most $O(k)$ neighbors in $G(\ell - 1)$ and $G(\ell)$, the congestion of each intact server is at most $O(k)$ in each round. □

Since the d-dimensional k-ary butterfly is a subgraph of the d-dimensional extended k-ary butterfly, Lemma 3.12 follows.

Lemma 3.12. *The Butterfly Completion Stage of the Lookup Protocol of Basic IRIS guarantees that after $(2+o(1)) \log n$ rounds and with a congestion of at most $O(\log n)$ at each intact server it holds:*

- *For each crashed server a unique intact server as its representative is determined.*
- *Each server that is connected to a crashed server s via the standard k-ary butterfly knows the representative of s.*
- *Each intact server is the representative of at most one crashed server.*

Note that the additional edges of the extended k-ary butterfly are used only for the construction of the standard k-ary butterfly. Hence, once the standard

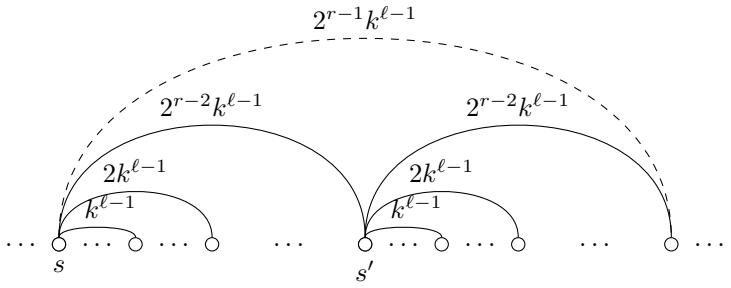

Figure 3.4: Visualization of the construction of $G(\ell)$ in round r in which the intact server s asks the server s' at distance $2^{r-2}k^{\ell}$ for its neighbors. The dashed edge denotes a connection with a maximum distance that s builds in round r.

k-ary butterfly has been established, the servers may omit the connections to the servers they are connected to in the extended k-ary butterfly but not in the standard k-ary butterfly.

3.3.1.2 Decoding Depth Computation

Once the k-ary butterfly has been re-established over intact servers, we can go ahead with collecting additional information. In particular, we are interested in the decoding work for specific data items. This is determined with the help of the following recursively defined function:

Definition 3.13 (Decoding Depth)**.** *For a node $u = (\ell, x)$ of $BF(k, d)$ the **decoding depth** $dd(u)$ is defined as:*

$$dd(u) = \begin{cases} 0 & \text{if } u \text{ is not crashed} \\ \infty & \text{if } \ell = d \text{ and } u \text{ is crashed} \\ \max\limits_{v \in C(u)} \{dd(v)\} + 1 & \text{if } \ell < d \text{ and } u \text{ is crashed} \end{cases}$$

*where $C(u)$ denotes the set of children of u in $LT(u)$, excluding one child with a greatest decoding depth among these children. The **decoding depth of a server** s_i is defined as $dd(s_i) = dd((0, i))$, and the **decoding depth of a subbutterfly** $BF(u)$ is defined as $dd(BF(u)) = \max\limits_{(0,x) \in BF(u)} dd((0, x))$.*

See Figure 3.5 for a visualization.

Note that $\mathrm{dd}(u) \in \{0, \ldots, \log_k n, \infty\}$. The decoding depth $\mathrm{dd}(u)$ of a butterfly node u immediately implies an asymptotical upper bound on the time needed for restoring the data of a crashed server. In this context note that the data

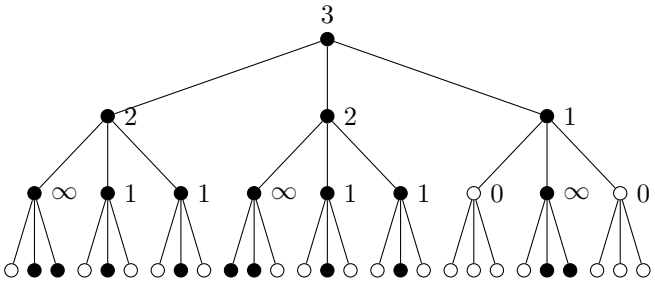

Figure 3.5: Visualization of decoding depth computation with some nodes and edges of the k-ary butterfly omitted. Black/white colored nodes represent crashed/intact nodes. The labels next to the nodes denote their decoding depth at the corresponding level.

stored at any butterfly node u at level ℓ with $\mathrm{dd}(u) \leqslant \log_k n - \ell$ can simply be recovered via a bottom-up information forwarding through the k-ary butterfly.

Lemma 3.14. *If $dd(s_i) = \ell \in \{0, \ldots, \log_k n\}$ for a crashed server s_i, then any data item that has been assigned to s_i can be recovered in time $O(\ell)$ by the nodes in $BF((\ell, i))$.*

In particular, Lemma 3.14 implies that the data stored at any server s with $\mathrm{dd}(s) < \infty$ can be recovered in time $O(\log_k n)$.

In a distributed fashion the decoding depth is computed as follows: Starting from level $\log_k n$, the servers compute the $\mathrm{dd}(u)$-values of the butterfly nodes level by level and disseminate them among their neighbors in the next lower level until the $\mathrm{dd}(\cdot)$-values of all nodes have been computed. This can certainly be done in $O(\log_k n)$ communication rounds with congestion $O(k)$ in each round. At the end every server s_i knows $\mathrm{dd}(s_i)$. Then, the servers compute the $\mathrm{dd}(BF(\cdot))$-values level by level in a way that, starting in level 0, each node u sends its $\mathrm{dd}(BF(u))$-value to all of its neighbors v in the next higher level, which will then be able to determine their $\mathrm{dd}(BF(v))$-value by taking the maximum of the received values. Hence, at the end every node u (resp. the server emulating u) knows $\mathrm{dd}(BF(u))$. This process also takes $O(\log_k n)$ communication rounds with congestion $O(k)$ in each round.

With Lemma 3.12 it follows:

Lemma 3.15. *The Preprocessing Stage takes at most $(2 + o(1)) \log n$ communication rounds with at most $O(\log^2 n)$ congestion at every intact server at each round.*

3.3.2 Probing Stage

With the Probing Stage, the actual lookup for the requested data items begins. Before we go into the details of the Probing Stage we provide a short overview

of the actions to be performed. In order to provide a precise overview of the protocol for the Probing Stage, we additionally present the algorithms in pseudocode.

Overview The idea of the Probing Stage is to forward a lookup request for each data piece d_i of a requested data item d along c paths from level $\log_k n$ to level 0 in the k-ary butterfly. The goal of this probing is to determine up to which level the c requests (also called probes) can be routed without a node on the paths becoming congested and without exceeding the decoding depth of one of the nodes on these paths. If enough probes reach level 0, the corresponding nodes can return the requested pieces to the server that issued the requests. This server can in turn recover the requested data item using the returned pieces. Otherwise, i.e., in case not enough probes reached level 0, the request for d will be assigned to a level $\ell \in \{1, \ldots, \log_k n\}$ at which not too many probes failed. Such a request will be further handled in the Decoding Stage.

For the routing of messages through the k-ary butterfly, we use the technique of splitting and combining. That is, whenever multiple messages with the same destination are supposed to be forwarded by a node v, then v combines all these messages into one message, memorizes the senders of the messages as the origins of it, and forwards that message towards its destination. As soon as the node v receives an answer to the previously sent message, it "splits" the answer by forwarding it to all of the origins of that message. See Figure 3.6 for a visualization of the technique of combining and splitting.

Details At the beginning of the Probing Stage, each intact server s that received a lookup request for some data item d chooses c intact servers $s_1(d), \ldots, s_c(d) \in V$ uniformly and independently at random (Algorithm 3, line 1). This can simply be realized by selecting c random servers in each round until c intact servers have been found (which takes $O(1)$ communication rounds, w.h.p., see Lemma 2.2). Based on these servers we define the following special paths through the k-ary butterfly.

Definition 3.16 (Probing Path)**.** *Consider a request for a data item d received by server s. The unique path from the butterfly node on level $\log_k n$ emulated by $s_i(d)$ to the butterfly node on level 0 emulated by the server that holds d_i is called the* ***probing path of*** *d_i* ***(with origin*** *$s_i(d)$****).***

See Figure 3.7 for a visualization of Definition 3.16.

The actual probing takes place in synchronized rounds. The first $\log_k n + 1$ rounds work as follows. In round 0, all probe messages are active, and their **origin** is declared to be the server s that initiates that probe by sending a

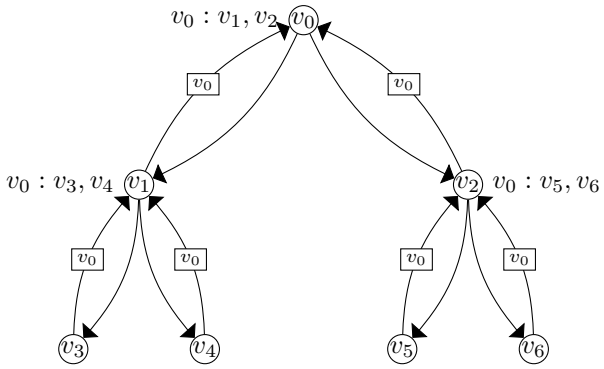

Figure 3.6: Visualization of combining and splitting of messages. The labels at the edges denote messages with destination v_0. The labels next to the nodes denote the combining of a message and the accordingly stored origins: e.g., node v_1 combined the messages from v_3 and v_4 to a single message and stored v_3 and v_4 as its origins.

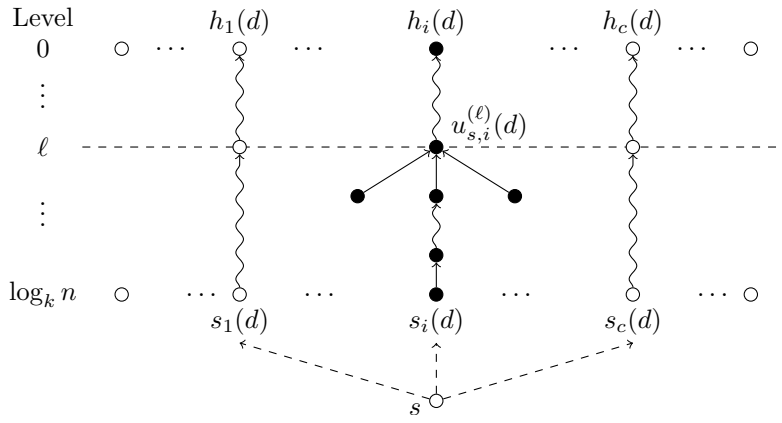

Figure 3.7: Visualization of the Probing Stage. The curved lines denote the paths along which the probe messages are sent.

probe(d, i) message to each server $s_i(d)$, $i \in \{1, \ldots, c\}$ (Algorithm 3, lines 3–5). The servers $s_i(d)$ are supposed to initiate the forwarding of these probes along the according probing paths (Algorithm 4). In round $r \in \{1, \ldots, \log_k n\}$, all probe messages that remain to be active are currently in a butterfly node v at level $\log_k n - r$. Node v checks the following conditions (Algorithm 5, lines 5–9):

- If the number of probe(d, i) messages received at the beginning of round r with different (d, i) pairs is more than αc (for a constant $\alpha > 2(1-\varepsilon)/\gamma$, for instance $\alpha = 9$), then v is called **congested**.
- If $\mathrm{dd}(BF(v)) > \log_k n - r$, then v is called **crashed**.

If v is congested or crashed, then v deactivates all probes it received in round r and informs their origins about the level in which that happened, i.e., the level of node v, by routing this information downwards the probing path. Otherwise, v distinguishes between the following two cases:

- If $\log_k n - r > 0$, i.e., v is not at level 0, then v first combines, for those pairs (d, i) with multiple probes, all of these probes into a single probe and declares itself as the new origin of that probe (Algorithm 5, lines 1–4). Then, v forwards all combined probes to the next node of their probing paths on level $\log_k n - r - 1$ (Algorithm 5, lines 10–13).
- If $\log_k n - r = 0$, i.e., the probes have reached their destination at level 0 in the butterfly, then v delivers the requested data pieces of its probes to their origins (by using splitting if needed) which then can decode the data item using Reed-Solomon coding. These probes are successful (Algorithm 5, lines 14–19).

The splitting of the messages works as follows: Let u be a node that is the origin of a probe (d, i), but not the initial origin of that probe. Whenever u receives a message concerning the probe (d, i), u forwards the message to all nodes it has previously (during the splitting process) stored as former origins of probe (d, i). By this, within $O(\log_k n)$ communication rounds all servers get informed about which of their c probes were successful or got deactivated at a level.

If a server s that is responsible for a lookup request for d receives at least $c/4$ requested data pieces, it can recover d from these pieces (Algorithm 3, lines 6–7). If s receives a dataNotFound(d, i) message, s answers with NULL and is done (Algorithm 3, lines 8–9). Note that for the decision a single dataNotFound(d, i) message suffices since this type of message is only sent if a request for a piece of a requested data item reaches an intact node u at level 0 that does not hold the requested piece (but would have to hold it if d would exist in the system).

In both of these cases the request for d does not need to be handled in the Decoding Stage any more.

Otherwise, s declares the request for d to belong to level ℓ (Algorithm 3, lines 10–12), which is defined as follows.

Definition 3.17 (Belong to)**.** *A request for a data item d is said to **belong to level** $\ell \in \{1, \ldots, \log_k n\}$ if ℓ is the smallest level that contains at least $c/2$ active (d, i) probes, i.e., (d, i) probes that were not deactivated at level $\ell' \geqslant \ell$.*

Note that if a request for d belongs to level $\ell \in \{1, \ldots, \log_k n\}$, then more than $c/2$ probes were deactivated at levels $\ell' \geqslant \ell - 1$.

Regarding the congestion, notice that each node always processes at most $k\alpha c$ messages. This is due to the fact that each node uses an internal congestion bound of αc and therefore never forwards more than αc messages upwards the butterfly and, hence, each node never receives more than $k\alpha c$ messages. Since the Probing Stage basically consists of a message forwarding bottom-up and top-down the k-ary butterfly, the following lemma follows.

Lemma 3.18. *The Probing Stage of Basic IRIS takes at most $O(\log_k n)$ communication rounds with at most $O(\log^2 n)$ congestion at every server in each round.*

Algorithms In the following we provide algorithms in pseudocode that describe the lookup protocol. For that purpose, we divide the view onto the algorithms into three classes of servers depending on the role that the server/node performing the algorithm has: the servers that receive a lookup request and initiate all further actions (Algorithm 3); the servers that initially receive probes and are supposed to forward those through the k-ary butterfly (Algorithm 4); and, finally, the servers that are in their emulation role of a butterfly node v (Algorithm 5).

In Algorithms 3, 4, and 5 we use the notation myVarname(probe(d, i)) in order to describe variables stored at specific nodes and we use probe(d, i).varname in order to describe tags assigned to and stored in a probe.

3.3.3 Decoding Stage

In the Probing Stage, each request has either been served or it has been declared to belong to a level $\ell \in \{1, \ldots, \log_k n\}$. In the Decoding Stage, the latter requests will be served by encoding appropriate subbutterflies. Again, we start with a short overview of this stage followed by a detailed description and the algorithms in pseudocode.

Algorithm 3 BASICIRISPROBING(s) On arrival of lookup request for data item d or answer at s

On arrival of lookup request for data item d:

Choose c intact servers $s_1(d), \ldots, s_c(d) \in V$ uniformly and independently ↳ at random.
for all $i \in \{1, \ldots, c\}$ **do**
    probe(d, i).state ← activated
    probe(d, i).origin ← $\{s\}$
    Send probe(d, i) message to $s_i(d)$.

On arrival of answers from all $s_1(d), \ldots, s_c(d)$:

if received at least $c/4$ data pieces from $s_1(d), \ldots, s_c(d)$ **then**
    Recover d using received pieces and answer request.
else if received dataNotFound(d, i) message **then**
    return NULL
else
    $\ell \leftarrow \arg\min\{\ell \in \{1, \ldots, \log_k n\} \mid \geqslant c/2$ probes were active at level $\ell\}$
    Declare the request for d to belong to level ℓ.

Algorithm 4 BASICIRISPROBING($s_i(d)$) On arrival of probes or answers at $s_i(d)$

On arrival of probe(d, i) message from server s:

▷ *Initiate forwarding through k-ary butterfly from level $\log_k n$ up to level 0*
$u \leftarrow$ butterfly node on level $\log_k n$ emulated by $s_i(d)$
$w \leftarrow$ butterfly node on level 0 emulated by the server that is supposed ↳ to hold d_i
probe(d, i).origin ← u
Send probe(d, i) message to the node on level $\log_k n - 1$ on the ↳ path from u to w in the k-ary butterfly.

On arrival of any answer:

Forward answer to server s.

Algorithm 5 BasicIRISProbing($u = (x, \ell)$) On arrival of probes at u stored in Q

▷ *Combine probes*
1: $Q' \leftarrow \varnothing$
2: **for all** probe$(d_1, i_1), \ldots,$ probe$(d_x, i_x) \in Q$, s.t. $d_1 = \ldots = d_x$, $i_1 = \ldots = i_x$ **do**
3: $Q' := Q' \cup \{$probe$(d_1, i_1)\}$
4: myOrig(probe$(d_1, i_1)) \leftarrow \{$probe$(d_1, i_1)$.origin$, \ldots,$ probe(d_x, i_x).origin$\}$

▷ *Check whether to deactivate probes and, if necessary, do so*
5: **if** $|Q'| > \alpha c$ (u congested) or $\mathrm{dd}(BF(u)) > \ell$ (u crashed) **then**
6: **for all** probe$(d, i) \in Q'$ **do**
7: probe(d, i).state := deactivated
8: **for all** $v \in$ myOrig(probe$(d, i))$ **do**
9: Route deac(d, i, ℓ) through butterfly towards v.

▷ *Forward probes*
10: **if** $\ell > 0$ **then**
11: **for all** probe$(d, i) \in Q'$ **do**
12: probe(d, i).origin $\leftarrow u$
13: Forward probe(d, i) message to the next node on level $\ell - 1$ on ↳ the path to their destination on level 0.

▷ *Probe reached destination at level* 0
14: **if** $\ell = 0$ **then**
15: **for all** probe$(d, i) \in Q'$ **do**
16: **if** data piece d_i exists at the server that emulates node u **then**
17: Forward probeSuccessful(d_i) to all $v \in$ myOrig(probe$(d, i))$.
18: **else**
19: Forward dataNotFound(d, i) to all $v \in$ myOrig(probe$(d, i))$.

Overview The Decoding Stage proceeds in $\log_k n$ phases. Each phase $\ell \in \{1, \ldots, \log_k n\}$ is dedicated to the handling of all requests belonging to level ℓ. That is, for each request for a data item d belonging to level ℓ we first determine whether enough subbutterflies, which contain those d_i's that have not been deactivated at a level $\ell' \geqslant \ell$, can be decoded without causing a too high congestion at any node. This is done by a broadcast in the according subbutterflies. If this is possible, these subbutterflies will be decoded such that the server responsible for the request for d receives sufficiently many pieces of d and it can recover d using Reed-Solomon codes. Otherwise, the request for d is said to belong to level $\ell + 1$, implying that it will be handled in the next phase of the Decoding Stage.

Details Each single phase $\ell \in \{1, \ldots, \log_k n\}$, beginning with $\ell = 1$, is dedicated to the decoding of the requests belonging to level ℓ and is divided into $O(\log_k n)$ rounds. In the first round of phase ℓ each server s that is responsible for a lookup request for some data item d that belongs to level ℓ chooses a set $\mathcal{A}(d) \subseteq \{1, \ldots, c\}$ of $c/2$ indices of probes that were active at level ℓ in the Probing Stage (Algorithm 6, lines 1–2). Each such server s then sends for each $i \in \mathcal{A}(d)$ a $\mathsf{decode}(start, d, i)$ message to $s_i(d)$ (Algorithm 6, line 4). This causes each such $s_i(d)$ to initiate the forwarding of a $\mathsf{decode}(fwd, d, i)$ message from the node on level $\log_k n$ of the probing path of d_i to the node on level ℓ of the probing path of d_i (Algorithm 7, lines 1–3). Note that by forwarding the message to that node on level $\log_k n$ of the probing path of d_i, $s_i(d)$ initially forwards the message to itself, but this does not cause any problems. For this purpose each node on level $\ell' > \ell$ on the probing path of d_i that received a $\mathsf{decode}(fwd, d, i)$ message simply forwards this message to the next node on the probing path on level $\ell' - 1$ (Algorithm 8, lines 2–4). As soon as a $\mathsf{decode}(fwd, d, i)$ message reaches the node at level ℓ of the probing path of d_i, i.e., node $u_{s,i}^{(\ell)}(d)$, it initiates the broadcast of that message in the butterfly $UT(u_{s,i}^{(\ell)}(d))$ (Algorithm 8, lines 5–7). See Figure 3.8 for a visualization of that broadcast. The goal of the broadcast in $UT(u)$ is to determine whether the subbutterfly $BF(u)$ is congested or not, where a congested subbutterfly is defined as follows:

Definition 3.19 (Congested Subbutterfly)**.** *A subbutterfly $BF(v)$ of a node v is called **congested** if at least one node from $BF(v)$ receives more than βck $\mathsf{decode}(\cdot, \cdot, \cdot)$ messages for different (d, i) pairs for a constant $\beta > 3/2$.*

In order to determine whether any $BF(u)$ is congested, each node u that received a $\mathsf{decode}(fwd, \cdot, \cdot)$ message during the broadcast is declared as **congested** if any of the following conditions is satisfied (Algorithm 8, lines 8–12)

1. u received more than βck decode$(\cdot, \cdot, \cdot)$ messages for different (d, i) pairs for an arbitrary but fixed constant $\beta > 3/2$.
2. u received at the beginning of this round a decode$(cong, \cdot, \cdot)$ message.

As long as the broadcast has not reached a node at level 0, each node u that received a decode$(\cdot, d, i)$ message simply forwards a decode(fwd, d, i) message, in case it is not congested or forwards a decode$(cong, d, i)$ message otherwise (Algorithm 8, lines 14–15). Once the broadcast reaches a node at level 0 (Algorithm 8, lines 16–19), each node that received at least one decode$(cong, d, i)$ message routes a decodeFinish$(cong, d, i)$ message to the origins of the (d, i) pair downwards the butterfly. All nodes u at level 0 that did not receive a decode$(cong, \cdot, \cdot)$ message, but instead received at least one decode(fwd, d, i) message, broadcast the information to start the decoding of $BF(u_{s,i}^{(\ell)}(d))$ to its children in $UT(u)$, i.e., downwards the butterfly by sending a decodeFinish$(startDec, d, i)$ message to its children (Algorithm 8, line 19). Recall that $BF(u_{s,i}^{(\ell)}(d))$ denotes the unique ℓ-dimensional subbutterfly that contains the node $u_{s,i}^{(\ell)}(d)$, which is the node at level ℓ on the probing path of d_i with origin $s_i(d)$. At the end of the decoding of $BF(u_{s,i}^{(\ell)}(d))$ all origins u at level ℓ of $BF(u_{s,i}^{(\ell)}(d))$ received a message decodeFinish$(data, d, i)$ for all pairs (d, i) for which they previously received a decodeFinish$(startDec, d, i)$ message. If the data piece d_i exists in the system it holds $data = d_i$. Otherwise it holds $data =$ NULL (Algorithm 9, line 5). As soon as the nodes at level ℓ have received the answer from the decoding or a decodeFinish$(\langle type \rangle, \cdot, \cdot)$ message with $type \in \{cong, fwd\}$ instead, the nodes forward this answer back to the origins of the (d, i) pairs (Algorithm 9, lines 6–9). Finally, after in total $O(\log n)$ rounds, these answers have reached their destinations $s_1(d), \ldots, s_i(d)$ and each $s_i(d)$ now forwards its answer back to the origin s of the request (Algorithm 9, line 11). If s received a decode(NULL, d, i) message, the requested data item d does not exist in the system and s returns NULL (Algorithm 6, lines 5–6). In case s receives at least $c/4$ data pieces of the requested data item d, it can recover d using Reed-Solomon codes. Otherwise, s declares the request for d to belong to level $\ell + 1$ (Algorithm 6, lines 8–9) implying that it will be handled again in the next phase of the Decoding Stage.

Using the symmetry of the k-ary butterfly and the way messages are forwarded, one can show the following lemma.

Lemma 3.20. *Let u be a butterfly node at level ℓ that received a* decode$(fwd, \cdot, \cdot)$ *message in phase ℓ. If $BF(u)$ is congested, then each node on level 0 of $BF(u)$ receives a* decode$(cong, \cdot, \cdot)$ *message after ℓ rounds.*

Proof. Let u be a butterfly node at level ℓ that received a decode request in phase ℓ and let $BF(u)$ be congested. By Definition 3.19, this implies that a node

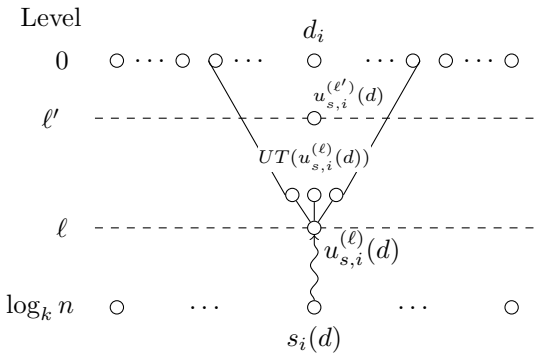

Figure 3.8: Visualization of the broadcast in phase ℓ of the Decoding Stage.

v from $BF(u)$ at level $\ell' < \ell$ receives more than βck $\mathsf{decode}(\cdot,\cdot,\cdot)$ messages for different (d,i) pairs. Notice that v cannot be at level ℓ since otherwise the subbutterfly $BF(u)$ would not have been chosen to be decoded.

First, assume v is contained in the upper tree $UT(u)$ of u. Since v is in the same k-block with the other nodes on level ℓ' in $UT(u)$, these nodes receive the same messages that v receives. Hence, all nodes on level ℓ' are congested and forward this information to all nodes on a level $\ell'' < \ell'$ in $UT(u)$, implying that all nodes on level 0 of $UT(u)$ receive a $\mathsf{decode}(cong,\cdot,\cdot)$ message after ℓ rounds in total.

Next, assume v is not contained in the upper tree $UT(u)$ of u. Then, there exists a node w on level ℓ, such that v is contained in $UT(w)$. From $v \in BF(u)$ and the definition of k-ary butterflies we can deduce $BF(u) = BF(w)$. Analogously to the previous case, we have that all nodes on level ℓ' of $UT(w)$ are congested and a $\mathsf{decode}(\mathsf{cong},\cdot,\cdot)$ message is forwarded to all nodes on level 0 of $BF(w) = BF(u)$.

□

Regarding the efficiency we get the following lemma.

Lemma 3.21. *The Decoding Stage of the Lookup Protocol of Basic IRIS takes at most $O(\log_k^2 n)$ communication rounds with at most $O(\log^3 n)$ congestion at every server in each round.*

Proof. Each phase of the Decoding Stage consists of a bottom-up traversal through the butterfly, followed by congestion checks and possibly decoding and returning of the answers to the servers $s_i(d)$. Each of these procedures requires $O(\log_k n)$ communication rounds. Since there are $\log_k n$ phases in the Decoding Stage, $O(\log_k^2 n)$ communication rounds are required. Analogously to the Probing Stage, the nodes use an internal congestion bound of βck and

receive in each round messages from at most k butterfly nodes. Hence, each server has a congestion of $O(\log^3 n)$ in each round. □

Algorithms Analogously to the Probing Stage, in the following we provide algorithms in pseudocode that describe the Decoding Stage of the Lookup Protocol.

Algorithm 6 BASICIRISDECODING(s) First and last round of phase ℓ, performed by server s with a request for d belonging to level ℓ

At the beginning of phase ℓ of the Decoding Stage:

$\mathcal{A}(d) \leftarrow \{i \in \{1, \ldots, c\} \mid \mathsf{probe}(d, i)$ was active at level ℓ in the Probing St.$\}$

If necessary, reduce $\mathcal{A}(d)$ to $c/2$ elements by removing arbitrary indices ↳ from it.

for all $i \in \mathcal{A}(d)$ **do**

    Send $\mathsf{decode}(start, d, i)$ message to $s_i(d)$.

On arrival of $\mathsf{decodeFinish}(\cdot, d, \cdot)$ messages (saved in Q) from all $s_1(d), \ldots, s_c(d)$:

if $\mathsf{decodeFinish}(\mathsf{NULL}, d, i) \in Q$ **then**

    return NULL

$\mathcal{S}(d) \leftarrow \{d_i \mid \exists\ \mathsf{decodeFinish}(d_i, d, i) \in Q$ with $d_i \neq \mathsf{NULL}\}$

if $|\mathcal{S}(d)| \geqslant c/4$ **then**

    Recover d using pieces in $\mathcal{S}(d)$ and Reed-Solomon codes.

else

    Declare the request for d to belong to level $\ell + 1$.

3.3.4 Differences and Similarities to Previous Works

The structure of the Probing Stage and the Decoding Stage of Basic IRIS resemble the one of the Contraction Stage and the Expansion Stage of the lookup protocol in [AS07; BSS09]. In order to make differences and similarities clear, we first need to shortly summarize some properties of the storage strategy in [AS07; BSS09]. Just as in Basic IRIS in [AS07; BSS09] a level based approach is used: i.e., for each server $\log_k n + 1$ virtual nodes are introduced and arranged into levels such that at each level from level 0 to level $\log_k n$ each server emulates exactly one virtual node. In order to store a data item d into the system presented in [AS07; BSS09], first, as in Basic IRIS, $c = \Theta(\log m)$ pieces of d are created via Reed-Solomon codes which are mapped to the nodes at level 0 using c hash functions $h_1, \ldots, h_c$. Then, roughly speaking, for each piece d_i of d a specific randomly chosen path through the virtual nodes from level $\log_k n$

Algorithm 7 BasicIRISDecoding$(s_i(d))$ performed by server $s_i(d)$ in phase ℓ

On arrival of **decode**$(start, \cdot, \cdot)$ *message from server* s *at the beginning of round* 2 *of phase* ℓ*:*

1: **for all** decode$(start, d, i)$ messages received from a server s **do**
2: myOrig$(d, i) \leftarrow$ myOrig$(d, i) \cup \{s\}$ ▷ *myOrig*(d, i) *initially empty*
3: Forward decode(fwd, d, i) message to node u on level $\log_k n$ of the probing path of d_i.

On arrival of **decodeFinish**$(\cdot, \cdot, \cdot)$ *messages saved in* Q*:*

4: **for all** decodeFinish$(\langle type \rangle, d, i) \in Q'$: **do**
5: **for all** servers $s \in$ myOrig(d, i) **do**
6: Forward decodeFinish$(\langle type \rangle, d, i)$ to s.

to the node responsible for $h_i(d)$ at level 0 is chosen and at each node of that path a copy of d_i is stored. Note that in contrast to the storage strategy in [AS07; BSS09], in Basic IRIS the virtual nodes need to be connected with each other in order to form appropriate complete k-bipartite subgraphs. Furthermore, in Basic IRIS, the data pieces are not solely stored at the different levels. Instead they are only stored at level 0 and then they completely encoded with each other, such that the encoding information is spread over all levels and servers.

The basic idea of the Contraction Stage of the lookup protocol in [AS07; BSS09] is similar to the Probing Stage of Basic IRIS. In the Contraction Stage in [AS07; BSS09], requests for the c pieces of d are forwarded along $O(\log n)$ randomly chosen paths through the $\log_k n + 1$ levels, beginning at level $\log_k n$ and ending at the nodes at level 0 that are responsible for $h_1(d), \ldots, h_c(d)$. Similarly to Basic IRIS, in [AS07; BSS09] at each level each node performs a congestion check and an "attacked" check. For this purpose in [AS07; BSS09] each node v at each level contacts $O(\log n)$ nodes in its "surrounding" in order to determine their current congestion and whether they are attacked. If too many nodes are congested or attacked, the request for the considered piece is deactivated. In contrast to this in Basic IRIS a node at level ℓ determines whether it is congested based on the messages it has previously received from nodes at level $\ell + 1$. The "attacked" check in Basic IRIS is more involved than the one in [AS07; BSS09]. That is, in Basic IRIS we not only determine the number of attacked servers in the current surrounding, but information about that in the complete ℓ-dimensional subbutterfly $BF(v)$. This is done via the decoding depth we already computed in the Preprocessing Stage for each subbutterfly $BF(v)$.

The structure of the Decoding Stage of Basic IRIS resembles the one of the Expansion Stage of the lookup protocol in [AS07; BSS09], in terms that in

Algorithm 8 BASICIRISDECODING$(u = (\ell', x))$ performed by node u at level ℓ' in round $\geqslant 3$ of phase ℓ on arrival of decode$(\cdot, \cdot, \cdot)$ messages saved in Q

1: Combine messages in Q to set Q' and set myOrig and origins of messages ↳ in Q' just as in lines 1–4 of Algorithm 5.

▷ *Consider nodes at level $\ell \leqslant \ell'$*
2: **if** $\ell < \ell'$ **then** ▷ *Level ℓ not reached ⇒ route mess. upwards the probing path*
3: Forw. decode(fwd, d, i) to node on level $\ell' - 1$ of probing path of d_i.
4: **return**
5: **else if** $\ell = \ell'$ **then** ▷ *Level ℓ reached ⇒ initiate broadcast in $UT(v_i(\ell))$*
6: Broadcast decode(fwd, d, i) message to children of u in $UT(u)$.
7: **return**

▷ *Determine whether u is congested*
8: **if** $|Q'| > \beta ck$ or decode$(cong, \cdot, \cdot) \in Q'$ **then**
9: u.state ← congested
10: $msgType \leftarrow cong$
11: **else**
12: $msgType \leftarrow fwd$

▷ *Forward received messages*
13: **for all** decode$(\cdot, d, i) \in Q'$ **do**
14: **if** $\ell' > 0$ **then**
▷ *Cong. check of sub-BF not finished ⇒ forward mess. upwards the BF*
15: Broadcast decode$(msgType, d, i)$ to children of u in $UT(u)$.
16: **else if** $msgType = cong$ **then**
▷ *Cong. check of sub-BF finished ⇒ forward results downwards the BF*
17: Route decodeFinish$(cong, d, i)$ to origins of (d, i) pair.
18: **else if** $msgType = fwd$ **then** ▷ *Destination reached*
▷ *Broadcast info to start decoding $BF(u_{s,i}^{(\ell)}(d))$ downwards the BF*
19: Broadcast decodeFinish$(startDec, d, i)$ to children of u in $LT(u)$ ↳ (thereby combine multiple messages for the same (d, i) pair ↳ to one message).

