ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Master Thesis

eQuus: A Provably Robust and Efficient Peer-to-Peer System

Thomas Locher
lochert@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2005

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisor: Stefan Schmid

Abstract

Peer-to-peer systems (p2p) are highly dynamic in nature and the topology of these networks changes rapidly. Such a system may consist of millions of peers joining only for a limited period of time, resulting in hundreds of join and leave events per second. While there has been a lot of research on peer-to-peer computing over the last few years, we believe that the dynamics of such systems has not received the appropriate amount of attention. In this thesis, we introduce eQuus, a novel distributed hash table (DHT) suitable for highly dynamic environments. eQuus guarantees that lookups are always fast—as far as both the total number of hops as well as the effective distance traveled is concerned—, although peers may join and leave the network at any time and concurrently. All desirable properties such as a small peer degree, small network diameter and small stretch are maintained without incurring a large message overhead.

Contents

Chapter 1

Introduction

Much research in the last few years has been devoted to the development of efficient, structured peer-to-peer network overlays and a plethora of peer-to-peer systems [17, 20, 22, 25, 28] has been proposed. Nowadays, file sharing applications, which clearly have popularized peer-to-peer systems, still require servers for various purposes such as resolving queries or bootstrapping newly arriving peers, much like the first popular file sharing application Napster [18]. However, structured peer-to-peer systems appear to be emerging slowly as e.g. the file sharing application eMule [8] contains an early implementation of the Kademlia protocol [17]. All the proposed structured peer-to-peer systems belong to the class of distributed hash tables (DHT). A DHT is a decentralized and distributed system that partitions an ID space among all participating peers which are responsible for all keys that lie in their respective fraction of the ID space. The resulting structure is referred to as an *overlay network*, a common term for a network built on top of an underlying network. In a DHT, looking up the peer that is responsible for any key can be done efficiently, typically requiring $\mathcal{O}(\log n)$ hops, where n denotes the current number of peers in the system. Practically all of the proposed DHTs further guarantee small routing tables at each participating peer and also a good load balancing among all peers.

In a peer-to-peer network, any node[1] might disappear or fail at any given time, while new nodes strive to join the network. Therefore it is essential that the system adapt itself quickly to changes, and both random and correlated failures be detected and compensated efficiently. The permanent joining and leaving of peers at high rates, inflicting constant changes in the system, is called "churn."

Apparently, in order to attain these additional goals of providing a high resilience to churn and correlated failures, more communication among the nodes is needed, thus these goals conflict with the objective to reduce the communication overhead to a minimum. Apart from those aspects, it is also desirable, when trying to find a particular node, to route along paths that are not much longer than the direct paths, i.e. we want a system guaranteeing a low stretch. Hence, other aspects, including fault tolerance, dealing with churn, locality etc. are also of great importance.

Most systems, however, do not take these fundamental aspects into account

[1] In the rest of the thesis, the terms "node" and "peer" are synonymous.

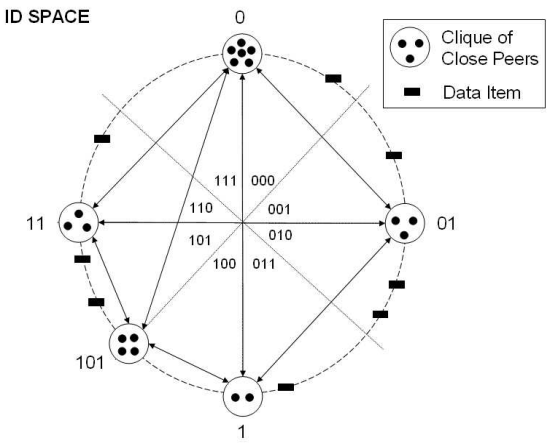

Figure 1.1: An example network consisting of 5 cliques is shown. Nodes that are close-by belong to the same clique and share the same ID. All nodes within the same clique are responsible for the same set of data items.

or only a small subset thereof. In general, most basic peer-to-peer overlays merely provide the functionality to find a particular node efficiently, given a key. As a result, any such basic overlay that does not consider these factors must be enhanced in order to meet these additional requirements.

We present a novel DHT, called eQuus[2], specifically designed to cope with high dynamics and failures without incurring a substantial message overhead. Furthermore, the system guarantees a low expected stretch, i.e. lookup paths are not much longer than the direct paths in expectation. The routing tables have size $\mathcal{O}(\log n)$ and keyed lookup requires $\mathcal{O}(\log n)$ messages with high probability, even if the network exhibits a highly dynamic behavior where nodes join and leave at a fast pace. Due to all these features, eQuus is ideal for large and dynamic networks.

As in many other DHTs, e.g. [17, 25], nodes are arranged in a hypercubic topology. In eQuus, however, *groups* of nodes that are close to each other according to the chosen proximity metric form the vertices of a partial hypercube. Within such a group, all nodes share the same node ID and all nodes know about each other[3]. Since the nodes in each group form a complete graph, these groups are denoted *cliques*.

In addition to the links to all other *clique members*, i.e. nodes that belong to the same clique, each node has links to nodes in other cliques. These additional links ensure that the entire system is connected and that paths from any node to another node are short. Data items are replicated among all nodes of a particular clique thereby introducing a natural form of redundancy. Since these nodes are close to each other, consistency can be maintained efficiently.

[2]Currently the most popular p2p network is the eDonkey network [7]. Equus is the latin word for horse which is a stronger and quicker animal than a donkey. Horses are gregarious animals which band together in order to protect each other, comparable to how robustness is established in the system by grouping nodes together.

[3]We will say that "node v knows about w" or alternatively "there is a link between node v and w", if node v stores the address of node w in its routing table.

What is more, this approach strongly reduces the communication overhead in the network in general, since nodes joining and leaving cliques only trigger communication among the nodes in the same clique and all other nodes in the system are not affected. Changes in the routing tables are only necessary if entire cliques appear or disappear, due to the arrival or departure of a large number of nodes. This entails that the dynamics of the network can be controlled better, for the life time of cliques is much longer than the life time of individual nodes.

Figure 1.1 depicts an example network consisting of 5 cliques. Apart from the affiliation of nodes to cliques, it is further shown how the nodes of the various cliques are interconnected.

Throughout this thesis, we assume that all nodes are uniformly distributed in a two dimensional Euclidean space. While this assumption is clearly not an ideal approximation of the node distribution in large networks such as the Internet, it allows for a simple formal analysis of some of the system's properties. Those results provide a good intuition that eQuus performs well in any real network.

The rest of the thesis is organized as follows. In Chapter 2, related work in this field is summarized. The design of the entire DHT is described in detail in Chapter 3. The ability of eQuus to cope with random failures and highly dynamic networks is treated in Chapter 4. The results of the simulations that have been run to demonstrate the good locality properties as well as the theoretical results are presented in Chapter 5. Chapter 6 shows how data items can be published and retrieved in eQuus. Directions for future research are outlined in Chapter 7, and Chapter 8 concludes.

Chapter 2

Related Work

Subsequent to the seminal work of Plaxton et al. [19], many p2p systems have been developed, e.g. [4, 5, 11, 13, 16, 17, 20, 25, 28]. All of these systems are scalable and provide fast lookups while each individual peer needs to store only a small amount of information about other peers in the network. In order to effectively cope with churn, any p2p system further has to provide specific mechanisms ensuring a high degree of fault-tolerance. While some solutions have been analyzed in this respect, it seems that only a few systems inherently incorporate fault-tolerance in the sense that robustness was a clear design goal from the beginning. We believe that this stands in contrast to the high dynamics observed in today's p2p networks.

In the following, we first review relevant related work on fault-tolerance. Fiat, Saia et al. [9, 23] have studied peer failures occurring in a worst-case fashion, for example caused by an adversary or a p2p worm exploiting the overlay network structure. The authors introduce a system where, with high probability, $(1 - \varepsilon)$-fractions of peers and data survive the adversarial removal of up to half of all nodes. However, the whole network has to be rebuilt from scratch if the total number of peers changes by a constant factor. Also Li et al. [15] analyze their system from a worst-case perspective. Using rigorous, formal proofs they show that their system tolerates concurrent, ongoing and asynchronous joins and leaves of peers. Unfortunately, leaving peers are not allowed to crash; instead, they have to execute an appropriate exit protocol. A more practical study by Rhea et al. [21] compares DHTs by simulation and shows that several structured p2p overlays cannot handle churn rates as high as those observed in today's p2p networks.

eQuus shares the most commonalities with the work by Kuhn et al. [14]. Similarly to eQuus, [14] achieves a better robustness by having more than one peer responsible for each ID, as opposed to assigning a unique ID to each node. In contrast to our work, their system has higher maintenance costs as it requires a background process estimating the current network size in order to balance the ID assignment through a global operation. On the contrary, eQuus reacts to imbalances using *local* merges or splits of adjacent IDs only. An additional advantage of eQuus is that it has a low expected stretch, as described in a later chapter.

In order to evaluate the performance of lookup operations in p2p systems, usually only the total number of hops is considered. In systems such as [14]

or [25], the neighbors of a given peer are determined by applying hash functions to the peer's IP address and potentially other parameters. Thus, the resulting structure is completely independent of the actual geographic peer locations. It is therefore possible that, with each hop, a node on a different continent is contacted, even if the target node is close-by, which is clearly not desirable. Furthermore, when data items are replicated among nodes with numerically close identifiers, constantly ensuring that the data items are stored on the correct nodes is potentially a costly operation, since these nodes might be very far away.

The approach taken by Pastry [22] when performing a lookup is to choose the peer that is the geographically closest among a possible set of neighbors. According to their simulations, Pastry achieves a low stretch of around 1.3 to 1.4. However, in contrast to our work, this is a heuristic approach and there are no provable bounds on the achieved stretch. While solutions with provably low stretches are well-known (e.g. [1, 2]), they usually lack the robustness property.

In contrast, eQuus tries to combine all these properties while at the same time having a low maintenance overhead.

Chapter 3

System Overview

In eQuus, groups of nodes that are close-by form *cliques*. Within such a clique c, each node has the same ID $c.id$, which is a bit string of a predefined length d. The length d of the IDs is referred to as the *dimension* of the network. Every clique has its own unique ID. The ith bit is denoted $c.id[i]$ and the sub-ID in the range $[i, j], i > j$ is denoted $c.id[i, j]$. Note that the highest order bit is the leftmost bit. Since these nodes share the same identifier, they are also responsible for the same fraction of the ID space. This has two interesting properties. First of all, these nodes ensure a certain degree of redundancy that is required lest data is lost due to the sudden departure or failure of a particular node. Second, if those nodes storing the same information are close to each other, establishing consistency among those nodes can be done quickly due to the short distance between them. The distance between nodes is measured with the metric $c : V \times V \rightarrow \mathbb{R}_0^+$, where V denotes the set of all nodes in the system.

Since the degree of each node should not exceed $\mathcal{O}(\log n)$, the number of nodes in a clique has to be limited. Once the number of nodes undershoots a certain threshold, we are confronted with a higher probability of data loss. This observation entails that, once the number of nodes reaches a specific upper bound, this clique has to be split into two cliques. Likewise, if the number of nodes reaches a certain lower bound, the remaining nodes in the clique have to join another clique, thus the two cliques have to merge. Hence it follows that, apart from the standard operations, such as JOIN and LOOKUP, two additional operations, namely MERGE and SPLIT, are essential in eQuus.

We will first present the link structure that guarantees connectivity and fast lookups. Subsequently, the JOIN procedure is specified, followed by a detailed description of the MERGE and SPLIT procedure. In the last part of this chapter, the LOOKUP procedure is presented.

Note that there is no need for a specific LEAVE protocol. After a node could not be contacted by any clique member for a certain period of time, it is simply excluded from the clique and thus, from the system.

3.1 Link Structure

A certain amount of links between the nodes in the network has to be maintained and updated periodically, in order to establish a structured network in

which lookups are fast, the permanent joining and leaving of nodes is handled efficiently and a high degree of resilience to random as well as correlated failures is guaranteed.

As mentioned in the previous chapter, each node v knows about all other members of its clique c. Apart from these links, each node v has links to nodes in other cliques in order to guarantee connectivity and fast lookups in the network.

The routing is basically a generalized form of routing on hypercubes, where b bits are corrected in each hop. The number of bits b that can be corrected in a single hop is denoted the *base* of the system. In this kind of routing mechanism, referred to as *prefix-routing*, for each of the $2^b - 1$ other values for the highest order block of b bits of its ID, a clique is stored in the routing table whose ID starts with the corresponding block of b bits. In addition, for each of the $2^b - 1$ other values for the second block of b bits, a clique is stored whose ID shares the same highest order block and the second block matches the corresponding block of b bits etc. For example, setting $b := 2$, a node in the clique with ID 10001101 would store cliques whose IDs start with the following prefixes:

Block	Prefixes
1	00, 01, 11
2	1001, 1010, 1011
3	100000, 100001, 100010
4	10001100, 10001110, 10001111

This procedure is very similar to the one described for instance in [22], the main difference being that in eQuus, each entry represents an entire clique and not a single node. Let Π_c^b denote the set of prefixes that are relevant for clique c with ID $c.id$, given base b.

In order to ensure permanent connectivity, a constant number of k nodes are known in each of the approximately $(2^b-1)\lceil \log_{2^b} n \rceil$ cliques in the routing table, thus in total $k(2^b - 1)\lceil \log_{2^b} n \rceil$ links are stored in the routing table, for a given base b. The average lookup path requires about $\lceil \log_{2^b} n \rceil$ hops, as we will show in Section 3.5. In comparison to $b = 1$, by setting $b := 4$, only $\frac{2^4-1}{4} = 3.75$ times more contacts have to be stores, but the number of hops is reduced by a factor of 4. It is checked periodically if the nodes in the routing table are still alive. Every time a node in a specific clique is contacted, it returns a fresh, random list of k live nodes in the corresponding clique and this list is stored, replacing the old list in the routing table. In case all k nodes of a certain clique stored in the routing table have failed since the last request, which is an improbable event, another node in the own clique can be contacted in order to get a fresh list of nodes in this clique. Because all nodes in a clique store links to the same cliques, however they do not share the same subset of nodes in this clique, it is very likely that there is another clique member storing the address of a node in the corresponding clique that is still alive.

