Dynamic Load Balancing in Peer-to-Peer Networks

Dissertation

by

Miroslaw Korzeniowski

Faculty for Electrical Engineering, Computer Science and Mathematics
Department of Computer Science and Heinz Nixdorf Institute
University of Paderborn, Germany

Paderborn, November 2005

Reviewers:

- Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn, Germany
- Prof. Dr. Burkhard Monien, University of Paderborn, Germany

Acknowledgments

I am grateful to my advisor, Friedhelm Meyer auf der Heide, who let me work at my own pace, steering the course of my work only slightly whenever it was needed. I really appreciate his trust and the fact that my work was always evaluated by its results and not by the time I spent in the office.

Christian Scheideler can be thought of as an unofficial co-advisor of this thesis. Cooperation with him significantly accelerated my progress and increased my will to work and hope of success in the field of theoretical computer science.

I also would like to thank the former and current members or close friends of the group "Algorithms and Complexity" not only for scientific support, but mainly for being so friendly and helpful in everyday life. Above all I would like to thank André Brinkmann, Marcin Bieńkowski, Valentina Damerow, Mirek Dynia, Gereon Frahling, Jarek Kutyłowski, Peter Mahlmann, Harald Räcke, Kay Salzwedel, Christian Schindelhauer, Christian Sohler, and Martin Ziegler.

Special thanks I owe to Mirek Dynia and Jarek Kutyłowski who proof-read parts of this thesis several times. I would also like to especially thank Marcin Bieńkowski who was a co-author of the papers on which this thesis if based.

Being abroad is not easy even if the atmosphere at work is so friendly, and this is why the piece of Poland we created here was so important for me. I would like to mention the most important polish friends I had here, especially the ones who have been here during the last several months of writing this thesis: Ania, Asia, Beata, Ewa, Iwonka, Jarek, Marcin, Mirek, Ola, and Radek.

Contents

CHAPTER 1

Introduction

In the last few years Peer-to-Peer networks have become very popular both in everyday internet use and academic community. The term "Peer-to-Peer" literally means "equal to equal" and stands for computers building a network on such a basis in opposition to a *server based* approach, in which there is a central server to which all participants connect and which provides services.

In a Peer-to-Peer network, all participants act as clients, servers and routers at the same time. One of the advantages of such networks is their high *robustness*. They have to be designed in such a way that even unexpected failure of a large fraction of peers should not switch off the system as a whole. Through replication even loss of data can be avoided up to some extent. Another important advantage is the *scalability* of the network. All of the participants contribute their resources such as bandwidth, storage capacity and computational power. Thus, when more nodes arrive and the demand for the resources increases, their availability increases, too. If the system is designed with care, it can grow beyond the numbers that could be attended to by a central server. Yet another advantage that we would like to mention is the *lack of infrastructure*. Dedicated servers are usually expensive both in the initial cost and maintenance. Peer-to-Peer systems are meant to be self organizing and the necessary resources are contributed by the users themselves, so both the initial and maintenance cost are minimal.

The most popular application of Peer-to-Peer systems is *file sharing*. A user of such a system can publish a file that he owns and from the moment of publication all other users can download this file. The Peer-to-Peer structure is usually used only for the purposes of publication and finding of files. The transfer of the found file between two users is performed using a direct connection. Spreading of new releases of linux distributions is an example of a useful application of file sharing. Regrettably, such applications are used mainly for distribution of copyright protected files such as music or movies. This thesis, however, omits the legal and moral issues related to the Peer-to-Peer networks and concentrates on the algorithmic aspects of Peer-to-Peer networks as distributed systems.

The system that is considered the precursor of Peer-to-Peer networks was *Napster* which appeared in 1997. In fact, the system was based on a central server which was used for indexing the published files. Each participant could connect to the server and publish his own file or find a file previously published by someone else. After receiving an information about the publisher, the seeker could connect to the publisher and download the file without any further mediation from the server. Such an approach is called a *hybrid approach*, as it contains the elements of both server-based and Peer-to-Peer networks.

A hybrid approach deprives the system of one of its most important advantages, namely the lack of a single point of failure. A successful attack on the server can break the whole system and even if the environment is not hostile, the speed of the server bounds the speed of the whole system. *Gnutella* was a pure Peer-to-Peer system meant to deal with this problem. It connected the computers in a random fashion and if a peer wanted to find a file it flooded its neighborhood of bounded diameter. Such approach, however, has two other disadvantages. One of them is, that flooding is a very inefficient communication method already in networks of middle size. The other is that even if an item that we are looking for exists somewhere in the network, the bounded flooding approach is not guaranteed to find it. It favors the popular files, neglecting the rare ones.

This is where the algorithmic approach comes into the scene. In 1999 four designs appeared almost at the same time: Chord [SMK+01], CAN [RFH+01], Tapestry [HKRZ02] and Pastry [RD01]. They were all based on *consistent hashing*, which works as follows. There are two components of the approach: the *keyspace partitioning* and the *overlay network*. The former means that the approach is based on a virtual space (the d-dimensional cube $[0,1]^d$ for CAN and the interval $[0,1)$, where 0 and 1 are glued together to form a ring, for all other approaches) into which all items are hashed and which is partitioned among the participants called nodes. Each node chooses a point in the space and is responsible for the piece of the space lying around it. For example, each point of the space can be attributed to the closest node. If an item is to be published, a global hash function h is computed on its name yielding a point in the space. The information about the item is stored on the node responsible for the point. When a node wants to find an item, it computes the hash value on its name and contacts the node responsible for this value. It can do it using *key based routing*, a greedy scheme that trying to go from a node s to a point t, jumps to the neighbor of s that is closest to t. The network is built in such a way that the greedy scheme works efficiently, i.e. needs $O(d \cdot n^{1/d})$ hops in CAN and $O(\log n)$ hops in other approaches, where n is the number of nodes present in the network.

1.1. Notation and tools

Throughout the thesis we use the following terms, techniques, and tools.

First of all, we use two terms to denote the participants of the network. When we want to stress, that we mean a computer participating in the network, we call it a **peer**. If we do not talk about the computer but rather about the abstraction that represents it in the virtual space, we call it a **node**.

We use the following terms to talk about important routing parameters. For a network and a system of paths defined between each pair of nodes, **Dilation** (denoted as D) is the length of the longest of these paths. It can obviously be as high as the diameter of the network, but as it depends not only on the network but also on the system of the chosen paths, it can be even higher. Another routing parameter, **Congestion** (denoted as C) is defined for a set of messages, each of which has a source and a destination and travels between them along a fixed path. It bounds from above the number of messages that go through a single node (sometimes one defines separately edge- and node-congestion; in this thesis, however, we use only the node-congestion). The time needed to perform a routing task is $\Omega(D + C)$ and using for example the protocol from [MV99], routing can be finished in time $O(D + C + \log n)$.

We need two important terms to talk about distributions of random variables. The first one is well established in the algorithm design community. The second one is introduced for clarity of notation.

We say that a random variable $X(X_1, \ldots, X_n)$ being a function of n variables (these can be for example n results of throwing a dice) is **bounded by $O(f(n))$, with high probability**, if for any $l \geq 0$ there exists c such that $\Pr[X \geq c \cdot f(n)] \leq \frac{1}{n^l}$. This means that we can play with the constant in the O-notation in order to obtain any polynomially low failure probability. The same definition with adequate changes applies to the Ω and Θ notations. Sometimes we use the abbreviation **w.h.p.**

We say that a random variable $X(X_1, \ldots, X_n)$ being a function of n variables is **bounded by $O(f(n))$, with constant probability**, if for any ϵ there exists c such that $\Pr[X \geq c \cdot f(n)] \leq \epsilon$. This means that we can play with the constant in the O-notation in order to obtain arbitrarily small but constant failure probability. The same definition with adequate changes applies to the Ω and Θ notations. Sometimes we use the abbreviation **w.c.p.**

A large part of this thesis is based on the **Balls into Bins** games. The randomized game is very simple: we throw n balls into m bins uniformly at random, each one independently from others. Then, one can examine the probability of different events like at least one bin being empty, or distributions like the number of balls in the fullest bin. We use three important tools that emerge in the context of balls into bins games. The first one is the **Birthday Paradox**, the second one is **Negative Dependence** and the last one is the **Paradigm of Two Random Choices**. The birthday paradox examines the event that at least one bin contains two balls in the end in the case when $m = \Theta(n^2)$. The negative dependence [DR98] result proves among others such intuitive results as Pr[1st bin non-empty] $\geq$ Pr[1st bin non-empty|2nd bin non-empty]. The latter is clear, since if there is a ball in the second bin, there are fewer balls that could fill the first one. However, the strict proof of this and more general results is not straightforward. The paradigm of two random choices consideres the balls-into-bins game with two modifications. One of them is that balls are thrown one by one and another one is that at least two bins are chosen for each ball when it should be thrown into a bin it is thrown to the one that is least full at the moment of throwing. This paradigm shows that in a game modified in such a way the load of the most loaded bin is much lower in the end.

In order to show that our results are concentrated, that is that they almost always run in the time we claim they do and that they almost always yield the results we claim they do, we use the following version of **Chernoff bounds**, which are proved for example in [Sch00]. Let $X = \sum_{i=1}^{n} X_i$ be a sum of n independent indicator (taking only values 0 or 1) random variables $X_1, \ldots, X_n$. Then for any $0 \leq \delta \leq 1$, $\Pr[X < (1-\delta) \cdot \mathrm{E}[X]] \leq \exp(-\frac{\mathrm{E}[X] \cdot \delta^2}{2})$ and for any $\delta \geq 0$ $\Pr[X > (1+\delta) \cdot \mathrm{E}[X]] \leq \exp(-\frac{\mathrm{E}[X] \cdot \min\{\delta, \delta^2\}}{3})$. We use Chernoff bounds mostly in case when $X = \Omega(\log n)$, which yields that $X = \Theta(\mathrm{E}[X])$ with high probability.

1.2. Chord and Distance Halving

We elaborate on Chord [SMK+01] and an approach by Naor and Wieder [NW03b, NW03a, Wie05] called *Distance Halving* or the *Continuous-Discrete Approach*, as these are the main topologies on which we base.

Chord is based on a ring of integers in range $0 \ldots 2^b - 1$, which means that both the addresses of the peers and of the stored items are b-bits numbers. The parameter b is sufficiently large, so that the space can contain any reasonable number of nodes and items. For each point in space its predecessor and successor are defined in a natural way. The predecessor of a point is the node with the largest ID smaller than the point, and if such a node does not exist, the node with the largest ID. The successor of a point is the node with the smallest ID larger than the point, and if such a node does not exist, the node with the smallest ID. Each point on the ring is attributed to its successor, which means that all items hashed to this point are stored on its successor and all links from other nodes pointing to the point are incoming edges of the successor.

We identify the nodes with their positions on the ring. The nodes on the ring form an approximation of hypercubic connections, that is, each node v has an edge to $\mathrm{succ}((v+2^i) \bmod 2^b)$ for each $0 \leq i < b$. Thanks to these edges, any point in the network can be reached in $O(b)$ hops using the following greedy routing. When a node v wants to route a message to a point a it finds the largest i such that $(v + 2^i) \mod 2^b$ lies between v

and a and sends the message to $\mathrm{succ}((v + 2^i) \mod 2^b)$. The intuition behind it is that in each step the distance between v and a halves. The authors of Chord prove also, that if there are n nodes present in the system, routing needs at most $O(\log n)$ hops, which is important for small n (much smaller than 2^b). The degree of each node depends on the placement of nodes in the space and is $O(\log n)$ in expectation and $O(\log^2 n)$, with high probability, if nodes are placed uniformly at random. The first part of this thesis shows extensions of the original Chord that assure (among other improvements) that the degree in the network is logarithmic, with high probability. The second part of this thesis uses a construction similar to Chord in order to efficiently perform rotations in a balanced binary tree.

Even if the degree of the network is logarithmic, it is not optimal for logarithmic distance. For an arbitrary network and an arbitrary node r the number of nodes in distance at most i from r can be bounded by $\frac{d^{i+1}-1}{d-1}$, where d is the degree of the network. If the network has n nodes, then at least one of them has to be in distance $\Omega(\log_d n)$ from r. We say that a network of degree d together with a routing scheme is degree-dilation optimal if its dilation is at most $O(\log_d n)$. The design by Naor and Wieder is an example of a degree-dilation optimal scheme. The construction starts from defining the connections for a continuous graph consisting of all points in the interval $[0, 1)$. A point $x \in [0, 1)$ is connected to $\frac{x}{d}, \frac{x+1}{d}, \ldots,$ and $\frac{x+d-1}{d}$, which is a construction derived from the de Bruijn graph. The continuous construction is then transformed into a discrete construction as follows. Each node v is responsible for the interval $I_v = [v, \mathrm{succ}(v))$, that is, the interval from itself to its successor. In the discrete version there is an edge between nodes v and u, if and only if there exist $x \in I_v$ and $y \in I_u$ such that there is an edge between x and y in the continuous version.

The "routing" procedure is where the approach takes its name from. If a message is to be routed from a node v to a place t, two messages travel virtually at the same time from v and t. At each step the messages are routed along the edge $x \to \frac{x+i}{d}$, with the same random choice of i for both of them. It is proved that with each such step the distance between the two messages is shortened by a factor d (halved for $d = 2$, hence "distance halving"). Since the smallest distance at which the messages can be, not being on the same node, is at least inversely polynomial in the number of servers, after $O(\log_d n)$ steps the messages meet.

The implementation of the scheme first sends the message along $\Omega(\log_d n)$ edges, each chosen randomly from the d possibilities and all random choices are stored in the message in form of a stack. Then, the message is moved slightly within the interval it is currently in and sent backwards using the same types of edges it used before but in inverse order. The distortion is such that in the end it lands exactly in t.

Similarly as with Chord, also in case of the Continuous-Discrete Approach one needs the network to be balanced in order to prove some properties of the network (e. g.

constant degree for constant *d*). For this, one needs one of our balancing schemes from the first part of this thesis or a similar algorithm. The second part of this thesis uses the de Bruijn graph to simulate parent-to-child edges of a binary tree.

1.3. Half-life

In order to bound the communication cost of one of our algorithms we will need the following notion from [LNBK02]. This is introduced, because the system cannot be treated as a process that starts and ends but rather as something that runs forever. The notion of half-life lets us measure the communication not as the total bandwidth used, but rather as rate at which each node has to communicate.

Definition 1.1 ([LNBK02]). *If there are n live nodes at time t, then:*

1. *the doubling time from time t is the time that elapses before n additional nodes arrive*
2. *the halving time from time t is the time required for half of the nodes alive at time t to depart*
3. *the half-life τ_t from time t is the smaller of the doubling and halving times from time t*
4. *the half-life τ of the entire system is the minimum half-life over all times t*

We cited the whole definition but in our approach we only need a notion for half-life from arbitrary time t, namely τ_t. In Chapter 5 we will analyze the behavior of a load balancing algorithm during a time interval starting in a time step t and ending after τ_t, that is lasting a half-life. We will also use the term half-life for naming such time interval. The notion of half-life is especially useful because it has been proved that in each half-life each node should get some number of messages in order for the network not to fall apart. We elaborate on this below.

Since the system is dynamic and not supervised, some communication is needed in order to keep the network connected. We assume the simplest model of handling arrivals and departures. At any time a node can notify another node about its presence in the network, incurring a communication cost of one message. We allow to send notifications even if the two nodes are not connected. However, the goal of sending notifications is to keep the system balanced using minimal number of messages. We cite the following theorem which bounds from below the communication occurring in the system.

Theorem 1.2 ([LNBK02]). *Any n-node Peer-to-Peer network that remains connected with high probability for any sequence of joins and leaves with half-life τ must notify every node with an average of $\Omega(\log n)$ messages per τ time.*

In the proof a specific scenario is considered. Nodes join the system according to Poisson process with rate λ and stay in the system according to an exponential distribution with rate parameter μ. Each node which gets on average fewer than k messages, loses all its neighbors with probability $\left(1 - \frac{1}{e-1}\right)^k$, so $\Omega(\log n)$ messages are necessary for the node to have at least one connection to another node, with high probability.

If we prove, that our load balancing algorithm generates communication of $O(\log^2 n)$ per node per half-life, we may say, that its communication cost is amortized against the network-maintenance and that it is $O(\log n)$ competitive.

1.4. Our contribution

As mentioned above, Chord and the Continuous-Discrete approach are both based on a ring into which nodes are inserted and each node is responsible for some part of the ring, for example the interval from itself to its direct successor. In case when nodes choose the places for themselves uniformly at random (and in a truly distributed environment this seems the best choice) the sizes of the pieces of the space they are responsible for vary in size. This causes overloading of some nodes not only with larger numbers of items they have to store but also with more communication, as a node responsible for a large piece of the space has to keep connections with many nodes responsible for smaller chunks. The first part of this thesis presents algorithms that balance such load and guarantee that the final ratio between the most and the least loaded nodes is constant.

First of all, we show two distributed algorithms which yield constant approximation of the number of participants of the network. Such approximation is needed in both load balancing algorithms to decide about the role a peer should play in the network. Namely, they make it possible for a node to decide if its load is too high or too low.

The first load balancing algorithm, AlgStat , works in a static environment and starts with any distribution of nodes on the ring. It is able to migrate some nodes, so that in the end each node is responsible for an interval of the same length (up to a constant factor). The whole routine takes only constant number of rounds, where each round lasts as long as it takes to deliver a message between any two peers. The price for such low reaction time is high communication cost. Specifically, linear number of nodes are in a process of sending a message at each point in time.

The second load balancing algorithm, AlgDyn , also starts with an arbitrary distribution of nodes on the ring. It is meant to run in a distributed environment and the emphasis is put on low communication cost. We prove that in a half-life, each node sends at most logarithmic number of messages, and the total communication cost in the network is at most by a logarithmic factor larger than the cost of maintaining the system. Unfortunately, the reaction of the scheme is slower, that is, it takes logarithmic number of rounds to repair the system and only the imbalance present in the beginning of the half-life is repaired. The imbalance introduced in the present half-life is going to be repaired in the next time period.

In the second part of this thesis, we introduce a class of structured memory models that we call *transparent memory models*. Transparent memory models are memory models in which memory accesses can be emulated in a scalable overlay network with constant work. Peer-to-Peer networks are perfect candidates as a support for transparent data structures.

As a specific example of a transparent data structure, we introduce a transparent memory model called the *hypertree memory model* and design a scalable, dynamic overlay network that can emulate memory accesses in this model with constant work. The hypertree is a family of binary trees in which each node has additional shortcut edges leading to all its "cousins". The overlay network design we propose is a hybrid of the Continuous-Discrete approach and Chord.

We show how to implement a binary search tree in the hypertree memory model with the property that the amortized work for insert, delete and search operations is asymptotically the same as for the best binary search trees in the pointer model, yet it can efficiently recover *all* remaining information under *arbitrary* memory faults. Moreover, in the emulation, every node in the dynamic overlay network only has to perform a worst-case logarithmic amount of work for any insert, delete or search operation. The recovery operations work on a reactive basis, that is, they generate communication cost only if failures occur and the total cost is only by a logarithmic factor larger than the number of failures.

1.5. Bibliographic notes

The results presented in this thesis have all been accepted for publication in various international conference proceedings. The static scenario of the first part of the thesis was presented at the 4th International Workshop on Peer-to-Peer Systems (IPTPS 2005) [BKM05]. The dynamic one was presented at the 9th International Conference on Principles of Distributed Systems (OPODIS 2005) [BK05]. The result from the second part of the thesis was presented at the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2005) [KS05].

Part I.

Load Balancing in Distributed Hash Tables

Chapter 2

Introduction to Load Balancing

In the first part of this thesis we tackle the problem of balancing the load in a Peer-to-Peer network. We present algorithms which assure that each node is responsible for a piece of the virtual space of the same size up to a constant factor. This should guarantee that the responsibilities for maintaining the network are distributed in a fair way and that no node is overloaded.

The Peer-to-Peer networks which we are considering are based on consistent hashing [KLL+97] with ring topology like Chord [SMK+01], Tapestry [HKRZ02], Pastry [RD01], and the Continuous-Discrete Approach [NW03a, NW03b]. The exact structure of the topology is not relevant. It is, however, important that each peer has direct links to its successor and predecessor on the ring and that there is a routine that lets any peer contact the peer responsible for any given point in the network in time D, which depends on n, the current number of peers present in the system. In the systems, that we are interested in, $D \in O(\log n)$.

In the load balancing algorithms we assume synchronous network behavior. This means that all nodes work with approximately the same speed and that messages are guaranteed to reach their destinations in bounded time. In Chapter 4 we assume additionally that it is possible to synchronize each node with its successor and predecessor, so that the part of code after such synchronization is run only if the part before it has been finished by all nodes. Thus, we achieve the execution of the algorithm divided into synchronized rounds.

Below, we introduce notation for the division of the $[0, 1)$ ring among n nodes participating in the network, length of interval for which a node is responsible and a crucial parameter of a network defined in this way, namely *smoothness*.

Definition 2.1. *A placement of n nodes on the unit ring* generates *n intervals – one from each node to its direct successor. A node i is* responsible *for the interval I_i from itself to its successor. For a node i, the* absolute length *of its interval I_i is denoted as $l(I_i)$.*

Definition 2.2. *The* smoothness *of a network is the ratio of the length of the longest interval to the length of the shortest interval.*

The smoothness is a parameter, which informs about three aspects of load balance.

- Storage load of a peer. The longer its interval is, the more data has to be stored in the peer. If we assure, that the smoothness is constant, then in a system in which $\Omega(n \cdot \log n)$ items are distributed uniformly at random on the ring, each of the n peers is responsible for the same number of items (up to constant factors), with high probability.
- Degree of a node. A node responsible for a longer interval has a higher probability of being contacted by many nodes responsible for short intervals which increases its in-degree.
- Congestion and dilation. In the Continuous-Discrete Approach [NW03a, NW03b], constant smoothness is assumed in order to prove that routing parameters like congestion and dilation are kept low.