Algorithm 9 BASICIRISDECODING$(u = (\ell', x))$ performed by node u at level ℓ' in round $\geqslant 3$ of phase ℓ on arrival of decodeFinish$(\cdot, \cdot, \cdot)$ messages saved in Q

On arrival of decodeFinish$(startDec, \cdot, \cdot)$ *messages saved in* Q*:*

1: **if** $\ell' < \ell$ **then**
2: Broadcast decodeFinish$(startDec, d, i)$ to all children of u in $LT(u)$.
3: **else if** $\ell' = \ell$ **then**
4: Start decoding of $BF(u)$ via bottom-up approach.
5: At the end of the decoding all origins at level ℓ of $BF(u)$ receive
↳ message decodeFinish$(data, d, i)$ for all pairs (d, i) at node u, with
↳ $data = d_i$ if d_i exists in the system and $data =$ NULL otherwise.

On arrival of first decodeFinish$(data, d, i)$ *messages:*

6: Q': combined set of all decodeFinish$(\cdot, \cdot, \cdot)$ messages received in phase ℓ
7: **for all** decodeFinish$(\langle type \rangle, d, i) \in Q'$ **do**
8: **if** $\ell' < \log_k n$ **then** ▷ *Downwards forw. along probing paths not finished*
9: Forward decodeFinish$(\langle type \rangle, d, i)$ to all $v \in$ myOrig(d, i).
10: **else** ▷ *Messages arrived at level* $\log_k n$ *and* u *is emulated by a server* $s_i(d)$
11: Forward messages to origins s

each phase $\ell \in \{1, \ldots, \log_k n\}$ the requests that have been assigned to level ℓ are handled. However, the single phases of the Decoding Stage in Basic IRIS are more involved. In phase ℓ of the Expansion Stage in [AS07; BSS09] each node at level ℓ with a request for a data piece d_i broadcasts that requests in its current "surrounding" in order to retrieve a copy of d_i. In contrast to this, in phase ℓ of the Decoding Stage of Basic IRIS, each node v at level ℓ with a request for a data piece d_i initiates the decoding of the complete ℓ-dimensional subbutterfly $BF(v)$. For that purpose, it additionally previously needs to determine whether $BF(v)$ can be decoded without any node becoming congested which is done via a broadcast in $BF(v)$.

3.4 Correctness Analysis of the Lookup Protocol

In this section we show that at the end of the lookup protocol each lookup request has been served correctly. The analysis of that is divided into two parts: The analysis of the Probing Stage (Section 3.4.2) and the analysis of the Decoding Stage (Section 3.4.3). Similar to the analysis of the lookup protocol in [AS07; BSS09] we show that for both of these stages the number of requests belonging to a level exponentially decreases from level 1 to level $\log_k n$.

For this purpose, we make use of hash functions with certain expansion properties, similar to [AS07; BSS09], which we present in Section 3.4.1. Prior to the work of [AS07; BSS09], these expansion properties have already been introduced and used in [AS06].

3.4.1 Robust Hash Functions

Recall that $\mathcal{U}$ is the key universe and $m = |\mathcal{U}|$. For any subbutterfly B let $V(B)$ be the set of servers emulating the nodes of B. Let $\mathcal{H}$ be the collection of hash functions $h_1, \ldots, h_c$. With this, a bundle and an expander are defined as follows:

Definition 3.22 (Bundle)**.** *Let $S \subset \mathcal{U}$ be a set of keys and $k \in \mathbb{N}, b \in \mathbb{Q}$. A set of tuples $F \subseteq S \times \{1, \ldots, c\}$ is called a b-**bundle** of S, if every $d \in S$ has exactly b many pairs (d, i) in F. Given $h_1, \ldots, h_c$ and a level $\ell \in \{0, \ldots, \log_k n\}$, let $\Gamma_{F,\ell}(S)$ be the union of the servers involved in these pairs at level ℓ, i.e., $\Gamma_{F,\ell}(S) = \bigcup_{(d,i)\in F} V(BF(u_{s,i}^{(\ell)}(d)))$.*

Definition 3.23 (Expander)**.** *Given a $0 < \sigma < 1$, we call $\mathcal{H}$ a (b, σ)-**expander** if for any $\ell \leqslant \log_k n$, any $S \subseteq \mathcal{U}$ with $|S| \leqslant \sigma n/k^\ell$, and any b-bundle F of S, it holds that $|\Gamma_{F,\ell}(S)| \geqslant k^\ell |S|$.*

Later on in the analysis we use b-bundles in order to indicate for a set of data items S with a certain property (e. g., belonging to a crashed or congested request) a set of $b \cdot |S|$ ℓ-dimensional subbutterflies $BF(u_{s,i}^{(\ell)}(d))$ that witness that property. Hence, in our analysis $\Gamma_{F,\ell}(S)$ is defined as the set of servers that emulate a node in any of these $b \cdot |S|$ witnessing butterflies.

In contrast to the consideration of these ℓ-dimensional subbutterflies $BF(u_{s,i}^{(\ell)}(d))$, in the analysis of [AS07; BSS09] sets of points from the $[0, 1)$ interval are considered that are at distance ℓ from $h_i(d)$ for an appropriately chosen distance measure.

The following lemma is a slightly generalized version of Lemma 1 in [AS07] and Claim 2.13 in [BSS09].

Lemma 3.24. *Let $\rho, \sigma > 0$ with $0 < \sigma < 1$, $\rho\sigma \leqslant 1/2$ and $c \geqslant 2\rho \log m$. If the hash functions $\mathcal{H} = \{h_1, \ldots, h_c\}$ are chosen uniformly and independently at random, then $\mathcal{H}$ is a $(c/\rho, \sigma)$-expander, w.h.p.*

Although Scheideler et al. proved a similar version of Lemma 3.24 in [AS07; BSS09], for the sake of completeness we present an adapted proof here.

Proof. Suppose that for randomly chosen hash functions $h_1, \ldots, h_c$, $\mathcal{H}$ is not a $(c/\rho, \sigma)$-expander. Then, there exists an $i \leqslant \log_k n$, a set $S \subseteq \mathcal{U}$ with $|S| \leqslant \sigma n/k^i$, and a c/ρ-bundle F of S with $|\Gamma_{F,i}(S)| < k^i|S|$. The probability $p_{s,i}$ that such a set S of size s exists is at most

$$\binom{m}{s}\binom{cs}{cs/\rho}\binom{n/k^i}{s}\left(\frac{s}{n/k^i}\right)^{cs/\rho}$$

for the following reasons: There are $\binom{m}{s}$ ways of choosing a subset $S \subset U$. Furthermore, there are $\binom{cs}{cs/\rho}$ ways of choosing cs/ρ pairs (d, j) for F and at most $\binom{n/k^i}{s}$ ways of choosing a set W of s butterflies of dimension i witnessing a bad expansion of the pairs in F. The fraction of collections $\mathcal{H}$ for which the selected pairs (d, j) indeed have the property that $BF(u^{(i)}_{s,j}(d)) \in W$ is equal to $(\frac{s}{n/k^i})^{cs/\rho}$, because the hash functions $h_1, \ldots, h_c$ are chosen independently and uniformly at random. Next, we show $p_{s,i} < 1/m^s$, implying that by summing over all possible values of s and i we obtain a probability of having a "bad" c/ρ-bundle of less than 1 for m sufficiently large, which proves the lemma.

Since for all $a, b \in \mathbb{N}$ with $b < a$ it holds $\binom{a}{b} \leqslant ((ea)/b)^b$, using the conditions on c, ρ and σ in the lemma and m sufficiently large it holds:

$$\begin{aligned}
p_{s,i} &\leqslant \left(\frac{em}{s}\right)^s (\rho e)^{cs/\rho}\left(\frac{en}{sk^i}\right)^s\left(\frac{sk^i}{n}\right)^{cs/\rho} \\
&= \left(\frac{em}{s}\right)^s (\rho e)^{cs/\rho} e^s \left(\frac{sk^i}{n}\right)^{-s}\left(\frac{sk^i}{n}\right)^{cs/\rho} \\
&= \left[\frac{em}{s}\cdot\left(\rho e^{1+\rho/c}\cdot\left(\frac{sk^i}{n}\right)^{1-\rho/c}\right)^{c/\rho}\right]^s \\
&\overset{(*)}{\leqslant} \left[m\cdot\left(\rho e^{1+\rho/c}\cdot\sigma^{1-\rho/c}\right)^{c/\rho}\right]^s = \left[m\cdot(\rho\sigma e)^{c/\rho}\right]^s \\
&\overset{(**)}{\leqslant} \left[m\cdot\left(\frac{1}{2}\right)^{c/\rho}\right]^s \overset{(***)}{\leqslant} \frac{1}{m^s}
\end{aligned}$$

In $(*)$ we used $s \geqslant e$ and $\sigma \geqslant sk^i/n$, in $(**)$ we used $\rho\sigma \leqslant 1/2$, in $(***)$ we used $c \geqslant 2\rho \log m$. □

We remark that the hash functions have to form a $(c/4, \sigma)$-expander for some constant σ for our lookup protocol to work, but they do not have to be chosen

at random. The proof above just illustrates that if they are chosen at random, they will form a $(c/4, \sigma)$-expander, w.h.p.

In our analysis we will use the before mentioned expansion properties in the following form, which is an implication of Lemma 3.24.

Corollary 3.25. *Let S be a set of data items and let F be a (c/ρ)-bundle of S with $\rho > 0$. Then, for any $\ell \in \{0, \ldots, \log_k n - 1\}$ and $0 < \sigma < 10$ with $\rho\sigma \leqslant 1/2$ it holds: If $|\Gamma_{F,\ell}(S)| < \sigma n$, then it holds $|S| < \sigma n/k^\ell$.*

Proof. From Lemma 3.24 we can conclude that for any (c/ρ)-bundle F' of a set of data items S' and any $\ell \in \{0, \ldots, \log_k n - 1\}$ with $|S'| \leqslant \sigma n/k^\ell$ it holds $|\Gamma_{F',\ell}(S')| \geqslant k^\ell |S'|$. By the prerequisite we know $|\Gamma_{F,\ell}(S)| < \sigma n$. For the sake of contradiction assume $|S| \geqslant \sigma n/k^\ell$. If $S = \sigma n/k^\ell$, then by Lemma 3.24, we get $|\Gamma_{F,\ell}(S \geqslant k^\ell |S|$ which contradicts the preliminary $|\Gamma_{F,\ell}(S)| < k^\ell |S|$. Now assume $S > \sigma n/k^\ell$. We know that for a set S' with maximum value $|S'| = \sigma n/k^\ell$, it holds $|\Gamma_{F,\ell}(S')| \geqslant k^\ell |S'| = \sigma n/k^\ell$. Furthermore, for $|S| > |S'|$ with $S' \subset S$ it holds $|\Gamma_{F,\ell}(S)| \geqslant |\Gamma_{F,\ell}(S')|$. Hence, when defining $S' \subset S$ to be any subset of S that consists of $\sigma n/k^\ell$ elements, we get $|\Gamma_{F,\ell}(S)| \geqslant |\Gamma_{F,\ell}(S')| \geqslant k^\ell |S|$ which again contradicts the preliminary $|\Gamma_{F,\ell}(S)| < k^\ell |S|$. □

3.4.2 Analysis of the Probing Stage

In the following we show that in the Probing Stage not "too many" requests are declared to belong to the same level. To be more precise, we show that if the adversary can crash at most $\gamma n^{1/\log\log n}$ servers, then the number of requests belonging to a level exponentially decreases such that there is at most a logarithmic number of requests belonging to the last level, level $\log_k n$. Lemma 3.26 formalizes this.

Lemma 3.26. *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/9$. Then, at the end of the Probing Stage of the Lookup Protocol of Basic IRIS, the number of requests belonging to level $\ell \in \{1, \ldots, \log_k n\}$ is at most $2\gamma n/k^{\ell-1}$.*

Lemma 3.26 can be shown by adapting the analysis in [AS07] (see Lemmas 4 and 5). Before we give a proof, we point out the main differences to the analysis in [AS07] and introduce some required definitions and claims.

An important new ingredient in the analysis of Lemma 3.26 compared to [AS07] is the concept of witness trees which are defined as follows.

Definition 3.27 (Witness Tree)**.** *A **witness tree** of a butterfly node u at level $\ell \in \{0, \ldots, \log_k n\}$ is defined to be a complete binary subtree of the lower tree $LT(u)$ of depth $\log_k n - \ell$ that only consists of nodes emulated by crashed servers.*

As the following lemma states, there is a direct relation between the decoding depth and witness trees.

Lemma 3.28. *Let u be a butterfly node at level $\ell \in \{0, \dots, \log_k n\}$. The lower tree $LT(u)$ contains a witness tree if and only if $dd(u) > \log_k n - \ell$.*

Proof. Let u be a butterfly node at level $\ell \in \{0, \dots, \log_k n\}$. First, assume $\mathrm{dd}(u) > \log_k n - \ell$. By induction on $\ell = \log_k n, \dots, 0$ we show that $LT(u)$ contains a witness tree.

For the induction base let $\ell = \log_k n$. Hence, u is at the last level of the k-ary butterfly and it holds $\mathrm{dd}(u) > 0$. By Definition 3.13, we know that u is crashed, which implies the claim.

Now, let $\ell < \log_k n - 1$ and assume the claim holds for all $\ell' > \ell$. Since $\mathrm{dd}(u) > \log_k n - \ell > 1$, by Definition 3.13 there exist two nodes v, w at level $\ell + 1$ in $LT(u)$ with $\mathrm{dd}(v) > \log_k n - \ell - 1$ and $\mathrm{dd}(w) > \log_k n - \ell - 1$. By the induction hypothesis there exist witness trees T_v and T_w for v and w. Since T_v and T_w are subtrees of $LT(u)$, the tree T induced by connecting u to the roots of T_v and T_w is a complete binary subtree of $LT(u)$ of depth $\log_k n - \ell$, i.e., T is a witness tree of u.

Next, assume $LT(u)$ contains a witness tree. Analogously to the first part, by induction on $\ell = \log_k n, \dots, 0$ we show $\mathrm{dd}(u) > \log_k n - \ell$. Note that for the depth $depth(LT(u))$ it holds $depth(LT(u)) = \log_k n - \ell$.

For the induction base let $\ell = \log_k n$. In this case $depth(LT(u)) = 0$, i.e., $LT(u)$ consists of the single node u at level $\log_k n$ which is crashed. With Definition 3.13 we get $\mathrm{dd}(u) = 1 > \log_k n - \ell = 0$.

Now, let $\ell < \log_k n$ and assume the claim holds for all $\ell' > \ell$. Let T_u be a witness tree of $LT(u)$. Since $depth(LT(u)) = depth(T_u) > \log_k n - \ell > 0$, node u has two children v, w at level $\ell + 1$ in T_u. Since T_u is a witness tree, the trees T_v and T_w rooted at v, w that are subtrees of T_u are witness trees of v and w. By the induction hypothesis this implies $\mathrm{dd}(v) > \log_k n - (\ell - 1)$ and $\mathrm{dd}(v) > \log_k n - (\ell - 1)$. With Definition 3.13 we get $\mathrm{dd}(u) \geqslant \max\{\mathrm{dd}(v), \mathrm{dd}(u)\} + 1 > \log_k n - \ell$. □

Recall that by Lemma 3.32 the data stored at any node u with $\mathrm{dd}(u) \leqslant \log_k n$ can be recovered correctly. Together with Lemma 3.28 this implies that the data stored at any node u that does not have a witness tree can be recovered correctly. Hence, if there does not exist any witness tree at a butterfly node at level 0, then the complete data stored in the k-ary butterfly can be recovered.

Notice that due to the structure of the $BF(k, d)$, the leaves in any witness tree are distinct. Since a complete binary tree of depth $\log_k n$ has $2^{\log_k n}$ leaves, the k-ary butterfly cannot contain a witness tree of depth $\log_k n$ if the adversary crashes less than $2^{\log_k n} = n^{1/\log\log n}$ servers. In particular, if the adversary crashes less than $n^{1/\log\log n}$ servers, then the complete k-ary butterfly can be recovered.

Next, we introduce the definition of crashed and congested butterflies and requests. Note that in Definition 3.19 we already provided a definition of a congested subbutterfly with a different bound than the one used in the following definition. In the following of this section we will refer to Definition 3.29 whenever we denote a subbutterfly or a request to be congested.

Definition 3.29 (Crashed/Congested Subbutterfly/Request)**.** *Let v be a node at level ℓ of the k-ary butterfly. Then, the ℓ-dimensional k-ary subbutterfly $BF(v)$ is called*

- ***crashed** if at least 2^ℓ servers from $BF(v)$ are crashed,*
- ***congested** (w.r.t. the probing) if the servers from $BF(v)$ receive in total more than $k^\ell \alpha c/2$ probes for different (d, i) pairs in round ℓ.*

A request for a data item d is called

- ***crashed** at level ℓ if there are $r = c/4$ crashed subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell$ and $i_1, \ldots, i_r$ being pairwise different,*
- ***congested** (w.r.t. the probing) at level ℓ if there are $r = c/4$ congested subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell$ and $i_1, \ldots, i_r$ being pairwise different.*

We can now give an overview of the proof of Lemma 3.26. The idea of this proof is as follows: First, we show that whenever a (d, i) probe is deactivated by a node v on a level ℓ, then $BF(v)$ is crashed or congested, w.h.p. (Lemma 3.30). Moreover, if a request for a data item d is declared to belong to level ℓ, then at least $c/2$ of its (d, i) probes have either been deactivated because of crashed subbutterflies or because of congested subbutterflies at level $\ell - 1$ or higher. Therefore, many requests belonging to level ℓ imply many crashed or congested subbutterflies at level $\ell - 1$. But since only a limited fraction of them can be crashed or congested, only a limited fraction of the requests can belong to level ℓ.

Lemma 3.30. *Whenever a (d, i) pair is deactivated on a level $\ell \geqslant 0$ by a node v, then $BF(v)$ is crashed or congested, w.h.p.*

Proof. If (d, i) was deactivated due to $\mathrm{dd}(BF(v)) > \ell$ with $v = u_{s,i}^{(\ell)}(d)$, then by Lemma 3.28, $BF(v)$ contains at least 2^ℓ crashed servers, i.e., by Definition 3.29 $BF(v)$ is crashed.
Now assume (d, i) was deactivated due to a too high congestion at v. Then, according to the protocol, v received in round ℓ probe messages for more than

αc different (d, i) pairs. Since the starting points for the lookup requests are chosen uniformly at random, it holds $E\left[|\mathcal{M}_\ell(w)|\right] = E\left[|\mathcal{M}_\ell(w')|\right]$ for all w, w' at level ℓ in $BF(v)$ with $\mathcal{M}_\ell(w)$ being the set of (d, i) pairs with probes received by node w. Thus, Chernoff bounds (Lemma 2.1) can be applied, implying

$$Pr\left[|\mathcal{M}_\ell(w)| \geqslant (1+\delta)E\left[|\mathcal{M}_\ell(w)|\right]\right] \leqslant e^{-\min\{\delta,\delta^2\}E[|\mathcal{M}_\ell(w)|]/3}$$

for all $\delta \geqslant 0$ and all $w \in BF(v)$. Setting $\delta = 1/2$ gives $E\left[|\mathcal{M}_\ell(w)|\right] \geqslant 2\alpha c/3$ for all $w \in BF(v)$, w.h.p. Hence, the expected number of (d, i) pairs for which a probe has been sent to $BF(v)$ is at least $2\alpha k^\ell c/3$, w.h.p. Furthermore, with M being the number of (d, i) pairs for which a probe has been sent to $BF(v)$, Chernoff bounds (Lemma 2.1) imply

$$Pr\left[M \leqslant \frac{2(1-\delta)}{3}\alpha c k^\ell\right] \leqslant e^{-\delta^2\alpha c k^\ell/3} \quad \text{for all } \delta \in [0,1].$$

With $\delta = 1/4$ we get that there are more than $\alpha c k^\ell/2$ (d, i) pairs for which a probe has been sent to $BF(v)$, w.h.p., i.e., $BF(v)$ is congested. □

Note that a similar version of Lemma 3.30 can also be found in [AS07; BSS09] (Claim 2 in [AS07], Claim 2.14 in [BSS09]).

Lemma 3.30 implies the following corollary that allows us to prove Lemma 3.26 by upper bounding the number of requests that are crashed or congested at level $\ell - 1$.

Corollary 3.31. *If a request belongs to a level $\ell \in \{1, \ldots, \log_k n\}$, then this request is crashed or congested at level $\ell - 1$.*

Proof. If a request for d belongs to level ℓ, then by Definition 3.17, we know that more than $c/2$ probes for (d, i) pairs have been deactivated at a level $\ell' \geqslant \ell - 1$. Lemma 3.30 implies that more than $c/2$ subbutterflies are congested or crashed at level $\ell' \geqslant \ell - 1$. For the sake of contradiction assume that the request for d is neither crashed nor congested at level $\ell - 1$. By Definition 3.29, this means that there are only less than $c/4$ subbutterflies at level $\ell' \geqslant \ell - 1$ that are congested and only less than $c/4$ subbutterflies at level $\ell' \geqslant \ell - 1$ that are crashed. All in all, this gives that there are less than $c/2$ subbutterflies at level $\ell' \geqslant \ell - 1$ that are congested or crashed, which yields a contradiction. □

With these tools we are now ready to prove Lemma 3.26.

Proof of Lemma 3.26: First notice that by Lemma 3.30 it holds that if a request belongs to level ℓ, then more than $c/2$ nodes that received a probe for this lookup request are congested or crashed at level $\ell - 1$. This, together with Lemma 3.30 implies that if a lookup request for some data item d belongs to

level ℓ, then d must be crashed or congested at level $\ell - 1$. For the proof of this lemma we upper bound the number of crashed requests at level $\ell - 1$ and the number of congested requests at level $\ell - 1$ by $\gamma n/k^{\ell-1}$ each. Corollary 3.31 then yields the claim. With Definition 3.29 this means that in total less than $2c\gamma n/(4k^\ell) = c\gamma n/(2k^\ell)$ subbutterflies at a level $\ell' \geqslant \ell$ are congested or crashed. Furthermore, for a request to belong to level ℓ at least $c/2$ probes have to be deactivated at a level $\ell' \geqslant \ell$. With Lemma 3.30 this implies that at least $c/2$ subbutterflies are congested or crashed at a level $\ell' \geqslant \ell$. Hence, the number of requests belonging to level ℓ is upper bounded by $(c\gamma n/(2k^\ell))/(c/2) = \gamma n/k^\ell$.

Upper bound on the number of crashed data items. Let S be a maximum set of data items for which there are requests that are crashed at level $\ell - 1$. Our goal is to show that $|S| < \gamma n/k^{\ell-1}$. By Definition 3.29, for each $d \in S$ there exist $r = c/4$ subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_1, \ldots, \ell_r \geqslant \ell - 1$ and $i_1, \ldots, i_r$ being pairwise different. Hence, the set $F := S \times \{i_1, \ldots, i_r\}$ is a $c/4$-bundle F of S. By Lemma 3.28, each crashed subbutterfly at a level ℓ' contains at least $2^{\ell'}$ crashed servers. On the other hand we know that the adversary crashes at most εn servers with $\varepsilon < \gamma \cdot 2^{\log_k n}/n$. Since each subbutterfly at a level $\ell' \geqslant \ell - 1$ contains $k^{\ell'}$ servers, in total we get that the number of servers covered by all $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ is upper bounded by

$$\frac{\varepsilon n \cdot k^{\ell'}}{2^{\ell'}} = \frac{\varepsilon n}{2^{\ell'}/k^{\ell'}} < \frac{\gamma n \cdot 2^{\log_k n}/n}{2^{\log_k n}/n} = \gamma n.$$

Since this set of servers is exactly the set $\Gamma_{F,\ell-1}(S)$, we get $|\Gamma_{F,\ell}(S)| < \gamma n$. Thus, since $4 \cdot \gamma \leqslant 1/2$ Corollary 3.25 can be applied which implies $|S| < \gamma n/k^{\ell-1}$.

Upper bound on the number of congested data items. Let S be a maximum set of data items for which there is a request that is congested at level $\ell - 1$. Analogously to the case of crashed requests, we can construct a $(c/4)$-bundle F of S. First, we show that for a sufficiently large α there exist less than a γ-fraction of congested subbutterflies on level $\ell - 1$ for all $\ell \in \{1, \ldots, \log_k n\}$. By Definition 3.29, a subbutterfly on level $\ell - 1$ is congested, if it receives more than $\alpha c k^{\ell-1}/2$ probes for different (d, i) pairs. Since there are at most $(1 - \varepsilon)n$ lookup requests in total, at most $c(1 - \varepsilon)n$ probes arrive at level $\ell - 1$. Thus, at most $c(1-\varepsilon)n/(\alpha c k^{\ell-1}/2) = 2(1-\varepsilon)n/(\alpha k^{\ell-1})$ subbutterflies can be congested at level $\ell - 1$. Since there are exactly $n/k^{\ell-1}$ disjoint subbutterflies at level $\ell-1$, the fraction of congested subbutterflies at level $\ell-1$ is upper bounded by $\frac{2(1-\varepsilon)n/(\alpha k^{\ell-1})}{n/k^{\ell-1}} = 2(1 - \varepsilon)/\alpha$. Hence, for $\alpha > 2(1 - \varepsilon)/\gamma$, at most a γ-fraction of the subbutterflies on level $\ell - 1$ is congested for all $\ell \in \{1 \ldots, \log_k n\}$. That is, all of the congested subbutterflies $BF(s_i^{(l_i)}(d))$ with $(d, i) \in F$ together contain

at most a γ-fraction of the servers on level $\ell - 1$. Again, with Corollary 3.25 we can deduce $|S| < \gamma n/k^{\ell-1}$ for $\gamma = 1/9$. □

3.4.3 Analysis of the Decoding Stage

Analogously to Lemma 3.26, for the Decoding Stage the following lemma holds:

Lemma 3.32. *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/9$. Then, at the beginning of each subphase $\ell \in \{1, \ldots, \log_k n\}$ of the Decoding Stage of the Lookup Protocol of Basic IRIS, the number of requests belonging to level ℓ is at most $\varphi n/k^\ell$ with $\varphi \leqslant 3\gamma k$.*

Proof. Analogously to the proof of Lemma 8 in [AS07], we can show the claim by induction on ℓ. By Lemma 3.26, at most $2\gamma n$ requests belong to level 1 at the beginning of phase 1, which is upper bounded by $\varphi n/k$ for $\varphi \leqslant 3\gamma k$. For the induction step, let $\ell \in \{1, \ldots, \log_k n - 1\}$ and assume that the induction hypothesis holds for level ℓ. We show that the number of requests that will be propagated to level $\ell + 1$ during phase ℓ is upper bounded by $\gamma n/k^\ell$. Since by Lemma 3.26 at most $2\gamma n/k^\ell$ requests belong to level $\ell + 1$, we get that in total at most $2\gamma n/k^\ell + \gamma n/k^\ell$ requests belong to level $\ell + 1$, which is equal to $\varphi n/k^{\ell+1}$ for $\varphi \leqslant 3\gamma k$.

The proof of the induction step is similar to the proof of Lemma 3.26. That is, we upper bound the number of data items with requests belonging to level ℓ by constructing an appropriate bundle and applying expansion properties. With this and Lemma 3.26, we can then derive the claim. In order to determine the number of data items with requests belonging to level ℓ, we first determine the number of congested subbutterflies of dimension ℓ. Note that throughout this proof we use the notion of congested subbutterfly as specified in Definition 3.19. That is, if the subbutterfly $BF(u)$ of a node u is congested, then by Definition 3.19 there exists a node in $BF(u)$ that receives more than βck decode messages for different (d, i) pairs. It holds

$$\beta ck > 3/2 \cdot ck = \frac{3\gamma kc}{2\gamma} \geqslant \varphi c/(2\gamma).$$

Hence, a congested subbutterfly $BF(u)$ of a node u receives more than $\varphi c/(2\gamma)$ decode messages for different (d, i) pairs. By the induction hypothesis there are at most $\varphi n/k^\ell$ requests belonging to level ℓ. For each of these requests, the forwarding of $c/2$ decode messages is initiated at the beginning of phase ℓ. Hence, in total at most $c/2 \cdot \varphi n/k^\ell$ messages arrive at level ℓ, implying that the number of congested subbutterflies of dimension ℓ is less than $(c/2) \cdot (\varphi n/k^\ell)/(\varphi c/2\gamma) = \gamma n/k^\ell$.

Let S be a maximum set of data items with congested requests at level ℓ. Similarly to the proof of Lemma 3.26, there exists a $c/4$-bundle F for S. Since there are less than $\gamma n/k^\ell$ congested subbutterflies of dimension ℓ and since each subbutterfly of dimension ℓ contains k^ℓ nodes, less than γn servers simulate a node of a congested subbutterfly of dimension ℓ, i.e., $|\Gamma_{F,\ell}(S)| < \gamma n$. With Corollary 3.25 it follows $|S| \leqslant \gamma n/k^\ell$.

□

Hence, less than $\Theta(k)$ data items with lookup requests participate in the last phase, phase $\log_k n$, of the Decoding Stage and therefore each node receives in this phase decoding requests for less than $\Theta(k)$ different data items. Thus, there cannot be a congested subbutterfly any more. This, together with the fact that the decoding depth of $BF(k, d)$ is less than $\log_k n$ when crashing at most $\gamma \cdot 2^{\log_k n} = \gamma n^{1/\log\log n}$ nodes with $\gamma < 1/9$, implies that all remaining data items can be decoded at the end.

Chapter 4

Enhanced IRIS

In the following we extend the previously presented system, Basic IRIS, to **Enhanced IRIS** which can handle up to a constant fraction of the servers to be crashed with a redundancy of $O(\log n)$.

Just as Basic IRIS, Enhanced IRIS is based on two articles by Eikel and Scheideler [ES13; ES15]: While the presentation of Enhanced IRIS in [ES13] is rather limited to

While the article in the proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) [ES13] presents the main ideas of the construction of the storage strategy of Enhanced IRIS, the article in the Journal of ACM Transactions on Parallel Computing [ES15] additionally provides a more detailed description of the lookup protocol of Enhanced IRIS.

Theorem 4.1 summarizes the main properties of Enhanced IRIS.

Theorem 4.1 (Enhanced IRIS Main Theorem)**.** *Assume an insider adversary crashes less than εn many servers, with ε being a sufficiently small constant. Then, using only a logarithmic redundancy, Enhanced IRIS correctly serves any set of lookup requests ($O(1)$ per intact server) after at most $O(\log^3 n)$ communication rounds with a congestion of at most $O(\log^3 n)$ at every server in each round, w.h.p.*

Before we describe the encoding strategy of Enhanced IRIS (Section 4.2) and the lookup protocol (Section 4.3) we need to introduce some further preliminaries (Section 4.1).

4.1 Preliminaries

Instead of using a simple parity coding strategy to recover the data of any crashed server within a k-block, we need a more complex coding strategy

that can recover from any two crashed servers within a k-block. Here, we use the EVENODD scheme [Bla+95]. EVENODD is a 2-erasure correcting code that uses only exclusive OR operations and is optimal in terms of redundancy. When using this scheme, we obtain the following results.

Lemma 4.2. *For any k-block B with node sets $(\ell, x_1), \ldots, (\ell, x_k)$ and $(\ell+1, x_1), \ldots, (\ell+1, x_k)$ in which at most two $(\ell+1, x_j)$ are crashed, the information in the remaining nodes $(\ell+1, x_i)$ suffices to recover $d(\ell, x_1), \ldots, d(\ell, x_k)$.*

In order to encode k data items with each other, each of size z, EVENODD adds in total two further data blocks, each of length z. When cutting these blocks into k pieces of equal size (up to an additive 1), we need to append $2z/k$ parity bits (up to an additive 1) to each data block. Hence, for the redundancy achieved via EVENODD, the following Lemma holds.

Lemma 4.3. *For any k-block B with node sets $(\ell, x_1), \ldots, (\ell, x_k)$ and $(\ell+1, x_1), \ldots, (\ell+1, x_k)$ it holds: $|d(\ell+1, x_i)| \leqslant (1 + 2/k)|d(\ell, x_i)|$ up to an additive 1.*

Another aspect in which Enhanced IRIS deviates from Basic IRIS is that the k-blocks are no longer solely organized in a k-ary butterfly. Instead, we additionally make use of permutations with certain expansion properties.

Definition 4.4 (Expansion)**.** *Let U be a set of N nodes that are organized into N/K groups of K consecutive nodes each. Let $\pi : U \to U$ be a permutation of U and let the set of nodes $\pi(U)$ also be organized into N/K groups of K consecutive nodes each. The permutation π is said to have an **expansion** γ, if for any set S of at most $N/(12K^5)$ groups and any set $W \subseteq U$ that contains at least three nodes from each group of S, it holds that the set of nodes $\pi(W)$ contains nodes from at least $\gamma|S|$ many groups from $\pi(U)$.*

Later, for the storage strategy we use permutations in order to describe how to group nodes into blocks of size k. For that purpose, we make use of permutations with a sufficiently high expansion in order to guarantee that the servers are spread "well" across the k-blocks.

See Figure 4.1 for a visualization.

One can show that for sufficiently large N there always exists a permutation on U with an expansion of at least $5/4$.

Lemma 4.5. *Let $N, K \in \mathbb{N}$ and $N \geqslant 12K^5$. Let U be a set of N nodes that are organized into N/K groups of K consecutive nodes each. Then there exists a permutation on U with an expansion of at least $(1 + \delta)$ for a constant $\delta \geqslant 1/4$.*

Proof. Let the permutation π be chosen uniformly at random from all permutations on U. Let $p(s)$ be the probability that there exists a set S of groups with

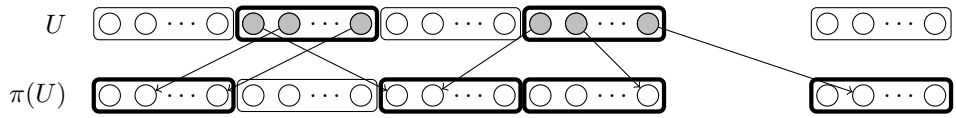

Figure 4.1: Visualization of a permutation π with an expansion of $2/4$. The upper nodes denote the order of the nodes in U, the lower nodes denote the order of the nodes in $\pi(U)$. The rounded rectangles around the nodes denote the groups. The thick rounded rectangles around the upper nodes denote the groups in S, The gray nodes denote the set W. The thick the ones around the lower nodes denote the groups in $\pi(U)$ that contain nodes from $\pi(W)$.

