An overview of the Distributed K-ary System

Konstantin Welke

March 16, 2007

Seminar paper for
Peer-to-Peer Networks (WS 2006/07)
Arne Vater and Prof. Schindelhauer
Professorship for Computer Networks and Telematik
Department of Computer Science
University of Freiburg

Abstract

This seminar paper describes the the Distributed K-Ary System (DKS). DKS is a robust and configurable peer-to-peer overlay network. It has 3 parameters: N, k and f. N is the maximum number of nodes in a network, each node can be reached within $(log_k N) + 1$ hops and the routing table in each peer has the size $k * log_k N$. Furthermore, $f - 1$ nodes can fail simultaneously without damaging the network.

One key property is that there is no seperate mechanism to maintain routing tables: during normal operation, all errorneous routing table entries are corrected on the fly by a mechanism called correction-on-use.

Contents

List of Figures

1 Introduction

1.1 Motivation

Many people first think of Napster or Bittorrent when hearing the "peer-to-peer network". While there is much research on the problem of distributing data to many users, it is by far not the only research on peer-to-peer networks. One big subject are Distributed Hash Tables (DHT), which is a distributed system that can store and retrieve key/value-pairs. There is a big number of approaches for efficient design of DHTs. The first academic papers on DHT appeared in 2001 and described systems such as CAN [8], Chord [10], Pastry [9] and Tapestry [11]. Of course, research did not stop there. One approach for the formalization of certain methods is the Distributed K-Ary System (DKS) [2]. DKS claims to more general approach to the problem of Distributed Hash Table construction, so that other existing networks can be seen as instantiations of a DKS[1]. These most notably include Chord, Pastry and Kademlia [7]. This implies a certain similarity in the structure and algorithms used in the different approaches. The basic idea in this formalization is point out common problems and find efficient approaches that can be applied to all networks that can be seen as a DKS. While it would go beyond the scope of this seminar paper to discuss whether the different DHT concepts really are instantiations of DKS, DKS itself seems to be an interesting subject. This is why this seminar paper describes and discusses the data structures and algorithms of DKS.

1.2 Literature and Sources

The main paper that describes the Distributed K-Ary System is [2]. This document is not a technical specification but focuses on the scientifically interesting aspects of the protocol, while leaving out some details. It covers the general idea and concepts of DKS, the network structure and how a peer joins and leaves the network. It also covers how routing, hash table lookups, error correction and recovery from node failure are handled. Finally, it describes 2 conducted experiments, in which simulated DKS networks are compared to the Chord system. As DKS sees itself as an extension of, among others, the Chord system [10], it is hard to understand this paper without prior knowledge about Chord.

In order to compensate for node failures, every key/value-pair is stored in the network multiple times. [2] briefly describes a mechanism called *successor-list* in order to realize this replication, just as the Chord system does. However, the paper [5] discusses various existing replication mechanisms and introduces *symmetric replication.* Symmetric replication is used as the new replication mechanism for DKS.

The correction-on-use mechanism of DKS is discussed in further detail in [4].

In his PhD thesis [6], Ali Ghodsi presents the concepts and algorithms for DKS in very much detail. He begins with a short description of the Chord system and then discusses different algorithmic approaches for this system. This gradually leads to DKS. This thesis is very good in explaining the ideas and concepts used in DKS, and the reasons why they are used. Also, the author gives proof to many important properties such as liveliness and deadlock-freeness of

[1][3], p. 27ff

their locking system. Many times, the naive version of an algorithm is first presented and then refined in order to give a final algorithm. This approach of presenting the algorithms creates a good understanding; however, it is sometimes hard to tell whether a presented version of an algorithm is the final version or if another refinement is presented later in the thesis, as this is not made very explicit by the author.

[3] also describes DKS, but focuses on comparing different approaches to distributed hash tables.

The source code for the Java implementation of DKS [1] is available at the DKS website. There, one can inspect the implementations of the concepts and algorithms, or try DKS out for oneself.

2 Network structure

The Distributed K-Ary System is an peer-to-peer overlay network. It provides an infrastructure for distributed storing and retrieving of key/value-pairs, a distributed hash table. Furthermore, is allows for efficient broadcast and multicast message passing between nodes. One of the main ideas behind DKS is the following observation:

> "In P2P systems in which at any time, the number of lookups and key/value (or document) insertions is significantly higher than the number of joins, leaves and failures, the cost incurred by active correction is unnecessary." [2], page 1

This is why in DKS all there is never any pure maintenance traffic to correct the network state: all communication required to update the state of the network is piggybacked to actual user traffic. This is called *correction-on-use*.