The algorithms we describe rearrange the positions of the nodes on the ring so that the smoothness becomes constant, independently of the original situation. In the process of designing the algorithms, we noticed that it is very hard if not impossible to devise a truly distributed and local scheme which would on the one hand react instantly to system changes and on the other hand would not send unnecessary messages. In our designs, we propose two tradeoffs. The first algorithm reacts to all changes as quickly as possible but has very high communication cost. The second one adjusts its communication to the dynamics of the system but its reactions to the system changes are a little bit slower. Both algorithms need to know the current number of nodes present in the system at least with constant accuracy, so we start the description from showing two algorithms which yield constant approximation of the number of network participants.

2.1. Our results and organization of the first part

In the rest of this chapter we shortly describe the work that others did in the area of load balancing in Peer-to-Peer networks. It has been a very popular topic in the last few years and about a dozen results concerning it have been published. We also show a result that is considered folklore but the concrete statement and proof of which are hard to find. Namely, we show that, if n nodes are placed on the ring independently and uniformly at random, the smoothness is $\Omega(n \cdot \log n)$, with constant probability. This means that the most obvious solution, i.e. choosing IDs of nodes randomly, yields too high differences in load.

In Chapter 3 we show how to approximate the number of nodes present in the network. The nodes work independently in a truly distributed fashion and each of them is able to compute for itself a constant approximation of the current number n of nodes. We present two schemes. One is due to Awerbuch and Scheideler [AS04a] and the other is our independent design. Our algorithm needs an upper bound of $\log n$ as its input. The correctness of our approximation and load balancing schemes depends on the correctness of the bound, whereas their communication cost is proportional to the value of the bound. Choosing this parameter to be 40, we err in the estimation at most by a factor of 4 for networks of reasonable size (from 1000 to 2^{40} nodes). Either of the two approximation schemes can be used for the two load balancing algorithms we show in the subsequent chapters.

In Chapter 4 we show a self-stabilizing distributed algorithm that rebalances the system in the static case, that is when no nodes join or leave it. Such a strong assumption is needed in the analysis, but the algorithm itself is designed to work in a dynamic environment. The main advantage of the scheme is that it reacts to any changes immediately and needs only logarithmic time to repair the system from an arbitrarily bad situation. Its main drawback is its very high communication cost. Even if the system is perfectly balanced, each node sends a message directly after its previous messsage has been delivered, so at any point in time there are linear number of unnecessary messages traveling through the system.

In Section 4.3 we present the results of the experimental evaluation that we performed. As the constants emerging from the theoretical analysis are in the range of thousands, it may seem that our algorithms are useless in practical applications. Our experiments show that with proper selection of parameters, smoothness can be reduced from thousands to 10 or below.

In Chapter 5 we extend and modify the scheme so that it can be analyzed as an ever-running process. We show formally that in a half-life each node sends $O(\log n)$ messages to logarithmic distance, which yields total neighbor-to-neighbor communication cost of $O(\log^2 n)$ per node. The algorithm can play against an oblivious adversary but needs some time to catch up with the dynamics of the system. The latter is modelled in the following way. An adaptive adversary chooses a point in time in which the assessment of the algorithm starts. In a half-life starting in this point in time an oblivious adversary generates join and leave operations. We show that our algorithm smooths the imbalance present at the beginning of the considered half-life. The imbalance generated by the adversary during the half-life is going to be balanced later. The delay of the reaction to system changes is the price for the bounded communication cost.

2.2. Other work on load balancing

Load balancing has been a crucial issue in the field of Peer-to-Peer networks since the design of the first network topologies like Chord [SMK+01]. In the original Chord paper, it was proposed that each real peer works as $\log n$ virtual nodes, thus greatly decreasing the probability that some peer will get a large part of the ring. Some extensions of this method were proposed by Rao et al. [RLS+03] and Godfrey et al. [GLS+04], who introduce more schemes based on virtual nodes and evaluate them experimentally. Unfortunately, such an approach increases the out-degree of each peer by a factor of $\log n$, because each peer has to keep all the links of all its virtual nodes.

The paradigm of many random choices [MRS00], from the area of throwing balls into bins games, was used by Byers et al. [BCM03] and by Naor and Wieder [NW03a, NW03b]. When a peer joins, it contacts $\log n$ random places in the network and chooses to cut the longest of all the found intervals. This yields constant smoothness, with high probability.

A similar approach was proposed by Adler et al. [AHKV03]. They improve the paradigm of many random choices in the following way. All bins build a hypercubic structure, where each bin has direct connections to $\log n$ other bins. Such a structure can be easily simulated using Chord and other networks with hypercubic shortcuts. When a ball should be thrown into a random bin, instead of checking the load of $\log n$ randomly chosen bins, it chooses just one random bin and checks its neighbors. The authors prove, that this one random choice generates $\log n$ pseudorandom choices, which are sufficient to spread the load evenly among all bins, with high probability.

The two approaches above have a certain drawback. They both assume that peers join the network sequentially. What is more important, they do not provide analysis for the problem of balancing the intervals afresh when peers leave the network.

Karger and Ruhl [KR04a, KR04b] propose a scheme, in which each node chooses $\Theta(\log n)$ places in the network and takes responsibility for only one of them. This can change, if some nodes leave or join, but each node migrates only among the $\Theta(\log n)$ places it chose, and after each operation $\Theta(\log\log n)$ nodes have to migrate on expectation. The algorithm uses only tiny messages for checking the network state, and the communication cost in a half-life can be bounded to $\Theta(\log n)$ messages sent to logarithmic distance per peer. The scheme is claimed to be resistant to attacks thanks to the fact that each node can only join in logarithmically bounded number of places on the ring. However, Awerbuch and Scheideler [AS04a] show, that such a scheme cannot be secure and that more sophisticated algorithms are needed to provide provable security. The reasoning for this is that with the IPv6 protocol the adversary has access to thousands of IP numbers, and she can join the system with the ones falling into an interval that she has chosen. She does not have to join the system with each possible IP, because the hash functions are public and she can check usefulness of an IP offline.

Manku [Man04] presented a scheme based on a virtual binary tree that achieves constant smoothness with low communication cost for peers joining or leaving the network. It is also shown that the smoothness can be diminished to as low as $(1 + \epsilon)$ with communication cost per operation increased to $O(1/\epsilon)$. All nodes form a binary tree, where some of them (called *active*) are responsible for perfect balancing of subtrees rooted at them.

Kenthapadi and Manku [KM05] present a scheme that extends [Man04] and generalizes [AHKV03]. Nodes are grouped into clusters of size h. Each node checks k clusters when it joins the network (In [AHKV03], $h = \log n$ and $k = 1$). The authors prove that it is sufficient, that $h \cdot k = \Omega(\log n)$ to guarantee that the resulting smoothness is constant with high probability. Similarly to the other two approaches basing on the paradigm of many random choices, this algorithm does not consider nodes leaving the network.

2.3. Uniform distribution of nodes on the ring

In this section we examine the smoothness in case when all peers choose the places for themselves independently uniformly at random. We show that it is very likely that the resulting smoothness is $\Theta(n \cdot \log n)$. In order to be able to treat the nodes thrown into the $[0, 1)$ interval as balls thrown into bins we define a natural partition of the $[0, 1)$ interval into identical sectors, each of which is later treated as a bin. The nodes placed on the ring will be treated as balls thrown into these bins.

Definition 2.3. *The* canonical division *of the unit ring into m sectors is a division into m disjoint intervals (called sectors) of length $\frac{1}{m}$, where one of the sectors is the interval $[0, \frac{1}{m}]$. The latter is* the first sector*; the subsequent sectors are nubmered from* 2 *to m.*

The following result belongs to folklore but we were not able to actually find it in literature. Therefore, we prove it here for completeness

Theorem 2.4. *If n nodes are distributed uniformly and independently at random on the unit ring the following hold:*

1. *The longest interval generated by the nodes has length $\Theta\left(\frac{\log n}{n}\right)$, with high probability*
2. *The shortest interval generated by the nodes has length $\Theta\left(\frac{1}{n^2}\right)$, with constant probability*

Proof. For the first claim, consider the canonical division of the unit ring into $m = \frac{c \cdot n}{\ln n}$ sectors of length $\frac{1}{m}$. First, we bound the probability that any sector is empty if $c > 0$ is sufficiently small, i. e. sectors are sufficiently long. If all sectors are non-empty then the longest interval generated by the nodes has length at most $\frac{2}{m} = \frac{2 \cdot \ln n}{c \cdot n} \in O\left(\frac{\log n}{n}\right)$. For sufficiently small c and large enough n:

$$\begin{aligned}
\Pr(\text{there exists an empty sector}) &\leq m \cdot \Pr(\text{the first sector is empty}) \\
&\leq m \cdot \left(1 - \frac{1}{m}\right)^n = \frac{c \cdot n}{\ln n} \cdot \left(1 - \frac{\ln n}{c \cdot n}\right)^{\frac{c \cdot n}{\ln n} \cdot \frac{\ln n}{c}} \\
&\leq \frac{c \cdot n}{\ln n} \cdot \left(\frac{1}{e}\right)^{\frac{\ln n}{c}} \leq \frac{c \cdot n}{\ln n} \cdot \frac{1}{n^{1/c}} \leq \frac{1}{n^{2/c}}
\end{aligned}$$

On the other hand, if there is at least one empty sector, then the length of the longest interval generated by the nodes, is at least $\frac{1}{m} = \frac{\ln n}{c \cdot n} \in \Omega\left(\frac{\log n}{n}\right)$ and for sufficiently large $c > 1$ and large enough n:

$$\begin{aligned}
\Pr(\text{all sectors are non-empty}) &= \Pr(\text{the first is sector non-empty}) \\
&\quad \cdot \prod_{i=2}^{m} \Pr(\text{sector } i \text{ is non-empty} \quad | \\
&\quad\quad \text{sectors } 1 \ldots i-1 \text{ are non-empty}) \\
&\overset{(1)}{\leq} \prod_{i=1}^{m} \Pr(\text{sector } i \text{ is non-empty}) = \prod_{i=1}^{m} \left(1 - \left(1 - \frac{1}{m}\right)^n\right) \\
&= \prod_{i=1}^{\frac{c \cdot n}{\ln n}} \left(1 - \left(1 - \frac{\ln n}{c \cdot n}\right)^n\right) \leq \left(1 - \frac{1}{n^{2/c}}\right)^{\frac{c \cdot n}{\ln n}} \leq \frac{1}{e^{\sqrt{n}}} \leq \frac{1}{n^{2/c}}
\end{aligned}$$

In inequality (1) above we used the negative dependence of non-emptiness of particular sectors. The latter is easy to explain: when in a balls into bins game, for a concrete bin we have the information that some other bins already used some balls, then the probability, that the chosen bin is empty, decreases. This statement is formally proved in [DR98].

For the second claim of the theorem, consider the canonical division of the unit ring into $m = c \cdot n^2$ sectors of length $\frac{1}{m}$. Again, we treat the sectors as m bins into which we throw n balls. First, we give an upper and lower bound for the probability that there exists a sector containing at least two nodes. Actually, we consider the opposite event.

$$\Pr(\text{each sector contains} \leq 1 \text{ node}) = 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdots \left(1 - \frac{n-1}{m}\right)$$

Now, to give an upper bound for the event that there exists a sector containing at least two nodes we estimate the calculated probability for large enough n and sufficiently large $c > 0$ $\left(c \geq \frac{1}{2\cdot\ln(2/(2-\epsilon))}\right)$ in the following way:

$$\Pr(\text{each sector contains} \leq 1 \text{ node}) \;\geq\; \left(1 - \frac{n}{m}\right)^{n} = \left(1 - \frac{1}{c\cdot n}\right)^{c\cdot n\cdot\frac{1}{c}} \geq e^{-\frac{1}{2\cdot c}} \geq 1 - \frac{\epsilon}{2}$$

so with arbitrarily large (but smaller than 1 by a constant) probability, each sector contains at most one node.

The latter does not guarantee a lower bound for the length of the shortest interval generated by the nodes. Even if each sector of length $\Theta\left(\frac{1}{n^2}\right)$ of the canonical division contains at most one node it may happen that two nodes lie very close. But if we perform a second experiment in which all sectors are shifted by $\frac{1}{2\cdot m}$, that is the first sector starts in the point $\frac{1}{2\cdot m}$ (in the middle of the first sector from the first experiment), and we use the random choices from the first experiment (the nodes are thrown exactly into the same places on the ring), then we have the same bound for two nodes being in the same sector for the second experiment. Both experiments are dependent but still the probability that in any of them there exists a sector with two nodes is bounded by ϵ. Now, if two nodes were in distance at most $\frac{1}{4\cdot m}$, they would be in the same sector either in the first or in the second experiment. Thus, with arbitrarily large (but smaller than 1 by a constant) probability the shortest interval generated by the nodes has length $\Omega\left(\frac{1}{n^2}\right)$.
In order to lower bound the probability that there exists a sector containing at least two nodes, we choose sufficiently small $c > 0$ $\left(c \leq \frac{1}{4\cdot\ln\frac{1}{\epsilon}}\right)$:

$$\Pr(\text{each sector contains} \leq 1 \text{ node}) \;\leq\; \left(1 - \frac{n}{2\cdot m}\right)^{\frac{n}{2}} = \left(1 - \frac{1}{2\cdot c\cdot n}\right)^{2\cdot c\cdot n\cdot\frac{1}{4\cdot c}} \leq e^{-\frac{1}{4\cdot c}} \leq \epsilon$$

so with arbitrarily high probability (but smaller than 1 by a constant) there exists a sector which contains two nodes and thus the shortest interval generated by the nodes has length at most $\frac{1}{m} = \frac{1}{c\cdot n^2} \in O\left(\frac{1}{n^2}\right)$. ■

Chapter 3

Approximating the Current Number of Nodes

In this chapter we present two very similar schemes which are used to approximate the number of nodes present in the network. The first one is based on the work of Awerbuch and Scheideler [AS04a] and the second one was independently designed by us [BK05]. In both schemes all nodes insert a fixed number of markers (markers are special items or connections) into the $[0, 1)$ ring. In the first one each node inserts one marker and has to communicate to its logarithmic neighborhood, and in the second one, each node inserts logarithmic number of markers and draws conclusion basing only on the interval for which it is responsible. Both schemes yield a constant approximation of the number of nodes. Our scheme needs an upper bound on $\log n$ in order to work properly. In Chapter 5 it will be shown that our scheme is more compatible with our balancing algorithms.

The actual purpose of both approximation algorithms is to provide a tool to categorize intervals according to their load. If we know whether a node's interval is too long, too short or just fine, we can decide if it needs help, can help, or is content with its load.

Both approximation schemes can work with both balancing algorithms. However, we describe them in such a way that the details in the 1-marker scheme are adjusted to the self-stabilizing algorithm AlgStat presented in Chapter 4 and the details in the $\log n$-markers scheme are adjusted to the bounded-communication algorithm AlgDyn presented in Chapter 5.

3.1. The 1-marker scheme

We describe a scheme based on the approach of Awerbuch and Scheideler [AS04a], who give an algorithm which yields a constant approximation of n in every node assuming that the nodes are distributed *uniformly at random* in the interval $[0, 1]$. We cannot make such an assumption, since our balancing algorithms heavily change the positions of nodes on the ring. Besides, our balancing algorithms can start with any (even chosen by an adversary) distribution of nodes on the ring. Assuming that they are distributed uniformly would deprive our results of this advantage.

As a preparation for the actual approximation algorithm, each node keeps a connection to one random position on the ring (instead of such special connections, special items can be inserted into the virtual space). This position is called a *marker*. All markers present in the system define weights of intervals in the following way.

Definition 3.1. *For a node i responsible for interval I_i its weight $w(I_i)$ is the number of markers, which fall into I_i.*

The positions of markers are not fixed all the time. After finding an initial random position for its marker, each node performs the following infinite process. It keeps its marker in the found position but at the same time chooses a new random position and contacts a node responsible for it. After successfully contacting the node responsible for the chosen place, it migrates its marker there and starts looking for another random position. Thus, the position of each marker is fixed only for D rounds, the time it takes for any node to contact any other node in the network.

The process of constantly changing the positions of markers is needed for the following reason. We show that for a fixed random configuration of markers our balancing algorithm works properly with high probability. However, since the process runs forever, and nodes are allowed to leave and join (and thus change the positions of their markers), a bad configuration may (and will) appear at some point in time. Thanks to the continuous changing of the positions of markers, the probability of failure in time step t is independent of the probability of failure in time step $t + D$, since in the time interval $[t, t + D]$ each node changes the position of its marker at least once. This enables the process to recover even if a bad event occurs.

The Algorithm APPROX-1 is meant to run on each node i responsible for interval I_i independently from other nodes. A node i approximates the number of nodes present in the network in the following way. The goal is to build an interval $[a, b)$ of length $\Theta\left(\frac{\log n}{n}\right)$. The beginning of interval $[a, b)$ is fixed to a_i, i.e. the beginning of the interval $I_i = [a_i, b_i)$ for which i is responsible (the first line of the algorithm). Initially, the right end b of the constructed interval is b_i (lines $2 - 3$).

The condition in line 4 is selected so, that the interval $[a, b)$ grows (in lines $5 - 6$) as long as it is too short. If it is shorter than $O\left(\frac{\log n}{n}\right)$, its weight is smaller than $O(\log n)$ with high probability and it is compared to $\log(1/l([a, b))$ which is $\Omega(\log n)$. Thus, the interval grows until its length is $\Omega\left(\frac{\log n}{n}\right)$.

Algorithm 1 APPROX-1 (node i responsible for interval $I_i = [a_i, b_i)$)

```
1: $a \leftarrow a_i$
2: $b \leftarrow b_i$
3: $j \leftarrow i$
4: while $w([a,b)) < \log \frac{1}{l([a,b))}$ do
5:     $j \leftarrow j.succ$
6:     $b \leftarrow b_j$
7: while $w([a,b)) > \log \frac{1}{l([a,b))}$ do
8:     decrease $b$ to the position of the last marker in $[a,b)$
9: increase $b$ so that $w([a,b)) = \log \frac{1}{l([a,b))}$
10: return $l([a,b))$
```

In lines 7 – 9 it is shortened to $\Theta\left(\frac{\log n}{n}\right)$, which is done locally using only information from the last node contributing to the interval $[a,b)$. Awerbuch and Scheideler prove that the interval $[a,b)$ returned by the algorithm has length $\Theta\left(\frac{\log n}{n}\right)$, with high probability. This is stated formally in the following lemma from [AS04a]. Besides, they prove that in case when nodes are distributed uniformly at random, the computation time and communication cost of APPROX-1 are bounded by $O(\log n)$, with high probability.

Lemma 3.2. *The interval [a,b) returned by the algorithm* APPROX-1 *fulfills* $\alpha \cdot \frac{\log n}{n} \leq l([a,b)) \leq \beta \cdot \frac{\log n}{n}$ *for constants* α *and* β*, with high probability.*

We reformulate this lemma in order to get an approximation of the number of nodes n from an approximation of $\frac{\log n}{n}$. The following theorem states the exact result.

Theorem 3.3. *Let* $l([a,b))$ *be the length of an interval found by node i. Let* n_i *be the solution of* $\log x - \log\log x = \log(1/l([a,b)))$. *Then with high probability* $\frac{n}{\beta^2} \leq n_i \leq \frac{n}{\alpha^2}$.

Similarly to constantly changing the positions of markers, the algorithm APPROX-1 is also executed time after time. This is meant to assure that the information about the number of nodes present in the network is as fresh as possible. In case when nodes are initially distributed uniformly at random, the information is at most $O(\log n)$ time steps old, as it takes $O(\log n)$ time steps to execute APPROX-1 . In the model in Chapter 4, the situation is static, that is no nodes leave or join the system during the execution of load balancing. We may assume that each node executes APPROX-1 once before the actual load balancing procedure. This contributes $\Theta(\log n)$ time steps to the running time.

In the design of the balancing algorithm we assume that each peer has computed n_i. Additionally, there are global constants l and u such that $l \cdot n_i \leq n \leq u \cdot n_i$, for each i.

We categorize intervals according to their lengths, whereas the categorization is slightly influenced by the quality of the computed approximation. An interval can move between categories without changing its absolute length. This can be caused by the change of n_i when the distribution of markers in the neighborhood of i changes, especially when the size of the network grows or shrinks. An interval I_i is called:

very short if $l(I_i) \leq \frac{1}{2 \cdot l \cdot n_i}$

short if $\frac{1}{2 \cdot l \cdot n_i} \leq l(I_i) \leq \frac{4}{l \cdot n_i}$

medium if $\frac{4}{l \cdot n_i} \leq l(I_i) \leq \frac{8 \cdot u}{l^2 \cdot n_i}$

long if $l(I_i) \geq \frac{8 \cdot u}{l^2 \cdot n_i}$

From the definition and from the quality of approximation we can conclude that each medium or long interval has length at least $\frac{4}{l \cdot n_i} \geq \frac{4}{n}$, since $l \cdot n_i \leq n$. Thus, there are at most $\frac{n}{4}$ medium and long intervals present in the network at any point in time. On the other hand, long intervals are defined so that by halving a long interval we never obtain a short interval. This comes from the fact that for each short interval I_i it holds that $l(I_i) \leq \frac{4}{l \cdot n_i} \leq \frac{u \cdot 4}{l \cdot n}$ and if a long interval is halved, the lengths of the halves are at least $\frac{4 \cdot u}{l^2 \cdot n_i} \geq \frac{u \cdot 4}{l \cdot n}$. The following fact assembles the results of this section

Fact 3.4. *The approximation scheme* APPROX-1 *together with the classification of intervals' lengths have the following properties:*

- *each node is able to find to which category it belongs,*
- *all but very short intervals have lengths $\Omega\left(\frac{1}{n}\right)$,*
- *all but long intervals have lengths $O\left(\frac{1}{n}\right)$,*
- *there are at most $\frac{n}{4}$ medium and long intervals,*
- *after gluing two adjacent short intervals, one never gets a long interval,*
- *after halving a long interval, one never gets a short interval.*

3.2. The $\log n$-markers scheme

In the second scheme, which is our independent design, we look at weights of intervals instead of their lengths. The expected length of an interval is $1/n$ and each node would like to be responsible for an interval of length equal to $\Theta(1/n)$. However, in order to be able to decide if an interval is shorter or longer than $1/n$, we need to know n. Instead of looking at lengths of intervals, nodes look at their weights and we tinker with markers so that for each interval it is known at first sight, if it is too light or too heavy. Besides, we show how from the information about the interval's weight and absolute length one can compute an approximation of its relative length and thus also an approximation of n.

