Dynamic & fault-tolerant P2P-topologies

Master Thesis

Author(s):
Schmid, Stefan

Publication date:
2004

Permanent link:
https://doi.org/10.3929/ethz-a-004803263

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DIPLOMA THESIS

Dynamic & Fault-Tolerant P2P-Topologies

Stefan Schmid

Prof. Dr. Roger Wattenhofer
Fabian Kuhn

Distributed Computing Group
Dept. of Computer Science
ETH Zurich, Switzerland

11^{th} May - 10^{th} September 2004

Abstract

Until now, the analysis of fault tolerance of peer-to-peer (p2p) systems usually only covers random faults of some kind. Contrary to traditional algorithmic research, faults as well as joins and leaves occurring in a worst-case manner in p2p systems are hardly considered. Most fault tolerance analyzes are static in the sense that it is shown that a system tolerates a certain number of simultaneous faults. The much more realistic dynamic case where faults steadily occur has not found much attention. The goal of this thesis is to work towards a general understanding of fault tolerance especially for the case of dynamic and worst-case failures.

The thesis introduces a simple dynamic model where a malicious adversary — controlling the arrivals and departures of the peers — and a repairing algorithm take turns. The insights gained from this model are then used to study the more realistic scenario where a repairing algorithm maintains certain properties of the topology against a *concurrent* adversary.

Besides the comparison of different fault-tolerance models, the thesis presents a distributed hash table which maintains an efficient search structure as well as a low peer degree in spite of the worst-case failures. By a dynamic analysis we prove that no data item is lost by the system.

Contents

Chapter 1

Introduction

Peer-to-peer (p2p) systems and applications are distributed systems without any centralized control — both a bottleneck for scalability and a single point of failure — where the software running at each node is equivalent in functionality. These systems emerged as a new distributed computing paradigm because of their potential to harness the computing power of the hosts composing the network.

p2p systems may consist of thousands of peers and are characterized by a high dynamics in the sense that peers may join the system, leave the system voluntarily or simply crash at any moment of time. For example, the median session duration in the multimedia file sharing system Napster [1] was approximately one hour [13]. Besides the composition of the system, also the *total number* of participating peers can vary significantly over time, and it is therefore crucial to control the evolution of the system in order to guarantee desirable properties such as a low peer degree or a low network diameter.

Following the seminal work of Plaxton et al. [10], an assortment of variants of p2p systems have been proposed in literature, such as CAN [12], Chord [15], and Tapestry [18]. However, most fault-tolerance analyzes of these systems are either static or cover only random faults. For example, experimental evidence is supplied that Tapestry is robust against random faults, while Chord is *provably* resilient to a constant fraction of random node failures.

This document focuses on *dynamic worst-case failures*. We introduce a simple dynamic model where a malicious *adversary*, which — having complete visibility of the entire state of the system — controls the joins and leaves in the system, alternates with a *repairing algorithm*. The goal of the repairing algorithm is to re-establish certain properties of the system, for example a high connectivity such that the topology remains one connected component in the next adversarial round. Armed with the insights we get from this simple model, the more realistic scenario where the adversary acts *concurrently* to the repairing algorithm is considered. We will see that it is sometimes possible to transform a repairing algorithm running in time $O(1)$ to work also in the concurrent model.

The thesis is organized as follows: Chapter 2 introduces the models that will be used throughout this document. In particular, it describes the adversarial operations and formally defines the objective of the repairing algorithm. In Chapter 3 we start our analysis with a very simple topology based on the ring. After the comparison of different models for the case where a repairing algorithm runs in a phase of quiescence, a concurrent model is studied. Chapter 4 presents an algorithm which allows peers to

aggregate information in a p2p system in a distributed fashion. The properties of this algorithm are useful for the maintenance of the topologies introduced in later chapters. A main emphasis in laid on Chapter 5, where a simulated d-dimensional hypercube topology is introduced. First, we will present a repairing algorithm which maintains a low peer degree and a low network diameter against a concurrent adversary which inserts and removes $\Theta(d)$ peers per time interval of constant length. We will then show that this system can also be deployed as an efficient distributed hash table which never loses data. Chapter 6 extends these results to the skip graph topology. We conclude our work in Chapter 7, where we also give some directions for future research projects.

Chapter 2

Model

We consider a graph $G = (V, E)$, where V represents the set of peers and E describes the adjacency relations of these peers. If not stated otherwise, the classic synchronous *message passing* model is studied, where in every round, a node can send a message to each of its adjacent nodes; local computations are assumed to take no time.

The dynamics of the system is given by a non-oblivious adversary which may insert at most J and remove at most L nodes. We assume that new nodes always arrive at nodes which already belong to the system. Algorithm 1 gives a formal description of the adversarial operations; an example is depicted in Figure 2.1.

Algorithm 1 Adversary

1: INPUT: Graph $G = (V, E)$, J, L
2: choose $\mathcal{L} \subseteq V$ where $|\mathcal{L}| \leq L$;
3: choose $\mathcal{J} := \{j_0, ..., j_k\}$ where $k < J$;
4: $V' := (V \setminus \mathcal{L}) \cup \mathcal{J}$;
5: $E_{\mathcal{L}} := \{\{v, u\} \mid (\{v, u\} \in E) \wedge (\{v, u\} \cap \mathcal{L} \neq \phi)\}$;
6: (* assume j_i joins at node v_i *)
7: $E' := (E \setminus E_{\mathcal{L}}) \cup (\bigcup_i \{j_i, v_i\})$;
8: OUTPUT: Graph $G' = (V', E')$

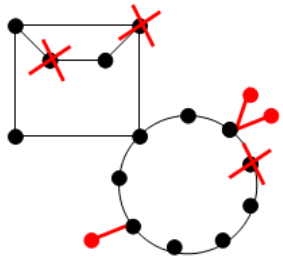

Figure 2.1: Operations of the Adversary

Usually, we start our analysis with a simple dynamic model where the repairing algorithm and the adversary take turns, see Figure 2.2. The goal of a repairing algorithm

is to re-establish — in a *phase of quiescence* — certain properties of the topology after the adversarial round.

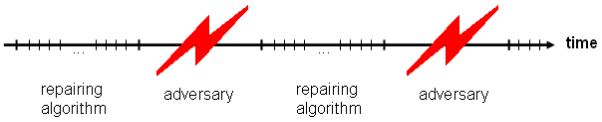

Figure 2.2: Repairing and Adversarial Rounds

More formally, we consider a predicate $\Gamma(G)$ which is true if and only if the topology G fulfills certain properties, for example "graph G is isomorphic to a hypercube". Let $\mathcal{A}_{adv}(J, L)$ denote an adversary which may insert at most J and remove at most L peers per adversarial phase. We say that $\mathcal{A}_{rep}$ is a repairing algorithm with respect to an adversary $\mathcal{A}_{adv}(J, L)$ if it re-establishes Γ after each adversary round, that is

$$\Gamma(G) \rightarrow \Gamma(\mathcal{A}_{rep}(\mathcal{A}_{adv}(J, L)(G))). \tag{2.1}$$

An interesting question in this context is: Given an adversary $\mathcal{A}_{adv}(J, L)$, how many rounds of quiescence are minimally needed by any $\mathcal{A}_{rep}$ that fulfills statement (2.1)? Mostly, we will focus on algorithms running in constant time.

After having studied this simple dynamic model, we analyze the more realistic model where the adversary acts concurrently to the repairing algorithm. In this context, we consider an adversary $\mathcal{A}_{adv}(J, L, \delta)$ which can join at most J and remove at most L peers *during any time period* of δ rounds. The repairing algorithm $\mathcal{A}_{rep}$ is required to maintain certain properties *all the time*, or, alternatively, to establish the properties whenever the adversary has been inactive for a certain time period. In the latter case, $\mathcal{A}_{rep}$ may *never* actually achieve $\Gamma(G)$. However, note that the fact that a repairing algorithm establishes $\Gamma(G)$ after a certain phase with no adversarial changes normally implies the existence of weaker predicates which are *always* true: For example the fact that certain variants of a ring are perfectly repaired by $\mathcal{A}_{rep}$ after a constant number of rounds of adversarial inactivity implies that the graph contains a ring as a subgraph *at any moment of time* — if the ring had been disconnected, repairing would take longer.

Finally, we assume peers to act always in perfect accordance with $\mathcal{A}_{rep}$, that is, Byzantine behavior of any kind is not considered here.

Chapter 3

k-Ring

3.1 Introduction

We begin our studies with a very simple topology which is based on the ring, the *drosophila melanogaster* of distributed computing. Note that a ring has some properties that are undesirable in a p2p system, for example it has a network diameter which is linear in the total number of nodes. However, our objective is to get insights to the nature of fault-tolerance *per se*, that is, independently of other criteria, and in this respect we consider the ring as a good starting point. Moreover, many current systems use a ring as a sub-component, for example also Chord [15].

The topology we will analyze is a special instance of an $\mathcal{N}$-circulant graph (Definition 3.1) called the k*-ring* (Definition 3.2).

Definition 3.1 ($\mathcal{N}$-Circulant Graph). *An* $\mathcal{N}$-circulant graph *of order* n *is a graph* $G = (V, E)$ *where* $V = \{0, ..., n-1\}$ *and* $E = \{\{i, i + j \ (mod\ n)\} \mid j \in \mathcal{N}\}$. $\mathcal{N}$ *is called the* connection set. *For example,* $\mathcal{N} := \{1, ..., \lfloor n/2 \rfloor\}$ *gives the complete graph* K_n.

Definition 3.2 (k-Ring). *We call an* $\mathcal{N}$*-circulant graph with connection set* $\mathcal{N} := \{1, 2, ..., k\}$ *a* k-ring. *Figure 3.1 shows an example for* k*=3.*

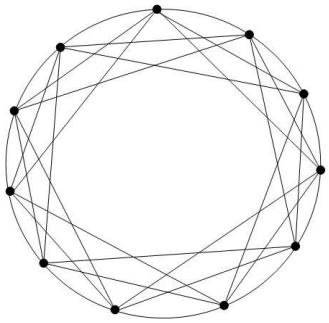

Figure 3.1: 3-Ring

3.2 Repairing in $O(1)$ Quiet Rounds

First, we consider the simple dynamic model introduced in Chapter 2 where the repairing algorithm and the adversary take turns. That is, after the adversary has inserted at most J and removed at most L peers, the repairing algorithm runs in a phase of quiescence. We concentrate on repairing algorithms which re-establish a perfect k-ring in a *constant* number of quiet rounds. In the following, if not stated otherwise, we always mean mod n.

Theorem 3.1. *If $L \geq k$, there is no algorithm which rebuilds the k-ring in $O(1)$ rounds of quiescence.*

Proof. Consider the case where $n \gg k$ and where k nodes in a row fail, i.e., nodes $\{i, i+1, ..., i+k-1\}$ for some i. Since node $i-1$ is not adjacent to node $i+k$, the ring is broken. To construct the edge $\{i-1, i+k\}$, a message has to be routed in the opposite direction along the broken ring, requiring time $\Omega(n)$. □

Consider the repairing algorithm presented in Algorithm 2. In the first round, every node sends a packet $\mathcal{P}$(list of joiners, hop-count) to all of its former neighbors.[1] The message sent to the i^{th} former neighbor is initialized with hop-count i. In the next rounds, every node v sorts the received packets by the hop-count in order to handle packets from closer nodes earlier. If it is not the first packet from the corresponding sender, the packet is simply ignored. v then updates its neighbors and forwards the packet if it is the furthest surviving neighbor of the sender and if the information is interesting for other nodes as well. We assume that a forwarded packet keeps the the *original* sender's address. Finally, when v has collected its new k neighbors on both sides, it assigns the correct neighbors to its joiners. The joiners of a node u are always inserted on the left side of u.[2] We now prove some properties of this algorithm.

Lemma 3.2. *Algorithm 2 assigns each node k neighbors on each side after finitely many steps if $L < k$.*

Proof. We prove that no packet $\mathcal{P}$ can get lost because of a missing link and that it will always be forwarded if it contains information needed by other nodes.

If less than k nodes crash, it holds that each node has at least one surviving neighbor on both sides. Moreover, a node forwards a packet $\mathcal{P}$ if, before receiving $\mathcal{P}$, $k - oldsize - 1 -$sizeof(joiners[])>0. Note that v needs $k - oldsize$ new neighbors on the side on which it received the packet, hence its neighbors on the opposite side need at most $k - oldsize - 1 -$sizeof(joiners[]) new neighbors, because they additionally have v and v's joiners as neighbors. □

Lemma 3.3. *Algorithm 2 preserves the global order of the surviving nodes on the ring and inserts the joiners of a node directly to its left.*

Proof. A packet from the right is appended by `append [pjoiners|`u`]`, a packet from the left by `append [`u`|pjoiners]`; therefore, every node on the ring sees the same order with respect to a single node u. It remains to prove that the packets itself are always well ordered. All packets that arrive in the same round are sorted by the hop-count, and closer nodes are inserted first. Moreover, it is not possible that a packet from

[1] Of course, because of the leaves, not all recipients are still alive. We assume that a message to a peer which has left the system is simply lost.

[2] See the `append` function.

Algorithm 2 k-Ring

```
1: (* node v *)
2: given: oldleft[k], oldright[k], joiners[];
3: newleft[k], newright[k]; (* empty *)
4: STEP 1:
5: for all neighbors i do
6:    SEND P(joiners[], i) TO oldleft[i]; SEND P(joiners[], i) TO oldright[i];
7: end for
8: STEP >1:
9: RECV all packets P(pjoiners[], pcount) FROM u;
10: (* u is the original sender of P *)
11: for all packets in order of increasing pcount do
12:    if packet from right neighbor then
13:       oldrsize := sizeof(newright[]);
14:       append [pjoiners|u] to newright[];
15:       if u is furthest surviving neighbor then
16:          if (k − oldrsize − 1−sizeof(joiners[]))> 0 then
17:             for i := 1 to k do
18:                SEND P(pjoiners[], pcount + i) TO oldleft[i];
19:             end for
20:          end if
21:       end if
22:    end if
23:    if packet from left neighbor then
24:       analogously, appending [u|pjoiners] to newleft[];
25:    end if
26: end for
27: if ¬((sizeof(newright[])≥ k) ∧ (sizeof(newleft[])≥ k)) then
28:    goto STEP >1;
29: end if
30: LAST STEP:
31: oldright[] := newright[];
32: for all elements of joiners[] do
33:    send joiners[i] its right neighbors [joiners[i − 1], ..., joiners[1], v, newright[1],
       newright[2], ...];
34: end for
35: oldleft[] := [joiners[1], joiners[2], ..., newleft[1], newleft[2], ...];
36: for all elements of joiners[] do
37:    send joiners[i] its left neighbors [joiners[i + 1], joiners[i + 2], ..., newleft[1],
       newleft[2], ...];
38: end for
```

a closer node w_1 arrives in a later round than a packet from a node w_2 which is further away. □

Lemma 3.4. *If $L < k$ and $J = any$, Algorithm 2 terminates after at most 4 steps.*

Proof. If there are only leaves and no joins, in the worst-case, a node v must get packets from node $v + 2k - 1$, which will be v's new k^{th} neighbor. In a perfect k-ring, a packet can travel k hops in one step, i.e., $s \cdot k$ hops in s steps. Moreover, a crashed node can delay a packet by at most 1 step, given that $L \leq k - 1$. Since there are at most $k - 1$ failures, a packet can get at least to node $v + s \cdot k - (k - 1)$ in s steps. For $s = 3$ we get $v + 2k + 1 > v + 2k - 1$, while $s = 2$ is not enough: $v + k + 1 < v + 2k - 1$ for $k \geq 3$. In the last step, a node integrates its joiners. □

Corollary 3.5. *If $L < k$ and $J = any$, there exists an algorithm which reconstructs the k-ring in time $O(1)$.*

Theorem 3.6. *If there are no joins, it is possible to repair $L \leq k - 1$ leaves in 2 steps. This is optimal for $k \geq 3$.*

Proof. At least 2 steps: Consider the case where $n \gg k$, $k \geq 3$ and $k - 1$ nodes in a row fail, i.e., nodes $\{i, i + 1, ..., i + k - 2\}$ for some i. $\{i - 1, i + k - 1\}$ is the only edge that bridges the gap. It is impossible to establish edge $\{i - 2, i + k\}$ in only one step.

At most 2 steps: Consider an algorithm which distributes information optimally: It sends all its neighbors (and not just itself, as was the case in Algorithm 2) to all its neighbors. This saves exactly one step compared to Algorithm 2. Moreover, it is not necessary to integrate the joiners, which saves another step compared to Lemma 3.4. . □

Theorem 3.7. *It is possible to repair $L \leq 1$ leaves in 1 step if $k = 2$. This is optimal.*

Proof. For the special case where $k = 2$, one step is sufficient. Consider the consecutive nodes ...$(i - 2)(i - 1)i(i + 1)(i + 2)$..., where node i leaves. All nodes $< (i - 2)$ or $> (i + 2)$ can keep their neighbors, and hence this takes no time at all. Edge $\{(i - 2), (i + 1)\}$ can be established in one step: $(i - 1)$ sends the corresponding information to $(i - 2)$ and $(i + 1)$. By symmetry, this holds also for the other side. □

Theorem 3.8. *It is possible to repair $L \leq k - 1$ leaves and J joins in 3 steps. This is optimal for $k \geq 3$.*

Proof. At least 3 steps: In the worst case, a node v must know a joiner of node $v + 2k - 1$, since $k - 1$ leaves took place in-between. Sending a packet over $2k - 1$ hops when $k - 1$ consecutive nodes have left may take up to three steps: One to the gap, one across the gap and one to the destination node. (If $k = 2$, either the sender or the destination node must be adjacent to the gap, therefore two steps are enough.) Figure 3.2 shows an example: It takes 3 steps to send the joiner of node $i - 2$ to node $i + k$ and vice versa.

At most 3 steps: Consider again Algorithm 2 where every node sends all its joiners in the first step. It takes 3 steps until every former node knows its new k neighbors on both sides. With a little change of our algorithm, it is also possible that all *joiners* know their $k + k$ neighbors: In the first step, a node informs its joiners about the other joiners. Moreover, in the second step, a node saves all joiners about which it has heard by the received packets. Thus, it can send the packets in the third round not only to a neighboring node v, but also to v's joiners directly. □

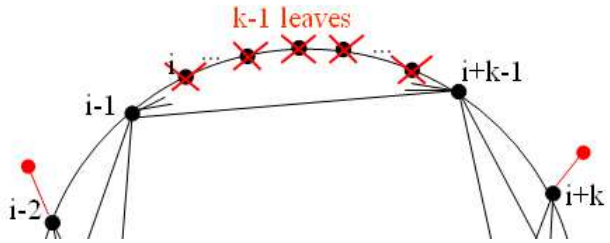

Figure 3.2: Figure for Theorem 3.8

Note that Theorems 3.6, 3.7 and 3.8 even hold if nodes store all senders from the packets they have seen and forward future packets directly to them. However, we won't consider this routing optimization further here.