While DKS aims to be very efficient, some aspects cannot be globally optimized but involve some sort of trade-off. This is why DKS introduces 3 parameters that specify some of its properties and can be adjusted to suit ones needs: N, k and f. These parameters will be used throughout this paper.

N is the maximum number of nodes that can be part of a DKS. The reason behind this is that every node is assigned a unique identifier upon joining the network, and the network is organized in a ring-structure, where the node identifier increases in clockwise direction. Thus, a highest node number is needed in order to close the ring from the highest node identifier to the lowest.

k is the search-arity of the network. Searching for a value in the network can be modeled as a k-ary tree with the node that initiates the search as the root node. The higher the value of k, the less messages are needed to perform a search, but the bigger are the routing tables. k should be chosen to fulfill the formula $N = k^L$, $L \in \mathbb{N}$, where L is called the number of levels in the search, or the depth of the search tree.

f is the fault-tolerance of the network. In any given DKS, all information is stored f times in the network. Thus, $f - 1$ nodes may fail simultaneously, which means they can leave the network without prior notification, and the network will recover gracefully without loosing data. However, this behavior is not advised. This is a pure safety mechanism in order to recover from link failures and node crashes. f should be chosen to be a divisor of N.

2.1 Definitions

The following definitions are needed to facilitate the discussion of DKS. All definitions concerning DKS are taken from [6]. This seminar paper assumes the reader is familiar with the Internet Protocol and does not explain things such as TCP/IP.

2.1.1 Overlay network

An *overlay network* is a network built on top of one or multiple underlying networks, realizing a new addressing scheme. The underlying network is called *underlay network*. In order for one node a of the overlay network to address another node b, node a either must know node b's underlay network address or know another node c of the network who knows node b's underlay network address. This can be applied recursively, so that node a only needs to know a node c who believes to be able to reach node b. The total number of nodes that were needed to transfer the message from a to b (including a, but excluding b) is called *hop count* or *distance* or *route length* from a to b. The total number of other nodes in the overlay network to which any node can address directly is called *degree* or *routing table size*. It is a desired property for an overlay network to have both a low *maximum distance* and *degree* with respect to the total or maximum number of nodes in the network. There usually is a trade off between *maximum distance* and *degree*.

In this seminar paper, I will use the terms “DKS”, “overlay network” or simply “the network” interchangeably.

2.1.2 Distributed Hash Table

A hash table is a data structure that stores *values* under a given *key*, so that the corresponding value to a given key can efficiently be retrieved. While a hash table is usually stored in one computer and has average insertion and lookup efficiency of O(1), a *Distributed Hash table (DHT)* is redundantly stored in multiple computers and has worse lookup and insertion costs. A DHT is usually realized in an overlay network, where the *degree* corresponds to overlay network maintenance cost as nodes join and leave, while the *maximum distance* corresponds to lookup and insertion cost.

A DHT maps key/value pairs to nodes in an overlay network. Each key/value-pair is thus assigned the identifier of the node it is associated with. If a node is said to store identifier i, this means that the node stores all key/value-pairs that are mapped to identifier i.

2.1.3 Modulo Arithmetic

DKS is organized in a ring structure, each node has a unique integer identifier n in the range of $0 \leq n < N$. Arithmetic calculations are performed on these integers. The result of such calculations should be a valid identifier. This is why modulo arithmetics are performed. In order to point this out, the following notation is used:

- $a \oplus b = (a + b)\, modulo\, N$
- $a \ominus b = (a - b)\, modulo\, N$

2.1.4 Distance

The *distance between two identifiers* is defined as the difference of two identifiers in clockwise direction:

- $d(a, b) \;=\; b \ominus a$

2.1.5 Successor and Predecessor

As a DKS can have fewer than N nodes. In other words, not every identifier corresponds to an actual node present in the System. This is why the set $\mathcal{P}$ is introduced, which consists of all nodes that are actually connected to the DKS.