Let Δ be an upper bound on $\log n$ common for all nodes present in the system. Each node keeps connections to $\delta = \alpha \cdot \Delta$ random positions on the ring, for sufficiently large global constant α. Similarly as in the previous section, these connections are called markers and define weights in the same way as in the Definition 3.1.

It is clear that $E[w(I_i)] = \delta \cdot l(I_i) \cdot n$, so $n_i = \frac{w(I_i)}{\delta \cdot l(I_i)}$ is a good estimate for n. In the following theorem we state precisely the relation between the length and the weight of an interval and prove, using Chernoff bounds, that the quality of the estimate is sufficient.

Theorem 3.5. *For any constant $w_{\min} > 0$, there exist constants $l_{\min}$ and $l_{\max}$, such that for an arbitrary interval I_i, time step t, and sufficiently large α, the following holds with high probability. In time step t, if $w(I_i) \geq w_{\min} \cdot \delta$, then $\frac{l_{\min}}{\delta \cdot n} \cdot w(I_i) \leq l(I_i) \leq \frac{l_{\max}}{\delta \cdot n} \cdot w(I_i)$; otherwise $l(I_i) \in O\left(\frac{1}{n}\right)$.*

Proof. For sufficiently large constant s and an interval I of length $l(I) = \frac{s}{n}$, let a random variable X denote the weight of I. The expected value of X is $E[X] = s \cdot \delta$ and Chernoff bounds guarantee that for sufficiently large α, $X \geq w_{\min} \cdot \delta$, with high probability. Thus, if for an interval I_i it holds that $w(I_i) \leq w_{\min} \cdot \delta$, then with high probability $l(I_i) \leq \frac{s}{n} \in O\left(\frac{1}{n}\right)$.

For an interval I with $l(I) \geq \frac{s}{n}$, let a random variable X denote its weight again. Since its expected value $E[X] = n \cdot \delta \cdot l(I) \in \Omega(\log n)$, Chernoff bounds guarantee that with high probability $X \in \Theta(E[X])$. Let the constants in the Θ notation be such that with high probability $\frac{1}{l_{\max}} \cdot n \cdot \delta \cdot l(I) \leq X = w(I) \leq \frac{1}{l_{\min}} \cdot n \cdot \delta \cdot l(I)$. Then for an interval I_i, with high probability $\frac{l_{\min}}{\delta \cdot n} \cdot w(I_i) \leq l(I_i) \leq \frac{l_{\max}}{\delta \cdot n} \cdot w(I_i)$.

■

Without using the knowledge about weights, it is hard for the nodes to compare their length to the average length (which is $1/n$) due to the lack of knowledge about the exact number of participants n. On the other hand it is easy for them to compare their weight to the average weight, since the latter is a global constant equal to δ.

Similarly to the previous section, we categorize the intervals according to their weight. However, this time the categorization has a parameter $w_{\min}$ which states the threshold to consider an interval very light. We assume that this threshold satisfies $0 \leq w_{\min} < 16$. An interval can change its category even without changing its absolute length. It happens when it gains or loses weight, which is caused by other intervals joining or leaving the network. However, since the maximum dilation D in the network is $O(\log n)$, it takes other nodes at most $O(\log n)$ time steps to update the information about their markers, so the information about the category to which a node currently belongs is at most $O(\log n)$ time steps old. An interval I_i is called:

very light if $w(I_i) \leq w_{min} \cdot \delta$

light if $w_{min} \cdot \delta \leq w(I_i) \leq 16 \cdot \delta$

medium if $16 \cdot \delta \leq w(I_i) \leq 32 \cdot \delta$

heavy if $w(I_i) \geq 32 \cdot \delta$

Such categorization together with Theorem 3.5 lets us look at weights of intervals in situations in which we would like to know their relative lengths. The load balancing algorithm will reorganize the positions of nodes on the ring so that the resulting intervals have weights betwen $w_{min} \cdot \delta$ and $32 \cdot \delta$. By Theorem 3.5, this implies that in the end the smoothness is constant, with high probability.

Notice that from the definition we can conclude that each medium or heavy interval has weight at least $16 \cdot \delta$, and since the total weight of the network is $n \cdot \delta$, there are at most $\frac{n}{16}$ medium and heavy intervals in the network at any point in time. Besides, for any not very light interval I_i, its length is proportional to its weight $l(i) = \Theta\left(\frac{w(I_i)}{\delta \cdot n}\right)$.

The following fact summarizes the results of this section which will be needed in Chapter 5.

Fact 3.6. *The approximation algorithm together with the classification of weights have the following properties, all holding with high probability:*

- *each node is able to find to which category it belongs,*
- *all but very light intervals have lengths $\Omega\left(\frac{1}{n}\right)$,*
- *all but heavy intervals have lengths $O\left(\frac{1}{n}\right)$,*
- *there are at most $\frac{n}{16}$ medium and heavy intervals,*
- *after gluing two adjacent light intervals, one never gets a heavy interval,*
- *after halving a heavy interval, one never gets a light interval.*

CHAPTER 4

Self Stabilizing Load Balancing

In this chapter we describe the algorithm AlgStat, a self-stabilizing scheme that starting with an arbitrary placement of nodes on the ring, repairs the network in a constant number of rounds, where a single round lasts as long as it takes a peer to send a message to some other peer in the network, i.e. D time steps. We describe the algorithm, so that nodes work in synchronized rounds, which is not crucial for the correctness but significantly simplifies the description of the algorithm. However, it is important, that at least some synchronicity is guaranteed, that is that all nodes are of the same speed up to constant factors and that messages send from neighbor to neighbor take only bounded time to reach their destination. In such case, one can use techniques similar to those described in Chapter 5 instead of dividing the execution into rounds.

For the simplicity of the analysis, we fix a static situation with some number n of peers in the system, which means that no nodes join or leave the system. Such assumption is needed in the analysis but the algorithm itself is designed in such a way that it can be implemented also in a dynamic system. We do not prove the latter, but we show the results of experiments that we performed and that indicate good behavior of the algorithm.

4.1. The load balancing algorithm AlgStat

In Chapter 3 we divided the intervals created by the placement of nodes on the ring into four classes: very short, short, medium, and long. In the rest of this chapter we need only the properties of the classification stated in Fact 3.4 and we will not use the concrete values defined in Chapter 3.

As mentioned above, we assume that the network is frozen for the execution of the balancing procedure. Therefore we may think that the algorithm Approx-1, which estimates the number of nodes in the network is run once in the beginning. Since

Algorithm 2 ALGSTAT (node i responsible for interval I_i)

```
1:  if $I_i$ is short then
2:      i.state ← staying
3:      if i.pred is short then
4:          with probability $\frac{1}{2}$ i.state ← leaving
5:      synchronize
6:      if i.state = leaving and i.pred.state=staying then
7:          $p \leftarrow$ random([0,1))
8:          $P \leftarrow$ the node responsible for $p$
9:          send help proposals to $P$ and its $\mathcal{F} \cdot \log(u \cdot n_i)$ successors on the ring
10:         if a node $j$ accepts then
11:             migrate to the middle of interval $I_j$
12:         at any time if any node proposes then reject
13: else if $I_i$ is medium then
14:     if node $j$ contacts then
15:         reject
16: else if $I_i$ is long then
17:     wait for help proposals
18:     if node $j$ contacts then
19:         let $j$ migrate to the middle of $I_i$
```

no nodes join or leave the network in the meantime, the next calls to APPROX-1 will categorize the intervals identically to the initial call.

The algorithm ALGSTAT will minimize the length of the longest interval, but we also have to take care that no interval is too short. Therefore, before we begin the routine, we force all nodes responsible for very short intervals to leave the network. By doing this, we assure that the length of the shortest interval in the network will be bounded from below by $\Omega\left(\frac{1}{n}\right)$. We have to explain why this does not destroy the structure of the network.

First of all, it is possible that we remove a huge fraction of the nodes. It is even possible that a very long interval appears, even though the network was balanced before. This is not a problem, since the algorithm will rebalance the system. Besides, if ALGSTAT is used also for new nodes at the moment of joining, this initialization will never be needed. We do not completely remove the nodes with too short intervals from the network. The number of nodes n and thus also the number of markers is unaffected, and the removed nodes will later act as though they were simply responsible for short intervals. Each of these nodes can contact the network through the node responsible for its marker. Another possibility is for each such node to have a virtual position on the ring and contact the network through a node responsible for this position.

The algorithm AlgStat works in rounds. The routine works differently for different nodes, depending on the initial node's interval's length. The pseudocode accounting for all types of intervals is depicted above. Below, we elaborate on the behavior of all types of nodes.

The most interesting part of the algorithm takes place in nodes responsible for short intervals (lines 1 – 12). The purpose of the nodes responsible for short intervals is to find and help nodes responsible for long intervals, thus reducing their load. The first thing a node responsible for a short interval does is to make a decision if it can send a help proposal in this round. The variable `state` stores the decision of a node: it is set either to `leaving` or `staying` which means that the node is or is not willing to help, respectively. In line 2 the node initially decides to stay, as in order to migrate it has to check that its neighborhood fulfills additional conditions. The first condition, checked in line 3 is that the predecessor of the node is short too. If this is the case, and the node migrates, its predecessor overtakes its job and is not overloaded. The second condition, checked in line 6 is that the predecessor decides not to migrate in this round. In order to maximize the number of nodes which fulfill both conditions and are willing to migrate, each node, after checking the first condition, decides to change its state to `leaving` only with probability $\frac{1}{2}$ in line 4. Before checking the second condition all nodes are synchronized in line 5. It assures that a node checks the state of its predecessor only if it has already been fixed. In the next section we prove that in each round, linear number of nodes are willing to migrate, with high probability.

If a node responsible for a short interval decides to send a help proposal, it sends it to a random place p on the ring chosen in line 7. In lines 9 – 12 it checks whether the node responsible for p or any of its $\Theta(\log n)$ successors is responsible for a long interval and needs help. The distance through which a help proposal should be forwarded is dependent on a constant $\mathcal{F}$ which is a parameter of the algorithm. The larger the constant, the longer the distance. If a node responsible for a long interval is found, the node sending a help proposal migrates to its middle. In the next section we prove that linear number of proposals together with each proposal visiting logarithmic number of nodes, suffice to eventually cut each long interval into medium intervals with high probability. In line 12 the algorithm assures that if a help proposal comes to a node responsible for a short interval it is not accepted.

The medium intervals are on the one hand short enough, so that nodes responsible for them do not need help, and on the other hand, they are too long to send help offers themselves. Therefore the pseudocode for them (lines 13 – 15) is the simplest: it rejects all help proposals.

As long as a node is responsible for a long interval, it wants to split. It does not send messages calling for help, but rather waits for messages with help offers. A node responsible for a long interval awaits help proposals (lines 16 – 17) and splits as soon as one comes (lines 18 – 19).

4.2. The analysis of ALGSTAT

All properties of the algorithm ALGSTAT are stated and proved in the following theorem.

Theorem 4.1. ALGSTAT *has the following properties, all holding with high probability:*

1. *In each round each node incurs a communication cost of at most $O(D + \log n)$.*
2. *The total number of migrated nodes is within a constant factor from the number of migrations generated by an optimal centralized algorithm with the same initial network state.*
3. *Each node is migrated at most once.*
4. *$O(1)$ rounds are sufficient to achieve constant smoothness.*

Proof. The first statement of the theorem follows easily from the algorithm due to the fact that each node responsible for a short interval sends a message to a random destination which takes time at most D and then consecutively contacts the successors of the found node. This incurs additional communication cost of at most $\mathcal{F} \cdot (\log n + \log u)$. Additionally, in each round each node changes the position of its marker and this operation also incurs communication cost at most D.

If a node tries to leave the network and join it somewhere else, it is certain that its predecessor is short and is not going to change its location. This assures, that the predecessor will take over the job of the migrating node and it will not become long. Therefore, no long interval is ever created and as a result of the algorithm long intervals are cut into halves. Both ALGSTAT and an optimal centralized algorithm have to cut each long interval into medium intervals. Let M and S be the upper thresholds for the lengths of a medium and short interval, respectively, and I be an arbitrary long interval with length $l(I)$. An optimal algorithm needs at least $\lceil l(I)/M \rceil$ cuts, whereas ALGSTAT always cuts an interval in the middle and performs at most $2^{\lceil \log(l(I)/S) \rceil}$ cuts, which can be at most constant times larger because M/S is constant. This proves the second statement of the theorem.

The statement that each node is migrated at most once follows from the reasoning below. A node is migrated only if its interval is short. Due to the gap between the upper threshold for short interval and the lower threshold for long interval, after being migrated a node never takes responsibility for a short interval, so it will not be migrated again.

In order to prove the last statement of the theorem, we show the following two lemmas. The first one shows how many nodes responsible for short intervals are willing to help during a constant number of rounds. The second one states how many helpful nodes are needed so that AlgStat succeeds in balancing the system.

Lemma 4.2. *For any constant $a \geq 0$, there exists a constant c, such that in c rounds at least $a \cdot n$ nodes send help proposals, with high probability.*

Proof. As stated before, the length of each medium or long interval is at least $\frac{4}{n}$ and thus at most $\frac{1}{4} \cdot n$ intervals are medium or long. Therefore, we have at least $\frac{3}{4} \cdot n$ nodes responsible for short intervals.

We number all the nodes in order of their position on the ring with numbers $0, \ldots, n-1$. For simplicity we assume that n is even, and divide the set of all nodes into $n/2$ pairs $P_i = (2i, 2i+1)$, where $i = 0, ..., \frac{n}{2} - 1$. Then, there are at least $\frac{1}{2} \cdot n - \frac{1}{4} \cdot n = \frac{1}{4} \cdot n$ pairs P_i, which contain indexes of two short intervals. Since the first element of a pair is assigned state `staying` with probability at least $1/2$ and the second element state `leaving` with probability $1/2$, the probability that the second element is eager to migrate is at least $1/4$. For two different pairs P_i and P_j, migrations of their second elements are independent. We stress here that this reasoning only bounds the number of nodes able to migrate from below. For example, we do not consider first elements of pairs which also may migrate in some cases. Nevertheless, we are able to show that the number of migrating elements is large enough. Notice also that even if in one round many of the nodes migrate, it is still guaranteed that in each of the next rounds there will still exist at least $\frac{3}{4} \cdot n$ short intervals.

The above process stochastically dominates a Bernoulli process with $c \cdot n/4$ trials and single trial success probability $p = 1/4$. Let X be a random variable denoting the number of successes in the Bernoulli process. Then $E[X] = c \cdot n/16$ and Chernoff bound guarantees that $X \geq a \cdot n$, with high probability, if we only choose c large enough with respect to a. ■

In the following lemma we deal with cutting one long interval into medium intervals.

Lemma 4.3. *There exists a constant b such that for any long interval I, after $b \cdot n$ help proposals are generated overall, the interval I will be cut into medium intervals, w.h.p.*

Proof. For the further analysis we will need that $l(I) \leq \frac{\log n}{n}$, therefore we first consider the case where $l(I) > \frac{\log n}{n}$. We first estimate the number of help proposals that have to be generated in order to cut I into intervals of length at most $\frac{\log n}{n}$. We depict the process of cutting I on a binary tree in the following way. Let I be the root of this tree and its children the two intervals into which I is cut after it receives the first help proposal. The tree is built further in the same way and achieves its lowest level when its nodes have length s such that $\frac{1}{2} \cdot \frac{\log n}{n} \leq s \leq \frac{\log n}{n}$. The tree has height at most $\log n$. If a leaf gets $\log n$ help proposals, it can use them to cover the whole path from itself to the root. Such covering is a witness that this interval will be separated from others. Thus, if each of the leaves receives $\log n$ help proposals, the interval I will be cut into intervals of length at most $\frac{\log n}{n}$.

Let b_1 be a sufficiently large constant and consider the first $b_1 \cdot n$ help proposals. We will bound the probability that one of the leaves gets at most $\log n$ of these help proposals. Let X be a random variable depicting how many help proposals fall into a leaf J. The probability that a help proposal hits a leaf is equal to the length of this leaf and the expected number of help proposals that hit a leaf is $E[X] \geq b_1 \cdot \log n$. Chernoff bound guarantees that, if b_1 is large enough, the number of help proposals is at least $\log n$, with high probability.

There are at most n leaves in this tree, so each of them gets sufficiently many help proposals, with high probability. In the further phase we assume that all the intervals existing in the network are of length at most $\frac{\log n}{n}$.

Let J be any of such intervals. Consider the maximal possible set K of predecessors of J, such that their total length is at most $2 \cdot \frac{\log n}{n}$ (excluding J). Maximality assures that $l(K) \geq \frac{\log n}{n}$. The upper bound on the length assures that even if the intervals belonging to K and J are cut (“are cut” in this context means “have been cut”, “are being cut” and/or ”will be cut”) into smallest possible pieces (of length $\frac{2}{n}$), their number does not exceed $6 \cdot \log n$. The parameter $\mathcal{F}$ is sufficiently large, so that if a help proposal hits some of them and is not needed by any of them, it is forwarded to J and can reach its end. We consider only the help proposals that hit K. Some of them will be used by K and the rest will be forwarded to J.

Let b_2 be a constant and Y be a random variable denoting the number of help proposals that fall into K in a process in which $b_2 \cdot n$ help proposals are generated in the network. We want to show that, with high probability, Y is large enough, i.e. $Y \geq 2 \cdot n \cdot (l(J) + l(K))$. The expected value of Y can be estimated as $E[Y] = b_2 \cdot n \cdot l(K) \geq b_2 \cdot \log n$. Again, Chernoff bound guarantees that $Y \geq 6 \cdot \log n$, with high probability, if b_2 is large enough. This is sufficient to cut both K and J into medium intervals.

Now taking $b = b_1 + b_2$, finishes the proof of Lemma 4.3. ■

Combining Lemmas 4.2 and 4.3 and setting $a = b$, finishes the proof of Theorem 4.1. ■

Both the approximation and load balancing algorithms are analyzed in a static environment but they can also run as everlasting processes. In such case the following corollary summarizes the results of Theorem 4.1 and Chapter 3.

Corollary 4.4. *If the algorithms* Approx-1 *and* AlgStat *are executed as an everrunning processes and the system is static for* $\Theta(\log n)$ *time steps, they rebalance the system so, that smoothness is constant, with high probability. The probability of success in time step* t *is independent of the probability of success in time step* $t + \Theta(\log n)$.

4.3. Experimental evaluation

In order to adjust the parameters and examine the practical usefulness of our approximation scheme (the one using $\log n$ markers) and the algorithm AlgStat we performed experiments on a single machine. In all tests the underlying space was the ring of natural numbers $\{0, \ldots, 2^{64} - 1\}$ with arithmetics mod 2^{64}.

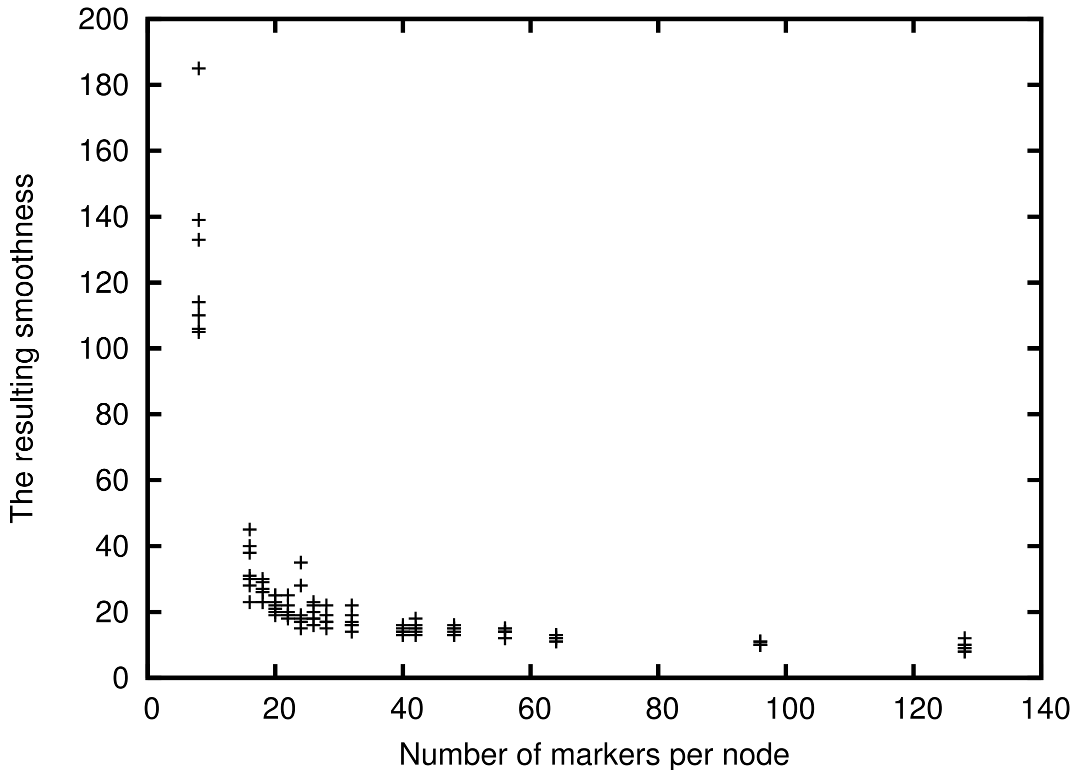

Figure 4.1.: Smoothness according to approximation of $\log n$

There were two groups of experiments. The first group was to find the best parameters for the algorithm in the static case. In the tests in this group n nodes were inserted in the very beginning and the system was static (no nodes joined or left during the execution of the algorithm. The number of nodes n was 2^5, 2^{11}, 2^{14} or 2^{17}; they were chosen to be powers of two only for easier implementation. We present the results for the highest number of nodes, that is for approximately 130 thousands nodes.