3.3 Voluntary Leaves

So far, we have assumed that the peers leave the network quietly. This model has the advantage that it covers also the case where peers crash. However, it is possible to show that the system can tolerate more leaves in a scenario where peers never fail but always leave the network voluntarily. We call such a scenario a *goodbye model*, see Definition 3.3.

Definition 3.3 (Goodbye Model). *We call a setting where nodes call a function F to finalize before leaving* goodbye model. *In this function, it is possible to send a message to all neighbors, but the departure of the node can not be delayed.*

Theorem 3.9. *Under the goodbye model, the k-ring can be re-established in time $O(1)$ if and only if $L \leq 2k - 1$.*

Proof. $\Rightarrow$: In the worst case, a node v loses $2k - 1$ neighbors. We show that a link to node $u := v + 2k$ can be established in constant time. It can then be used to distribute neighbor information, similarly to Algorithm 2. Consider a finalize function F which simply sends all neighbor information to all neighbors. Node $v + k$ knows both v and u — they are its k^{th} neighbors. Calling F informs v about u and vice versa; the ring remains closed.

$\Leftarrow$: On the other hand, if $L \geq 2k$ nodes in a row fail, there is no node w which could inform the nodes adjacent to the gap. □

However, observe that there is a form of equivalence between the goodbye model and the model where peers may crash.

Theorem 3.10. *It is possible to simulate the goodbye model at the cost of one additional step in the repairing algorithm.*

Proof. The idea is to execute the commands in function F as a last step of the quiet phase. If a node v really leaves, all information it would send in the goodbye model already exists at its neighbors. On the other hand, if v remains alive, this information can simply be ignored. □

3.4 A Concurrent Model

The two models presented so far are both rather theoretical: In real networks, there is no time of quiescence during which an algorithm can repair the k-ring; rather, nodes may join or fail *concurrently* to the ring's maintenance. We refer to this setting as the *concurrent model*, see Definition 3.4.

Definition 3.4 (Concurrent Model). *In the* concurrent model, *an adversary can remove up to L and insert up to J nodes* in every round. *Hence, there are no quiet rounds and the repairing process runs concurrently to the adversary.*

There are many crucial differences between the concurrent model and the models with a phase of quiescence:

- If nodes v and u send a packet to each other at time t, it is not guaranteed that these packets also arrive at the same time t', see Example 3.1.
- It is not possible anymore to send a packet to a node v at time t such that v can forward it at time $t+1$, because it may have left the network by then. Thus, we have to use an algorithm in which *every* node forwards packets.
- When a packet arrives, the enclosed information may not be true anymore; for example, the sender may already have left in the meantime.
- In Algorithm 2, a node v integrates its joiners only by the end of the phase of quiescence. In this more dynamic model, it is crucial to make v's joiners independent of v as soon as possible to handle the case that v is removed from the network.

Example 3.1. *Consider Figure 3.3 and assume that node w leaves at the beginning of time $t = 2$, i.e.,* before *sending its messages of that round. Hence, v can send its packet via node w (arrival time at u: $t = 2$), while u's packet has to take another path (arrival time at v: $t = 3$).*

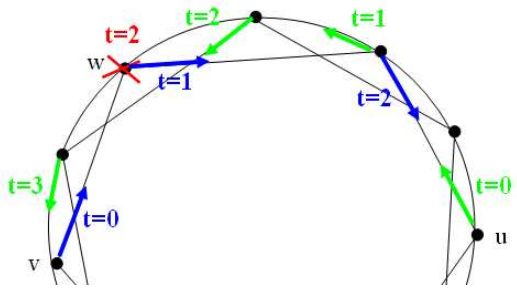

Figure 3.3: Figure for Example 3.1

Yet, under the assumption that all nodes are strongly synchronized modulo a certain number,[3] it is possible to prove a relation between the model with a phase of quiescence and the concurrent model.

[3] If the network starts with a single node, every node can simply assign its joiners the current state.

Theorem 3.11. *Under the assumption of strongly synchronized nodes, and if there are no joins, it holds that: If, in a instantaneous model, an edge e appears at time t, and if there is an algorithm $\mathcal{A}_q$ which reconstructs a perfect k-ring in s steps of quiescence given at most L leaves per adversary round, then there exists a concurrent algorithm $\mathcal{A}_c$ which creates e at time $t+s+1$ or earlier, if there are at most L leaves during any time interval of length $s+1$.*

Proof. Consider a concurrent algorithm where every node sends all its neighbor information to all its neighbors at times t iff $t \equiv 0 \pmod{s+1}$. These packets have a time-to-live of $s+1$ rounds. In all other rounds, *every* node simply forwards the incoming packets. (Broadcast is used to ensure that no packet will get lost because of node failures.) Further, assume that every node stores the information about the $k+L$ closest nodes on each side. For the ordering, a hop-count is used with respect to the situation when $t \equiv 0 \pmod{s+1}$ for the last time. First, we show that this algorithm would yield a perfect k-ring at times $t \equiv 0 \pmod{s+1}$ with respect to the *current* state of the system if a ping took no time.

Assume that at time 0, the k-ring is perfect. If L nodes leave at once, $\mathcal{A}_q$ reconstructs all edges e=(u, v) in s steps. But if it is possible to send information from u to v in s steps if all L nodes leave at once, it is also possible for $\mathcal{A}_c$ to send this information from u to v if the L leaves are distributed over the time interval of size s. For a reason that will become clear later in the text, consider a concurrent algorithm $\mathcal{A}_c$ where every node sends only information about itself in the first round, and not about its neighbors. This costs at most one additional step compared to $\mathcal{A}_q$. Thus, at time $s+1$, $\mathcal{A}_c$ has provided any node w with at least as many neighbors as $\mathcal{A}_q$. However, w does not know which of these neighbors are still alive, and hence a ping is performed, from which it is possible to derive the $2k$ current neighbors.

Of course, in a real system a ping takes two rounds. We can change $\mathcal{A}_c$ to handle also this more realistic scenario: As a simple solution, w treats all $k+L$ neighbors as if they were alive. At the cost of some extra storage — $k+2L$ neighbors have to be stored instead of $k+L$ — this ensures that an edge is always used with a delay of at most $s+1$ steps. A ping is therefore not necessary anymore. It remains to prove that $\mathcal{A}_c$ is still able to order the neighbors correctly. For this, we use again a hop-count which is now incremented with respect to all *potential* neighbors at the last time when $t \equiv 0 \pmod{s+1}$; this facilitates a consistent ordering, as these potential neighbors have been the correct nearest nodes at time $t-(s+1)$. □

It seems not possible to prove something similarly strong for the case of joins without changing the semantics. Hence, we use a weaker relation for the two models.

Theorem 3.12. *If there is an algorithm $\mathcal{A}_q$ with a time of quiescence of s steps tolerating J joins and L leaves per adversary round, there is also a concurrent repairing algorithm $\mathcal{A}_c$ which tolerates J joins and L leaves distributed over any time interval of size s, with the following property: If $\mathcal{A}_q$ establishes a perfect k-ring after each phase of quiescence, $\mathcal{A}_c$ constructs also a perfect k-ring if there have been no changes for the last $2s$ rounds.*

Proof. We adapt algorithm $\mathcal{A}_c$ as described in the proof of Theorem 3.11 to handle also joins. As a simple solution, assume that every node v delays its joiners until $t \equiv 0 \pmod{s}$: If v leaves the network earlier, its joiners have to try again at some other node. At times $t \equiv 0 \pmod{s}$, every node broadcasts itself plus the identifiers of its joiners; this information is then forwarded in the remaining rounds. A node which has heard

about the joiners of a neighboring node v in the first round sends all packets for v also to v's joiners in the remaining rounds for the case that v crashes. For the ordering we use hop-counts together with the rule to integrate joiners always on a predefined side.

Analogously to the proof of Theorem 3.11 it follows that at times $t \equiv 0 \pmod{s}$ every node knows about its k current neighbors on each side, except for the joiners that have arrived during the last s steps which have not been taken into account yet. Note that here we do not lose a step compared to $\mathcal{A}_q$ because of not sending the neighbors in the first round, so s steps are indeed sufficient. If after $t \equiv 0 \pmod{s}$ there are no changes during the next s consecutive rounds, $\mathcal{A}_c$ establishes a perfect k-ring. On the other hand, it takes at most s rounds until $t \equiv 0 \pmod{s}$ for the next time, and the claim holds. □

Chapter 4

Aggregation of Distributed State

DASIS [2], the *distributed approximative system information service*, allows to aggregate information in a p2p system. Similar ideas can be found in Astrolabe [16] and Willow [17]. After a short introduction to DASIS, we show that DASIS has some interesting properties which will be used by the dynamic and fault-tolerant systems based on the hypercube and on the skip graph presented in Chapters 5 and 6.

DASIS is built on top of the regular p2p structure. The basic idea is as follows: Every node v with bit string $b_0...b_k$ is considered to be an "expert" on all the sub-domains of all the prefixes of its bit string (that is, for $b_0...b_i, i \in [-1, k]$). The expert knowledge is constructed inductively through information exchange with the neighbor peers. The node is by definition an expert about its own sub-domain $b_1...b_k$. Also, the node v can deduce the state in sub-domain $b_0...b_i$ by aggregating its own knowledge on sub-domain $b_0...b_{i+1}$ (which is available by induction) with the knowledge provided by neighbor node u about sub-domain $b_0...\overline{b_{i+1}}$. In the end, node v can deduce the state of the whole system — the sub-domain of the empty prefix ε. Figure 4.1 gives an example: Node 001 knows about the state of the sub-domain 000 by its prefix buddy 000, about the sub-domain 01 by its buddy 011 and finally about the sub-domain 1 by its buddy 1100.

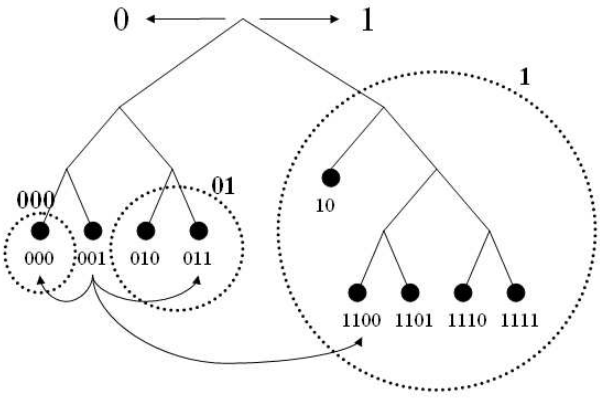

Figure 4.1: DASIS

Assume that the information or the state of each node is of type T. For aggregation,

we use a commutative function $\odot : TxT \mapsto T$. Further, let $initval \in T$ denote a default value which depends on the aggregation function and assume that $\Xi : V \mapsto T$ locally computes the new state of a node.

Example 4.1. *Consider a graph where every node stores a certain number of coins, and the goal is to compute the total number of coins in the system. In this case, $\Xi(v)$ returns the current number of coins at node v, the type T is integer ($T := \mathbb{N}$), we use the sum function ($\odot := +$) and the default value is $initval := 0$.*

Algorithm 3 gives a complete description of our synchronous information aggregation system based on DASIS. Note that k can be different for every node, but we assume that the bit string of a node is never a prefix of the bit string of another node.

Algorithm 3 DASIS Aggregation

```
1:  (* algorithm running on node b_0...b_k *)
2:  prefix_agg[b_0...b_k, b_0...b_{k-1}, ..., b_0, ε] = initval;
3:  prefix_agg[b_0...b_k] := Ξ(b_0...b_k) ;
4:  while true do
5:     (* new round *)
6:     for j := 0 to k do
7:        SEND prefix_agg[b_0...b_j] TO buddy of domain b_0...b_j-bar;
8:        s_{b_0...b_j-bar} := RECV FROM buddy of domain b_0...b_j-bar;
9:     end for
10:    for all j ∈ [0,k]: prefix_agg'[b_0...b_{j-1}] :=prefix_agg[b_0...b_j] ⊙ s_{b_0...b_j-bar};
11:    prefix_agg'[b_0...b_k] := Ξ(b_0...b_k);
12:    prefix_agg:=prefix_agg';
13: end while
```

Theorem 4.1. *Assuming a synchronous model where the nodes' states change at the beginning of every round, Algorithm 3 ensures that all nodes always store the same value prefix_agg[ε], where ε is the empty prefix.*

Proof. We prove by induction over the prefix length that all nodes sharing the same prefix $b_0...b_j$ have the same value prefix_agg[$b_0...b_j$].

$j = k$: Since there is only one node with prefix $b_0...b_k$, this is trivially true.

$j \to (j-1)$: We assume that all nodes sharing the prefix $b_0...b_j$ propagate the same value prefix_agg[$b_0...b_j$], say x, in every round. Analogously, all nodes with prefix $b_0...\overline{b_j}$ always have the same value prefix_agg[$b_0...\overline{b_j}$], say y. Therefore, according to Algorithm 3, all nodes with prefix $b_0...b_{j-1}$ will have prefix_agg'[$b_0...b_{j-1}$]$= x \odot y = y \odot x$. □

Of course, the aggregated value for the empty prefix does not reflect the currently correct state of the network, and prefix_agg[ε] may not even correspond to any real state the system has been in. However, it is easy to see that for the special case of a d-dimensional hypercube (see Chapter 5) or a perfect skip graph (see Chapter 6), where the nodes' identifiers all have the same length, the value of the empty prefix at time t is the correct state of the system of time $t - d$.

It is expensive to send all information to all neighbors in every step. However, it is also possible to run Algorithm 3 in an event-driven way: A node sends new information only if one of the aggregated prefix values changes. For obvious reasons, consistent

prefix values can also be achieved in this more efficient model (sending no information is simply interpreted as "no state change").

Finally, we analyze an asynchronous event-driven system, where it takes an arbitrary but finite amount of time to send a message over a link. In a distributed system without synchronized clocks, it is impossible to ensure that all nodes change a prefix value at the same moment of time. However, some crucial properties remain valid. Assume that each node stores for every prefix $b_0...b_i$ the aggregated values for the domains $b_0...b_i$ and $b_0...\overline{b_i}$ and the value $b_0...b_i \odot b_0...\overline{b_i}$. If the state of a node $v = b_0...b_k$ changes from $\Xi(v)$ to $\Xi(v)'$, v updates *all* its prefix values $\epsilon, b_0, ..., b_0...b_k$ as follows: prefix_agg$'[b_0...b_k] := \Xi(v)'$ and prefix_agg$'[b_0...b_{i-1}]$:=prefix_agg$'[b_0...b_i] \odot$ prefix_agg$[b_0...\overline{b_i}]$ for $i \in [0, k]$, where prefix_agg$[b_0...\overline{b_i}]$ stores the old value of the alternative prefix which hasn't changed. Then the neighbors are informed. Similarly, when a node hears about a change of an alternative prefix $b_0...\overline{b_i}$, it updates prefix_agg$[b_0...\overline{b_i}]$ and calculates the new values of the smaller prefixes, i.e., prefix_agg$'[b_0...b_{i-1}]$:= prefix_agg$[b_0...b_i] \odot$ prefix_agg $'[b_0...\overline{b_i}]$ and prefix_agg$'[b_0...b_{j-1}]$:= prefix_agg$'[b_0...b_j] \odot$ prefix_agg $[b_0...\overline{b_j}]$ for $j \in [0, i-1]$ and immediately afterwards sends a message to the corresponding prefix buddies.

Theorem 4.2. *In an asynchronous event-driven system, where it takes an arbitrary time to send a message over a link, and if the links are FIFO, it holds that the algorithm presented above provides the same aggregated value for the empty prefix to every node when there is no message on its way. Moreover, this value describes the correct state of the system at that moment of time.*

Proof. Every node u hears about a change $\Xi(v) \rightarrow \Xi(v)'$ of a node v exactly once, namely by the prefix buddy representing the sub-tree in which the change has happened. Moreover, since the edges are FIFO, a later change $\Xi(v)' \rightarrow \Xi(v)''$ at node v also arrives later at node u, because there is a unique path on which changes are propagated from v to u. Finally, under the assumption that local computations take no time and that messages are triggered immediately after a change, it holds that when there is no message on its way, then all changes have been accounted for. Thus, all nodes store the correct state of the system at that time. □

Consider the case where edges are not FIFO. Assume a system consisting of only two nodes $v := b_0$ and $u := \overline{b_0}$. Further, assume that v changes its state at times t and t', sending its new values $\Xi(v)$ and $\Xi(v)'$ to u. These are the only two changes that will ever happen. Without the FIFO property, $\Xi(v)'$ may arrive *before* $\Xi(v)$, causing u to store a different value for the empty prefix than v. Thus, it is impossible to guarantee consistent values with our algorithm if the edges are not FIFO. However, under the assumption that the aggregation function $\odot$ has the additional property that it is invertible, we can achieve consistency by sending *differences* instead of absolute values. So assume that all elements $\tau \in T$ have an inverse element τ^{-1} such that $\tau \odot \tau^{-1} = initval$ — for example, $\odot := +$ has this property: $\tau^{-1} \in \mathbb{N} := -1 \cdot \tau$, where $\cdot$ denotes the multiplication in $\mathbb{Z}$. If v sends $(\Xi(v))^{-1} \odot \Xi(v)'$ to u instead of $\Xi(v)'$, u finally stores the aggregated value $\Xi(v) \odot (\Xi(v))^{-1} \odot \Xi(v)' = (\Xi(v))^{-1} \odot \Xi(v)' \odot \Xi(v) = \Xi(v)'$, the same value as v.