The *successor S(x)* of an identifier x is defined as the first connected node in clockwise direction, starting at node x:

- $S(x) = x \oplus min\{d(x, y) \,|\, y \in \mathcal{P}\}$

However, the *next successor* of a node x is defined as the first connected node in clockwise direction, starting after node x:

- $succ = S(x \oplus 1)$

This is also called *successor pointer* or simply *succ*.

Similarly, the *predecessor pred* of a node x is defined as the first connected node in counter-clockwise direction:

- $pred = x \oplus max\{d(x, y) \,|\, y \in \mathcal{P}\}$

Please note that what I call *next successor* is simply called *successor* in the existing literature. However, I found it sometimes confusing to have one name for two different things. Example given, each node has a successor pointer to its next existing successor (in the sense of *next successor*). Each node is also its own successor (in the sense of *successor*).

2.1.6 Responsible node

A node n is *responsible* for an identifier i if and only if it is the *successor* of that identifier:

- $S(i) = n$

The responsible node for identifier i stores all key/value-pairs which are mapped to identifier i.

2.1.7 Interval

Furthermore, DKS has the notion of *intervals*. An interval [n, m[consists of n and all following nodes in clockwise direction, up to and excluding m:

- $[n, m[= \{x \,|\, d(n, x) < d(n, m)\}$

[n, m[can also be written as [n, m).

The successor of an interval is the first node that exists in that interval, or *nil* if no nodes exist in that interval.

2.2 Basic structure

As previously mentioned, DKS is organized in a ring-structure, where each node is assigned an unique identifier n, with $0 \leq n < N$. More often that not, the network is not fully populated. Each node stores a route to its existing *predecessor pred* and *next successor succ*. The DKS illustrated in figure 1, is such an example. Each gray circle represents an actual node present in the DKS, while an empty, white node represents an identifier for which no actual node exists. Node 0 has a route to its predecessor 15 and to its successor 3.

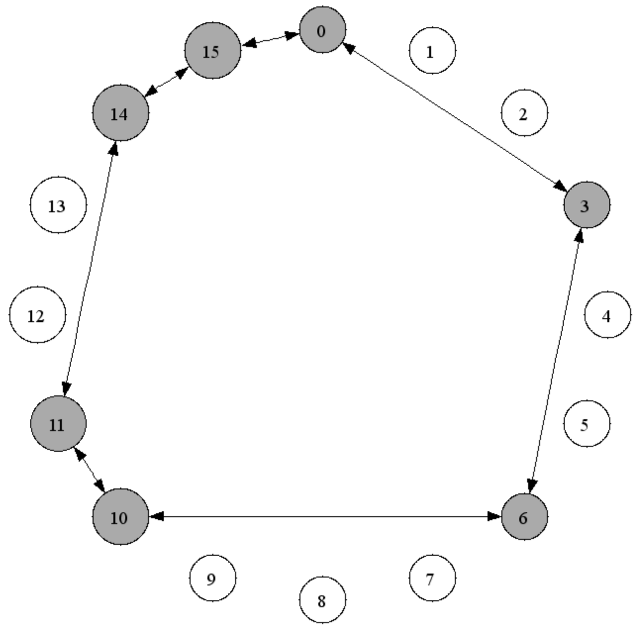

Figure 1: A small DKS with N=16.

Each node n in a DKS divides its view of the network into L levels which consist of k intervals each. At the first level, the network is split into k intervals $[n, n \oplus \frac{N}{k}[$, $[n \oplus \frac{N}{k}, n \oplus 2 * \frac{N}{k}[$, ... $[n \oplus (k-1) * \frac{N}{k}, n[$. This means that the whole ring is divided into k equal parts. Higher levels l always divide the first interval of their previous level $l-1$ into k parts, until at level L, the first interval of level $L-1$ is split into the intervals $[n, n \oplus 1[$, $[n \oplus 1, n \oplus 2[$, ... $[n \oplus (k-1)$, $n \oplus k[$. Thus, each interval at level l consists of $\frac{N}{k^l}$nodes.

To each interval I, a responsible node R is assigned, which is S(I), according to the node's best knowledge. While it it better to choose the actual S(I), it is legal to point to any node in I.

Note that this means that a node is always responsible for the its first interval of any given level.

Also note that each node has a different view about intervals and levels. Figure 2 shows the levels and intervals for node 0. For node 1, everything should be rotated by one node, for node 16 everything should be rotated by 90 degrees, etc.