The first parameter we tested was the number of markers that each peer should insert into the network. The number of markers δ ranged from 8 to 128 and the other parameters were set as follows. Each help proposal was forwarded through 128 nodes to assure quick convergence. A node could send a help proposal if its interval glued with the interval of its predecessor had weight at most δ. An interval was considered very light if its weight was below $\delta/2$. We measured the final smoothness in the system and noticed that setting δ to 40 already yields smoothness below 20 and values larger than 64 do not improve the system much. The final smoothness for different δ parameters is depicted in Figure 4.1.

With the number of markers fixed to 64 we performed tests which had to estimate what intervals should be considered very light. The parameter deciding how far a request should be forwarded was set to 128 again. The final smoothness for different weights of a very light interval is depicted in Figure 4.2. We propose to consider an interval very light if its weight is below $\frac{7 \cdot \delta}{8} = 56$.

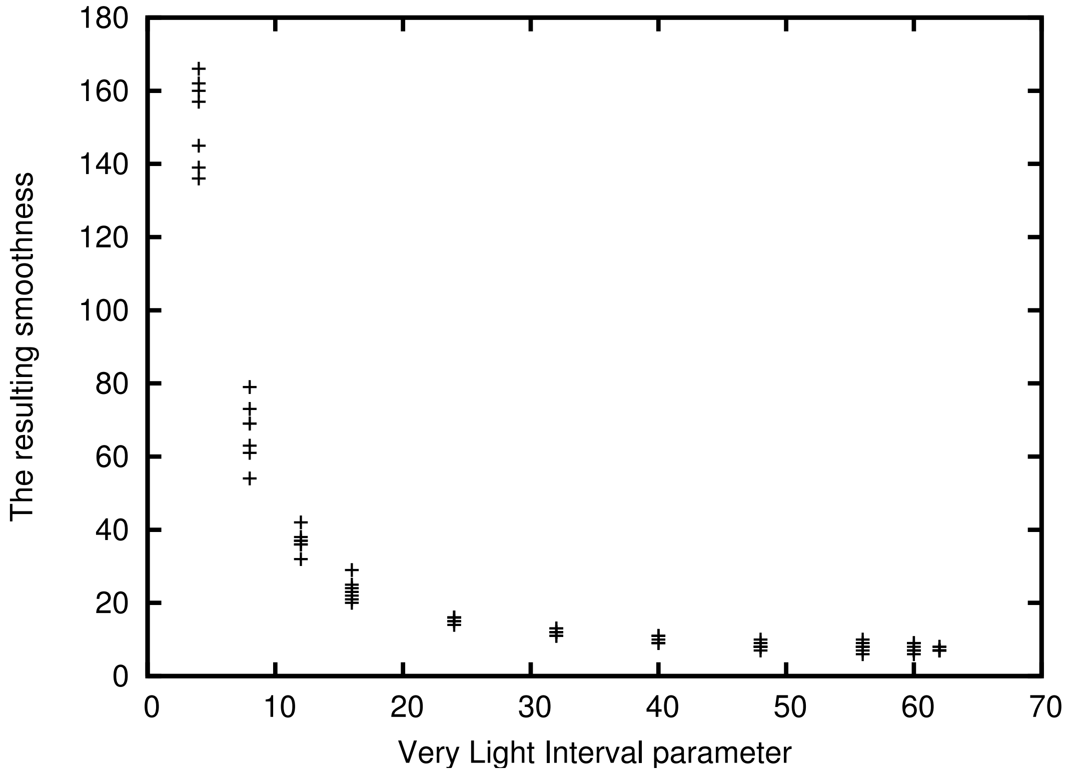

Figure 4.2.: Smoothness according to the weight of the very light interval

The last set of tests for the static scenario was developed to optimize the distance to which a proposal should be forwarded. This time the output of the analysis was not smoothness but the time it takes for the scheme to stabilize. From Figure 4.3 it can be seen that already very short forwarding gives good results. We chose to forward the proposal through 16 nodes.

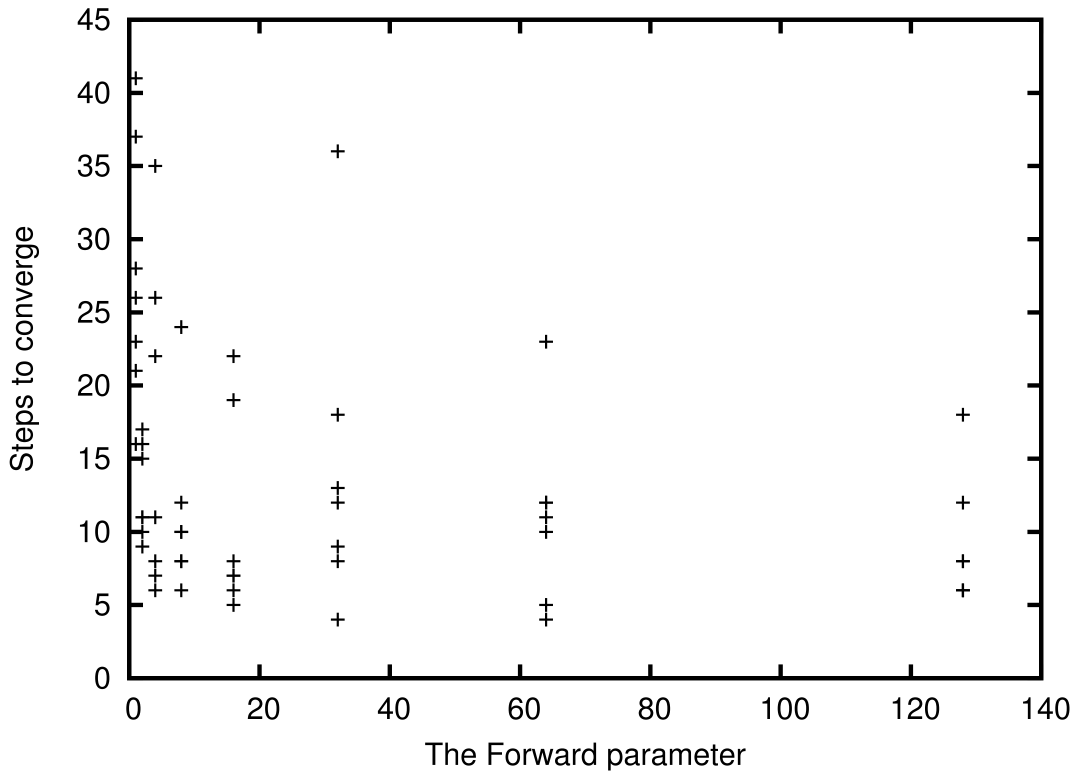

Figure 4.3.: Convergence according to the forward parameter

To test the behavior of our algorithm in a dynamic scenario, we took the following model from [PRU01]. In each time step the number of nodes joining at this time step is chosen according to the Poisson distribution with mean λ, and for each node its time to stay in the system is chosen according to the exponential distribution with mean μ. We tested the behavior of the system for two cases. In the first one these parameters were set to $(\lambda, \mu) = (10, 100)$, which yields the expected number of nodes present in the system at any time step equal to $\lambda \cdot \mu = 1000$. In the second case the parameters were $(\lambda, \mu) = (100, 100)$ yielding the expected number of nodes present in the system equal to $\lambda \cdot \mu = 10000$. In both cases the smoothness never exceeded 14 and was at most 9 in 97% of time steps. For comparison, if we insert the nodes uniformly and independently at random the smoothness varies approximately between 2500 and $3 \cdot 10^4$ for 1000 nodes. For 10000 nodes it is usually approximately between $3 \cdot 10^4$ and $7 \cdot 10^6$.

CHAPTER 5

Bounding The Communication Cost

In this chapter we describe a communication cost efficient algorithm ALGDYN . This scheme reacts slower to the changes taking place in the system but its communication cost can be amortized against the communication cost of keeping the network connected. Similarly to ALGSTAT , ALGDYN can start with arbitrary placement of nodes on the ring but there are a few differences in the model describing the system. First of all, the system is dynamic and the rate at which nodes join and leave the system is chosen by an oblivious adversary. Secondly, the model of synchronicity we use for ALGDYN is much weaker than the one used for ALGSTAT . We still assume that all nodes have the same speed up to constant factors and that messages sent between neighbors are guaranteed to reach their destinations in a bounded time. However, this time each node works for itself and it can even happen that some nodes perform the scheme while others sleep. Last, we analyze the behavior of the algorithm in a bounded time interval which has length equal to half-life. We prove that in this time interval each node generates traffic of at most $O(\log^2 n)$ neighbor-to-neighbor messages. This is at most by a logarithmic factor larger than the communication cost of keeping the system connected. We show that ALGDYN repairs the imbalance present in the system in the beginning of the considered time interval. The imbalance introduced by the adversary during the interval has to be coped with later.

5.1. The load balancing algorithm ALGDYN

In Chapter 3 we classified nodes according to their weights and said that the algorithm ALGDYN would reorganize the positions of nodes on the ring, so that the resulting intervals have weights between $w_{\min} \cdot \delta$ and $32 \cdot \delta$. By Fact 3.6, this implies that in the end the smoothness is constant, with high probability.

The algorithm minimizes the weight of the heaviest interval, but we also have to take care that no interval is too light. We do a similar thing to what we did in the self-stabilizing scheme, that is, we make all nodes responsible for very light intervals leave the network. They still keep their markers, so that the total weight stored in the system is unaffected. Such nodes are still called very light nodes, even though they do not actively participate in network construction anymore, i.e. none of them is responsible for any part of the $[0, 1)$ ring.

Each of the very light nodes knows its virtual position on the ring and contacts the network through the node responsible for that position. We assume that the node is not heavy, since it could immediately split using the very light node. If very light nodes choose their virtual positions uniformly at random, they increase the in-degree of the nodes through which they communicate at most by $O\left(\frac{\log n}{\log\log n}\right)$, with high probability.

The routine works differently for different nodes, depending on the node's interval weight. Except for the classification of nodes according to the weights of their intervals we also distinguish two types of light nodes. In order to be able to bound the communication cost we divide the light nodes into two groups: *active* and *passive*. The pseudocodes for different types of nodes are listed at the end of the chapter. They are all meant to run in infinite loops, so that before every loop execution the current weight of a node is checked in order to decide which part of code should be run.

The most important part of the algorithm is its behavior for the active light nodes. The procedure which active light nodes execute periodically can be divided into three main parts. The first part is a locking protocol in which it is settled which nodes are allowed to participate in the second part. The second part relies on sending help proposals and possibly migrating and the third part decides how long a node should stay active.

An active node, in order to send a help proposal, has to make sure that its predecessor will overtake its job in case the proposal is accepted – it has to *lock* the predecessor for the time of sending the proposal. Nodes use the following types of messages in order to communicate with their successors and predecessors. A node i tells a predecessor that i would like to send a help proposal in this turn and also lock the predecessor by sending a `lock-request` message. In order to tell the predecessor that i is not willing to send help proposals at the moment, i sends a `no-lock-request` message to *i.pred*. To inform the successor about the decision, i sends a `lock-allow` or `lock-deny` message in order to allow or deny the successor to lock i, respectively. If a node i succeeds to lock its predecessor and i sends its help proposal, it has to inform the predecessor about it. It does so by sending an `unlock` message to it.

The main part of the locking protocol is defined in lines 1 – 12. Every time the procedure is executed, each node makes a random decision if it should try to send a help proposal or let its successor send one. Each of these possibilities has probability $\frac{1}{2}$. After making the decision, a node informs its predecessor and successor about it (lines

5 – 6 and 8 – 9) and waits for their answers. In this point, the scheme syncrhonizes itself, that is, each node waits for messages from its neighbors, independently of its decision (lines 5 – 7 and 10 – 12) and also independently of the current weights of its neighbors.

For each node there are a few possibilities, depending on the decisions of itself and the decisions of its neighbors. The most important two possibilities are when the decision of the considered node *i* is compatible with the decision of the proper neighbor. One case is when the node *i* decides to try to send a help proposal and its predecessor *i.pred* allows it to; this is the code in lines 18 – 30. The other case (symmetrical to the previous one) is when the node *i* decides to allow the successor to send a help proposal and the successor wants to send it; this is the code in lines 13 – 17.

In case when a node decides to let its successor send a help proposal and the successor wants to do it, the node is not allowed to take any actions while the successor is looking for a heavy node. That is because the node may have to overtake the job of its successor when it finds a heavy interval. Therefore in lines 14 – 17 the considered node waits for an information from the successor that the successor has finished its search. However, the node is not allowed to ignore messages from its predecessor during this period of time. Therefore, whenever any kind of message comes from its predecessor, the considered node always sends a message telling the predecessor that our node is not willing to send help proposals at the moment.

If a node makes a decision to send a help proposal and it has a permission from its predecessor, it is also careful not to ignore messages from its successor. Whenever, during the procedure of looking for heavy intervals, a message from the successor comes, the node answers that it does not allow the successor to send any help proposals. The main part of the algorithm is the search for a heavy node and possible migration in order to help. This is done by choosing a random position on the ring (in line 21) and the node *j* responsible for it (in line 22). If it is possible to migrate to exactly the chosen position, the node *i* does so and if it is not, it checks a logarithmic number of successors of *j*. If any of them is heavy, *i* migrates to the middle of its interval (lines 26 – 28). Independently of the success of the search, in the end the predecessor of *i* is unlocked (line 29).

The last part of the algorithm for active light nodes is responsible for bounding the number of sent help proposals. Every time the node *i* sends a help proposal, the number of help proposals which still have to be sent is decreased and checked if it fell down to 0. If so, the node becomes passive (lines 30 – 32).

The part of the algorithm responsible for handling passive light intervals is much simpler, since they do not take active part in sending help proposals. The algorithm may be divided into two blocks. The first one (lines 1 – 9) assures that active light intervals do not fall into deadlocks waiting for messages from neighbors. The second part is responsible for activating a passive node. A passive light node becomes active whenever the set of the markers it stores, changes (lines 10 – 12). When it becomes active, the initial number of messages it wants to send is set to maximum, that is to the constant $\mathcal{A}$. This is done in lines $10 - 12$.

A passive node i does not want to send help proposals but does not mind overtaking the load of its successor. For the first reason, whenever its predecessor awaits a message, the node i tells it that i does not want to lock it (lines $1 - 2$). If the successor of i wants to migrate, i wants to enable it to. Therefore, it sends a message letting the successor send help proposals and waits until it has finished (lines $3 - 7$). If the successor sends a non-requesting message, i only answers because the successor awaits an answer (lines $8 - 9$).

The only task of nodes responsible for medium and heavy intervals is to properly answer the requests of their neighbors when they await an answer. Neither medium nor heavy intervals want to send help proposals or overtake the job of their successors. Therefore, when a message comes to a medium or heavy node i from its predecessor it is answered with a message telling it that i does not want to send help proposals. When a message from the successor of i comes, the successor is not allowed to send help proposals. It is important that a requesting neighbor is always answered to avoid deadlocks in the communication protocol. Active light nodes check weight of their predecessors but it might happen that this weight grows afterwards.

5.2. The performance and dynamics model

We model an oblivious adversary, against which the cost of the algorithm is measured in the following way.

1. The adversary generates a pattern of insertions and removals of nodes in any way she wants as long as she does not measure the performance of our algorithm. Thus, she can generate the worst possible starting situation in an adaptive way, in the sense, that she sees the random choices of ALGDYN .

2. The adversary chooses a moment in time t_0, at which she wants to start the measurement. The number of nodes at time t_i for $i \geq 0$ is denoted by n_i. As mentioned above, the distribution of the nodes on the ring at t_0 can be arbitrary. However, it has to be an output of our ALGDYN , so there are no very light intervals in the system.

3. The adversary chooses the patterns of insertions and removals for the current phase. She can decide about insertions online in an adaptive way, that is the number of nodes inserted at time i may depend on the behavior of the algorithm up to time $i-1$. The removals are generated in an oblivious way, i.e. the adversary decides about Time To Live (TTL) of each node at the moment of its insertion; for

the sake of the analysis, the TTLs of the nodes present in the network at time t_0, are assumed to be generated at t_0. The algorithm does not know the TTLs.

4. At every time step nodes are inserted and removed according to the pattern chosen by the adversary. The algorithm responds with communication and possible migrations of nodes.

5. The phase ends after a half-life, that is when the adversary either inserts n_0 new nodes or removes $\frac{n_0}{2}$ of the nodes present in time step t_0. (τ_{t_0} time steps). The adversary is switched off and the algorithm is allowed to communicate and migrate in an extension of $\Theta(\log n)$ time steps.

In order to precisely express the performance of the algorithm, we consider the following virtual process after the half-life and the extension have ended. For each node, which left during the considered half-life, we re-insert it into the network in its last position, provided that such an insertion does not create a very light interval. We perform the re-insertions sequentially in any order, so that there are no ambiguities with parallel checks. In Theorem 5.1 we prove that the system is balanced, i.e. the smoothness is constant, after such reinsertions. Thus, our algorithm copes with the imbalance present in the network at the beginning of the half-life. The imbalance created by the adversarial deletions of nodes will be repaired in the next half-life.

5.3. Performance and communication cost of AlgDyn

We prove the following theorem characterizing the communication cost of AlgDyn in a half-life and stating how well the algorithm rebalances the system. In order to be able to bound the communication cost, we set $\mathcal{A}$ (the number of messages a node sends after it is activated) to be a sufficiently large constant.

Theorem 5.1. *In the defined model* AlgDyn *fulfills the following properties, both with high probability.*

1. *The neighbor-to-neighbor communication cost of* AlgDyn *in the considered half-life is at most $O(\log^2 n)$. This is at most by a logarithmic factor larger than the cost of keeping the network connected.*

2. *If we re-insert the nodes removed in the current half-life as described above, smoothness is constant at the end of the half-life.*

For further analysis we need the following claims about the behavior of the network in a half-life.

Lemma 5.2. *Denote the number of nodes in the network in the beginning of a half-life by n_0, the number of nodes present at time t by n_t, and the number of markers at time t as m_t, for $t \geq 0$. Then during the considered half-life:*

1. *the number of nodes in the network at time t is bounded by $\frac{n_0}{2} \leq n_t \leq 2 \cdot n_0$,*
2. *the number of markers in the network at time t is bounded by $\frac{n_0 \cdot \delta}{2} \leq m_t \leq 2 \cdot n_0 \cdot \delta$,*
3. *the number of insert / delete operations ranges from $\frac{n_0}{2}$ to $\frac{5 \cdot n_0}{2}$.*

Proof. Let the nodes present at time t_0 be called *old* and the nodes inserted during the half-life *new*. Before the half-life ends the adversary can remove at most $\frac{n_0}{2}$ old nodes and insert at most $2 \cdot n_0$ new nodes. Thus, the total number of nodes ranges from $\frac{n_0}{2}$ to $2 \cdot n_0$ and the number of markers ranges from $\frac{n_0 \cdot \delta}{2}$ to $2 \cdot n_0 \cdot \delta$.

It is straightforward that removing $n_0/2$ nodes is the fastest way to end the half-life, whereas inserting $n_0 - 1$ new nodes and removing them and $n_0/2$ old ones is the best way for the half-life to last long. Thus, the last statement follows. ■

Using Lemma 5.2, we can prove the communication-competitiveness of AlgDyn .

Proof of Statement 1 **of Theorem 5.1.** For the exact analysis we modify our approximation and balancing algorithms slightly. Instead of δ markers, each node inserts $\delta/2$ insertion-markers and $\delta/2$ deletion-markers into the system. If a passive light node i stores a marker of a node j then i is activated in two cases: if j joins the network and the marker is an insertion-marker or if j leaves the network and the marker is a deletion-marker. Since the adversary is oblivious, we may think that the deletion-markers are generated at the moment when a node leaves the network. Thus, when a node joins or leaves the network, it contacts $\Theta(\delta)$ places chosen uniformly at random at the moment in which the operation happens. Since, from Lemma 5.2, the total number of joins and leaves in a half-life is $\Theta(n)$, the total number of contacted places in a half-life is $\Theta(n \cdot \delta) = \Theta(n \cdot \log n)$, all of which are independent.

There are a few types of communication cost that each node incurs. The first one, we consider is the cost of inserting or removing markers when a node joins or leaves the network. Each node inserts or removes δ markers in such a situation and contacts δ random places. If $\delta = O(\log n)$ this means total cost of $O(D \cdot \delta) = O(\log^2 n)$ direct (neighbor-to-neighbor) messages.

The most direct cost incurred in the process of load balancing is the cost for sending help proposals. Each node sends at most $\mathcal{A} \cdot O(\delta) = O(\log n)$ proposals to random places, with high probability. Each of these messages is sent to a distance of $O(D + \log n) = O(\log n)$, so the total neighbor-to-neighbor cost is $O(\log^2 n)$.

One of two indirect cost types is when a node tries to communicate to its predecessor but the predecessor is busy sending a help proposal and always answers with `lock-deny` messages. We amortize this cost against the cost of sending help proposals by the predecessor. This means that for each node, this cost is $\mathcal{A} \cdot O(\delta) \cdot O(D + \log n) = O(\log^2 n)$

Another indirect communication cots is when an active light node i and its active or passive light predecessor take random decisions. The probability that as a result of these decisions, i sends a help proposal is at least $\frac{1}{4}$ and the total number of help proposals i is going to send is $O(\delta) = O(\log n)$. With high probability, each successful decision making is preceded by at most $O(\log n)$ unsuccessful ones. Thus, with high probability, the number of unsuccessful decision making messages is $O(\delta \cdot \log n) = O(\log^2 n)$

The total neighbor-to-neighbor communication cost incurred by a single node is $O(\log^2 n)$ and by Theorem 1.2 each node has to receive $\Omega(\log n)$ messages in a half life, so the communication cost of the approximation and balancing algorithms is at most $O(\log n)$ times larger than the cost of keeping the system connected. This finishes the proof of the first statement of Theorem 5.1. ■

It remains to prove that such communication is still sufficient to keep the system balanced. First, we show that the number of help proposals in a half-life is sufficiently high.