Assume that in the beginning, it holds for every node $v := b_0...b_k \in V$: $\Xi(v) := initval$. If there is a change $\Xi(v) \rightarrow \Xi(v)'$, v sends $(\Xi(v))^{-1} \odot \Xi(v)'$ to all prefix buddies, and updates prefix_agg$'[b_0...b_i]$:= prefix_agg$[b_0...b_i] \odot (\Xi(v))^{-1} \odot \Xi(v)'$ for $i \in [-1, k]$. Similarly, receiving a change $(\Xi(w))^{-1} \odot \Xi(w)'$ from a prefix buddy u with prefix $b_0...\overline{b_i}$, v updates prefix_agg $[b_0...b_j]'$:= prefix_agg$[b_0...b_j] \odot (\Xi(w))^{-1} \odot$

$\Xi(w)'$ for $j \in [-1, i-1]$ and propagates $(\Xi(w))^{-1} \odot \Xi(w)'$ to the remaining (smaller) prefix buddies.

Theorem 4.3. *In an asynchronous event-driven system without the FIFO property, this algorithm guarantees that every node stores the same value for the empty prefix if there is no message on its way. Moreover, this value describes the correct state of the system at that moment of time.*

Proof. Obviously, every change $(\Xi(v))^{-1} \odot \Xi(v)'$ is sent exactly once to every node — hence, it is a *broadcast* — by the buddy-structure of DASIS. Since $\odot$ is commutative, every node stores the same and correct value in a time of quiescence. □

On the other hand, FIFO can also be achieved in a non-FIFO system by using a time-stamp per link: The sender simply tags each message with a number which is incremented for each message, and hence the receiver can sort the messages and handle them in a FIFO order. This solution has the advantage that $\odot$ does not have to be invertible.

Chapter 5

Hypercube

5.1 Introduction

The hypercube topology — see Definition 5.1 — is characterized by a logarithmic node degree and a logarithmic network diameter (in the number of nodes).

Definition 5.1 (d-Dimensional Hypercube). *A d-*dimensional hypercube *is a graph $G = (V, E)$, where $V = \{0,1\}^d$ and $E = \{(u_0...u_{d-1}, v_0...v_{d-1}) \mid \sum_0^{d-1} \mid u_i - v_i \mid = 1\}$, i.e., two nodes are adjacent if and only if their Hamming distance is 1. If two nodes u and v differ in their i^{th} bit, we say* nodes u and v are neighbors across dimension i. *Figure 5.1 gives an example for $d = 3$.*

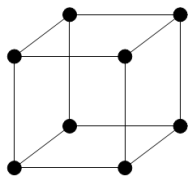

Figure 5.1: 3-Dimensional Hypercube

In this chapter, a *simulated hypercube* is studied, where the hypercube's nodes are represented by *several* peers. One goal of a repairing algorithm maintaining such a simulated hypercube is to guarantee that every node has at least one peer at every moment of time. A way to achieve this is to distribute the peers evenly among all nodes, equalizing potentially biased adversarial churn. Different questions concerning the uniform distribution of peers on a hypercube will be addressed in Chapter 5.2. Chapter 5.3 then presents two repairing algorithms for the simulated hypercube, one running in $\Theta(d)$ and the other one in $O(1)$ quiet rounds. Finally, Chapter 5.4 introduces a distributed hash table based on the simulated hypercube. The corresponding repairing algorithm maintains an efficient search structure against a *concurrent* adversary and ensures that no data is lost.

5.2 Token Distribution

Distributing peers evenly among the nodes of a simulated hypercube is a special instance of a *token distribution problem* — a fundamental problem to solve on a parallel computer or distributed network, first posed by Peleg and Upfal [8]. This problem has its origins in the area of load balancing, where the workload is modelled by a number of *tokens* or jobs of unit size; the main objective is to distribute the total load equally amongst the processors. Such load balancing problems arise in a number of parallel and distributed applications including job scheduling in operating systems, packet routing, large-scale differential equations and parallel finite element methods. More applications can be found in [14]. In this section, we will study different algorithms for the token distribution on a d-dimensional hypercube.

Basically, the goal of every token distribution algorithm is to minimize the maximum difference of the number of tokens at any two nodes in the graph, called the *discrepancy* (see Definition 5.2).

Definition 5.2 (Discrepancy). *Let a be the minimum number of tokens and let b be the maximum number of tokens at any node in a hypercube H. We say that hypercube H has a* discrepancy *of $b - a$.*

Token distribution problems can be classified into two categories: *static* and *dynamic*. In the static variant, an initial token distribution is given and the main objective is to re-distribute these tokens evenly amongst all nodes of the graph "in a phase of quiescence". In the dynamic variant on the other hand, the load is dynamic, that is, there are new tokens arriving and old tokens leaving all the times. We will consider the two variants in turn. Finally, an algorithm for the *weighted* token distribution problem on the hypercube is presented. Here, each node is characterized by a fixed weight and the goal is to distribute the tokens in a way that nodes with more weight get more tokens than nodes with little weight.

5.2.1 Static Token Distribution

First, we assume that the tokens are *fractional*, that is, infinitely divisible. Of course, in our case where tokens represent peers, this assumption is not realistic. However, it allows a first comparison of different token distribution algorithms, and, as we will see, simplifies also the analysis of the integer token distribution.

Consider the following algorithm: Every node v, having a tokens, sends $a/(d+1)$ tokens to each of its d neighbors. Unfortunately, although the algorithm converges, it may run forever.

Theorem 5.1. *This algorithm may never terminate with a perfect distribution.*

Proof. Consider a 2-dimensional hypercube where node 00 has a, 01 has b, 10 has c and 11 has d tokens at the beginning. After the first step ($i = 0$), node 00 has $\frac{a+b+c}{3}$, 01 has $\frac{a+b+d}{3}$, 10 has $\frac{a+c+d}{3}$ and 11 has $\frac{b+c+d}{3}$ tokens.

Note that after step i and if i is even, a node 00 has a numerator of the form $xa + xb+xc+(x-1)d$, 01 of the form $xa+xb+(x-1)c+xd$, 10 of the form $xa+(x-1)b+xc+xd$ and 11 of the form $(x-1)a+xb+xc+d$. If i is odd, 00 has a numerator of the form $(y+1)a+yb+yc+yd$, 01 has $ya+(y+1)b+yc+yd$, 10 has $ya+yb+(y+1)c+yd$ and 11 has $ya + yb + yc + (y + 1)d$. Therefore, before the balancing of step i, the discrepancy is $max\{|a-b|/3^i, |a-c|/3^i, |a-d|/3^i, |b-c|/3^i, |b-d|/3^i, |c-d|/3^i\}$.

Therefore, even if the convergence is — at least for the 2-dimensional case — very fast, there is no guarantee about the absolute discrepancy after a certain number of steps. □

The next algorithm we will study is shown in Algorithm 4: In step i (for i from 0 to $d-1$), every node $v := b_0...b_i...b_{d-1}$ having a tokens balances its tokens with only one adjacent node, namely its neighbor in dimension i, $u := b_0...\overline{b_i}...b_{d-1}$, having b tokens, such that both nodes end up with $\frac{a+b}{2}$ tokens. Figure 5.2 illustrates the execution of Algorithm 4 for $d = 3$. This algorithm has some nice properties.

Algorithm 4 Hypercube Token Distribution

```
1: (* algorithm running on node b_0...b_{d-1} *)
2: my_id := b_0...b_{d-1};
3: T_{my_id} :=tokens at this node;
4: for i := 0 to d − 1 do
5:    buddy_id := b_0...b̄_i...b_{d-1};
6:    SEND |T_{my_id}|/2 tokens to node buddy_id;
7:    update T_{my_id} accordingly;
8:    T_{buddy_id} :=REVC tokens from node buddy_id;
9:    T_{my_id} := T_{my_id} ∪ T_{buddy_id};
10: end for
```

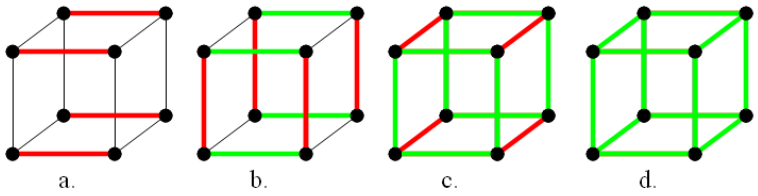

Figure 5.2: Token Distribution on Hypercube (Algorithm 4)

Theorem 5.2. *In case of fractional tokens, Algorithm 4 results in a perfect distribution after d steps.*

Proof. We prove by induction that after the i^{th} iteration, all nodes with the same postfix of length $d-1-i$ have the same number of tokens.

$i = 0$: Let v be node 0α having a tokens and u its neighbor 1α having b tokens, for an arbitrary $d-1$ bit vector α. After balancing, both nodes have $\frac{a+b}{2}$ tokens.

$i \to i+1$: Consider two i-dimensional sub-cubes H_0 and H_1 consisting of all nodes with postfixes 0α and 1α respectively, where α is an arbitrary bit string of length $d-1-(i+1)$. By the induction hypothesis, all nodes in H_0 have the same number of tokens, say a, and all nodes in H_1 have the same number of tokens, say b. After step $i+1$, all nodes in $V(H_0) \cup V(H_1)$ — sharing the postfix α — will have $\frac{a+b}{2}$ tokens. To see this, consider the nodes $v_0 = \zeta 0\alpha$, $v_1 = \eta 0\alpha \in V(H_0)$, and $u_0 = \zeta 1\alpha$, $u_1 = \eta 1\alpha \in V(H_1)$, where ζ and η are arbitrary bit vectors of length $i+1$. During the exchange of round $i+1$, v_0 balances with u_0 and v_1 with u_1. Obviously, all four nodes end up with $\frac{a+b}{2}$ tokens. □

Algorithm 4 is optimal in the following sense: There is no algorithm which can guarantee an upper bound on the absolute discrepancy in less than d steps, and after d steps, Algorithm 4 reaches the optimal discrepancy 0.

Theorem 5.3. *Every algorithm which, for every initial distribution, can balance the tokens with a discrepancy less than C needs at lest d steps, for some arbitrary C.*

Proof. Consider a d-dimensional hypercube H with 2^d nodes and the following initial distribution: Node 1^d has $C \cdot 2^d$ tokens, and all other nodes have no tokens at all. Having C tokens per node on average, there is always a node v on H with at least C tokens. On the other hand, no algorithm can move any tokens to node $u := 0^d$ in less than d steps. The discrepancy between v and u is at least C tokens. □

Hence, Algorithm 4 has some desirable properties if the tokens are fractional. Moreover, with a little modification, the same algorithm works also for integer tokens: If two nodes v (a tokens) and u (b tokens) are balanced, one node will end up with $\lceil (a+b)/2 \rceil$ tokens and the other one with $\lfloor (a+b)/2 \rfloor$ tokens. In the following, we will assume that this balancing is performed in two sub-steps: In the first sub-step, nodes u and v inform each other about the number of tokens they store, so in the second step the larger node can send (the arbitrarily rounded) difference to the smaller node.

Theorem 5.4. *This integer variant of Algorithm 4 yields a discrepancy of at most d after d steps.*

Proof. We show that after round i, the discrepancy among the nodes sharing the same postfix of length $d-1-i$ is at most $i+1$.

$i = 0$: Balancing a node v having a tokens with a node u having b tokens results in one node having $\lceil (a+b)/2 \rceil$ and the other one having $\lfloor (a+b)/2 \rfloor$ tokens, and it holds that $(\lceil (a+b)/2 \rceil - \lfloor (a+b)/2 \rfloor) \leq 1$.

$i \to i+1$: Consider two sub-cubes H_0 and H_1 that will be balanced in round $i+1$. By the induction hypothesis, the discrepancy within H_0 and H_1 is $\leq i+1$. Let a_{min}, resp. a_{max} be the minimal, resp. maximal number of tokens in a node in H_0. b_{min} and b_{max} are defined analogously for H_1. In the worst case, a maximum node in H_0 is balanced with a maximum node in H_1, which results in a node having $\lceil (a_{max} + b_{max})/2 \rceil$ tokens. Similarly, balancing the minimum nodes yields a node with $\lfloor (a_{min} + b_{min})/2 \rfloor$ tokens. The difference is at most $i+2$: $\lceil (a_{max} + b_{max})/2 \rceil \leq \lceil (a_{min} + i + 1 + b_{min} + i + 1)/2 \rceil = \lceil (a_{min} + b_{min})/2 \rceil + (i+1) \leq \lfloor (a_{min} + b_{min})/2 \rfloor + (i+2)$. □

Unfortunately, this bound is tight. Consider the following initial token distribution: Assign $a + f(v)$ tokens to node v, where $f : v \in V \mapsto$ (number of 1-bits of node v). Since neighboring nodes have a discrepancy of exactly 1, the token distribution algorithm may leave the distribution unchanged. Moreover, the integer version of Algorithm 4 does not even guarantee that *adjacent* nodes have a discrepancy of at most 1 in the final distribution.

Theorem 5.5. *Even adjacent nodes may have a discrepancy of d after running the integer version of Algorithm 4.*

Proof. We recursively construct such a worst case, where node 01^{d-1} has d tokens and its adjacent node 1^d has none.

$d = 2$: Consider the initial distribution where nodes 00, 10, 01, 11 have 2, 1, 1, 0 tokens respectively. In the first step, the pair (00, 10) is rounded up in favor of 00, and

(01, 11) in favor of 01; in the second step, (00, 01) to 01 and (10, 11) to 10. The claim holds.

$d-1 \rightarrow d$: Consider a $(d-1)$-dimensional sub-cube H_0. By the induction hypothesis, node $01^{d-2}0$ has $d-1$ tokens and $1^{d-1}0$ has none. With the same rounding strategy, it must be possible to construct a $(d-1)$-dimensional sub-cube H_1 where every node has exactly one token more than its corresponding node in H_0, i.e., $01^{d-2}1$ has d tokens and $1^{d-1}1$ has 1. In the next step, we balance the sub-cubes as follows: The pair ($01^{d-2}0$, $01^{d-2}1$) is rounded up in favor of $01^{d-2}1$, so 01^{d-1} has d tokens, and the pair ($1^{d-1}0$, $1^{d-1}1$) in favor of $1^{d-1}0$, yielding no tokens at all at node 1^d. All other pairs are rounded arbitrarily. □

So far, we have not specified which node gets the additional token if two nodes with an odd sum of tokens are balanced. We will show that if the additional token is assigned in a smart way to one of the two nodes, the final worst-case discrepancy is smaller. We will call an algorithm which decides for each edge which node gets the additional token a *rounding strategy* $\Upsilon : E \mapsto V$.

First, we present a rounding strategy which yields a discrepancy of at most $\lceil d/2 \rceil$: If a node $v = b_0...b_i...b_{d-1}$ is balanced with a node $u = b_0...\overline{b_i}...b_{d-1}$, the potential rounding token is assigned to v if $i \oplus (\bigoplus_{j=0}^{d-1} b_j)$ is even, otherwise it is assigned to u. Here, $\oplus$ and $\bigoplus$ are functions for addition modulo 2.

Theorem 5.6. *For this rounding strategy it holds that the final discrepancy never exceeds* $\lceil d/2 \rceil$.

Proof. Consider a d-dimensional hypercube, a node $v = b_0...b_i...b_{d-1}$ and its neighbor $u = b_0...\overline{b_i}...b_{d-1}$. Obviously, v and u have different parity sums, and hence the strategy is well-defined. Note that while $\bigoplus_{j=0}^{d-1} b_j$ is constant, i changes parity in every step, so v is rounded up at most every second time. We prove by induction that if d is even, the discrepancy is at most $d/2$. The correctness of the claim for the case where d is odd is a simple consequence.

$d = 0$: Trivial.

$d \rightarrow d+2$: Consider four d-dimensional sub-cubes H_{00} having a maximum of λ_{max}^{00} tokens, H_{10} (maximum λ_{max}^{10}), H_{01} (maximum λ_{max}^{01}) and H_{11} (maximum λ_{max}^{11}). By the induction hypothesis we know that the minimum numbers of tokens are at least $\lambda_{min}^{00} = \lambda_{max}^{00} - \frac{d}{2}$, $\lambda_{min}^{10} = \lambda_{max}^{10} - \frac{d}{2}$, $\lambda_{min}^{01} = \lambda_{max}^{01} - \frac{d}{2}$ and $\lambda_{min}^{11} = \lambda_{max}^{11} - \frac{d}{2}$. First, H_{00} is balanced against H_{10} and H_{01} against H_{11}, then H_{00} against H_{01} and H_{10} against H_{11}.

By our construction the new minimum node has at least

$$min\left\{\left\lfloor \frac{\lceil \frac{\lambda_{max}^{00}+\lambda_{max}^{10}}{2} \rceil + \lfloor \frac{\lambda_{max}^{01}+\lambda_{max}^{11}}{2} \rfloor}{2} \right\rfloor - \frac{d}{2}, \left\lfloor \frac{\lfloor \frac{\lambda_{max}^{00}+\lambda_{max}^{10}}{2} \rfloor + \lceil \frac{\lambda_{max}^{01}+\lambda_{max}^{11}}{2} \rceil}{2} \right\rfloor - \frac{d}{2}\right\}$$

tokens, the new maximum node has at most

$$max\left\{\left\lceil \frac{\lceil \frac{\lambda_{max}^{00}+\lambda_{max}^{10}}{2} \rceil + \lfloor \frac{\lambda_{max}^{01}+\lambda_{max}^{11}}{2} \rfloor}{2} \right\rceil, \left\lceil \frac{\lfloor \frac{\lambda_{max}^{00}+\lambda_{max}^{10}}{2} \rfloor + \lceil \frac{\lambda_{max}^{01}+\lambda_{max}^{11}}{2} \rceil}{2} \right\rceil\right\}$$

It is easy to verify that the discrepancy grows at most by 1:

$$\left\lceil \frac{\lceil \frac{\lambda_{max}^{00}+\lambda_{max}^{10}}{2} \rceil + \lfloor \frac{\lambda_{max}^{01}+\lambda_{max}^{11}}{2} \rfloor}{2} \right\rceil \leq \left\lfloor \frac{\lceil \frac{\lambda_{max}^{00}+\lambda_{max}^{10}}{2} \rceil + \lfloor \frac{\lambda_{max}^{01}+\lambda_{max}^{11}}{2} \rfloor}{2} \right\rfloor + 1$$

and

$$\left\lceil \frac{\lceil \frac{\lambda^{00}_{max}+\lambda^{10}_{max}}{2} \rceil + \lfloor \frac{\lambda^{01}_{max}+\lambda^{11}_{max}}{2} \rfloor}{2} \right\rceil \leq \left\lfloor \frac{\lfloor \frac{\lambda^{00}_{max}+\lambda^{10}_{max}}{2} \rfloor + \lceil \frac{\lambda^{01}_{max}+\lambda^{11}_{max}}{2} \rceil}{2} \right\rfloor + 1$$

□

Is there a rounding strategy that defines *off-line*, i.e., independently of the initial distribution, for each edge $e = \{u, v\}$ on the hypercube whether u or v gets the overflow token and that guarantees an optimal discrepancy of at most one after d steps of our algorithm? Consider two $(i-1)$-dimensional sub-cubes H_0 and H_1 with $\forall v \in V(H_0) \Rightarrow v$ has either a or $a+1$ tokens and $\forall u \in V(H_1) \Rightarrow u$ has either b or $b+1$ tokens. We require that Υ distributes the tokens in H_0 and H_1 such that "a-nodes" are never balanced with "b-nodes" (resulting in a node with $\lfloor \frac{a+b}{2} \rfloor$) when at the same time $(a+1)$-nodes are balanced against $(b+1)$-nodes (resulting in a node with $\lceil \frac{a+b}{2} \rceil + 1$), which would yield a discrepancy of two if $(a+b) \equiv 1 \pmod 2$. Assume that H_0 has A tokens in total and H_1 has B. Moreover, let $A \bmod 2^{i-1} = \alpha$ and $B \bmod 2^{i-1} = \beta$, i.e., H_0 has α nodes with $a+1$ tokens and H_1 has β nodes with $b+1$ tokens. A possible strategy for Υ would be to place H_0's additional tokens at nodes 0, ..., $\alpha - 1$ and H_1's additional tokens at nodes $2^i - 1$, ..., $2^i - \beta$, given an arbitrary order on hypercubes.