Additional k links lead to nodes in the clique that is the predecessor in the ID space and another k links lead to nodes in the successor clique. Thus, the total number of links is bounded by $\mathcal{U} + k(2^b - 1)\lceil \log_{2^b} n \rceil + 2k = \mathcal{O}(\log n)$ links, where $\mathcal{U}$ denotes the upper bound on the number of nodes in a clique. Knowing both the predecessor clique and the successor clique is necessary for the MERGE and also the SPLIT operation. We say that a clique c, or any node in clique c, is

responsible for a data item, if its key is in the range $[c.id, c.successor.id)$, where $c.successor.id$ denotes the ID of the successor clique of c. In the following, let N_v denote the neighborhood of node v consisting of all cliques in its routing table.

The routing table consists of the *PRED/SUCC table*, storing the IDs of the predecessor clique and the successor clique, the *link table*, storing the IDs of approximately $\lceil (2^b - 1) \log_{2^b} n \rceil$ other cliques and the *clique tables*, storing the links to nodes for each clique.

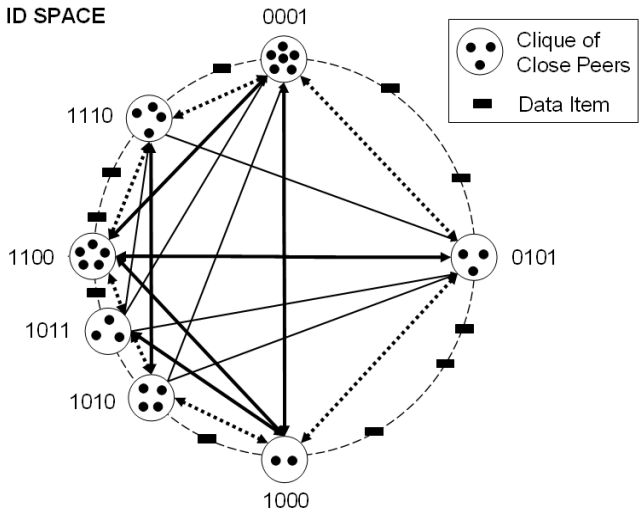

Figure 3.1: An example network consisting of 7 cliques is shown. It is further displayed how these cliques are interconnected. The base b of the network is 2.

Another network, depicted in Figure 3.1, is used in order to demonstrate the use of the various tables. The base b of the network is 2 in this example. Note that there are both bidirectional links, which are depicted using thick lines, and unidirectional links, depicted using thin lines, between the cliques. There is a unidirectional link from clique c to clique c' if the ID $c'.id$ is stored in the routing table of c but not vice versa. This is the case if the other clique has a different entry for this specific prefix in its routing table. Dashed lines connect neighboring cliques in the ID space. In the following, some of the routing tables of the clique with ID 1010 are presented.

PRED/SUCC table of clique 1010

Predecessor	Successor
1000	1011

Link table of clique 1010

Prefix	Clique ID
00	0001
01	0101
11	1110
1000	1000
1001	-
1011	1011

Note that there can be "holes" in the link table. In the example, there is no clique whose ID starts with 1001, thus the link table entry for this prefix remains empty.

Each node stores a clique table for each clique in the PRED/SUCC and link table. In those clique tables, the addresses of k nodes that belong to the corresponding clique are stored, together with the distance to this node, if this information is known. There is one clique table that is special, denoted *clique member table*. This particular clique table stores the addresses of each node in its own clique plus the distances to those nodes, thus the difference to a regular clique table is that it contains up to $\mathcal{U}$ entries, since cliques can contain up to $\mathcal{U}$ nodes. All the information about the data item this clique is responsible for is stored in the *data table*. This table is the subject of a latter chapter.

```
1: for all π ∈ Π_c^b do
2:   if ζ(π) ∉ [c.id, c.successor.id) then
3:     if linkTable.hasEntry(π) then
4:       UPDATE_LINK(π);
5:     else
6:       response = LOOKUP(ζ(π));
7:       c̃ = response.getClique();
8:       if ψ(ζ(π), c̃.id) ≥ |π| then
9:         linkTable.setLink(π, c̃.id);
10:        cliqueTable = cliqueTables.getTableForPrefix(π);
11:        [a_1, ..., a_k] = response.getAddresses();
12:        cliqueTable.setAddresses([a_1, ..., a_k]);
13:      end if
14:    end if
15:  end if
16: end for
```

Algorithm 3.1: UPDATE_ROUTING_TABLE (Code for node v in clique c)

It is essential to keep the entries in the routing tables accurate. The algorithm that ensures the accuracy of routing information is depicted in Algorithm 3.1. Note that the algorithm is correct for arbitrary bases b.

The following functions are used in this algorithm as well as in several other algorithm in this chapter. The function $\zeta : \{0,1\}^l \to \{0,1\}^d$ appends zeros to a bit string of length $l \leq d$, i.e. $\zeta(s) := s \mid\mid 0^{d-|s|}$ where $|s| \leq d$. The function $\psi : \{0,1\}^l \times \{0,1\}^l \to \{0,\ldots,l\}$ determines the length of the shared prefix of two bit strings of length l, for example $\psi(101, 100) = 2$.

All prefixes $\pi \in \Pi_c^b$ have to be checked in the algorithm. However, if a string $\zeta(\pi)$ is in the range $[c.id, c.successor.id)$, there is no need to check this specific prefix π, because either the clique c itself is responsible for any bit string with this prefix or the ID of the successor clique starts with this prefix, thus no other clique has to be found for this prefix. If this is not the case and there is an entry for this prefix, UPDATE_LINK(π) has to be called. Otherwise, there is a hole at this position and we have to search for the clique that is responsible for this prefix. If the responsible clique has a prefix of the desired length, it is added to the link table and the addresses of k nodes in this clique are stored in the clique table for this prefix.

```
 1: ID = linkTable.getIDForPrefix(π);
 2: cliqueTable = cliqueTables.getTableForPrefix(π);
 3: received = FALSE;
 4: while cliqueTable.hasEntry() and not received do
 5:    w = cliqueTable.getNode();
 6:    try{
 7:       send([LINK_UPDATE_REQUEST,π,ID]) to w;
 8:       response = receiveMessage();
 9:       received = TRUE;
10:    }catch(timeOut){
11:       cliqueTable.remove(w);
12:    }
13: end while
14: if received then
15:    if response.getType() == LINK_OK then
16:       [a_1, ..., a_k] = response.getAddresses();
17:       cliqueTable.setAddresses([a_1, ..., a_k]);
18:    else if response.getType() == NEW_LINK then
19:       newID = response.getID();
20:       [a_1, ..., a_k] = response.getAddresses();
21:       linkTable.setLink(π, newID);
22:       cliqueTable.setAddresses([a_1, ..., a_k]);
23:    else
24:       linkTable.removeEntry(π);
25:    end if
26: else
27:    linkTable.removeEntry(π);
28: end if
```

Algorithm 3.2: UPDATE_LINK(π) (Code for node v in clique c)

If there is an entry in the routing table, then UPDATE_LINK(π) is invoked, see Algorithm 3.2. This procedure sends LINK_UPDATE_REQUESTs to nodes in the targeted clique, until it receives an answer. Depending on the type of the received message, the link table entry and the corresponding clique table are updated. If the link is still up-to-date (LINK_OK), the k fresh addresses are retrieved and stored. If a new and somewhat superior link is available (NEW_LINK), the link table entry for this prefix and the clique table are updated accordingly. Otherwise, the message indicates that the clique does no

longer exist and no suitable replacement has been found (LINK_LOST). Consequently, the entry for the prefix π is removed.

```
1: {Received from node u in clique ĉ for prefix π with ID id in its table}
2: c̃ = arg max_{c'∈{N_v∪{c}} | ψ(ζ(π),c'.id)≥|π|} ψ(c'.id[d − |π|, 1], ĉ.id[d − |π|, 1]);
3: if c.id == id and c == c̃ then
4:    [a_1, ..., a_k] = getNodesFromClique(c);
5:    send([LINK_OK,π, [a_1, ..., a_k]]) to u;
6: else
7:    if c̃ ≠ ∅ then
8:       [a_1, ..., a_k] = getNodesFromClique(c̃);
9:       send([NEW_LINK,π, c̃.id,[a_1, ..., a_k]]) to u;
10:   else
11:      send([LINK_LOST,π]) to u;
12:   end if
13: end if
```

Algorithm 3.3: LINK_UPDATE_REQUEST RECEIVED (Code for node v in clique c)

How a node reacts upon receiving a LINK_UPDATE_REQUEST is described in Algorithm 3.3. The node first computes the best possible clique for prefix π and the ID of the requesting clique among all cliques stored in its routing table and also the clique it belongs to itself.

The best possible clique is determined using a strategy that we call *low indegree strategy*. Among all those cliques, it chooses the clique with the longest matching suffix following the specified prefix. The function ψ is used in order to determine the clique with the longest matching suffix, by considering only the part of the ID after the shared prefix. Provided that the clique that the request has been sent to is still the clique with the longest matching suffix and the ID id stored in the routing table of node u is still correct, then a newly compiled list of k clique members can be returned. If either the ID is no longer correct, due to a recent merging, or if there is a better clique in the neighborhood (or both), then node u is informed about this new contact, by sending a NEW_LINK message. In case there is no node anymore with the desired prefix, a LINK_LOST message is sent back to node u in order to invalidate this obsolete link table entry.

Using this low indegree strategy has two consequences. First, when looking at a network in which all updates after MERGE and SPLIT operations have been performed, the link structure is well-defined. This is interesting, given the high flexibility of the link table entries. The second property is more technical-oriented: The expected indegree is bounded by $\mathcal{O}(\log n)$.

The explanation is simple. In expectation, the same amount of clique IDs are "around" the ID of node v as there are cliques whose IDs are numerically close to the clique in the link table entry for a certain prefix. This means that a constant number of links point to the own clique while others point to cliques with a numerically closer ID. This holds for all prefixes, thus the expected indegree is $\mathcal{O}(\log n)$ and hence the name of the strategy. Figure 3.2 shows how a link table entry is updated after a clique split.

It is clear that each entry in the link table can be updated in constant time. For those prefixes in Π_c^b for which there is no entry in the link table, a

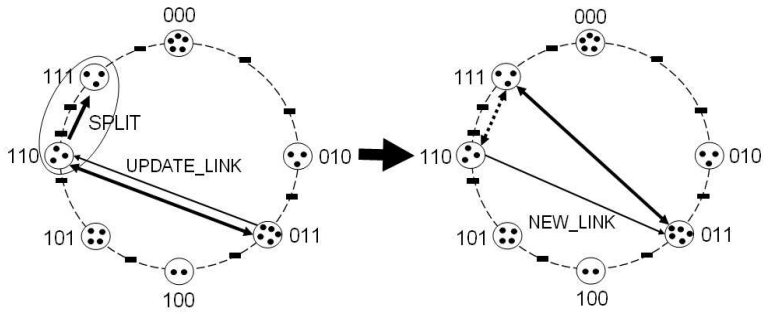

Figure 3.2: The nodes in the clique with ID 011 **will learn that the clique with ID** 110 **split as soon as a NEW_LINK message is received. The nodes in the clique with ID** 011 **can then update their routing tables accordingly.**

single lookup call determines whether a clique with the desired prefix exists in the network. Even if there is no such clique, a small number of nodes in the clique that is responsible for this fraction of the ID space can be stored. The next update can then be started at this clique, which will very likely result in lookups requiring not more than one or two hops.

3.2 Description of the JOIN Algorithm

In order to ensure that all nodes belonging to the same clique are close to each other and also that good locality properties such as a low stretch are maintained, a newly arriving node must join the closest clique in the system.

```
 1: counter = 0;
 2: progressMade = TRUE;
 3: [bestNode, bestDist] = getBootstrapNode();
 4: while counter < Λ and progressMade do
 5:    send(JOIN_REQUEST) to bestNode;
 6:    [v_1, ..., v_z] = receiveSet();
 7:    sendToSet(MEASURE_DISTANCE) to [v_1, ..., v_z];
 8:    [(v_1, l_1), ..., (v_z, l_z)] = receiveMeasurements();
 9:    shortestDist = l_i | (v_i, l_i) ∈ [(v_1, l_1), ..., (v_z, l_z)] ∧ l_i ≤ l_j ∀j ∈ [1, z];
10:    closestNode = v_i | (v_i, l_i) ∈ [(v_1, l_1), ..., (v_z, l_z)] ∧ l_i ≤ l_j ∀j ∈ [1, z];
11:    if shortestDist < bestDist then
12:       bestNode = closestNode;
13:       bestDist = shortestDist;
14:    else
15:       progressMade = FALSE;
16:    end if
17:    counter = counter + 1;
18: end while
19: send(JOIN) to bestNode;
20: stateInformation = receiveStateInformation();
```

Algorithm 3.4: JOIN (Code for node v joining the system)

How a given node determines the closest clique is straightforward, see Algorithm 3.4. The overall procedure is similar to the mechanism described in [27]. In the first step, a node contacts an arbitrary bootstrap node. The contacted node returns the address of one node of each clique in its routing table. Let z denote the number of cliques stored. The new node then contacts all those nodes and determines the closest one, e.g. by sending several ping messages and waiting for the replies. Subsequently, a JOIN request is sent to the closest node which again returns addresses of cliques in its routing table. This step is repeated until the closest clique has been found, which will happen in $\mathcal{O}(\log n)$ rounds. In practice, an upper bound on the number of rounds ought to be specified in order to avoid looping between cliques. In Algorithm 3.4, the number of rounds is bounded by Λ, which should be in the order of $\frac{d}{b}$, because each round basically determines additional b bits of the ID of the closest clique. After sending a JOIN message to a node in the closest clique, this contacted node will inform all the other nodes in the clique about the arrival of a new node and give the new node all the information it needs to become a fully integrated clique member, which includes the ID of the clique, the addresses of all other clique members, the routing table and also the necessary information about all data items whose keys lie in this clique's fraction of the ID space. The routing table can be copied from any other clique member, since they all share links to the same cliques. The total message complexity is obviously $\mathcal{O}(\log^2 n)$, since $\mathcal{O}(\log n)$ messages have to be sent in each of the $\mathcal{O}(\log n)$ rounds.