Lemma 5.3. *There exist $\frac{n_0}{16}$ light nodes alive during the whole half-life which send at least $\mathcal{A}$ help proposals and still are not accepted anywhere. With high probability, a fraction $\frac{1}{3}$ of the help proposals they want to send is sent in the half-life or in the extension.*

Remark 5.4. *If our model was static and we considered only one half-life and an extension, we could prove that each node can finish its communication in logarithmic time. In the model we consider it is possible that a node is active much longer than the duration of the half-life, even with the extension.*

Proof. There are at least $n_0/2$ nodes which are alive during the whole half-life - we call them *immortals*. Since an interval is light when its weight is below $16 \cdot \delta$ and, by Lemma 5.2, the total weight in the network is at most $2 \cdot n_0 \cdot \delta$, at most $n_0/8$ nodes are not light, and thus at every time step at least $\frac{3}{8} \cdot n_0$ immortals are light.

Consider the distribution of the immortals in the beginning of the half-life and ignore other nodes for a while. We couple the immortals in the following way. An arbitrary immortal is chosen as the first element of the first pair and its successor is the second element of the first pair. Consecutive immortals (according to the order on the ring) build consecutive pairs. We want to lower bound the number of pairs with two light nodes. There are $n_0/4$ pairs, at most $n_0/8$ of which can have at least one medium or heavy node. Thus, in at least $n_0/8$ pairs, an immortal will be willing to migrate when it is activated. Notice that presence of other nodes can only decrease the weight of some immortals and cannot discourage any immortals from sending proposals.

As a node only wishes to share its load with another node, if its current weight exceeds $32 \cdot \delta$, $n_0/16$ nodes suffice to store all load. Thus, only $n_0/16$ immortals have a chance to migrate and at least $n_0/16$ send the maximum number $\mathcal{A}$ of messages.

Assume that in order to send a message, an active light node does not have to lock its predecessor. Then all $\mathcal{A}$ messages sent by a node will be sent in time $O(\log n)$. Suppose that the part of the algorithm responsible for trying to lock a neighbor is performed immediately. Then at every time step, if an active light node is not sending a message, it means that either it cannot lock its predecessor or that it has been locked by its successor. In either case it implies that one of its neighbors is sending a message. This means that at least $\frac{1}{3}$ of the messages will be delivered in the time, in which all messages would be delivered if nodes could send messages without locking their predecessors.

We assign additional $\Theta(\log n)$ time steps to each active light node. Each time there is a chance for an active light node to synchronize with its predecessor and the synchronization does not take place, we pay for such waste of time from this additional cost. There are two reasons why the synchronization should not occur. One is that the considered node decides to try to let its successor lock it. Another is that the predecessor of the node decides not to allow the node to lock it. If within the $\Theta(\log n)$ additional time steps a node succeeds $\mathcal{A}$ times, it manages to send all its messages. At each trial the node succeeds with probability at least $\frac{1}{4}$. With high probability, there is at least one success in $\Theta(\log n)$ time steps, and thus also $\mathcal{A}$ successes in $\Theta(\log n)$ trials. Also, all nodes use $\Theta(\log n)$ additional time steps to succeed $\mathcal{A}$ times, with high probability. This finishes the proof of Lemma 5.3. ■

Below we prove the second part of Theorem 5.1. As writing all constants exactly significantly decreases the readability of the proof, we simplify it by assuming that nodes operate on lengths instead of weights and that an interval is considered very short, if its length is below $\frac{1}{n}$. This assumption is justified by Fact 3.6.

Proof of Statement 2 **of Theorem 5.1.** From Lemma 5.3, we know that in the half-life (or in the extension guaranteed by the definition of the model) $\Theta(n)$ proposals from immortals are generated in the system, and that we can make the constant in the Θ notation arbitrarily large just by increasing the $\mathcal{A}$ constant in the algorithm.

Let $y \in [0, 1)$ be an arbitrary real number on the ring. Consider the interval I_y of length $\frac{\log n}{n}$ starting in y. Let x and z be equal to $y - \frac{\log n}{n}$ and $y + \frac{\log n}{n}$ respectively. I_x and I_z are intervals of the same length as the length of I_y starting in x and z, respectively. In other words, I_x and I_z are the intervals preceding and succeeding I_y, respectively.

Chernoff bound guarantees that, with high probability, the number of help proposals from immortals that fall into I_x is $\Theta(\log n)$. We say that a proposal *reaches* z, if it contacts the node responsible for z. The parameter $\mathcal{F}$ is chosen so, that even if between x and z there are as many nodes as possible, a proposal travelling from x and forwarded through all the nodes can reach z.

At this point of the analysis we allow the adversary to be adaptive. She can choose, which of the mortal nodes die and when. When she removes a node from the network, another node can become heavy, and it should be split by an immortal. Our analysis is based on the fact that when an immortal helps a heavy node, it does it at least until the end of the half-life. The adversary may remove some nodes from I_y after all proposals from immortals have been forwarded through I_y and the algorithm repairs such imbalance in the next half-life.

Below we show that the following event happens at least $2 \cdot \log n$ times: a help proposal that hit I_x reaches z at the moment when there is a node in I_z. If such an event happens, then either there are no heavy nodes in I_y, or the immortal is inserted somewhere between y and the node in I_z. At most $2 \cdot \log n$ immortals can be inserted in the interval $I_y \uplus I_z$, so after $2 \cdot \log n$ such events there are no heavy nodes in I_y.

First, we show that $\Theta(\log n)$ of the proposals reach z in the process of forwarding (not necessarily when there exists a node in I_z). This is true, since in I_x and I_y there is place for at most $2 \cdot \log n$ of them. Notice that we cannot be sure which proposals (in order of coming) reach z: it may happen that the first one gets through, right after that many nodes die in the network, and the next immortals split the heavy intervals that appear in I_x and I_y.

We introduce a notion of a *steady barrier* as the first node behind z on the ring which will survive until the end of the half-life. At the beginning of the process we can bound the distance between y and the barrier by 1. When during our process an immortal reaches z and there is no node in I_z then the immortal is inserted between y and the current steady barrier and becomes the new steady barrier. In the worst case, the distance between y and the steady barrier halves each time when it happens, so after fewer than $\log n$ such events there is a steady barrier in I_z. Thus, $\log n$ immortals can be used to assure that there is a steady barrier in z and further $2 \cdot \log n$ to assure that I_y is balanced.

If we divide the ring into $\frac{n}{\log n}$ disjoint intervals each of length $\frac{\log n}{n}$, then the probability that the algorithm fails for at least one of them is at most n times larger than the probability of failure for a single interval. Thus, the algorithm AlgDyn succeeds on the whole ring, with high probability. ■

Algorithm 3 AlgDyn (an active light node i responsible for an interval I_i)

```
1: if i.pred is not light then
2:     change i to passive light
3: decision ← random({0, 1})
4: if decision = 0 then
5:     send a lock-request message to i.pred
6:     send a lock-deny message to i.succ
7: else
8:     send a no-lock-request message to i.pred
9:     send a lock-allow message to i.succ
10: repeat
11:     do nothing
12: until received messages from i.succ and i.pred
13: if decision = 1 and received a lock-request message from i.succ then
14:     repeat
15:         if received a message from i.pred then
16:             send a no-lock-request to i.pred
17:     until received an unlock message from i.succ
18: else if decision = 0 and received a lock-allow message from i.pred then
19:     if a message lock-request from i.succ comes during execution of lines 22 – 30 then
20:         send a lock-deny to i.succ
21:     p ← random([0, 1))
22:     j ← the node responsible for p
23:     if j is heavy and migrating to p does not produce a very light or light interval
        then
24:         migrate i to the point p
25:     else
26:         contact j and its $\mathcal{F} \cdot \Delta$ successors
27:         if a node k accepts then
28:             migrate to the middle of $I_k$
29:     send an unlock message to i.pred
30:     activity level ← activity level – 1
31: if activity level = 0 then
32:     change i to passive light
```

Algorithm 4 AlgDyn (a passive light node i responsible for an interval I_i)

```
1: if a message comes from i.pred then
2:     send a no-lock message to i.pred
3: if a lock-request message comes from i.succ then
4:     send a lock-allow message to i.succ
5:     repeat
6:         do nothing
7:     until received an unlock message from i.succ
8: if a no-lock-request message comes from i.succ then
9:     send a lock-deny message to i.succ
10: if any change in markers stored in $I_i$ then
11:     i.activity level ← $(A)$
12:     change i to active light
```

Algorithm 5 AlgDyn (a medium or heavy node i responsible for an interval I_i)

```
1: if a message comes from i.pred then
2:     send a no-lock message to i.pred
3: if a lock-request message comes from i.succ then
4:     send a lock-deny message to i.succ
```

Part II.

Transparent Data Structures

Chapter 6

Introduction to Transparent Data Structures

In the second part of this thesis we propose a new class of memory models for implementing data structures so that they can be emulated in a distributed environment in a scalable and robust way. For this we use Peer-to-Peer networks as a basis. We demonstrate the effectiveness of our approach by looking at a specific memory model and implementing a binary search tree in it.

Most of the data structures in the algorithms literature are based on one of the following two basic models: the pointer model or the linear addressable memory model. In the linear addressable memory model we have a linear addressable memory and every read and write request to a memory cell can be processed at unit cost. In the pointer model we only have memory cells and labeled pointers interconnecting the memory cells. Instead of directly addressing the memory cells, all read and write accesses have to be handled via pointers. The standard assumption is that forwarding a request along a pointer can be done at unit cost. For papers on various pointer machine models see, for example [BAG92, CD93, GK89, vEB90].

Both models are useful for the design of data structures for a single machine. However, in a distributed environment both models have their limitations. The basic problem with these models is that they give too much freedom in designing a data structure. This can make it tempting to design data structures that are hard to emulate in a distributed environment in a scalable, efficient, and robust way. *Scalability* means that a low degree overlay network can be used for the emulation of the data structure, *efficiency* means that operations in the data structure can be emulated with a low time and work overhead, and *robustness* means that the data structure can recover from (a potentially large number of arbitrary or random) memory faults. If a data structure is scalable, efficient, and robust, Peer-to-Peer networks should form a very good foundation for it. If it is implemented on top of them, it should be able to use their main advantage, namely be able to access a huge amount of resources.

6.1. Our contribution and organization of the second part

We introduce a class of structured memory models that we call transparent memory models. Transparent memory models are memory models in which memory accesses can be emulated in a scalable overlay network with constant work overhead. The transparency of the proposed models means that it should be possible to design data structures without detailed knowledge about the underlying network.

As a specific example, we introduce a transparent memory model called the hypertree memory model, which is based on the structures of a binary tree and a hypercube. We design a scalable, dynamic overlay network that can emulate memory accesses in this model with constant work. The network is based on the Continuous-Discrete Approach [NW03a, NW03b] with its de Bruijn links but has additionally hypercubic links like in Chord [SMK$^+$01]. The model is designed specifically to emulate a binary tree but it is sufficiently general to support other data structures, e.g., heaps.

We show how to implement a search tree in the hypertree memory model with the property that the amortized work for insert, delete and search is the same as for the best search trees in the pointer model, and additionally it can efficiently recover *all* remaining information under *arbitrary* memory faults. Moreover, in the emulation, every node in the dynamic overlay network only has to perform a worst-case logarithmic amount of work for any insert, delete or search operation.

In the rest of this chapter, we describe two classic memory models in the context of faulty memory cells and robustness. We discuss why the linear addressable memory model and the pointer model are inadequate for achieving scalability, efficiency, and robustness in a distributed environment. Then, we give a detailed definition of *transparent memory models*. We also describe the literature on search structures, as our example is a binary search tree. In this description we concentrate on the robustness aspect of data structures.

In Chapter 7 we define the hypertree memory model, which is a complete binary tree with a set of hypercubic connections connecting nodes of each level. Such structure can be emulated using de Bruijn connections for tree edges and Chord like connections for the hypercubic edges. Using the Continuous-Discrete Approach, we develop a dynamic graph which can emulate the hypertree with dilation at most 1, i.e. for each pointer in the hypertree, the memory cells it connects are stored on the same node or on nodes neighboring in the emulating network.

In Chapter 8 we present a specific application of the hypertree model, namely a balanced binary search tree. The balance in the tree is achieved by assuring that for each node the sizes of its left and right subtrees differ by only a constant factor. This assures logarithmic height of the tree. The balancing of the tree is performed by procedures similar to the procedures used to balance AVL and red-black trees, but in our model moving a subtree incurs a cost proportional to its size and takes time proportional to its height. With careful analysis we can show that the amortized cost of all search tree operations is logarithmic per operation and that each single operation takes at most logarithmic time.

6.2. Linear addressable memory model in a distributed environment

Consider the problem of emulating a data structure in the linear addressable memory model in a distributed environment. Since in this model it may be hard to predict the access pattern without knowing the input in advance, a universal shared memory implementation may have to be used. A vast number of shared memory platforms have already been developed. In the context of Peer-to-Peer networks one can easily adapt the Distributed Hash Tables like Chord [SMK+01], CAN [RFH+01], Pastry [RD01], and Tapestry [HKRZ02] to build a scalable shared memory platform.

However, the problem with any scalable universal shared memory implementation is that it has an inherent (time and work) overhead of essentially $\Omega(\log n / \log\log n)$. This is because by definition, a scalable solution requires the shared memory to be distributed in a system interconnected by an overlay network of at most polylogarithmic degree. The diameter of a network of polylogarithmic degree is at least $\Omega(\log n / \log\log n)$ and if a data structure and a network are not adjusted to one another we have to assume that this is the the distance which has to be traveled at each access in the data structure. The time overhead may be reduced with the help of pipelining techniques but reducing the work overhead can be very difficult. It usually requires knowledge about the access structure so that parts of the data structure that have a high access correlation (such as a subheap in a heap) can be stored together or at a small hop-distance in the distributed system and therefore fewer transmissions of requests over the network are necessary.

However, since in a pure shared memory model there is no incentive for a programmer to implement data structures that have a high degree of locality, there may not be a high access correlation that can be exploited. Providing the programmer with a model in which each memory access costs only 1 unit independently of the access pattern, we have to take into account that the access pattern is arbitrary, even adversarial. Hence, it is not surprising that pure shared memory models have been criticized as being inadequate for distributed systems (e.g. [WWWK94]) and people have considered alternative models including LogP [CKP+93], BSP [GV94], QSM [Ram97], and many others. For a comprehensive discussion of these models we refer the reader to the book by Savage [Sav98].

Robustness can also be a problem. A general technique for obtaining a robust, distributed shared memory platform is to use redundancy. However, redundancy is expensive and cannot protect against all possible bad events that may happen. Therefore, it is important to also consider events in which an item in a data structure may not be recoverable. Here, compact representations of data structures can help. Consider, for example, the array implementation of a heap. If an element in the heap disappears, then a simple reheap operation on the array can repair the heap. For other data structures, a recovery may not be as easy as for the heap, and the linear addressable memory model does not provide a natural way of dealing with it due to its structure-less nature.

6.3. Pointer model in a distributed environment

If we use the pointer model instead of the linear addressable memory model, then we can get rid of some of the problems, such as identifying the access structure. For example, by maintaining a network connection for each pointer in the data structure, we can obtain a faithful (i.e., constant overhead) emulation of the original data structure in the distributed system. However, if the data structure is much larger than the number of processing units available for its emulation, then even though the data structure itself may be scalable, embedding it in the wrong way into the distributed system may result in a non-scalable overlay network for the emulation of the data structure. Embedding a data structure in the right way (i.e., so that each pair of elements connected by a pointer lie at a small distance from one another in the network) is a non-trivial problem in general and has been extensively investigated in the context of network embeddings and partitions in the past (e.g. [Lei92, MS90]). This problem can be removed when embedding the data structure into a shared space, but as mentioned above, a scalable implementation of a shared space has an inherent overhead of $\Omega(\log n / \log\log n)$.

Data structures based on the pointer model can have serious problems with robustness if they are not designed carefully. Pointer structures such as linear lists and trees, for example, are not useful for a distributed environment because a single failure of a memory cell (such as the root of a search tree) can make the whole data structure or a large part of it inaccessible. Therefore, more fault-tolerant pointer structures have been investigated. See, for instance, the work by Aumann and Bender [AB96] or on Skip Graphs [AS03], Skip Nets [HJS+03], and deterministic variants of the skip graph [AS04b, HM04]. Since these pointer structures have a high expansion, they can suffer many memory cell faults and still have a large connected component. However, those parts disconnected from the rest of the structure are lost. Our goal, instead, is to find mechanisms so that *all* information in the data structure that did not get lost due to memory faults can be recovered and reorganized so that the data structure is back in a legal state.

6.4. Transparent memory models

Transparent memory models belong to the class of structured memory models which are defined as follows.

Definition 6.1. *A* structured memory model *consists of:*

- *a countably infinite set U of memory cells,*
- *an infinite family of pointer structures $\mathcal{H} = \{H_n = (U_n, F_n) \mid n \in \mathbb{N}, U_n \subseteq U, F_n \subseteq U_n \times U_n\}$,*

where H_n is a connected graph for all $n \in \mathbb{N}$ and $U = \bigcup_{n\in\mathbb{N}} U_n$. We demand also that $U_n \subseteq U_{n+1}$ and $F_n \subseteq F_{n+1}$.

In such a model, requests can only be exchanged between adjacent cells and every such request can be processed at a unit cost.

Definition 6.2. *A structured memory model is called* transparent *if there is an infinite family $\mathcal{G} = \{G_n = (V_n, E_n) \mid n \in \mathbb{N}\}$ of graphs of at most polylogarithmic degree such that for all $n \in \mathbb{N}$ and $m \in \mathbb{N}$, H_m can be mapped to G_n:*

- *with dilation at most 1,*
- *with load at most $O(\log(|U_m|) + |U_m|/|V_n|)$.*

This means that, for all n and m, there is a mapping $f : U_m \to V_n$ of the cells in H_m to the nodes in G_n such that the number of cells stored in a single node is at most by a logarithmic term larger than average (for all $v \in V_n$, $|f^{-1}(v)| = O(\log(|U_m|) + |U_m|/|V_n|)$), and a connection between any two cells is emulated by an edge or by storing them on the same node (for all $\{v, w\} \in F_m$, $f(v) = f(w)$ or $\{f(v), f(w)\} \in E_n$).

We think that transparent memory models will succeed in implementing of diverse data structures on top of Peer-to-Peer networks. However, we bound our present considerations to only one model, namely the *hypertree memory model*. It turns out that for the hypertree model, there is not only an infinite family of static graphs that can emulate H with dilation at most 1 but we can also specify dynamic graphs (i.e., nodes can continuously enter and leave the graph) that can emulate H with dilation at most 1 at any time.

6.5. Other work on search structures

Work on search structures has a long history. The most popular search structures are probably AVL trees, red-black trees, skip lists, and splay trees. Whereas all of these structures are efficient allowing to process insert, delete, and search operations with a cost of $O(\log n)$, none of these structures is robust to memory faults.

There are basically two approaches of making search structures robust to memory faults: using high-expansion pointer structures such as Skip Graphs [AS03] or the Hyperring [AS04b], or embedding the search structure in a compact form into an array [AL90, BDFC00, BFJ02, IKR81]. The first approach cannot recover from arbitrary memory faults but can at least make sure that the number of non-faulty entries that get disconnected from the largest connected component of non-faulty entries is within a constant factor of the faulty entries [AW05]. The second approach allows, in principle, recovery from arbitrary memory faults, but it is not clear how much time would be needed for that. But even if there were an efficient recovery mechanism, the fact that insert and delete operations in these search structures require an amortized work of $\Theta(\log^2 n)$ [AL90, BDFC00, BFJ02, IKR81], which is also believed to be best possible [BDFC00, DZ90, DSZ94], makes them not particularly attractive.

Hence, it is rather surprising that we can demonstrate the feasibility of a search structure with amortized $O(\log n)$ work for insert and delete operations and worst case $O(\log n)$ work for search operations that can recover efficiently to a maximum possible extent from *arbitrary* memory faults.

CHAPTER 7

Hypertree Memory Model

In this chapter we describe a specific memory model, the hypertree model. We demonstrate that the hypertree memory model can be emulated by scalable, dynamic graphs in an efficient and robust way. For this we first introduce a new family of static graphs and then demonstrate how to turn them into dynamic graphs that allow the hypertree memory model to be emulated with dilation 1. This new family, the de Bruijn cube, consists of graphs that are the union of the de Bruijn graph and the hypercube. We need to take the union of these graphs because neither the de Bruijn graph nor the hypercube can emulate our hypertree model with dilation 1 but their combination can. The dynamic version of the de Bruijn cube, a continuous de Bruijn cube, is solely based on the Continuous-Discrete Approach by Naor and Wieder.

In the hypertree memory model, $U = \{0,1\}^*$, i.e., the cells in the model are labeled by binary strings. The structure H interconnecting these cells is defined as follows. (Recall that the *Hamming distance* $H(v,w)$ of any two bit strings $v, w \in \{0,1\}^k$ is equal to $\sum_{i=1}^{k} |v_i - w_i|$.)

Definition 7.1. *A* hypertree $H = (U, E)$ *is an infinite binary tree in which*

- *the root has the label* ϵ *(the empty label),*
- *every node* $v \in V$ *is connected to* $v0$ *and* $v1$ *(the tree edges) and*
- *every node* $v \in V$ *is connected to every node* w *with* $|v| = |w|$ *and Hamming distance* $H(v,w) = 1$ *(the hypercube edges), i.e.,* v *and* w *only differ in one bit.*

We designed the hypertree specifically to support an efficient implementation of a search tree because the tree edges will be needed for the search tree structure and the hypercube edges will be needed for the balancing.