Lemma 5.7. *There is no off-line rounding strategy Υ which places tokens with respect to such an order for every initial distribution. More generally, it is impossible to avoid balancing maximum against maximum and minimum against minimum at the same time, yielding a final discrepancy greater than one.*

Proof. Let $|H| :=$ (number of tokens in H) mod 2^d. Consider two 2-dimensional sub-cubes H_0 and H_1 with $|H_i| = 2$ for $i \in \{0, 1\}$. There are two possible configurations to avoid a discrepancy of two in the 3-dimensional hypercube: Place H_0's rounding tokens on the top at 01 and 11 (and hence H_1's rounding tokens at the bottom), see Figure 5.3.a on the left, or place them along one of the diagonals, e.g. 01 and 10 (H_1 uses the other diagonal), see 5.3.a on the right. In the upcoming section, we will discuss both possibilities in turn.

Tokens at 01 and 11: We use only the left half of Figure 5.3. Consider the initial distribution of Figure 5.3.b. Note that there is no rounding strategy to achieve the desired configuration if we first balance dimension 1 and then dimension 0.

Hence, we can concentrate on the case where we balance dimension 0 first. There are only two possible rounding strategies, depicted in 5.3.c and 5.3.d, where the arrow points to the node which gets the rounding token if there is any. We consider these cases in turn.

For $|H_0| = 2$, we always achieve configuration 5.3.a. If $|H_0| = 3$, there are two possible configurations to place all three rounding tokens, see 5.3.e and 5.3.f. However, it is not possible for every initial distribution to place the rounding tokens this way: 5.3.g gives a counter example for 5.3.e, 5.3.h is a counter example for 5.3.f.

The same holds for 5.3.d: Here, 5.3.i gives a counter example for 5.3.e and 5.3.j is a counter example for 5.3.f.

Tokens at 01 and 10: For this section, we use the right half of Figure 5.3. For symmetry reasons, we can concentrate on balancing dimension 0 first and then dimension 1. Consider the initial distribution of Figure 5.3.b. Again, there are two possible

rounding strategies, shown 5.3.c in and 5.3.d. For the case where $|H_0| = 3$, the counter examples for 5.3.e and 5.3.f are shown in 5.3.g, 5.3.h, 5.3.i and 5.3.j.

Hence, no rounding strategy that prevents balancing maximum against maximum and minimum against minimum for the case $|H_0| = 2$ and $|H_1| = 2$ can ensure the position of the third additional token in H_0 if $|H_0| = 3$, which — as a consequence — may collide with the additional token of H_1 given $|H_1| = 1$. □

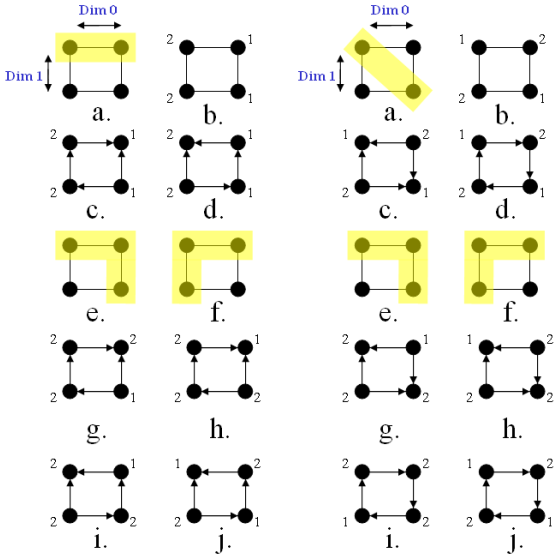

Figure 5.3: Rounding Strategy Υ for H_0

It is even possible to show that for every off-line rounding strategy the discrepancy grows *linearly* in d.

Theorem 5.8. *For every off-line rounding strategy Υ, the worst-case discrepancy is in $\Omega(d)$.*

Proof. We show that for any rounding strategy instance, it is possible to construct an initial distribution which yields a discrepancy of at least $\lceil \frac{d+2}{4} \rceil$. Two $(i-1)$-dimensional sub-cubes H_0 and H_1 are balanced on 2^{i-1} edges, k pointing to H_0 (rounding up in favor of H_0), for $k \in \{0, ..., 2^{i-1}\}$ and $2^{i-1} - k$ pointing to H_1. Without loss of generality assume that $k \geq 2^{i-2}$. Further, let $\mathcal{S}$ be the set of edges pointing to H_0 and let $\mathcal{S}_{H_b}$ be the endpoints of these edges which belong to the sub-cube H_b for $b \in \{0, 1\}$. In our initial distribution, $a + \theta_e$ tokens are assigned to nodes in $\mathcal{S}_{H_0}$, $a + \theta_e - 1$ to the nodes in $\mathcal{S}_{H_1}$ and $a + \theta_e$ to all other nodes, where θ_e is a constant defined for each edge e such that the largest node has exactly a tokens. Figure 5.4 shows such a distribution where $t_i := a + \theta_e$: Sub-cube H_0 has at least 2^{d-2} incoming edges, hence, according to our initial distribution, it gets at least 2^{d-2} tokens more than H_1.

More formally, consider the series of sub-cubes $(H^{(0)}, ..., H^{(d)})$ recursively defined as follows: $H^{(d)}$ is the whole d-dimensional hypercube; for $i \in \{1, ..., d\}$, $H^{(i-1)}$ is the $(i-1)$-dimensional sub-cube (across dimension $d-1-i$) of $H^{(i)}$ which has more incoming edges. Consider a node $v = b_0...b_{d-1-i}...b_{d-1}$ and its neighbor

$u = b_0...\overline{b_{d-1-i}}...b_{d-1}$ where it holds that $(v \in V(H^{(i)})) \wedge (u \in V(H^{(i-1)}))$ (this implies that $v \notin V(H^{(i-1)})$) and $i > 0$. In the initial token distribution,

$$\text{tokens at } v := \begin{cases} \text{tokens at u minus 1} & \text{, if } u = \Upsilon(v,u) \\ \text{tokens at u} & \text{, if } v = \Upsilon(v,u) \end{cases}$$

If $i = 0$, v is assigned a tokens. Hence, balancing the two sub-cubes of $H^{(i)}$ requires no token transfer at all. Given this description of the initial token distribution, the total number of tokens in the graph can be computed. We will show that the result implies the existence of a node having at most $a - \lceil \frac{d+2}{4} \rceil$ tokens after the distribution; together with the fact that there exists a node having a tokens, the claim follows.

We do a bottom-up analysis, always counting the number of tokens in the biggest i-dimensional component $H^{(i)}$. $H^{(0)}$ has a tokens. Adding a node with $a-1$ tokens gives $2a-1$ tokens in $H^{(1)}$. $H^{(1)}$ has at least one incoming edge, and hence it is merged with a hypercube having at most $2a-2$ tokens, yielding $4a-3$ tokens in $H^{(2)}$. $H^{(3)}$ has at least two incoming edges, therefore it has at most $(4a-3)+(4a-5) = 8a-8$ tokens. And so forth.

Let δ_i be $a2^i$ minus the total number of tokens in $H^{(i)}$, that is, $\delta_0 = 0$, δ_1=1, δ_2=3, δ_3=8 etc. So there are at most $a2^d - \delta_d$ tokens in the whole hypercube by our construction, $a2^{d-1} - \delta_{d-1}$ in the larger half and $(a2^d - \delta_d) - (a2^{d-1} - \delta_{d-1})$ in the smaller half of the last step (across dimension 0). We have

$$\begin{aligned} \delta_k &= 2\delta_{k-1} + 2^{k-2} = ... = \\ &= 2^j \delta_{k-j} + j2^{k-2} = ... = \\ &= 2^{k-1}\delta_1 + (k-1)2^{k-2} = (k+1)2^{k-2}. \end{aligned}$$

Since there are $\frac{(a2^d-(d+1)2^{d-2})-(a2^{d-1}-d2^{d-3})}{2^{d-1}} = a - \frac{d+2}{4}$ tokens on average in the smaller half $V(H^{(d)}) \setminus V(H^{(d-1)})$, there must be a node having at most $\lfloor a - \frac{d+2}{4} \rfloor = a - \lceil \frac{d+2}{4} \rceil$ tokens.

Alternatively, it is possible to calculate the total number of tokens in the smaller half of the last step directly: $\widetilde{\delta}_0 = 1$, $\widetilde{\delta}_1 = 2$, $\widetilde{\delta}_2 = 5$, etc. and

$$\begin{aligned} \widetilde{\delta}_k &= 2\widetilde{\delta}_{k-1} + 2^{k-2} = ... = \\ &= 2^j \widetilde{\delta}_{k-j} + j2^{k-2} = ... = \\ &= 2^{k-1}\widetilde{\delta}_1 + (k-1)2^{k-2} = (k+3)2^{k-2}. \end{aligned}$$

For $k = d-1$, we get again $\lceil \frac{(d+2)2^{d-3}}{2^{d-1}} \rceil = \lceil \frac{d+2}{4} \rceil$. □

Remark 5.1. *The technique of the proof of Theorem 5.8 can be used to show that the bound of Theorem 5.4 is tight. For this, we consider a hypercube where* all *two i-dimensional sub-cubes H_0 and H_1 which are balanced have 2^i edges in the* same *direction. The discrepancy per edge is exactly 1 and the initial distribution is a* fix-point *for this rounding instance, that is, the distribution is never changed during the execution of our token distribution algorithm. Starting with a node v having a tokens, the biggest 1-dimensional hypercube has $2a-1$ tokens, the biggest 2-dimensional hypercube has $4a-4$, i.e., $\delta_0 = 0$, $\delta_1 = 1$, $\delta_2 = 4$, $\delta_3 = 12$ and so on. We have*

$$\delta_k = 2\delta_{k-1} + 2^{k-1} = k2^{k-1}$$

Hence there are $a - \frac{d}{2}$ tokens in the whole hypercube on average. Because of the symmetric structure of the initial distribution, the minimal node has $a - d$ tokens.

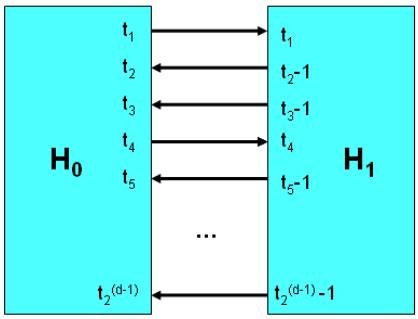

Figure 5.4: Initial Distribution for Discrepancy $\Omega(d)$

Next, we will study an *online* rounding strategy which depends on the initial distribution. A simple idea is to assign the additional token to the node which had less tokens before balancing. Unfortunately, given an initial distribution where every node v has $a+f(v)$ tokens, where $f : v \in V \mapsto$ (number of 1-bits of node v), the final discrepancy is as large as d. To see this, note that after balancing across dimension i, the distribution is isomorphic to the distribution before balancing with $b_0...b_i...b_{d-1} \mapsto b_0...\overline{b_i}...b_{d-1}$. There may smarter online rounding strategies which achieve smaller discrepancies. However, it seems that these strategies require more information on the initial distribution than the one that can be obtained locally by the neighboring nodes. We will not consider this approach further here.

Last but not least, Υ could be computed randomly, assigning the potential rounding token to one of the endpoints of the edge according to the outcome of a perfect coin flip. First, we show that that the probability that a d-dimensional hypercube has a discrepancy of d with a randomized rounding strategy is very small. We need some helper lemmas.

Lemma 5.9. *If during the execution of the integer version of Algorithm 4 on hypercube H there is an i-dimensional sub-cube with a discrepancy less than i, the final discrepancy is less than d.*

Proof. First, we show that combining two sub-cubes H_0 and H_1 with discrepancies Δ_{H_0} and Δ_{H_1} respectively results in a discrepancy of at most $\max\{\Delta_{H_0}, \Delta_{H_1}\}$ if $\Delta_{H_0} \neq \Delta_{H_1}$. Without loss of generality, assume that $\Delta_{H_0} > \Delta_{H_1}$. Let λ_b be the minimum and $\widehat{\lambda_b}$ the maximum node in H_b, for $b \in \{0, 1\}$. In the worst case, we compare the minimums and the maximums and have one rounding error, hence the new minimum node has $\lfloor \frac{\lambda_0+\lambda_1}{2} \rfloor$ and the maximum node has $\lceil \frac{\widehat{\lambda_0}+\widehat{\lambda_1}}{2} \rceil = \lceil \frac{\lambda_0+\Delta_{H_0}+\lambda_1+\Delta_{H_1}}{2} \rceil \leq \lceil \frac{\lambda_0+\Delta_{H_0}+\lambda_1+\Delta_{H_0}-1}{2} \rceil = \Delta_{H_0} + \lceil \frac{\lambda_0+\lambda_1-1}{2} \rceil = \Delta_{H_0} + \lfloor \frac{\lambda_0+\lambda_1}{2} \rfloor$.

It follows that the discrepancy may only grow if we balance two sub-cubes of the same discrepancy. On the other hand, balancing two sub-cubes of the same discrepancy can increase the discrepancy at most by one. □

Lemma 5.10. *If the final discrepancy after the integer token distribution on a d-dimensional hypercube is d, there is exactly one maximum and one minimum node.*

Proof. Proof by induction.

$d = 1$: The claim holds trivially.

$d-1 \rightarrow d$: From Lemma 5.9 we know that H may only have a discrepancy of d if the two $(d-1)$-dimensional sub-cubes H_0 and H_1 have a discrepancy of $d-1$. Moreover, by our induction hypothesis, H_0 and H_1 both have exactly one node with λ_b tokens and one with $\lambda_b + (d-1)$, for $b \in \{0,1\}$ and some arbitrary $\lambda_b \in \mathbb{N}$. The resulting d-dimensional hypercube has a discrepancy of d if and only if $parity(\lambda_0 + (d-1)) \neq parity(\lambda_1 + (d-1))$, and if the two maximum nodes are balanced against each other, resulting in exactly one node with $\lfloor \frac{\lambda_0+\lambda_1}{2} \rfloor + d$ tokens. Similarly, the new minimal node can only be either the minimal node of H_0 or the minimal node of H_1 in order that $H := H_0 \cup H_1$ has a discrepancy of d. □

We can now calculate the probability that the random rounding strategy yields a discrepancy of d tokens.

Theorem 5.11. *Let ε denote the event that a d-dimensional hypercube has a final discrepancy d under a randomized rounding strategy. It holds that $Pr[\varepsilon] \leq (\frac{1}{2})^{2^d+2^{d-1}-d-1}$.*

Proof. Let $p(i)$ be the probability that an i-dimensional hypercube has a discrepancy of i. It holds that $p(i) = \frac{1}{2}\frac{1}{2^{i-1}}p(i-1)p(i-1)$ for $i \in \{1...d\}$: The probability that an i-dimensional hypercube has a discrepancy of i is the probability that both its sub-cubes have a discrepancy of $i-1$ times the probability that the parity of the maximum nodes is different and rounding takes place, times the probability that both maxima are balanced against each other. Since we do not consider the minima at all, we may overestimate. As a recursion basis, we know that $p(0) = 1$, so $p(1) = \frac{1}{2}$ and $p(2) = \frac{1}{16}$ etc.

We have

$$\begin{aligned}
p(d) &= \frac{1}{2}\frac{1}{2^{d-1}}p(d-1)p(d-1) = ... = \\
&= \frac{1}{2}...\left(\frac{1}{2}\right)^{2^{i-1}} \frac{1}{2^{d-1}}...\left(\frac{1}{2^{d-i}}\right)^{2^{i-1}} (p(d-i))^{2^i} = ... = \\
&= \frac{1}{2}...\left(\frac{1}{2}\right)^{2^{d-1}} \frac{1}{2^{d-1}}...\left(\frac{1}{2^0}\right)^{2^{d-1}} (p(0))^{2^d} = \\
&= \prod_{i=0}^{d-1}\left(\frac{1}{2}\right)^{2^i} \prod_{i=1}^{d-1}\left(\frac{1}{2^{d-i}}\right)^{2^{i-1}} = \\
&= \left(\frac{1}{2}\right)^{\sum_{i=0}^{d-1} 2^i} \left(\frac{1}{2}\right)^{2^d-d-1} \leq \left(\frac{1}{2}\right)^{2^{d-1}} \left(\frac{1}{2}\right)^{2^d-d-1}.
\end{aligned}$$

□

Of course, a statement about the expected discrepancy is of greater interest.

Theorem 5.12. *Let X be the random variable for the final discrepancy in a d-dimensional hypercube. It holds that $E[X] < 3$.*

Proof. It is possible to apply our technique of the proof for Theorem 5.8 also here. However, the difference of the number of tokens in two sub-cubes is now given by a random variable.

In our randomized rounding scheme, the rounding direction of each edge it determined by a perfect coin flip. Let X_i be the random variable denoting the number of incoming edges of a $(d-1-i)$-dimensional sub-cube. Since there are 2^{d-1-i} edges

connecting two $(d-1-i)$-dimensional sub-cubes H_0 and H_1, X_i has the binomial distribution $X_i \sim \mathcal{B}(2^{d-1-i}, 1/2)$. If rounding happens on every edge, the sub-cubes H_0 and H_1 differ by δ_i tokens after balancing, where $\delta_i := 2 \cdot |X_i - E[X_i]|$, and the random variables δ_i are mutually independent.