The first clique in the system bears the ID 0^d. As soon as it contains $\mathcal{U}$ nodes, it is split into two cliques, each obtaining half of the nodes. One clique keeps the ID 0^d while the other clique gets the new ID 10^{d-1}. The new clique with ID 10^{d-1} is the successor clique of the clique with ID 0^d and vice versa. Different strategies as to how the two sets are constructed can be applied. A good strategy, as far as maintaining a good locality is concerned, is to find the $\frac{\mathcal{U}}{2}$ nodes that are closest to nodes in the preceding clique in the ID space. Those nodes keep their ID while the other half changes their ID and forms a new clique. This strategy does not work at the beginning, because there is only one clique in the system. A solution to this problem is to find the node in the "center" of the clique and have it decide how the nodes are to be split up. The nodes with ID 0^d are then responsible for all data items whose key is in the range $[0^d, 01^{d-1}]$ and the nodes with ID 10^{d-1} are responsible for the keys in the range $[10^{d-1}, 1^d]$ in the ID space. A newly arriving node always joins the closest clique. This clique is split again once its size reaches $\mathcal{U}$ and the new clique gets the ID in the middle between the ID of the current clique and the successor clique etc.

Nodes within a clique have to communicate permanently in order to keep their routing tables consistent. Since all members of a clique are likely to be close to each other according to the chosen proximity metric and since links to cliques far away have to be checked only occasionally, a high percentage of the generated traffic is over short distances.

3.3 Description of the MERGE Algorithm

In case a node joins or leaves a clique, it has to be verified that the number of nodes in the clique is still in the range $[\mathcal{L},\mathcal{U}]$, where $\mathcal{L}$ and $\mathcal{U}$ denote the lower and

upper bound on the number of nodes in a clique, respectively. It is important to set both $\mathcal{L}$ and $\mathcal{U}$ to reasonable values, such that enough redundancy is introduced and data loss can be avoided, but the routing tables do not become too large. In the following, let $\mathcal{L} := \frac{d}{2}+1$ and $\mathcal{U} := 2d-1$, which yields adequate values for typical values for d, such as 64 or 128.

Once the size of a clique reaches $\mathcal{L}-1 = \frac{d}{2}$, it has to merge with the preceding clique in the ID space. We assume that each clique has a leader, e.g. the node with the lowest IP address. Note that there is no need for a leader election since all nodes in a clique know about each other. What is more, the leader neither has any particularly hard tasks nor is he a focal point in any procedure, thus the leader is never a bottleneck. Having a leader is required due to the asynchronous nature of communication between nodes. The leader acts as a synchronizer for both MERGE and SPLIT operations. In case the leader fails while performing any of these two operations, the new leader can—upon determining that the old leader failed—resume the operation without much delay. When the leader realizes that after a clique member left, the clique merely contains $\frac{d}{2}$ nodes, the procedure MERGE is called.

```
1: sendToSet(MERGE_PENDING) to cliqueMembers;
2: [answer_1, ..., answer_m] = receiveSet();
3: if MERGE_NACK ∈ [answer_1, ..., answer_m] then
4:    establishConsistency();
5: else
6:    send([MERGE_REQUEST,stateInformation]) to u ∈ c.predecessor;
7:    newState = receiveStateInformation();
8:    sendToSet([MERGING,newState]) to cliqueMembers;
9:    send([NEW_PRED,c.predecessor.id]) to w ∈ c.successor;
10:   c.id = c.predecessor.id;
11:   c.predecessor = newState.getPredecessor();
12:   cliqueMembers = cliqueMembers ∪ newState.getCliqueMembers();
13:   linkTable = newState.getLinkTable();
14:   dataTable = dataTable ∪ newState.getDataTable();
15: end if
```

Algorithm 3.5: MERGE (Code for leader v in clique c)

The MERGE operation is summarized in Algorithm 3.5. At first, the leader asks the other m clique members, if they are ready to merge. This is denied if there is no consensus on the number of nodes in the clique. In this situation, the procedure establishConsisteny() aims at reestablishing the necessary consistent state. This procedure is not further discussed. Otherwise, the request to merge is sent to a node belonging to the predecessor clique, along with all the necessary state information, i.e. the addresses of all clique members, the ID of the successor clique and also the information stored in the data table about all data items this clique has been responsible for. In return, it receives the state information it needs to update its tables, which is forwarded to all clique members. The successor clique is also informed about the merging in order to adapt its predecessor link. All nodes in the merging clique have to adopt the ID, the predecessor clique and also the entire link table of their former predecessor clique. Both the clique member tables and the data tables of the two previously

separated cliques have to be combined. Nodes in all other cliques that have links to the clique that has just merged will be informed about the topology change as soon as they update the corresponding routing table entry.

```
1: {Received request and state information newState from node u}
2: send(MERGE_REQ_RECEIVED) to leader;
3: response = receiveMessage();
4: if response == MERGE_OK then
5:    sendToSet([ADD_STATE_INFO,newState]) to cliqueMembers;
6:    send(stateInformation) to u;
7:    c.successor = newState.getSuccessor();
8:    cliqueMembers = cliqueMembers() ∪ newState.getCliqueMembers();
9:    dataTable = dataTable ∪ newState.getDataTable();
10: else
11:    timeOut()   { Retry after timeout }
12: end if
```

Algorithm 3.6: MERGE_REQUEST RECEIVED (Code for node v)

The node receiving the MERGE_REQUEST first contacts the leader which coordinates the MERGE and SPLIT operations. If the leader sends a MERGE_OK message, state information is forwarded to the other clique members which update their state information accordingly.

It is possible that after the merging of two cliques, they are split again almost immediately. This is the case if one clique consists of at least $\frac{3}{2}d$ nodes. By merging two cliques and splitting them again, as opposed to merely moving nodes from the clique with fewer nodes to the larger clique, an improved partitioning of the ID space is achieved. For example, if the clique with ID 1010 has to merge with its predecessor clique with ID 1001 and its successor clique has ID 1111, then splitting clique 1001 afterwards results in a clique with ID 1100 between the two cliques with IDs 1001 and 1111, respectively. Hence, the ID space is partitioned locally in an optimal manner.

3.4 Description of the SPLIT Algorithm

In case the size of a clique exceeds $\mathcal{U}$, the SPLIT procedure is initiated in which half of the nodes form a new clique. The nodes in the old and the newly created clique are then responsible for approximately half of the data items they were responsible before. In order to ensure a certain locality, the nodes closer to the preceding clique in the ID space keep their ID while the others get the new, higher ID, which is the ID between the current ID and the ID of the successor clique. The closest node to the preceding clique determines the set of nodes that will keep the old ID and the set of nodes that will get a new ID. The new ID is determined using the σ-function, see Algorithm 3.7.

The procedure SPLIT is described in greater detail in Algorithm 3.8. In this procedure, the leader initiates the splitting by asking for measurements of the distance to the predecessor clique. Each of the m clique member returns the closest distance measured to any node in the predecessor clique.

The successor clique is informed that it has a new predecessor clique, by sending a NEW_PRED message including the ID $\sigma(c.id, c.successor.id)$ of its

```
1: newID = (c.id + c.successor.id) ≫ 1;
2: if c.id[d] > c.successor.id[d] or c.id = c.successor.id then
3:    newID = newID ⊕ 10^{d−1};
4: end if
5: return newID;
```

Algorithm 3.7: σ-function applied to clique c.

```
1: sendToSet(SPLIT_PENDING) to cliqueMembers;
2: [answer_1, ..., answer_m] = receiveSet();
3: if SPLIT_NACK ∈ [answer_1, ..., answer_m] then
4:    establishConsistency();
5: else
6:    sendToSet(PREDECESSOR_DIST) to cliqueMembers;
7:    [(v_1, l_1), ..., (v_m, l_m)] = receiveMeasurements();
8:    u = getClosestNodeFromPredecessor();
9:    allDist = [(v_1, l_1), ..., (v_m, l_m)] ∪ [(v, u)];
10:   closestNode = v_i | (v_i, l_i) ∈ allDist ∧ l_i ≤ l_j ∀j ∈ [1, m + 1];
11:   send(FORM_CLIQUES) to closestNode;
12:   send([NEW_PRED,σ(c.id, c.successor.id)]) to w ∈ c.successor;
13: end if
```

Algorithm 3.8: SPLIT (Code for leader v in clique c)

new predecessor to a node belonging to the successor clique. The leader sends a FORM_CLIQUES message to the node that is closest among all clique members. This particular node v is then responsible for the actual operation, see Algorithm 3.9.

Node v partitions all nodes into two sets of equal size. The set keepID consists of the nodes that are closest to v, including node v itself. The other half of the nodes belongs to the set changeID. The nodes in the set changeID are those nodes that will form a new clique with the ID $\sigma(c.id, c.successor.id)$. In the next step, it forwards the set keepID to all other clique members. The clique members can then, depending on whether they belong to the set keepID or not, update their PRED/SUCC table and their clique member table. The nodes in the set changeID further have to build a new link table; however, they can very likely keep the higher entries in the table. It is easy to see that the length of the prefix shared with the old ID determines the number of link table entries that can be kept.

Both cliques are then responsible for approximately half of the data items that they were responsible before and can remove the information about the data items whose keys do not lie in their respective ranges $[c.id, c.successor.id)$ from their data tables.

3.5 Description of the LOOKUP Algorithm

The main operation that the network has to be able to perform is lookup a clique, given a specific key $s \in \{0,1\}^d$. We define the metric $\delta : \{0,1\}^d \times \{0,1\}^d \rightarrow \{0,\ldots,d\}$ as follows. For any bit two strings s, s' of length d, $\delta(s,s') = i \iff s[d, i+1] = s'[d, i+1] \wedge s[i] \neq s'[i]$, where $s[i,j], i > j$, denotes the substring

```
 1: keepID = getClosestCliqueMembers();  { including v }
 2: changeID = cliqueMembers - keepID;
 3: sendToSet([KEEP_ID,keepID]) to cliqueMembers;
 4: if v ∈ keepID then
 5:    cliqueTable = cliqueTables.getSUCCTable();
 6:    [a_1, ..., a_k] = chooseAddresses(changeID);
 7:    cliqueTable = cliqueTable.setAddresses([a_1, ..., a_k]);
 8:    c.successor.id = σ(c.id, c.successor.id);
 9:    cliqueMembers = keepID;
10: else
11:    cliqueTable = cliqueTables.getPREDTable();
12:    [a_1, ..., a_k] = chooseAddresses(keepID);
13:    cliqueTable = cliqueTable.setAddresses([a_1, ..., a_k]);
14:    c.predecessor.id = c.id;
15:    c.id = σ(c.id, c.successor.id);        { New ID }
16:    cliqueMembers = changeID;
17:    linkTable.cutToRange([d, d − ψ(c.id, c.predecessor.id) + 1]);
18: end if
19: dataTable.cutToRange([c.id, c.successor.id));
```

Algorithm 3.9: FORM_CLIQUES RECEIVED (Code for node v in clique c)

of s in the range $[i, j]$. Thus, $\delta(s, s')$ is the position of the highest order bit at which the two bit strings s and s' differ. It holds that $\delta(s, s') \geq 0$ and $d(s, s') = 0 \iff s = s'$. Furthermore, $d(s, s') = d(s', s)$ (symmetry) and $d(s, s'') \leq d(s, s') + d(s', s'')$ (triangle equality). The triangle equality holds because if $d(s, s'') = i$, then either $d(s, s') = i$ or $d(s', s'') = i$ and the fact that distances are always positive. For any two bit strings $s, s', |s| = |s'| = d$, it further holds that $\delta(s, s') + \psi(s, s') = d$. This metric allows for a simple specification of the lookup protocol. It is further useful when it comes to analyzing the locality properties of eQuus.

The following simple algorithm performs this task independent of the chosen base, see Algorithm 3.10. It is first checked if node v in clique c is responsible for the key s, by testing if s lies between its ID and the ID of the successor clique. If this is not the case, it will forward the request to a node in a clique with a longer matching prefix. If there is no such node in the routing table, two cases have to be considered. If $s > v.id$, the request is forwarded to the clique with the largest ID in N_v, subject to the constraint that the length of the matching prefix is not reduced. Otherwise, i.e. $v.id > s$, the request is simply forwarded to a node in the predecessor clique.

The following theorems summarize the properties of the LOOKUP algorithm of eQuus. In this analysis, it is assumed that all routing tables are accurate, i.e. each node knows about all other cliques that currently exist in the system and are relevant for its routing table. Note that this does not mean that all lists of k nodes for each clique must be up-to-date. Merely the information about the existence of cliques must be accurate and at least one of the k nodes of each clique stored in the routing table must be still alive.

Theorem 3.5.1 *The LOOKUP algorithm is correct, i.e. it always finds the clique responsible for any key, if the routing table entries are accurate.*

```
1: {u started LOOKUP request for key s}
2: if s ∈ [c.id, c.successor.id) then
3:    send(LOOKUP_DONE,c.id) to u;
4: else
5:    c̃ = arg min_{c'∈N_v} δ(c'.id, s);
6:    if not δ(c̃.id, s) < δ(c.id, s) then
7:       if s > c.id then
8:          c̃ = arg max_{c'∈N_v:δ(c'.id,s)=δ(c.id,s)} c'.id;
9:       else
10:         c̃ = c.predecessor;
11:      end if
12:   end if
13:   n =getNodeFromClique(c̃);
14:   send(LOOKUP,u,s) to n;
15: end if
```

Algorithm 3.10: LOOKUP (Code for node v in clique c)

PROOF. Let the start node be v in clique c. It tries to find the clique responsible for the key s. If the key is in its own fraction of the ID space, i.e. $s \in [c.id, c.successor.id)$, then the LOOKUP procedure terminates correctly. Otherwise, it checks its routing table for a clique whose ID has a longer matching prefix. If such a clique exists, then the request will be forwarded to this clique. The new clique is closer to the responsible clique, since the matching prefix is at least one bit longer.