7.1. A family of graphs for the hypertree model

Consider the following two well-known classes of graphs.

Definition 7.2 (de Bruijn). *For any $d \geq 0$, the d-*dimensional de Bruijn graph *$DB(d)$ is an undirected graph $G = (V, E)$ with node set $V = \{0,1\}^d$ and edge set E that contains all edges $\{v, w\}$ with $v = (v_1, \dots, v_d)$ and $w \in \{(x, v_1, \dots, v_{d-1}) \mid x \in \{0,1\}\}$.*

Definition 7.3 (Hypercube). *For any $d \geq 0$, the d-*dimensional hypercube *$HC(d)$ is an undirected graph $G' = (V, E')$ with node set $V = \{0,1\}^d$ and edge set E' that contains an edge between any two nodes $v, w \in V$ with $H(v, w) = 1$.*

When combining these two graphs, we obtain the following new class of graphs.

Definition 7.4 (de Bruijn cube). *For any $d \geq 0$, the d-*dimensional de Bruijn cube *$DC(d)$ is an undirected graph $G_d = (V_d, E_d)$ with node set $V_d = \{0,1\}^d$ and edge set $E_d = E \cup E'$, where $E = \{\{v, w\} : v = (v_1, \dots, v_d), w \in \{(x, v_1, \dots, v_{d-1}) : x \in \{0,1\}\}\}$ and $E' = \{\{v, w\} : v, w \in V, H(v, w) = 1\}$, i.e. E and E' define respectively de Bruijn and hypercubic sets of edges on the same set of nodes.*

Given a finite binary string $b = (b_1 b_2 \dots b_k)$, let $b^R = (b_k \dots b_2 b_1)$ and $\text{prefix}_{k'}(b) = (b_1 \dots b_{\min\{k,k'\}})$. For any two binary strings a and b, $a \circ b$ represents their concatenation. In order to map the cells of any H_d to the nodes of any $G_{d'}$, we use the following mapping $f_{d,d'} : U_d \to V_{d'}$:

$$f_{d,d'}(s) = \text{prefix}_{d'}(s^R \circ r) \quad \text{for all } s \in U_d \setminus \{\epsilon\}$$

where $r \in \{0,1\}^{d'}$ is the bit string the root ϵ is mapped to in f and can be selected in an arbitrary way. Intuitionally, if a node of the H_d is mapped to a node $v = (v_1, \dots, v_{d'})$ of the $G_{d'}$, then its children in H_d are mapped to $(0, v_1, \dots, v_{d'-1})$ and $(1, v_1, \dots, v_{d'-1})$, that is to two of the de Bruijn neighbors of v in $G_{d'}$. This mapping has the following important property:

Theorem 7.5. *For any $d \in \mathbb{N}$, $d' \in \mathbb{N}$ and $r \in \{0,1\}^{d'}$, the embedding of H_d into $G_{d'}$ via $f_{d,d'}$ has a dilation of at most 1 and a load of at most $|U_d|/|V_{d'}| + d + 1$.*

Proof. For every node v of H_d, its edges to $v0$ and $v1$ can be emulated by the de Bruijn edges of $G_{d'}$ and every connection between v and w with Hamming distance $H(v, w) = 1$ in H_d can be emulated by the hypercubic edges of $G_{d'}$.

Every level $0 \leq i \leq d$ of H_d is evenly distributed among the nodes of $G_{d'}$. If $i \leq d'$, then each node of $G_{d'}$ has load at most 1. Otherwise, each node of $G_{d'}$ has load exactly $2^{i-d'}$. If we sum the load over all $d+1$ levels, the maximum load of a node is $|U_d|/|V_{d'}| + d + 1$. ■

Hence, the hypertree model satisfies the properties of a transparent memory model, that is, it can be emulated by a de Bruijn cube with low dilation and also low load as long as the label sizes of the allocated memory cells are not too widely spread.

A particularly interesting property of the de Bruijn cube is that the root of the hypertree model can be placed at *any* position in $[0,1)$. This allows to emulate multiple hypertree memories simultaneously in a load-balanced manner, which is very useful when running multiple applications on top of the same distributed system.

7.2. Dynamic graphs for the hypertree model

Using the Continuous-Discrete Approach of Naor and Wieder [NW03b], one can transform the family of de Bruijn cubes into a dynamic de Bruijn cube suitable for Peer-to-Peer systems. Interpreting every binary label $(v_1, \ldots, v_d)$ as a point $x = \sum_{s=1}^{d} v_s/2^s$ in $[0,1)$, we can formulate the following continuous variant of the de Bruijn cube.

Definition 7.6. *The* continuous de Bruijn cube *consists of the space $V = [0,1)$ and a set of functions $f_0, f_1 : V \to V$ and $g_s : V \to V$, $s \geq 1$ with*

- *$f_s(x) = (x+s)/2$ for both $s \in \{0,1\}$ (which represents the de Bruijn edges) and*
- *$g_s(x) = x \oplus 2^{-s}$ for all $s \in \mathbb{N}$ (which represents the hypercube edges), where $x \oplus y$ is the bit-wise XOR of (the binary representations of) x and y.*

Next we show how to convert this back into a discrete graph, using the Continuous-Discrete Approach of Naor and Wieder [NW03b]. Recall from the first part of this thesis, that for a placement of a set of n nodes on the $[0,1)$ ring, a node i is responsible for interval I_i from itself to its successor $\mathrm{succ}(i)$.

Definition 7.7. *The* de Bruijn cube *$DC(V)$ of a node set $V \subset [0,1)$ is an undirected graph with node set V that contains an edge $\{i,j\}$ for every pair of points $i, j \in V$ with*

- *$j = \mathrm{succ}(i)$ (which connects the nodes to a ring), or*
- *$f_s(I_i) \cap I_j \neq \emptyset$ for some $s \in \{0,1\}$, or*
- *$g_s(I_i) \cap I_j \neq \emptyset$ for some $s \geq 1$.*

Here, we also allow self-loops.

This definition implies the following crucial fact:

Fact 7.8. *For any set of points V and any two points $x, y \in [0,1)$ with $y = f_s(x)$ or $y = g_s(x)$ for some s it holds for the owner i of x and the owner j of y that $\{v, w\}$ is an edge in $DC(V)$.*

As we will see below, the fact implies that the hypertree memory model can be emulated with dilation at most 1 in $DC(V)$ for *any* node set V. In the following, we assume that the nodes in V always have a sufficiently large but finite precision (i.e., bit representation).

7.3. Joining and leaving the dynamic de Bruijn cube

In the following we assume that the network is balanced by the procedures from the first part of this thesis or similar or that at least the nodes choose their IDs uniformly at random. Then, the following theorem shows bounds for the diameter and degree of the network.

Theorem 7.9. *If V is a set of n nodes in $[0, 1)$ forming a balanced network, then $DC(V)$ has both maximum degree and diameter of $O(\log n)$, w.h.p. If the IDs of nodes are chosen uniformly at random, then the diameter is $O(\log n)$ and the maximum degree is $O(\log^2 n)$, w.h.p.*

Proof. The continuous de Bruijn cube is constructed in such a way that the neighborhood of an interval I forms logarithmic (in the number of nodes n) number of intervals of length the same as the length of I, up to constant factors. If the network is balanced, then only constant number of nodes can be responsible for each of these neighbors, provided that only one node is responsible for I. If nodes are distributed uniformly at random, then the length of I is $O\left(\frac{\log n}{n}\right)$, with high probability. Thus, the total length of all neighbors of I is $O\left(\frac{\log^2 n}{n}\right)$, and Chernoff bounds guarantee that, with high probability, $O\left(\log^2 n\right)$ nodes are responsible for the whole neighborhood of I.

In both cases, the shortest interval has length at least $\Omega\left(\frac{1}{n^2}\right)$ and the hypercubic connections assure that the diameter of the network is at most $2 \cdot \log n + O(1)$. ■

Whenever a new node joins or an old node leaves V, we aim at adjusting the connections in $DC(V)$ so that Definition 7.7 is satisfied. Interestingly, this is very easy (see also [NW03b]):

Suppose that a new node $j \in [0, 1)$ wants to join V, and suppose that $i = \text{pred}(j)$, i.e. i is the closest predecessor of j in V. Given that $I_i = [i, k)$, it follows that $I_i^{new} = [i, j)$ and $I_j = [j, k)$. Thus, according to the definition of $DC(V)$, the new edges of i and the edges of j are a subset of the old edges of i, and all edges not associated with a point in I_i remain unchanged. Hence, by routing the join request of j to i and communicating with i and its adjacent nodes, j can be fully integrated into the system.

Suppose that a node j wants to leave V, and suppose that $i = \text{pred}(j)$. Then it suffices that j forwards all of its edges to i before leaving, because i will take over j's region. Hence, we get:

Theorem 7.10. *Given any node set V of n nodes in $DC(V)$ forming a balanced network, it takes at most $O(\log n)$ work to process a join or leave operation, w.h.p. If the IDs of nodes are chosen uniformly at random, then a join or leave operation takes at most $O(\log n)$ work, w.h.p.*

7.4. Robustness

The hypertree space model also has the advantage that, in principle, it allows efficient recovery from *arbitrary* memory faults (as long as the distributed system emulating it can recover). We call a data structure implementation in the hypertree model *compact* if, in the fault-free state, the memory cells of the data structure form a connected component in the hypertree. A data structure is called *recoverable* if it has a recovery mechanism allowing it to recover from arbitrary lost memory cells. Suppose that we use the following strategy whenever a node i establishes a new edge to a node j: *Wake up the recovery mechanism in all memory cells in i.* Then the following result holds:

Theorem 7.11. *Any compact data structure implementation in the hypertree model that can recover from arbitrary lost memory cells can be efficiently emulated by a scalable overlay network so that also in the emulation it can recover from arbitrary lost memory cells.*

Proof. Follows from the fact that the data structure is compact and that any lost memory cell will eventually be detected once the edge in the network emulating a connection to a still working memory cell has been recovered. ■

We use the term "efficient" in the theorem because the recovery mechanism only needs to be invoked in the case that there is an edge change in the network. As long as the network is static, no checks have to be performed (under the assumption that local memory is reliable). This repair procedure will be much clearer in the next chapter when we show its implementation for a specific example, namely a binary search tree.

CHAPTER 8

Search Tree

In this chapter we give a concrete example of a transparent data structure, a binary search tree. The tree has the usual property of a binary search tree that all items stored in a left subtree of a node are not larger than all items from the right subtree. The balance is assured by the property that for each node the sizes of its left and right subtree differ by at most constant factor. This property yields logarithmic height of the tree.

Such definition of the tree makes it possible that a single balancing operation incurs linear cost, but we show how to perform the balancing in such a way that the amortized cost per any tree operation is logarithmic. Such a property is still not satisfying in a distributed environment, as it is also desirable that no operation burdens a single node too much. We show that our balancing procedure needs time proportional to the height of the tree, and thus load of each node is at most logarithmic.

In the end, we describe how the tree can recover from arbitrary failures. Thanks to the logarithmic bound of the height of the tree and the underlying hypertree structure, it is possible to repair the tree in logarithmic time and with cost by a logarithmic factor larger than the number of failures. In the end, the tree looks as though the broken items were simply removed. The balancing and recovery procedures are described independently, that is, in the description of the balancing procedures we assume there are no failures or recovery procedures running and in the description of the recovery procedure we assume the tree is balanced.

We define a balanced binary search tree in the following way.

Condition 8.1. *For each node r of the tree T where the subtrees rooted in the left and right child of r are α and β, respectively:*

1. *r stores at most one entry*
2. *if r stores an entry and r is not the root, also the parent of r stores an entry*
3. *for all $a \in \alpha$ and $b \in \beta$ $a \leq r \leq b$ (i.e., the tree is sorted)*

4. *$\frac{|\alpha|}{3} - 1 \leq |\beta| \leq 3 \cdot |\alpha| + 1$ (i.e., the tree is balanced)*

5. *r stores the weight of the subtree rooted in it*

In addition, the root of the search tree must be at position ϵ, the root of the hypertree model. The first two conditions ensure that the tree is stored in a compact form and that data loss in case of peer failures is kept low. The third condition is the usual search tree condition. The fourth condition assures balance in the tree and gives a logarithmic upper bound on the height of the tree.

In order to perform operations on the tree and maintain the balance in it we use the following primitives, each of them working on a subtree T_v rooted in a node v:

move upwards v is moved to its parent and T_v is moved so, that it remains the subtree rooted in v

move downwards v is moved to its child and T_v is moved so, that it remains the subtree rooted in v

move sideways v is moved to its tree-sibling and T_v is moved so, that it remains the subtree rooted

Lemma 8.2. *Each of the primitive operations:*

- *takes constant time to be planned*
- *takes logarithmic time in the size of the moved subtree*
- *needs communication linear in the size of the moved subtree*

Proof. When r takes a decision to perform a primitive operation it gives proper orders to its children and moves along one edge, which takes constant time and communication since communication with the children and moving upwards or downwards can use the tree edges and moving sideways can use the hypercubic edges of the Hypertree. Similarly, when a node gets an order to move from its parent, it forwards the order to its children (if it has any) and moves along one edge. This also takes constant time and communication. Thus, the total communication cost is bounded by the number of nodes in the subtree and the total time is bounded by the height of the subtree. ■

The basic tree operations are performed as follows. Each of them starts in the root of the tree.

Search(a) If the item in the root is equal to a, the address of the root is returned. Otherwise the Search operation is performed recursively in the left (if a is smaller than the value in the root) or right (if a is larger) subtree. When the subtree does not exist (e.g. when we are in a leaf), the item is reported to be not found.

Insert(a) First, Search(a) is performed. If a is not found, then a is inserted at the proper place below the leaf reached by Search(a). In case a is found, the operation continues recursively to any subtree, as in this case we are allowed to insert it recursively in any of the subtrees.

Delete(a) First, Search(a) is performed. Let x be the found node (if it exists). If it is a leaf, it is simply removed from the tree. If not, let y be the rightmost node in its left subtree (or the leftmost node in its right subtree if the left subtree is empty). We remove a from x and move the item from y to x. The node y was either a leaf or its right (left) subtree was empty. If it was a leaf, we can simply remove it. Otherwise the whole non-empty subtree is moved upwards.

Below we show that as long as the tree is balanced, each of the above operations takes $\log(n)$ time, where n is the initial number of nodes.

Lemma 8.3. *An isolated Search or Insert operation or any m consecutive Delete operations preserve Condition 8.1 (except point 4) in the tree with logarithmic cost per operation.*

Proof. A Search operation does not change the structure of the tree at all. Since the tree fulfills the Condition 8.1 and thus is balanced, its height is $O(\log n)$ and the height bounds the cost of the Search operation.

During an Insert operation a path from the root to the new position of a is traversed. The cost of the operation is bounded by the new height of the tree, which is by at most 1 larger than the old height, which is $O(\log n)$. The new node is inserted as an extension of a path from the root which assures the first two properties. The third property is assured by the way the path is built - in the same way as a Search operation would visit the nodes in the tree. The weights of subtrees have to be updated only for the nodes on the path from the root to the new position of a which can be done at the moment of traversing the path.

Since the Delete operations can only decrease the height of the tree, we can assume that during all m operations the height of the tree is $O(\log n)$. Thus, it suffices to prove that a single Delete operation preserves Condition 8.1 and has cost bounded by the height of the tree. A single Delete operation takes care that no hole appears — when x is removed from the tree and it was not a leaf, its place is filled by another node y which in turn can be removed from the tree without creating another hole. Obviously, no new information is inserted anywhere except into a hole created by removing a node. Thus, the first two properties are fulfilled. If x is a leaf, removing it does not influence the order of other items in the tree in any way. If it is not, y contains the largest item smaller than x (or the smallest item larger than x), so moving it to the place previously occupied by x does not change the order, either. Finally, if y had a subtree, moving it upwards as a whole, only compresses the tree but does not change the order as well. The cost of moving the tree is linear in its size but the latter is constant since the other subtree of y does not exist and the tree is balanced. Updating the weights of subtrees can be done during traversal of the path from the root to y (or to x if x was a leaf). ■

After performing an operation on the tree it is easy to check whether the tree is still balanced, as each node that participated in the operation can check whether its children fulfill the balance condition (Condition 8.1, property 4). If they do not, a rebalancing routine is needed. The simplest way would be to rebalance the whole subtree so that it is perfectly balanced — such approach has logarithmic amortized cost per operation (we omit the proof). However a single rebalancing routine can take linear time, which is unacceptable in a distributed environment. In the next section we show a routine that has logarithmic amortized cost per operation and that performs each rebalancing routine in logarithmic time. The latter implies that each node participating in the network has to send $O(\log n)$ messages during the rebalancing routine. Since no node is overburdened, we call this approach *fair*.

8.1. Amortized analysis

Before we reveal the balancing algorithm, we prepare the ground for it by some definitions. In order to analyze the cost of the algorithm, we use amortized analysis. Whenever our algorithm has to perform a tree operation it is paid some number of virtual coins. The algorithm has to pay a coin for each operation which takes constant time. If the cost of the current tree operation is lower than the number of coins given to the algorithm, the algorithm can save the spare coins in form of potential. If the cost is higher, the algorithm has to use some of its savings. We have to assure that the algorithm is never in debt, that is, its potential is never negative. The number of coins that the algorithm is paid for each operation is its amortized cost per operation.

We define a potential function for each node r of the tree in the following way. The potential of the tree is defined as the sum of potentials of all its nodes.

Definition 8.4. *For a tree T with subtrees α and β the* potential *stored in its root r is the difference between the weights of α and β:*

$$\phi_r = ||\alpha| - |\beta||$$

Obviously, for each node r the potential in r is never negative. Thus, the potential of the whole tree T is never negative either. We prove below that if we can attribute $\Theta(\log n)$ additional cost (in the form of virtual coins) to each Insert and Delete operation, we can use such capital to ensure that the potential in all nodes of the tree is in accordance with the definition. Later, we can use the stored coins to pay for a rebalancing routine.

Lemma 8.5. *For an Insert or Delete operation $O(\log n)$ additional coins suffice to update the potentials in the tree there where it is necessary.*

Proof. As the tree is balanced before the operation the height of the tree is $O(\log n)$. When an Insert or Delete operation is performed, a path P from the root to a leaf is visited. For each node r in P at most one of the two subtrees below r grows or shrinks by one item. Thus, ϕ_r changes by at most 1 and the total potential changes by at most the length of P which is $O(\log n)$. ∎

The balancing algorithm uses rotations, known from other approaches (see for example "Introduction to Algorithms" [CLR89]) for balancing the tree, as a basic routine. We do not define rotations as a written description would not be as clear as pictures. The reader is encouraged to check that the tree depicted in Figure 8.1 can be easily transformed into the tree depicted in Figure 8.2 by a constant number of primitive operations. The same holds for transforming the tree from Figure 8.3 into the tree in Figure 8.4 and the tree in Figure 8.5 into the tree in Figure 8.6.

In Section 8.3 we show a technical *local balancing procedure* with the proof of the following properties.

Lemma 8.6. *If for a node r the subtrees rooted in the children or r are balanced (all nodes fulfill point 4 of Condition 8.1, called the balance condition further on), then the local balancing procedure rebuilds the tree so that:*

- *r fulfills the balance condition*
- *on each path from r to a leaf at most one node does not fulfill the balance condition*
- *only a constant number of primitive operations are used*
- *the potential stored in the tree decreases and the total communication cost is at most by a constant factor larger than the decrease in potential*

For a node r the local balancing procedure checks 3 levels of descendants of r and, depending on the weights of the trees rooted in them, it performs some rotations. If some other balancing procedures are currently being executed lower in the tree they do not influence the procedure in r. It is only important that no node from the highest 3 levels is participating in any rotations. The trees below the third level are only moved as a whole and individual nodes of such trees are moved after they have finished the rotations started previously.

The balancing of the tree is done in a bottom-up-down fashion, that is it is started in the leaves, it follows in the direction of the root and comes back to the leaves . If imbalance appears in the tree rooted in a node r, r is informed about it by one of its children and it happens after the child has finished the balancing. If r does not fulfill the balance condition at that point in time it performs the necessary rotations and tells those of its descendants that lost their balance property that they should run the balance procedure. Lemma 8.6 guarantees that each of these descendants can run the procedure without delay, since all subtrees below them are balanced. After constant number of time-steps all children and grandchildren of r fulfill the balance property and r can inform its parent about the change in weight and tell it to start the balancing.

The following theorem states the main result of this chapter, namely it shows that the proposed tree maintenance algorithm works with logarithmic amortized cost per tree operation and that each tree operation needs at most logarithmic time to finish.

Theorem 8.7. *The total communication cost of the balancing procedure is $O(\Delta_\phi + m \cdot \log n)$ and the time of the balancing procedure is $O(\log n)$ where Δ_ϕ is the total decrease of potential, n is the initial number of nodes in the tree, and m is the number of Insert or Delete operations executed right before the balancing.*

Proof. The communication cost is twofold: the cost of executing the primitive operations and the cost of child-parent communication to inform about possible weight-changes and to give orders. According to Lemma 8.6 the total cost of all primitive operations is $O(\Delta_\phi)$. If a node communicates to its parent it informs about the change of weight and the total number of such messages is m each travelling a distance of $\log n$. If a node communicates to its descendants it does so only after it has performed a local balancing procedure (which decreased potential) and it sends a constant number of messages. Thus, the total number of such messages is $O(\Delta_\phi)$.