Assume that the in the final distribution the maximum node v has a tokens. We show that the average number of tokens in the system is at least $a - \frac{\sum_{i=0}^{d-1} 2^i E[\delta_i]}{2^d}$, by counting the average number of tokens in the biggest i-dimensional sub-cubes which contain v for $i \in [0, d]$. Obviously, the 0-dimensional sub-cube consists only of v and has a tokens in total. In the next step, this sub-cube is combined with another 0-dimensional sub-cube having $a - \delta_{d-1}$ tokens. The resulting 2-dimensional hypercube having $2a - \delta_{d-1}$ tokens is combined with a hypercube having δ_{d-2} tokens less, so there are $4a - 2\delta_{d-1} - \delta_{d-2}$ tokens in total, and so forth. Obviously, after d steps, we have $a2^d - \sum_{i=0}^{d-1} 2^i \delta_i$ tokens in the whole d-dimensional hypercube, $a - \frac{\sum_{i=0}^{d-1} 2^i \delta_i}{2^d}$ on average.

It remains to calculate $E[\delta_i]$, which is twice the mean deviation of the binomial distributed random variable X_i. We have

$$E[\delta_i] = 2 \cdot \frac{1}{2^{2^{d-1-i}}} \cdot \sum_{j=0}^{2^{d-1-i}} \binom{2^{d-1-i}}{j} \left| j - \frac{2^{d-1-i}}{2} \right| \overset{(1)}{\leq} 2\sqrt{\pi 2^{d-1-i}}$$

Inequality (1) is due to Chernoff, see Appendix A.

Hence

$$\begin{aligned} \sum_{i=0}^{d-1} 2^i E[\delta_i] &= 2\sqrt{\pi} \sum_{i=0}^{d-1} 2^i 2^{\frac{d-1-i}{2}} = 2\sqrt{\pi} 2^{\frac{d-1}{2}} \sum_{i=0}^{d-1} 2^{\frac{i}{2}} \\ &= 2\sqrt{\pi} 2^{\frac{d-1}{2}} \sum_{i=0}^{d-1} (\sqrt{2})^i = 2\sqrt{\pi} 2^{\frac{d-1}{2}} \frac{(\sqrt{2})^d - 1}{\sqrt{2} - 1} \\ &\leq \frac{\sqrt{\pi}}{\sqrt{2} - 1} 2^d \end{aligned}$$

Thus, having a node with a tokens, the average number of tokens is at least $a - \frac{\sqrt{\pi}}{\sqrt{2}-1}$. By symmetry, the expected final discrepancy is twice as much. Moreover, according to Stirling's approximation, see also Appendix A, we overestimated by a factor of at least π, and therefore the total discrepancy is $2 \cdot \frac{1}{\pi} \cdot \frac{\sqrt{\pi}}{\sqrt{2}-1} \doteq 2.73$. □

5.2.2 Dynamic Token Distribution

We turn our attention now to the more interesting *dynamic* token distribution problem and assume that at the beginning of each step, a "token adversary" inserts at most J and removes at most L tokens at arbitrary nodes. We consider again our token distribution algorithm (Algorithm 4), which in the dynamic case cycles forever over the dimensions, that is, after balancing dimension $d-1$ it will again start with dimension 0. It can be shown that for any initial distribution and in the case of non-fractional tokens, after the first d steps of Algorithm 4, the discrepancy will forever be bounded by $d + 2J + 2L$.

Lemma 5.13. *In the dynamic* fractional *token distribution, the number of tokens at a node depends only on the adversarial token insertions and deletions of the last d steps and on the total number of tokens in the system. It does not depend on the history of changes that lie more in the past!*

Proof. Assume that a total amount of T tokens are distributed in two different ways on the d-dimensional hypercube. According to Theorem 5.2, each node has exactly $\frac{T}{2^d}$ tokens after d steps in the absence of an adversary. On the other hand, the token insertions and removals of the adversary that happen in-between can be treated as an independent superposition, as the corresponding operations are all linear. □

Based on Lemma 5.13, we prove that the discrepancy of the dynamic integer token distribution is at most d tokens larger than the discrepancy in the fractional case at every moment of time.

Lemma 5.14. *Consider an arbitrary node $v \in V(H)$ in the d-dimensional hypercube H. Let $|v|_t^{int}$ denote the number of tokens at v for the (dynamic) integer token distribution at time t and analogously $|v|_t^{frac}$ for the (dynamic) fractional token distribution. Moreover, an arbitrary adversary $\mathcal{A}_{adv}(J, L, \delta)$ is assumed. It holds that $\forall t : |(|v|_t^{int} - |v|_t^{frac})| \leq \frac{d}{2}$.*

Proof. Assume that initially, i.e., for $t = 0$, the integer and the fractional distribution are the same. For symmetry reasons, it is sufficient to show the upper bound $|v|_t^{int} \leq |v|_t^{frac} + \frac{d}{2}$. First, we prove by induction that $|v|_t^{int} \leq |v|_t^{frac} + \frac{t}{2}$ for the first t steps.

$t = 1$: If v has a tokens and is balanced with node u having b tokens, we have $|v|_1^{int} \leq \lceil \frac{a+b}{2} \rceil$ and $|v|_1^{frac} = \frac{a+b}{2}$ and it holds that $|v|_1^{int} - |v|_1^{frac} \leq \frac{1}{2}$. Note that this remains true when the adversary now changes the number of tokens at v by ψ tokens: $(|v|_1^{int} + \psi) - (|v|_1^{frac} + \psi) = |v|_1^{int} - |v|_1^{frac} \leq \frac{1}{2}$.

$t \to t + 1$: Consider two nodes u and v which are balanced. It holds that

$$\begin{aligned} |v|_{t+1}^{int} &\leq \left\lceil \frac{|v|_t^{int} + |u|_t^{int}}{2} \right\rceil \leq \left\lceil \frac{\left\lfloor |v|_t^{frac} + \frac{i}{2} \right\rfloor + \left\lfloor |u|_t^{frac} + \frac{i}{2} \right\rfloor}{2} \right\rceil \\ &\leq \frac{\left\lfloor |v|_t^{frac} + \frac{i}{2} \right\rfloor + \left\lfloor |u|_t^{frac} + \frac{i}{2} \right\rfloor}{2} + \frac{1}{2} \\ &\leq \frac{|v|_t^{frac} + |u|_t^{frac} + i + 1}{2} = |v|_{t+1}^{frac} + \frac{i+1}{2}. \end{aligned}$$

The second inequality is due to the induction hypothesis and the fact that $|v|_t^{int}$ and $|u|_t^{int}$ are integers. Again, the difference between $|v|_t^{int}$ and $|v|_t^{frac}$ is not affected by the activity of the adversary.

Thus, a node can deviate at most by $\frac{d}{2}$ from its fractional value in d steps if the initial integer and fractional distributions are the same. However, this also holds for all later steps. To see this, consider the state of the system at time $\hat{t} = d + i$ for $i = 1, 2, ...$ Let's define a fractional distribution which is the same as the integer distribution at time $\hat{t} - d$: $|v|_{\hat{t}-d}^{\widetilde{frac}} := |v|_{\hat{t}-d}^{int}$. As we have already shown, at time $\hat{t}$, it holds that $|v|_{\hat{t}}^{int} \in [|v|_{\hat{t}}^{\widetilde{frac}} - \frac{d}{2}, |v|_{\hat{t}}^{\widetilde{frac}} + \frac{d}{2}]$. However, by Lemma 5.13, it also holds that $|v|_{\hat{t}}^{frac} = |v|_{\hat{t}}^{\widetilde{frac}}$. □

Let ϕ_{frac} and ϕ_{int} be the discrepancy at any time of the fractional and the integer token distribution algorithm respectively.

Corollary 5.15. *It holds that $\phi_{int} - \phi_{frac} \leq d$ given an arbitrary token adversary.*

Theorem 5.16. *Given an adversary $\mathcal{A}_{adv}(J, L, 1)$, in a d-dimensional hypercube it always holds that $\phi_{int} \leq 2J + 2L + d$, if the initial distribution is perfect (discrepancy 0).*

Proof. We show that the *fractional* discrepancy ϕ_{frac} is bounded by $2J+2L$. Together with Corollary 5.15, the claim follows. Let $J_t \leq J$ and $L_t \leq L$ be the insertions and deletions that happen at the beginning of step t. First, we consider the case of joins only, i.e., $L_t = 0$. Assume that all J_t tokens are inserted at node $v = b_0...b_i...b_{d-1}$ where $i := t \bmod d$. In the upcoming paragraph, all indices are implicitly modulo d. In step t, according to the token distribution algorithm, v keeps $J_t/2$ tokens and sends $J_t/2$ to node $u = b_0...\overline{b_i}...b_{d-1}$. In step $t+1$, $J_t/4$ are sent to nodes $b_0...b_i\overline{b_{i+1}}...b_{d-1}$ and $b_0...\overline{b_i b_{i+1}}...b_{d-1}$, and so on. Thus, after step $t+d-1$, every node in the d-dimensional hypercube has the same share of $\frac{J_t}{2^d}$ tokens from that insertion. We conclude that a node can have at most all insertions of this step, half of the insertions of the last step, a quarter of all insertions two steps ago and so on:

$$\underbrace{J_t + \frac{J_{t-1}}{2} + \frac{J_{t-2}}{4} + ... + \frac{J_{t-(d-1)}}{2^{d-1}}}_{< 2J} + \underbrace{\frac{J_{t-d}}{2^d} + \frac{J_{t-(d+1)}}{2^d} + \frac{J_{t-(d+2)}}{2^d} + ...}_{\text{shared by all nodes}}$$

Since $J_{t-i} \leq J$ for $i = 0, 1, 2, \ldots$, we have $\phi_{frac} \leq 2J$. For the case of only token deletions, the same argument can be applied, yielding a discrepancy of at most $2L$. Finally, if there are both insertions and deletions which do not cancel out each other, we have $\phi_{frac} \leq 2J + 2L$. □

5.2.3 Weighted Token Distribution

Assume a heterogenous system where some peers are more powerful than others and assume that a token represents a job of unit size. In such a scenario, instead of distributing the tokens uniformly among all nodes, it may make sense that powerful peers are assigned more tokens than weak peers. We call a setting where tokens have to be distributed with respect to such a criteria a *weighted token distribution problem*. In the following, only the static problem is studied.

More formally, let ω_i be the weight of a node $i \in V$. Furthermore, let λ_i^0 be the number of tokens the initial distribution assigns to i, λ_i' the final number of tokens at i, and λ_i the number of tokens at some moment of time during the execution of the weighted token distribution algorithm. We require that for every node $v \in V$ it holds that $\lambda_v' = \frac{\omega_v}{\sum_{i \in V} \omega_i} \cdot \sum_{i \in V} \lambda_i^0$.

For simplicity, we consider arbitrary divisible tokens only here (fractional case). Note that the equal distribution problem is a special instance of this problem for $\omega_i \equiv 1$. How can Algorithm 4 be adapted for this more general problem? One idea might be to balance two nodes v and u over a dimension as follows: v sends $\frac{\omega_u}{\omega_u + \omega_v} \cdot \lambda_v$ tokens to u and u sends $\frac{\omega_v}{\omega_u + \omega_v} \cdot \lambda_v$ to v. It can quickly be verified that this yields only a very bad approximation of the desired distribution.

The correct solution needs the aggregated weights of both sub-cubes which are balanced — a typical application for DASIS (see Chapter 4)! Let $\omega_{b_0...b_j}$ denote the sum of weights in the component consisting of all nodes with prefix $b_0...b_j$. Our weighted token distribution algorithm is shown in Algorithm 5 and runs in $\Theta(d)$ rounds.

Algorithm 5 Weighted Token Distribution

(* algorithm running on node $v := b_0...b_{d-1}$ *)
run DASIS ($\odot := +$, $initval := 0$, values $\omega_i \in \mathbb{R}$); (* $\Theta(d)$ rounds *)
for $j := 0$ to $d-1$ **do**
  SEND $\frac{\omega_{b_0...\overline{b_j}}}{\omega_{b_0...b_j}+\omega_{b_0...\overline{b_j}}} \cdot \lambda_v$ tokens TO node $b_0...\overline{b_j}...b_{d-1}$;
  RECV tokens and update λ_v;
end for

Theorem 5.17. *Given a d-dimensional hypercube H, Algorithm 5 assigns each node*

$$\lambda'_v = \frac{\omega_v}{\sum_{i\in V(H)}\omega_i} \cdot \sum_{i\in V(H)} \lambda_i^0$$

tokens.

Proof. Consider a node $v := b_0...b_{d-1}$. In the following, $H_{b_0...b_j}$ denotes the set of all nodes $v \in V(H)$ with prefix $b_0...b_j$. By induction after step j, there are

$$\frac{\sum_{i\in V(H_{b_0...b_j})}\omega_i}{\sum_{i\in V(H)}\omega_i} \sum_{i\in V(H)} \lambda_i^0$$

tokens in the sub-cube of nodes with prefix $b_0...b_j$. Hence, when $j = d-1$, node v alone has

$$\frac{\omega_v}{\sum_{i\in V(H)}\omega_i} \sum_{i\in V(H)} \lambda_i^0$$

tokens.

$j = -1$: For the empty prefix, the claim holds trivially.

$j \to j+1$: By our hypothesis, there are

$$\frac{\sum_{i\in V(H_{b_0...b_j})}\omega_i}{\sum_{i\in V(H)}\omega_i} \cdot \sum_{i\in V(H)} \lambda_i^0$$

tokens in the sub-cube with prefix $b_0...b_j$. In step $j+1$, every $u \in V(H)$ with prefix $b_0...\overline{b_{j+1}}$ sends

$$\frac{\omega_{b_0...b_{j+1}}}{\omega_{b_0...b_{j+1}}+\omega_{b_0...\overline{b_{j+1}}}}\lambda_u$$

tokens to a node with prefix $b_0...b_{j+1}$, and vice versa. Thus, summing over all nodes with prefix $b_0...b_{j+1}$:

$$\frac{\sum_{i\in V(H_{b_0...b_j})}\omega_i}{\sum_{i\in V(H)}\omega_i} \sum_{i\in V(H)} \lambda_i^0 \frac{\sum_{i\in V(H_{b_0...b_{j+1}})}\omega_i}{\sum_{i\in V(H_{b_0...b_{j+1}})}\omega_i + \sum_{i\in V(H_{b_0...\overline{b_{j+1}}})}\omega_i}$$

$$= \frac{\sum_{i\in V(H_{b_0...b_{j+1}})}\omega_i}{\sum_{i\in V(H)}\omega_i} \sum_{i\in V(H)} \lambda_i^0$$

. □

5.2.4 Related Work

The basic idea of our token distribution algorithm (Algorithm 4) has independently been invented by Cybenko [5] where it is called the "dimension-exchange method". Cybenko shows also that this algorithm yields a perfect discrepancy after d steps of quiescence in the fractional case. Moreover, Plaxton [9] has shown that this algorithm yields a discrepancy of d tokens in the worst case if the tokens are integer.

5.3 Simulated Hypercube Topology

After studying some issues concerning scalability, we present two different kinds of fault-tolerant simulated hypercubes in this chapter. The repairing algorithms run both in a time of quiescence and tolerate a constant number of joins and leaves per adversarial round.

5.3.1 Scalability

The nodes of a simulated hypercube are represented by several peers. In the following, we assume that the peers of the same node are completely connected (*intra-connections*), while the peers of two adjacent nodes are connected completely bipartite (*inter-connections*). Now consider such a simulated hypercube of a fixed dimension d, where an adversary sporadically inserts and removes arbitrary peers. Obviously, there are two problems to be addressed: First, if the adversary inserts all peers at the same node, the peer degree grows linearly with these insertions. The solution here is to distribute the peers among the nodes all the times, for example based on the ideas presented in Chapter 5.2, or based on a Hamiltonian cycle (see later in this chapter). However, even with a perfectly uniform distribution, the peer degree still depends linearly on the total number of peers in the system. Hence, the second problem to be addressed is the change of the dimension of the simulated hypercube: If more and more peers join the d-dimensional system, the nodes should expand to the next dimension, and vice versa if there are many leaves.

As a first approach, assume that a node $v = b_0...b_{d-1}$ whose number of peers reaches a certain threshold splits into two nodes $v := b_0...b_{d-1}0$ and $v' := b_0...b_{d-1}1$, where both new nodes get half of the peers. The problem of this approach is that different nodes expand at different times because they have a different amount of peers. We have to make sure that the hypercube does not degenerate, that is, we would like that the simulated topology always looks "similar to a hypercube". For example, we might postulate that a node can expand to dimension $d + 2$ only if all other nodes have been expanded to dimension $d + 1$, and vice versa for the shrinking hypercube. A simple idea to achieve this is to run the peer distribution algorithm presented in Chapter 5.2 with the difference that if a node v — which has already expanded — is balanced with a non-expanded neighbor u, it represents also v'. While this trick works fine if peers were fractional, see Example 5.1, it does not work in reality as the discrepancy is a function in d (see Chapter 5.2).

Example 5.1. *Figure 5.5 shows an example where only node 01 is not expanded to dimension 2. Balancing dimension 0, node 110 represents both the peers of itself and those of its expanded node 111, and for dimension 1 node 000 represents the peers at 000 and those at 001. Thus, in the final distribution, node 01 will have double the*

number of peers of all other nodes and will be the first to exceed a given threshold and expand, yielding a perfect 3-dimensional hypercube.

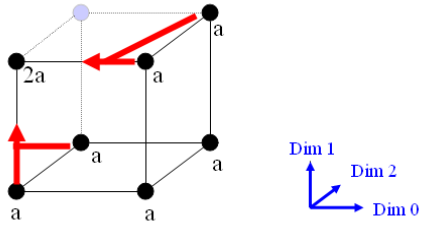

Figure 5.5: Figure for Example 5.1

In Chapter 5.3.2 we present a repairing algorithm which ensures that an expanded node will never expand again before all other nodes have been expanded as well, and vice versa for the reductions. This is achieved by distributing the peers with respect to a Hamiltonian cycle. An alternative approach is presented in Chapter 5.3.3: Based on the state aggregation algorithm presented in Chapter 4, the nodes will expand or reduce simultaneously, that is, in the same round, although they have not exactly the same number of peers.