If there is no such clique in N_v, two cases have to be distinguished. If $s > c.id$, then we have to forward the request to a clique with a larger ID. Let $i = \delta(c.id, s)$, thus $c.id[i] = 0$, $s[i] = 1$ and $c.id[j] = s[j]$ for all $i < j \leq d$. We assume that routing tables are accurate, thus there is no clique $\tilde{c}$ in the network whose ID is larger than the ID of c, $\delta(\tilde{c}, s) = i$ and $\tilde{c}.id[i] = 1$, otherwise v would know about it. Thus, it is best to forward the request to the clique with the largest ID among all cliques $\tilde{c}$ in N_v for which it holds that $\delta(\tilde{c}, s) = i$. There is always at least one clique satisfying this constraint. Assuming no such clique exists in the routing table, then it follows that $\delta(c.successor.id, s) > i$ and therefore $c.successor.id > s$, in contradiction to the assumption that $s \notin [c.id, c.successor.id)$.

Similarly, if $c.id > s$, it holds that $c.id[i] = 1$ and $s[i] = 0$. If there is no clique $\tilde{c}$ such that $\delta(\tilde{c}.id, s) < \delta(c.id, s)$, then it holds for all cliques with a lower ID than c that their identifiers are lower than s. Naturally, this also holds for the predecessor of c and thus the lookup terminates at the predecessor, because $c.predecessor.id < s < c.id$.

Therefore, requests are routed closer and closer to the destination in every step, which concludes the proof.

□

It is also essential to bound the number of hops required to reach the target clique. The following theorem states that, w.h.p.[1], the number of hops does not exceed $\lceil \log_{2^b} n \rceil$ plus a small constant.

[1] By "with high probability," or short "w.h.p.," we mean with probability at least $1 - \frac{1}{n}$, where n is the number of nodes in the system.

Theorem 3.5.2 *If all n nodes are uniformly distributed, then a LOOKUP terminates successfully after at most $\lceil \log_{2^b} n \rceil + o(1)$ hops w.h.p., if the routing table entries are accurate.*

PROOF. In the first step, we will show that the number of hops is upper bounded by the number of bits that are needed to uniquely identify all cliques.

Let node v in clique c be the node initiating the LOOKUP call for key s and let $\delta(c.id, s) = i$. Note that if there is no clique $\tilde{c}$ such that $\delta(\tilde{c}.id, s) < \delta(c.id, s)$, we cannot correct the i^{th} bit, thus we cannot argue that the distance measured with the metric δ decreases in each step. However, we can argue that the number of bits that still have to be considered is decreasing with each hop. Apparently, if there is a clique $\tilde{c}$ such that $\delta(\tilde{c}.id, s) < \delta(c.id, s)$, forwarding the request to this clique will fix at least the i^{th} bit and all bits in the range $[i, d]$ do not have to be considered anymore, thus the search space will be reduced by at least one bit. If we are in the situation that there is no such clique, then we know that the i^{th} bit does no longer have to be corrected, since there is no clique that could fix it. Hence, the search space is reduced by at least one bit in each step.

Similarly, for general bases b, each step reduces the search space by at least b bits, if all $(2^b - 1)\lceil \log_{2^b} n \rceil$ cliques in the routing table are still alive.

Now we will show that $\lceil \log n + 4 \rceil$ bits suffice with high probability to uniquely identify all cliques. Let the ball $B_v(\alpha)$ denote the set of nodes around node v at a distance of at most α. For all $x \in [0, 1]$, it holds that

$$\begin{aligned} p(|B_v(x\Delta)| \geq 2d) &= \sum_{i=2d}^{n} \binom{n}{i} (x^2\pi)^i (1 - x^2\pi)^{n-i} \\ &< \binom{n}{2d} (x^2\pi)^{2d} \\ &< \left(\frac{ne}{2d}\right)^{2d} (x^2\pi)^{2d}, \end{aligned}$$

where Δ denotes the diameter of the network. Let the random variable X denote the event that there is a node v such that $|B_v(x\Delta)| \geq 2d$. Setting $x := \sqrt{\frac{2(d - \ln n)}{ne\pi}}$, it follows that

$$\begin{aligned} p(X) &\leq \sum_{v \in V} p(|B_v(x\Delta)| \geq 2d) < \left(\frac{nex^2\pi}{2d}\right)^{2d} n \\ &< \left(\frac{ne2(d - \ln n)\pi}{ne\pi 2d}\right)^{2d} n < \left(1 - \frac{2\ln n}{2d}\right)^{2d} n \\ &< e^{-2\ln n} n < \frac{1}{n}. \end{aligned}$$

Thus, with high probability, there are less than $2d$ nodes at a distance of at most $x\Delta$ for each node. All cliques and consequently all IDs are spread in a two dimensional Euclidean space. Whenever a clique splits into two, both cliques occupy half of the original space and the diameter of each of those two subspaces is only a factor of $\frac{1}{\sqrt{2}}$ of the original diameter in expectation. It has to be determined how often the Euclidean space must be partitioned until the diameter of each subspace is at most $x\Delta$. If this is the case, there is no subspace

that contains $2d$ or more nodes and thus no subspace has to be divided any further, with high probability. The Euclidean space must be partitioned i times so that $\frac{1}{\sqrt{2}^i}\Delta \leq \frac{x}{\sqrt{2}}\Delta$. This holds if $i := \log n + 4$, and thus $\log n + 4$ bits suffice, since each splitting costs one bit.

Hence, with high probability, the number of hops is at most $\lceil \frac{\log n + 4}{b} \rceil$, for a specific base b.

□

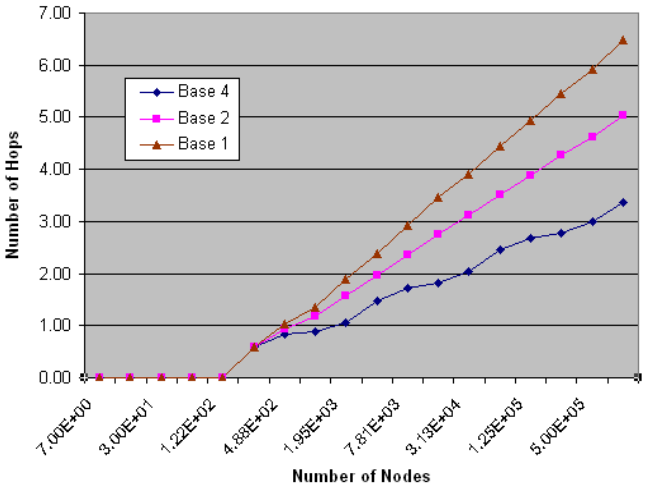

Figure 3.3: The average number of hops depending on the number of nodes in the system and the chosen base b is displayed. For each number of nodes n and for each base b, 10'000 lookups initiated at a random node and searching for a random key have been performed. The dimension d of the network is 64 in each run.

While the number of hops is already small with high probability, even fewer hops are required on average. According to simulations, the average number of hops is lower than $\lceil \log_{2^b} n \rceil$, if nodes are uniformly distributed. Figure 3.3 depicts the average number of hops required in a network of dimension $d = 64$, consisting of up to one million nodes. 10'000 lookups initiated at a random node and searching for a random key s have been performed for base $b = 1, 2$, and 4. As long as the number of nodes is lower than $2d = 128$, the entire network is a complete graph in which each node is responsible for the entire ID space, therefore the number of hops is always 0. The main reason why the average number of hops is less than $\lceil \log_{2^b} n \rceil$ is because there are only $\Theta(\frac{n}{d})$ cliques.

It is worth noting that the distribution of the number of hops required varies remarkably, depending on the base b, see Figure 3.4. While the distribution resembles a normal distribution for $b = 1$, the distribution for $b = 4$ is one-sided and a large fraction of all lookups require about the average number of hops.

In Figure 3.5, an example network consisting of 10'000 nodes is displayed. The nodes that belong to cliques whose IDs have the prefixes 00, 01, 10, and 11 are marked blue, red, yellow, and grey, respectively. The dimension d is 64 and there are 119 cliques in total. A single lookup with base $b = 1$ is depicted, in which a node in the clique with ID 0011001 contacts a node in the clique with

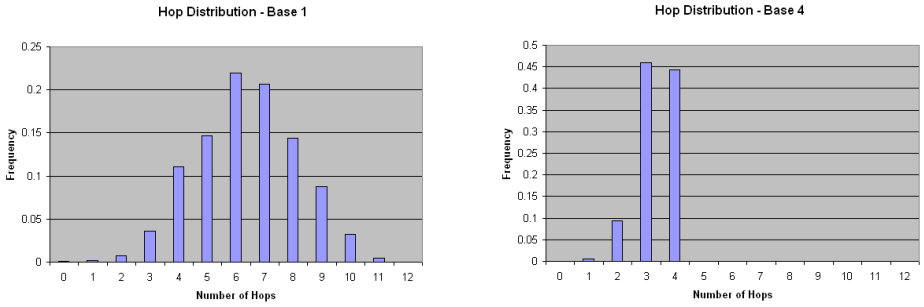

Figure 3.4: In a network of dimension $d = 64$ containing 1'000'000 nodes, 10'000 lookups initiated at a random node and searching for a random key s have been performed for base $b = 1$ and 4. The distribution of the number of hops required in those 10'000 lookups is displayed for both cases.

ID 1111100. The lookup path consists of four hops and is merely a factor of 1.472 times longer than the direct path.

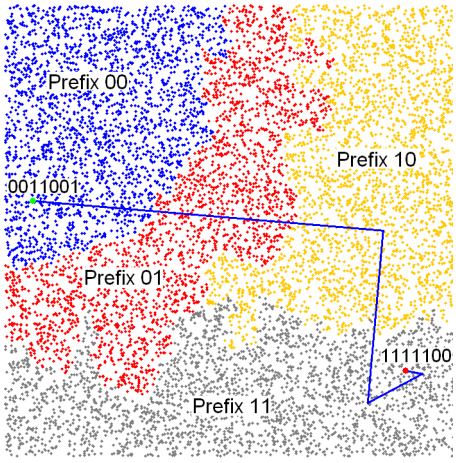

Figure 3.5: In this example network of dimension $d = 64$ and base $b = 1$, 10'000 nodes are distributed in a two dimensional Euclidean space. The nodes in the cliques whose IDs have the prefixes 00, 01, 10, and 11 are marked with different colors. A lookup path consisting of four hops is shown, starting at a node in the clique with ID 0011001 and terminating at a node in the clique with ID 1111100. The total path is only 1.472 times longer than the direct path.

Since each node has k links to all cliques in its routing table, it is very likely—this probability depends on the choice of k—that the lookup will reach the destination, because the request can be forwarded to another node, in case one node on the path does no longer respond.

So far, we have assumed that all routing entries contain correct information about the state of the network, which is—even though changes, i.e. MERGE and SPLIT operations can be performed fast and do not occur often locally—overly

optimistic.

For example, it is possible that a clique has just been split and the i^{th} bit could have been corrected directly, had the node forwarding the request already learnt about it, resulting in an additional hop. The LOOKUP algorithm, however, is still correct. More accurate lookups can be achieved by asking for an update in case the request cannot be forwarded to a clique that reduces the distance according to the metric δ. The drawback of this approach is that more messages have to be exchanged and, as a result, searching takes longer. Since this situation rarely occurs and the LOOKUP algorithm behaves correctly in each scenario, this matter is not further investigated.

Chapter 4

Fault Tolerance

Replicating data by creating cliques of nodes that all cover the same portion of the ID space ensures a certain degree of robustness by itself. Even in case of a correlated failure of $\frac{d}{2}$ nodes, there is at least one node left that can merge with the previous clique, because each clique consists of more than $\frac{d}{2}$ nodes. Ensuring that no data is ever lost is one of the main objectives of any fault-tolerant system. In the first part of this chapter, we will show that the probability of data loss is very low, even if communication in the entire network failed for a relatively long period of time.

Apart from preventing data loss, it is essential to keep up the network structure, even in the presence of churn. In order to quantify the resilience of eQuus to churn, it is desirable to derive bounds on the induced message overhead, depending on the number of JOIN and LEAVE events and the size of the network. This is the subject of the second part of this chapter.

4.1 Communication Failures

Under normal operation, each node refreshes its routing table by regularly requesting new lists of k live nodes from each clique stored in its routing table. By performing this update frequently, the probability that a considerable fraction of the nodes whose addresses are stored in the routing table are no longer alive is negligible.

In fact, the probability that any data is lost is even very low in case all communication ceases for a long period of time. Let $\lambda(p)$ be the period in which each node disappears with probability p. For instance, in a network of dimension $d = 64$ consisting of one million nodes, the probability that no data is lost, if no communication occurs in a period of $\lambda(\frac{1}{2})$ is higher than 0.99999. Increasing the number of nodes to one billion, the probability is still higher than 0.99. This holds because an entire clique has to fail before being able to merge and the probability that any clique fails is less than $p^{\frac{d}{2}}$. The total number of cliques is lower than $\frac{2n}{d}$ and thus the probability that no data is lost is higher than $(1 - p^{\frac{d}{2}})^{\frac{2n}{d}}$.

If the life time of nodes and the total number of nodes that will ever participate simultaneously can be estimated, the probability of data loss will be arbitrarily small by setting the update frequency to an appropriate value.

4.2 Dealing with Churn

eQuus effectively deals with churn, by reducing the number of topology changes that affect nodes in different cliques. Whenever a node joins or leaves the network, communication is primarily needed between members of the corresponding clique and no other routing table update is required due to this event. Changes in the routing tables of various cliques are only required if a clique either splits or merges with its predecessor. Hence, in order to evaluate the resistance to churn, it suffices to show that it takes a large number of join and leave events globally, before any clique either has to split or merge.