$|S| =: s \leqslant N/(12K^5)$ and a set of triples W from these groups such that $\pi(W)$ contains at most $(1+\delta)s$ many groups. In the following, let $\gamma \leqslant 1/(12K^4)$ such that $s = \gamma N/K$. We will show that $p(s) < 1$, which proves the lemma. $p(s)$ can be upper bounded by:

$$p(s) \leqslant \binom{N/K}{s}\binom{K}{3}^s\binom{N/K}{(1+\delta)s}\left(\frac{(1+\delta)s}{N/K}\right)^{3s} \tag{4.1}$$

where $\binom{N/K}{s}$ is the number of possibilities for choosing s groups, $\binom{K}{3}^s$ the number of possibilities of choosing a triple in each of the selected groups, $\binom{N/K}{(1+\delta)s}$ the number of possibilities for choosing $(1+\delta)s$ groups that the triples have to map to, and $\left(\frac{(1+\delta)s}{N/K}\right)^{3s}$ an upper bound on the probability that all of the triples are indeed mapped to the $(1+\delta)s$ groups. Equation (4.1) is upper bounded by

$$\begin{aligned}
&\left(\frac{e(N/K)}{s}\right)^s\left(\frac{eK}{3}\right)^{3s}\left(\frac{e(N/K)}{(1+\delta)s}\right)^{(1+\delta)s}\left(\frac{(1+\delta)sK}{N}\right)^{3s}\\
\leqslant&\left(\frac{3}{\gamma}\right)^s\left(\frac{e^3K^3}{3}\right)^s\left(\frac{e}{(1+\delta)\gamma}\right)^{(1+\delta)s}((1+\delta)\gamma)^{3s}\\
\leqslant&\left(\frac{e}{\gamma}\right)^s\left(\frac{K^3}{6}\right)^s\left(\frac{e}{(1+\delta)\gamma}\right)^{(1+\delta)s}((1+\delta)\gamma)^{3s}\\
=&\left(\frac{e^{2+\delta}(1+\delta)^{2-\delta}}{6}\right)^s(K^4\gamma)^{(1-\delta)s}
\end{aligned}$$

For $\delta = 1/4$ the first term is at most $4^{(1-\delta)s}$ and the second term is at most $(1/12)^{(1-\delta)s}$, so altogether, $p(s) \leqslant (1/3)^{(1-\delta)s}$. When summing up over all $1 \leqslant s \leqslant N/(12K^5)$, with the geometric sum we get an overall probability of

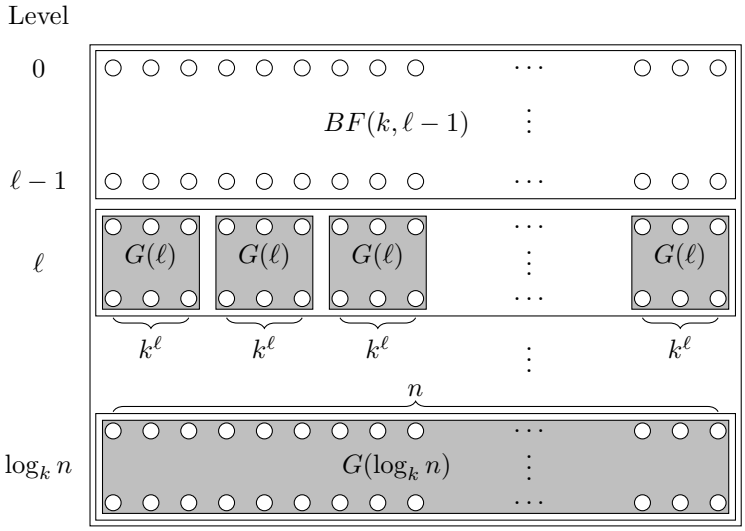

Figure 4.2: Visualization of the underlying topology used in Enhanced IRIS where ℓ denotes the first level with $k^\ell \geqslant 12(\log k)^5$.

less than 0.8 for the expansion of π to be at most $(1 + \delta)$, which completes the proof. □

4.2 Storage Strategy

Similarly to the encoding in Basic IRIS, we introduce an underlying topology that consists of $\log_k n$ main levels where the levels ℓ with $k^\ell \geqslant 12(\log k)^5$ are divided into further sublevels, as described in the following. In general, the description of this topology is divided into three parts depending on ℓ.

Part 1 $k^\ell < 12(\log k)^5$

Part 2 $k^\ell \geqslant 12(\log k)^5$ and $\ell < 6$

Part 3 $k^\ell \geqslant 12(\log k)^5$ and $\ell \geqslant 6$

Figure 4.2 provides a high-level overview of the encoding of the different levels.

Part 1 For the encoding of the data for all levels ℓ with $k^\ell < 12(\log k)^5$, we reuse the encoding via the k-ary ℓ-dimensional subbutterfly as presented in Basic IRIS (Section 3.1). Since k^ℓ asymptotically grows slower than $12(\log k)^5$, the inequality $k^\ell > 12(\log k)^5$ can only hold for k^ℓ constant. Hence, the levels ℓ with $k^\ell < 12(\log k)^5$ can tolerate a constant fraction of crashed nodes in each subbutterfly while still being able to decode all data.

In case of $k^\ell \geqslant (12 \log k)^5$ (Part 2 and Part 3) we introduce n/k^ℓ graphs for each level ℓ, called $G(\ell)$, which are divided into further sublevels, each consisting of k^ℓ nodes. We encode the data items within each $G(\ell)$ by using edge sets between the sublevels with certain expansion properties (defined later). In particular, for each level ℓ we do not proceed with the encoding of the resulting data items from the last (sub)level of the graphs from level $\ell - 1$ but restart the encoding in level ℓ with the original data items.

This level-based encoding approach allows us to define the decoding depth of Enhanced IRIS analogously to the decoding depth of Basic IRIS.

Definition 4.6 (Decoding Depth)**.** *For a graph $G(\ell)$ let L denote the number of levels in $G(\ell)$. The **decoding depth of a node** u at a sublevel $i \in \{0, \ldots, L-1\}$ of $G(\ell)$ is now defined as follows:*

$$dd(u) = \begin{cases} 0 & \text{if } u \text{ is not crashed} \\ \infty & \text{if } i = L-1 \text{ and } u \text{ is crashed} \\ \max\limits_{v \in C(u)} \{dd(v)\} + 1 & \text{if } i < L-1 \text{ and } u \text{ is crashed} \end{cases}$$

*where $C(u)$ denotes the neighbors of the k-block of u in level $i+1$ excluding any two nodes of biggest decoding depth among these neighbors. Analogously to Basic IRIS, the **decoding depth of a server** s_j is defined as $dd(s_j) = dd((0, j))$.*

Note that if the decoding depth of a node u at sublevel i in $G(\ell)$ is more than d with $i + d \leqslant L$, then it must be possible to embed a complete *ternary* tree of crashed nodes with root u and depth d in $G(\ell)$.

Part 2 Now, suppose that ℓ satisfies $k^\ell \geqslant 12(\log k)^5$ and $\ell < 6$. In order to describe the encoding of the data items on such a level ℓ, for each block of k^ℓ consecutive nodes on level ℓ we introduce a graph $G(\ell)$ that consists of $L_1 = 20 \log k$ sublevels, each consisting of k^ℓ nodes. In order to construct $G(\ell)$, we choose a permutation $\pi_\ell^{(1)}$ that has an expansion of at least $5/4$ for $N = k^\ell$ nodes and a group size of $K = K_1 = \log k$. In the following let (i, x) denote the virtual node from sublevel i and column x in $G(\ell)$. Partition the nodes of each sublevel of $G(\ell)$ into groups of K_1 consecutive nodes. Each node (i, x) in some group B in sublevel i of $G(\ell)$ is connected to all nodes $(i+1, \pi_\ell^{(1)}\ell(y))$ with $(i, y) \in B$. This establishes complete bipartite graphs of K_1 nodes on sublevel i and $i+1$, called K_1-*blocks* (see Figure 4.3). $G(\ell)$ is simulated by N servers with server s_i simulating the L_1 nodes $(0, i)$, $(1, \pi_\ell^{(1)}(i))$, $(1, \pi_\ell^{(1)}(\pi_\ell^{(1)}(i)))$, and so on.

We are now ready to describe the encoding of a set of K_1 data blocks $d_0, \ldots, d_{K_1-1}$. Initially, d_j is placed in node $(0, j)$ of $G(\ell)$, for all $j \in \{0, \ldots, K_1 -$

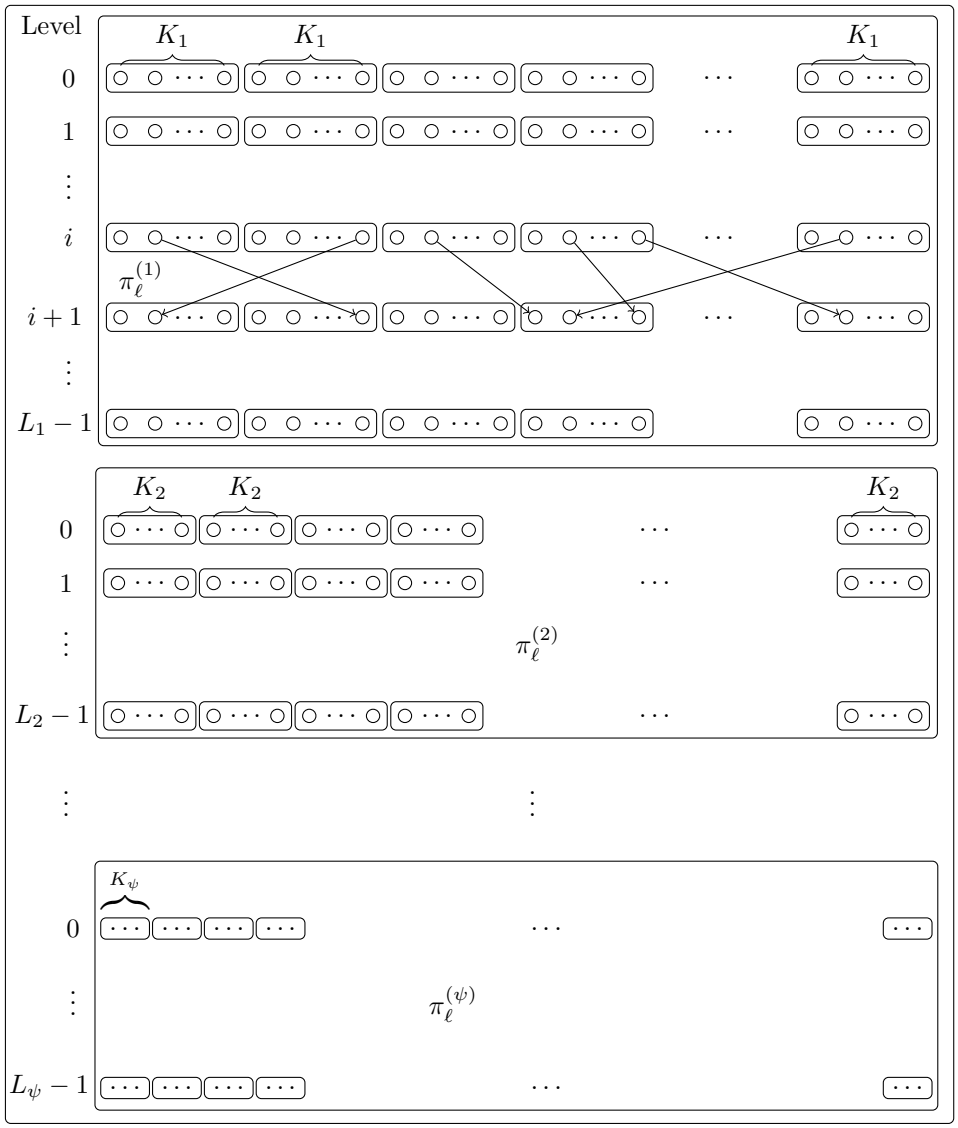

Figure 4.3: Visualization of $G(\ell)$ and its K_i-blocks with $\psi \leqslant \log^* k$.

$1\}$. Given that in sublevel i, $i \in \{0, \ldots, L_1 - 1\}$, we have already assigned data blocks $d(i, j)$ to the nodes (i, j), $j \in \{0, \ldots, K_1 - 1\}$, for each K_1-block B of sublevel i, we compute the data blocks for sublevel $i + 1$ using the EVENODD coding strategy and assign them to the nodes of that K_1-block in sublevel $i + 1$.

In the following our goal is to extend $G(\ell)$ by adding more sublevels to it such that whenever a data block encoded in $G(\ell)$ cannot be recovered, at least a constant fraction of the servers emulating $G(\ell)$ must be crashed.

Suppose the data of a node $(0, x)$ in $G(\ell)$ cannot be recovered: i.e., the decoding depth of $(0, x)$ is larger then L_1. Hence, $G(\ell)$ contains a ternary tree with root $(0, x)$ and depth L_1 that consists only of crashed nodes. Unfortunately, the leaves of this tree are not guaranteed to be distinct anymore like for the binary witness trees in the k-ary butterfly. But, due to the expansion property of $\pi_\ell^{(1)}$, we know that this ternary tree must cover at least $3(5/4)^{L_1-1}$ crashed servers at its leaves. Since by Definition 4.4 the number of groups considered for the computation of the expansion is upper bounded by $N/(12(K_1)^5)$ we get that at least $\min\{3(5/4)^{L_1-1}, N/(12K_1^5)\}$ crashed servers are covered by the nodes at sublevel L_1 in $G(\ell)$. Furthermore, it holds

$$3(5/4)^{L_1-1} \overset{(*)}{\geqslant} 3(5/4)^{4\ell \log k - 1} \overset{(**)}{\geqslant} (12/5)k^\ell \overset{(***)}{\geqslant} N/(12\log^5 k)$$

where in $(*)$ we used $L_1 = 20 \log k \geqslant 4\ell \log k$, in $(**)$ we used $(5/4)^{4\ell \log k} \geqslant k^\ell$, and $(***)$ holds due to $N = k^\ell$ and $12 \log^5 k \geqslant 5/12$. Thus, at least a $1/(12K_1^5)$-fraction of the N servers simulating $G(\ell)$ is crashed. Now, consider the following two cases:

Case 1 $K_1^\ell \leqslant 12(\log K_1)^5$. In this case K_1 is a constant and therefore a constant fraction of the servers simulating $G(\ell)$ is crashed.

Case 2 $K_1^\ell > 12(\log K_1)^5$. Define $K_2 = \log K_1 = \log\log k$, $L_2 = 20 \log K_1$ and add L_2 further additional sublevels to $G(\ell)$ such that $G(\ell)$ now consists of $L_1 + L_2$ sublevels in total. Using a permutation $\pi_\ell^{(2)}$ with an expansion of $5/4$ for $N = k^\ell$ nodes and group size $K = K_2$, we connect the nodes between level i and $i + 1$ of $G(\ell)$ for $i \in \{L_1, L_1 + 1 \ldots, L_1 + L_2 - 2\}$ to complete bipartite graphs just as for the first L_1 levels of $G(\ell)$. The encoding in the next L_2 sublevels now works as follows: First, assign the data blocks $d(L_1 - 1, j),\ j \in \{0, \ldots, n - 1\}$, that have already been assigned to level $L_1 - 1$ of $G(\ell)$ to the nodes (L_1, j). Next, for each K_2-block at sublevel $i \in \{L_1, \ldots, L_1 + L_2 - 2\}$ we compute the data blocks for sublevel $i + 1$ using the EVENODD coding strategy and assign them to the nodes of that K_2-block in sublevel $i + 1$.

With the same arguments from above, the number of crashed servers that are now covered by the nodes at sublevel $L_1 + L_2$ in $G(\ell)$ increases

to at least $N/(12K_2^5)$. Again we need to consider two cases: If $K_2^\ell \leqslant 12(\log K_2)^5$, then, analogously to case 1, K_2 is a constant, implying that a constant fraction of the servers emulating $G(\ell)$ is crashed.
If $(K_2)^\ell > 12(\log K_2)^5$ we continue with the extension of $G(\ell)$ as for K_1 and K_2: i.e., define $K_3 = \log K_2$, $L_3 = 20 \log K_2$ and add L_3 further additional sublevels to $G(\ell)$. Just as for the case of K_1 and K_2 interconnect the K_3-blocks of the sublevels with each other using a permutation $\pi_\ell^{(3)}$ with an expansion of $5/4$ for N nodes and group size $K = K_3$.

We continue with the process of extending $G(\ell)$ after until after at most $\psi := \log^* k$ extensions[1] K_i is a constant, implying that a constant fraction of the servers emulating $G(\ell)$ is crashed whenever the data of some node $(0, x)$ in $G(\ell)$ cannot be recovered.

Part 3 It remains to describe the encoding of the data blocks on level ℓ with $k^\ell \geqslant 12(\log k)^5$ and $\ell \geqslant 6$. Similarly to the construction in Part 2 we first construct a graph $G(\ell)$ using a permutation $\pi_\ell^{(1)}$ with $N = k^\ell$ nodes, group size $K = K_1 = k$, and $L_1 = 4 \log N$ levels to get the number of crashed servers in $G(\ell)$ up to $N/(12K_1^5)$. Then we continue with the $\log^*$-construction as in part 2 (starting with $K_1 = \log k$ and $L_1 = 20 \log k$), until we get the number of crashed servers up to a constant fraction before the decoding of a data block can fail.

Note that in Basic IRIS we encode in each level $\ell \in \{2, \ldots, \log_k n - 1\}$ those data blocks with each other that are the result of the encoding of the previous level $\ell - 1$. That is, we also re-encode the previously computed parity bits in the k-blocks. In contrast to this, in Enhanced IRIS for each graph $G(\ell)$ the encoding of the data blocks is initiated with the original data blocks that are supposed to be encoded in this layer instead.

By this, we achieve a logarithmic redundancy for Enhanced IRIS.

Lemma 4.7. *Enhanced IRIS has an overall redundancy of $O(\log n)$.*

Proof. In the following let z denote the maximum size of the input data blocks that are supposed to be encoded with each other. First, we upper bound the amount of information a server s stores for the encoding of a single graph $G(\ell)$. Let $S_1(i)$, $i \in \{0, \ldots, L_1 - 1\}$, denote the amount of information server s stores for the encoding of the first $i + 1$ sublevels of $G(\ell)$. By Lemma 4.3, it holds: $S_1(i) \leqslant (1 + 2/k)S_1(i - 1)$ for all $i \in \{1, \ldots, L_1\}$. Since $S_1(0) = z$, we get $S_1(i) \leqslant (1 + 2/k)^i z$. Hence, the amount of data $S_1(L_1)$ that server s stores after the encoding of the first $L_1 + 1$ sublevels of $G(\ell)$ is upper bounded by $(1 + 2/k)^{L_1} z \leqslant e^{2L_1/k} z = e^{40 \log k/k} z = O(z)$. By the same arguments the

[1] $\log^* n$ is the number of times the logarithm has to be applied to n until the result is at most 2.

amount of data that server s stores after the encoding of the first $\sum_{i=1}^{\psi}(L_i+1)$ sublevels of $G(\ell)$ is upper bounded by $\Pi_{i=1}^{\psi}(1+2/k)^{L_i}z$. Since there are at most $\log^* k$ level extensions in $G(\ell)$, the overall amount of data that server s stores after the encoding of one complete $G(\ell)$ graph is upper bounded by $\Pi_{i=1}^{\log^* k}(1+2/k)^{L_i}z = z \cdot 2^{O(\log^* k)}$. Hence, for all $O(\log_k n)$ graphs $G(\ell)$ the overall amount of data stored at s after the complete encoding of the underlying topology used in Enhanced IRIS is upper bounded by $z \log_k n \cdot 2^{O(\log^* k)} = O(z \log n)$, which proves the claim. □

4.3 Lookup Protocol

Recall that the lookup protocol of Basic IRIS consists of three stages: the Preprocessing Stage, the Probing Stage, and the Decoding Stage. In the following we describe how to adapt these stages in order to work for Enhanced IRIS.

4.3.1 Preprocessing Stage

Analogously to Basic IRIS, the Preprocessing Stage consists of two substages: the Recovery Stage and the Decoding Depth Computation Stage. In the Recovery Stage we first determine a unique representative for each crashed server, such that at the end each intact server knows the representative of each crashed server it is connected to in the k-ary butterfly. This can be done in the same manner as in Basic IRIS. Afterwards the $G(\ell)$ graphs are recovered. That is, each intact server is introduced to the representative of each crashed server it is connected to in any K_i-block of any $G(\ell)$ graph. Since there is no deterministic computation rule for the K_i-blocks in the $G(\ell)$ graphs, each server additionally needs to store to which server it is connected in the underlying topology of the storage strategy created by the $G(\ell)$ graphs. For that purpose, note that for each graph $G(\ell)$ in each group of L_i levels, $i \in \{1,\dots,\varphi\}$ with $\varphi \leqslant \log^* k$, each server is connected to $(L_i-1)\cdot K_i$ other servers. Since $K_1 = L_1 = \Theta(\log k)$ and $K_1 = \max\{K_2,\dots,K_\varphi\}$, $L_1 = \max\{L_2,\dots,L_\varphi\}$, we get that each server is connected to $O(\log^* k \log^2 k)$ other servers in $G(\ell)$. Hence, in in total each server needs to store $O(\log_k n \log^* k \log^2 k)$ connections. But this does not cause a problem, since in any case each server additionally stores information about each other server in the system (e.g., addresses of the servers). In order to recover the $G(\ell)$ graphs, each intact server s sends a message to each server s' it is connected to in any of the $G(\ell)$ graphs by routing the message $(\mathsf{id}(s), \mathsf{id}(s'))$ along the unique path in the k-ary butterfly from s to s'. Since the k-ary butterfly has already been recovered correctly, eventually this message reaches the server s' in case s' is not crashed and otherwise the representative $\mathsf{rep}(s')$ of s'. This initiates s' (or $\mathsf{rep}(s')$) to forward the message $(\mathsf{id(s')}, ID)$ back to s along

the unique path from s' to s in the k-ary butterfly, where $ID = \mathsf{id}(s')$ in case s' is not crashed and $ID = \mathsf{id}(\mathsf{rep}(s'))$ otherwise. Since all intact servers forward their messages in the k-ary butterfly, by the Borodin Hopcraft bound [BH82] a congestion of $\Omega(\sqrt{n}/k)$ may occur at any intact server. Using the analysis of Valiant's trick [Val82] one can show that there exist permutations with the desired expansion properties (as defined above) that additionally guarantee a congestion of at most $O(\log n)$ at each node in each round of the routing strategy described above.

Once the underlying topology has been recovered, the decoding depth of the nodes in $G(\ell)$, as defined in Definition 4.6, can be computed analogously to the decoding depth in Basic IRIS. Besides the decoding depth of a node at the levels in $G(\ell)$, each node from each $G(\ell)$ also needs to compute the decoding depth of $G(\ell)$, which is analogously defined to the decoding depth of a subbutterfly in Basic IRIS. That is, $\mathrm{dd}(G(\ell)) = \max\{\mathrm{dd}(u) \mid u \text{ is a node on level } 0 \text{ in } G(\ell)\}$. Since the computation of $\mathrm{dd}(G(\ell))$ for a single graph $G(\ell)$ takes $depth(G(\ell))$ rounds after in total $\log_k n \log^* k \log k$ rounds each server that emulates a node from $G(\ell)$ is aware of $\mathrm{dd}(G(\ell))$.

The following lemma is easy to check.

Lemma 4.8. *The Preprocessing Stage takes at most $O(\log_k n \log^* k \log k)$ communication rounds with at most $O(\log^2 n)$ congestion at every intact server at each round. Furthermore, at the end of the Preprocessing Stage it holds:*

1. *Each intact server knows the representatives of all crashed servers it is connected to in the k-ary butterfly and the ones it is connected to in any of the $G(\ell)$ graphs.*
2. *Each intact server that emulates a node from any $G(\ell)$ knows $dd(G(\ell))$.*

4.3.2 Probing Stage

The purpose of the Probing Stage is to determine for each lookup request the level $\ell \in \{0, \ldots, \log_k n\}$ it belongs to (as defined later). Analogously to Basic IRIS, the Probing Stage consists of $\log_k n$ rounds. First, each intact server that received a lookup request for some data item d chooses c intact servers $s_1(d), \ldots, s_c(d)$ uniformly at random. The following rounds are dedicated to the forwarding of a (d, i) probe, for each $i \in \{1, \ldots, c\}$, along the unique path in the k-ary butterfly from the node on level $\log_k n$ that is emulated by $s_i(d)$ to the node on level 0 that is emulated by the server responsible for $h_i(d)$. As in Basic IRIS, we call this path the **probing path** of d_i. Additionally, also just as in Basic IRIS, during the forwarding of the probes the technique of combining and splitting is used. In each round $r \in \{0, \ldots, \log_k n\}$ each node u that received a probe message determines whether it is congested or crashed at level r. A node u that received probe messages is called **congested** if it

receives more than $\alpha \log n$ probes for different (d, i) pairs (for a sufficiently large constant α) in the current round. Different from Basic IRIS, in Enhanced IRIS we say a node u is **crashed** at level r if $\mathrm{dd}(G(u)) = \infty$, where $G(u)$ denotes the graph $G(\log_k n - r)$ that is (besides other servers) emulated by the server that emulates u. If u is congested or crashed, u deactivates all (d, i)-probes it received in the current round by sending the message $(\mathsf{fail}, d, i, \log_k n - r)$ to all roots of these probes (by using the technique of splitting if necessary). Otherwise, u forwards each probe to the next nodes of the according paths on level $\log_k n - r - 1$ (by using the technique of combining if necessary). After at most $O(\log_k n)$ rounds, each server s that received a lookup request for some data item d has received a fail-message for all of its probes that have been deactivated. Using this information, s defines its lookup request to belong to the minimum level such that less than $c/2$ of its probes have been deactivated at that level or higher. Hence, at the end of the Probing Stage, each lookup request belongs to a level $\ell \in \{0, 1, \ldots, \log_k n\}$. All lookup requests that belong to level 0 can immediately be answered, while all lookup requests belonging to a level $\ell > 0$ will be handled in phase ℓ of the Decoding Stage.

It is easy to see that the Probing Stage of Enhanced IRIS satisfies the following property.

Lemma 4.9. *The Probing Stage of the Lookup Protocol of Enhanced IRIS takes at most $O(\log_k n)$ communication rounds with at most $O(\log^2 n)$ congestion at every server in each round.*

The following lemma can be shown analogously to Basic IRIS (Lemma 3.26).

Lemma 4.10. *Assume an insider adversary crashes at most εn servers, for a sufficiently small constant $\varepsilon > 0$. Then, at the end of the Probing Stage of the Lookup Protocol of Enhanced IRIS, the number of requests belonging to level $\ell \in \{1, \ldots, \log_k n\}$ is at most $\gamma n / k^{\ell - 1}$, for a sufficiently small constant γ.*

4.3.3 Decoding Stage

Analogously to Basic IRIS, the Decoding Stage is divided into $\log_k n$ phases where phase $\ell \in \{0, \ldots, \log_k n\}$ is dedicated to the decoding of the data items with lookup requests that belong to level ℓ. In phase $\ell \in \{0, \ldots, \log_k n\}$ each server s that received a lookup request for some data item d that belongs to level ℓ performs or initiates the following tasks:

1. Choose $c/2$ pairs (d, i) that have not been deactivated in the Probing Stage until level ℓ. For $i \in \{1, \ldots, c\}$ let $G_i(\ell)$ denote the graph from level ℓ that is (besides other servers) emulated by the server responsible for $h_i(d)$.

2. For each of the $c/2$ previously chosen (d, i) pairs, determine whether $G_i(\ell)$ could be decoded without nodes from $G_i(\ell)$ becoming congested, i.e., receiving more than $O(ck)$ decode requests for different (d, i) pairs. If any node from $G_i(\ell)$ would become congested when decoding $G_i(\ell)$, we call $G_i(\ell)$ *congested*. For a graph $G_i(\ell)$ let $BF(G_i(\ell))$ denote the k-ary butterfly that consists of the same servers as $G_i(\ell)$. Determining whether $G_i(\ell)$ is congested can be done just like determining whether the k-ary subbutterfly $BF(G_i(\ell))$ is congested, as already described in Basic IRIS by performing a broadcast in $BF(G_i(\ell))$.

3. If less than $c/4$ of the $G_i(\ell)$ graphs are congested, initiate the decoding of $c/4$ of the non-congested $G_i(\ell)$ graphs. Since the decoding depth of each $G_i(\ell)$ graph considered in level ℓ of the Decoding Stage is not exceeded, it is possible to completely decode $G_i(\ell)$ and retrieve the requested data pieces.

If at the end of phase ℓ server s receives at least $c/4$ decoded pieces, then s can recover the requested data item. Otherwise, s denotes the request to belong to level $\ell + 1$ and handles it again in the next phase, phase $\ell + 1$.

It is easy to see that the Probing Stage of Enhanced IRIS satisfies the following property.

Lemma 4.11. *The Decoding Stage of the Lookup Protocol of Enhanced IRIS takes at most $O(\log_k^3 n)$ communication rounds with at most $O(\log^3 n)$ congestion at every server in each round.*

Analogously to Basic IRIS (Lemma 4.12), one can show the following lemma.

Lemma 4.12. *Assume an insider adversary crashes at most εn servers, for a sufficiently small constant $\varepsilon > 0$. Then, at the beginning of each subphase $\ell \in \{1, \ldots, \log_k n\}$ of the Decoding Stage of the Lookup Protocol of Enhanced IRIS, the number of requests belonging to level ℓ is at most $\varphi n/k^\ell$ with $\varphi = \Theta(k)$.*

Lemma 4.12 implies that all remaining data items can be decoded at the end without causing excessive congestion at any node.

All in all, Lemmas 4.8, 4.9, 4.11, and 4.12 now imply the main theorem, Theorem 4.1.

Chapter 5

RoBuSt

In this chapter we present RoBuSt, a crash-failure-resistant distributed storage system. Just as Basic IRIS, RoBuSt tolerates an insider adversary that knows everything about the system and can use this knowledge in order to crash a huge fraction of all servers. While Basic IRIS and Enhanced IRIS are restricted to the serving of lookup requests, RoBuSt can additionally handle write requests correctly. RoBuSt is based on an article by Eikel et al. that has been published in the proceedings of the 18th International Conference on the Principles of Distributed Systems (OPODIS) [ESS14].

Recall that Basic IRIS partitions the set of data items that are supposed to be stored in the system into sets of n data items each. These sets are separately encoded with each other into so-called layers (cf. Section 3.2). In order to answer lookup requests for a set of data items, Basic IRIS first tries to answer these requests by forwarding the request for the c pieces of each requested data item through the k-ary butterfly towards the storage location (cf. Section 3.3.2). The requests that could not be answered during this approach are further handled by trying to recover according parts of the layers (cf. Section 3.3.3). We were able to show that by this approach each request is answered within a polylogarithmic number of rounds with only a polylogarithmic work for each server in each round (cf. Section 3.4).

Unfortunately, the situation is different when considering write requests. This is due to the fact that whenever a data item is added or changed, the complete corresponding layer has to be re-encoded. This becomes problematic in case the handling of a set of write requests requires more than a polylogarithmic number of layers to be re-encoded. In particular, when handling a set of $O(n)$ write requests it may happen that $\Theta(n)$ layers have to be re-encoded,

which is too expensive for our purposes. In order to circumvent this problem, RoBuSt stores its data items into so-called buckets that hold $O(n)$ data items each and that are arranged in a tree structure. Furthermore, for each data item d there is not only a single bucket in which to store that data item, but a logarithmic number of buckets that are a potential storage location. The internal storage strategy of a single bucket is very similar to the storage strategy of a single layer: i.e., we reuse the Butterfly Coding Strategy here. The main new idea of RoBuSt is the arrangement of the buckets and the way of traversing the constructed bucket tree and extracting and inserting data items into it for the handling of write requests.

All in all, in this chapter we show the following theorem.

Theorem 5.1 (RoBuSt Main Theorem)**.** *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/24$. Then, using only a logarithmic redundancy, RoBuSt correctly serves any set of lookup and write requests (at most $O(1)$ per intact server) after at most $O(\log^3 n)$ communication rounds with a congestion of at most $O(\log^3 n)$ at every server in each round, w.h.p.*

The further structure of this chapter is as follows: In Section 5.1 we recap the model properties that were assumed in Basic IRIS and that also hold for RoBuSt, and we emphasize differences in the model. Afterwards, in Section 5.2 we present the storage strategy which is used by RoBuSt. Section 5.3 provides a description of the Write Protocol, followed by a description of the Lookup Protocol in Section 5.4. Since the correctness analysis of the Lookup Protocol is rather involved we postponed it to a separate section (Section 5.5).

5.1 Preliminaries

Just as in Basic IRIS, we assume that time is divided into periods, where one period is sufficiently long in order to serve a set of lookup and write requests. We consider a batch-based insider adversary, which may at the beginning of each period select up to $\gamma n^{1/\log\log n}$ servers to be crashed during that period with $\gamma = 1/24$. Besides lookup requests of the form $\mathsf{lookup}(k)$ for $k \in \mathcal{U}$, we also allow write requests which are of the form $\mathsf{write}(k, d)$ for $k \in \mathcal{U}$ and a data item d. Note that via write requests users can also modify or remove data from the system. We assume the size of the universe to be polynomial in n, i.e., $m := |\mathcal{U}| = n^p$, for a constant p. That is, $\Lambda := p \log n$ bits are required for addressing the keys. For a data item d, we define the **address** of d by $\mathsf{key}(d) = d_{\Lambda-1} \dots d_1 d_0 \in \{0,1\}^\Lambda$ and let $\mathsf{bit}_i(d) := d_i$. Analogously to Basic IRIS, in order to achieve a logarithmic storage redundancy we require the size of data items to be at least $\Omega(\log n)$.

An additional challenge we have to deal with when considering write requests is the existence of so-called outdated servers. We denote a server s to

be **outdated**, if s holds data items or information that is not up-to-date. Such a situation may occur when data is written into the system while the server s is crashed. In this case, once s is not crashed anymore, s would not be aware of data that has been added to the system or just modified during the periods in which s was crashed. In order to deal with outdated servers, we will make use of timestamps whenever data is written in the system. For this purpose, we assume the existence of an internal clock at each server in order to store timestamps whenever data is (re-)written. Here, we only require the internal clocks to run at the same frequency such that the servers have a common understanding of the beginning and end of the rounds. In particular, we do not require the internal clocks to be synchronized. However, as we will describe later, it is still possible for the servers to detect whether they are outdated. As we will see later, for our purposes it also suffices to simply store a hash value whenever data is (re-)written instead of storing a timestamp. In this context, we only require the hash function for creating these hash values to be collision free.

Table 5.1 provides an overview of variables and their bounds that are commonly used in this chapter.

Term	Bound	Description
p		Exponent in n^p which denotes the address length
Λ	$= p \log n$	Number of zones/depth of bucket tree/address length
γ	$= 1/24$	Constant in fraction of crashed servers from $n^{1/\log\log n}$ servers
ε	$< \gamma n^{1/\log\log n - 1}$	Fraction of crashed servers
α	$> (1-\varepsilon)/\gamma$, e.g., $\geqslant 12$	Constant in congestion bound in Probing Stage
β	$> 1/10$	Constant in congestion bound in Decoding Stage
c	$\geqslant 12 \log m$	Number of pieces created for each data item
$\mathcal{D}(B)$		Set of data items stored in bucket B

Table 5.1: Variables commonly used in the presentation of RoBuSt.

5.2 Storage Strategy

Our data structure is based on a binary tree with $\Lambda + 1$ levels, also called **zones**. We call the nodes of each zone **buckets** where each bucket will hold a set of data items. Before we describe the internal storage strategy of the single buckets, we present the underlying structure of the data structure formed

by the buckets, called the **bucket tree** (see Figure 5.1). Zone 0 consists of a single bucket, bucket B_ε. Each bucket B that is not in zone Λ has two children, denoted by 0-child(B) and 1-child(B). For each data item d there is not only a single possible bucket in which to store d, but $\Lambda + 1$ possible buckets, one in each zone. Bucket B_ε may hold any data item. Bucket 0-child(B_ε) from zone 1 may hold all data items d with $\mathsf{bit}_0(d) = 0$, bucket 1-child(B_ε) may hold all data items d with $\mathsf{bit}_0(d) = 1$. In general, for a bucket B in zone z, the bucket 0-child(B) in zone $z + 1$ may hold all data items d that may be held by B with $\mathsf{bit}_z(d) = 0$; the bucket 1-child(B) in zone $z + 1$ may hold all data items d that may be held by B with $\mathsf{bit}_z(d) = 1$.