5.3.2 Repairing in Time $\Theta(d)$ with a Hamiltonian Cycle

In order to prevent that an expanded node is expanded again before all other nodes have been expanded, or analogously, to prevent that a reduced node is reduced again before all other nodes have been reduced, a special form of peer distribution is used, based on a Hamiltonian cycle, see Definition 5.3.

Definition 5.3 (Hamiltonian Path, Hamiltonian Cycle). *A directed path passing through all vertices $v \in V$ of a graph is called* Hamiltonian path. *A* Hamiltonian cycle *is a cycle containing all vertices $v \in V$.*

While the decision problem whether a general graph has a Hamiltonian cycle is $\mathbb{NP}$-complete [4], there is always a Hamiltonian cycle on a hypercube. We show how such a cycle can efficiently be computed.

Theorem 5.18. *Algorithm 6 defines a correct Hamiltonian cycle on a d-dimensional hypercube.*

Proof. Except for the last step, Algorithm 6 always visits the neighbor in the highest possible dimension which is still unvisited. We show by induction that this strategy produces a Hamiltonian path. Let $u \rightsquigarrow v$ denote a path from a node u to a node v. It holds that, starting at node $b_0 b_1 ... b_{d-1}$, this strategy produces a Hamiltonian path $b_0 b_1 ... b_{d-1} \rightsquigarrow \overline{b_0} b_1 ... b_{d-1}$.

$d = 1$: The path has only one edge $(b_0, \overline{b_0})$, and the claim holds.

$d \to d+1$: Let H_0 be the d-dimensional sub-cube of all nodes $b_0\{0,1\}^d$ and H_1 be the d-dimensional sub-cube of all nodes $\overline{b_0}\{0,1\}^d$. By the induction hypothesis, our algorithm first visits all nodes of H_0 on the path $u := b_0 b_1 ... b_{d-1} b_d \rightsquigarrow v' := b_0 \overline{b_1} b_2 ... b_{d-1} b_d$. The highest dimensional neighbor of v' which is still unvisited is node $u' := \overline{b_0} b_1 b_2 ... b_{d-1} b_d$. From there, according to the induction hypothesis, our

Algorithm 6 Hamiltonian Cycle

```
1: (* start at node b_0...b_{d-1} *)
2: visited[2^d]:=empty;
3: for i :=1 to 2^d − 1 do
4:    for j :=0 to d − 1 do
5:       if ¬ visited[b_0...b̄_{d-1-j}...b_{d-1}] then
6:          visited[b_0...b_{d-1-j}...b_{d-1}]:=true;
7:          visit b_0...b̄_{d-1-j}...b_{d-1};
8:          break;
9:       end if
10:   end for
11: end for
12: visit b_0...b_{d-1};
```

algorithm produces the Hamiltonian path $\overline{b_0 b_1} b_2 ... b_{d-1} b_d \rightsquigarrow v := \overline{b_0} b_1 b_2 ... b_{d-1} b_d$ on H_1. The resulting path on all nodes $\{0,1\}^{d+1}$ of the $(d+1)$-dimensional hypercube is $u \rightsquigarrow v'(v', u')u' \rightsquigarrow v$. Obviously, this path is also Hamiltonian.

Hence, Algorithm 6 produces the correct Hamiltonian path $b_0 b_1 ... b_{d-1} \rightsquigarrow \overline{b_0} b_1 ... b_{d-1}$ and then returns to node $b_0 b_1 ... b_{d-1}$. Since $\{b_0 b_1 ... b_{d-1}, \overline{b_0} b_1 ... b_{d-1}\} \in E$, the claim holds. Figure 5.6 shows an example for $d = 4$. □

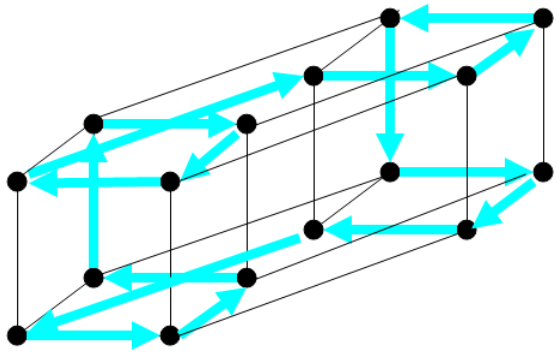

Figure 5.6: Hamilton Cycle on a 4-Dimensional Hypercube

Our repairing algorithm runs in a phase of quiescence of $\Theta(d)$ steps and works as follows: Assume that in a d-dimensional hypercube, every node is simulated by between C and $4C$ peers for some arbitrary constant C. If the number of peers at a node exceeds $4C$, the node sends the superfluous peers to a special node given by a distributed pointer HCP; on the other hand, if the number of peers falls beyond C, new peers are requested from the same special node, see Algorithm 7. Like this, we ensure that changes happen only at that special node. The idea is that the distributed pointer HCP always points to the last node that has been expanded. We will describe a repairing algorithm that tolerates at most $L := C - 1$ leaves and $J := C$ joins per adversarial round.

First consider the case where in the adversarial round, there have only been at most $C - 1$ leaves and no joins. The HCP node receives all superfluous peers after $d + 1$

Algorithm 7 Scalable Hypercube in Repairing Time $\Theta(d)$

1: **if** number of peers $= (C - x)$ where $x > 0$ **then**
2: SEND "requesting x peers" TO HCP;
3: **end if**
4: **if** number of peers $= (4C + x)$ where $x > 0$ **then**
5: SEND x peers TO HCP;
6: **end if**

steps — not d, as some nodes are already expanded. We know that before the adversary round, HCP had an extension node with at least C peers. Therefore, the HCP node can satisfy all L leaves, but maybe has to reduce its extension node and send HCP back to its predecessor, that is, the last node that has been expanded. On the other hand, if there are only C joins and no leaves, the HCP node checks whether its extension node or itself has still enough space for the received peers. If not, the HCP forwards all peers to its successor node on the Hamiltonian cycle defined on the d-dimensional hypercube. If this successor node has to expand to accommodate the new peers, it becomes the new HCP. The case where leaves *and* joins happen is straight-forward: The HCP node uses overflow peers to satisfy the leaves and the remaining leaves if any are handled as described above. At the end of each phase of quiescence, all nodes are informed about the new position of the distributed pointer HCP by a broadcast.

Thus, the nodes expand well ordered with respect to a Hamiltonian cycle. When the expansion is complete for a dimension d at node v, the HCP returns to the starting point, where the expansion for dimension $d + 1$ starts. Figure 5.7 shows the growth of the hypercube and the position of the HCP pointer. We conclude that this algorithm

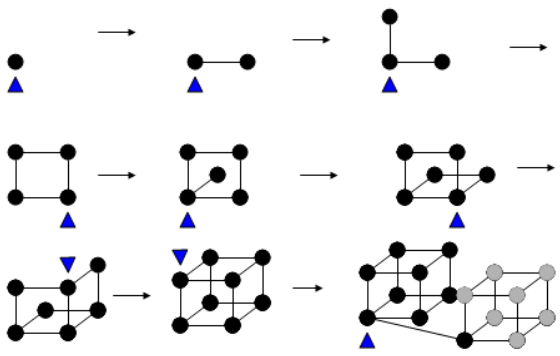

Figure 5.7: Growth of Hypercube and HCP Position

works in a phase of quiescence of $\Theta(log\ N)$ steps, where N is the current number of peers in the system. The presented algorithm is a very simple version and could be optimized in several respects.

5.3.3 Repairing in Time $O(1)$ with DASIS

The simulated hypercube system presented in the previous section has several drawbacks, for example it is maintained by a repairing algorithm which needs a logarithmic period (in the total number of peers in the system) of quiescence. We now introduce a

repairing algorithm which runs in a constant number of rounds. Again, the hypercube's nodes are simulated by a clique of peers, and peers of adjacent nodes are connected completely bipartite. The repairing algorithm makes use of two components which have already been presented:

- To distribute the peers evenly across the network nodes, we use the integer, concurrent version of our token distribution algorithm (Algorithm 4).
- The information aggregation system DASIS, see Chapter 4, is used to estimate the total number of peers in the system. It allows all nodes to expand or reduce at the same time. Like this, the topology is always a perfect hypercube.

First, note that for the special case of hypercubes, DASIS has another important property.

Theorem 5.19. *For the special case of a d-dimensional hypercube, at time t, DASIS provides each node with an aggregated value which describes the correct state of the system at time $t - d$.*

Proof. As has been pointed out already, DASIS guarantees that two nodes with the same prefix $b_0...b_i$ always store the same value for this prefix. Since all nodes in a hypercube have the same number of bits, the information is propagated on a perfect binary tree, that is, the prefix $b_0...b_i$ stores the aggregated state of the nodes $b_0...b_i\{0,1\}^{d-1-i}$ at time $t-(d-1-i)$. □

The basic idea of our repairing algorithm is simple: In the phase of quiescence, each node exchanges the new estimations of the total number of peers in the system with its prefix buddies (one round of DASIS). Moreover, it balances the peers with its adjacent node in dimension i according to our token distribution algorithm (Algorithm 4), where i is incremented modulo d for every execution of the repairing algorithm. If the estimated average number of peers per node μ — the estimated total number of peers in the system divided by 2^d — falls beyond a given threshold LT, the hypercube shrinks ($d := d-1$), and analogously, if the average exceeds a given threshold UT, the hypercube grows ($d := d+1$). The repairing algorithm is summarized in Algorithm 8. Note that these steps can indeed be executed in a constant number of rounds. In the following, we will refer to a complete execution of the repairing algorithm as a *phase*.

Algorithm 8 Repairing Algorithm in Time $O(1)$

```
1: (* node b_0...b_{d-1} *)
2: DASIS round: exchange estimated total number of peers with prefix buddies;
3: update estimation vector prefix_agg[] accordingly;
4: balance peers across dimension i;
5: i := i + 1 mod d;
6: if (prefix_agg[ε]/2^d) < LT then
7:    merge nodes b_0...b_{d-1} and b_0...b̄_{d-1} to b_0...b_{d-2};
8:    d := d − 1;
9: end if
10: if (prefix_agg[ε]/2^d) > UT then
11:    split node b_0...b_{d-1} to b_0...b_{d-1}0 and b_0...b_{d-1}1;
12:    d := d + 1;
13: end if
```

Consider the thresholds $LT := 9C + d$ and $UT := 36C + 2d$ for some arbitrary constant $C > 0$. We claim that, given an adversary $\mathcal{A}_{adv}(C, C)$ which may insert at most C and remove at most C peers per adversarial round,

1. a node is represented by more than C peers at any moment of time,
2. no node will ever have more than $40C + 3d$ peers,
3. the network diameter is bounded by $O(\log N)$, where N is the total number of peers in the system, and
4. the peer degree is bounded by $O(\log^2 N)$.

Note that the criteria 1 and 2 imply the criteria 3 and 4.

Theorem 5.20. *Let c_1, c_2, c_3 and c_4 be constants greater than zero. In a simulated hypercube with between $c_1C + c_2d$ and $c_3C + c_4d$ peers per node, where peers within a node form a clique (complete* intra-connections*), and where peers between adjacent nodes are connected completely bipartite (*inter-connections*), the network diameter is in $\Theta(log\ N)$ and the peer degree is bounded by $\Theta(log^2\ N)$, where N is the total number of peers in the system.*

Proof. There are 2^d nodes with $\Theta(d)$ peers each, yielding $\Theta(d2^d)$ peers in total. Note that $2^d < d2^d < 2^{2d}$ for $d > 1$, and hence $d = \Theta(log\ N)$. The network diameter is d, so the first claim holds. Moreover, note that a peer has $\Theta(d)$ intra-node connections and $d \cdot \Theta(d)$ inter-node connections, yielding a peer degree of $\Theta(log^2\ N)$. □

It remains to prove the first two criteria.

Theorem 5.21. *A node will always have between $C+1$ and $40C+3d$ peers. Moreover, it holds that after a dimension change from d_{old} to d_{new}, $\mu \in [9C+d_{new}, 36C+2d_{new}]$ for at least $d_{new} + 1$ phases.*

Proof. We consider the cases where the average number of peers per node μ falls beyond the lower threshold $9C+d_{old}$ or exceeds the upper threshold $36C+2d_{old}$ in turn. Note that such an event will lead to a dimension change with a delay of d_{old} phases only, see Theorem 5.19. We prove that after the change, $\mu \in [9C+d_{new}, 36C+2d_{new}]$ for at least $d_{new} + 1$ phases, so the dimension remains stable for at least $2d_{new} + 1$ phases. Moreover, this implies — together with Theorem 5.16 — that the discrepancy before the next change is limited by $2J+2L+d_{new} = 2C+2C+d_{new} = 4C+d_{new}$. We consider phase t, where the nodes learn that $\mu \notin [9C + d_{old}, 36C + 2d_{old}]$.

Case $\mu < 9C + d_{old}$: At time $t - d_{old}$, it held that $\mu < 9C + d_{old}$ while at time $t - d_{old} - 1$ we had $\mu \geq 9C + d_{old}$. In $d_{old} + 1$ phases, there are at most $(d_{old} + 1)C$ leaves, so $\mu \geq 9C+d_{old}-\frac{(d_{old}+1)C}{2^{d_{old}}} \geq 8C+d_{old}$ before merging. Clearly, there must have been a node with at least $8C + d_{old}$ peers, so, given the discrepancy of $4C + d_{old}$ (see Theorem 5.16), every node has at least $4C$ peers before merging.

What about the maximum? At time $t - d_{old}$, $\mu < 9C + d_{old}$, and there have been at most $d_{old}C$ joins in d_{old} steps, so $\mu < 9C + d_{old} + \frac{d_{old}C}{2^{d_{old}}} < 10C + d_{old}$ before merging, and $\mu < 20C + 2d_{old}$ afterwards. The maximum node has less than $24C + 3d_{old} = 24C + 3d_{new} + 3$ peers.

Next, we show that $\mu \geq 9C + d_{new}$ for the next $d_{new} + 1$ phases after a reduction. At time $t - d_{old} - 1$, $\mu \geq 9C + d_{old} = 9C + d_{new} + 1$. The reduction doubles the average number of peers per node, so $\mu \geq 18C + 2d_{new} + 2$. Further, there

are at most $(d_{old}+1+d_{new}+1)C = (2d_{new}+3)C$ leaves in the meantime, so $\mu \geq 18C + 2d_{new} + 2 - \frac{(2d_{new}+3)C}{2^{d_{new}}} \geq 18C + 2d_{new} - 1$.

Finally, $\mu \leq 36C+2d_{new}$ for $d_{new}+1$ phases. At time $t-d_{old}$, $\mu < 9C+d_{new}+1$, so $\mu < 18C + 2d_{new} + 2$ after the reduction. There are at most $(d_{old}+d_{new}+1)C = (2d_{new}+2)C$ joins, so $\mu < 18C + 2d_{new} + 2 + \frac{(2d_{new}+2)C}{2^{d_{new}}} \leq 18C + 2d_{new} + 4$.

Case $\mu > 36C + 2d_{old}$: At time $t - d_{old}$, $\mu > 36C + 2d_{old} = 36C + 2d_{new} - 2$, so $\mu > 18C + d_{new} - 1$ after splitting; there are at most $d_{old}C = (d_{new}-1)C$ leaves in d_{old} steps, so $\mu > 18C + d_{new} - 1 - \frac{(d_{new}-1)C}{2^{d_{new}}} > 18C + d_{new} - 2$. According to Theorem 5.16, the minimum node has more than $14C - 2$ peers after splitting. At time $t - d_{old} - 1$, $\mu \leq 36C + 2d_{old}$, and there are at most $(d_{old}+1)C$ joins. So before splitting, $\mu \leq 36C + 2d_{old} + \frac{(d_{old}+1)C}{2^{d_{old}}} \leq 36C + 2d_{old} + 1$, and the maximum node has at most $40C + 3d_{old} + 1$ peers.

Next, we show that $\mu \geq 9C+d_{new}$ for the next $d_{new}+1$ phases after the expansion. At time $t - d_{old}$, $\mu > 36C + 2d_{old} = 36C + 2d_{new} - 2$, so $\mu > 18C + d_{new} - 1$ after the expansion. Moreover, there are at most $(d_{old}+d_{new}+1)C = 2d_{new}C$ leaves, and $\mu > 18C + d_{new} - 1 - \frac{2d_{new}C}{2^{d_{new}}} \geq 17C + d_{new} - 1$. Finally, $\mu \leq 36C + 2d_{new}$ for the next $d_{new}+1$ steps: At time $t - d_{old} - 1$, $\mu \leq 36C + 2d_{old} = 36C + 2d_{new} - 2$, so $\mu \leq 18C + d_{new} - 1$ after the expansion; moreover, there are at most $(d_{old} + 1 + d_{new} + 1)C = (2d_{new}+1)C$ joins, so $\mu \leq 18C + d_{new} - 1 + \frac{(2d_{new}+1)C}{2^{d_{new}}} < 20C + d_{new} - 1$. □

5.3.4 Worst-Case Fault-Tolerance vs. Scalability

The simulated hypercube topologies presented so far tolerate a constant number of leaves, no matter how many peers the system actually contains. Of course, in a real network, peers leave independently, and it would be nice if twice as many leaves are tolerable as the number of peers in the system is doubled. Of course, this goal stands in direct tension to the scalability goal.

Theorem 5.22. *In the worst case, a graph G can tolerate at most*

$$min_{v \in V(G)} deg_G(v) - 1$$

leaves where $deg_G(v)$ denotes the degree of a node $v \in V(G)$.

Proof. Consider a node u with $deg_G(u) = min_{v \in V(G)} deg_G(v)$. If $min_{v \in V(G)} deg_G(v)$ leaves happen, node u may lose all its neighbors in the worst case and hence gets disconnected from the network. □

Thus, in the best case the fault-tolerance correlates linearly with the minimum degree of any node in the graph.

5.4 DHT and Concurrent Adversary

5.4.1 Distributed Hash Table

A fundamental problem that confronts p2p systems is to efficiently locate the peer that stores a particular data item. This operation is related to hashing and is therefore sometimes also known as distributed hashing in conjunction with distributed hash tables (DHTs). While so far we have concentrated on the repairing of the simulated hypercube topology, we must now also ensure that no data is lost during the churn.