The balancing procedure starts in the leaves of the tree and each node needs only constant time to initiate the necessary primitive operations and let its parent start the balancing. After $O(\log n)$ steps, each node has initiated the balancing. It takes constant time in each node and ends after reaching the leaves, i. e. after $O(\log n)$ time steps. ■

8.2. Recovery

In this section we describe how our search tree can recover from arbitrary node failures. We assume the simplest possible model of failures defined as follows. All failures happen at a single time step and there are no failures during the recovery. Also, no failures happen during load balancing, that is, the tree is balanced at the moment when failures happen. The system as a whole is synchronized, i.e. we may assume that communication takes place in discrete time steps and that in a single time step a message sent from neighbor to neighbor is guaranteed to reach it. In the following we use the term *nodes* to denote peers building the de Bruijn Cube and the term *items* to denote values stored in a search tree.

It should be clear that failures happen in nodes building the system and as a result all items stored in an interval for which a broken node was responsible are lost. When a node takes over a previously broken interval it does not know if there were items belonging to any tree in this interval. Still, the algorithm has to repair the structure of all trees which stored their items in this particular interval.

The recovery proceeds in three phases. In the first phase new special items are created in the places of lost items. Instead of a key, a node can store a *hole* which means that it should get an item to store in the later phases. We create holes only on paths from survivors to the root, where survivors are the items that were not broken in the failure. If a *whole* subtree (in particular a single leaf) disappears we do not reconstruct it. Thus, the first phase is responsible for connecting all parts of the *original* tree (from before the failure) and forming an *intermediate* tree, which is a minimum subgraph of the original tree in which each surviving item has all ancestors. After building the intermediate tree, we start the second phase in which we plan how to fill in the holes. This is done in a top-down fashion. If the root of a subtree contains a hole, it sends a request to fill it to one of its subtrees and then runs the fill-in procedure in each child, provided that the subtree rooted in the child will still contain some items after serving all the requests from its ancestors. In the third phase, the items which were ordered to fill-in the holes, do so by travelling upwards. In the end the rebalancing procedure is run in the root of the tree, but this is no longer a part of the recovery process.

As mentioned in Chapter 7, with a proper wake-up mechanism in the de Bruijn Cube emulating a hypertree, we can assume that nodes which take over the role of the nodes that failed, run a wake up mechanism in the newly acquired intervals. When a node discovers that its successor on the ring is no longer present in the system it takes over its job. It cannot access the items previously stored in the newly acquired interval as they are irrevocably lost but it can contact the intervals responsible for the parents and children of any items previously stored there. If there are items in the neighboring (along the de Bruijn edges) intervals which had tree neighbors in the restored interval, we start the wake up procedure in these items. If not, in particular if the parent or child interval has also just been restored, we do nothing. Such procedure starts the wake up mechanism in each item that survived the failure but the parent or at least one child of which was lost.

Each woken item has information about the original weight of its subtree and knows the address of the root of the tree. Basing on the information about the weight of its subtree it computes an estimate of where a leaf could have been below. If the weight is

w, a good estimate is any number between $l = \frac{1}{2} \cdot \log w$ and $h = 3 \cdot \log w$ (the former is an exact value guaranteed by the balance of the tree, the latter is a bound for $\log_{4/3} w$, as in a balanced tree all leaves have to be on such heights. A woken item sends two messages upwards, one of which can reach the root of the tree very fast and the other carries more information. The first message is called *fast* and the second *slow*. One purpose of the fast messages is to wake up the whole intermediate tree, so that each item knows about the failure. Another purpose is to inform every item if it should expect slow messages from its children. The fast messages also carry information about the time when the failure happened. The slow messages carry information about the updated weight of the tree (the number of survivors in each subtree), but each item has to be sure it has gathered this information from its whole subtree before sending the information to its parent. Below we describe the first phase in details.

Each item that woke up after the failure immediately sends a fast message to its parent and if the parent does not exist, a hole is created in its place. Each item that receives a fast message from one of its children and has not sent a fast message to its parent, sends a fast message immediately too. The following lemmas state the properties of the fast messages that we will need later. The expression *possible location of a leaf in the subtree of an item* means that for the original weight of the subtree rooted in the considered item, there exists such balanced distribution of items in the subtree that there is a leaf in the mentioned location.

Lemma 8.8. *All fast messages are sent within* $3 \cdot \log n$ *time steps from the moment of the failure. After this time, each item of the intermediate tree knows a possible location of a leaf in its subtree.*

Proof. The height of the original tree is smaller than $3 \cdot \log n$, as the tree is balanced. Each survivor can compute information about a leaf in its subtree from the information about the weight of the subtree. For a broken item we consider two cases. Either its whole subtree was broken and it does not belong to the intermediate tree, or at least one of its descendants has survived and will send an information about a leaf in its subtree. Such information is also valid for the considered item.

If an item has received a fast message from one of its children, it needs one time step to send it to its parent, so the height of the tree is an upper bound for the time in which fast messages are sent. ■

Lemma 8.9. *If an item knows (either because it survived or because it received a fast message from one of its children) that there is a leaf at height h in its subtree then fast messages from its children come within* $6 \cdot h$ *time steps or never.*

Proof. If there is a leaf in the subtree of the considered item at height h, it means that its weight before the failure was at most 4^h, as in every balanced tree of weight w the shallowest leaves are at height $\frac{1}{2} \cdot \log w$. On the other hand, in a balanced tree of weight

$w = 4^h$, the deepest leaves are at height smaller than $3 \cdot \log w$, so the height of the considered subtree is at most $6 \cdot h$. If there was at least one survivor and at least one broken item in the subtree, a message from it will come in time bounded by its height, that is $6 \cdot h$. ■

The purpose of the slow messages is to update the information about the weights of subtrees in all items of the intermediate subtree. Each item v acts as follows, concerning the slow messages. When it has an information about the failure together with an information about a leaf at height h in its subtree, it waits $6 \cdot h$ time steps for possible fast messages from its children. If a fast message comes from a child, then v waits for a slow message from this child. The slow message will carry the updated weight of the tree of the considered child. If no fast message comes from a child within the time limit, there are two cases. If a child is alive, we may assume that the whole subtree of the child is alive. If not, we may assume that the whole subtree of the child is dead. In the first case, we can read the information about the weight from the information stored in the child. In the second we know the weight is 0. After calculating the weights of the subtrees of its children, v calculates the weight of its subtree (calculating its weight as 0, in case it is a hole) and passes the information in a slow message to its parent. The correctness of the computed weight is obvious due to the simple structure of a tree. It remains to prove that the weight of the whole intermediate tree is calculated in logarithmic time. It is proved in the following lemma.

Lemma 8.10. *All slow messages are sent within* $24 \cdot \log n$ *time steps from the failure.*

Proof. From Lemma 8.8 we know that each item receives a fast message within $3 \cdot \log n$ time steps and that it waits for a fast message from its second child for at most $6 \cdot h$ time steps, where h is the height of the leaf it knows about. The height of the whole tree is bounded by $3 \cdot \log n$, so all items finish waiting for fast messages within $(3 + 18) \cdot \log n$ time steps.

Let t_0 be the time step in which all items have received the fast messages or made sure that fast messages will never come. Let t_i, for $i > 0$, be consecutive time steps. We prove by induction that the heights of all items which have not sent their slow message by time step t_i are bounded by $(3 \cdot \log n) - i$, that is, that in time step $t_{3 \cdot \log n}$ the procedure finishes. In time step t_0 all items at level $3 \cdot \log n$ can send their slow messages, as they do not have children to wait for. This proves the induction base. For the inductive step assume that all alive items from levels higher than $(3 \cdot \log n) - (i - 1)$ have sent their messages at latest in time step t_{i-1}. By time step t_i items at level $(3 \cdot \log n) - i$ receive messages from their alive descendants and know that they will never receive messages from dead ones, so they can send their slow messages in this time step.

Since the height of the intermediate tree is at most $3 \cdot \log n$, in time step $3 \cdot \log n$, all items will have sent their slow messages. We know that $t_0 \leq 21 \cdot \log n$, so the total time needed to send all slow messages is bounded by $24 \cdot \log n$. ■

In the second phase of the recovery process, requests are generated to refill the failed and newly built items, that is holes. Holes generate requests to items which can refill them in a top-down fashion. The procedure starts from the root and an item can start it only when its parent allows it to. If an item is allowed to start the procedure and it is not a hole, it simply forwards the permission to its children but only to which sent a fast message in the first phase. If it is a hole, before it allows its children to act, it performs the following actions. If both of its subtrees are empty (the information comes from the weight stored in the root of each subtree) it finishes the second phase, as it does not need to be filled because it does not belong to the intermediate tree. In other case, it chooses a non-empty subtree from which it wants to take an item. If it chose the right subtree, it wants the leftmost item in it and if it is the left subtree, it wants the rightmost item there. It sends a request downwards and the message carrying the request has the following properties. One is that it will not be overtaken by a permission to start the second phase. Another is that it can already update the weight information in all items on its way. And the last one is that a requesting message from a parent is never overtaken by any message from a child, so when an item gets a permission to start the second phase, its weight has already been updated by all requesting messages sent by all its ancestors.

Since each item needs at most two time steps, one to possibly send a requesting message to one of its subtrees and another to send or forward a permission message to both of its children, the whole second phase is finished in at most $6 \cdot \log n$ time steps.

In the third and last phase, the items which received request messages are moved upwards to fill the holes which sent the requesting messages. The time of the phase is bounded by the height of the original tree, that is $3 \cdot \log n$. After the phase has finished, a balancing operation can be started in the leaves, which then follows to the root.

In the first and second phase there are only messages on paths from broken items to the root of the tree, so each message can be attributed to a broken item below. Moreover, each item sends only a constant number of messages, so the total communication cost is bounded by $O(m \log n)$, where m is the number of failures. Similarly in sending requests in the second phase and in the third phase broken items send messages which can travel along paths in the tree, so the total cost is bounded by $O(m \log n)$, too. Thus, we come to the following theorem about the recovery algorithm.

Theorem 8.11. *The recovery algorithm allows the search tree of size n to recover from any set of m failures in $O(\log n)$ time using at most $O(m \log n)$ work.*

8.3. Local balancing procedure

Below we present a procedure used to take local decisions if imbalance appears in the tree. We assume that the procedure is run in an item r and that the descendants of r up to three levels fulfill the balance condition. After performing the balancing in r, we will guarantee that on each path from r to a leaf at most one item loses balance in its subtree. Intuitionally, it can be thought of as waves of balancing going from the root to the leaves. The new wave that we introduce, while balancing r, will never catch or overtake the waves that existed and moved in the direction of leaves before.

Proof of Lemma 8.6. Let n be the number of items in the considered tree. For simplicity of notation for any (sub)tree α its name denotes also its *weight*, i.e. the number of items it contains. The simplest starting situation is depicted in Figure 8.1. We will give more detailed views as we explore more details about the tree. Imbalance in the network means simply that $0 \leq \alpha \leq \frac{n}{4}$. In order to cope with it, we consider the following cases.

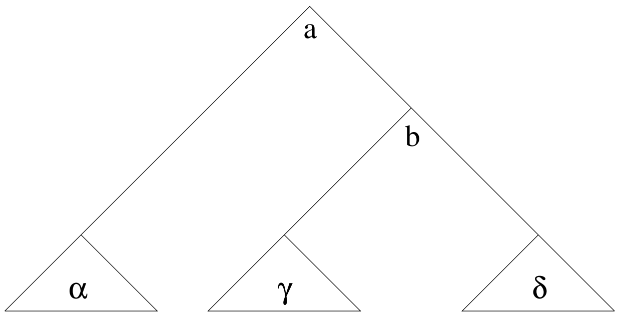

Figure 8.1.: Initial state of the tree, the simplest case

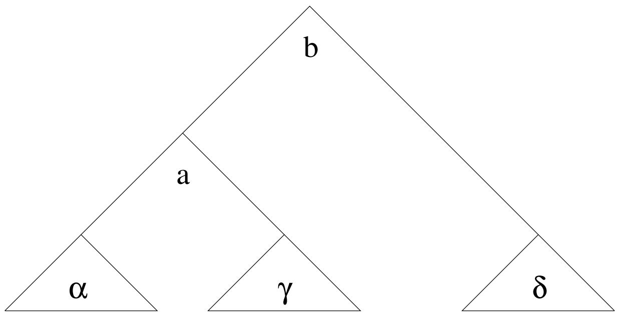

Figure 8.2.: The tree after rotation, the simplest case

1. If $\delta \geq \frac{5n}{16}$ then we perform a single rotation so that the tree looks as in Figure 8.2. After the rotation the balance in the root is ensured by $\frac{n}{4} < \frac{5n}{16} \leq \delta \leq \frac{3}{4} \cdot (\delta + \gamma) \leq \frac{3n}{4}$. Before the rotation $\phi(a) = \delta + \gamma - \alpha$ and $\phi(b) = |\delta - \gamma|$. After the rotation the potentials should be $\phi'(a) = |\alpha - \gamma|$ and $\phi'(b) = |\alpha + \gamma - \delta|$. Below we show that $\Delta_\phi := \phi(a) + \phi(b) - \phi'(a) - \phi'(b) = \Omega(n)$, i.e. the initial potential is sufficient to store the resulting potential and pay for the rotation. We consider three cases:

   - $\delta \leq \gamma$: $\Delta_\phi = (\gamma + \delta - \alpha) + (\gamma - \delta) - (\gamma - \alpha) - (\gamma + \alpha - \delta) = \delta - \alpha \geq \frac{n}{16}$
   - $\gamma \leq \delta \leq \alpha + \gamma$: $\Delta_\phi = (\gamma + \delta - \alpha) + (\delta - \gamma) - (\alpha + \gamma - \delta) - |\alpha - \gamma| = 3\delta - 2\alpha - \gamma - |\alpha - \gamma| = \delta - \alpha + 2\delta - 2\max\{\alpha, \gamma\} \geq \delta - \alpha \geq \frac{n}{16}$
   - $\alpha + \gamma \leq \delta$: $\Delta_\phi = (\delta + \gamma - \alpha) + (\delta - \gamma) - (\delta - \alpha - \gamma) - |\alpha - \gamma| = \delta + \gamma - |\alpha - \gamma| \geq \delta - \alpha \geq \frac{n}{16}$

   Obviously, any imbalance can only be in a.

2. If $\frac{3n}{16} \leq \delta \leq \frac{5n}{16}$ and $\frac{3n}{16} \leq \alpha \leq \frac{4n}{16}$ we assume that the tree looks as depicted in Figure 8.3 and we perform a rotation so that it looks as in Figure 8.4. From the assumptions we have $\frac{6n}{16} \leq \alpha + \delta \leq \frac{9n}{16}$, so $\frac{7n}{16} \leq \epsilon + \lambda \leq \frac{10n}{16}$ and $\frac{7n}{64} \leq \epsilon, \lambda \leq \frac{30n}{64}$. The balance in the root is assured by $\frac{n}{4} < \frac{19n}{64} \leq \alpha + \lambda \leq \frac{46n}{64} < \frac{3n}{4}$. We need to show that $\Delta_\phi := \phi(a) + \phi(b) + \phi(c) - \phi'(a) - \phi'(b) - \phi'(c) = (\delta + \lambda + \epsilon - \alpha) + |\lambda - \epsilon| + (\lambda + \epsilon - \delta) - |\lambda - \alpha| - |\alpha + \lambda - \delta - \epsilon| - |\epsilon - \delta| = \Omega(n)$ We consider two cases.

   - If $\alpha + \lambda \geq \delta + \epsilon$ then we have $\Delta_\phi = \lambda + \epsilon - \alpha + 2 \cdot \max\{\lambda, \epsilon\} - 2 \cdot \max\{\lambda, \alpha\} + 2 \cdot \min\{\epsilon, \delta\}$. Now if $\lambda \geq \alpha$ then $\Delta_\phi \geq \lambda + \epsilon - \alpha \geq \frac{3n}{16}$. Otherwise $\Delta_\phi \geq 2\lambda + 2\epsilon - 3\alpha \geq \frac{2n}{16}$.
   - If $\alpha + \lambda \leq \delta + \epsilon$ then we have $\Delta_\phi = \lambda + \epsilon - \alpha + 2 \cdot \max\{\lambda, \epsilon\} - 2 \cdot \max\{\epsilon, \delta\} + 2 \cdot \min\{\alpha, \lambda\}$. Now if $\epsilon \geq \delta$ then $\Delta_\phi \geq \lambda + \epsilon - \alpha \geq \frac{3n}{16}$. Otherwise $\Delta_\phi \geq 2\lambda + 2\epsilon - \alpha - 2\delta + 2 \cdot \min\{\alpha, \lambda\} \geq \frac{14n}{16}$.

   Obviously imbalance can only be in a or c, that is at only one level in each branch.

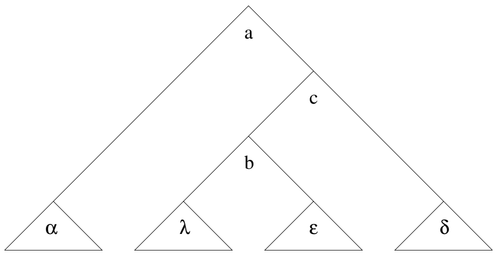

Figure 8.3.: Initial state of the tree, intermediate case

3. If $\frac{3n}{16} \leq \delta \leq \frac{5n}{16}$ and $\frac{2n}{16} \leq \alpha \leq \frac{3n}{16}$ we assume that the tree looks as depicted in Figure 8.3 and we perform a rotation so that it looks as in Figure 8.4. From the assumptions we have $\frac{5n}{16} \leq \alpha + \delta \leq \frac{8n}{16}$, so $\frac{8n}{16} \leq \epsilon + \lambda \leq \frac{11n}{16}$ and $\frac{8n}{64} \leq \epsilon, \lambda \leq \frac{33n}{64}$. The balance in the root is assured by $\frac{n}{4} \leq \frac{16n}{64} \leq \alpha + \lambda \leq \frac{45n}{64} < \frac{3n}{4}$. We need to show that $\Delta_\phi := \phi(a) + \phi(b) + \phi(c) - \phi'(a) - \phi'(b) - \phi'(c) = (\delta + \lambda + \epsilon - \alpha) + |\lambda - \epsilon| + (\lambda + \epsilon - \delta) - |\lambda - \alpha| - |\alpha + \lambda - \delta - \epsilon| - |\epsilon - \delta| = \Omega(n)$ We consider two cases.

   - If $\alpha+\lambda \geq \delta+\epsilon$ then we have $\Delta_\phi = \lambda+\epsilon-\alpha+2\cdot\max\{\lambda,\epsilon\}-2\cdot\max\{\lambda,\alpha\}+2\cdot\min\{\epsilon,\delta\}$. Now if $\lambda \geq \alpha$ then $\Delta_\phi \geq \lambda + \epsilon - \alpha \geq \frac{5n}{16}$. Otherwise $\Delta_\phi \geq 2\lambda + 2\epsilon - 3\alpha \geq \frac{7n}{16}$.
   - If $\alpha+\lambda \leq \delta+\epsilon$ then we have $\Delta_\phi = \lambda+\epsilon-\alpha+2\cdot\max\{\lambda,\epsilon\}-2\cdot\max\{\epsilon,\delta\}+2\cdot\min\{\lambda,\alpha\}$. Now if $\epsilon \geq \delta$ then $\Delta_\phi \geq \lambda + \epsilon - \alpha \geq \frac{5n}{16}$. Otherwise $\Delta_\phi \geq 2\lambda + 2\epsilon - \alpha - 2\delta + 2 \cdot \min\{\alpha, \lambda\} \geq \frac{7n}{16}$.

   The same as in the previous case, imbalance can only be in *a* or *c*.

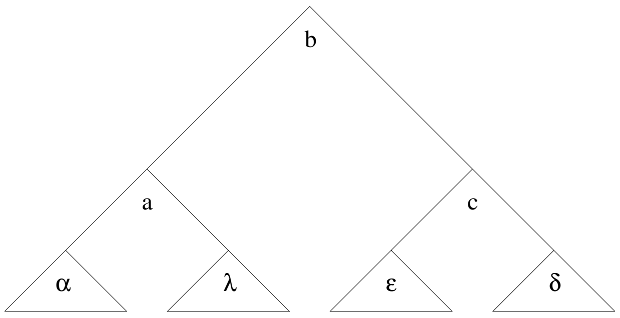

Figure 8.4.: The tree after rotation, intermediate case

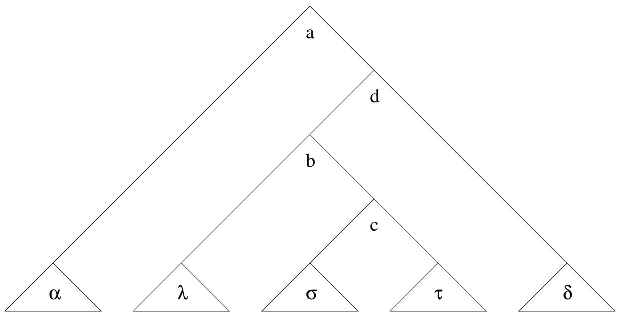

Figure 8.5.: Initial state of the tree, the most advanced case

4. If $\frac{3n}{16} \leq \delta \leq \frac{5n}{16}$ and $\frac{n}{16} \leq \alpha \leq \frac{2n}{16}$ and $\frac{12n}{64} \leq \lambda \leq \frac{36n}{64}$ we assume that the tree looks as depicted in Figure 8.3 and we perform a rotation so that it looks as in Figure 8.4. From the first two assumptions we have $\frac{4n}{16} \leq \alpha + \delta \leq \frac{7n}{16}$, so $\frac{9n}{16} \leq \epsilon + \lambda \leq \frac{12n}{16}$ and $\frac{9n}{64} \leq \epsilon, \lambda \leq \frac{36n}{64}$ but from the last assumption we have also $\frac{12n}{64} \leq \lambda$. The balance

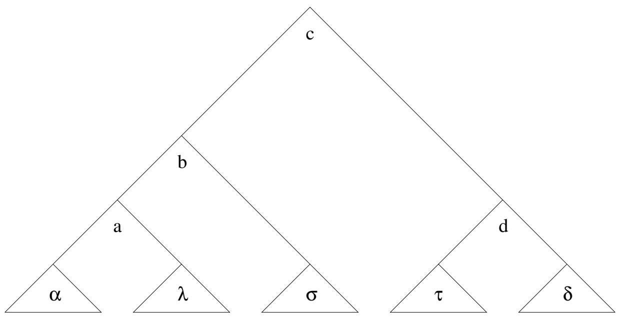

Figure 8.6.: The tree after rotation, the most advanced case

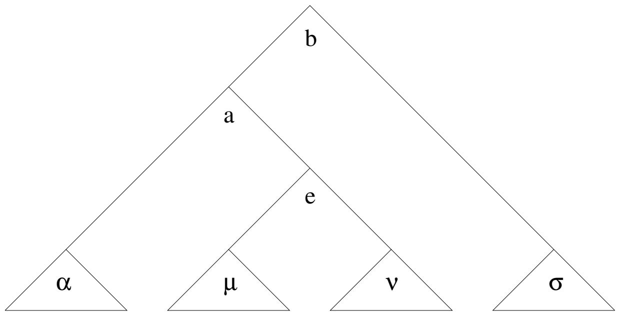

Figure 8.7.: Zoom of the most advanced case

in the root is assured by $\frac{n}{4} \le \frac{16n}{64} \le \alpha + \lambda \le \frac{44n}{64} < \frac{3n}{4}$. We need to show that $\Delta_\phi := \phi(a) + \phi(b) + \phi(c) - \phi'(a) - \phi'(b) - \phi'(c) = (\delta + \lambda + \epsilon - \alpha) + |\lambda - \epsilon| + (\lambda + \epsilon - \delta) - |\lambda - \alpha| - |\alpha + \lambda - \delta - \epsilon| - |\epsilon - \delta| = \Omega(n)$ We consider two cases.