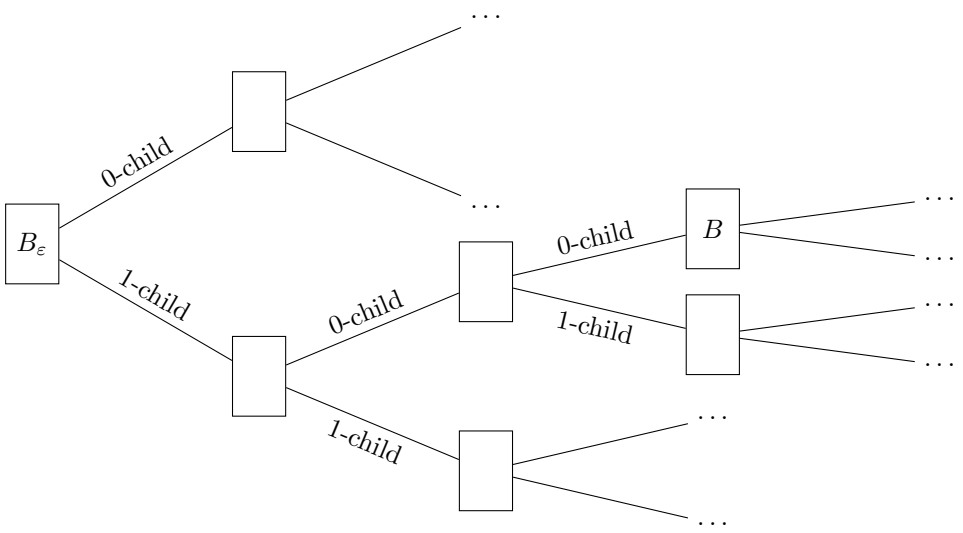

Figure 5.1: Visualization of the bucket tree. Here, B may hold all data items d whose three least significant bits are 001.

In the following, let $\mathcal{B}$ be the set of all buckets and let $\mathsf{bucket}(z, d) : \{0, \ldots, \Lambda\} \times U \to \mathcal{B}$ be a function that returns the unique possible bucket of a data item d at zone z. Initially, each bucket does not contain any data. During the runtime of the system the following invariant is satisfied: Each bucket, excluding bucket B_ε, stores either 0 or between n and $2n$ data items. Bucket B_ε stores at most $2n$ data items.

In the following, we present the internal storage strategy of the single buckets. The idea of storing a set D of data items into a bucket B is to reuse the basic concepts of the storage strategy for individual layers from IRIS. In contrast to IRIS, we additionally store a timestamp $t(B)$ for bucket B, which is used for the handling of outdated information a server might hold if it was crashed in a previous period in which write requests were served. While in IRIS we

used c fixed hash functions in order to map the data pieces to the servers, in RoBuSt each bucket holds its c own hash functions and whenever a bucket is (re-)written, we choose c new hash functions uniformly at random for that bucket.

In short, whenever a set D of data items is supposed to be stored into a single bucket B, from a central point of view the following steps are performed by the servers:

Step 1: Create $c \geqslant 18 \log m$ pieces $d_1, \ldots, d_c$ for each data item $d \in D$ using Reed-Solomon codes.

Step 2: Choose c hash functions $h_1, \ldots, h_c : \mathcal{U} \to \mathcal{S}$ uniformly and independently at random and map the pieces of each data item $d \in D$ to the servers using these hash functions.

Step 3: Encode the data pieces created in Step 1 with each other using the k-ary Butterfly Coding Strategy (see Section 3.1).

Step 4: The servers agree on a common timestamp t for bucket B. Each server then defines its timestamp $t_s(B)$ for bucket B as $t_s(B) = t$.

Since the hash functions in Step 2 are chosen uniformly at random every time data is written into a bucket, with Chernoff bounds (Lemma 2.1) it follows that at the end of Step 2 each server holds $\Theta(c)$ data pieces, w.h.p.

Recall that the k-ary Butterfly Coding Strategy required an internal error correcting code for the encoding of a single k-block. In RoBuSt we reuse the internal error correcting code that was also used in Basic IRIS (Section 3.2.1). At the end of Step 4, several data pieces and parity information have been mapped to each server s_i for bucket B. In the following $b_i(B)$ denotes the concatenation of these data pieces and parity information to a single data block. If it is clear from the context, we may also omit the B and just write b_i.

In the following, just as in IRIS, let $BF(k, d)$ be a k-ary butterfly with $n = k^d$ and with server s_i, $i \in \{0, \ldots, n-1\}$, emulating the butterfly nodes $(0, i), \ldots, (d, i)$. By Lemma 3.4 of IRIS, the storage amount of each server s_i, $i \in \{0, \ldots, n-1\}$, required for the encoding of a single layer is upper bounded by $(1 + e)z$, where z denotes the maximum size of the data blocks stored at any server s_j, $j \in \{0, \ldots, n-1\}$. Since in RoBuSt we additionally only hold c hash functions and a timestamp for each bucket, we get that the storage amount of each server of a single bucket in RoBuSt is also upper bounded by $(1+e)z$. Since there may exist outdated data items in the system, but for each zone of the bucket tree at most one, there are at most $\Lambda + 1 = O(\log n)$ many copies of each data item in the system, as described further later, which implies the following lemma.

Lemma 5.2. *RoBuSt has an overall redundancy of $O(\log n)$.*

5.3 Write Protocol

In the following let D be the set of data items for which intact servers received write requests where each intact server receives $O(1)$ write requests. For a data item d that is stored in the system, denote the c pieces that have been created from d using Reed-Solomon coding as $d_1, \ldots, d_c$. Furthermore, denote the server that is holding d_1 (after the pieces have been distributed among the n servers) as the server **maintaining** d.

The write protocol is divided into two stages: The Preprocessing Stage and the Writing Stage. The goal of the Preprocessing Stage is to determine for each crashed server a unique representative from the set of intact servers just as it is done in the Butterfly Completion Stage of Basic IRIS (Section 3.3.1). In contrast to IRIS, we do not need to compute the decoding depth here that gives information about the minimum level of the butterfly that the decoding must be initiated from. Instead, we perform a similar check online during the Writing Stage. In the Writing Stage we mainly actually write the data items into the buckets by a top-down traversal through the bucket tree.

5.3.1 Preprocessing Stage

In this stage, for each crashed server s, a unique intact server is determined, called the **representative** of s, such that at the end of this stage each intact server is the representative of at most two servers. The idea of the representatives is to let them take over the role of the according crashed servers in actions (e.g., routing, computations) that the crashed servers are supposed to perform. For this, we additionally need to ensure that each intact server knows the representatives of all crashed servers it is connected to in the underlying k-ary butterfly.

The determination of the representatives and the introduction of the representatives to the appropriate servers can be done in the same manner as in the Butterfly Completion Stage of IRIS (Section 3.3.1). By Lemma 3.12 of IRIS, this takes at most $(2 + o(1)) \log n$ rounds with a congestion of at most $O(\log n)$ at each server in each round.

In order to keep the presentation of RoBuSt as clear as possible, in the following with s_i we refer to the server s_i itself if s_i is intact and we refer to the representative of s_i if s_i is crashed.

5.3.2 Outline of the Writing Stage

In the following, for a bucket B the set $\mathcal{D}(B)$ denotes the set of data items that are stored in bucket B. The idea of inserting a set D of data items into the systems is as follows: First, we try to insert the data items in bucket B_ε. For this purpose we check whether B_ε does not contain "too many" data items.

If this is not the case, we insert all data items from D into B_ε. Otherwise, we choose a new set of n data items D_1 from $\mathcal{D}(B_\varepsilon) \cup D$ with the property that these data items belong to the same bucket B_1 in zone 1. We remove the data items in D_1 from B_ε and add all data items from $D \backslash D_1$ to B_ε. By this, no data item from D or B_ε is discarded. Next, we proceed with D_1 and B_1 just as we did with D and B_ε. That is, if B_1 does not contain "too many" data items, we insert all data items from D_1 into B_1. Otherwise, we select an appropriate set D_2 of data items from $\mathcal{D}(B_1) \cup D_1$ to be handled for a bucket B_2 in zone 2. This traversal through the bucket tree along a single path proceeds until a bucket has been found that does not contain "too many" data items. As we will discuss later, this approach actually terminates at the latest at the last zone of the bucket tree.

In order to keep the specification of our system simple, we first give a high-level overview of how to process a set of write requests before describing the details of the steps necessary in the following subsections. The Writing Stage consists of up to $\Lambda + 1$ phases. Each phase $z \in \{0, \ldots, \Lambda\}$ deals with a single bucket B_z from zone z only and receives a set of data items D_z to be inserted into B_z. At the beginning, $D_0 := D$ is the set of all data items for which there are write requests. Phase $z \in \{0, \ldots, \Lambda\}$ consists of the following steps:

Step 1: Completely decode B_z and send all decoded pieces of a data item $d \in \mathcal{D}(B_z)$ to the server maintaining d (see Section 5.3.3).

Step 2: If $|\mathcal{D}(B_z) \cup D_z| \leqslant 2n$: Add the data items from D_z to $\mathcal{D}(B_z)$, choose c new hash functions $h_1, \ldots, h_c : U \to \mathcal{S}$ uniformly at random for B_z, and re-encode B_z (see Section 5.3.4).

Step 3: Else ($|\mathcal{D}(B_z) \cup D_z| > 2n$):

a) The intact servers agree on a subset $D_{z+1} \subseteq \mathcal{D}(B_z) \cup D$ of size n with the property that for all $d, d' \in D_{z+1}$, $\mathsf{bit}_z(d) = \mathsf{bit}_z(d') = b \in \{0, 1\}$ (see Section 5.3.5).

b) Re-encode the data items in $(D_z \cup \mathcal{D}(B)) \backslash D_{z+1}$ in bucket B_z and choose c new hash functions $h_1, \ldots, h_c : U \to \mathcal{S}$ uniformly at random for B_z (see Section 5.3.3).

c) Set B_{z+1} :=0-$\mathsf{child}(B_{z+1})$ if $b = 0$ and B_{z+1} :=1-$\mathsf{child}(B_{z+1})$ if $b = 1$ and propagate the data items in D_{z+1} to the next phase.

Note that within a single period only a few of all buckets are (re-)written. Hence, in particular, a server can be outdated w.r.t. a specific bucket, while not being outdated for other buckets. In order to illustrate this, consider the following scenario: Assume a server s holds a piece of a data item d in bucket B and d is supposed to be modified in a period p in which s is crashed. Then

in period p, s cannot update its version of its piece of d as it is supposed to. Afterwards, in any period $p' > p$ in which s is not crashed any more the information and data s stores for bucket B is outdated, but its information and data stored for a different bucket that has not been written while s was crashed is still up-to-date. In this case we call s **outdated w.r.t. bucket** B.

5.3.3 Details on the Decoding of a Bucket (Step 1)

At the beginning of each phase of the Writing Stage (see Step 1 in Section 5.3.2), for each data item d belonging to the current bucket B, all pieces of d are decoded and sent to the server maintaining d. In particular, the data pieces stored at crashed servers are recovered and stored by the representatives of the crashed servers.

In order to exclude outdated data from the computations, the outdated servers need to be determined before the actual decoding is performed. For this purpose, each server s chooses $\Theta(\log n)$ servers uniformly at random and asks these servers for their timestamp. As already mentioned in Step 4 of the storage strategy (Section 5.2), whenever data is (re-)written in a bucket, the intact servers agree on a common timestamp t for that bucket which each intact server s' stores in $t_{s'}(B)$. Hence, all servers that were intact during the last writing of bucket B hold the same timestamp for that bucket. Since the adversary crashes only at most $\gamma n^{1/\log\log n}$ servers, only the same number of servers can be outdated w.r.t. bucket B. With Chernoff bounds (Lemma 2.1) it follows that more than half of the servers chosen by s are not outdated w.r.t. bucket B. Hence, if the timestamp $t_s(B)$ that server s holds for bucket B is not equal to the timestamp it received most often from the chosen servers, s sets $t_s(B) = -\infty$ which marks s as outdated w.r.t. bucket B. In the following decoding process of bucket B, whenever a server s transmits data to another server, it additionally transmits its timestamp $t_s(B)$ it holds for bucket B. By this, the servers can easily detect outdated servers and exclude their data from the computations.

The actual decoding is done by a bottom-up approach that proceeds in $\log_k n + 1$ rounds. Algorithm 10 describes the actions each node u at level $\log_k n - r$ performs in round $r \in \{0, \ldots, \log_k n\}$ for the decoding of bucket B. Whenever a node u receives data during the decoding process, it first excludes all data received from a crashed or outdated node. If u receives at least $k - 1$ data blocks from intact servers, it recovers the data it is supposed to hold and also updates its timestamp $t_u(B)$. Finally, u forwards the possibly recovered data with a correct timestamp, or it forwards incorrect data which is identified by a timestamp of $-\infty$.

Just as in Basic IRIS, one can show via a witness tree argument that after the decoding process of a bucket B, each server successfully recovered the data

Algorithm 10 RoBuStBucketDecoding called in round r by node u at level $\log_k n - r$

```
1: if u crashed then
2:     t_u(B) = −∞, b = NULL
3: else
4:     b ← data that u stores for bucket B
5: if u not at level log_k n then
       ▷ Reduce set of received messages to set of messages with maximum timestamp
6:     Q ← set of messages u received from its children in LT(u)
7:     Q ← {(b, time) ∈ Q | time > −∞}                ▷ exclude outdated data
   ▷ Recover data, if possible
8: if |Q| ⩾ k − 1 then
       ▷ Recover data stored in u for bucket B using received data
9:     b ← recovered data
       ▷ Note: t = t' for all (b, t), (b', t') ∈ Q
10:    t_u(B) = timestamp in any message from Q
11: if u not at level 0 then
12:    Forward message (b, t_u(B)) to u's children in UT(u).
```

it stored for bucket B. Analogously to Basic IRIS (Definition 3.27), a **witness tree** of an ℓ-dimensional k-ary butterfly is defined to be a complete binary tree of depth ℓ that is rooted at a butterfly node at level 0 and consists of crashed and outdated servers only.

The following lemma can be proven analogously to Lemma 3.14 and Definition 3.27.

Lemma 5.3. *Let u be a butterfly node at level $\ell \in \{0, \ldots, \log_k n\}$. The subbutterfly $BF(u)$ can correctly be recovered if and only if $BF(u)$ does not contain a witness tree.*

Since the adversary may crash only at most $\gamma 2^{\log_k n - 1} = (\gamma/2) \cdot n^{1/\log\log n}$ servers, in total only less than $2^{\log_k n}$ servers can be crashed or outdated w.r.t. bucket B. Hence, there cannot exist a witness tree for bucket B, implying bucket B can correctly be recovered, which is possible via a bottom-up information transfer as described by Algorithm 10.

Note that the c hash functions for the pieces of a data items were chosen uniformly and independently at random when B was encoded, and after the adversary had chosen the set of crashed servers. Thus, each server holds $O(\log n)$ data pieces for bucket B, w.h.p. Furthermore, each server maintains at most one data item, w.h.p. This implies that the decoding of a bucket causes a congestion of at most $O(\log n)$ at each server in each round, w.h.p. All in all, we get:

Lemma 5.4. *After* $\log_k n + 1$ *rounds with a congestion of* $O(\log n)$ *at each server in each round, each server maintaining a data item* d *in the current bucket* B *completely knows* d*, w.h.p.*

5.3.4 Details on the Encoding of a Bucket (Step 2, Step 3b)

In the following we describe how a set of data items is re-encoded into a bucket, as required in Step 2 and Step 3b. Note that the reencoding of a bucket not only consists of simply encoding the data items belonging to that bucket but also of some additional steps described in the following.

First, in contrast to Basic IRIS, server s_0 chooses c hash functions $h_1, \ldots, h_c : \mathcal{U} \to \mathcal{S}$ uniformly at random that will be used to map data pieces of this bucket to servers. While in Basic IRIS the hash functions that map data pieces to servers are never changed, we need to choose new hash functions for a bucket B whenever B is (re-)encoded. The reason for this is that otherwise the adversary would be able to generate write requests that overload certain servers when distributing data pieces while (re-)writing a bucket. Note that the hash functions need to satisfy certain expansion properties just as in IRIS, but if c is chosen sufficiently large ($c \geqslant 12 \log m$) they do so, w.h.p. (more information is provided in Section 5.5). After that, s_0 distributes the c hash functions as well as its current timestamp to all other intact servers s_i. This distribution can be realized by simply broadcasting the hash functions in the k-ary butterfly from the node on level $\log_k n$ that is emulated by s_0 to all nodes on level 0. Each intact server then stores the c hash functions and sets $t_s(B)$ to the timestamp it received from s_0. By this, whenever a bucket is (re-)written, all intact servers hold the same timestamp for that bucket. Note that server s_0 could have also simply created a hash value for the current bucket using a collision-free hash function instead of using the current timestamp from its internal clock.

Each server s_i now creates for each data item that it maintains, or for which it has received write requests and that are not propagated to the next phase, the c pieces $d_1, \ldots, d_c$ of d using Reed-Solomon coding. Here, the c pieces of d are created in such a way that any $c/3$ pieces of them suffice to recover d. Afterwards, the data piece d_j, $j \in \{1, \ldots, c\}$, is sent to the server s_i that is responsible for $h_j(d)$. Unfortunately, a server s_i does not necessarily know the representative of the server s' if that server is crashed. Thus, instead of sending the data pieces directly, the servers forward the data pieces to the according servers by initiating a bottom-up routing in the underlying k-ary butterfly such that at the end each server s' responsible for any $h_j(d)$, $1 \leqslant j \leqslant c$, or its representative in case s' is crashed, has received the according data piece d_j. Since the hash functions $h_1, \ldots, h_c$ are chosen uniformly at random, with Valiant's trick [Val82] it follows that this process causes a congestion of at most

$O(c \log n)$.

After the pieces of data items have been distributed, the servers encode the data items in $(\mathcal{D}(B_z) \cup D_z) \backslash D_{z+1}$ with each other via the k-ary Butterfly Coding Strategy as described in Section 3.1. Note that the set of data blocks a server s_i may already have stored for bucket B_z before the (re-)encoding process is completely overwritten in this process. All in all we get the following lemma.

Lemma 5.5. *The encoding of a bucket takes $O(\log n)$ communication rounds with a congestion of at most $O(\log^2 n)$ at each server in each round.*

5.3.5 Details on Counting and Selecting (Step 3, Step 3a)

In the following we present the process of determining the number of data items in $\mathcal{D}(B_z) \cup D_z$ and the elements of the set D_{z+1} (if necessary) for a phase z in a distributed fashion. Note that in a previous step all data items from $\mathcal{D}(B_z)$ have successfully been decoded: i.e., the data items stored at both crashed and intact servers are correctly recovered, whereas the data items of the crashed servers are known to their representatives.

This process is divided into two parts: First, each server s_i determines for $j \in \{0, 1\}$ the number of data items d from $\mathcal{D}(B_z) \cup D_z$ with $\mathsf{bit}_z(d) = j$, which we define as $\mathsf{num}_j(i)$. Note that $|\mathcal{D}(B_z) \cup D_z| = \mathsf{num}_0(i) + \mathsf{num}_1(i)$. Second, if $\mathsf{num}_0(i) + \mathsf{num}_1(i) > 2n$, we determine the set of data items D_{z+1} that will be handled in the next phase of the Writing Stage. Recall that the set D_{z+1} is defined to be a set of n data items from the set $\mathcal{D}(B_z) \cup D_z$ whose z-th bit of their addresses equal, i.e., $\mathsf{bit}_z(d) = \mathsf{bit}_z(d') = b \in \{0, 1\}$ for all $d, d' \in D_{z+1}$.

In the following let $B_{z,i} \subseteq \mathcal{D}(B_z)$ denote the set of data items from bucket B_z that server s_i maintains. Furthermore, for the set D_z of data items to be inserted into B_z, we define $D_{z,i} \subseteq D_z$ as the set of data items with a write request at server s_i. We will also use the notion of server s_i maintaining data item $d \in D_z$ if there is a write request for d at server s_i. Note that $D_{z+1} \subseteq \bigcup_{i=0}^{n-1} (D_{z,i} \cup B_{z,i})$.

Computation of $\mathsf{num}_0(i) + \mathsf{num}_1(i)$: First of all, each server s_i, $i \in \{0, \ldots, n-1\}$ initializes a tuple $(\mathsf{num}_0, \mathsf{num}_1)$ where num_j, $j \in \{0, 1\}$, is the number of data items $d \in B_{z,i} \cup D_{z,i}$ with $\mathsf{bit}_{z+1}(d) = j$. These tuples are now forwarded bottom-up in the underlying k-ary butterfly, where each intermediate node sums up all tuples it received and forwards the result to the next higher level. More precisely, each server s_i first sends its tuple $(\mathsf{num}_0, \mathsf{num}_1)$ to each of the k neighbors of the butterfly node $(\log_k n, i)$ in the underlying k-ary butterfly. Any intermediate node v on level ℓ, $0 < \ell < \log_k n$, sets num_j, $j \in \{0, 1\}$, as the sum of all k received num_j-values and sends the tuple $(\mathsf{num}_0, \mathsf{num}_1)$ to its k neighbors on level $\ell - 1$ in the underlying k-ary butterfly. Finally, each server

s_i on level 0 sums up all tuples received from neighbors on level 1 and stores the result in $(\mathsf{num}_0(i), \mathsf{num}_1(i))$. All in all it follows:

Lemma 5.6. *After $\log_k n$ rounds, each server s_i knows the number of data items $d \in \mathcal{D}(B_z) \cup D_z$ with $\mathsf{bit}_{z+1}(d) = j$ for $j \in \{0,1\}$. Additionally, in every round, each server sends and receives at most $2k$ messages.*

Each server s_i now computes $\mathsf{size}(B_z) = \mathsf{num}_0(i) + \mathsf{num}_1(i)$ and checks whether $\mathsf{size}(B_z) > 2n$. In case $\mathsf{size}(B_z) \leqslant 2n$, the data items from $\mathcal{D}(B_z)$ and D_z are re-encoded in bucket B_z (see Section 5.3.4) and the Writing Stage is finished. Otherwise, the servers need to agree on the bucket B_{z+1} and a set D_{z+1} of n data items from $\mathcal{D}(B_z) \cup D_z$ with $\mathsf{bit}_z(d) = \mathsf{bit}_z(d') = b \in \{0,1\}$ for all $d, d' \in D_{z+1}$ that will be handled in the next phase of the Writing Stage. That is, if $\mathsf{num}_0(i) > n$, we set $B_{z+1} = 0\text{-}\mathsf{child}(B_z)$ and $b = 0$. If $\mathsf{num}_1(i) > n$ we set $B_{z+1} = 1\text{-}\mathsf{child}(B_z)$ and $b = 1$.

Determination of D_{z+1}: In the following define D as the set of data items d from $\mathcal{D}(B_z) \cup D_z$ with $\mathsf{bit}_{z+1}(d) = b$: i.e., D is the set of potential data items for the set D_{z+1}. The determination of D_{z+1} is done by a top-down approach in the tree $LT((0,0))$ of the k-ary butterfly. In the following, we assume that each node v in $LT((0,0))$ stored the tuples $(t_{1,0}, t_{1,1}), (t_{2,0}, t_{2,1}), \ldots, (t_{k,0}, t_{k,1})$ it received from its children $v_1, \ldots, v_k$ in $LT((0,0))$ during the previous process (the bottom-up counting of $|D_{z+1}|$): i.e., $t_{i,b}$ is the number of data items $d \in D$ that are maintained by a node from $LT(v_i)$.

During the process for determining D_{z+1} the nodes send two types of messages: full and $\mathsf{partly}(x)$, $x \in \mathbb{N}$. If a node v receives a full, this is supposed to indicate that all data items d which a node from $LT(v)$ maintains with $\mathsf{bit}_z(d) = b$ will belong to the set D_{z+1}. A message $\mathsf{partly}(x)$ at a node v indicates that x data items still need to be chosen for the set D_{z+1}.

At the beginning, the butterfly node $(0,0)$ issues the message $\mathsf{partly}(n)$ on itself. Depending on the message a node v receives, it performs the actions described in the following and visualized in Figure 5.2.

$\mathsf{partly}(x)$: If v is not on level $\log_k n$, let $v_1, \ldots, v_k$ denote the children of v in $LT((0,0))$. Determine the greatest index $p \in \{1, \ldots, k\}$ such that $y \leqslant x$ with $y := \sum_{i=1}^{p} t_{i,b}$. Send full to $v_1, \ldots, v_p$. If $x - y > 0$, send $\mathsf{partly}(x - y)$ to v_{p+1}. That is, all data items $d \in D$ that are maintained by a node in any of the lower trees $LT(v_1), \ldots, LT(v_b)$ will be added to D_{z+1}. The remaining data items that will be chosen for D_{z+1} are maintained by nodes from $LT(v_{p+1})$.

If v is on level $\log_k n$, the server s_i emulating v randomly chooses x data items $d \in B_{z,i} \cup D_{z,i}$ with $\mathsf{bit}_{z+1}(d) = b$ to belong to D_{z+1}.

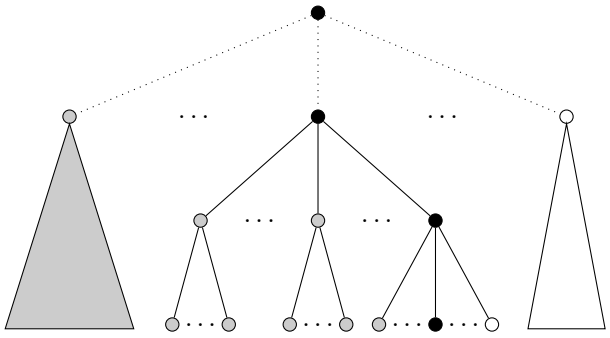

Figure 5.2: Visualization of the computation of the set D_{z+1}. Each gray node received a full message, i.e., it is emulated by a server s_i and adds all data items $d \in D_{z,i} \cup B_{z,i}$ with $\mathsf{bit}_{z+1}(d) = b$ it stores to D_{z+1}. Each black node received a full message, i.e., it is emulated by a server s_i and adds only a part of its data items $d \in D_{z,i} \cup B_{z,i}$ with $\mathsf{bit}_{z+1}(d) = b$ it stores to D_{z+1}.

full: If v is not on level $\log_k n$, v sends a full-message to each of its children in $LT((0,0))$. If v is on level $\log_k n$, the server s_i emulating v chooses all data items $d \in B_{z,i} \cup D_{z,i}$ with $\mathsf{bit}_{z+1}(d) = b$ to belong to D_{z+1}.

Each server s_i that has chosen a data item $d \in B_{z,i} \cup D_{z,i}$ to belong to D_{z+1} during the previously described process, handles this data item in the next phase of the Writing Stage just as if s_i has received a write request d.

The following lemma is easy to check.

Lemma 5.7. *If the servers have already determined the number of data items $d \in \mathcal{D}(B_z) \cup D_z$ with $\mathsf{bit}_{z+1}(d) = j$ for $j \in \{0,1\}$, then after $\log_k n$ additional rounds of the previously described selection process it holds:*

1. *Each server s_i knows which of the data items in $B_{z,i} \cup D_{z,i}$ are supposed to be (re-)encoded in bucket B_z and which of them are propagated to the next phase of the Writing Stage.*
2. *In every round of the selection process, each server s sends and receives at most $2k$ messages.*
3. *The number of data items that are decided to belong to bucket B_z (and thus will be encoded in this bucket) is at most $2n$.*

It remains to distribute for each data item from D_{z+1} a write request among the n servers such that each server s_i is responsible for a constant number of these write requests, w.h.p. For this purpose we make use of a technique presented by Dietzfelbinger and Meyer auf der Heide [DM93]. This technique

makes use of the γ-collision rule, $\gamma \geqslant 2$ constant, and is based on the following idea: First, using three randomly chosen hash functions h_1, h_2, h_3, for each data item three potential target servers are determined. Each round consists of three phases. In phase $r \in \{1, 2, 3\}$ each server to which at most γ data items have been mapped via h_r accept these data items. All remaining servers reject the data items mapped to them for this round. These three phases are executed sequentially while only considering the data items that have not been accepted yet as long as each data item has been accepted by a server. Dietzfelbinger and Meyer auf der Heide showed that after $O(\log \log n)$ of these rounds all data items have been assigned to a server and each server received $O(1)$ data items, w.h.p. Since in our setting the servers cannot necessarily communicate directly with each other but need to route messages through the k-ary butterfly the communication between any two servers takes at most $O(\log_k n) = O(\log n / \log \log n)$ rounds, implying an overall runtime of the above procedure of $O(\log n)$.

All in all, regarding the efficiency of the Write Protocol we get the following lemma.

Lemma 5.8. *The Write Protocol of RoBuSt takes at most $O(\log^2 n)$ many communication rounds with at most $O(\log^2 n)$ congestion at every server in each round, w.h.p.*

5.4 Lookup Protocol

In order to keep the specification of our system simple, we provide the description of the Lookup Protocol as a separate protocol that is executed after the execution of the Write Protocol. The Lookup Protocol is divided into two stages: the Preprocessing Stage (Section 5.4.1) and the Zone Examination Stage (Section 5.4.2). The former is similar to the Preprocessing Stage of the Write Protocol (Section 5.3.1). The latter is performed for each zone individually and is split into two further stages: the Probing Stage and the Decoding Stage. The basic idea of the Probing Stage is to answer a request by directly collecting a sufficient number of data pieces. If this is not possible, either because too many of the servers holding a piece are crashed or due to excessive congestion, in the Decoding Stage the data item is tried to be recovered by utilizing the distributed coding described in Section 5.2.

Note that both a Probing Stage as well as a Decoding Stage can be found in Basic IRIS (Section 3.3.2 and Section 3.3.3), too. While they match in their general structure, there are important differences that are caused by the differences in the underlying structure and the implications of the write functionality. For example, in RoBuSt it may happen that servers store obsolete data items (e.g., data items that are not up-to-date).

5.4.1 Preprocessing Stage

The Preprocessing Stage is exactly the same as the Preprocessing Stage of the Write Protocol, described in Section 5.3.1. If at least one write request has been handled in the current period, we can thus skip this part and re-use the already established k-ary butterfly and the unique representatives.

5.4.2 Zone Examination Stage

In the following let $\mathcal{D}$ be the set of data items for which a lookup request arrived at an intact server. The idea of this stage is to successively perform a lookup for each $d \in \mathcal{D}$ in each zone until an up-to-date copy of d has been found and returned to the appropriate server. The Zone Examination Stage consists of at most $\Lambda + 1$ phases, one phase for each zone.

In each phase $z \in \{0, \ldots, \Lambda\}$, beginning with $z = 0$, each server with an unserved lookup request for some data item d initiates a lookup request for d in bucket $\mathsf{bucket}(z, d)$, i.e., in the bucket the data item d belongs to in zone z. Any server that receives a copy of the data item it has requested during the lookup in zone z, as described in the following, returns that copy and is finished. All remaining lookup requests are handled in the next phase, phase $z := z + 1$. This procedure is repeated until each lookup request is served.

Handling a set of lookup requests in one phase z is done by performing the Probing Stage and the Decoding Stage which are similar to the Probing and Decoding Stage of Basic IRIS. Therefore, in the following we only recap the actions to be performed in those stages and highlight the differences between these stages in Basic IRIS and RoBuSt.

5.4.2.1 Probing Stage

In the following let s be a server that holds an unserved lookup request for a data item d at the beginning of phase z. The idea of the Probing Stage is to either achieve $c/3$ up-to-date pieces such that d can be recovered, or to assign the request for d to a level $\ell \in \{1, \ldots, \log_k n\}$ (as defined later) in order to further handle the request in the next stage, the Decoding Stage.

On a high-level view, in phase z, server s with a lookup request for data item d performs the following steps.

Step 1: Acquire the current hash functions and the timestamp $t_s(B)$ for bucket $B := \mathsf{bucket}(z, d)$.

Step 2: Choose c intact servers $s(d_1), \ldots, s(d_c)$ uniformly and independently at random.

Step 3: For each $i \in \{1, \ldots, c\}$ forward a probe$(d, i, t_s(B))$ message along the probing path of d_i beginning at the node at level $\log_k n$ emulated by $s_i(d)$ and ending at the node at level 0 that is emulated by the server responsible for $h_i(d)$.

Note that the first step, acquiring the hash functions, is not part of the Probing Stage in Basic IRIS. In RoBuSt this step is required since s may have been crashed in the last period in which a write occurred in bucket bucket(z, d) and therefore new hash functions have been chosen for that bucket. In this case s would hold an outdated timestamp $t_s(B)$ for bucket B. Acquiring the current hash functions and the current timestamp for bucket B works as follows: First, s chooses $\Theta(\log n)$ servers uniformly at random and from these servers s selects the intact ones. With Chernoff bounds (Lemma 2.1) and as already explained in Section 5.3.3 it also follows that more than half of the chosen servers are neither crashed nor outdated w.r.t. bucket B. Server s then asks one of these intact servers for the timestamp t they hold for bucket B. In case s already holds an up-to-date timestamp for bucket B, s is finished with the first step. Otherwise, s updates its timestamp $t_s(B)$ to the timestamp t and asks one of the previously chosen intact servers for the current hash functions $h_1, \ldots, h_c$ for bucket B. Note that throughout this process each server only receives $O(\log n)$ requests.

Once s knows the correct hash functions, its goal is to retrieve at least $c/3$ pieces of d which is done in the same manner as in Basic IRIS: i.e., s forwards c probes for the c data pieces of d bottom-up through the k-ary butterfly (Steps 2 and 3). Algorithm 11 recaps the actions of a node u at level $\ell \in \{0, \ldots, \log_k n\}$ during this forwarding process. In order to decrease the congestion, just as in IRIS, in this process the nodes use the technique of splitting and combining while forwarding messages.

In this probing process the differences to the Probing of Basic IRIS are:

- In the congestion bound we use a different constant α. Here we require $\alpha \geqslant 12$ (Algorithm 11, line 1).
- In RoBuSt we do not perform a decoding depth check. The consequences of this will be argued in the Decoding Stage (Section 5.4.2.2).
- Before a node u processes a probe (d, i) it needs to check whether it is outdated w.r.t. the considered bucket $B = \text{bucket}(z, d)$. In case u is outdated w.r.t. bucket B, u returns a fail(0) message to the origins of probe (d, i) (Algorithm 11, lines 6–8).