There are several possibilities how data items can be stored on a simulated d-dimensional hypercube. Here, we assume that a data item is redundantly stored by the peers of the node to which the data item hashes. That is, a data item (for example a song) with a identifier id (for example the title of the song) is stored at the node whose identifier matches the first d bits of $hash(\text{id})$, where $hash(\cdot)$ is a hash function mapping the identifier space to $\{0,1\}^\infty$. Of course, for all what matters in practice, a hash function like SHA[1] having 160-bit output is sufficient. Note that this solution has the disadvantage that the data items and the nodes are glued together. That is, in the worst-case, some nodes get much more data items than others, and a load balancing which would equalize these differences is not possible.

We have not specified yet on which peers of the node a data item is replicated. As a rule, of course, we'd like to store copies at as little peers as necessary, while always guaranteeing that the adversary can not remove all replicas of a given data item.

5.4.2 Concurrent Adversary

We will show that our distributed hash table based on the simulated hypercube is resilient to *concurrent* adversarial churn, which is more realistic than our simple dynamic model with phases of quiescence. This has major implications on the *maintenance algorithm*, as has already been pointed out in the chapter about k-rings. We assume here that the adversary always acts at the beginning of a round. Again, the changes that an adversary can do are specified with respect to *time intervals*.

5.4.3 Resilience to Logarithmic Churn

We give now a complete description of the new simulated hypercube system, which tolerates $J := d+1$ joins and $L := d+1$ leaves in any time period of 6 rounds, hence — according to the formalism presented in Chapter 2 — an adversary of type $\mathcal{A}_{adv}(d+1, d+1, 6)$. Note that this implies that the fault-tolerance is no longer constant, but grows logarithmical with the total number of peers in the system.

One idea would be to store the data items on *all* peers of the data item's node. However, we will show that this is not necessary to tolerate $\mathcal{A}_{adv}(d+1, d+1, 6)$. Moreover, this approach would have the disadvantage that each time a peer has to change the node during the peer distribution algorithm, it has to delete all data items of the old node an insert all data items of the new node.

This motivates the division of the peers of a node v into two categories: a *core* $\mathcal{C}_v$ of at most $2d+3$ peers and a *periphery* $\mathcal{P}_v$ consisting of the remaining peers. The data items of node v will only be stored by the core peers, while the peripheral peers are used for the peer distribution algorithm.

In order to save some links, we assume that all peers within the same node are sill completely connected (*intra-connections*). Additionally, every peer is connected to all *core* peers of the neighboring nodes (*inter-connections*). Figure 5.8 shows an example of this new simulated hypercube topology for $d = 2$.

In the next section we will present a maintenance algorithm which maintains this simulated hypercube topology. In particular, it guarantees that (1) there is always at least one core peer per node, hence no data items will ever get lost, and that (2) each node has between $3d+10$ and $45d+86$ peers at every moment of time. Note that, by a

[1] "Secure Hash Algorithm", National Institute of Standards and Technology, NIST FIPS PUB 186, U.S. Department of Commerce, 1994.

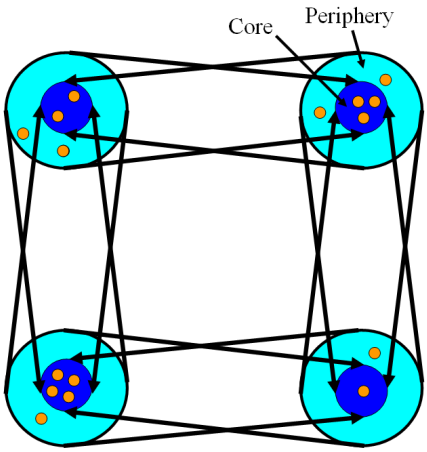

Figure 5.8: 2-Dimensional Simulated Hypercube

similar analysis as in the proof of Theorem 5.20 this implies that the network diameter is bounded by $\Theta(\log N)$, where N is the total number of peers in the system, and the peer's out-degree is in $\Theta(\log^2 N)$.

5.4.4 The 6-Round Maintenance Algorithm

The *6-round (maintenance) algorithm* maintains the simulated hypercube topology described in the previous section given an adversary $\mathcal{A}_{adv}(d+1, d+1, 6)$. In particular, it ensures that

1. every node has at least one core peer all the times and hence no data is lost;
2. each node has between $3d + 10$ and $45d + 86$ peers at every moment of time;
3. only peripheral peers are moved between nodes, thus the unnecessary copying of data is avoided.

In the following, we will again refer to a complete execution of all six rounds (ROUND 1 - ROUND 6) of the maintenance algorithm as a *phase*. Basically, the 6-round algorithm works similar to the repairing algorithm described in Chapter 5.3.3: It balances the peers across one dimension in every phase and estimates the total number of peers in the system with DASIS. If the average number of peers per node μ falls beyond the lower threshold $LT := 8d + 16$, the hypercube shrinks, and if it exceeds the upper threshold $UT := 40d + 80$, it grows. After a detailed description of the six rounds, we prove that the criteria presented above are indeed fulfilled.

In ROUND 1, every peer within each node v sends its ID plus the IDs of its joiners of the last phase to all former adjacent peers within v, such that each peer within v learns the set of the currently active peers in v. ROUND 2 to ROUND 6 are then based only on the ID set of this snap-shot, which may not represent the current state of the system anymore. In our description, the following convention is used: By "$\mathcal{C}_v$ send a packet to $\mathcal{P}_v$" we mean that every peer π_1 in core $\mathcal{C}_v$ *which is still alive* sends that packet to each *surviving* peer π_2 in the periphery $\mathcal{P}_v$, and analogously for other formulations involving an ID set.

We describe the i^{th} phase (mod d) and assume that $d > 0$. The case where $d = 0$ is straight-forward and not explained further here. The following notation is used: If a hypercube grows to dimension $d+1$, $\underline{v}$ denotes a node in the lower d-dimensional sub-cube and $\overline{v}$ denotes $\underline{v}$'s adjacent node in the high sub-cube. Analogously, if a hypercube shrinks to dimension $d-1$, $\overline{v}$ denotes a node in the high $(d-1)$-dimensional sub-cube that will be removed and $\underline{v}$ is $\overline{v}$'s adjacent across dimension $d-1$.

ROUND 1

Outline: Each node v makes a snapshot of the currently active peers, denoted by the ID set $\mathcal{S}_v$. The later rounds will only be based on these sets.

Sent Messages: Each peer of a node v sends a packet with its own ID and the (potentially empty) ID set of its joiners to all adjacent peers *within* v.

ROUND 2

Outline: Based on the snapshot of ROUND 1, the core peers of a node v know the total number of peers in the node, $size(v) := |\mathcal{S}_v|$. This information is needed for the peer distribution algorithm and for the estimation of the total number of peers in the system.

Local Computations: The core peers compute $size(v) := |\mathcal{S}_v|$.

Sent Messages: Each peer informs its joiners about $\mathcal{S}_v$. The core peers $\mathcal{C}_v$ additionally send the number $size(v)$ to their neighboring core $\mathcal{C}_u$, where node u is v's neighbor in dimension i — the node with which v has to balance its peers in this phase. The core also exchanges the new estimated total number of peers in its domains with the corresponding adjacent cores.

ROUND 3

Outline: At the beginning of this round, every peer within a node v knows $\mathcal{S}_v$, and the transfer for the peer distribution algorithm can be prepared. Let v again be an arbitrary node and u its adjacent node in dimension i. We assume that $size(v) > size(u)$; the case where $size(v) \leq size(u)$ is analogous and not described further here. The ID set $\mathcal{T}$ of peers that have to move from node v to node u are the $\frac{size(v)-size(u)}{2}$ (arbitrarily rounded) peers in the periphery $\mathcal{P}_v$ having the smallest identifiers.

Local Computations: The peers in each node v compute the new periphery $\mathcal{P}_v := \mathcal{S}_v \setminus \mathcal{C}_v$. The core remains the same.

Sent Messages: All cores forward the information about the new estimated total number of peers in the system to their peripheral peers. Moreover, the core of the larger node $\mathcal{C}_v$ sends the identifiers of the to be transferred peers $\mathcal{T}$ to $\mathcal{C}_u$, and the number $\frac{size(v)-size(u)}{2}$ to the new periphery $\mathcal{P}_v$.

ROUND 4

Outline: The transfer for the peer distribution algorithm is continued. Moreover, this round prepares the dimension reduction if necessary.

Sent Messages: The core $\mathcal{C}_u$ informs the peers in $\mathcal{T}$ about all neighboring cores $\mathcal{C}_{u_j}$, where u_j is the neighbor of u in dimension j for $j \in [0, d-1]$, about $\mathcal{C}_u$ itself, about $\mathcal{S}_u$ and about its peripheral peers $\mathcal{P}_u$. Moreover, $\mathcal{C}_u$ informs its own periphery $\mathcal{P}_u$ about the newcomers $\mathcal{T}$.

If the estimated total number of peers in the system is beyond the threshold, the core peers of a node $\overline{v}$ which will be reduced send their data items plus the identifiers of all their peripheral peers (with respect to the situation *after* the transfer) to the core of their adjacent node $\underline{v}$.

ROUND 5

Outline: This round finishes the peer distribution, establishes the new peripheries, and prepares the building of a new core. If the hypercube has to grow in this phase, the nodes start to split, and vice versa if the hypercube is going to shrink.

Local Computations: Given the number $\frac{size(v)-size(u)}{2}$, the peripheral peers $\mathcal{P}_v$ can compute the set $\mathcal{T}$ selecting the $\frac{size(v)-size(u)}{2}$ smallest elements in $\mathcal{P}_v$. From this, the new periphery $\mathcal{P}_v := \mathcal{P}_v \setminus \mathcal{T}$ is computed. Analogously, the peers in node u (including $\mathcal{T}$) can compute the new periphery $\mathcal{P}_u := \mathcal{P}_u \cup \mathcal{T}$.

Then, all peers of each node v calculate the new core $\mathcal{C}_v^{new}$: It consists of the peers of the old core which have still been alive in ROUND 1, i.e., $\mathcal{C}_v^{old} := \mathcal{C}_v \cap \mathcal{S}_v$, plus the $2d + 3 - |\mathcal{C}_v \cap \mathcal{S}_v|$ smallest IDs in the new periphery $\mathcal{P}_v$, denoted by $\mathcal{C}_v^{\triangle}$. Hence, the new core is given by $\mathcal{C}_v^{new} := \mathcal{C}_v^{old} \cup \mathcal{C}_v^{\triangle}$, and the new periphery by $\mathcal{P}_v^{new} := \mathcal{P}_v \setminus \mathcal{C}_v^{\triangle}$.

If the hypercube has to grow in this phase, the smallest $2d + 3$ peers in the new periphery $\mathcal{P}_{\underline{v}}^{new}$ become the new core of the expanded node, $\mathcal{C}_{\overline{v}}$. Half of the remaining peripheral peers, the ones with the smaller identifiers, build the new periphery $\mathcal{P}_{\overline{v}}$, and the other half becomes $\mathcal{P}_{\underline{v}}$. All these operations can be computed locally by every peer.

Sent Messages: The old core $\mathcal{C}_v^{old}$ informs all its neighboring nodes (i.e., their old cores) about the new core $\mathcal{C}_v^{new}$. Moreover, $\mathcal{C}_v^{old}$ sends its data items to the peers in $\mathcal{C}_v^{\triangle}$.

If the hypercube is about to grow, $\mathcal{C}_{\underline{v}}^{old}$ sends the necessary data items to the core peers of the new node, $\mathcal{C}_{\overline{v}}$. Moreover, $\mathcal{C}_{\underline{v}}^{old}$ informs its neighboring (old) cores about the IDs of its expanded core $\mathcal{C}_{\overline{v}}$.

If the hypercube is about to shrink, all cores $\mathcal{C}_{\underline{v}}^{old}$ inform their periphery about the peers arriving from the expanded node and the peers in the expanded node about the new core $\mathcal{C}_{\underline{v}}^{new}$ and its periphery. $\mathcal{C}_{\underline{v}}^{old}$ copies also the data items of $\mathcal{C}_{\overline{v}}^{old}$ to the peers $\mathcal{C}_{\underline{v}}^{\triangle}$.

ROUND 6

Outline: Building the new cores and accomplishing the dimension change if necessary.

Local Computations: If the hypercube has been reduced, every peer can now compute the new periphery $\mathcal{P}_{\underline{v}}$.

Sent Messages: The old core $\mathcal{C}_v^{old}$ forwards the information about the new neighboring cores to the peers $\mathcal{C}_v^{\Delta} \cup \mathcal{P}_v$.

If the hypercube has grown, $\mathcal{C}_{\underline{v}}^{old}$ forwards the expanded cores of its neighboring nodes to *all* peers in its expanded node $\overline{v}$. Note that his requires that $\mathcal{C}_{\underline{v}}^{old}$ remembers the peripheral peers that have been transferred to $\overline{v}$ in ROUND 5.

5.4.5 Analysis

We show that, given an adversary $\mathcal{A}_{adv}(d+1, d+1, 6)$ which inserts and removes at most $d+1$ peers per phase, the 6-round algorithm indeed guarantees at least one core peer per node at every moment of time, and that no core peer ever has to change the node for the peer balancing. Moreover, we prove that every node has always between $3d+10$ and $45d+86$ peers if $d > 0$, which implies a logarithmic network diameter.

First, consider a much simpler system without any notion of core and periphery, where the maintenance algorithm simply runs the peer distribution algorithm and the information aggregation algorithm to count the total number of peers in the system, and expands or reduces the hypercube with respect to the thresholds $LT = 8d + 16$ and $UT = 40d + 80$ presented above. Moreover, assume that these operations are performed in quiet phases, where the adversary may remove at most $d+1$ and add at most $d+1$ peers only in-between.

Lemma 5.23. *For this simpler system, it holds that every node in the simulated d-dimensional hypercube has at least $3d + 10$ and at most $45d + 86$ peers at every moment of time. Moreover, after the hypercube has changed its dimension from d_{old} to d_{new}, the dimension will remain stable for at least $2d_{new} + 1$ phases.*

Proof. We consider the cases where the average number of peers per node μ falls beyond the lower threshold $8d_{old}+16$ or exceeds the upper threshold $40d_{old}+80$ in turn. Note that such an event will lead to a dimension change with a delay of d_{old} phases only, see Theorem 5.19. We prove that after the change, $\mu \in [8d_{new}+16, 40d_{new}+80]$ for at least $d_{new}+1$ phases. The dimension remains stable for at least $2d_{new}+1$ phases which implies — together with Theorem 5.16 — that the discrepancy before the next change is limited by $2(d_{new}+1) + 2(d_{new}+1) + d_{new} = 5d_{new} + 4$.

Case $\mu < 8d + 16$: At time $t - d_{old}$, it held that $\mu < 8d_{old} + 16$ while at time $t - d_{old} - 1$ we had $\mu \geq 8d_{old} + 16$. In $d_{old}+1$ phases, there are at most $(d_{old}+1)(d_{old}+1) = d_{old}^2 + 2d_{old} + 1$ leaves, so $\mu \geq 8d_{old} + 16 - \frac{d_{old}^2+2d_{old}+1}{2^{d_{old}}} > 8d_{old} + 14$ before merging. Clearly, there must be a node with more than $8d_{old} + 14$ peers, hence, given the discrepancy of $5d_{old} + 4$ (see Theorem 5.16), every node has more than $3d_{old} + 10$ peers before merging.

What about the maximum? At time $t - d_{old}$, $\mu < 8d_{old} + 16$, and there have been at most $d_{old}(d_{old}+1)$ joins in d_{old} steps, so $\mu < 8d_{old} + 16 + \frac{d_{old}(d_{old}+1)}{2^{d_{old}}} < 8d_{old} + 18$ before merging, and $\mu < 16d_{old} + 36$ afterwards. The maximum node has less than $21d_{new} + 61$ peers.

Next, we show that $\mu \geq 8d_{new} + 16$ for the next $d_{new} + 1$ phases after a reduction. At time $t - d_{old} - 1$, $\mu \geq 8d_{old} + 16 = 8d_{new} + 24$. The reduction doubles the average number of peers per node, so $\mu \geq 16d_{new} + 48$. Further, there are at most

$(d_{old}+1)(d_{old}+1)+(d_{new}+1)(d_{new}+1) = 2d_{new}^2+6d_{new}+5$ leaves in the meantime, so $\mu \geq 16d_{new}+48-\frac{2d_{new}^2+6d_{new}+5}{2^{d_{new}}} > 16d_{new}+41 > 8d_{new}+16$.

Finally, $\mu \leq 40d_{new}+80$ for $d_{new}+1$ phases. At time $t-d_{old}$, $\mu < 8d_{new}+24$, so $\mu < 16d_{new}+48$ after the reduction. There are at most $d_{old}(d_{old}+1)+(d_{new}+1)(d_{new}+1) = 2d_{new}^2+5d_{new}+3$ joins, so $\mu < 16d_{new}+48+\frac{2d_{new}^2+5d_{new}+3}{2^{d_{new}}} < 16d_{new}+54 < 40d_{new}+80$.

Case $\mu > 40d+80$: At time $t-d_{old}$, $\mu > 40d_{old}+80 = 40d_{new}+40$, so $\mu > 20d_{new}+20$ after splitting; there are at most $d_{old}(d_{old}+1) = d_{new}^2-d_{new}$ leaves in d_{old} steps, so $\mu > 20d_{new}+20-\frac{d_{new}^2-d_{new}}{2^{d_{new}}} > 20d_{new}+19$. According to Theorem 5.16, the minimum node has more than $15d_{new}+15$ peers after splitting. At time $t-d_{old}-1$, $\mu \leq 40d_{old}+80$, and there are at most $(d_{old}+1)(d_{old}+1) = d_{old}^2+2d_{old}+1$ joins. So before splitting, $\mu \leq 40d_{old}+80+\frac{d_{old}^2+2d_{old}+1}{2^{d_{old}}} < 40d_{old}+82$, and the maximum node has at most $45d_{old}+86$ peers.