In the following, we consider a system in a steady state, where it is equally probable that the next global event will be either a node joining or a node leaving the system. After an initial period of growth, the number of nodes that are simultaneously in the system does not change quickly anymore, because every node joining is basically compensated by another node leaving the system. We model this behavior by setting $p(JOIN) = p(LEAVE) = \frac{1}{2}$.

Definition 4.2.1 (Stable Network) *A network is said to be in a* stable state*, if the probability that the next event is a JOIN event is equal to the probability that the next event is a LEAVE event, i.e.* $p(JOIN) = p(LEAVE) = \frac{1}{2}$.

After a clique split, the two new cliques contain d nodes each, unless the clique had to split because it had previously merged with another clique, in which case both cliques could consist of up to $\frac{5}{4}d$. The average size of a clique is evidently in the order of d. In order to simplify the analysis, an idealized form of our system in which each clique consists of exactly d nodes is considered.

Definition 4.2.2 (Balanced Network) *A network is said to be* balanced*, if all cliques have size d, where d denotes the dimension of the network.*

In a stable and balanced system, many JOIN and LEAVE events have to occur locally, before a specific clique has to either merge or split. What is more, only a small fraction of all global operations affect any specific clique, assuming that those events are uniformly distributed.

The following lemma bounds the expected number of JOIN and LEAVE events on a single clique, before this clique has to either split or merge.

Lemma 4.2.1 *In a stable network,* $\frac{d^2}{2}$ *JOIN/LEAVE events on a single clique are required in expectation, before a MERGE or SPLIT operation has to be performed.*

Proof. The clique is split if the number of nodes in it has reached $2d$, and it is merged with another clique if its size has been reduced to $\frac{d}{2}$. This can be interpreted as a random walk starting at position $\frac{d}{2}$ in the range $[0, \frac{3}{2}d]$. Let the random variable S_i denote the number of steps that are required until one of the endpoints of the specified range is reached when starting at position i. In general it holds that $E[S_i] = 1 + \frac{1}{2}(E[S_{i-1}] + E[S_{i+1}])$, thus $E[S_{i+1}] = 2(E[S_i] - 1) - E[S_{i-1}]$. By induction it follows that $E[S_i] = i(\frac{3}{2}d - i)$. This holds for $i = 0$ and $i = \frac{3}{2}d$, because $S_0 = S_{\frac{3}{2}d} = 0$. Assuming that it holds for

all values $\leq i$, the induction step works as follows.

$$\begin{aligned} E[S_{i+1}] &= 2(E[S_i]-1) - E[S_{i-1}] \\ &= 2(i(\frac{3}{2}d - i) - 1) - (i-1)(\frac{3}{2}d - (i-1)) \\ &= (i+1)(\frac{3}{2}d - (i+1)). \end{aligned}$$

This concludes the inductive proof. Setting the starting point to $\frac{d}{2}$, the expected number of events is $E[S_{\frac{d}{2}}] = \frac{d}{2}(\frac{3}{2}d - \frac{d}{2}) = \frac{d^2}{2}$, and the claim follows.

□

Let N denote the number of cliques in the network and let m denote the number of JOIN/LEAVE events. The probability that the next event occurs at any given clique is $\frac{1}{N}$, according to our uniform distribution model. It is essential to estimate the expected maximum number of JOIN and LEAVE occurring at any clique.

Lemma 4.2.2 *If the network consists of N cliques and m JOIN/LEAVE events occur, the expected maximum number of JOIN/LEAVE events on any clique is bounded by $\mathcal{O}(\frac{m}{N} + \log N)$.*

Proof. Let the random variable $X_N^i(m)$ denote the number of events occurring at clique i in a network consisting of N cliques in which m JOIN/LEAVE events occur. We want to derive a bound for $E[\max_{1\leq i\leq N} X_N^i(m)]$. It holds that

$$\begin{aligned} E[2^{\max_{1\leq i\leq N} X_N^i(m)}] &\leq E[\sum_{i=1}^{N} 2^{X_N^i(m)}] \\ &\leq N \cdot E[2^{X_N^i(m)}] \\ &\leq N \cdot \left(\sum_{j=0}^{m} 2^j \binom{m}{j} \left(\frac{1}{N}\right)^j \left(1 - \frac{1}{N}\right)^{m-j} \right) \\ &\leq N \cdot \left(\sum_{j=1}^{m} \left(\frac{me}{j}\right)^j \left(\frac{2}{N}\right)^j e^{-\frac{m-j}{N}} + 1 \right) \\ &\leq N \cdot \left(\sum_{j=1}^{m} \left(\frac{2me}{jN}\right)^j e^{-\frac{m-j}{N}} + 1 \right) \\ &\leq N \cdot \left(\sum_{j=1}^{m} e^{(\frac{2me}{jN}-1)j} e^{-\frac{m-j}{N}} + 1 \right) \\ &\leq N \cdot \left(\sum_{j=1}^{m} e^{-j+\frac{2me}{N}-\frac{m}{N}+\frac{j}{N}} + 1 \right) \\ &\leq N \cdot \left(e^{\frac{(2e-1)m}{N}} \sum_{j=0}^{m} e^{(\frac{1}{N}-1)j} \right) \\ &\leq N \cdot \left(e^{\frac{(2e-1)m}{N}} \frac{1}{1-e^{-\frac{1}{2}}} \right) \\ &= \mathcal{O}(N 2^{\mathcal{O}(\frac{m}{N})}) \end{aligned}$$

Due to the convexity of expectation it holds that $2^{E[X]} \leq E[2^X]$ for all random variables X. Hence it follows that

$$E[\max_{1 \leq i \leq N} X_N^i(m)] \leq \log(E[2^{\max_{1 \leq i \leq N} X_N^i(m)}]) = \mathcal{O}(\frac{m}{N} + \log N).$$

□

Now we are in the position to derive a lower bound for the expected number of global events before any clique has to either merge or split, depending on the number of nodes currently in the system.

Theorem 4.2.1 *If the number of nodes in a stable and balanced network is n and the nodes are uniformly distributed, then $\Omega(n \log n)$ JOIN/LEAVE events are required in expectation before either a MERGE or SPLIT operation has to be performed.*

Proof. By Lemmas 4.2.1 and 4.2.2, a MERGE or SPLIT operation has to be performed if $\frac{d^2}{2} = \mathcal{O}(\frac{m}{N} + \log N)$. Therefore, it holds that $m = \Omega(Nd^2 - N \log N)$. Due to the fact that it is a balanced network, it further holds that $n = Nd$. Since $d = \Omega(\log n)$, we get that

$$m = \Omega(n \log n - \frac{n}{d} \log \frac{n}{d}) = \Omega(n \log n).$$

□

This implies that, in this model, the system's resilience to churn increases as the network grows, since the number of cliques increases and most updates are limited to updates within the cliques.

We can now remove the constraints that the network has to be stable and balanced and give a more general theorem.

Theorem 4.2.2 *If there are n nodes in a network and the nodes are uniformly distributed, then $\Omega(n)$ JOIN/LEAVE events are required in expectation before either a MERGE or SPLIT operation has to be performed.*

Proof. In the general case, only $\Theta(d)$ events are required before any clique has to either merge or split. This holds, since the expected number of nodes in a clique is $\Theta(d)$ and thus $\Theta(d)$ nodes have to either join or leave, independent of the probabilities of those events, before a MERGE or SPLIT operation has to be performed. It follows that $m = \Omega(Nd - N \log N)$. Since $N = \Theta(\frac{n}{d})$, it holds that

$$m = \Omega(n - \frac{n}{d} \log n + \frac{n}{d} \log d) = \Omega(n).$$

□

These theorems show that, in expectation, a large number of JOIN and LEAVE events is required before any relevant topology change occurs. Consequently, in large networks, the permanent joining and leaving can be handled efficiently. Due to the rare occurrence of those relatively costly operations, it is simple for the network to update the routing tables in due time and thus maintain the desired structure.

Chapter 5

Locality

In the first part of this chapter, the locality properties of eQuus are analyzed formally in the uniform distribution model. The second part presents results of the system in the same model obtained by simulation. The goal of the formal analysis is to derive upper bounds on the expected total path lengths and the expected stretch, while the second part presents results of several simulations run in order to determine the locality properties in an emulated environment.

We assume in this chapter that the proximity metric is simply the Euclidean distance, i.e. $c(u, v) = ||u - v||_2$, on the two dimensional Euclidean plane in which all nodes lie. This assumption is obviously not an optimal approximation for large networks such as the Internet. However, if the delay is used as the proximity metric, then our assumption is reasonable, since there is a correlation between distance and delay [29].

5.1 Formal Analysis of the Locality Properties

Some other DHTs do not consider the problem of having potentially long lookup paths. In expectation, each hop incurs a delay of $\frac{\Delta}{2}$, where Δ denotes the network diameter, thus the expected path length of a path consisting of h hops is $\frac{h}{2}\Delta$. If h is large, these paths can become very long, although the destination node might be very close to the node initiating the lookup call, see Figure 5.1. The goal is to guarantee that all paths are only a small factor longer than the direct paths.

Unfortunately, our JOIN procedure does not yield a good worst-case bound on the stretch. The following lemma states that the stretch can become very large in the worst case.

Lemma 5.1.1 *If the dimension of the network is d and the base is 1, the stretch is $\Omega(2^d)$ in the worst case.*

Proof. Let $c(v, w) = \Delta$, where v and w are nodes that belong to the cliques with IDs 0^d and 10^{d-1}, respectively. Let the clique with ID 11^{d-2} be "in the middle" between the other two cliques, but slightly closer to the clique with ID 10^{d-1}. Repeating this step, cliques with IDs 1^i0^{d-i} can be positioned closer and closer to the clique with ID 0^d, see Figure 5.2.

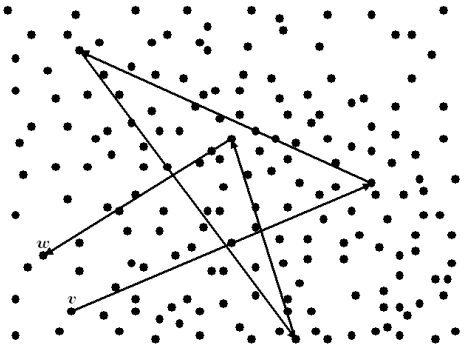

Figure 5.1: Node v initiates a lookup call. Each hop leads to a node that is far away. Thus, the total path is long, even though the target node w is close to v.

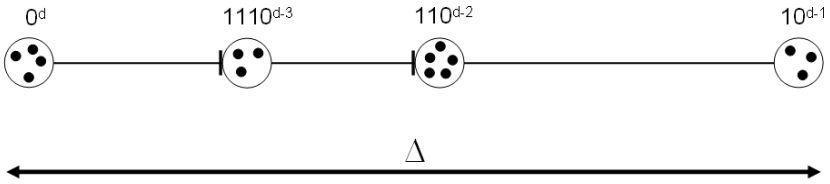

Figure 5.2: In this scenario, the stretch of a lookup path from a node in the clique with ID 0^d to a node in the clique with ID 1^d is $\Omega(2^d)$.

For arbitrarily small epsilon, the stretch of the path from a node in the clique with ID 0^d to a node in the clique with ID 1^d is

$$\lim_{\epsilon \to 0} \frac{\Delta \sum_{i=0}^{d-1} (\frac{1}{2})^i - \epsilon}{\frac{\Delta}{2^{d-1}} + \epsilon} = \frac{\Delta(2 - \frac{1}{2^{d-1}})}{\frac{\Delta}{2^{d-1}}} = 2^d - 1.$$

□

The worst-case stretch can be reduced by choosing a larger base. However, the stretch remains exponential in d in the worst case.

Lemma 5.1.2 *If the dimension of the network is d and the base is b, the stretch is $\Omega(2^{d-b})$ in the worst case.*

PROOF. We use the same technique as in the previous proof. Let $c(v, w) = \Delta$, where v and w are nodes that belong to the cliques with IDs 0^d and $1^b 0^{d-b}$, respectively. By continuously positioning a clique with ID $1^{b+j}0^{d-b-j}$ between the cliques with IDs 0^d and $1^{b+j-1}0^{d-b-j+1}$, where $j \in [1, d-b-1]$, the resulting stretch of the path from a node in the clique with ID 0^d to a node in the clique with ID 1^d is

$$\lim_{\epsilon \to 0} \frac{\Delta(2 - \frac{1}{2^{d-b}}) - \epsilon}{\frac{\Delta}{2^{d-b}} + \epsilon} = 2^{d-b+1} - 1.$$

□

Not only the stretch can be large in the worst case, but also the maximum path length. In order to show this, the notion of a *full clique* has to be introduced.

Definition 5.1.1 (Full Clique) *A clique c is said to be* full*, if it contains $2d$ nodes and it cannot split anymore, i.e. $c.id + 1 = c.successor.id$.*

The following lemma states that solely $\Omega(d^2)$ nodes have to join the network before any clique is full.

Lemma 5.1.3 *At least $d^2 + 2d$ nodes have to join the network before any clique can be full.*

PROOF. As soon as $2d$ nodes join the clique with ID 0^d, it is split into two cliques with IDs 0^d and 10^{d-1}. The clique with ID 10^{d-1} is split again if d additional nodes join this clique. Inductively, it follows that after $d + 1$ steps, the clique with ID 1^d is created. Since d nodes have to join in each step, the total number of JOIN operations is at least $(d+1)d$. The cliques with IDs $1^{d-1}0$ and 1^d, respectively contain only d nodes, thus d nodes still have to join either one of those cliques in order to turn them into full cliques. Thus, in total at least $(d + 1)d + d$ nodes have to join.

□

We can use the fact that each clique has only a limited capacity to prove that the longest path between any two nodes can be very long in the worst case. In fact, if the base is 1, there is a joining sequence of nodes such that the maximum path length between any two nodes is $d\Delta$, which is the longest possible path.