Just as in the Probing Stage of Basic IRIS, as soon as a butterfly node on level $\log_k n$, which is emulated by a previously chosen server $s(d_i)$, $i \in \{1, \ldots, c\}$, receives an answer for the request for d_i (i.e., a fail(ℓ)/dataNotFound(d) message,

Algorithm 11 RoBuStProbing$(u = (x, \ell), z)$ performed by node u on level ℓ in zone z

▷ *Determine whether u not at level 0 and congested*
1: **if** $\ell > 0$ **and** u received $> \alpha c$ probe$(\cdot)$ messages for different probes **then**
2: Send fail(ℓ) message to the origins of the probes received.
3: **else if** $\ell \neq 0$ **then** ▷ *u not congested*
4: Forward all probes received to the according butterfly nodes ↳ on level $\ell - 1$ of the according probing paths.
5: **else** ▷ *u is at level 0, i.e., probes reached their destination*
6: **for all** probes for a (d, i) pair node u received **do**
▷ *If u is crashed, we assume $t_u(B)$ to be $-\infty$*
7: **if** $t_u(B) < t_{max}$ **then** ▷ *u's version of bucket(z, d) is not up-to-date*
8: Send a fail(0) message to the origins of the probe for (d, i).
9: **else** ▷ *u's version of bucket(z, d) is up-to-date*
10: **if** u holds piece d_i of d **then**
11: Send d_i to the origins of the probe for (d, i).
12: **else**
13: Send dataNotFound(d) message to origins of probe for (d, i).

or the piece d_i) it forwards this answer to the server that initiated the request for d_i. These answers ensure that after $O(\log_k n)$ rounds the server s with a lookup request for a data item d receives for all initially sent probe$(\cdot)$ messages a piece of d, or a dataNotFound(d) message, or the level at which the probing failed.

Analogously to Basic IRIS, we define a request to belong to a specific level as follows.

Definition 5.9 (Belong to)**.** *A request for a data item d is said to **belong to level** $\ell \in \{1, \ldots, \log_k n\}$ if ℓ is the smallest level that contains at least $(5/6)c$ active (d, i) probes, i.e., (d, i) probes that were not deactivated at level $\ell' \geqslant \ell$.*

The only difference in this definition for Basic IRIS and RoBuSt is that in Basic IRIS only at least $c/2$ active probes at a level ℓ for a data item d were needed in order to belong to level ℓ. Note that if a data item belongs to a level ℓ, then at least $(5/6)c$ of its probes successfully passed level ℓ and got deactivated later in the probing (i.e., in a level $\ell' < \ell$).

Depending on which kinds of answers a server s with a lookup request for d has received, it reacts just as in Basic IRIS with the only differences being that in RoBuSt $c/3$ pieces are required to recover a data item and the definition of "belong to". More precisely:

- If s receives at least $c/3$ up-to-date pieces of d, s recovers d using Reed-Solomon coding and answers the request.

- Else if s receives a dataNotFound(d) message, s returns NULL.
- Else if s receives more than $(2/3)c$ fail$(\cdot)$ messages, s declares the request for d to **belong to level** ℓ with ℓ as defined in Definition 5.9.

At the end of the Probing Stage, each request for a data item d has either been served correctly (this is the case if the requests for $(5/6)c > c/3$ pieces of d pass level 0) or the request belongs to a level $1 \leqslant \ell \leqslant \log_k n$.

Analogously to IRIS, for the runtime and the congestion the following lemma holds.

Lemma 5.10. *The Probing Stage of the Lookup Protocol of RoBuSt takes at most $O(\log_k n)$ communication rounds with at most $O(\log^2 n)$ congestion at every server in each round.*

5.4.2.2 Decoding Stage

The Decoding Stage in RoBuSt works analogously to the Decoding Stage in Basic IRIS with the following main differences: First, we need to consider that servers may be outdated. Second, it may be necessary to abort the decoding of a subbutterfly due to too many crashed or outdated servers. In Basic IRIS this was not necessary due to the decoding depth check performed in the Probing Stage.

For the sake of completeness we recap the adapted Decoding Stage here. The Decoding Stage is divided into $\log_k n$ subphases, where in subphase $\ell \in \{1, \ldots, \log_k n\}$ all requests that have been defined to belong to level ℓ are handled. On a high-level view, in subphase ℓ of the Decoding Stage, for each data item d with a lookup request at a server s that belongs to level ℓ the following steps are performed.

Step 1: Server s randomly chooses $(5/6)c$ requests for data pieces of d that were not deactivated at level ℓ or earlier in the Probing Stage.

Step 2: For each of the chosen requests for a data piece d_i it is determined whether $BF(u_{s,i}^{(\ell)}(d))$ can be decoded without any node receiving more than βck messages for different pieces d_i with $\beta > 1/10$. Such a node will be called **congested**. (Note that in contrast to this in Definition 3.19 of Basic IRIS we require $\beta > 3/2$.) Here, $u_{s,i}^{(\ell)}(d)$ denotes the butterfly node on level ℓ of the probing path of d_i with origin $s_i(d)$. Whether a node $BF(u_{s,i}^{(\ell)}(d))$ is congested is determined via a broadcast of a message for (d, i) in $BF(u_{s,i}^{(\ell)}(d))$ just as it is in done in the Decoding Stage of Basic IRIS.

Step 3: If it is detected that for at least $c/3$ pairs (d, i) no node of the subbutterflies $BF(u_{s,i}^{(\ell)}(d))$ is congested, s initiates for $c/3$ of these pairs the decoding of the according subbutterflies. Although no congestion can occur during the decoding, in contrast to Basic IRIS the decoding of the subbutterflies can fail due to too many outdated servers and due to too many crashed servers (as described below).

Step 4: If one of the following cases occurs in the previous step, the server s changes the request for d to belong to level $\ell + 1$ such that it will be processed again in the next subphase of the Decoding Stage.

(i) It is not possible to decode $c/3$ subbutterflies $BF(u_{s,i}^{(\ell)}(d))$ without causing excessive congestion at a node in $BF(u_{s,i}^{(\ell)}(d))$.

(ii) During the decoding of any $BF(u_{s,i}^{(\ell)}(d))$, it is detected that too many nodes from $BF(u_{s,i}^{(\ell)}(d))$ are outdated or congested (as described below).

Step 5: If data item d exists in the system and if $c/3$ pieces of d have been recovered successfully, these pieces are returned to the server s such that s can recover d via Reed-Solomon coding and answer the request. Otherwise, s answers that d does not exist in the system.

In Step 2 the handling of outdated and crashed servers during the decoding works as follows: For each subbutterfly $BF(u_{s,i}^{(\ell)}(d))$ that is supposed to be decoded, the servers that emulate a node at level ℓ of $BF(u_{s,i}^{(\ell)}(d))$ first acquire the current timestamp $t_z(B)$ for bucket $B := \mathsf{bucket}(z, d)$, just as in Step 1 of the Probing Stage. At the end of this process, each of these servers knows whether it is outdated with respect to the bucket B. Whenever during the bottom-up decoding of the $c/3$ subbutterflies $BF(u_{s,i}^{(\ell)}(d))$ more than one node in any k-block is emulated by an outdated or crashed server, the decoding of the according subbutterfly $BF(u_{s,i}^{(\ell)}(d))$ is aborted and the server $s_i(d)$ is informed about that. This additional check replaces the decoding depth check that was performed in the Probing Stage in Basic IRIS. That is, during the decoding of a subbutterfly, outdated servers are treated as crashed servers. Since the error correcting code we used in the k-ary Butterfly Coding Strategy guarantees the correction of a single error, the previously described approach guarantees that whenever a data piece d_i is recovered, the most recent version of that data piece is recovered.

Analogously to Basic IRIS, the following Lemma holds.

Lemma 5.11. *The Decoding Stage of the Lookup Protocol of RoBuSt takes at most $O(\log n)$ communication rounds per subphase with at most $O(\log^3 n)$ congestion at each server in each round.*

Since the Lookup Protocol may require a traversal through all zones of the bucket tree with Lemma 5.10 and Lemma 5.11 the following lemma is implied.

Lemma 5.12. *The Lookup Protocol of RoBuSt takes at most $O(\log^2 n)$ communication rounds with at most $O(\log^3 n)$ congestion at every server in each round.*

5.5 Correctness Analysis of the Lookup Protocol

Since the Lookup Protocol of RoBuSt resembles the Lookup Protocol of IRIS, their analyses are also similar. The main differences we need to cope with are the existence of outdated data items in the system which may cause a probe to fail in the Probing Stage and the missing precomputed decoding depth which may cause a request to be aborted in the Decoding Stage due to too many outdated or crashed servers.

For the analysis we need the term of a crashed subbutterfly which is defined as follows.

Definition 5.13 (Crashed Subbutterfly)**.** *Let v be a butterfly node at level $\ell \in \{0, \dots, \log_k n\}$. The subbutterfly $BF(v)$ is called **crashed** at level ℓ if at least $\lceil 2^{\ell-1} \rceil$ servers from $BF(v)$ are crashed.*

Note that we need the ceiling function in Definition 5.13 only for the special case of $\ell = 0$.

Regarding outdated servers, one can show that whenever a data item is (re-)written in a bucket, not too many pieces of that data item are mapped to subbutterflies that contain too many crashed servers. Hence, for each data item not too many of its data pieces were supposed to be stored at crashed servers. The following lemma formalizes this.

Lemma 5.14. *Assume the adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/24$. Then, for any data item d that is (re-)written in a bucket, and any level $\ell \in \{0, \dots, \log_k n\}$, at most $c/6$ pieces of d that are mapped to subbutterflies $BF(v)$ (for some node v at level ℓ) that contain at least $\lceil 2^{\ell-1} \rceil$ crashed servers, w.h.p.*

For a visualization of Lemma 5.14 see Figure 5.3.

Proof. Let d be a data item, and let $\ell \in \{0, \dots, \log_k n\}$ be a fixed level in the underlying k-ary butterfly. First of all, we show that the fraction of crashed subbutterflies at level ℓ is at most 2γ. Recall that each subbutterfly at level ℓ contains exactly k^ℓ servers. Obviously, for $\ell = 0$, the fraction of crashed

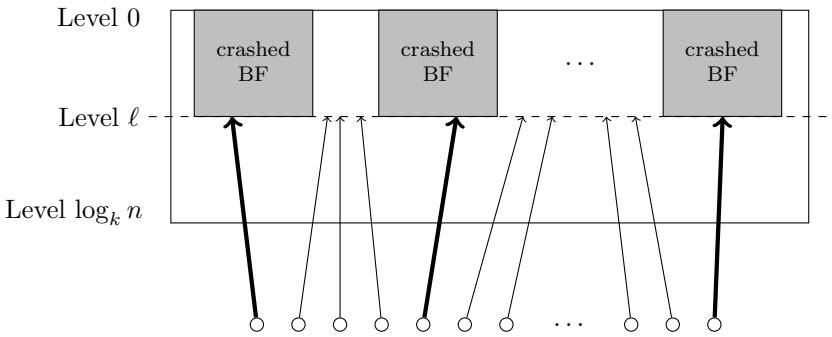

Figure 5.3: Visualization of Lemma 5.14. The lower circles denote the data pieces of a data item that is (re-)written in the bucket. The arrows point to the subbutterflies of level ℓ that the according data piece is mapped to. The gray rectangles denote the subbutterflies at level ℓ that contain at least $\lceil 2^{\ell-1} \rceil$ crashed servers.

butterflies at level ℓ is at most 2γ. Thus, in the following, we assume $1 \leqslant \ell \leqslant \log_k n$. Since in total there at most $\gamma \cdot n^{1/\log\log n}$ crashed servers and a butterfly at level $\ell \geqslant 1$ is crashed if it contains at least $2^{\ell-1}$ crashed servers, we get that only at most $(\gamma \cdot n^{1/\log\log n})/2^{\ell-1}$ butterflies at level ℓ can be crashed. Since the number of butterflies at level ℓ is n/k^ℓ, we get that the fraction of crashed butterflies at level ℓ is upper bounded by

$$\gamma \cdot \frac{n^{1/\log\log n}}{2^{\ell-1} \cdot n/k^\ell} = \gamma \cdot \frac{2^{\log_k n}}{2^{\ell-1} \cdot k^{\log_k n - \ell}} \leqslant 2\gamma$$

Next, we introduce a binary random variable X_i which is 1 if and only if data piece d_i is mapped to a subbutterfly that is crashed at level ℓ. That is, $Pr[X_i = 1] \leqslant 2\gamma$. Hence, with $X := \sum_{i=1}^{c} X_i$ we get $\mu := E[X] \leqslant 2\gamma c$. With Chernoff bounds (Lemma 2.1) it follows that for all $\delta \geqslant 0$ it holds:

$$Pr[X \geqslant (1+\delta)\mu] \leqslant e^{-\delta^2\mu/2(1+\delta/3)}$$

In particular, for $\delta = 5$ we get

$$Pr[X \geqslant c/6] \leqslant e^{-\delta^2 \cdot 2\gamma c/2(1+\delta/3)} = e^{-75\gamma c/8}$$

Hence, at most $c/6$ pieces of d are mapped to subbutterflies that are crashed at level ℓ, w.h.p.

□

While in IRIS we required the hash functions $h_1, \ldots, h_c$ to form a $(c/4, 1/9)$-

expander, in RoBuSt we require them to form a $(c/6, 2\gamma)$-expander. By Lemma 3.24 it holds that if $h_1, \ldots, h_c$ are chosen uniformly and independently at random with $c \geqslant 12 \log m$, then the hash functions form a $(c/6, 2\gamma)$-expander.

5.5.1 Analysis of the Probing Stage

Analogously to Basic IRIS for the Probing Stage of the Lookup Protocol of RoBuSt it can be shown that the number of requests belonging to a level exponentially decreases.

Lemma 5.15. *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/24$. Then, at the end of the Probing Stage of the Lookup Protocol of RoBuSt, the number of requests belonging to level $\ell \in \{1, \ldots, \log_k n\}$ is at most $2\gamma n/k^{\ell-1}$.*

The proof of Lemma 5.15 is similar to the proof of Lemma 3.26 in IRIS, but here we need to take the following differences into account:

- Recall that in IRIS we precompute a decoding depth for each node such that in the Probing Stage of IRIS a request can be aborted at any level if the decoding depth of a node is exceeded. In contrast to this, in the Probing Stage of RoBuSt the only level at which requests are aborted due to crashed nodes is level 0. This is detected by comparing the timestamps with each other (Algorithm 11, lines 6–8).
- In contrast to IRIS, we also have to take into account that nodes may store outdated information because they were crashed in a period in which new data was written.
- A minor further difference are the constants in the definition of a node belonging to a level $\ell \in \{1, \ldots, \log_k n\}$ and in the congestion bound. While in IRIS we only require $c/2$ active probes to belong to level ℓ, in RoBuSt we require $(5/6)c$ probes to belong to level ℓ.

Analogously to the proof of Lemma 3.26, in the proof of Lemma 5.15 we use the terms of congested subbutterflies and congested/crashed request which, as a small reminder, are defined as follows:

Definition 5.16 (Congested/Crashed Subbutterfly/Request)**.** *Let v be a node at level ℓ in the butterfly. The subbutterfly $BF(v)$ is called*

- ***congested** at level ℓ of the Probing Stage if the servers in $BF(v)$ receive more than $k^\ell \alpha c/2$ messages for different d_i pieces in total when the requests are processed at level ℓ.*

A request for a data item d is called

- ***congested*** *at level ℓ of the Probing Stage if there are at least $r = c/6$ congested subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $l_i \geqslant \ell - 1$, $r = c/6$, and $i_1, \ldots, i_r$ being pairwise different.*
- ***crashed*** *at level ℓ if there are at least $r = c/6$ crashed subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell$ and $i_1, \ldots, i_r$ being pairwise different.*

Proof of Lemma 5.15*:* In the following we say a request for a data piece d_i of a data item d is **aborted** at level ℓ if the probing for d_i did not successfully pass level ℓ: i.e., a fail(ℓ) message was sent to the origins of the (d, i) pair. Depending on the level ℓ, a request for a data item d can be aborted at a node u for the following reasons:

- At level $\ell > 0$: The request for d can be aborted only if u is congested.
- At level $\ell = 0$: The request for d can be aborted if u is outdated or crashed.

In contrast to Basic IRIS, for upper bounding the number of requests belonging level $\ell \in \{1, \ldots, \log_k n\}$, we distinguish between level 1 and all other levels $\ell > 1$.

Upper bound on number of requests belonging to level 1: By the definition of "belong to", it holds that for each request for a data item d that belongs to level 1, at least $(5/6)c$ probes for d were active at level 1. Furthermore, only less than $c/3$ pieces have passed level 0, since if at least $c/3$ requests for pieces of d would have passed level 0, the request for d could have been answered. Hence, more than $(5/6)c - c/3 = c/2$ requests for pieces of d must have been aborted at level 0, which by the previous observation can only be due to outdated and/or crashed nodes at level 0. By Lemma 5.14, we know that at most $c/6$ pieces of d have been aborted at level 0 due to outdated nodes. Thus, for any data item d belonging to level 1, more than $(2/6)c > c/6$ requests for pieces of d must have been aborted due to crashed nodes. Analogously to Corollary 3.31, one can show that whenever a request belongs to level 1, the request is crashed at level 0. Hence, by upper bounding the number of crashed requests at level 0 we also get an upper bound on the number of requests belonging to level 1.

Let S be a maximum set of data items with crashed requests belonging to level 1. We will show: $|S| < 2\gamma n$. First, construct a set F in the following way: for each $d \in S$, we choose $c/6$ indices i with the property that the request for d_i is aborted at level 0 due to crashed nodes and add these (d, i) to F. Note that F is a $c/6$-bundle F of S. Since the adversary can crash only at most $2\gamma n^{1/\log\log n} < 2\gamma n$ servers, the number of servers covered by all $BF(s_i^{(0)}(d))$ with $(d, i) \in F$ is at most $2\gamma n$. Since $\Gamma_{F,0}(S)$ is exactly the set of these servers, it

holds: $|\Gamma_{F,0}(S)| < 2\gamma n$. Just as in the proof of Lemma 3.26 with Corollary 3.25 we can deduce $|S| < 2\gamma n$ for $\gamma = 1/24$.

Upper bound on number of requests belonging to level $\ell > 1$: Next, for any level $\ell > 1$, we bound the number of requests belonging to level ℓ. Recall that the only reason for a piece of a data item to be aborted on a level $\ell > 0$ is due to excessive congestion at a node at level ℓ. Similarly to Corollary 3.31, one can show that whenever a request belongs to a level $\ell > 1$, this request is congested at level $\ell - 1$. Thus, if many data items belong to level ℓ, then many subbutterflies must be congested at level $\ell - 1$. However, as we will prove, only a constant fraction of the subbutterflies at level $\ell - 1$ can be congested, which implies that only a constant fraction of all data items can belong to level ℓ.

Fix $1 < \ell \leqslant \log_k n$. In order to upper bound the number of requests that are congested at level $\ell-1$, let S be a maximum set of data items that with a request congested at level $\ell-1$. We will show: $|S| < 2\gamma n/k^{\ell-1}$. In the same way as in the proof of Lemma 3.26 we construct a $c/6$-bundle F of S (adding, for each $d \in S$, $c/6$ indices i to F with the property that $BF(u_{s,i}^{(\ell_i)}(d))$ is congested). We first show that for α sufficiently large, specifically for $\alpha > 1/\gamma$, less than a fraction of 2γ of all butterflies at level $\ell - 1$ can be congested. Recall that a subbutterfly on level $\ell - 1$ is congested if it receives more than $\alpha c k^{\ell-1}/2$ probes for different (d, i) pairs. Let ε be the maximum fraction of servers, the adversary may crash. Since there are at most $(1-\varepsilon)n$ lookup requests in total, at most $c(1-\varepsilon)n$ probes arrive at level $\ell - 1$. Thus, at most $c(1 - \varepsilon)n/(\alpha c k^{\ell-1}/2) = 2(1 - \varepsilon)n/(\alpha k^{\ell-1})$ subbutterflies can be congested at level $\ell - 1$. Since there are exactly $n/k^{\ell-1}$ disjoint subbutterflies at level $\ell - 1$, the fraction of congested subbutterflies at level $\ell - 1$ is upper bounded by $2(1 - \varepsilon)nk^{\ell-1}/(\alpha k^{\ell-1}n) = 2(1 - \varepsilon)/\alpha$. Hence, for $\alpha > (1 - \varepsilon)/\gamma$, less than a fraction of 2γ of the subbutterflies on level $\ell - 1$ can be congested. That is, all of the congested subbutterflies $BF(u_{s,i}^{(\ell)}(d))$ with $(d, i) \in F$ together contain less than a 2γ-fraction of the subbutterflies on level $\ell - 1$. This implies $|\Gamma_{F,\ell-1}(S)| < 2\gamma n$. Just as in the proof of Lemma 3.26 with Corollary 3.25 we can deduce $|S| < 2\gamma n/k^{\ell-1}$ for $\gamma = 1/24$. □

5.5.2 Analysis of the Decoding Stage

Analogously to Basic IRIS, for the Decoding Stage of the Lookup Protocol of RoBuSt, the number of requests belonging to a level exponentially decreases such that in the last phase of the Decoding Stage at most $O(k)$ requests have to be handled while all other requests have already been answered in the previous phases.

Lemma 5.17. *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/24$. Then, at the beginning of each subphase $\ell \in \{1, \ldots, \log_k n\}$ of the*

Decoding Stage of the Lookup Protocol of RoBuSt, the number of requests belonging to level ℓ is at most $\varphi n/k^\ell$ with $\varphi \leqslant 6\gamma k$.

For the proof of Lemma 5.17 we also use the terms of congested subbutterflies and congested requests which are defined just as in the Probing Stage (Definition 5.16, Definition 5.16). Recall that in Basic IRIS we used a decoding depth check in the Probing Stage, such that in the Decoding Stage no decoding of a subbutterfly could fail due to too many crashed servers. Since we do not use this decoding depth check in RoBuSt and since servers may be outdated, in the Decoding Stage of RoBuSt we additionally need to handle crashed subbutterflies and crashed requests (see Definitions 5.13 and 5.16).

Note that during the congestion check in the first part of the Decoding Stage of each phase, the handling of requests may also be aborted due to congested nodes. Hence, in the analysis we also need to consider congested subbutterflies, which in the Decoding Stage are defined as follows (analogously to Definition 3.19 of Basic IRIS).

Definition 5.18 (Congested Subbutterfly)**.** *Let v be a butterfly node at level $\ell \in \{1, \ldots, \log_k n\}$. We call the subbutterfly $BF(v)$* ***congested*** *if at least one node from $BF(v)$ receives messages for more than βck different (d, i) pairs with $\beta > 5/2$.*

Furthermore, due to possibly outdated servers, we require the following lemma.

Lemma 5.19. *For any request for a data item d that is neither crashed nor congested, at most $c/6$ pieces of d can fail due to outdated servers.*

Proof. Let d be a data item with a request that is neither crashed nor congested: i.e., there are less than $c/6$ crashed subbutterflies, which means that for less than $c/6$ data pieces the corresponding ℓ-dimensional subbutterflies contain more than $2^{\ell-1}$ crashed servers. By Lemma 5.14, at most $c/6$ data pieces of the remaining pieces are mapped to subbutterflies that contain at least $2^{\ell-1}$ outdated servers. In the following we argue that all remaining data pieces can correctly be recovered, implying the lemma. Hence, the subbutterflies of the data pieces that have neither been mapped to a crashed subbutterfly nor to a subbutterfly that contains at least $2^{\ell-1}$ outdated servers contain less than $2^{\ell-1} + 2^{\ell-1} = 2^\ell$ crashed or outdated servers. In particular, these subbutterflies do not contain a witness tree. By Lemma 5.3, these subbutterflies can correctly be recovered. □

Analogously to Corollary 3.31 of the analysis of the Lookup Protocol of Basic IRIS, one can show the following lemma.

Lemma 5.20. *Consider a request for a data item d that belongs to level $\ell \in \{1, \ldots, \log_k n\}$. It holds:*

(i) If at least $c/6$ requests for data pieces of d are aborted during the butterfly decoding in subphase ℓ due to too many crashed nodes, then the request for d is crashed at level ℓ.

(ii) If at least $c/6$ requests for data pieces of d are aborted during the butterfly decoding in subphase ℓ due to a congested node, then the request for d is congested at level ℓ.

Proof. We start with proving (i). By Lemma 5.3, we know that any ℓ-dimensional subbutterfly $BF(u)$ can be decoded correctly if and only if $BF(u)$ does not contain a witness tree. Hence, if $c/6$ requests for data pieces are aborted due to too many crashed nodes, then there exist $c/6$ subbutterflies that contain at least $2^{\ell-1}$ crashed nodes.

Part (ii) immediately follows from Definition 5.18. □

Now, we are ready to prove Lemma 5.17.

Proof of Lemma 5.17: We prove the lemma by induction on ℓ. The basis ($\ell = 1$) holds by Lemma 5.15. For the induction step, let $\ell \in \{1, \ldots, \log_k n - 1\}$ and assume that the induction hypothesis holds for level ℓ. We show that the number of requests that will be propagated to level $\ell + 1$ during subphase ℓ is at most $4\gamma n/k^\ell$. Together with Lemma 5.15, this means that at the beginning of subphase $\ell + 1$, at most $4\gamma n/k^\ell + 2\gamma n/k^\ell$ requests belong to level $\ell + 1$, which is upper bounded by $\varphi n/k^{\ell+1}$ and thus proves the induction step.

First, we show that each request for a data item that is neither crashed nor congested can be answered correctly at the end of subphase ℓ: i.e., the only requests that can be propagated to level ℓ are the ones that are crashed or congested at level ℓ. Hence, it then remains to upper bound the number of requests that are crashed and the number of requests that are congested at level ℓ.

Note that any request for a piece d_i of a data item d in subphase ℓ of the Decoding Stage can be aborted for only one of the following three reasons: First, too many servers storing information about d_i are crashed in the current period. Second, too many servers storing information about d_i are outdated (i.e., they were crashed when the bucket storing d was last updated). Third, a node from the subbutterfly $BF(u_{s,i}^{(\ell_i)}(d))$ is congested during the congestion check. By Lemma 5.20, for each request for a data item d belonging to level ℓ that is neither crashed nor congested, less than $c/6 + c/6$ requests for data pieces for d are aborted during the butterfly decoding in subphase ℓ due to too many crashed or congested nodes. Since in total there are at least $(5/6)c$ requests for data pieces of d in subphase ℓ with Lemma 5.19 we get that at least $(5/6)c - (2/6)c - c/6 = c/3$ data pieces can be recovered correctly, implying that the request for d is correctly served at the end of subphase ℓ.

In the following, we show that at most $2\gamma n/k^\ell$ requests are crashed at level ℓ and at most $2\gamma n/k^\ell$ requests are congested at level ℓ, which with our previous observations proves the lemma.

Upper bound on the number of crashed requests: Let S be a maximum set of data items with requests that are crashed at level ℓ. We show: $|S| < 2\gamma n/k^\ell$. Recall that a data item d is crashed at level ℓ if there exist at least $r = c/6$ pairwise different subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell$ that are crashed, i.e., each of them contains at least $2^{\ell_i - 1}$ crashed servers. For each $d \in S$, let $d_{i_1}, \ldots, d_{i_r}$ be $c/6$ such data pieces of d fulfilling this property. Further, define $F := \{(d, i_1), \ldots, (d, i_r) \mid d \in S\}$. Then, F is a $c/6$-bundle of S. Since a subbutterfly of level ℓ' contains $k^{\ell'}$ servers in total, and since a crashed subbutterfly of level ℓ' contains at least $2^{\ell'-1}$ crashed nodes, a $2^{\ell'-1}/k^{\ell'}$ fraction of the servers of a crashed subbutterfly on level ℓ' are crashed, which is at least $2^{\log_k n - 1}/n$ for any $1 \leqslant \ell' \leqslant \log_k n$. Since in total only at most $\gamma n^{1/\log\log n} = \gamma \cdot 2^{\log_k n}$ servers are crashed, the number of servers covered by all $BF(u_{s,i}^{(\ell_i)}(d))$ with $(d, i) \in F$ is less than $2\gamma n$. Since $\Gamma_{F,\ell}(S)$ is exactly the set of these servers, it holds: $|\Gamma_{F,\ell}(S)| < 2\gamma n$. With Corollary 3.25 we get $|S| < 2\gamma n/k^\ell$.

Upper bound on the number of congested requests: For the upper bound on the number of congested requests, recall that we call a subbutterfly $BF(v)$ of a node v congested if the servers in $BF(v)$ receive more than βck messages for different (d, i) pairs. Since $\beta > 5/2$, it holds that $\beta ck > 5\varphi c/(6 \cdot 2\gamma)$, which implies that a congested subbutterfly $BF(v)$ of a node v receives more than $5\varphi c/(6 \cdot 2\gamma)$ $\mathsf{decode}(d, i, t)$ messages for different (d, i) pairs. By the induction hypothesis and due to the fact that we send $5c/6$ $\mathsf{decode}(\cdot)$ messages per data item in the Decoding Stage, there are at most $5c/6 \cdot \varphi n/k^\ell$ $\mathsf{decode}(\cdot)$ messages in total, which means that there are less than $5c/6 \cdot \varphi n/k^\ell \cdot 6 \cdot 2\gamma/(5c\varphi) = 2\gamma n/k^\ell$ congested subbutterflies of dimension ℓ.

Let S be a set of data items with requests congested at level ℓ. Similar to the previous part about crashed data items, we can construct a $c/6$-bundle F for S. Since there are less than $2\gamma n/k^\ell$ congested subbutterflies of dimension ℓ and since each subbutterfly of dimension ℓ contains k^ℓ nodes, $|\Gamma_{F,\ell}(S)| < 2\gamma n$. Again, with Corollary 3.25 we get $|S| < 2\gamma n/k^\ell$, which completes the proof. □

Chapter 6

OSIRIS

In the previous chapters we considered an insider adversary that was allowed to crash a huge fraction of the servers. As a consequence, data stored at the servers may be outdated (when the servers become available again) or even completely inaccessible. Nevertheless, the servers still were able to detect crashed servers by using a crash detector and they were even able to detect outdated data by using timestamps whenever data is written into the system. However, the problem becomes even more challenging when considering servers whose storage may be corrupted arbitrarily. Note that we do not consider Byzantine servers here. Instead we assume an insider adversary that may arbitrarily corrupt the storage of the servers but it may not corrupt the main memory (holding the values and data computed during the execution of the protocol) and the protocol description. While crash failures at the servers or clients imply the absence of data items and therefore can be handled by applying techniques presented in the previous chapters, it is much more challenging to detect and handle corrupted data items. Standard error detecting or correcting codes do not scale in our case since the number of attacked servers we allow is asymptotically orders of magnitude larger than polylog(n), which normally requires a storage redundancy of the same magnitude.

An application of this kind of storage failures can be found in the context of DNS spoofing attacks. For that purpose, consider a setting where the n servers are split up into two entities each, a client c_i and a storage server s_i. The clients are reliable and do not contain any additionally storage except for the main memory. However, each client c_i is able to store data at storage server s_i. For this purpose, client c_i knows the domain name of storage server s_i. As is customary, the domain names of the storage servers are managed by

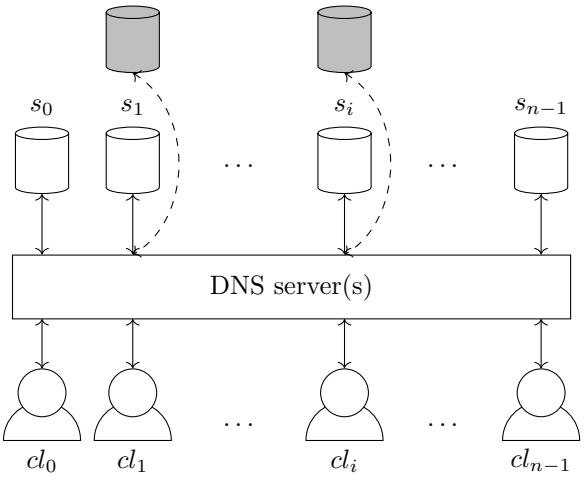

Figure 6.1: Model visualization. The upper entities denote the storage servers, the lower entities denote the clients. The dashed lines pointing to the gray servers denote a redirection of the domain names of storage server s_1 and s_i to a storage server maintained by an adversary containing arbitrary data.

domain name system (DNS) servers that translate the domains into numerical IP addresses. Using the approach of consistent hashing, it is easy to design a distributed storage system in which the clients act as worker nodes that receive and answer requests for data held at the storage servers. However, many difficulties and challenges arise when considering the existence of an insider adversary that knows everything about the system and has the ability to use this knowledge in order to run a DNS spoofing attack on several DNS servers which causes the DNS servers to return incorrect (arbitrarily chosen) IP addresses. By these means, the adversary may falsify the IP addresses of storage servers such that the returned addresses point to storage servers maintained by the adversary itself. This enables the adversary to arbitrarily corrupt the data returned to the clients (see Figure 6.1).

In order to increase the readability and to not blow up the construction and hide the main innovations behind our system, in the following we assume the first mentioned view onto the system where we solely are given n servers whose storage may be arbitrarily corrupted except for their main memory and their protocol description. However, all protocols and proofs given equally apply to the model with separated clients and storage servers.

In the following of this chapter we present OSIRIS, a scalable distributed storage system that is provably robust against an insider adversary who can corrupt the storage of a large fraction of servers. At the same time, OSIRIS

only needs a logarithmic storage redundancy. Despite this powerful attack, our system is able to correctly serve any set of lookup and write requests with $O(1)$ requests per server in at most polylog(n) time and work per server.

Theorem 6.1 (OSIRIS Main Theorem)**.** *Assume an insider adversary arbitrarily corrupts at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/64$. Then, using only a logarithmic redundancy, OSIRIS correctly serves any set of lookup and write requests (at most $O(1)$ per server) after at most $O(\log^2 n)$ communication rounds with a congestion of at most $O(\log^3 n)$ at every server in each round, w.h.p.*

In the following we recap the main model specifications and introduce further preliminaries (Section 6.1). Afterwards, we present the storage strategy used in OSIRIS (Section 6.2). In Sections 6.3 and 6.4 we describe a lookup and write protocol. Since the correctness analysis of the lookup protocol is rather involved, we shifted it into a separate Section (Section 6.5).

6.1 Preliminaries

Just as in Chapter 5, we propose protocols for two types of requests: lookup requests and write requests. As well as in all previous chapters, we assume a synchronous time model, where time proceeds in rounds that are divided into periods. We assume a batch-based adaptive insider adversary: i.e., the adversary has complete knowledge of the system and can use this knowledge to arbitrarily corrupt the storage of up to $\gamma n^{1/\log\log n}$ servers (excluding their main memory and their protocol description) at the beginning of each period with $\gamma = 1/64$. We call this kind of failures caused by the adversary **storage failures**. Furthermore, we denote a server to be **corrupted** if the adversary corrupted the storage of that server. Note that a corrupted server is not aware of being corrupted. A server that is not corrupted is called **intact**. In addition to causing storage failures, the adversary also specifies the set of requests sent to the servers. That is, we consider worst-case storage corruptions and worst-case requests.