Next, we show that $\mu \geq 8d_{new}+16$ for the next $d_{new}+1$ phases after the expansion. At time $t-d_{old}$, $\mu > 40d_{old}+80 = 40d_{new}+40$, so $\mu > 20d_{new}+20$ after the expansion. Moreover, there are at most $d_{old}(d_{old}+1)+(d_{new}+1)(d_{new}+1) = 2d_{new}^2+d_{new}+1$ leaves, and $\mu > 20d_{new}+20-\frac{2d_{new}^2+d_{new}+1}{2^{d_{new}}} > 20d_{new}+17 \geq 8d_{new}+16$. Finally, $\mu \leq 40d_{new}+80$ for the next $d_{new}+1$ steps: At time $t-d_{old}-1$, $\mu \leq 40d_{old}+80 = 40d_{new}+40$, so $\mu \leq 20d_{new}+20$ after the expansion; moreover, there are at most $(d_{old}+1)(d_{old}+1)+(d_{new}+1)(d_{new}+1) = 2d_{new}^2+2d_{new}+1$ joins, so $\mu \leq 20d_{new}+20+\frac{2d_{new}^2+2d_{new}+1}{2^{d_{new}}} < 20d_{new}+24 < 40d_{new}+80$. □

In our real system, repairing takes six rounds and runs *concurrently* to the adversary. However, as all operations in the whole phase are based upon the state of ROUND 1, a phase can be considered as running uninterruptedly, that is, as if the adversary inserted $d+1$ and removed $d+1$ peers only *between* the phases. Thus, Lemma 5.23 also holds in our system. However, we additionally have to postulate that there is always at least one *core peer*. By Lemma 5.23, it is always possible to select $2d+3$ core peers in ROUND 5 with respect to the state of ROUND 1. These peers have to survive until ROUND 6 of the next phase, so for twelve normal rounds in total; however, as the adversary $\mathcal{A}_{adv}(d+1,d+1,6)$ may remove at most $2d+2$ peers in twelve rounds, this clearly holds.

Finally, we show that there are indeed enough peripheral peers in ROUND 3 such that core peers do not have to change the node for the peer distribution.

Lemma 5.24. *In* ROUND *3, it holds that* $|\mathcal{P}_v| > \frac{size(v)-size(u)}{2}$.

Proof. By Lemma 5.23, we know that $size(v) \geq 3d+10$ and $size(u) \geq 3d+10$. As v has at most $2d+3$ core peers, we have $|\mathcal{P}_v| \geq size(v)-(2d+3) \geq size(v)-size(u) > \frac{size(v)-size(u)}{2}$. □

Theorem 5.25. *Given an adversary $\mathcal{A}_{adv}(d+1,d+1,6)$ which inserts and removes at most $d+1$ peers per phase, the 6-round algorithm ensures that (1) every node has at least one core peer all the times and hence no data is lost; (2) each node has between $3d+10$ and $45d+86$ peers at every moment of time, yielding a logarithmic network diameter; (3) only peripheral peers are moved between nodes, thus the unnecessary copying of data is avoided.*

Proof. The three criteria follow directly from Lemmata 5.23 and 5.24. □

Chapter 6

Skip Graph

6.1 Introduction

Skip graphs are a novel randomized distributed data structure based on skip lists [11] and have independently been proposed in [3] and [7]. There exists also a deterministic variant of a skip graph [6].

In this chapter, we present a p2p system based on a *perfect* skip graph as defined in Definition 6.1. We will use again our simulation approach, where a skip graph's node is represented by a group of peers. We will show that many components we used for the repairing algorithm of the simulated hypercube can be extended to the skip graph topology.

Definition 6.1 (Perfect Skip Graph). *A* perfect skip graph *is a network* $G = (V, E)$*, where* $V = \{0, ..., 2^h - 1\}$ *for some* $h \in \mathbb{N}$*, and* $E = \{\{u, v\} \mid u \equiv v + 2^i \ (mod\ 2^h)$ *for* $i = 0...h - 1\}$*. A skip graph consists of* h *levels and we call the rings belonging to level* i *the* i-rings*; moreover we call* u*'s neighbors* $v + 2^i$ *and* $v - 2^i$ *on level* i v*'s* i-neighbors*. Figure 6.1 gives an example for* $h = 4$ *(cyclic edges not shown).*

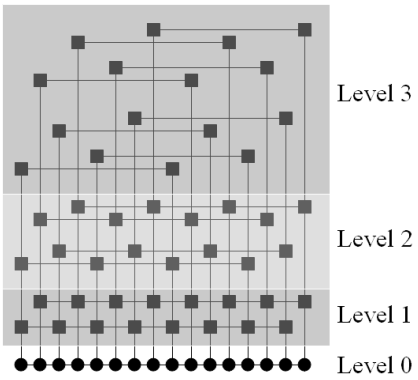

Figure 6.1: Skip Graph for $h = 4$

The rest of this chapter is organized as follows: In Chapter 6.2 we show that it is possible to adapt both the peer distribution algorithm *and* the algorithm to estimate

the total number of peers in the system for the perfect skip graph. Moreover, we will sketch an algorithm to change "the dimension" of a skip graph, that is, the operations to be performed when all nodes expand ($h := h + 1$) or reduce ($h := h - 1$). With this components, it is obviously possible to achieve a p2p system that is robust to the same adversarial changes as the simulated hypercube presented in Chapter 5.4. Finally, in Chapter 6.3 we will address some issues concerning the data items that may be stored by a simulated skip graph.

6.2 Simulated Perfect Skip Graph

6.2.1 Token Distribution

Algorithm 9 is the natural extension of the token distribution algorithm presented for hypercubes (see Algorithm 4). Because of the similarity of the two algorithms, all results hold also here. Figure 6.2 shows an example for $h = 3$.

Algorithm 9 Token Distribution on Perfect Skip Graph

1: $my\_id := u$;
2: $\mathcal{T}_{my\_id}$:=tokens at this node;
3: **for** $i := 0$ to $h - 1$ **do**
4: **if** my_id mod $2^{i+1} < 2^i$ **then**
5: $buddy\_id := i$-neighbor to the right;
6: **else**
7: $buddy\_id := i$-neighbor to the left;
8: **end if**
9: SEND $|\mathcal{T}_{my\_id}|/2$ tokens to node $buddy\_id$;
10: update $\mathcal{T}_{my\_id}$ accordingly;
11: $\mathcal{T}_{buddy\_id}$:=REVC tokens from node $buddy\_id$;
12: $\mathcal{T}_{my\_id} := \mathcal{T}_{my\_id} \cup \mathcal{T}_{buddy\_id}$;
13: **end for**

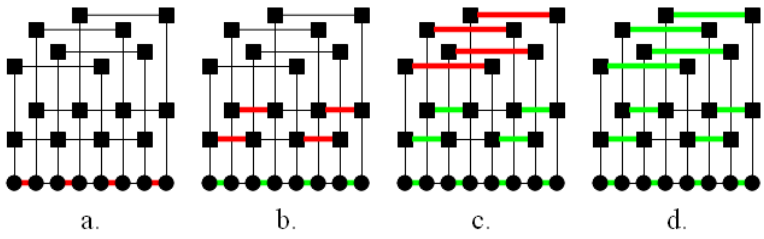

Figure 6.2: Skip Graph Token Distribution

6.2.2 DASIS

DASIS can be used to estimate the total number of peers in the simulated perfect skip graph system. To see this, consider the binary representation of a node $v =$

$(b_0...b_i...b_{h-1})_2$ of the perfect skip graph; obviously, v is connected to the necessary prefix-buddies: On level i for $i \in [0, h-1]$, v uses its right i-neighbor $v + 2^i = (b_0...\overline{b_{h-1-i}}...b_{h-1})_2$ if $b_i = 0$, and if $b_i = 1$ its left i-neighbor $v - 2^i = (b_0...\overline{b_{h-1-i}}...b_{h-1})_2$. Again, as all nodes have the same number of bits, at time t, all nodes store the same estimated value of the total number of peers in the system, and this value corresponds to the exact state of the network at time $t - h$.

6.2.3 Repairing Algorithm

Given the algorithms to distribute peers and to count the total number of peers in the system, we can apply the repairing algorithms of the hypercube also here. We will briefly sketch the local operations to be performed when the perfect skip graph increases its height: The description is given in Algorithm 10, and Figure 6.3 shows an example.

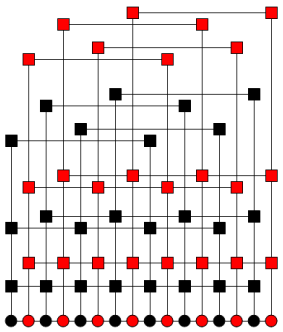

Figure 6.3: Expanding the Perfect Skip Graph

Reducing the dimension is also simple: Every node v for which $v \equiv 1 \pmod 2$ can just leave the network (maybe after copying its data to the left 0-neighbor). A node v with $v \equiv 0 \pmod 2$ has to shift all neighbors one level down: The new i-neighbors are the old $(i+1)$-neighbors.

6.3 Load Balancing and Range Queries

In the distributed hash table based on the d-dimensional simulated hypercube, a data item has been stored at the node whose identifier matches the first d bits of the hash value of the data item's identifier. This solution has two disadvantages: First, it glues the data items and the nodes together, making load balancing impossible. Moreover, as the hash function scrambles the name space of the identifiers, *similarity searches* become inefficient.

For the skip graph, we consider an alternative approach: Instead of mapping a data item to a fixed node, we require only that the data items are always sorted on the 0-ring with respect to their *real* identifiers. Besides efficient range queries, this allows also a very limited form of load balancing: If a node v has much more data items than its 0-neighbors, it can send some of its largest data items (with respect to their identifiers) to its right neighbor and some of its smallest data items to its left neighbor. This raises the question of how data items can still efficiently be found if they are no longer bound

Algorithm 10 Growth of Perfect Skip Graph

```
1: (* node v *)
2: create new node v';
3: for all levels i := 0...(h − 1) do
4:    i-neighbors become (i + 1)-neighbors;
5: end for
6: for all levels i := 1...h do
7:    SEND v' TO both i-neighbors;
8: end for
9: for all levels i := 1...h do
10:   (w'_0, w'_1) := RECV FROM i-neighbors w_0 and w_1;
11:   if i=1 then
12:      v' has 0-neighbors v and w_1;
13:      v has 0-neighbors w'_0 and v';
14:      v' has 1-neighbors w'_0 and w'_1;
15:   else
16:      v' has i-neighbors w'_0 and w'_1;
17:   end if
18: end for
19: h := h + 1;
```

to a fixed node. The solution is that every node stores the identifiers of the minimum and the maximum element of its adjacent nodes. Of course, there lies some overhead in updating these values all the times, but on the other hand it allows to make normal use of the “express lanes” of a skip list for efficient look-ups.

Chapter 7

Conclusions

The dynamics of the p2p systems presently in use is hardly understood. In this thesis we have introduced a simple dynamic model where there is a phase of quiescence between the joins and leaves of the peers. For both the k-ring topology and the simulated hypercube topology it was possible to adapt a repairing algorithm running in $O(1)$ rounds of quiescence to run also *concurrently* to an adversary which can perform a certain number of changes in a time interval of constant length. However, it is not obvious whether such a transformation exists for arbitrary topologies.

While the focus of Chapter 3 was on the presentation and comparison of different fault-tolerance models, Chapter 5 introduced a realistic distributed hash table, which maintains an efficient search structure and a peer degree $\Theta(d^2)$ against an adversary that triggers $\Theta(d)$ changes per time interval of constant length. However, there are still several possibilities to improve this system. For example it would be nice if the fault-tolerance is linear in the peer degree. We presume that there are alternative ways to inter-connect the hypercube's nodes to achieve this goal. Another desirable improvement of our system would be a mechanism for a graceful degradation or self-stabilization in the case of faults beyond the restrictions of the adversary.

We feel that several ideas presented in this document are directly applicable to other p2p systems. For example, it is possible to simulate a variety of other topologies, e.g. Chord. Moreover, some of our contributions such as the simple dynamic model or the dynamic analysis of the token distribution algorithm on the hypercube and on the perfect skip graph may be of interest on their own, i.e., beyond the applications for which we have used them.

Note that our main emphasis was on the maintenance of certain p2p *topologies*. The issue of the *data* usually stored by these systems has not been addressed for the k-ring at all, and the other systems lack a reasonable mechanism to distribute the data items uniformly amongst the peers. For example, as an orthogonal approach, it would also be possible to consider a "data insertion/deletion adversary".

Besides the open problems already mentioned, there is a variety of questions to be addressed in future research projects on the fault-tolerance of dynamic p2p topologies — not only for worst-case failures. For example

- Asynchronous systems: Real distributed systems are never synchronous. Many of our algorithms presented for the synchronous model work also in asynchronous systems, for example by using local synchronizers. However, such solutions usually come at the cost of an increased message complexity.

- Byzantine behavior: How can we cope with peers which do not act in perfect accordance with our protocols?
- Link failures
- Other topologies: E.g., maintenance of a dynamic skip graph which is not simulated?

Bibliography

[1] Napster. www.napster.com.

[2] K. Albrecht, R. Arnold, M. Gähwiler, and R. Wattenhofer. Aggregating Information in Peer-to-Peer Systems for Improved Join and Leave. In *4th IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2004.

[3] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 384–393. Society for Industrial and Applied Mathematics, 2003.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press, 2001.

[5] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. In *J. Parallel Distrib. Comput.*, volume 7, pages 279–301, 1989.

[6] N. Harvey and J. Munro. Deterministic skipnet. *Inf. Process. Lett.*, 90(4):205–208, 2004.

[7] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[8] D. Peleg and E. Upfal. The token distribution problem. In *SIAM J. Comput.*, volume 18, pages 229–243, 1989.

[9] C. G. Plaxton. Load balancing, selection sorting on the hypercube. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 64–73. ACM Press, 1989.

[10] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320. ACM Press, 1997.

[11] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[13] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems, 2002.

[14] B. A. Shirazi, K. M. Kavi, and A. R. Hurson. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[16] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[17] R. van Renesse and A. Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *IPTPS*, 2004.

[18] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, 2001.

Appendix A

Mean Deviation

In this appendix two different approaches to approximate the mean deviation of the symmetric binomial distribution are presented.

A.1 Approximation with Chernoff

Fact A.1 (Chernoff Lower Tail). *Let $X_1, ..., X_N$ be independent Bernoulli variables with $Pr[X_i = 1] = p_i$. Let $X := \sum_i X_i$ denote the sum of the X_i and let $\mu := E[X] := \sum_i p_i$ be the expected value for X. For $\epsilon \in (0, 1]$,*

$$Pr[X < (1-\epsilon)\mu] < \left(\frac{e^{-\epsilon}}{(1-\epsilon)^{(1-\epsilon)}}\right) < e^{-\mu\epsilon^2/2}.$$

Fact A.2.

$$\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}.$$

Theorem A.3. *Let $X \sim \mathcal{B}(n, 1/2)$ be binomially distributed with parameters n and $p = 1/2$. The expectation of the deviation from the mean $n/2$ is upper bounded by*

$$E[|X - n/2|] \leq \sqrt{\pi n}.$$

Proof. Let p_δ denote the probability that the deviation from the mean is at least δ, that is, $p_\delta = Pr[|X - n/2|] \geq \delta$. By symmetry, we have $p_\delta = 2 \cdot Pr[X \leq n/2 - \delta]$. For the expected deviation of the mean, we have

$$E[|X - n/2|] = \sum_{\delta=1}^{n/2} \delta \cdot Pr[|X - n/2| = \delta] = \sum_{\delta=1}^{n/2} p_\delta. \tag{A.1}$$

We can bound p_δ using Chernoff:

$$p_\delta = 2 \cdot Pr[X \leq n/2 - \delta] \leq 2e^{-\delta^2/n}. \tag{A.2}$$

Combining (A.1) and (A.2), we can bound the mean deviation by

$$\begin{aligned} E[|X - n/2|] &= \sum_{\delta=1}^{n/2} p_\delta \leq 2 \cdot \sum_{\delta=1}^{n/2} e^{-\delta^2/n} \\ &< 2 \cdot \sum_{\delta=1}^{\infty} e^{-\delta^2/n} \\ &< 2 \cdot \int_{\delta=1}^{\infty} e^{-\delta^2/n} d\delta \\ &= \sqrt{\pi n}. \end{aligned}$$

The integral after the last inequality can be calculated using Fact A.2 and the substitution $t = \delta\sqrt{n}$ and $d\delta = \sqrt{n}dt$. This concludes the proof. □

Remarks: There are two minor details which are neglected for readability of the above proof. First, although the Chernoff inequality gives an upper bound only for $Pr[X < (1-\epsilon)\mu]$, we use it for $Pr[X \leq (1-\epsilon)\mu]$. Second, the first equation in the proof holds for even n. For odd n, the deviation from the mean is not integral. Both issues can easily be solved.

A.2 Approximation with Stirling

The mean deviation of the symmetrical binomial distribution is given by:

$$\text{mean deviation} = 2^{-n} \sum_{k=0}^{n} \binom{n}{k} \left|k - \frac{n}{2}\right| = \begin{cases} \frac{n!!}{2(n-1)!!} & \text{, if } n \text{ odd} \\ \frac{(n-1)!!}{2(n-2)!!} & \text{, if } n \text{ even} \end{cases}$$

where $n!!$ is a double factorial, i.e.

$$n!! \equiv \begin{cases} n \cdot (n-2) \cdot ... \cdot 5 \cdot 3 \cdot 1 = \frac{(n+1)!}{2^{(n+1)/2}(\frac{n+1}{2})!} & \text{, if } n > 0 \text{ is odd} \\ n \cdot (n-2) \cdot ... \cdot 6 \cdot 4 \cdot 2 = 2^{n/2}(\frac{n}{2})! & \text{, if } n > 0 \text{ is even} \\ 1 & \text{, if } n = -1, 0 \end{cases}$$

According to Stirling's approximation[1] we have

$$\sqrt{2\pi} n^{n+1/2} e^{-n+1/(12n+1)} < n! < \sqrt{2\pi} n^{n+1/2} e^{-n+1/(12n)} \tag{A.3}$$

After some calculations, the following conjecture emerges, whose correctness can easily be verified.

$$\text{mean deviation} = 2^{-n} \sum_{k=0}^{n} \binom{n}{k} \left|k - \frac{n}{2}\right| \leq \sqrt{\frac{n}{\pi}}$$

[1] The double inequality A.3 is actually an extended version of Stirling's approximation.

Appendix B

Acknowledgments

After a very practical semester thesis in computer science which consisted to a large extent of programming, and a “classic” literature semester thesis in political economics, my minor subject, I was looking for something in the area of theoretical computer science for my diploma thesis. I decided in favor of this initially rather open thesis offered by the distributed computing group, because the potentially wide range of research areas appealed to me.

Four months later, I am very happy with my decision. I think I have got an insight to many aspects of how research in theoretical computer science works. Beginning with a general topic, first having to find interesting questions and models, up to the writing of a conference paper, I experienced the whole procedure.

I want to thank Fabian and Roger for many useful inputs and feedbacks.

Finally, I am grateful to my parents for their support during the course of my studies.