Each node maintains, in addition to routes to its predecessor and successor, a routing table which consists of the $L * k$ entries. For each level, there are k entries which point to the responsible nodes of the k intervals of that level. Thus, the total size of the routing table is logarithmic to the maximum number of nodes, as k is an arbitrary chosen constant and $L = log_k N$.

All routing table entries form the set RT.

Furthermore, each node keeps a backlist BL of nodes that point to it. This backlist will be presented in detail in section 4.2 on page 21.

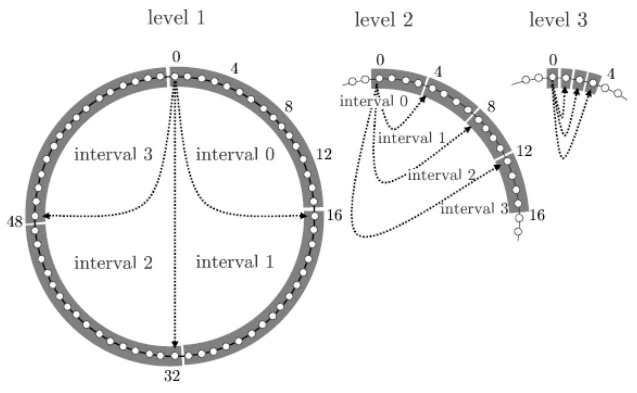

Figure 2: Interval view of node 0 for levels 1, 2 and 3 for a DKS with N=64, k=4. Figure taken from [6], page 97.

As having a route to a valid next successor is crucial for the correct functioning of the network, each node maintains a *successor list*, which consists of the f *next successors* of n. In figure 1, the successor list of node 6 would consist of node 10, 11 and 14 for $f = 3$. Similarly, a *predecessor list* with node n's f *predecessors* is maintained.

Please note that while [4] discusses topology maintenance in detail, it does not mention a backlist *BL*. Instead *successor list* and *predecessor list* are called *FrontList* and *BackList*. Meanwhile, [6] does not mention a *predecessor list*.

2.2.1 Connections

Every node keeps active TCP connections to each peer in its routing table *RT*. The TCP protocol is used because of its reliability and FIFO properties. This makes sure that messages are sent successfully in chronological order as long as the connection is alive.

[6] mentions that sometimes, a direct connection to a node cannot be established, as he could be behind a firewall or network address translation. However, no solution to this problem is proposed.

2.3 Addressing nodes using lookup

A node p in general cannot communicate with any other node q, as it needs to know q's underlay network address. Furthermore, if node q does not exist, it needs to address it q's *successor* S(q) instead. If q exists, q = S(q), so the general approach is to search for S(q).

In order to find out S(q), node p performs a *lookup*, which sends a message the node r, to which it has a route and which has the closest distance to S(q). $r = \{min(d(x, q)) \,|\, x \in RT\}$.

Node r then passes on this message using the same scheme, until it eventually reaches S(q). Note that a node n is S(q) if q is between n's predecessor and n: $n = S(q) \Leftrightarrow n \leq q < n.pred$, so every node can easily check if it is S(q). If the message finally reaches S(q), S(q) can directly respond to p, as p's contact information was piggybacked in the message.

As the distance is at least divided by k with each message, and the maximum distance is $N - 1$, every node can be reached with $log_k N$ hops, if all routing information up-to-date and no failures occur.

2.4 Overshooting

Sometimes, it might be tempting to send a message to the believed successor of a node in order to minimize the number of messages being sent. If this actually increases the distance to the destination node, it is called *overshooting*. DKS forbids overshooting because it violates the property that the distance to the destination node decreases with each hop. In a worst-case scenario, this could greatly increase the total number of messages passed.

A typical case of overshooting is when a node a wants to send a message to a different node S(b), which it assumes to be node c to which it has a route; and where $c \neq a$ and $c \neq b$. An optimistic assumption would be that the view of node a is correct, and the total number of messages is reduced by sending directly to c. However, if $c \neq S(b)$, c would need to send the message to S(b), to which is typically has a very big distance (in clockwise direction), as node a assumed S(b) = c. So instead of reducing the distance to node S(b), the distance was increased to a value close to the maximum. As this could happen multiple times during a lookup, the number of messages used to reach S(b) could vastly increase. This is why overshooting is strictly forbidden.