Since on the one side we assume the adversary to be adaptive (i.e., it may choose a new set of servers to be corrupted at the beginning of each period) and on the other side we assume the number of attacked servers to be upper bounded by $\gamma n^{1/\log\log n}$, we additionally make the following assumption on the storage state of the attacked servers: Whenever a server s is attacked in period p and it is "released" again in period $p' > p$ (i.e., it is not attacked any more), then at the beginning of period p' the storage state of server s is reset to its storage state at the end of period $p - 1$. The assumption of automatically resetting the states of previously corrupted servers can in the context of DNS spoofing attacks be motivated as follows: Whenever the adversary decides

to reset the falsification of an IP address of a server s in order to falsify the IP address of another server, the IP address for server s returned by the DNS system points to server s again. However, server s is not aware of the changes made during the time its IP address was falsified. In particular, the storage of server s is the same as it was just before the IP address of s was falsified.

Note that by this reset mechanism, a server s that was corrupted during a period p in which data has been written into the system is not aware of that change once it is not corrupted anymore. Similarly to RoBuSt, we call such a server **outdated**.

While in the previous chapters (Chapters 3–5) we assumed the servers to initially build a clique, we can weaken this requirement in this chapter and instead only require the servers to be connected with each other via a $\log n$-ary butterfly (see Section 2.2). Hence, each server is not required to have a quadratic degree anymore but a degree of $O(\log^2 n / \log\log n)$ suffices. We are able to weaken our requirements in that way since the protocols of the previously presented systems required the determination of representatives of the crashed servers in a preprocessing stage which again required the servers to form a clique. However, in the setting considered in this chapter we are not able to determine intact representatives for the corrupted servers, since we cannot differentiate between corrupted and intact servers. Instead, we will come up with other techniques in order to deal with the existence of corrupted servers.

Since the storage strategy of OSIRIS will make use of authentication techniques, we require the existence of one-way hash functions $g : \{0,1\}^* \to \{0,1\}^{\log^2 n}$ that cannot be inverted by a polynomially bounded adversary with a reasonable amount of work with high probability if n is sufficiently large. Since the adversary must not be able find collisions in the one-way hash functions we use, we additionally require the adversary to be polynomially bounded.

In order to guarantee a logarithmic storage redundancy, we assume the size of data items to be at least $\Omega(\log^4 n)$ and at most polylog(n). However, bigger data items could still be handled by our system, but in this case the maximum message size is proportional to the size of the biggest data item in the system. Alternatively, huge data items can be split into several chunks such that each of them is of size at most polylogarithmic in n.

In order to authenticate data blocks, the storage strategy of OSIRIS makes use of the well-known Merkle trees [Mer79].

Definition 6.2 (Merkle tree)**.** *Let $b_1, \ldots, b_N$, $N \in \mathbb{N}$, be a set of data blocks and let g be a one-way hash function. A tree T is called a **Merkle tree** of $b_1, \ldots, b_N$ if it holds: $b_1, \ldots, b_N$ are stored at the leaves of T, every leaf is assigned the hash value of the data item it stores, and every inner node is assigned the hash value of the concatenation of*

the child nodes' assignments.

See Figure 6.2 for a visualization of a Merkle tree.

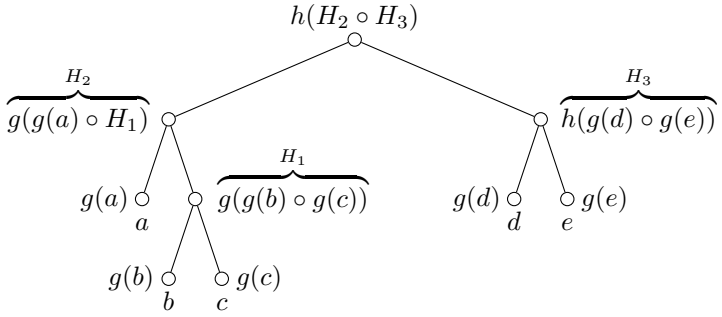

Figure 6.2: Example of a Merkle tree. The variables below the child nodes denote the data stored at those nodes; the remaining values denote the hash values assigned to the particular nodes.

Given a tree T with data blocks assigned to its leaves, we use the notion **compute the Merkle tree of** T for the procedure of computing the hash values of all nodes of T beginning with the hash values of the data blocks assigned to the leaves of T and assigning the computed hash values to the nodes. In the following let g be a one-way hash function with $|g(x)| = O(\log^2 n)$ for all x. For a data block b, we use the notion $g(b)$ to denote the hash value of b; whereas for a node u in a Merkle tree, we use the notion $g(u)$ to denote the Merkle hash value that is assigned to u.

Table 6.1 provides an overview of variables and their bounds that are commonly used in this chapter.

Term	Bound	Description
γ	$= 1/64$	Constant in fraction of corrupted servers from $n^{1/\log\log n}$ servers
ε	$< \gamma n^{1/\log\log n}$	Fraction of corrupted servers
α	$> 2(1-\varepsilon)/\gamma$, e.g., $\geqslant 32$	Constant in congestion bound in Probing Stage
β	$> 5/2$	Constant in congestion bound in Decoding Stage
c	$\geqslant 16 \log m$	Number of pieces created for each data item

Table 6.1: Variables commonly used in the presentation of OSIRIS.

6.2 Storage Strategy

In OSIRIS we reuse the concepts of the underlying data structure already used in RoBuSt. That is, data items are stored into buckets that are arranged in a tree structure (see Section 5.2). In order to be able to handle storage failures, instead of solely crash failures as RoBuSt does, we need to revise the internal storage strategy of a single bucket, which is described in the following.

Let $b_1, \ldots, b_n$ be n data blocks that have already been assigned to the servers and that are supposed to be stored into a single bucket. This storing procedure not only consists of simply storing the plain data at the servers, but also of encoding the data blocks with each other and of computing authentication information for the data blocks which again will also be encoded with each other. For the encoding we build upon the framework of the **k-ary Butterfly Coding Strategy**, as already introduced in IRIS and RoBuSt and develop a novel code for use in this framework that suits our needs. This distributed code is presented in Section 6.2.1, followed by a descriptions of the steps necessary to perform in order to store the data blocks $b_1, \ldots, b_n$ in a single bucket in Section 6.2.2.

6.2.1 Internal Distributed Error Detecting and Correcting Code

One of the main ingredients of the Butterfly Coding Strategy (Section 3.2) is a distributed code that guarantees the correction of a single block if the faulty block is known and that appends only some parity bits to the input data blocks in the result. In OSIRIS we replace this code with a new code that not only guarantees the correction of a single data block but is additionally able to detect a single faulty data block (if it is the only one) in a given set of data blocks that are encoded with each other using this strategy. This code builds on Hamming codes and in its result only some parity bits are appended to the input data blocks. Furthermore, the concepts of this code can be used to transform any centralized single-error detecting and correcting coding strategy that appends bits to the original message only into a *distributed* single-error detecting and correcting coding strategy.

Before we provide a detailed description of the code, we introduce some basic notions. For a bit string x of length z let $\textsc{HammingCode}(x) = (x, p_1 \ldots p_{r(z)})$ denote the computation of the Hamming code [Ham50] on input x with result $(x, p_1 \ldots p_{r(z)})$, where the p_i denote the parity bits generated during the computation of the Hamming code of x and $r(z)$ denotes the number of those parity bits for x. Note that we do not use standard Hamming codes, but a slightly modified version of Hamming codes, with the only difference being that the parity bits are not distributed between the single bits of the input strings, but they are appended to the end of the input string. Obviously, this

modification harms neither the correctness nor the efficiency of Hamming codes. Furthermore, for $j \in \{1, \ldots, z\}$, $x(j)$ denotes the j-th bit of x. The concatenation of two bit strings x and y is denoted by $x \circ y$.

The encoding of k data blocks $b_1, \ldots, b_k \in \{0,1\}^z$ (visualized in Figure 6.3) resulting in k encoded data blocks $b'_1, \ldots, b'_k$ works as follows:

1. For $j \in \{1, \ldots, z\}$ define $W_j := b_1(j) \circ \ldots \circ b_k(j)$ as the bit string consisting of the j-th bit of $b_1, \ldots, b_k$. Apply HammingCode(W_j) for all $j \in \{1, \ldots, z\}$, resulting in (W_j, T_j) with $|T_j| = r(k)$.

2. Define $T := T_1 \circ T_2 \ldots \circ T_z$ and cut T into k pieces of equal size denoted by $L_1, \ldots, L_k$.

3. Set $b'_i = b_i \circ L_i \circ L_{prev(i)} \circ L_{next(i)}$, $i \in \{1, \ldots, k\}$, where $next(i) = i \mod k + 1$ and $prev(i) = (i - 2) \mod k + 1$.

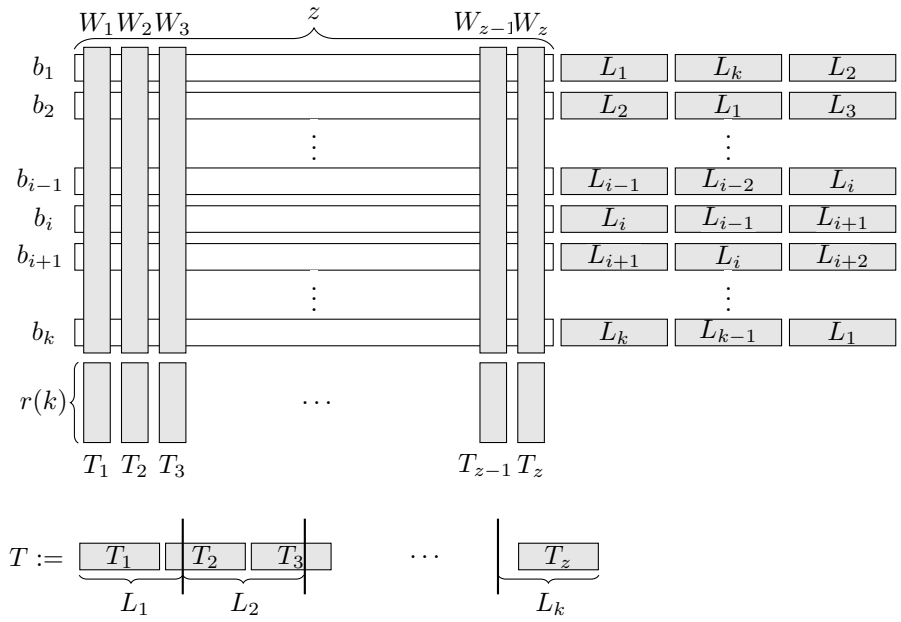

Figure 6.3: Visualization of the error detecting and correcting coding strategy.

As the following lemma formalizes, the presented code ensures that in an encoded block of k data blocks, a single erroneous data block can be detected and corrected.

Lemma 6.3. *Let the data blocks $b_1, \ldots, b_k$ be encoded with each other using the previously described distributed coding strategy resulting in $b'_1, \ldots, b'_k$. If a single*

data block from $b'_1, \ldots, b'_k$ is corrupt, then the information in the remaining b'_i suffices to detect and correct the erroneous data block.

Proof. W.l.o.g. we may assume that server s_i holds data block b_i, $i \in \{1, \ldots, k\}$. For server $s_i,\ i \in \{1, \ldots, k\}$, let $\tilde{b'_i} := \tilde{b_i} \circ \tilde{L_i} \circ \tilde{L}_{prev(i)} \circ \tilde{L}_{next(i)}$ denote the data block s_i stores for the currently considered bucket. That is, if $\tilde{b'_i}$ is not corrupt, $\tilde{b'_i} = b'_i$; otherwise $\tilde{b'_i}$ differs in at least one bit from b'_i.

In the following let $\tilde{b'_c}$ be the only corrupted data block in $\tilde{b'_1}, \ldots, \tilde{b'_k}$. That is, at least one of the following cases holds: $\tilde{b_c} \neq b_c$, $\tilde{L}_{prev(c)} \neq L_{prev(c)}$, $\tilde{L}_{next(c)} \neq L_{next(c)}$, or $\tilde{L}_c \neq L_c$.

Via the following procedure the servers can detect $\tilde{b'_c}$ to be corrupted and repair it:

1. Correction of $\tilde{L}_c$:
   - Each server s_i sends $\tilde{L}_{prev(i)}$ to server $s_{prev(i)}$ and $\tilde{L}_{next(i)}$ to server $s_{next(i)}$.
   - For server s_i let $\tilde{L}_{i,1}$ and $\tilde{L}_{i,2}$ be the two data blocks s_i received from $s_{prev(i)}$ and $s_{next(i)}$. Note that in case none of the blocks $\tilde{b'_i}, \tilde{b'}_{prev(i)}, \tilde{b'}_{next(i)}$ is corrupted it holds: $\tilde{L_i} = \tilde{L}_{i,1} = \tilde{L}_{i,2} = L_i$.
   - Since we assume only a single data block to be corrupted, for at most one data block $\tilde{L} \in \{\tilde{L_i}, \tilde{L}_{i,1}, \tilde{L}_{i,2}\}$ it holds $\tilde{L} \neq L$, while for the remaining two data blocks $\tilde{p}_1, \tilde{p}_2 \in \{\tilde{L_i}, \tilde{L}_{i,1}, \tilde{L}_{i,2}\}$ it holds $\tilde{p}_1 = \tilde{p}_2 = L_i$.
   - If a server s_i detects that exactly one of the received data blocks $\tilde{L}_{i,1}$ or $\tilde{L}_{i,2}$ is not equal to $\tilde{L_i}$, then the server that sent the data block that did not equal $\tilde{L_i}$ is the corrupted server and $\tilde{L_i} = L_i$, i.e., c_i does not hold corrupted data.
   - If a server s_i detects $\tilde{L_i} \neq \tilde{L}_{i,1}$ and $\tilde{L_i} \neq \tilde{L}_{i,2}$, then s_i is the corrupted server (since only a single server was assumed to be corrupted) and updates $\tilde{L}_c = \tilde{L}_{c,1}$.
   - If $\tilde{L_i} = \tilde{L}_{i,1} = \tilde{L}_{i,2}$, neither s_i, nor $s_{prev(i)}$, nor $s_{next(i)}$ is corrupted and it holds $\tilde{L_i} = L_i$.
2. Correction of $\tilde{b_c}$: Each server s_i sends b_i to all $k-1$ other servers such that each server can compute T_j for all $j \in \{1, \ldots, z\}$. Since s_c is the only corrupted server, each server s_i can detect (and possibly correct) each corrupted bit $b_c(j)$, $j \in \{1, \ldots, z\}$ such that at the end for each server s_i it holds $\tilde{b_i} = b_i$.

At the end of this procedure for each server s_i it holds: $\tilde{b'_i} = b'_i$. □

6.2.2 Storage Strategy of a Single Bucket

In the following we present the steps required for storing n data items in a bucket B of OSIRIS. In order to keep the description as clear as possible, some of these steps are presented as a centralized algorithm, but their transformation into a distributed one is straightforward.

Step 1: Use Reed-Solomon codes in order to create $c = 8 \log m$ pieces of each of the n data items d such that each set of $\lfloor c/4 \rfloor$ of these pieces suffices to recover d. For each data item d map the previously created c pieces $d_1, \ldots, d_c$ of d to the servers using c hash functions $h_{(1,B)}, \ldots, h_{(c,B)} : \mathcal{U} \to [0, 1)$ chosen uniformly at random. Since the hash functions are chosen uniformly at random, by the Chernoff bounds (Lemma 2.1), each server will hold c data pieces, w.h.p. We call the concatenation of the data pieces mapped to a server c_i **data block** b_i. We also use the notion of b_i being **assigned to** server c_i in bucket B.

Step 2: Encode $b_1, \ldots, b_n$ with each other using the Butterfly Coding Strategy (see Section 3.2 while using the code presented in Section 6.2.1 as the internal error detecting and correcting code). Let DATA-BF(B) denote the k-ary butterfly consisting of the virtual nodes that store the data pieces and the parity information computed during this encoding process.

Step 3: For each butterfly node $u_i = (\log_k n, i)$, $i \in \{1, \ldots, n\}$, at level $\log_k n$ of the k-ary butterfly compute the Merkle tree of the complete k-ary upper tree $UT(u_i)$ rooted at u_i. Since the hash value $g(u)$ of a butterfly node u is in general different for different buckets, we call $g(u, B')$ the hash value of u in bucket B'. The Merkle tree of $UT(u_i)$ is determined in a top-down fashion (i.e., from level 0 to level $\log_k n$) by computing the hash values of all nodes in $UT(u_i)$ for bucket B. Here, the leaves are labeled with $g(b_1, B), \ldots, g(b_n, B)$ and each inner node v is labeled with $g(g(v_1, B) \circ \ldots \circ g(v_k, B), B)$, where $v_1, \ldots, v_k$ denote the children of v in $UT(u_i)$. With $\text{MT}_{B'}(u_i)$ we refer to the computed Merkle tree rooted at u_i for a bucket B'. Moreover, at each butterfly node v with children $v_1, \ldots, v_k$ in $UT(u_i)$ we not only store the hash value $g(v, B)$, but additionally store the hash values $g(v_1, B), \ldots, g(v_k, B)$: i.e., each butterfly node v stores $G_B(v) := g(v, B) \circ g(v_1, B) \circ \ldots \circ g(v_k, B)$ for bucket B.

Step 4: At each server s we store all strings $G_B(u)$ of the nodes u that server s emulates in bucket B. Let $G_B(s)$ denote the concatenation of all these $G_B(u)$. Encode the data blocks $G_B(s_0), \ldots, G_B(s_{n-1})$ with each other using the Butterfly Coding Strategy (see Section 3.2 again while using

the code presented in Section 6.2.1 as the internal error detecting and correcting code). Let MERKLE-BF(B) denote the k-ary butterfly consisting of the virtual nodes that store the strings $G_B(s_0), \ldots, G_B(s_{n-1})$ and the according parity information computed during this encoding process.

Note that by this procedure we create two virtual k-ary butterflies: DATA-BF(B), which holds the encoded data items, and MERKLE-BF(B), which holds the encoded Merkle values of the Merkle trees.

One can show that this storage strategy requires only a constant redundancy for each server and each bucket (Lemma 6.4).

Lemma 6.4. *For a server s let $d(s)$ denote the concatenation of data and additional information computed during the previously described encoding process for storing a set of n data items, each of length z, into a bucket B. It holds: $|d(s)| \leqslant \lambda \cdot z$ for a sufficiently large constant $\lambda > 0$.*

Proof. First, we analyze the storage $S(z)$ required for each server when encoding n data blocks, each of length z, with each other using the Butterfly Coding Strategy. With $r(k)$ denoting the number of parity bits appended to a string of length k when encoding that string with Hamming codes, it holds: $S(z) := \sum_{\ell=1}^{\log_k n} z \cdot \left(1 + \frac{3r(k)}{k}\right)^{\ell}$. Using $(1+x) \leqslant e^x$ for $x \geqslant 1$ this term can be upper bounded by $\log_k n \cdot z \cdot e^{\frac{3 \log_k n \cdot r(k)}{k}}$. Since $r(k) \leqslant \log k + 1$ [Ham50], the exponent in this term is upper bounded by $\frac{3 \log n \cdot (\log k + 1)}{\log k \cdot k}$, which can be upper bounded by a constant such that all in all we get $S(z) \leqslant \rho \log_k n \cdot z$ for a sufficiently large constant $\rho > 0$.

Now we analyze the storage required for each server in each of the above steps:

Step 1: Using Reed-Solomon codes instead of storing the plain data item the storage overhead needed for storing c pieces of a data item d_j with length z increases only by a constant factor [RS60]. This implies that after the mapping of c pieces to each server, each server stores data blocks of length $z' \in O(z)$.

Step 2: With the above consideration, the encoding of the n data blocks $b_1, \ldots, b_n$ with each other increases the storage overhead at any server by a factor of $\rho \log_k n$ for a sufficiently large constant $\rho > 1$.

Step 3: After the computation of all Merkle trees, by Lemma 6.7, each butterfly node u on level ℓ, $1 \leqslant \ell \leqslant \log_k n$, additionally stores $k+1$ hash values (its own hash value $g(u, B)$ and the hash values of its k children in $UT(u)$), each of length $O(\log^2 n)$. The butterfly nodes on level 0 only

need to store their own hash value. Hence, for the storage amount of a single butterfly node for a single bucket B it holds: $|G_B(u)| = O(k \cdot \log^2 n)$.

Step 4: Since server s_i, $i \in \{1, \ldots, n\}$, emulates all butterfly nodes in column i, s_i stores $O(k \log_k n)$ hash values in total. Since each hash value has a length of $O(\log^2 n)$, server s_i needs to store a data block of length $O(k \log_k n \log^2 n)$. The final encoding of all these data blocks using the Butterfly Coding Strategy increases the storage amount only by a constant factor of $\rho' \log_k n$ for $\rho' > 1$ constant.

All in all, for data blocks of length $z \in \Omega(\log_k^2 n \log^2 n)$, as assumed in Section 6.1, for the required storage amount $d(s)$ at a server s it holds: $|d(s)| \leqslant \lambda z$, for a sufficiently large constant $\lambda > 0$. □

As already argued in Section 5.2, when data items are updated or deleted it may happen that old versions of the modified data item remain in the storage system, but only at most one data item for each zone of the bucket tree. Since there are at most $\Lambda + 1 = O(\log n)$ zones in total, Lemma 6.4 implies that OSIRIS requires an overall redundancy of $O(\log n)$.

Lemma 6.5. *OSIRIS has an overall redundancy of* $O(\log n)$.

6.3 Lookup Protocol

Recall that the data items stored in OSIRIS are not stored in a single bucket, but they are stored in several buckets that are arranged in a tree-like structure, the bucket tree. For each data item there are $\Theta(\log n)$ possible buckets where to store that data item. These buckets form a path from the root of the bucket tree to one of its leaves. Hence, in order to serve a lookup request for some data item d, we traverse this path, beginning at the root and at each bucket we try to serve the lookup request for d by performing the protocol described in the following. In order to handle a set of up to n lookup requests, we traverse the bucket tree beginning at the root and consider at each level several buckets that are potential storage locations for the requested data items in parallel.

Note that a server does not only store a single data piece or data item for a bucket. Instead it stores a data block that contains several data pieces and coding information. Hence, when searching for a specific data piece located at a server s in a bucket B, we do not just recover that single data piece, but we recover the complete data block stored at server s for bucket B.

Before we provide a detailed description of the Lookup Protocol, we give an overview of the stages and their results.

6.3.1 Outline of the Lookup Protocol

The Lookup Protocol consists of two main stages executed sequentially: The Probing Stage and the Recovery Stage.

The Probing Stage (see Section 6.3.2) consists of $O(\log_k n)$ phases with the following result: For each request for a data item d, either the request is correctly answered or the request is assigned to a level $\ell \in \{1, \ldots, \log_k n\}$. In the latter case, for sufficiently many data pieces d_i of d the first $\log_k n - \ell$ entries of the hash chain of d_i are correctly stored in the according butterfly nodes of MERKLE-BF(B), where B denotes the currently considered bucket for d.

The Recovery Stage (see Section 6.3.3) handles the requests that have not been served in the Probing Stage and consists of two further sequentially performed substages, both consisting of $O(\log_k n)$ subphases: The Hash Chain Recovery Stage (see Section 6.3.3.1) and the Data Recovery Stage (see Section 6.3.3.2), which, in short, work as follows:

1. *Hash Chain Recovery Stage:* In subphase $\ell \in \{1, \ldots, \log_k n\}$, the requests that have been assigned to level ℓ before are handled. That is, the remaining ℓ entries of the hash chains of sufficiently many pieces of the considered data items are tried to be recovered. This process may fail due to excessive congestion at a node which causes the request for the according data item to be (re-)assigned to level $\ell + 1$. At the end of this stage it holds: For sufficiently many data pieces of any requested data item (whose request has not been served in the Probing Stage) the remaining entries of sufficiently many of the data piece's hash chains are correctly recovered.

2. *Data Recovery Stage:* In subphase $\ell \in \{1, \ldots, \log_k n\}$, the requests that have been assigned to level ℓ before are handled. At the end of phase $\ell \in \{1, \ldots, \log_k n\}$ of this stage it holds: Sufficiently many data pieces of each requested data item (whose request has not been served in the Probing Stage) are recovered (using the Butterfly Coding Strategy) and verified using the hash chain. If the verification of too many data pieces fails, the request is reassigned to level $\ell + 1$ and further handled in the next phase of the Data Recovery Stage. Otherwise, the requested data item can be recovered using RS-codes. At the end of the overall stage each previously unserved lookup request will be served correctly.

Recall that at the beginning of the Lookup Protocol of IRIS and RoBuSt a Preprocessing Stage is performed in which for each attacked server a unique intact server as its representative is determined. Since in the setting considered for OSIRIS we are not able to detect whether a server is attacked (i.e., corrupted) or not, we cannot determine representatives for these servers. Consequently, we do not need to perform a Preprocessing Stage here.

6.3.2 Probing Stage

In the Probing Stage, each server s with a request for a data item d tries to retrieve and verify at least $c/4$ pieces of d.

High-Level overview: First, each server s with a request for a data item d chooses c servers $s_1(d), \ldots, s_c(d)$ uniformly at random and asks each server $s_i(d)$ to retrieve and verify piece i of d: i.e., d_i. If at least $c/4$ of these servers answer with verified pieces, s recovers d using these pieces and Reed-Solomon codes and serves the request. Otherwise, the request for d cannot be served in the Probing Stage, but will instead be handled again in the Recovery Stage (see Section 6.3.3).

Details on the Probing Stage: In the following, in order to reduce the number of messages sent, we use the technique of Splitting and Combining for sending messages along the probing path just as in IRIS and RoBuSt. This technique at each intermediate node simply combines all requests with the same target and takes care of splitting the corresponding subsequently received answers again.

Each server s with a request for a data item d first chooses c servers $s_1(d), \ldots, s_c(d)$ uniformly at random and also informs these servers about being chosen such that these servers can initiate the retrieval and verification of data piece d_i belonging to bucket B. The retrieval and verification of a data piece d_i belonging to a bucket B and initiated by a server $s_i(d)$ consists of the following procedures:

Step 1: *Initialization:* $s_i(d)$ initiates sending the request for d_i bottom-up along the probing path of d_i in the Merkle butterfly MERKLE-BF(B) from level $\log_k n$ to level 0. In the following let the **hash chain** of d_i be the list consisting of $\log_k n$ entries, where each entry represents the hash values a butterfly node on the probing path of d_i holds in $\mathrm{MT}_B(u)$. That is, the j-th entry of the hash chain of d_i is of the form $(g(u_j, B), g(u_j^{(1)}, B), \ldots, g(u_j^{(k)}, B))$, where u_j denotes the j-th node on the probing path of d_i and $u_j^{(1)}, \ldots, u_j^{(k)}$ denote the children of u_j in $UT(u_j)$.

Step 2: *Congestion check and acquisition of the correct hash values:* Each node v on a level $\ell > 0$ on the probing path of d_i that receives requests for data pieces first performs a simple congestion check (note that v may lie on several probing paths and that if v is on level ℓ then all nodes that receive requests in the current round are on level ℓ): If the number of messages received in the current round is greater than $\alpha \cdot c$ (for an arbitrary fixed constant $\alpha \geqslant 32$), it stops the forwarding of all requests received in the current round and informs all origins $s_i(d)$ of these

requests about that (via top-down routing along the corresponding probing paths). If the number of messages received in the current round is at most $\alpha \cdot c$, node v tries to obtain the correct hash values for itself and its children in $UT(v)$ (details on this are provided in Section 6.3.2.1). If this is not successful, v stops the forwarding of the request for d_i and informs its origin $s_i(d)$ about that (via top-down routing along the corresponding probing paths).

Step 3: *Destination reached:* When a request initiated from a node u at level $\log_k n$ has reached its destination node w on level 0, w checks whether $g(b_i, B)$ matches the Merkle hash value that w's parent in $UT(u)$ stores for it, where b_i is the data block stored at w containing d_i. By the above process, w.h.p. this is true if and only if b_i is correct. In that case, w tries to extract d_i from b_i. Depending on whether this fails (which means that d does not exist) or is successful, w then sends a dataNotFound(d) message or the data piece d_i, respectively, back along the probing path to node u. In the other case in which the Merkle hash values do not match, w informs (via top-down routing along the probing path of d_i) node u about the fact that the request failed because d_i could not be verified.

Step 4: *Answering:* If the probing was successful, i.e., the block b_i to which d_i belongs was verified correctly, $s_i(d)$ forwards the data piece d_i or a dataNotFound(d) message to server s that received the lookup request for d. If the probing failed, i.e., the request for d_i was aborted at a level $\ell' \in \{0, \ldots, \log_k n\}$ (which may happen either due to the congestion check in Step 2, or because the data item is corrupt, as determined in Step 3), $s_i(d)$ notifies server s about the level ℓ' at which the fail occurred.

As mentioned before, requests that could not be answered in the Probing Stage will be handled again in the Recovery Stage. For this we need the following notion.

Definition 6.6 (Belong to)**.** *We say a request for a data piece d_i **failed at level** ℓ if the forwarding of that request stops at a node at level ℓ in the Merkle Hash Butterfly (for any of the reasons mentioned in Steps 2 – 4). A request d that could not be answered in the Probing Stage is said to **belong to level** $\ell \in \{1, \ldots, \log_k n\}$, where ℓ is the smallest level such that at least $(5/8)c$ pieces of d have not failed at level ℓ or greater.*

6.3.2.1 Details on Obtaining the Correct Hash Values

In the following we describe how the nodes can make sure that they store the correct hash values for themselves and their children as required in Step 2

of the probing. For this purpose, we make use of the following lemma that follows from the symmetry of the k-ary butterfly.

Lemma 6.7. *Let u and v be two butterfly nodes on level $\ell \in \{1, \ldots, \log_k n\}$ with $BF(u) = BF(v)$. Then, for any bucket B that stores data it holds: $g(u, B) = g(v, B)$.*

Note that a direct implication of Lemma 6.7 is that all nodes at level $\log_k n$ in the Merkle Hash Butterfly store the same Merkle Hash value.

As we will see, obtaining the correct hash values of the children and itself for a node u on the probing path is done by exploiting Lemma 6.7 and using that inductively the parent of u on the probing path has correctly recovered the hash value of its children before. Since the first node on the probing path (the node emulated by $s_i(d)$ on level $\log_k n$) does not have any parent and the last node of the probing path (the node emulated by $s_i(d)$ on level 0) does not have any children, the recovery of the hash values for these nodes work in a different way than for the nodes at a level $\ell \in \{1, \ldots, \log_k n - 1\}$ on the probing path. In the following let u be a node on the probing path of a piece d_i of a data item d that is supposed to recover its hash values in Step 2 of the Probing Stage.

Case 1: u is at level $\log_k n$.

1. u sends the hash value $g(u, B)$ and the hash values $g(u^{(1)}, B), \ldots, g(u^{(k)}, B)$ it stores for its children $u^{(1)}, \ldots, u^{(k)}$ in $UT(u)$ to the server s that initiated the probing for data item d.
2. The server s receives the hash values sent from all c servers $s_1(d), \ldots, s_c(d)$ it contacted at the beginning of the Probing Stage and selects the tuple T it received most often.
3. s sends the tuple T to all c servers $s_1(d), \ldots, s_c(d)$ which causes each server $s_i(d)$ that emulates node u at level $\log_k n$ to replace its values $g(u, B)$ and $g(u^{(1)}, B), \ldots, g(u^{(k)}, B)$ with the received hash values.

Case 2: u is at level $\ell < \log_k n$.

1. Node u chooses $\Theta(\log n)$ nodes on level ℓ from $BF(u)$ uniformly at random and asks them for their hash values (see Figure 6.4). Since the server emulating u is not necessarily directly connected to the servers emulating the chosen nodes, u informs the chosen nodes about being chosen via a bottom-up routing in the k-ary butterfly (see Section 2.2). Since the target nodes are chosen uniformly at random, by Valiant's trick [Val82] this does not yield more than a congestion of $O(\log^2 n)$ at any intermediate node.

2. For each of the $\Theta(\log n)$ hash value tuples $(\tilde{g}(x), \tilde{g}(x^{(1)}, B), \ldots, \tilde{g}(x^{(k)}, B))$ received by a node x chosen uniformly at random before, u performs the following tests:
   a) *Feasibility test:* Check whether the tuple is feasible: i.e., whether $g(\tilde{g}(x^{(1)}, B) \circ \cdots \circ \tilde{g}(x^{(k)}, B), B) = \tilde{g}(x, B)$.
   b) *Parent test:* If the tuple is feasible, ask u's parent node (on the probing path for the currently considered data piece d_i) for the hash value it stores for u and check whether this equals $\tilde{g}(x, B)$.
3. If for one of the values received from a node x both tests are successful, u uses the hash values obtained from x from now on, which also finishes the procedure of obtaining the current hash values for node u.
4. If for all hash values received from any node both tests fail, the probing for the current piece d_i of d is aborted (cf. Step 2 of the Probing Stage).

Case 3: u is at level 0. In this case u simply asks its parent node on level 1 on the probing path for d_i for its hash value and updates its own hash value to the value received.

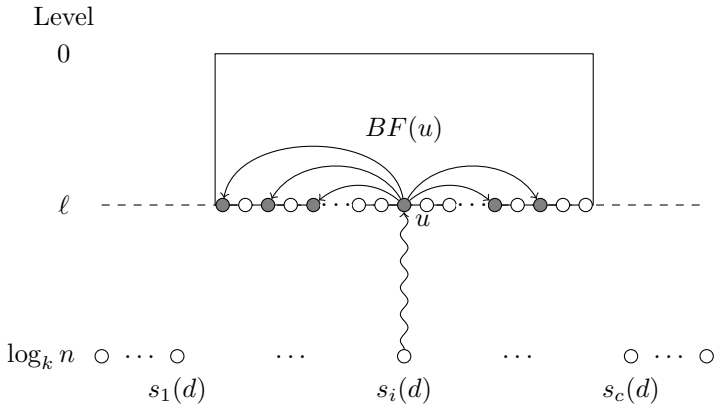

Figure 6.4: Visualization of the Probing Stage for the case that node u is not at the lowest level. The gray colored nodes denote the $\Theta(\log n)$ nodes u chooses uniformly at random.

As one can show, at the end of this procedure u stores the correct hash values for $g(u, B), g(u^{(1)}, B), \ldots, g(u^{(k)}, B)$ (Lemma 6.8) as long as only less than half of the servers in $BF(u)$ are corrupted or outdated.