Lemma 5.1.4 *If the dimension of the network is d and the base is 1, the maximum path length is $d\Delta$ in the worst case.*

PROOF. We will construct a worst-case example. After the clique with ID 0^d is split into the cliques with IDs 0^d and 10^{d-1}, all cliques with IDs in the range $[0^d, 10^{d-1}]$ are filled. This "filler set" is denoted F_1, see Figure 5.3.

If newly arriving nodes close to the clique with ID 0^d join the network, they have to join the clique with ID 10^{d-1}, since all cliques with lower IDs are full. Thus, if the clique with ID 10^{d-1} is split, the clique with ID 110^{d-1} is far away from the clique with ID 10^{d-1} and close to the clique with ID 0^d, in contrast to the desired locality property. Repeating this step, we let a second filler set F_2 of nodes close to the clique with ID 10^{d-1} join the network. These nodes fill up the entire ID space in the range $[10^{d-1}, 110^{d-2}]$. If newly arriving nodes close to the clique with ID 10^{d-1} join the network, they have to join the clique with ID 110^{d-2}, resulting in a new clique with ID 1110^{d-3} after a split operation, which is closer to the clique with ID 10^{d-1} than to the clique with ID 110^{d-2}, leading again to a possibly maximum distance according to the proximity metric. Inductively, in each of the d steps, the distance is Δ, thus the maximum path length is $d\Delta$.

□

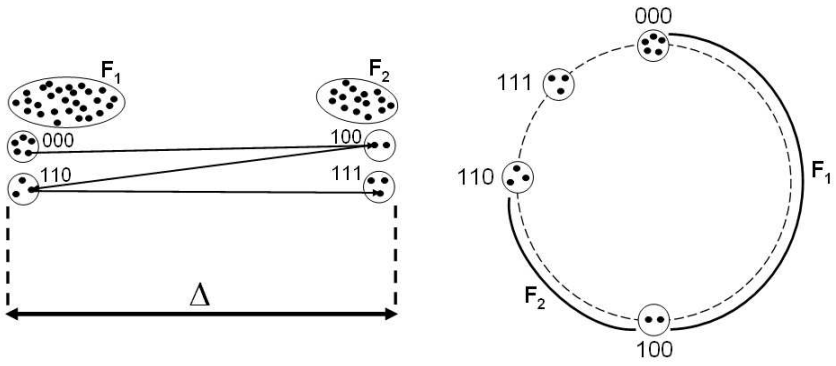

Figure 5.3: By using several "filler sets" F_i and forcing a specific joining sequence of nodes, the locality-aware joining mechanism of the system can be subverted. As a result, each hop incurs a maximum cost of Δ in the worst case.

By choosing larger filler sets, this result even holds for arbitrary bases. Note, however, that this case is not only highly improbable, but also unrealistic for a large ID space. The first filler set alone consists of $d2^d$ nodes, since each of its 2^{d-1} cliques has to consist of $2d$ nodes in order to be full. If only a relatively moderate fraction of the ID space is ever to be populated, this worst-case behavior cannot occur. Thus, it is appropriate to study the average case behavior.

Let $\Delta(\mathcal{S})$ denote the diameter of the network consisting of all nodes in the set $\mathcal{S}$. If node v in clique c performs a lookup for key s, where $\delta(c.id, s) = i$, only the subset $\mathcal{B}_v^i$ of all nodes u in any clique $\tilde{c}$ for which it holds that $\delta(\tilde{c}.id, s) \leq i$ have to be considered, due to the property of the LOOKUP procedure that the length of the shared prefix can only increase with each hop. More formally, let V be the set of all nodes in the network, then $\mathcal{B}_v^i := \{u \in V \mid u \in \tilde{c} \wedge \delta(\tilde{c}.id, s) \leq i\}$.

The following lemmas are used in order to establish our main result. The first lemma states that the expected diameter of the network consisting of all nodes in $\mathcal{B}_v^i$ is small if and only if i is small, i.e. the key s and the ID of clique c that node v belongs to share a long prefix.

Lemma 5.1.5 *Let node v in clique c be the node initiating the lookup call for key s. If $\delta(c.id, s) = i$, then it holds that $E[\Delta(\mathcal{B}_v^i)] \leq \frac{\Delta}{(\sqrt{2})^{d-i}}$.*

Proof. After a clique split into two, the area both cliques are responsible for is only half of the original area in expectation. A clique is responsible for a certain area if newly arriving nodes in this area send a JOIN message to a node in this clique. Since the area is halved in expectation, it holds that the diameter is a factor of $\sqrt{2}$ shorter in expectation. Therefore it holds that $E[\Delta(\mathcal{B}_v^{i-1})] = \frac{1}{\sqrt{2}} E[\Delta(\mathcal{B}_v^i)]$, and due to the fact that $\Delta(\mathcal{B}_v^i) \leq \Delta$ for all $v \in V$, the claim follows.

□

For the keys are random, any node in $\mathcal{B}_v^i$ has an equal chance to be the destination node of the lookup. Due to the uniform distribution of all nodes, the expected distance to the destination node is half of $\Delta(\mathcal{B}_v^i)$.

Lemma 5.1.6 *Let v in clique c be the node initiating the lookup call for key s and let u be the destination node. If $\delta(c.id, s) = i$, then the expected distance between v and u is $\frac{\Delta(\mathcal{B}_v^i)}{2}$.*

These lemmas suffice to prove the following upper bound on the expected stretch.

Theorem 5.1.1 *The expected stretch of lookup calls in eQuus is at most $\frac{2^{\frac{b}{2}+1}}{2^{\frac{b}{2}}-1}$ for a particular base b.*

PROOF. Let node v in clique c be any node initiating a lookup call for key s, let u be the destination node of the lookup and let $\delta(c.id, s) = i$. Lemma 5.1.6 states that $E[c(v, u) \mid d(c.id, s) = i] = \frac{\Delta(\mathcal{B}_v^i)}{2}$. This is true independent of the base b.

Since b bits are corrected with the first hop to node w in clique $\tilde{c}$, it holds that $\delta(c.id, \tilde{c}.id) \leq i - \min\{b, i\}$ and, according to Lemma 5.1.5, the expected diameter of $\mathcal{B}_w^{\delta(\tilde{c}.id,s)} \subseteq \mathcal{B}_v^i$ is bounded by $2^{-\frac{b}{2}}\Delta(\mathcal{B}_v^i)$.

Inductively, the total path length is therefore at most $\frac{1}{1-2^{-\frac{b}{2}}}\Delta(\mathcal{B}_v^i)$. The expected stretch is upper bounded by the ratio between the expected maximum total path length and the expected distance to the destination. Hence, it holds that

$$E[Stretch] \leq \frac{\frac{1}{1-2^{-\frac{b}{2}}}\Delta(\mathcal{B}_v^i)}{\frac{\Delta(\mathcal{B}_v^i)}{2}} = \frac{2^{\frac{b}{2}+1}}{2^{\frac{b}{2}}-1}.$$

□

Note that the expected total path length between any two nodes is at most $\frac{2^{\frac{b}{2}}}{2^{\frac{b}{2}}-1}\Delta$ in expectation, independent of the dimension d and the number of hops. Setting b to a moderately large value will incur an expected stretch of around 2. If $b = 4$, the expected stretch is already less than 3, which is already a satisfactory result. In the following section, this result is supplemented and affirmed by simulation.

5.2 Simulation of the Locality Properties

Various simulations have been run in order to study the locality properties of the system. In particular, the expected stretch has been analyzed. In Figure 5.4, the average stretch of lookups in networks of dimension $d = 64$ with up to one million nodes are depicted for the bases $b = 1, 2$ and 4.

10'000 lookups have been performed for each network size and base. Obviously, the stretch factor is 1 as long as each node has links to nodes in all other cliques. The stretch slowly increases, as the networks grows because more hops are needed in order to reach a node in the desired clique. However, the stretch only grows as long as it is below the constant expected stretch for the given base, a bound that is seemingly reached at around 1.5 for base $b = 4$. This is a much better result than the upper bound of $\frac{8}{3}$ on the expected stretch for $b = 4$ derived in the previous section.

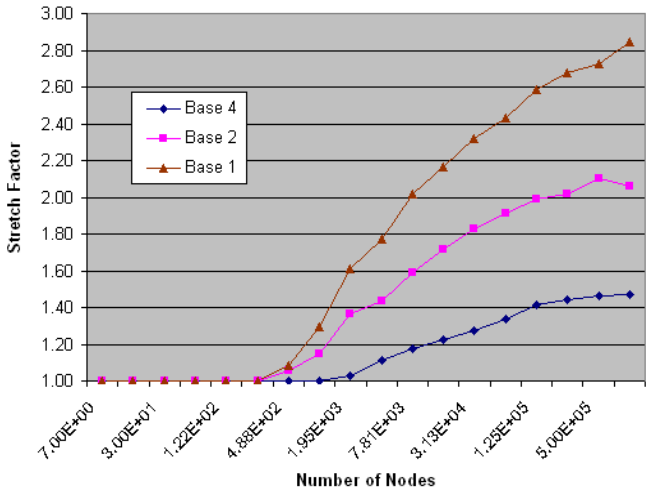

Figure 5.4: The average stretch depending on the number of nodes in the system and the chosen base b is displayed. For each number of nodes n and for each base b, 10'000 lookups initiated at a random node and searching for a random key have been performed. The dimension d of the network is 64 in each run.

In the second simulation, the effect of the dimension d on the stretch has been tested. Again 10'000 lookups have been performed for each network of size up to one million nodes for $b = 4$. The results for dimension $d = 32, 64$ and 128 are summarized in Figure 5.5. As expected, the stretch does not depend on the dimension directly. Choosing a large dimension will result in a slightly lower stretch, mainly due to the lower number of cliques in the network.

The results of these simulations show that the system has good locality properties, such as a low expected stretch and a low expected total path length. They further indicate that the analytical results are conservative, since the simulations yield better results.

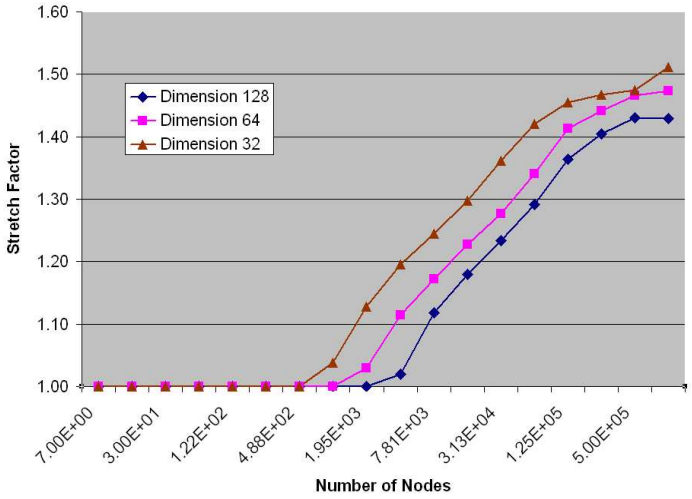

Figure 5.5: The average stretch depending on the number of nodes in the system and the dimension d of the system is displayed. For each number of nodes n and for each dimension d, 10'000 lookups initiated at a random node and searching for a random key have been performed. The base b is set to 4 in each simulation.

Chapter 6

DHT Mechanism

Data items can be published and retrieved using a limited amount of messages. A *DHT mechanism* specifies how data items are published and retrieved and also where the necessary information is stored. In this chapter, we will present a two-phase publishing and retrieval mechanism suitable for eQuus and also for many other distributed hash tables. A node v of clique c is responsible for a data item d, if the hash $h(d)$ of the data item lies in the range $[c.id, c.successor.id)$. Typically, each file shared in the system is hashed using a consistent hash function such as SHA-1, providing a fairly good load balancing. These hash functions h further guarantee that it is hard to find two data items d_1 and d_2, such that $h(d_1) = h(d_2)$. Guaranteeing a high degree of collision resistance is desirable, because each data item ought to have a unique key in the system.

The problem with this approach is that you cannot easily search specific data elements, unless you know the corresponding hash value. Each data item typically has a name. If it were possible to search using the name of the desired data item, then finding it would easier. This could be achieved by hashing the names and using these hashed names as indices in the hash table. Clearly, this approach has several drawbacks. First of all, if the name is slightly changed, then finding the data item is not possible anymore when the old name is used as the key. Second, even if a data item is found with the desired name, there is no guarantee that this is the desired data item.

The idea is to combine both approaches, i.e., use both hashed data items and hashed names in the hash table. In the next section, data items and the operations performed on those items are defined formally. Subsequently, the publishing and retrieval of data items using simple key words is presented.

6.1 Definitions of the Hashing Functions

Data items are triples (d, ν, i), where d is the actual data, ν is the name of the data item and i is meta information about the data item, e.g., the format or quality. We will use the following set of data items to illustrate both the publishing and retrieval of data items.