As a node is not allowed to overshoot, node a can only send a message directly to S(b) if S(b) is node a's next successor or if S(b) = b. In the former case, node a exactly knows S(b), as $S(b) = S(a \oplus 1)$, and in the latter case a cannot overshoot, as it sends the message to b, and the distance d(b,b) = 0, so it did not increase.

Note that this can increase the number of messages used to reach a node by 1: Consider a scenario where a node d would be reached in L hops if it existed. If it existed, the node that would send a message to it in the last hop is node n. If node d is not node n's successor, node n cannot be sure whether node d does exist, but it cannot be in the routing table of node n, as it does not exist. Thus, node d can only send the message to d's predecessor p, which is in d's routing table. Node p can then send the message to d. Thus, one more hope was introduced, raising the message complexity to $(log_k N) + 1$, if all routing tables are up-to-date and no failures occur.

2.5 Correction-on-use

Every node sends lookup messages only to nodes in its routing table RT. In order to check whether the entry in the routing table is correct, the information about the level and interval of the receiving node piggy-backed in the message. This way, the recipient can determine if the senders routing information is up-to-date, using only local information. It can calculate the interval from the senders perspective and determine if it is the first node in this interval, using its information about its *predecessor*. If it is not the first node in the interval, it tells the sender the address of the first node in the interval it knows about, using RT and its backlist.

Note that this mechanism is all that is needed in order to correct erroneous routing entries, as long as the initial observation holds that there is significantly more user traffic than nodes joining, leaving or failing. This means that no additional mechanisms, such as periodic stabilization like in the Chord system, are needed.

3 Applications

3.1 Distributed Hash Table

DKS can be used to realize one or more DHTs. In each lookup or insertion operation, the requesting node simply specifies which DHT he relates to. In order to store a key/value-pair in DKS, a node n first calculates the hash of the key $H(key)$ using a predefined hash function H which is known to all nodes. $H(key)$ returns a integer with $0 \leq H(key) < N$. As node $S(H(key))$ is responsible for storing this key/value-pair, node n sends the key/value to $S(H(key))$. The same process is used to retrieve key/value-pairs from nodes.

Note that if a node n joins, identifier responsibilities might switch to n. In this case, all key/value-pairs that n is responsible for are transferred to n. Similarly, if node n wants to leave, all its key/value-pairs are first transferred to the new responsible nodes.

The store and retrieve operations for DHT are be slightly modified to handle node failures. See section 4.4 on page 24 for details.

3.1.1 Usage example

In this example, node 0 first wants to retrieve the value of the key “foo”.

The DKS parameters are N=16, k=2, f=1. As the “k-arity” of the network is 2, the network has $log_2 16 = 4$ levels. node 0 divides the network into 2 intervals at level 1: interval 0 is [0, 8[and interval 1 is [8, 0[. Thus, responsible nodes for intervals 0 and 1 are node S([0, 8[), which is 0 and node S([8, 0[), which is 10, respectively. At level 2, the first interval of level 1 is further divided into 2 intervals: [0, 4[and [4, 8[. The responsible nodes are S([0, 4[) = 0 and S([4, 8[) = 6. At level 3, the previous level’s first interval is divided into 2 new intervals: [0, 2[and [2, 4[. The responsible nodes are 0 and 3. As level 4 is the last level, the intervals merely consist of one node: [0, 1[and [1, 2[. S([0, 1[) is, of course, 0; S([1, 2[) = *nil*, as no node is actually present in the interval. Note that if we allowed successors outside the interval to take the role of the responsible node, this would not introduce routes to new nodes: For each interval [a, b[, follows another interval [b, c[. So if we defined S([a, b[) as S(a), S([a, b[) would return a node n outside of [a, b[if [a, b[is empty. However, as n is the first node that exists in the network after a, it also is a first node of another interval that follows [a, b[. Thus, it is already in the routing table *RT*. In our example, S([1, 2[) is nil, S(1) is 3. We could add node 3 to node 0’s routing table for level 4, interval 1, but node 3 is already the responsible node for level 3, interval 1, so this would not allow to address any new nodes.