Lemma 6.8. *Let u be a node at level $\ell \in \{1, \ldots, \log_k n\}$ of the k-ary butterfly that is reached during the probing for a data piece d_i. If only less than half of the servers in $BF(u)$ are corrupted or outdated, then at the end of the previously described procedure for obtaining the current hash values, all nodes on the probing path of d_i from level $\log_k n$ up to level ℓ store the correct hash values, w.h.p.*

In order to prove this lemma, we first need to show the following lemma:

Lemma 6.9. *If the probing for a data piece d_i reaches a node u at level $\ell \in \{0, \ldots, \log_k n - 1\}$, then the hash chain from level $\log_k n$ up to level $\ell + 1$ is correctly stored in the corresponding nodes of the probing path of d_i.*

Proof. We show the claim by induction of ℓ. The inductions base is for $\ell = \log_k n - 1$. In the previous phase of the Probing Stage the node u at level $\log_k n$ of the probing path for d_i received the hash value tuple it is supposed to hold from the server s that initially received the lookup request for d. Since the servers $s_1(d), \ldots, s_c(d)$ were chosen uniformly at random by s, with Chernoff bounds (Lemma 2.1) we can conclude that only less than half of the servers from $s_1(d), \ldots, s_c(d)$ are corrupted or outdated, w.h.p. Hence, more than half of the servers hold the correct hash values, w.h.p., implying that the tuple $(g(u, B), g(u^{(1)}, B), \ldots, g(u^{(k)}, B))$ that server s received most often from the servers $s_1(d), \ldots, s_c(d)$ is the one with the correct hash values and this is also the tuple it forwards to to all servers $s_1(d), \ldots, s_c(d)$ such that finally node u also holds the correct hash values.

Next, let $\ell \in \{0, \ldots, \log_k n - 2\}$ and assume the claim holds for all $\ell' > \ell$. Let u be the node reached at level ℓ. By the induction hypothesis we know that the hash chain from level $\log_k n$ up to level $\ell + 2$ is correctly stored at the corresponding nodes of the probing path for d_i. Furthermore, since the probing reached node u, the parent node of u on level $\ell + 1$ passed both, the feasibility test and the parent test for a hash value tuple it received from one of the nodes from $BF(u)$ it has chosen before. Since the hash function used in the computation of the Merkle trees is a one-way hash function both tests can only be passed if the considered hash value tuple is correct which implies the claim. □

Now, we are ready to prove Lemma 6.8.

Proof of Lemma 6.8: Let u be a node at level $\ell \in \{0, \ldots, \log_k n\}$ reached during the probing for a data piece d_i.

First, consider the case $\ell = \log_k n$. For the same reasons as in induction base in the proof of Lemma 6.9 u will hold the correct hash values after performing the procedure described above.

Next, assume $\ell = 0$. By Lemma 6.9, we know that the parent node of u on the probing path for d_i holds the correct hash values and therefor sends u's correct hash value to u.

In the following, let $\ell \in \{0, \ldots, \log_k n - 1\}$. Note that any hash value tuple sent by a node w to node u that is neither corrupted nor outdated passes the two tests performed by node u. Since only less than half of the nodes in $BF(u)$ are corrupted or outdated, with Chernoff bounds (Lemma 2.1) it follows that at least one node w chosen by u is neither corrupted nor outdated, w.h.p. Hence, the hash value tuple $(\tilde{g}(w), \tilde{g}(w^{(1)}, B), \ldots, \tilde{g}(w^{(k)}, B))$ node u receives from w passes both tests. Since we assume the Merkle Hash function $g(\cdot, B)$ to be a one-way hash function, it follows that $\tilde{g}(w^{(1)}, B), \ldots, \tilde{g}(w^{(k)}, B)$ are also correct. By Lemma 6.9, the hash chain from level $\log_k n$ up to level $\ell + 1$ is correctly stored in the according nodes which implies the claim. □

All in all, we get that at the end of the Probing Stage the following lemma holds.

Lemma 6.10. *For each request for a data item d that belongs to a level $\ell \in \{1, \ldots, \log_k n\}$ it holds: For at least $(5/8)c$ data pieces d_i of d the first $\log_k n - \ell + 1$ entries of the hash chain of d_i are correctly stored in the according butterfly nodes of MERKLE-BF(B).*

6.3.3 Recovery Stage

The Recovery Stage is dedicated to the further handling of the lookup requests that could not be served in the Probing Stage. In the Recovery Stage, corrupt, missing or outdated hash values and pieces of data items are recovered by exploiting the Butterfly Coding Strategy. Recall that each request that could not be served in the Probing Stage is defined to belong to a level $\ell \in \{1, \ldots, \log_k n\}$. The Recovery Stage is divided into $\log_k n$ phases, where in phase ℓ all requests belonging to level ℓ are handled, starting with $\ell = 1$. For this purpose phase ℓ is divided into two further substages: the Hash Chain Recovery Stage and the Data Recovery Stage.

Before we provide a detailed description of these two substages, we present the actions to perform in these substages on a high-level view. In the Hash Chain Recovery Stage, for each piece d_i of a data item d belonging to level ℓ the system tries to recover the remaining ℓ entries of the hash chain of d_i, such that at the end of this process with Lemma 6.10 for sufficiently many pieces d_i the complete hash chain of d_i is recovered. If the recovery of the remaining ℓ entries of the hash chain does not succeed for at least $c/4$ pieces of d, the request for d is said to belong to level $\ell + 1$ and is further handled in the next phase of the Recovery Stage. Otherwise, the request for d is further handled in the Data Recovery Stage. In the Data Recovery Stage, the system tries to

recover sufficiently many pieces of data items for which the complete hash chain has been recovered successfully in the previous Hash Chain Recovery Stage. If at least $c/4$ pieces of a data item d are successfully recovered, these pieces are used to recover the complete data item d using Reed-Solomon codes, which afterwards is verified using the stored values of the hash chain. If the verification succeeds, the request can be answered. If the verification fails or if only less than $c/4$ pieces of d were successfully recovered, the request is said to belong to level $\ell + 1$ and further handled in the next phase of the Recovery Stage.

6.3.3.1 Hash Chain Recovery Stage

At the beginning of the Hash Chain Recovery Stage, we need to perform some preprocessing operations. For this purpose, recall that for each request for a data item d that belongs to level ℓ by Definition 6.6 there are $(5/8)c$ pieces of d with requests that were active at level ℓ of the Probing Stage. That is, these requests did not fail at level ℓ or any level $\ell' > \ell$. Let $\mathcal{I}(d, \ell)$ be an arbitrary set of indices of $(5/8)c$ of these pieces. Next, for all data items d with a request at a server s and all indices of active pieces of d, i.e., for all $i \in \mathcal{I}(d, \ell)$, server s forwards a notification to the node on level ℓ of the probing path of data piece d_i. Each node u that receives such a notification is supposed to initiate the recovery of the remaining ℓ elements of the according hash chain. Recall that all information necessary for this recovery is held by the nodes in $UT(u)$. See Figure 6.5 for a visualization. On a high-level view, the recovery of the hash values of a data piece d_i consists of the following steps (details on these steps can be found below):

Step 1: The servers investigate whether it is possible to decode the hash values of d_i in parallel with the decoding of the other pieces of data blocks to be recovered without a node becoming congested. Here, we call a node **congested** if it receives more than βck messages for different data pieces for an arbitrary fixed constant $\beta > 5/2$. If this is not possible without a node becoming congested, this information is returned to server $s_i(d)$ (via forwarding top-down along the probing path) and the recovery process for d_i is aborted for this phase. Otherwise, the servers proceeds with the next step.

Step 2: The servers decode the remaining ℓ entries of the hash chain of d_i using the Butterfly Coding Strategy.

Step 3: The hash values along the probing path of d_i are verified. This is necessary because the decoding may restore false hash values. Just as in Step 1 in case the verification fails, this information is returned

to $s_i(d)$ (via forwarding top-down the probing path). Otherwise, the requested data piece (or the information that the data piece does not exist in the system, respectively) is returned to $s_i(d)$, which forwards this information to the server s that initially received the request for d.

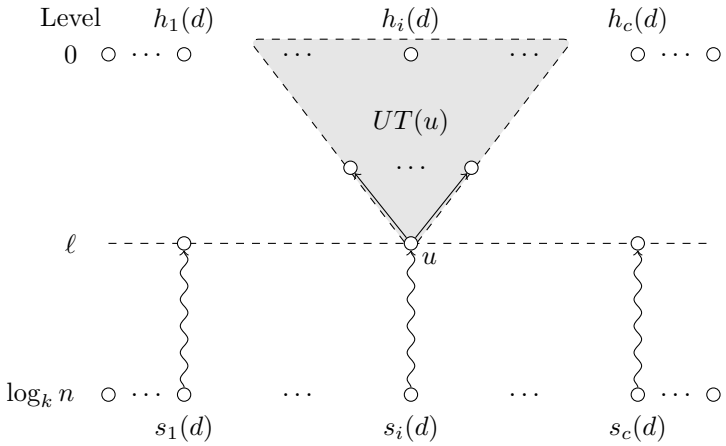

Figure 6.5: Visualization of the initial state in the Hash Chain Recovery Stage in phase ℓ with the curved paths denoting the probing paths of the data pieces $d_1, \ldots, d_c$.

At the end of this process each server s with a request for a data item d that belongs to level ℓ receives an answer for each of the $(5/8)c$ considered data pieces. If s discovers that more than $(3/4)c$ pieces of d cannot be recovered in the current stage, s classifies d to belong to level $\ell + 1$ and d will be considered in the next phase of the Recovery Stage again. Otherwise, s removes from $\mathcal{I}(d, \ell)$ all indices i for which d_i was deactivated during phase ℓ and proceeds to handle the request for d in the Data Recovery Stage.

At the end of phase ℓ of the hash chain recovery phase, for each request for a data item d that still belongs to level ℓ, it holds: $|\mathcal{I}(d, \ell)| \geqslant c/4$, and the complete hash chain for each d_i with $i \in \mathcal{I}(d, \ell)$ has been recovered correctly.

Note that each request for a piece of a data item that has successfully passed the Hash Chain Recovery Stage once in some phase ℓ can skip the Hash Chain Recovery Stage in all further phases $\ell' > \ell$ and directly enter the Data Recovery Stage every time.

Details on Step 1: Analogously to the congestion check in the Decoding Stage of IRIS and RoBuSt, Step 1 can simply be implemented by performing a broadcast in $BF(u)$ and letting the butterfly nodes thereby monitor their congestion and report it via a broadcast to the remaining nodes in case they get congested. Details on this can be found in Section 3.3.3. By the symmetry

of the k-ary butterfly the following lemma holds.

Lemma 6.11. *At the end of the congestion check in phase ℓ all nodes on level 0 in $BF(u)$ know whether $BF(u)$ contains a congested butterfly node or not.*

Details on Step 2: Let u be a node at level ℓ of the probing path of a data piece d_i with $i \in \mathcal{I}(d, \ell)$ for a data item d with a request belonging to level ℓ. In this step, the subbutterfly $BF(u)$ of MERKLE-BF(B) is decoded.

During the decoding of every k-block, whenever hash values in more than one node would need to be changed, we simply ignore the change and proceed in the next level without any changes performed. The reason for this is that if more than one of the nodes in a k-block would need to change its hash value due to a decoding step then there must be at least two nodes with false information participating in the current decoding step, in which case we want to abort the decoding process anyway.

Note that during the decoding, not only nodes emulated by corrupted servers may store false information: If there are more than two corrupted nodes in a k-block, their encoded values may be corrupted in such a way that the result of the decoding causes an intact node to change its (correct) values to wrong values. We call those servers **falsified**. Note that, since a falsification of intact nodes is possible, all nodes participating in the decoding of a subbutterfly in a phase ℓ must retain their previously stored values and restore their values at the beginning of phase $\ell + 1$. The decoding of a phase ℓ is finished after the last k-block at level 0 has been successfully decoded. Note that the decoded hash values may be incorrect (due to the influence of corrupted nodes in the decoding). Thus, the hash values still need to be verified, as is done in the next step.

Details on Step 3. The verification of the hash values of each piece d_i of a data item d is performed similarly to the verification in the Probing Stage. Recall that by Lemma 6.10 the hash chain of d_i from level 0 up to level ℓ has already been correctly recovered in the Probing Stage. Starting with node u at level ℓ, each node v on the probing path of d_i upwards to level 0 performs the following tests:

1. *Feasibility test:* v checks whether for the restored hash values of the children of v, i.e., for $g(v^{(1)}, B), \ldots, g(v^{(k)}, B)$, it holds: $g(g(v^{(1)}, B), \ldots, g(v^{(k)}, B)) = g(v, B)$.

2. *Parent test:* v checks whether its hash value $g(v, B)$ matches the one that its parent stores for v.

If any of the tests was not successful, the verification of all hash values of pieces of data items stored in $BF(v)$ is aborted (to achieve this, v broadcasts

this information in $UT(v)$). Furthermore, for each such piece d_i for which the verification at any level failed, the according server removes i from $\mathcal{I}(d, \ell)$. Finally, all requests for data items d belonging to level ℓ with $|\mathcal{I}(d, \ell)| < c/4$ are declared to belong to level $\ell + 1$. The requests for the remaining data items d with $|\mathcal{I}(d, \ell)| \geqslant c/4$ are handled immediately in the Data Recovery Stage.

6.3.3.2 Data Recovery Stage

The Data Recovery Stage is executed immediately after the Hash Chain Recovery Stage for each phase $\ell \in \{1, \ldots, \log_k n\}$. In phase $\ell \in \{1, \ldots, \log_k n\}$ all requests for data items that still belong to level ℓ are further handled. That is, all data pieces d_i, $i \in |\mathcal{I}(d, \ell)|$, of a data item d that has a request belonging to level ℓ are tried to be recovered and verified. Recall that due to the congestion check in the Hash Chain Recovery Stage for each request belonging to level ℓ it holds: For all data pieces d_i of d with $i \in |\mathcal{I}(d, \ell)|$ the ℓ-dimensional subbutterfly MERKLE-BF($u_{s,i}^{(\ell)}(d)$) can be completely decoded without causing any node to become congested. Just as in the previous chapters, $u_{s,i}^{(\ell)}(d)$ denotes the ℓ-th butterfly node on the probing path of d_i. Since MERKLE-BF($u_{s,i}^{(\ell)}(d)$) and DATA-BF($u_{s,i}^{(\ell)}(d)$) have the same structure, this implies that also DATA-BF($u_{s,i}^{(\ell)}(d)$) can be completely decoded without causing any node to become congested. Hence, in phase $\ell \in \{1, \ldots, \log_k n\}$ for each request for a data item d that belongs to level ℓ we decode all ℓ-dimensional subbutterflies DATA-BF($u_{s,i}^{(\ell)}(d)$) for each data piece d_i, $i \in |\mathcal{I}(d, \ell)|$. This process works analogously to the decoding process in the Hash Chain Recovery Stage.

After this decoding process, for all $i \in |\mathcal{I}(d, \ell)|$ the server that is supposed to hold d_i, verifies d_i by comparing $g(d_i, B)$ with the hash value assigned to butterfly node $u_{s,i}^{(0)}(d)$. In case these values equal, the data piece d_i is correct and will be forwarded to all servers that requested d_i by again using the technique of splitting and combining and routing through the k-ary butterfly. Otherwise, the data piece d_i is not correct and this information will also be forwarded in the same way to all servers that requested d_i. Each server $s_i(d)$ that received one of these answers sends this answer directly to the server s that initially received the request for data item d. If s receives at least $c/4$ pieces of d it uses Reed-Solomon codes in order to recover d and correctly answers the request. Otherwise, s denotes the request for d to belong to level $\ell + 1$, such that the request will again be handled in the next phase of the Data Recovery Stage.

As the following lemma states, the lookup protocol of OSIRIS requires only polylogarithmic congestion at each server and polylogarithmic time.

Lemma 6.12. *The Lookup Protocol of OSIRIS takes at most $O(\log_k^2 n)$ communication rounds with at most $O(\log^3 n)$ congestion at every server in each round.*

Proof. First, consider the Probing Stage. The Probing Stage basically consists of a bottom-up followed by a top-down traversal of the k-ary butterfly. For the verification of the hash values at each node u during the bottom-up traversal, node u needed to contact $O(\log n)$ nodes from the last level of $BF(u)$. For this purpose an additional bottom-up traversal of the butterfly is required, which is why the Probing Stage in total requires $O(\log_k^2 n)$ rounds. Recall that for the congestion check an upper bound of αc with $\alpha \geqslant 32$ constant is used. Hence, each node receives and sends at most $O(kc) = O(\log^2 n)$ messages in each round.

Next, consider the Recovery Stage which consists of $O(\log_k n)$ phases. For each phase the Hash Chain Recovery Stage and the Data Recovery Stage are executed subsequentially. The congestion check in the Hash Chain Recovery via a broadcast bottom-up through the k-ary butterfly requires $O(\log_k n)$ rounds. A possible decoding of the considered subbutterflies in the Hash Chain Recovery Stage and the Data Recovery Stage require further $O(\log_k n)$ rounds. Hence, in total the Recovery Stage consists of $O(\log_k^2 n)$ rounds. Since in the congestion check an upper bound of βck is used with $\beta > 5/2$ constant, no node sends or receives more than $O(ck^2) = O(\log^3 n)$ messages. □

6.4 Write Protocol

The write protocol of OSIRIS is responsible for processing the write requests. In principle, all data items are tried to be stored in the root bucket. Storing a set of data items in a single bucket works exactly as described in Section 5.3. Only if this would cause the root bucket to be overfull, i.e., to contain more than $2n$ data items, the system has to determine a set of data items that are going to be stored in a bucket in the next lower zone of the bucket tree. If this causes that bucket to be overfull, too, another bucket in the next zone has to be determined, and so on. For the sub-problem of determining suitable items and choosing the next bucket that is considered, we can use exactly the same process that is also used in RoBuSt (Section 5.3).

Recall that in each considered zone, at the beginning the considered bucket is completely decoded. While in RoBuSt we only needed to handle crashed servers, i.e., servers with failures we could simply detect, in OSIRIS we consider corrupted servers, i.e., servers with failures we cannot detect. Hence, instead of just recovering all data items stored in a bucket we additionally need to verify the recovered data items. For this purpose we make use of the previously described lookup protocol (Section 6.3). To be more precise, in order to recover all single data pieces stored in a bucket B, each server that is supposed to hold data piece d_1 of a data item d initiates a lookup request for d in bucket B. Since each bucket holds at most $2n$ data items which are distributed evenly

among the servers, w.h.p., we can process the initiated lookup requests using the lookup protocol described in Section 6.3. By this procedure, after at most a polylogarithmic number of rounds all data items stored in bucket B are correctly decoded and verified and the write protocol can proceed as described in Section 5.3.

Analogously to RoBuSt, we achieve a polylogarithmic upper bound on the number of communication rounds and the congestion at each server in each round.

Lemma 6.13. *The Write Protocol of OSIRIS takes at most $O(\log_k^2 n \log n)$ many communication rounds with at most $O(\log^3 n)$ congestion at every server in each round.*

6.5 Correctness Analysis of the Lookup Protocol

Just as in RoBuSt, we have to deal with outdated servers where a server s may become outdated whenever data is written in the system in a period t in which s was corrupted. In that case in any period $t' > t$ in which d has not been re-written, server s holds information about d that is not up-to-date without being aware of that. However, analogously to Lemma 5.14, one can show that whenever a data item is (re-)written in a bucket, not too many pieces of that data item are mapped to subbutterflies that contain too many corrupted servers.

Lemma 6.14. *Assume the adversary corrupts at most $\gamma \cdot n^{1/\log\log n}$ servers with $\gamma = 1/64$. Then, for any data item d that is (re-)written in a bucket, and any level $\ell \in \{0, \ldots, \log_k n\}$, at most $c/8$ pieces of d are mapped to subbutterflies $BF(v)$ (for some node v at level ℓ) that contain at least $\lceil 2^{\ell-2} \rceil$ corrupted servers, w.h.p.*

While in IRIS we required the hash functions $h_1, \ldots, h_c$ to form a $(c/4, 1/9)$-expander, in OSIRIS we require them to form a $(c/8, 1/64)$-expander. By Lemma 3.24, this requirement is satisfied if the hash functions $h_1, \ldots, h_c$ are chosen uniformly and independently at random with $c \geqslant 8 \log m$.

Just as in the analysis of the Lookup Protocol of IRIS and RoBuSt, we divide the correctness analysis of the Lookup Protocol into two parts: the analysis of the Probing Stage (Section 6.5.1) and the analysis of the Recovery Stage (Section 6.5.2).

6.5.1 Analysis of the Probing Stage

Analogously to IRIS and RoBuSt, for the Probing Stage of the Lookup Protocol of OSIRIS it can be shown that the number of requests belonging to a level exponentially decreases.

Lemma 6.15. *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/64$. Then, at the end of the Probing Stage of the Lookup Protocol of OSIRIS, the number of requests belonging to level $\ell \in \{1, \ldots, \log_k n\}$ is at most $2\gamma n/k^{\ell-1}$.*

The proof of Lemma 6.15 is similar to the proof of Lemma 5.15 in RoBuSt, although we need to be careful with the different conditions for a request to be aborted at a level. That is, while in RoBuSt a request for a data piece can only be aborted at level 0 by a node u if u was corrupted or outdated, in OSIRIS a request for a data piece d_i can only be aborted at level 0 if d_i cannot be verified. At levels $\ell > 0$ in RoBuSt a node can only abort a request if that node is congested. In OSIRIS a node can only abort a request at a level $\ell > 0$ if it is congested or if it cannot obtain the correct current hash value.

As for the proof of Lemma 5.15, we introduce the following notions.

Definition 6.16 (Congested/Unreliable/Outdated Subbutterfly)**.** *Let u be a butterfly node at level $\ell \in \{1, \ldots, \log_k n\}$. The subbutterfly $BF(u)$ is said to be*

- ***congested** at level ℓ if the servers in $BF(u)$ receive more than $\alpha c k^{\ell-1}/2$ probes for different data pieces in total when the requests are processed at level ℓ for an arbitrary but fixed constant $\alpha \geqslant 32$.*
- ***unreliable** at level ℓ if at least a quarter of the servers in $BF(u)$ are corrupted.*
- ***outdated** w.r.t. a bucket B if at least a quarter of the servers in $BF(u)$ do not store the latest version of bucket B.*

Definition 6.17 (Congested/Unreliable/Outdated Request)**.** *A request for a data item d is said to be*

- ***congested** at level ℓ if there exist pairwise different congested subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell - 1$, $r = c/8$.*
- ***unreliable** at level ℓ if there exist pairwise different unreliable subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell - 1$, $r = c/8$.*
- ***outdated** at level ℓ if there exist pairwise different subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $\ell_i \geqslant \ell - 1$, $r = c/8$ that are outdated w.r.t. d.*

With these notions we are ready to prove Lemma 6.15.

Proof of Lemma 6.15: Analogous to the proof of Lemma 5.15 we call a request for a data piece d_i of a data item d **aborted** at level $\ell \in \{0, \ldots, \log_k n\}$ if the probing for d_i did not successfully pass level ℓ. Depending on the level ℓ a request for a data piece d_i can be aborted at a node u for the following reasons:

- At level $\ell = 0$: The data piece d_i cannot be verified correctly.
- At level $\ell > 0$: Node u is congested or u could not obtain the correct current hash value.

By Definition 6.6, a request for a data item d belongs to level ℓ, if ℓ is the smallest level such that at least $(5/8)c$ requests for pieces of d have not been aborted at level ℓ or earlier. That is, a request for a data item d belongs to level ℓ if only less than $(3/8)c$ requests for pieces of d have been aborted at level ℓ or earlier and more than $(5/8)c$ requests for pieces of d have been aborted at any level $\ell' < \ell$.

In the following we upper bound the number of requests belonging to level ℓ by distinguishing between level 1 and all other levels $\ell > 1$.

Upper bound on the number of requests belonging to level $\ell = 1$: If a request for d belongs to level 1, then more than $(5/8)c$ requests for pieces of d have been aborted at level 0 and at least $(5/8)c$ requests for pieces of d have successfully passed level 1. Since a request for a data piece d_i can only be aborted at level 0 if d_i cannot be verified correctly, we have that for more than $(5/8)c$ data pieces d_i of d a nodes u at level 0 could not verify d_i.

For the following reasons the verification of a requested piece d_i may fail at a node u at level 0: (1) u does not hold the correct hash value for d_i, (2) u is corrupted, or (3) u is outdated w.r.t. bucket B.

Since any node at level 0 only asks its parent node on the probing path of the requested piece d_i for its hash value, with Lemma 6.9 it follows that whenever a node at level 0 is reached in the Probing Stage, then this node holds a correct hash value for d_i. Thus, the verification of a requested piece can only fail at node u at level 0, if u is corrupted or outdated w.r.t. bucket B.

Recall that if a request for a data item d belongs to level 1, then at least $(5/8)c$ requests for pieces of d have successfully passed level 1. Since $c/4$ verified pieces would have been sufficient to recover d, we know that more than $(5/8)c - (1/4)c = (3/8)c$ requests for pieces of d must have failed at level 0.

By Lemma 6.14, at most $c/8$ pieces of any data item are mapped to servers of a butterfly of dimension 1 that contains at least $\lceil 2^{\ell-2} \rceil = 1$ outdated server, w.h.p. In other words, at most $c/8$ data pieces are mapped to an outdated server, w.h.p. Hence, for each request belonging to level 1 the verification of at most $c/8$ pieces can fail due to corrupted servers, w.h.p.

Altogether, we get that for any request for a data item d belonging to level 1 the verification for more than $(3/8)c - (c/8) > c/8$ pieces of d must have failed at level 0 due to corrupted nodes, w.h.p. Hence, in order to upper bound the number of requests belonging to level 1, we need to upper bound the number

of requests for a data item d for which more than $c/8$ requests for pieces of d have failed at level 0 due to a corrupted node.

Let S be a maximum set of data items with requests that belong to level 1. We will show: $|S| < \gamma n$. We now construct a set F in the following way: For each $d \in S$, we choose $c/8$ indices i with the property that d_i is aborted at level 0 due to corrupted nodes and add these (d, i) to F. Note that F is a $c/8$-bundle of S. Since the adversary can corrupt at most $\gamma n^{1/\log\log n} < \gamma n$ servers, the number of servers covered by all $BF(u_{s,i}^{(0)}(d))$ with $(d, i) \in F$ is less than γn. Since $\Gamma_{F,0}(S)$ is exactly the set of these servers, it holds: $|\Gamma_{F,0}(S)| < \gamma n$. By Corollary 3.25, we get $|S| < \gamma n$, which completes the proof for level 1.

Upper bound on number of requests belonging to level $\ell > 1$:

Note that there are two reasons for why a piece of a data item can fail at a node u at level $\ell > 0$: Either because of congestion at u, or because u could not obtain the correct current hash values (both cases happen in Step 2). Analogous to Lemma 3.30 it can be shown that if a request for a piece d_i of a data item d fails at level $\ell \geqslant 1$ due to congestion, then $BF(u_{s,i}^{(\ell)}(d))$ is congested at level ℓ, w.h.p. Moreover, if a request for a piece d_i of a data item d fails at level $\ell \geqslant 1$ at a node u because u could not obtain the current hash values, then by Lemma 6.8 one of the following holds: At least a quarter of the servers in $BF(u)$ are corrupt, i.e., $BF(u)$ is unreliable, or at least a quarter of the servers in $BF(u)$ are outdated regarding the bucket d is in, i.e., $BF(u)$ is outdated w.r.t. d.

Furthermore, note that for a request for data item d that belongs to level a $\ell > 1$, by definition at least $(5/8)c$ pieces of d successfully passed level ℓ, but less than $(5/8)c$ requests for pieces of d passed level $\ell - 1$. Thus, we know that at least $(3/8)c$ requests for pieces of d have failed at level $\ell - 1$ or greater, implying that the request is congested, unreliable and/or outdated at level $\ell - 1$.

Thus, in order to prove the claim, we first show for each level ℓ, $1 < \ell \leqslant \log_k n$, that the number of requests that are congested at level $\ell - 1$ is upper bounded by $\gamma n/k^{\ell-1}$. Second, we prove that for each level $1 < \ell \leqslant \log_k n$ the number of requests that are unreliable at level $\ell - 1$ is upper bounded by $\gamma n/k^{\ell-1}$. Third, we show that there can be no outdated data items at any level. All in all, this yields that at most $2\gamma n/k^{\ell-1}$ requests can belong to any level $\ell > 1$.

For the congested requests, fix ℓ, $1 < \ell \leqslant \log_k n$. Let S be a maximum set of requests congested at level $\ell - 1$. We will show: $|S| < \gamma n/k^{\ell-1}$. As before, we construct a $(c/8)$-bundle F of S (adding, for each $d \in S$, $c/8$ indices i to F with the property that $BF(u_{s,i}^{(\ell_i)}(d))$ is congested). We first show that for α sufficiently large, less than a γ-fraction of all subbutterflies at level $\ell - 1$ can be congested. Recall that a subbutterfly on level $\ell - 1$ is congested if it receives

more than $\alpha c k^{\ell-1}/2$ probes for different (d, i) pairs. Let ε be the maximum fraction of servers that the adversary may block. Since there are at most n lookup requests in total, at most cn probes arrive at level $\ell - 1$. Thus, at most $cn/(\alpha c k^{\ell-1}/2) = 2n/(\alpha k^{\ell-1})$ subbutterflies can be congested at level $\ell - 1$. Since there are exactly $n/k^{\ell-1}$ disjoint subbutterflies at level $\ell - 1$, the fraction of congested subbutterflies at level $\ell - 1$ is upper bounded by $2/\alpha$. Hence, for $\alpha > 2/\gamma$, less than a γ-fraction of the subbutterflies on level $\ell - 1$ can be congested. That is, all of the congested subbutterflies $BF(u_{s,i}^{(\ell_i)}(d))$ with $(d, i) \in F$ together contain less than a γ-fraction of the nodes on level $\ell - 1$. This implies $|\Gamma_{F,\ell-1}(S)| < \gamma n$. By Corollary 3.25, this implies $|S| < \gamma n/k^{\ell-1}$.

For the unreliable requests, fix ℓ, $1 < \ell \leqslant \log_k n$. Let S be a maximum set of requests unreliable at level $\ell - 1$. We will show: $|S| < \gamma n/k^{\ell-1}$. Again, we construct a $(c/8)$-bundle F of S (by adding for each $d \in S$, $c/8$ indices i to F with the property that $BF(u_{s,i}^{(\ell_i)}(d))$ is unreliable). By definition, at least a quarter of the servers in any unreliable subbutterfly are corrupted. Therefore, if the adversary can corrupt only at most $\gamma n^{1/\log\log n}$ servers, then the number of servers covered by all $BF(u_{s,i}^{(\ell_i)}(d))$ with $(d, i) \in F$ must be less than γn for n sufficiently large. Since $\Gamma_{F,\ell-1}(S)$ is exactly the set of these servers, it holds: $|\Gamma_{F,\ell-1}(S)| < \gamma n$. By Corollary 3.25, this implies $|S| < \gamma n/k^{\ell-1}$.

For the outdated requests, by Lemma 6.14, only less than $c/8$ requests for pieces of each data item d can belong to outdated subbutterflies. Thus, no request can be outdated at any level by definition. This completes the proof. □

6.5.2 Analysis of the Recovery Stage

Analogously to IRIS and RoBuSt, for the Recovery Stage of the Lookup Protocol of OSIRIS, the number of requests belonging to a level exponentially decreases such that in the last phase of the Recovery Stage at most $O(k)$ requests have to be handled while all remaining requests have already been answered in the previous phases.

Lemma 6.18. *Assume an insider adversary crashes at most $\gamma n^{1/\log\log n}$ servers with $\gamma = 1/64$. Then, at the beginning of each subphase $\ell \in \{1, \ldots, \log_k n\}$ of the Recovery Stage of the Lookup Protocol of OSIRIS, the number of requests belonging to level ℓ is at most $\varphi n/k^\ell$ with $\varphi \leqslant 7\gamma k$.*

Again, before we prove Lemma 6.18, we introduce some notions. Note that here we use similar notation as for the proof of Lemma 6.15, but the meaning is slightly different.

Definition 6.19 (Congested/Unreliable Subbutterfly)**.** *Let u be a butterfly node at level $\ell \in \{1, \ldots, \log_k n\}$. Node u is called **congested**, if it receives more than $\beta c k^\ell$ messages for different data pieces. The subbutterfly $BF(u)$ is called*

- ***congested*** *at level ℓ if the congestion check in phase ℓ of the Hash Chain Recovery Stage determines that there is a congested butterfly node in $BF(u)$.*
- ***unreliable*** *at level ℓ if at least $2^{\ell-2}$ servers from $BF(u)$ are corrupted.*

Definition 6.20 (Congested/Unreliable Request)**.** *A request for a data item d is called*

- ***congested*** *at level ℓ if there exist pairwise different congested subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $r = c/8$ and $\ell_j \geqslant \ell$.*
- ***unreliable*** *at level $\ell \in \{1, \ldots, \log_k n\}$ if there exist pairwise different unreliable subbutterflies $BF(u_{s,i_1}^{(\ell_1)}(d)), \ldots, BF(u_{s,i_r}^{(\ell_r)}(d))$ with $r = c/8$ and $\ell_j \geqslant \ell$.*

In the proof of Lemma 6.18 we will use the fact that if not too many nodes in a subbutterfly $BF(u)$ are congested, corrupted, or outdated, then all data blocks stored in $BF(u)$ can be recovered.

Lemma 6.21. *For any phase ℓ of the Recovery Stage, any butterfly node u at level ℓ and any bucket B, if $BF(u)$ is not congested and if less than $2^{\ell-1}$ servers in $BF(u)$ are corrupted or outdated, then all data blocks from B stored at nodes at level 0 in $BF(u)$ can be recovered correctly.*

Proof. In the following, fix a phase ℓ of the Recovery Stage and a node u at level ℓ of the butterfly and assume that $BF(u)$ is not congested.

Recall that the decoding of a single k-block may falsify intact servers, which can happen only if at least two of the servers in the according k-block are corrupted or outdated and these servers then cause an intact server to overwrite correct data. In the following we call a node corrupted/outdated/falsified if that node is emulated by a corrupted/outdated/falsified server.

Furthermore, recall that the decoding of a subbutterfly $BF(u)$ in phase ℓ of the Recovery Stage is initiated from level ℓ, meaning that there cannot be any falsified nodes at level ℓ. For a node v at level $\ell - r$ of the butterfly, we say $T(v)$ is a **witness tree** of v if T is a complete binary subtree of $LT(v)$ of depth r consisting of corrupted, outdated or falsified nodes only.

The first step of our proof is to show: If v stores incorrect data and v's data cannot be restored during the recovery at level ℓ, then there exists a witness tree $T(v)$ of v. We show this claim by induction on $\ell - r \in \{0, \ldots, \ell - 1\}$ beginning with $\ell - 1$. For the base case, let v be a node at level $\ell - 1$ in the butterfly that is corrupted, outdated, or falsified. Then, by Lemma 6.3, v has at least two children w_1, w_2 in $LT(v)$ that store incorrect data. Then, the tree induced by v, w_1 and w_2 makes up a complete binary subtree of depth 1 that consists only of corrupted/outdated/falsified nodes.

For the induction step, let v be a node at level $\ell - r$, $r > 1$ in the butterfly that is corrupted, outdated, or falsified. Then, again by Lemma 6.3, at least two children w_1, w_2 of v in $LT(v)$ must be corrupted, outdated, or falsified. By the induction hypothesis, there exist witness tress $T(w_1)$ and $T(w_2)$ of depth $r-1$ of w_1 and w_2. The tree induced by connecting v to the roots of $T(w_1)$ and $T(w_2)$ makes up a witness tree for v.