- $(d_1, \langle \tau_1, \tau_2 \rangle, i_1)$
- $(d_1, \langle \tau_1, \tau_3 \rangle, i_2)$

- $(d_2, \langle \tau_1, \tau_2 \rangle, i_3)$
- $(d_3, \langle \tau_3 \rangle, i_4)$
- $(d_3, \langle \tau_1 \rangle, i_5)$

Each name ν consists of a sequence of terms or words τ_i, i.e. $\nu = \langle \tau_1, \ldots . \tau_l \rangle$, where l is the number of terms in the name ν. Let $\mathcal{T}$ be the set of all term sequences and let $\gamma : \mathcal{T} \to \mathcal{T}$ be a function that given a name, i.e. a sequence of terms, as input, returns a globally ordered sequence of the same terms. Due to this global ordering, permutations of the terms do not have to be considered. Let $\mathcal{F}$ be the set of all data elements d. Furthermore, let $\mathcal{H} : \mathcal{F} \to \{0,1\}^d$ be the consistent hash function that maps data elements to bit strings of length d and let $h : \mathcal{T} \to \{0,1\}^d$ be the consistent hash function that maps names to bit strings of length d. We further generalize the hash function h and the global ordering function γ, by allowing lists of names as input. The results are simply the list of all the corresponding hash values and the list of all the corresponding ordered sequences of terms, respectively: $h : [\mathcal{T}_1, \ldots, \mathcal{T}_m] \to [h(\mathcal{T}_1), \ldots, h(\mathcal{T}_m)]$ and $\gamma : [\mathcal{T}_1, \ldots, \mathcal{T}_m] \to [\gamma(\mathcal{T}_1), \ldots, \gamma(\mathcal{T}_m)]$. Using these generalized functions, we can now easily express the essential function when it comes to indexing data items in the network. The basic form of the function is $\Psi : \mathcal{T} \to h(\gamma(2^{\mathcal{T}}))$, where 2^X denotes the power set of X. The idea is that, for each subsequence of the terms, the γ-function is applied, thus the terms are ordered in each subsequence. Subsequently, each element of this list of ordered term sequences is hashed, producing a list of $2^{|\mathcal{T}|}$ hash values. This function has to be modified, since we do not want this list of hash values to be arbitrarily large. We therefore modify the γ-function in two ways. First, in the ordered list, all terms that bear little significance are removed. For example, terms like "a" or "the" can typically be dropped without losing any information. Second, if after the removal of insignificant terms the number of terms still exceeds $\log d$, then other terms are ignored as well, in order to reduce the number of terms to $\log d$. This is not a serious restriction, since every data element can usually be described using a few terms only. How the additional terms are removed is not investigated any further. A simple heuristic would be to take the first $\log d$ terms and remove the last ones, because typically the first few terms contain the essential information about the data item. Let γ^* be this modified function. We can then modify our index function and get

$$\Psi^* : \mathcal{T} \to h(\gamma^*(2^{\mathcal{T}})).$$

Note that the resulting list of hash values has a length of at most $d - 1$, since the empty sequence is always removed.

6.2 Description of the PUBLISH Algorithm

If a node has a data item (d, ν, i) that has to be inserted in the network, the following steps are carried out.

First, the hash of the file $\mathcal{H}(d)$ is computed. The clique that is responsible for this bit string is looked up using a SEARCH call with $\mathcal{H}(d)$ as its argument. Once a list of nodes in this clique is obtained, a PUBLISH notification is sent to such a node, including the hash $\mathcal{H}(d)$, ν and i. The node that received this message verifies if it is really responsible for this data item by checking if the

hash value lies between the ID of its own clique and the ID of its successor. If this is the case, it is checked if there is already an entry for this hash value in the *data hash table*. The data hash table contains, for each hash value $\mathcal{H}(d)$, a list of the addresses of all currently active nodes in the system possessing a copy of the data item (d, ν, i) and also all meta information received for this data item. For example, assuming there is a clique that is responsible for both $\mathcal{H}(d_1)$ and $\mathcal{H}(d_3)$, its data hash table contains the following entries.

Hash	Titles	Meta Information	Holders
$\mathcal{H}(d_1)$	$\langle\tau_1, \tau_2\rangle, \langle\tau_1, \tau_3\rangle$	i_1, i_2	$a_1, \ldots, a_r$
$\mathcal{H}(d_3)$	$\langle\tau_3\rangle, \langle\tau_1\rangle$	i_4, i_5	$a_1, \ldots, a_s$

If there is no entry for this particular hash value, an entry is created. Then, it is checked if the title ν is already stored under this hash value. If it is not, the title is added, together with the meta information i. Only if this is a new title in the data hash table, for a new or an old hash value, the hash values $\Psi^*(\nu)$ of the name ν are computed and the necessary information, i.e. $\mathcal{H}(d), \nu$ and i, is forwarded to all other members of the clique, in order for them to update their data hash tables as well. Otherwise, the clique members only have to learn that a new holder of the data item d has joined the network. Subsequently, for each hash in $\Psi^*(\nu)$, an ADD_INDEX message is sent to a node in the clique responsible for the specific hash value, probably after looking it up, together with the hash $\mathcal{H}(d)$ and the name ν. Each node that receives such an ADD_INDEX message also verifies if it is truly responsible for this hash value. If this is the case, the node checks if there is a table for this hash value in its *name hash table*. This table consists of multiple tables, namely one for each hash value. For each of those tables, the hash value acts as its ID. There is an entry in such a table for each name ν that contains a subset s whose hash value $h(\gamma^*(s))$ matches the ID of the table and for each such name, the hash values of the data items bearing this particular name are stored. For example, all nodes in the clique responsible for $h(\langle\tau_1\rangle)$ contain the following table.

$h(\langle\tau_1\rangle)$

Titles	Hashes
$\langle\tau_1, \tau_2\rangle$	$\mathcal{H}(d_1), \mathcal{H}(d_2)$
$\langle\tau_1, \tau_3\rangle$	$\mathcal{H}(d_1)$
$\langle\tau_1\rangle$	$\mathcal{H}(d_3)$

Note that there are both several titles for the same data hash value and several data hash values for a particular title.

If there is no table for this particular hash value h, a new table with ID h is created. Afterwards, it is checked whether the name ν is already stored in this name hash table. If it is not stored, it is added to the table, together with the hash value $\mathcal{H}(d)$. This hash value and the title are also sent to all other nodes in the clique, thereby ensuring that all name hash tables are kept consistent. This operation concludes the publishing of a data item.

Figure 6.1 summarizes the whole publishing process. Node v publishes a data item, by sending a PUBLISH message to the clique responsible for this data item, which is clique c_1 in this case. An update message is sent to all other nodes in this clique and all other cliques that are responsible for the hashes in

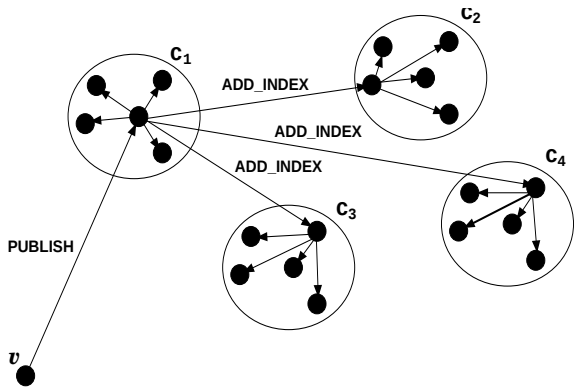

Figure 6.1: The messages sent in order to publish a data item.

$\Psi^*(\nu)$, which are cliques c_2, c_3 and c_4. Within those cliques, this information is broadcast in order to establish consistency.

6.3 Description of the RETRIEVE Algorithm

When a node wants to retrieve a certain data item, it has to choose an appropriate search string $\sigma = \langle \tau_1, \ldots, \tau_p \rangle$, in order to find the desired item.

The functions γ^* and h are applied to this search string and the clique c responsible for this hash value is looked up. A node in this clique is queried, using σ as the argument. A node receiving such a query verifies if it is indeed responsible for it, by computing $h(\gamma^*(\sigma))$ and checking if its in the range $[c.id, c.successor.id)$. After this has been verified, the node returns a list of all names $\nu_1, \ldots, \nu_t$ such that $\sigma \subseteq \nu_j$, i.e. the name ν_j contains all terms of the search string σ, for all $1 \leq j \leq t$ and all the data hashes associated with those names. However, it is more useful to have a list of all hash values and, for each hash value, a list of all the names associated with this hash value. The node answering the query can perform this transformation and return the transformed list straight away or the node issuing the query might transform the list after receiving it. Either way, the transformation has to be performed, since the issuing node is primarily interested in obtaining a list of hash values. The various names associated with a particular hash value is merely a criterion for the node to determine if it is interested in the corresponding data item. As a second and arguably more accurate and effective criterion for the quality of a data item, a node can obtain additional information of a particular data item, by inquiring all meta information available from any node in the clique responsible for the hash value of the data item. If the meta information were already stored in the name hash table, then this lookup would not be necessary. However, since the meta information is strongly connected to the hash value of the data item itself and not to the hash of any name the data item might have, it is reasonable to store this information in the data hash table. What is more, all nodes that possess a copy of the data item contact the nodes in the clique that is responsible for the hash of the data item, thus all meta information is naturally accumulated in the data hash table, which is not true for the name

hash table.

If all meta information has been received and the node is interested in acquiring a copy, it will send another request to a node in the clique responsible for the particular data item. Upon receiving such a request, the node in this clique will return the list of all addresses of the holders that are currently in the system. The issuing node can then contact any number of those nodes in order to initiate the transfer, potentially from many sources at once.

A clear advantage of obtaining a list of several or even all holders as opposed to solely querying the immediate surroundings in order to find the nearest holder according to a suitable proximity metric[1] is that it is more robust in the sense that a new request only has to be started if all nodes in the received list have disappeared. By sending ping messages, it cannot only be controlled which nodes are still alive, but it also provides a simple way to continuously select the closest source—or sources—among all live nodes.

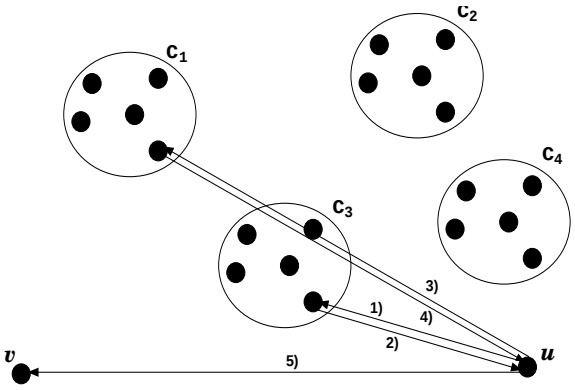

Figure 6.2: Messages sent in order to retrieve a particular data item.

Figure 6.2 depicts the process of finding nodes that possess a copy of the desired data item. In the first step, node u sends its search string σ to a node in the clique responsible for this search string, which is clique c_3 in this example. Upon receiving a list of all hashes in the second message, node u sends a message to a node in clique c_1 in order to obtain a list of all the nodes that own a copy of the data item. After receiving this list, node u can obtain a copy, e.g. by requesting a transfer from node v, which is a holder of the desired data item.

6.4 Formal Analysis of the DHT Mechanism

A pivotal concern of any publishing and retrieval mechanism of a DHT is the load balancing among all participating nodes in the system. The assignment of nodes to cliques in eQuus ensures that no node is responsible for much more data items than any other node. The following theorem states that the amount of data each node is responsible for is only a logarithmic factor larger than in the optimal case, with high probability.

[1] Other systems do so, mainly in order to ensure good locality properties such as a low stretch.

Theorem 6.4.1 *If all n nodes are uniformly distributed in a network of dimension d, each node is responsible for $\mathcal{O}(D\frac{d\log n}{n})$ data elements w.h.p.*

PROOF. The probability that a fraction ξ of the ID space is not divided by N cliques is clearly $(1-\xi)^N$. Thus the probability that a fraction 2^{-i} is not divided by $q2^i \ln N$ cliques is $(1-2^{-i})^{q2^i \ln N} < e^{-q\ln N} = N^{-q}$. By setting $i := \log(\frac{N}{q\log N})$, we get that the probability that a fraction $q\frac{\ln N}{N}$ is not divided by N cliques is less than N^{-q}. There are at most N such fractions, thus the probability that any of them is not divided is less than $N \cdot N^{-q} = N^{-q+1}$. By choosing $q \geq 3$, such that $n > (2d)^{\frac{q-1}{q-2}}$, it follows that $N^{-q+1} < \frac{1}{n}$, since $N > \frac{n}{2d}$. Because $N = \Theta(\frac{n}{d})$, it follows that each node has to store at most a fraction of $\mathcal{O}(D\frac{d\log n}{n})$ of all D data elements.

□

Guaranteeing that both the publishing and retrieval of data items can be accomplished fast is another objective of a DHT mechanism. In particular, the time and message complexities of both the PUBLISH and RETRIEVE operation are of interest. The following theorem summarizes the results for the PUBLISH operation.

Theorem 6.4.2 *If all n nodes are uniformly distributed in a network of dimension d and base b, The publishing of a data item has $\mathcal{O}(\log_{2^b} n)$ time and $\mathcal{O}(d^2)$ message complexity w.h.p.*

PROOF. First, we will proof that the message complexity is $\mathcal{O}(d^2)$. After routing to the clique responsible for the data item, which requires $\mathcal{O}(\log_{2^b} n)$ messages w.h.p. according to Theorem 3.5.2, another $\mathcal{O}(d)$ messages are required in order to broadcast this information within the clique. Apart from that, indexing messages have to be sent to at most $d-1$ cliques that are responsible for the indexing. Each of those lookups requires $\mathcal{O}(\log_{2^b} n)$ messages w.h.p. In each of those up to $d-1$ cliques, this indexing information has to be broadcast, which requires $\mathcal{O}(d)$ messages in each clique. In total, the message complexity is $\mathcal{O}(\log_{2^b} n) + \mathcal{O}(d) + (d-1)\cdot\mathcal{O}(\log_{2^b} n) + (d-1)\cdot\mathcal{O}(d) = \mathcal{O}(d^2)$. This holds due to the fact that d is much larger than $\log_{2^b} n$ for all reasonable values n. Routing to the clique that is responsible for the data item takes $\mathcal{O}(\log_{2^b} n)$ time w.h.p. Broadcasting the information within the clique can be achieved in $\mathcal{O}(1)$ time. Sending the indexing messages in parallel to all up to $d-1$ cliques that are responsible for the indexing requires $\mathcal{O}(\log_{2^b} n)$ time w.h.p. Broadcasting within those cliques requires again only constant time, thus the overall time complexity is $\mathcal{O}(\log_{2^b} n) + \mathcal{O}(1) + \mathcal{O}(\log_{2^b} n) + \mathcal{O}(1) = \mathcal{O}(\log_{2^b} n)$ w.h.p.

□

The RETRIEVE operation has even better complexities. For the sake of completion, we will state and proof the following theorem.