- If $\alpha+\lambda \ge \delta+\epsilon$ then we have $\Delta_\phi = \lambda+\epsilon-\alpha+2\cdot\max\{\lambda,\epsilon\}-2\cdot\max\{\lambda,\alpha\}+2\cdot\min\{\epsilon,\delta\}$. Now $\lambda \ge \alpha$ so $\Delta_\phi \ge \lambda + \epsilon - \alpha \ge \frac{7n}{16}$.
- If $\alpha+\lambda \le \delta+\epsilon$ then we have $\Delta_\phi = \lambda+\epsilon-\alpha+2\cdot\max\{\lambda,\epsilon\}-2\cdot\max\{\epsilon,\delta\}+2\cdot\min\{\lambda,\alpha\}$. Now if $\epsilon \ge \delta$ then $\Delta_\phi \ge \lambda+\epsilon-\alpha \ge \frac{7n}{16}$. Otherwise $\Delta_\phi \ge 2\lambda+2\epsilon-\alpha-2\delta+2\alpha \ge \frac{9n}{16}$.

The same as in the previous two cases, imbalance can only be in a or c.

5. If $\frac{3n}{16} \le \delta \le \frac{5n}{16}$ and $\frac{n}{16} \le \alpha \le \frac{2n}{16}$ and $\frac{9n}{64} \le \lambda \le \frac{12n}{64}$ we assume that the tree looks as depicted in Figure 8.5 and we perform a rotation so that it looks as in Figure 8.6. From the first two assumptions we have $\frac{4n}{16} \le \alpha + \delta \le \frac{7n}{16}$, so $\frac{9n}{16} \le \sigma + \tau + \lambda \le \frac{12n}{16}$ and $\frac{9n}{64} \le \sigma + \tau, \lambda \le \frac{36n}{64}$ but from the last assumption we have also $\lambda \le \frac{12n}{64}$, so $\frac{24n}{64} \le \sigma + \tau \le \frac{36n}{64}$ and $\frac{6n}{64} \le \sigma, \tau \le \frac{27n}{64}$. The balance in the root is assured by $\frac{n}{4} < \frac{19n}{64} \le \alpha + \lambda + \sigma \le \frac{47n}{64} < \frac{3n}{4}$. We need to show that $\Delta_\phi := \phi(a)+\phi(b)+\phi(c)+\phi(d)-\phi'(a)-\phi'(b)-\phi'(c)-\phi'(d) = (\delta+\lambda+\sigma+\tau-\alpha)+(\sigma+\tau-\lambda)+|\sigma-\tau|+(\lambda+\sigma+\tau-\delta)-(\lambda-\alpha)-|\alpha+\lambda-\sigma|-|\alpha+\lambda+\sigma-\delta-\tau|-|\delta-\tau| = \Omega(n)$. We notice that $\Delta_\phi = 3\sigma+3\tau+|\sigma-\tau|-|\alpha+\lambda-\sigma|-|\alpha+\lambda+\sigma-\delta-\tau|-|\delta-\tau| \ge \frac{72n}{64}+0-\frac{14n}{64}-\frac{29n}{64}-\frac{15n}{64} = \frac{14n}{64}$.

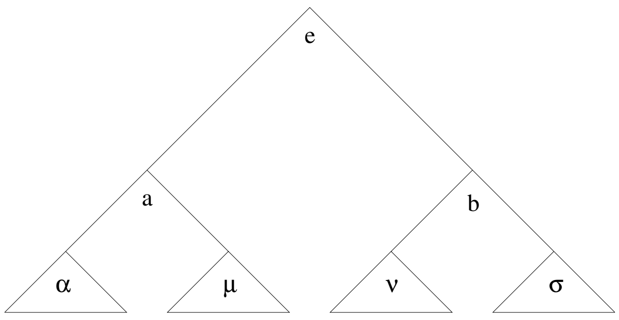

Figure 8.8.: Zoom of the most advanced case after rotation

Possible imbalance in d is not a problem, since it is the only imbalance in the right branch. There can be imbalance in c but a is balanced, since $\alpha \geq \frac{n}{16} \geq \frac{1}{3} \cdot \frac{3n}{16} \geq \lambda$ and $\alpha \leq \frac{2n}{16} < \frac{9n}{64} \leq \lambda$.

6. If $\frac{3n}{16} \leq \delta \leq \frac{5n}{16}$ and $0 \leq \alpha \leq \frac{n}{16}$ and $\frac{16n}{64} \leq \lambda \leq \frac{39n}{64}$ we assume that the tree looks as depicted in Figure 8.3 and we perform a rotation so that it looks as in Figure 8.4. From the first two assumptions we have $\frac{3n}{16} \leq \alpha + \delta \leq \frac{6n}{16}$, so $\frac{10n}{16} \leq \epsilon + \lambda \leq \frac{13n}{16}$ and $\frac{10n}{64} \leq \epsilon, \lambda \leq \frac{39n}{64}$ but from the last assumption we have also $\frac{16n}{64} \leq \lambda$. The balance in the root is assured by $\frac{n}{4} \leq \frac{16n}{64} \leq \alpha + \lambda \leq \frac{43n}{64} < \frac{3n}{4}$. We need to show that $\Delta_\phi := \phi(a) + \phi(b) + \phi(c) - \phi'(a) - \phi'(b) - \phi'(c) = (\delta + \lambda + \epsilon - \alpha) + |\lambda - \epsilon| + (\lambda + \epsilon - \delta) - (\lambda - \alpha) - |\alpha + \lambda - \delta - \epsilon| - |\epsilon - \delta| = \Omega(n)$ We consider two cases.

   - If $\alpha+\lambda \geq \delta+\epsilon$ then we have $\Delta_\phi = \lambda+\epsilon-\alpha+2\cdot\max\{\lambda,\epsilon\}-2\cdot\max\{\lambda,\alpha\}+2\cdot\min\{\epsilon,\delta\}$. Now $\lambda \geq \alpha$ so $\Delta_\phi \geq \lambda + \epsilon - \alpha \geq \frac{9n}{16}$.
   - If $\alpha+\lambda \leq \delta+\epsilon$ then we have $\Delta_\phi = \lambda+\epsilon-\alpha+2\cdot\max\{\lambda,\epsilon\}-2\cdot\max\{\epsilon,\delta\}+2\cdot\min\{\lambda,\alpha\}$. Now if $\epsilon \geq \delta$ then $\Delta_\phi \geq \lambda+\epsilon-\alpha \geq \frac{9n}{16}$. Otherwise $\Delta_\phi \geq 2\lambda+2\epsilon-\alpha-2\delta+2\alpha \geq \frac{10n}{16}$.

   The same as in cases 2,3 and 4, imbalance can only be in a or c.

7. If $\frac{3n}{16} \leq \delta \leq \frac{5n}{16}$ and $0 \leq \alpha \leq \frac{n}{16}$ and $\frac{10n}{64} \leq \lambda \leq \frac{16n}{64}$ we assume that the tree looks as depicted in Figure 8.5 and we perform a rotation so that it looks as in Figure 8.6. From the first two assumptions we have $\frac{3n}{16} \leq \alpha + \delta \leq \frac{6n}{16}$, so $\frac{10n}{16} \leq \sigma + \tau + \lambda \leq \frac{13n}{16}$ and $\frac{10n}{64} \leq \sigma + \tau, \lambda \leq \frac{39n}{64}$ but from the last assumption we have also $\lambda \leq \frac{16n}{64}$, so $\frac{24n}{64} \leq \sigma + \tau \leq \frac{39n}{64}$ and $\frac{6n}{64} \leq \sigma, \tau < \frac{117n}{256}$. The balance in the root is assured by $\frac{n}{4} = \frac{16n}{64} \leq \alpha + \lambda + \sigma$ and $\frac{1}{4} < \frac{18n}{64} \leq \tau + \delta$. We need to show that $\Delta_\phi := \phi(a)+\phi(b)+\phi(c)+\phi(d)-\phi'(a)-\phi'(b)-\phi'(c)-\phi'(d) = (\delta+\lambda+\sigma+\tau-\alpha)+(\sigma+\tau-\lambda)+|\sigma-\tau|+(\lambda+\sigma+\tau-\delta)-(\lambda-\alpha)-|\alpha+\lambda-\sigma|-|\alpha+\lambda+\sigma-\delta-\tau|-|\delta-\tau| = \Omega(n)$. We notice that $\Delta_\phi = 3\sigma+3\tau+|\sigma-\tau|-|\alpha+\lambda-\sigma|-|\alpha+\lambda+\sigma-\delta-\tau|-|\delta-\tau| \geq \frac{288n}{256}+0-\frac{77n}{256}-\frac{133n}{256}-\frac{69n}{256} = \frac{9n}{256}$

In this case a problem can appear if b is not balanced. Since $\sigma \le \frac{117n}{256} < \frac{120n}{256} = 3 \cdot \frac{10n}{64} \le 3 \cdot \lambda \le 3 \cdot (\alpha + \lambda)$, this means that $\sigma \le \frac{1}{3} \cdot (\alpha + \lambda)$ but $\frac{6n}{64} \le \sigma \le \frac{1}{3} \cdot (\alpha + \lambda) \le \frac{1}{3} \cdot \frac{18n}{64} = \frac{6n}{64}$, which means that $\alpha = \frac{4n}{64}$, $\lambda = \frac{16n}{64}$ $\sigma = \frac{6n}{64}$. In such case we take a closer look at λ. The interesting part of the tree is depicted in Figure 8.7. We perform a rotation so that the tree looks as in Figure 8.8. The tree rooted in e is balanced, since $\alpha + \mu, \nu + \sigma \le \frac{18n}{64} \le \frac{3}{4} \cdot \frac{24}{64} < \frac{3}{4} \cdot (\alpha + \mu + \nu + \sigma)$. Thus only a, b or d can be in imbalance but they are all in separate branches. The decrease in potential can cover the cost, since $\Delta_\phi = (\mu + \nu - \alpha) + (\alpha + \mu + \nu - \sigma) - (\mu - \alpha) - |\nu - \sigma| - |\alpha + \mu - \nu - \sigma| \ge \frac{12n}{64} + \frac{14n}{64} - \frac{8n}{64} - \frac{6n}{64} - \frac{10n}{64} = \frac{2n}{64}$.

The minimum decrease of potential in a rotation is $\frac{9n}{256}$. In each case a constant number of primitive rotations is performed and each of the primitive operations moves at most n nodes to constant distance. Thus, the total movement cost is at most linear. ■

CHAPTER 9

Summary and Outlook

In this thesis we concentrated on algorithmic aspects of Peer-to-Peer networks. This is the area in which hundreds if not thousands of scientists all over the world are interested and we were hoping to contribute something too.

9.1. Load balancing

Load balancing has been a problem addressed already in the first algorithmic designs of Peer-to-Peer networks and much work has been done concerning it. We considered only the area of smoothing the intervals for which the nodes are responsible and still one can see that there is no one and only solution of this problem, even though more than a dozen publications on this topic exist.

The algorithm ALGSTAT , we presented, is meant to perform the load balancing procedures as quickly as possible and it achieves this but pays with significant communication cost. The experiments show that the efficiency of the algorithm can be proved not only on paper but also in applications. Even though, the constants emerging from the analysis are in the range of thousands, the experiments show that the smoothness can be reduced from more than 2000 if the nodes are distributed uniformly to under 20, if we let our algorithm take care of the network.

The algorithm ALGDYN is meant to repair the main disadvantage of the first one. Its low communication cost can be provably bounded but the balance of the system is always a few steps behind its dynamics.

We do not provide experimental analysis for the second algorithm, as deeper considerations lead to a conclusion that it would be a project in itself. A model of the dynamics of the system is needed which captures the changing rates at which nodes join and leave. Intuitionally, one can think that for example in the morning there are many joins and in the evening many leaves. Such view is complicated by timezones, density of computer equipped population on the whole planet and different activities of people on weekends and working days. We do not have the knowledge or tools to devise and check the correctness of a model which would grasp all these intuitions. Except this, simulations should be executed at least on a cluster of computers, since a single computer was hardly capable of performing simulations for the simple model.

A significant question that remains open is how to balance the load in a heterogeneous system, where heterogeneity has at least a few independent aspects like storage capacity, communication bandwidth and computation power. To the best of our knowledge, the only means of treating this problem that has been suggested is introduction of virtual processors. Other approaches should be investigated in the nearest future.

An aspect that we did not investigate, is the balancing of the items stored in the system when the scarce resource is storage capacity and dealing with the popularity of items, when communication bandwith should be spared.

9.2. Data structures

The second part of this thesis can also be thought of as load balancing, thanks to its concentrating on balanced trees. We presented a scheme that lets implement a balanced binary search trees on top of a Peer-to-Peer network in a robust way. Hopefully, this is only the first step in the direction of implementing other useful data structures in such an environment. It is probably possible to use the Hypertree we have designed as a foundation of a simple heap structure. Other data structures will need more sophisticated designs.

We hope that our work on transparent data structures will advance the Peer-to-Peer networks as a tool for large scale distributed computation. This is also the direction of research we would like to persue in the nearest future.

We would like to concentrate on creating a model of computation based on virtual processes which can be migrated in a Peer-to-Peer network and thus achieve a huge distributed computation platform. In such a system each process works for itself communicating to other processes through messages or common data structures. For the second possibility, transparent data structures can be adapted.

Such a system could exploit the advantages of Peer-to-Peer networks and at the same time can be designed to counteract their deficiencies. For example the heterogeneity in different aspects like computing power, storage space and network bandwidth can be used to assign more processes, more backup burden or more responsibilities for network organization to peers strong in respective areas.

On the other hand robustness should be taken into consideration, as in a Peer-to-Peer network we have to assume that some nodes will not only leave but also fail without notice. Even using counteracting techniques like replication, we have to take into account that some data will be lost from time to time. Procedures which cope with such inconveniences should be very carefully designed and transparent data structures are the first step and a good example how it can be done.

Bibliography

[AB96] Yonatan Aumann and Michael A. Bender. Fault Tolerant Data Structures. In *Proc. of the 37th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 580–589, 1996.

[AHKV03] Micah Adler, Eran Halperin, Richard M. Karp, and Vijay Vazirani. A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks. In *Proc. of the 35th ACM Symp. on Theory of Computing (STOC)*, pages 575–584, 2003.

[AL90] Arne Andersson and Tony W. Lai. Fast Updating of Well-Balanced Trees. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 111–121, 1990.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *Proc. of the 14th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 384–393, 2003.

[AS04a] Baruch Awerbuch and Christian Scheideler. Group Spreading: A Protocol for Provably Secure Distributed Name Service. In *Proc. of the 31st Int. Colloquium on Automata, Languages, and Programming (ICALP)*, pages 183–195, 2004.

[AS04b] Baruch Awerbuch and Christian Scheideler. The Hyperring: A Low-Congestion Deterministic Data Structure for Distributed Environments. In *Proc. of the 15th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 318–327, 2004.

[AW05] James Aspnes and Udi Wieder. The Expansion and Mixing Time of Skip Graphs with Applications. In *Proc. of the 17th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 126–134, 2005.

[BAG92] Amir M. Ben-Amram and Zvi Galil. On Pointers Versus Addresses. *Journal of the ACM*, 39(3):617–648, 1992.

[BCM03] John W. Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems*, pages 80–87, 2003.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees. In *Proc. of the 41st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[BFJ02] Gerth Stlting Brodal, Rolf Fagerberg, and Riko Jacob. Cache-Oblivious Search Trees via Trees of Small Height. In *Proc. of the 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[BK05] Marcin Bienkowski and Miroslaw Korzeniowski. Bounding the Communication Cost in Dynamic Load Balancing of Distributed Hash Tables. In *Proc. of the 9th International Conference on Principles of Distributed Systems*, 2005.

[BKM05] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic Load Balancing in Distributed Hash Tables. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems*, 2005.

[CD93] Stephen A. Cook and Patrick W. Dymond. Parallel Pointer Machines. *Computational Complexity*, 3:19–30, 1993.

[CKP$^+$93] Dadiv E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahaj, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten van Eicken. LogP: Towards a Realistic Model for Parallel Computation. In *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1989.

[DR98] Devdatt P. Dubhashi and Desh Ranjan. Balls and Bins: A study in Negative Dependence. *Random Structures and Algorithms*, 13(2):99–124, 1998.

[DSZ94] Paul F. Dietz, Joel I. Seiferas, and Ju Zhang. A Tight Lower Bound for On-Line Monotonic List Labeling. In *Proc. of the 6th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 131–142, 1994.

[DZ90] Paul F. Dietz and Ju Zhang. Lower Bounds for Monotonic List Labeling. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–180, 1990.

[GK89] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a Parallel Pointer Machine with Applications to Set Expression Evaluation. In *Proc. of the 30th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 190–195, 1989.

[GLS$^+$04] B. Godfrey, K. Lakshminarayanan, S. Surana, Richard M. Karp, and Ion Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proc. of the 23rd Conference of the IEEE Communications Society (INFOCOM)*, 2004.

[GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[HJS$^+$03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[HKRZ02] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 41–52, 2002.

[HM04] Nicholas J. A. Harvey and J. Ian Munro. Deterministic SkipNet. *Information Processing Letters*, 90(4):205–208, 2004.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A Sparse Table Implementation of Sorted Sets. Technical Report RC 9146, IBM T.J. Watson Research Center, Yorktown Heights, New York, 1981.

[KLL$^+$97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Matthew S. Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 654–663, 1997.

[KM05] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized Algorithms Using Both Local and Random Probes for P2P Load Balancing. In *Proc. of the 17th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 135–144, 2005.

[KR04a] David R. Karger and Matthias Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems*, pages 131–140, 2004.

[KR04b] David R. Karger and Matthias Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proc. of the 16th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 36–43, 2004.

[KS05] Miroslaw Korzeniowski and Christian Scheideler. Transparent Data Structures, Or How to Make Search Trees Robust in a Distributed Environment. In *Proc. of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, 2005.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[LNBK02] David Liben-Nowell, Hari Balakrishnan, and David R. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proc. of the 21st annual ACM symposium on Principles of Distributed Computing(PODC)*, pages 233–242, 2002.

[Man04] Gurmeet Singh Manku. Balanced Binary Trees for ID Management And Load Balance in Distributed Hash Tables. In *Proc. of the 23rd annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 197–205, 2004.

[MRS00] Michael Mitzenmacher, Andra W. Richa, and Ramesh K. Sitaraman. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing. P. Pardalos, S.Rajasekaran, J.Rolim, and Eds. Kluwer, editors*, 2000.

[MS90] Burkhard Monien and Ivan Hal Sudborough. Embedding One Interconnection Network in Another. *Computing Supplementum*, 7:257–282, 1990.

[MV99] Friedhelm Meyer auf der Heide and Berthold Vöcking. Shortest Path Routing in Arbitrary Networks. *Journal of Algorithms*, 31(1):105–131, 1999.

[NW03a] Moni Naor and Udi Wieder. A Simple Fault Tolerant Distributed Hash Table. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems*, pages 88–97, 2003.

[NW03b] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *Proc. of the 15th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 50–59, 2003.

[PRU01] Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building Low-Diameter Peer-to-Peer Networks. In *Proc. of the 42nd IEEE symposium on Foundations of Computer Science (FOCS)*, pages 492–499, 2001.

[Ram97] Vijaya Ramachandran. QSM: A General Purpose Shared-Memory Model for Parallel Computation. In *Proc. of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 1–5, 1997.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proc. of the ACM SIGCOMM*, pages 161–172, 2001.

[RLS$^+$03] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems*, pages 68–79, 2003.

[Sav98] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison Wesley, 1998.

[Sch00] Christian Scheideler. Probabilistic Methods for Coordination Problems. Habilitation thesis, University of Paderborn, 2000.

[SMK$^+$01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of the ACM SIGCOMM*, pages 149–160, 2001.

[vEB90] Peter van Emde Boas. *Handbook of Theoretical Computer Science, Vol. A*, chapter Machine Models and Simulations, pages 1–66. Elsevier, 1990.

[Wie05] Udi Wieder. *The Continuous-Discrete Approach for Designing P2P Networks and Algorithms*. PhD thesis, 2005.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A Note on Distributed Computing. Technical report SMLI TR-94-29, Sun Microsystems Laboratories, 1994.