Next, assume for contradiction that for an arbitrary node u at level ℓ of the butterfly and an arbitrary bucket B that $BF(u)$ cannot be recovered correctly regarding B. That is, there is a node u at level 0 with a data block that cannot be recovered during the decoding. Thus, if $BF(u)$ cannot be recovered, then with our previous considerations there exists a witness tree $T(v)$ of a node v at level 0 of $BF(u)$. That is, $T(v)$ is a complete binary tree of depth ℓ that consists of corrupted, outdated and falsified nodes only. Since a complete binary tree of depth ℓ has 2^ℓ leaves, at least 2^ℓ of the nodes in $T(v)$ must be corrupted, falsified, or outdated. Since at most half of these nodes may have been falsified, we know that $T(v)$ must consist of at least $2^{\ell-1}$ corrupted or outdated nodes. Hence, according to the assumption there cannot exist a witness tree for a node at level 0 of $BF(u)$, implying that $BF(u)$ can be recovered. □

We are now ready to prove Lemma 6.18.

Proof of Lemma 6.18*:* We prove the lemma by induction on ℓ. The basis $\ell = 1$ holds by Lemma 6.15. For the induction step, let $\ell \in \{1, \ldots, \log_k n - 1\}$ and assume that the induction hypothesis holds for level ℓ. We show that the number of data items that cannot be recovered in phase ℓ (and will thus be propagated to level $\ell + 1$) is at most $5\gamma n/k^\ell$. Together with Lemma 6.15, this means that at the beginning of phase $\ell + 1$, at most $2\gamma n/k^\ell + 5\gamma n/k^\ell$ data items belong to level $\ell + 1$, which equals $\varphi n/k^{\ell+1}$ and thus proves the induction step.

First, note that a request for data item d cannot be answered after phase ℓ if and only if less than $c/4$ pieces of d could be recovered. Since for each data item d with a request belonging to level ℓ, requests for $(5/8)c$ pieces of d are initiated, we know that d cannot be recovered if and only if more than $(5/8)c - (1/4)c = (3/8)c$ requests for pieces of d failed.

Next, recall why a request for a piece of d can fail. For that purpose, let u be the node at level ℓ on the probing path of a data piece d_i. The request for d_i can fail due to one (or both) of the following reasons:

1. Data piece d_i cannot be decoded without causing excessive congestion. In this case, $BF(u)$ is congested according to Definition 6.19.

2. A verification failed which is due to exactly one of the following reasons:

a) A hash value along the hash chain of d_i could not be verified in the Hash Chain Recovery Stage.

b) The verification of d_i in the Data Recovery Stage failed.

If the first case does not hold, i.e., if $BF(u)$ is not congested, then in this case by Lemma 6.21 there are at least $2^{\ell-1}$ corrupted or outdated servers in $BF(u)$ (regarding the bucket that d belongs to).

In the following we show that the requests that are neither congested nor unreliable can be answered correctly in phase ℓ, implying that these requests will not be propagated to phase $\ell+1$. Afterwards, we show that the number of congested requests is upper bounded by $\gamma n/k^\ell$ and the number of unreliable requests is upper bounded by $4\gamma n/k^\ell$. Altogether this implies that in total at most $5\gamma n/k^\ell$ requests belong to level $\ell+1$.

First, consider a request for a data item d that is neither congested nor unreliable. For this request only less than $c/8$ subbutterflies are congested at level ℓ and only less than $c/8$ subbutterflies are unreliable at level ℓ, i.e., only less than $c/8$ subbutterflies contain at least $2^{\ell-2}$ corrupted servers. By Lemma 6.14, we know that at most $c/8$ pieces of each data item are mapped to a subbutterfly that contains at least $2^{\ell-2}$ servers, i.e., at most $c/8$ pieces of each data item are mapped to an unreliable subbutterfly. This implies that all remaining pieces of d (which are at least $(5/8)c - 3 \cdot (c/8) = c/4$ many) have less than $2^{\ell-2}$ corrupted servers in their subbutterflies at level ℓ and less than $2^{\ell-2}$ outdated servers in their subbutterflies at level ℓ, i.e., less than $2^{\ell-1}$ corrupted or outdated servers, and these subbutterflies are not congested. Thus, by Lemma 6.21, these at least $c/4$ pieces can be recovered. Since these $c/4$ pieces are sufficient for the recovery of the data item d using Reed-Solomon codes, these kind of requests, i.e., the requests that are neither congested nor unreliable, can correctly be served.

Next, we upper bound the number of congested and unreliable requests.

Upper bound on the number of congested requests: For the upper bound on the number of congested requests, recall that we call a subbutterfly $BF(v)$ of a node v congested if there is a server in $BF(v)$ that would have to handle more than βck requests for different pieces of data items. For $\beta > 5/2$, it holds that $\beta ck > 5\varphi c/(8\gamma)$, which implies that a congested subbutterfly $BF(v)$ of a node v receives more than $5\varphi c/(8\gamma)$ requests for different pieces of data items. By the induction hypothesis and due to the fact that we send $(5/8)c$ requests per data item, there are at most $(5/8)c \cdot \varphi n/k^\ell$ requests in total, which means that there are less than $\varphi n/k^\ell \cdot (5/8)c \cdot 8\gamma/(5c\varphi) = \gamma n/k^\ell$ congested subbutterflies of dimension ℓ. Let S be a set of data items with a request congested at level ℓ. Recall that a request for a data item d is congested at level ℓ if there exist at least $r = c/8$ subbutterflies $BF(u^{(\ell)}_{s,i_1}(d)), \ldots, BF(u^{(\ell)}_{s,i_r}(d))$ with $i_1, \ldots, i_r$ being

pairwise different that are congested. For each $d \in S$, let $d_{i_1}, \ldots, d_{i_r}$ be $c/8$ such indices fulfilling this property and define $F = \{(d, i_1), \ldots, (d, i_r) \mid d \in S\}$. Then, F is a $c/8$-bundle of S. Since, as we have shown, there are less than $\gamma n/k^\ell$ congested subbutterflies of dimension ℓ and since each subbutterfly of dimension ℓ contains k^ℓ nodes, $|\Gamma_{F,\ell}(S)| < \gamma n$. By Corollary 3.25, this implies $|S| < \gamma n/k^\ell$.

Upper bound on the number of unreliable requests: In the following define $\sigma = 4\gamma$. Let S be a maximum set of requests for a data item that are unreliable at level ℓ. We will show: $|S| < \sigma n/k^\ell$. Recall that a data item d is unreliable at level ℓ if there exist at least $r = c/8$ subbutterflies $BF(u^{(\ell)}_{s,i_1}(d)), \ldots, BF(u^{(\ell)}_{s,i_r}(d))$ with $i_1, \ldots, i_r$ being pairwise different that are unreliable, i.e., each of them contains at least $2^{\ell_i - 2}$ corrupted servers. For each $d \in S$, let $d_{i_1}, \ldots, d_{i_r}$ be $c/8$ such indices fulfilling this property and define $F = \{(d, i_1), \ldots, (d, i_r) \mid d \in S\}$. Then, F is a $c/8$-bundle of S. Since a subbutterfly of level ℓ' contains $k^{\ell'}$ servers in total, and since an unreliable subbutterfly of level ℓ' contains at least $2^{\ell'-2}$ corrupted nodes, a $2^{\ell'-2}/k^{\ell'}$ fraction of the servers of an unreliable subbutterfly of level ℓ' are corrupt, which is at least $2^{\log_k n - 2}/n$ for any ℓ', $1 \leqslant \ell' \leqslant \log_k n$. Therefore, if the adversary can corrupt at most $\gamma \cdot n^{1/\log\log n} = \sigma \cdot 2^{\log_k n - 2}$ servers, then the number of servers covered by all $BF(u^{(\ell)}_{s,i}(d))$ with $(d, i) \in F$ must be less than σn. Since $\Gamma_{F,\ell}(S)$ is exactly the set of these servers, it holds: $|\Gamma_{F,\ell}(S)| < \sigma n$. Since $\sigma \cdot 8 = 4\gamma \cdot 8 = 1/2$, Corollary 3.25 can be applied, which implies $|S| < \sigma n/k^\ell = 4\gamma n/k^\ell$. □

Note that by the bounds specified in Lemma 6.18, during the recovery in the last phase, i.e., phase $\log_k n$, no request can fail due to congestion. By the same arguments as used in the proof of Lemma 6.18, all data items can be recovered in the decoding. Thus, the proof of Lemma 6.18 also finishes the proof of the correctness of the lookup protocol.

Chapter 7

Conclusion and Outlook

We conclude this work by recapping the properties of the systems presented in the previous chapters and thereby highlighting the main ideas of their construction (Section 7.1). Finally, in Section 7.2, we discuss some remaining open questions in this field.

7.1 Conclusion

We presented the first distributed information and storage systems that are provably correct against insider adversaries, i.e., adversaries that know everything about the system and may use this knowledge in order to attack a large fraction of the servers. We began this sequel of distributed information and storage systems with the development of a distributed information system called Basic IRIS, which is provably correct against insider adversaries (c.f. Chapter 3). In particular, Basic IRIS correctly serves any set of lookup requests (with at most $O(1)$ requests per intact server) with polylogarithmic work and time despite the existence of an insider adversary that may crash up to $O(n^{1/\log\log n})$ servers. Nevertheless, Basic IRIS requires only a constant storage redundancy. The main new idea of Basic IRIS was the development of a distributed coding strategy, called k-ary butterfly coding, which hierarchically interweaves and encodes data blocks with each other such that for each data block stored in the system each server holds the data block itself or at least holds some coding information related to that block. This technique allowed us to tolerate an insider adversary that crashes asymptotically significantly more than a polylogarithmic number of servers while still only requiring at most a constant redundancy and polylogarithmic time and work for serving

lookup requests. The k-ary butterfly coding strategy was used as a building block in all systems presented in this work.

In Chapter 4 we presented Enhanced IRIS, which is a distributed information system that enhanced Basic IRIS in such a way that it tolerates an insider adversary that even crashes up to a constant fraction of all servers while asymptotically still achieving the same efficiency results regarding time and work needed for correctly serving lookup requests. This enhancement happens at the expense of the redundancy which increased by a logarithmic factor to $O(\log n)$. The main idea behind the enhancement was a restructuring of the underlying topology used for the hierarchical coding strategy via using permutations that fulfill certain expansion properties.

Building on the k-ary coding strategy and the lookup protocol developed for IRIS we developed a distributed storage system called RoBuSt with similar properties as Basic IRIS (c.f. Chapter 5). That is, RoBuSt provably correctly serves any set of lookup and write requests (with at most $O(1)$ requests per server) with polylogarithmic time and work despite the existence of an insider adversary that crashes up to $O(n^{1/\log\log n})$ servers. Thereby, RoBuSt requires only a logarithmic redundancy. For this purpose, we needed to develop a more advanced storage strategy than the one used in IRIS. In IRIS the data items stored in the system were partitioned into sets of size $\Theta(n)$ and each of these sets was separately encoded with each other via the k-ary butterfly coding strategy. When trying to serve $\Theta(n)$ write requests without changing the underlying storage strategy, it may happen that the data items to write are encoded with each other in $O(n)$ different sets. But this would cause all these $O(n)$ sets to be completely re-encoded which is too expensive. Hence, we needed to come up with a new and more involved strategy for partitioning the data items to store in the system into sets of size $\Theta(n)$, which we called buckets. This strategy also prescribes how to arrange these buckets efficiently such that at no time more than a polylogarithmic number of buckets is accessed for writing.

Finally, we strengthened the adversary considered even further (c.f. Chapter 6). While for the previous systems we restricted the adversary to crash servers, he may now corrupt the servers' storage. To be more precise, we assumed an insider adversary that may corrupt the storage of a large fraction of the servers, but he may neither corrupt the main memory of the servers nor their protocols. While the detection of crashed servers can easily be implemented, this is not so easy possible for corrupted servers, in particular this is hardly possible for adversarial corruptions. Instead, we needed to find a way how to cope with the existence of corrupted data items and still guarantee to correctly serve all lookup and write requests. For this purpose, we interweaved techniques from the field of authenticated data structures, namely Merkle hash trees, with the techniques we developed for IRIS and RoBuSt. By this, the

lookup protocol became significantly more complicated, since the servers always additionally need to reconstruct hash chains for the data pieces they are supposed to find. All in all, we developed OSIRIS, a distributed storage system that provably correctly serves any set of lookup and write requests (with at most $O(1)$ requests per server) despite the existence of an insider adversary that may corrupt the storage of up to $O(n^{1/\log\log n})$ server. Thereby, serving $O(n)$ lookup and write requests requires polylogarithmic time and work, while at the same time we only require a logarithmic redundancy.

7.2 Outlook

In the setting considered for the construction of OSIRIS, we restricted ourselves to somehow "semi-Byzantine" adversaries. That is, the adversary was allowed to corrupt data stored at the servers, but it was not allowed to corrupt the main memory or even the protocols of the servers. In contrast, a Byzantine adversary is allowed to completely take over the attacked servers and mimic an arbitrary behavior. A common approach to deal with Byzantine adversaries in distributed storage systems is the usage of so-called quorums [Kin+11]. A quorum is an accumulation of several servers such that the majority of the servers in each quorum are not Byzantine. Hence, by majority decision each quorum can agree on a common value. By this, if for each server in the system a quorum can be built, the quorum can take over the decisions of the server. In order for this approach to work, while still adhering to our desired efficiency bounds, we require a protocol that solves the Byzantine agreement problem with polylogarithmic time and work despite the existence of $O(n^{1/\log\log n})$ Byzantine servers. However, existing protocols for Byzantine agreement and building balanced quorums (i.e., quorums of size $O(\log n)$, where each server is contained in at most $O(\log n)$ quorums) in at most polylogarithmic time require $O(\sqrt{n})$ bits of communication that each server sends and processes [KS11; Kin+11], which is significantly more than we allow.

Even if we were able to build balanced quorums with polylogarithmic time and work only, in order to tolerate an adaptive adversary that takes over a large fraction of the servers, it still remains to figure out how to reset a server to its last state before he was attacked, once the adversary decides to "release" the server in order to take over another server. Recall that in Chapter 6 we assumed that a server who was attacked by the adversary for some period of time rejoins the system with the state it had before the attack. That is, a previously attacked server is out of date with respect to the buckets that were written during the time it was attacked. In the context of DNS spoofing attacks this is a reasonable assumption, because in that case the storage of the attacked server actually is not modified, but instead its IP entry in the DNS system has

been modified such that it might refer to another storage server that may hold arbitrary data. But if we leave this scenario and instead consider the problem of Byzantine servers, i.e., servers that may be taken over and also be released again by a malicious insider adversary, we need to provide a mechanism on how to reset a released server to the last state it had before it was attacked. However, when assuming an adversary that may completely take over a large fraction of the servers, there are several problems with the implementation of such a rollback mechanism. First, neither the non-attacked servers know which servers are Byzantine, nor can the servers that have been attacked and are released again can efficiently detect and repair that. That is, an attacked server does not notice when it is released again and therefore it does not know when to perform a rollback. The second problem is that even if a server knew when to perform a rollback, it is not aware of the point in time it became attacked and therefore it does not know up to which point it needs to perform a rollback. One possible way to provide such a rollback mechanism would be to implement some kind of version control system that uses hashes for the various states in order to detect corruptions. Via a binary search over the chain of states' hashes, the servers could then determine to which state to revert to. However, a problem of this approach is that the time needed for finding the state to revert to is only guaranteed to be logarithmic in the overall running time of the system, but not necessarily logarithmic in the number of servers. Even if all these problems regarding a rollback mechanism would have been solved, we still require the existence a secure storage location in order to store previous states or the history of changes with their corresponding hash values which we actually wanted to get rid of.

One restriction of our systems regarding its practical usage is the assumption of a batch based adversary. That is, time was divided into periods (each consisting of sufficiently many rounds) and the adversary had to choose the set of the attacked servers at the beginning of each period. In case of crash failures, as considered for Basic IRIS, Enhanced IRIS and RoBuSt, this requirement was necessary in order be able to determine representatives for the crashed servers. In the case of corrupted servers, as considered for OSIRIS, the restriction of the adversary to be batch based could be relieved and instead we could allow the adversary to also corrupt the permanent storage of servers during the periods as long as the maximum number of corrupted servers allowed is not exceeded. However, "released" servers still need to be detected and their storage state still needs to be reverted to the last point in time before they were corrupted. Additionally, after releasing a server the adversary must not corrupt another server instead until the rollback of the storage state of the released server has completely been finished.

A further interesting question is the realizability of our systems in an asynchronous setting. In any case, a straightforward transferability is not possible

due to several problems that may arise. One of the biggest difficulties in this context is that in a synchronous setting, we could always exploit the property that all servers have a common understanding of the current "state" of the protocol. In particular, at any time all servers performed computations associated with the same stage. This assumption does not hold anymore in an asynchronous setting. Especially in the preprocessing stage during the determination of representatives, this may cause problems as in this phase the servers still need to rebuild a new k-ary butterfly consisting only of intact servers. In order to ensure a correct proceeding of the further protocols, it is of great importance that all intact servers achieved a common understanding of the representatives. One way to avoid these problems is the usage of local synchronizers together with a timeout even though in that case slow servers may erroneously be assumed as crashed servers.

As already mentioned in Chapter 1, this work constitutes a proof of concept of distributed information and storage systems with the desired properties. In particular, a practical implementation of the developed systems is out of the scope of this work. However, it is an exciting question in how far our systems can be realized in practice and how they perform. It is not straightforward to actually implement our storage strategies and protocols which is, for instance, due to the difficulties regarding the synchronicity. However, instead, in a student work, a simulator of Basic IRIS working in a synchronous setting has been implemented with the goal of measuring the efficiency of Basic IRIS. The most noticeable observations in the measurements of the simulation was the effort (regarding the number of messages sent) spent in the different stages. The effort spent for the Decoding Stage significantly exceeded the one of the Preprocessing and Probing Stage. This can be explained by the very expensive broadcasts that have to be executed in each phase of the Decoding Stage. Furthermore, we examined in how far the functionality of Basic IRIS is still guaranteed when increasing the number of allowed crashed servers significantly more than tolerated by our analysis. For that purpose, we chose the servers to be crashed and the data items to be requested randomly. It turned out that even when considering an order of magnitude of more crashed servers as specified by our upper bounds, Basic IRIS still correctly answers almost all requests. This is not that surprising, since in the analysis of this work we always assumed an insider adversary, implying worst-case failures of the servers instead of random failures as considered in the simulations. However, the congestion caused in the Decoding Stage turned out to be the limiting factor of the system. All in all, one of the most challenging questions that need to be answered in order to implement our systems for practical usage, is whether and how the communication effort required in the lookup protocols (in particular in the Decoding Stage) can be reduced.

Bibliography

[Abe01] Karl Aberer. "P-Grid: A Self-Organizing Access Structure for P2P Information Systems". In: *Proceedings of the 9th International Conference on Cooperative Information Systems*. London, UK, UK: Springer Berlin Heidelberg, 2001, pp. 179–194. URL: http://dl.acm.org/citation.cfm?id=646747.701489.

[Ama] Amazon. *AWS*. URL: https://aws.amazon.com/ (visited on 11/24/2015).

[Ama08] Amazon. *Amazon S3 Availability Event*. 2008-07-20. URL: http://status.aws.amazon.com/s3-20080720.html (visited on 11/24/2015).

[And+14] Elli Androulaki, Christian Cachin, Dan Dobre, and Marko Vukolic. "Erasure-Coded Byzantine Storage with Separate Metadata". In: *CoRR* abs/1402.4958 (2014). DOI: 10.1007/978-3-319-14472-6_6.

[And03] David G. Andersen. "Mayday: Distributed Filtering for Internet Services". In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*. Vol. 4. Seattle, WA: USENIX Association, 2003, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1251460.1251463.

[App] Apple, Inc. *iCloud*. URL: http://www.apple.com/icloud/ (visited on 11/24/2015).

[AS06] Baruch Awerbuch and Christian Scheideler. "Towards a Scalable and Robust DHT". In: *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Cambridge, Massachusetts, USA: ACM, 2006, pp. 318–327. DOI: 10.1145/1148109.1148163.

[AS07] Baruch Awerbuch and Christian Scheideler. "A Denial-of-Service Resistant DHT". In: *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*. Ed. by Andrzej Pelc. Vol. 4731. Springer Berlin Heidelberg, 2007, pp. 33–47. DOI: 10.1007/978-3-540-75142-7_6.

[Aug+13] John Augustine, Anisur Rahaman Molla, Ehab Morsy, Gopal Pandurangan, Peter Robinson, and Eli Upfal. "Storage and Search in Dynamic Peer-to-peer Networks". In: *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA: ACM, 2013, pp. 53–62. DOI: 10.1145/2486159.2486170.

[Awe07] Baruch Awerbuch. "Towards Scalable and Robust Overlay Networks". In: *International Workshop on Peer-to-Peer Systems (IPTPS)*. 2007-02.

[BH82] Allan Borodin and John E. Hopcroft. "Routing, Merging and Sorting on Parallel Models of Computation". In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC)*. New York, NY, USA: ACM, 1982, pp. 338–344. DOI: 10.1145/800070.802209.

[Bit] BitTorrent, Inc. *BitTorrent*. URL: http://www.bittorrent.com (visited on 11/24/2015).

[Bla+95] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures". In: *IEEE Transactions on Computers* 44.2 (1995-02), pp. 192–202. DOI: 10.1109/12.364531.

[Bon+03] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. *The Zettabyte File System*. Tech. rep. 2003.

[BSS09] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. "A DoS-resilient Information System for Dynamic Data Management". In: *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architecture (SPAA)*. Calgary, AB, Canada: ACM, 2009, pp. 300–309. DOI: 10.1145/1583991.1584064.

[CDV13] Christian Cachin, Dan Dobre, and Marko Vukolic. "BFT Storage with 2t+1 Data Replicas". In: *CoRR* abs/1305.4868 (2013). URL: http://arxiv.org/abs/1305.4868.

[Che52] Herman Chernoff. "A measure of asymptotic efficiency for tests of a hypothesis based on the sums of observations". In: *Annals of Mathematical Statistics* 23 (1952), pp. 493–507. DOI: 10.1214/aoms/1177729330.

[CT05a] Christian Cachin and Stefano Tessaro. "Asynchronous Verifiable Information Dispersal". English. In: *Distributed Computing*. Ed. by Pierre Fraigniaud. Vol. 3724. Springer Berlin Heidelberg, 2005, pp. 503–504. DOI: 10.1007/11561927_42.

[CT05b] Christian Cachin and Stefano Tessaro. "Optimal Resilience for Erasure-Coded Byzantine Distributed Storage". English. In: *Distributed Computing*. Ed. by Pierre Fraigniaud. Vol. 3724. Springer Berlin Heidelberg, 2005, pp. 497–498. DOI: 10.1007/11561927_39.

[DM93] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. "Simple, Efficient Shared Memory Simulations". In: *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1993-07, pp. 110–119. DOI: 10.1145/165231.165246.

[ES13] Martina Eikel and Christian Scheideler. "IRIS: A Robust Information System Against Insider DoS-Attacks". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2013, pp. 119–129. DOI: 10.1145/2486159.2486186.

[ES15] Martina Eikel and Christian Scheideler. "IRIS: A Robust Information System Against Insider DoS Attacks". In: *ACM Transactions on Parallel Computing* 2.3 (2015-11), 18:1–18:33. DOI: 10.1145/2809806.

[ESS14] Martina Eikel, Christian Scheideler, and Alexander Setzer. "RoBuSt: A Crash-Failure-Resistant Distributed Storage System". In: *Proceedings of the 18th International Conference on the Principles of Distributed Systems (OPODIS)*. Vol. 8878. Springer Berlin Heidelberg, 2014, pp. 107–122. DOI: 10.1007/978-3-319-14472-6_8.

[FSY05] Amos Fiat, Jared Saia, and Maxwell Young. "Making chord robust to byzantine attacks". In: *Proceedings of the 11th European Symposium on Algorithms (ESA)*. Springer Berlin Heidelberg, 2005, pp. 803–814. DOI: 10.1007/11561071_71.

[Goo] Google. *Google Cloud Platform*. URL: https://cloud.google.com/storage/ (visited on 11/24/2015).

[Goo+04] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. "Efficient Byzantine-tolerant erasure-coded storage". In: *International Conference on Dependable Systems and Networks*. 2004-06, pp. 135–144. DOI: 10.1109/DSN.2004.1311884.

[GT01] Michael T. Goodrich and Roberto Tamassia. *Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing*. Tech. rep. Technical report, Johns Hopkins Information Security Institute, 2001.

[GTH02] MichaelT. Goodrich, Roberto Tamassia, and Jasminka Hasić. "An Efficient Dynamic and Distributed Cryptographic Accumulator*". English. In: *Information Security*. Ed. by AgnesHui Chan and Virgil Gligor. Vol. 2433. Springer Berlin Heidelberg, 2002, pp. 372–388. DOI: 10.1007/3-540-45811-5_29.

[Ham50] Richard W. Hamming. "Error Detecting and Error Correcting Codes". In: *Bell System Technical Journal* 26.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[Har+02] Nicholas J.A. Harvey, John Dunagan, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. *SkipNet: A Scalable Overlay Network with Practical Locality Properties*. Tech. rep. MSR-TR-2002-92. Note: The technical report previously numbered MSR-TR-2002-92, "A Polynomial-time Rescaling Algorithm for Solving Linear Programs," is now numbered MSR-TR-2003-09. Microsoft Research, 2002-12, p. 38. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=69993.

[HGR07] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. "Low-overhead Byzantine Fault-tolerant Storage". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. Stevenson, Washington, USA: ACM, 2007, pp. 73–86. DOI: 10.1145/1294261.1294269.

[HX05] Cheng Huang and Lihao Xu. "STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures". In: *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*. Vol. 4. San Francisco, CA: USENIX Association, 2005, pp. 15–15. DOI: 10.1109/TC.2007.70830.

[IBM15] IBM Security Intelligence Staff. *IBM 2015 Cyber Security Intelligence Index*. Tech. rep. IBM Security, 2015-06. URL: http://www-03.ibm.com/security/data-breach/2015-cyber-security-index.html.

[Int] Interplanetary Networks. *InterPlanetary File System (IPFS)*. URL: https://ipfs.io/ (visited on 11/24/2015).

[Kan+05] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. "Botz-4-sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI)*. Berkeley, CA, USA: USENIX Association, 2005, pp. 287–300. URL: http://dl.acm.org/citation.cfm?id=1251203.1251224.

[Kar+97] 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 29th Annual ACM Symposium on Theory of Computing (STOC)*. El Paso, Texas, USA: ACM, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[Kin+11] Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. "Load Balanced Scalable Byzantine Agreement Through Quorum Building, with Full Information". In: *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN)*. Bangalore, India: Springer Berlin Heidelberg, 2011, pp. 203–214. DOI: 10.1007/978-3-642-17679-1_18.

[KK03] M. Frans Kaashoek and David R. Karger. "Koorde: A Simple Degree-Optimal Distributed Hash Table". English. In: *Peer-to-Peer Systems II*. Ed. by M.Frans Kaashoek and Ion Stoica. Vol. 2735. Springer Berlin Heidelberg, 2003, pp. 98–107. DOI: 10.1007/978-3-540-45172-3_9.

[KMR02] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. "SOS: Secure Overlay Services". In: *Proceedings of the 2002nd Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. Pittsburgh, Pennsylvania, USA: ACM, 2002, pp. 61–72. DOI: 10.1145/633025.633032.

[KS11] Valerie King and Jared Saia. "Breaking the O(N2) Bit Barrier: Scalable Byzantine Agreement with an Adaptive Adversary". In: *Journal of the ACM (JACM)* 58.4 (2011-07), 18:1–18:24. DOI: 10.1145/1989727.1989732.

[KSW10] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. "Towards worst-case churn resistant peer-to-peer systems". English. In: *Distributed Computing* 22.4 (2010), pp. 249–267. DOI: 10.1007/s00446-010-0099-z.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982-07), pp. 382–401. DOI: 10.1145/357172.357176.

[LYL08] Xin Liu, Xiaowei Yang, and Yanbin Lu. "To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-node Botnets". In: *ACM SIGCOMM*. 2008, pp. 195–206.

[LYX10] Xin Liu, Xiaowei Yang, and Yong Xia. "NetFence: Preventing Internet Denial of Service from Inside out". In: *SIGCOMM Computer Communication Review* 40.4 (2010-08), pp. 255–266. DOI: 10.1145/1851275.1851214.

[MAD02] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. "Minimal Byzantine Storage". In: *Proceedings of the 16th International Conference on Distributed Computing (DISC)*. London, UK, UK: Springer-Verlag, 2002, pp. 311–325. URL: http://dl.acm.org/citation.cfm?id=645959.676126.

[Mer79] Ralph C. Merkle. "Secrecy, Authentication and Public Key Systems". PhD thesis. Stanford University, 1979-06.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. "Viceroy: A Scalable and Dynamic Emulation of the Butterfly". In: *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC)*. Monterey, California: ACM, 2002, pp. 183–192. DOI: 10.1145/571825.571857.

[MR97] Dahlia Malkhi and Michael Reiter. "Byzantine Quorum Systems". In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*. El Paso, Texas, USA: ACM, 1997, pp. 569–578. DOI: 10.1145/258533.258650.

[MV84] Kurt Mehlhorn and Uzi Vishkin. "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories". English. In: *Acta Informatica* 21.4 (1984), pp. 339–374. DOI: 10.1007/BF00264615.

[Nak09] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2009-05). URL: http://www.bitcoin.org/bitcoin.pdf.

[Net15] Arbor Networks. *Worldwide Infrastructure Security Report*. Tech. rep. IBM Security, 2015-06. URL: http://www-03.ibm.com/security/data-breach/2015-cyber-security-index.html.

[NN98] Moni Naor and Kobbi Nissim. "Certificate Revocation and Certificate Update". In: *Proceedings of the 7th Conference on USENIX Security Symposium*. Vol. 7. San Antonio, Texas: USENIX Association, 1998, pp. 17–17. DOI: 10.1109/49.839932.

[NW03] Moni Naor and Udi Wieder. "Novel Architectures for P2P Applications: The Continuous-discrete Approach". In: *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. San Diego, California, USA: ACM, 2003, pp. 50–59. DOI: 10.1145/777412.777421.

[PR04] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo Hashing". In: *Journal of Algorithms* 51.2 (2004-05), pp. 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

[PS02] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. "The Case for Cooperative Networking". In: *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*. London, UK, UK: Springer Berlin Heidelberg, 2002, pp. 178–190. URL: http://dl.acm.org/citation.cfm?id=646334.758993.

[PTT09] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. "Cryptographic Accumulators for Authenticated Hash Tables". In: *IACR Cryptology ePrint Archive* (2009).

[Pug90] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Communications of the ACM* 33.6 (1990-06), pp. 668–676. DOI: 10.1145/78973.78977.

[Rat+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. "A Scalable Content-addressable Network". In: *Proceedings of the 2001th Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. San Diego, California, USA: ACM, 2001, pp. 161–172. DOI: 10.1145/383059.383072.

[RD01] Antony I. T. Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*. London, UK, UK: Springer Berlin Heidelberg, 2001, pp. 329–350. URL: http://dl.acm.org/citation.cfm?id=646591.697650.

[RL05] Rodrigo Rodrigues and Barbara Liskov. "High Availability in DHTs: Erasure Coding vs. Replication". English. In: *Peer-to-Peer Systems IV*. Ed. by Miguel Castro and Robbert van Renesse. Vol. 3640. Springer

Berlin Heidelberg, 2005, pp. 226–239. DOI: 10.1007/11558989_21.

[RS60] Irving Reed and Golomb Solomon. "Polynomial codes over certain finite fields". In: *Journal of the Society of Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304. DOI: 10.1137/0108018.

[SMB02] Tyron Stading, Petros Maniatis, and Mary Baker. "Peer-to-Peer Caching Schemes to Address Flash Crowds". In: *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*. London, UK, UK: Springer Berlin Heidelberg, 2002, pp. 203–213. DOI: 10.1007/3-540-45748-8_19.

[SRS02] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. "A Lightweight, Robust P2P System to Handle Flash Crowds". In: *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 226–235. DOI: 10.1109/JSAC.2003.818778.

[Sta+05] Angelos Stavrou, Debra L. Cook, William G. Morein, Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. "WebSOS: An Overlay-based System For Protecting Web Servers From Denial of Service Attacks". In: *Journal of Communication Networks* 48.5 (2005-08), pp. 781–807. DOI: 10.1016%2Fj.comnet.2005.01.005.

[Sto+01] 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 2001th Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. San Diego, California, USA: ACM, 2001, pp. 149–160. DOI: 10.1145/383059.383071.

[Tam03] Roberto Tamassia. "Authenticated Data Structures". English. In: *Proceedings of the 11th European Symposium on Algorithms (ESA)*. Ed. by Giuseppe Di Battista and Uri Zwick. Vol. 2832. Springer Berlin Heidelberg, 2003, pp. 2–5. DOI: 10.1007/978-3-540-39658-1_2.

[Tor] Linus Torvalds. *Git*. URL: https://git-scm.com/ (visited on 11/24/2015).

[TT05] Roberto Tamassia and Nikos Triandopoulos. *Efficient Content Authentication over Distributed Hash Tables*. Tech. rep. 2005.

[TT07] Roberto Tamassia and Nikos Triandopoulos. "Efficient Content Authentication in Peer-to-Peer Networks". In: *Proceedings of the 5th International Conference on Applied Cryptography and Network Security*. Zhuhai, China: Springer Berlin Heidelberg, 2007, pp. 354–372. DOI: 10.1007/978-3-540-72738-5_23.

[TWB13] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. "Zigzag Codes: MDS Array Codes With Optimal Rebuilding." In: *IEEE Transactions on Information Theory* 59.3 (2013), pp. 1597–1616. DOI: 10.1109/TIT.2012.2227110.

[Val82] Leslie G. Valiant. "A Scheme for Fast Parallel Communication". In: *SIAM Journal on Computing* 11.2 (1982), pp. 350–361. DOI: 10.1137/0211027.

[WK02] Hakim Weatherspoon and John D. Kubiatowicz. "Erasure Coding vs. Replication: A Quantitative Comparison". In: *In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*. 2002, pp. 328–338. DOI: 10.1007/3-540-45748-8_31.

[XB99] Lihao Xu and J. Bruck. "X-code: MDS array codes with optimal encoding". In: *IEEE Transactions on Information Theory* 45.1 (1999-01), pp. 272–276. DOI: 10.1109/18.746809.

[Xu+06] Lihao Xu, Vasken Bohossian, Jehoshua Bruck, and David G. Wagner. "Low-density MDS Codes and Factors of Complete Graphs". In: *IEEE Transactions on Information Theory* 45.6 (2006-09), pp. 1817–1826. DOI: 10.1109/18.782102.

[YWA05] Xiaowei Yang, David Wetherall, and Thomas Anderson. "A DoS-limiting Network Architecture". In: *Proceedings of the 2005th Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. Philadelphia, Pennsylvania, USA: ACM, 2005, pp. 241–252. DOI: 10.1145/1080091.1080120.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. Tech. rep. Berkeley, CA, USA, 2001.