Theorem 6.4.3 *If all n nodes are uniformly distributed in a network of dimension d and base b, the retrieval of a data item has $\mathcal{O}(\log_{2^b} n)$ time and $\mathcal{O}(\log_{2^b} n)$ message complexity w.h.p.*

PROOF. In order to find the clique that is responsible for the hash value of the search string, $\mathcal{O}(\log_{2^b} n)$ messages are required w.h.p. This lookup also requires $\mathcal{O}(\log_{2^b} n)$ time. Once the necessary hashes are obtained, the clique responsible for the chosen hash value can be found requiring again $\mathcal{O}(\log_{2^b} n)$ messages in $\mathcal{O}(\log_{2^b} n)$ time w.h.p. At this point, the addresses of all holders can be obtained directly, thus both the time and message complexities are $\mathcal{O}(\log_{2^b} n) + \mathcal{O}(\log_{2^b} n) + \mathcal{O}(1) = \mathcal{O}(\log_{2^b} n)$ w.h.p.

□

Chapter 7

Outlook

While several essential aspects have been studied, some other important criteria still need to be considered. In the following sections, a variety of additional problems is presented. Section 7.1 deals with the issue of achieving an expedient assignment of nodes to cliques, thereby ensuring short lookup paths and a good load balancing. In Section 7.2, security problems in p2p systems are addressed. Lastly, Section 7.3 touches upon the subject of guaranteeing fair access to data items for all participating nodes. The goal is to design a scheme in which it is impossible or at least undesirable for any node to act selfishly.

7.1 ID assignment

Theorem 6.4.1 states that the load is balanced quite well among all nodes, if the nodes are uniformly distributed. The uniform distribution of nodes further ensures that, with high probability, lookup paths consist of at most $\lceil \log_{2^b} n \rceil + o(1)$ hops, according to Theorem 3.5.2.

The question is how a good ID assignment can be enforced such that the load is spread roughly equally among all nodes and the maximum number of hops remains low, even if the true node distribution does not bear any resemblance to a uniform distribution. One approach to solve this problem is to derive results for a more general class of node distributions. An often used assumption about the network density is the following. If there are κ nodes at a distance of at most r from any node v, then there are at most κ times a constant nodes within a radius of $2r$. Such a network is said to be *growth-bounded*.

Definition 7.1.1 (Γ-Growth-Bounded) *A network is Γ-growth-bounded for a constant $\Gamma > 1$, if for all nodes v and radiuses $r > 0$ such that $|B_v(r)| > 0$, it holds that $|B_v(2r)| \leq \Gamma |B_v(r)|$.*

Unfortunately, no acceptable worst-case bound can be derived in this model. On the contrary, it can be shown that a vast imbalance can be caused by a certain joining sequence of nodes, without violating the growth-bounded property.

Lemma 7.1.1 *If the minimum distance between any two nodes is 1, there is a joining sequence in a growth-bounded network such that the cliques whose IDs start with 1 are responsible for $\mathcal{O}(\Gamma^{\log \Delta})$ times more data items than the cliques whose IDs start with 0.*

PROOF. Let the first $2d$ nodes in the system be close to each other, such that after the clique split, the distance between the cliques with IDs 0^d and 10^{d-1} is around 1. The next batch of nodes joins at a distance of 2 to the clique with ID 0^d and distance 1 to the clique with ID 10^{d-1}. There can be up to Γ more cliques at distance 2. In general, approximately Γ^i cliques can join the system at a distance of 2^i to the clique with ID 0^d in the ^{ith} step. After $\log \Delta$ times, the maximum diameter is reached. All those nodes join a clique whose ID has the prefix 1, thus there are $\mathcal{O}(\Gamma^{\log \Delta})$ times more cliques whose IDs bear the prefix 1.

□

It is easy to see that there are also worst-case joining sequences in growth-bounded networks that lead to a large number of bits required to identify all cliques and thus resulting in lookup paths consisting of a potentially large number of hops. Of course, such joining sequences are improbable and will not occur in any real network. Therefore, the result does not measure the quality of the joining mechanism accurately.

Instead of trying to find likely node distributions for which it can be shown that the load is balanced with high probability, it is worthwhile to modify and extend the protocols in such a way that the desired properties hold independently of the node distribution. These load balancing mechanisms have to operate given the limited knowledge of any node in the system alone. Similar to the load balancing mechanism described in [20], in eQuus a clique can request the size of its predecessor and successor clique and use this knowledge in order to decide if some of its nodes have to migrate to one of those two cliques. Naturally, the clique with the lowest ID cannot send nodes to its predecessor clique and the clique with the highest ID cannot deliver nodes to its successor clique, since these cliques are potentially far apart. Ideally, the number of nodes in a clique is proportional to the size of the fraction of the ID space it is responsible for. This means that a clique that is responsible for twice as much data items than his predecessor clique ought to contain approximately twice as many nodes in order to increase the changes of splitting and thereby halving its fraction of the ID space.

In expectation, such a mechanism would obviously improve load balancing for any node distribution. It is, however, not easy to devise a simple, *local* load balancing scheme that provides good worst-case bounds. Moreover, this scheme has to cope with the permanent joining and leaving of nodes. This entails that the protocol has to react quickly, but without causing a large message overhead.

7.2 Security

We have shown in Section 4.1 that permanent node failures can be handled efficiently in eQuus. The system is inherently resistant to correlated failures, because there are always more than $\frac{d}{2}$ nodes in each clique. In case a much larger number of nodes fail, possibly due to a power outage, data loss cannot be avoided. This problem can again be tackled by extending the basic protocols.

The system can protect itself against correlated failures through *backup cliques*. The entire data table of any clique c is replicated on another clique $\tilde{c}$, its backup clique. The clique $\tilde{c}$ should be far away from c and it should be

clear for all other cliques which clique acts as a backup for any other clique. This can be accomplished e.g. by choosing $\tilde{c}$ such that $c.id \oplus 10^{d-1} \approx \tilde{c}$. There has to be an additional sophisticated protocol that initiates the mending of the ring structure once a certain fraction of cliques has failed. All the data tables have to be copied from the backup cliques to the cliques that are now responsible for these data items, once the ring structure has been reestablished.

Nodes that do not adhere to the protocols are another threat to the system. Such a mischievous node could pretend to belong to any clique, neglect its task to store certain information or refrain from forwarding requests etc. A node could ask other clique members whether a suspected node really belongs to this clique, thus there has to be a certain trust in the cliques. Since nodes in a clique are close-by, it is possible that an entire clique is dishonest. As in other p2p systems where potentially misbehaving nodes are ignored, the same can be done with cliques in eQuus. However, it is not clear what is supposed to happen to good nodes that belong to such evil cliques.

Another concern is the JOIN algorithm. It has to be guaranteed that a node joins the closest clique, otherwise the locality properties will be impaired. A mischievous node could pretend to have measured the distance to other cliques or fake those measurements in order to join its desired clique. It is not clear to what extend this behavior can be prevented.

Controlled attacks against the system pose another major security risk for any p2p system. A simple approach to circumvent a Sybil Attack [6] would be to limit the number of nodes in the system with the same IP address. In order to launch an attack, many computers would be needed in this case. The number of nodes in the system that share the same IP address can be found by hashing the IP address using a consistent hash function and asking the clique that is responsible for this bit string. Thus, cliques would further have to store information about the IP addresses of nodes in their data tables. There are many other attacks that can be launched against a p2p system. Specific algorithms have to be designed that allow for both a quick detection of such attacks and an effective recovery from them, before the system becomes unable to function.

7.3 Fairness

In p2p systems, nodes downloading data items from other nodes but without offering any data items themselves are often referred to as *free riders*. Free riding is a prevalent, undesirable phenomena in today's p2p networks. Studies have shown that a small fraction of peers in the Gnutella network [10] actually provide data items, while most peers are only consumers [3, 24]. Several proposals on how this problem can be tackled and a certain kind of "fair trading" can be established have been published in the last few years.

One particular approach is to introduce a currency in the system. In the economic framework for p2p resource sharing called KARMA [26], each download costs a certain fee which the node downloading this data item or chunk of a data item has to pay. By means of digital signatures, it is verified that the node has actually uploaded the chunk and that it got paid by the node that downloaded it. A set of nodes, denoted the *bank-set*, is responsible for the transactions of a certain node. This approach has several drawbacks. Since it has to be proved that the transactions were successful and that the accounts of

the nodes involved have been updated correctly, the message overhead is quite large. Moreover, the system suffers from inflation and deflation, because nodes are joining and leaving quickly, thereby adding to or subtracting from the total amount of credits in the system. A global operation is required to compensate for this effect.

Another intuitive approach is the use of reputations. In EigenRep [12], each node has a reputation depending on how all other nodes rate it. Among those other nodes, the ratings of those nodes that have a good reputation themselves outweigh the rating of the nodes that have a low rating. While this approach is much simpler as it does not require complex transactions, it also has some considerable drawbacks. In order to compute each node's reputation, the left principal eigenvector of the matrix consisting of the ratings of each node about all other nodes has to be computed in a distributed and iterative fashion. This operation has to be performed quickly, because it ought to be repeated regularly in order to include new nodes in the computations and avoid outdated reputations in the system.

The question is whether there are simpler methods to reduce free riding without elaborate transactions and global computation-heavy operations. One idea is to continuously reward the nodes that have uploaded the most. In a certain period of time, each node selects one node that has uploaded the most and sends a reward notification to the clique that is responsible for the gratification of this node, e.g. the clique that is responsible for a hash of the node's IP address. A node uploads to nodes that have recently received the most rewards among all nodes requesting data items from this node. An advantage of this approach is that nodes cannot harm one another. They can merely benefit from uploading and thereby collecting rewards. The drawback of this simple method is that colluding nodes might reward each other without providing anything to the system in reality. It is desirable to find a simple and effective scheme that reduces the occurrence of free riding remarkably without providing a basis for collusion.

Chapter 8

Conclusion

We presented a novel p2p system, called eQuus, that inherently exhibits a strong resilience to churn and peer failures. Moreover, the maintenance overhead is relatively low. Most communication triggered by the maintenance protocols is local in nature, thus maintenance operations can be performed quickly and not much wide-area traffic is induced. Each peer stores $\mathcal{O}(\log n)$ links and keyed lookup requires $\mathcal{O}(\log n)$ messages. eQuus further has good locality properties, such as a low expected stretch. It is therefore a suitable network overlay for large and highly dynamic networks.

While important factors such as fault tolerance and locality are considered, the system can still be augmented in various ways. When considering deliberate attacks on the network, the appearance or disappearance of a large number of peers in a short time span becomes realistic and appropriate actions need to be taken. While attacks can be launched against any p2p system, other security threats mainly pertain to eQuus, e.g. it is easier to spoof the own ID by claiming to belong to a different clique, since IDs in eQuus are not associated with IP addresses in a fraud resistant manner.

A comparison between several p2p systems as far as the message overhead is concerned—in an environment with high and low dynamics—would give additional insights. These are interesting and challenging directions for future research.

Bibliography

[1] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, and S. Ron. Practical Locality-Awareness for Large Scale Information Sharing. In *Proc. 4th Int. Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[2] I. Abraham, D. Malkhi, and O. Dobzinski. LAND: Stretch $(1 + \varepsilon)$ Locality-Aware Networks for DHTs. In *Proc. 15th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 550–559, 2004.

[3] E. Adar and B. A. Huberman. Free Riding on Gnutella. *First Monday*, 2000.

[4] J. Aspnes and G. Shah. Skip Graphs. In *Proc. 14th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 384–393, 2003.

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

[6] J. R. Douceur. The sybil attack. In *Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260, 2002.

[7] eDonkey. www.edonkey2000.com.

[8] eMule. www.emule-project.org.

[9] A. Fiat and J. Saia. Censorship Resistant Peer-to-Peer Content Addressable Networks. In *Proc. 13th Symp. on Discrete Algorithms (SODA)*, 2002.

[10] Gnutella. www.gnutella.com.

[11] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. 4th USENIX Symp. on Internet Technologies and Systems (USITS)*, 2003.

[12] S. Kamvar, M. Schlosser, and H. Garcia-Molina. Eigenrep: Reputation Management in P2P Networks. In *Proc. 12th International World Wide Web Conference (WWW 2003)*, 2003.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proc. of ACM ASPLOS*, 2000.

[14] F. Kuhn, S. Schmid, and R. Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *Proc. 4th Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[15] X. Li, J. Misra, and C. G. Plaxton. Active and Concurrent Topology Maintenance. In *Proc. 18th Ann. Conference on Distributed Computing (DISC)*, 2004.

[16] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. 21st Ann. Symp. on Principles of Distributed Computing (PODC)*, pages 183–192, 2002.

[17] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[18] Napster. www.napster.com.

[19] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proc. 9th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, 1997.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proc. of ACM SIGCOMM 2001*, 2001.

[21] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. USENIX Ann. Technical Conference*, 2004.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. 18th IFIP/ACM Int. Conference on Distributed Systems Platforms (MiddlewareS)*, 2001.

[23] J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu. Dynamically Fault-Tolerant Content Addressable Networks. In *Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[24] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement study of Peer-to-Peer File Sharing Systems. In *Proc. SPIE Multimedia Computing and Networking (MMCN2002)*, 2002.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM Conference*, 2001.

[26] V. Vishnumurthy, S. Chandrakumar, and E. Sirer. KARMA: A Secure Economic Framework for Peer-to-Peer Resource Sharing. In *Proc. 1st Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, 2003.

[27] M. Waldvogel and R. Rinaldi. Efficient Topology-Aware Overlay Network. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, NJ, USA, 2002.

[28] B. Y. Zhao, L. Huang, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.

[29] A. Ziviani, S. Fdida, J. F. de Rezende, and O. C. M. B. Duarte. Toward a measurement-based geographic location service. In *Proc. of the Passive and Active Measurement Workshop (PAM)*, Lecture Notes in Computer Science (LNCS) 3015, pages 43–52, Antibes Juan-les-Pins, France, 2004.