Node 0 wants to retrieve the value stored under key “foo”. H(“foo”) is 13, so node 13’s successor S(13) is responsible for “foo”. Node 13 is not in node 0’s routing table, nor in its backlist, so node 0 does not know whether node 13 is actually present in the network. In order establish a connection to node 13, it needs to retrieve its underlay network address information by performing a *lookup*. Node 0 embeds the request to retrieve the value of “foo” in the lookup in order to minimize the amount of messages sent. The node with the smallest distance to node 13 in node 0’s routing table is node 10, so the lookup message is sent to node 10. In the lookup message, it embeds that it considers node 10 to be responsible for level 1, interval 1. As node 10’s *predecessor* points outside

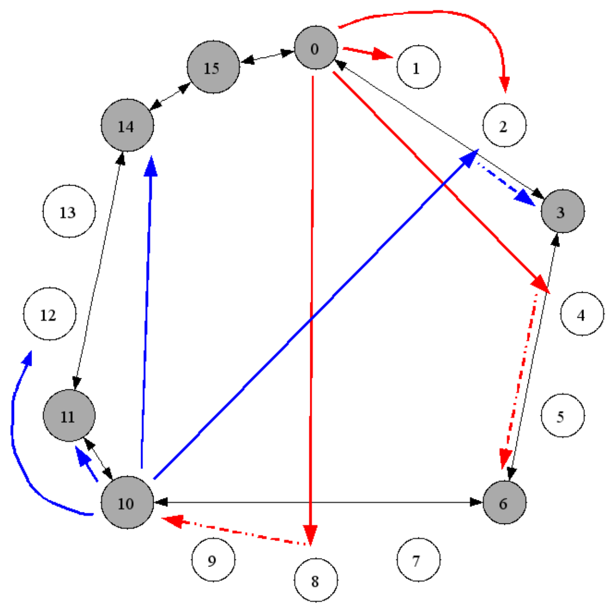

Figure 3: A DKS with N=16, k=2, f=1. Routing table entries for nodes 0 and 10. The continuous arrows indicate the start of the interval that is pointed to, while the dotted arrows indicate the successor of that interval.

Table 1: Node 0's view on the network

Node	Level no	Interval no	Interval	Responsible node
0	1	0	[0, 8[	0
0	1	1	[8, 0[	10
0	2	0	[0, 4[	3
0	2	1	[4, 8[	6
0	3	0	[0, 2[	0
0	3	1	[2, 4[	3
0	4	0	[0, 1[	0
0	4	1	[1, 2[	nil

Table 2: Node 10's view on the network

Node	Level no	Interval no	Interval	Responsible node
10	1	0	[10, 2[	10
10	1	1	[2, 10[	3
10	2	0	[10, 14[	10
10	2	1	[14, 2[	14
10	3	0	[10, 12[	10
10	3	1	[12, 14[	nil
10	4	0	[10, 11[	10
10	4	1	[11, 12[	11

the interval [8, 0[, node 10 knows that this information is correct. Node 10 then forwards this message to the node closest to node 13 in its routing table, which is node 11. Again, it piggybacks the information that node 10 considers node 11 to be the responsible node for level 4, interval 1. Node 11 finally knows that node 13 is not present in the network, as its *next successor* is node 14. Thus, node 14 knows that S(13) = 14 and sends back the lookup message to node 14, embedding that node 11 thinks that node 14 is its successor. Node 14 then processes the request and sends the value of "foo" to node 0. In total, 4 messages where used, which is smaller than the expected maximum $5 = (log_k N) + 1$.

Note that node 10 could argue that if node 13 existed, it should have a route to it for level 3, interval 1. As the route in this interval is *nil*, it is very likely that node 13 does not exist. As node 10 has a route to node 14, it could send the message directly to node 14, which should be S(13). However, node 10 cannot be sure that its routing table information is correct, so it does not know whether S(13) = 14 or not. Thus, sending directly to node 14 would be *overshooting*, as it increases the distance to the destination: d(10, 13) = 3, while d(14,13) = 15. This is why node 10 forwards the lookup to node 11 instead, thus reducing the distance.

3.2 Broadcast and Multicast

When a node n wants to send a message to all nodes in the DKS, it sends broadcast messages to all nodes it considers responsible for the intervals of level 1, including itself. Each node then forwards this message to all nodes of the next level, including itself, until level L is reached. An illustration of this can be seen on figure 4. In a network consisting of n nodes, this reaches all nodes with n messages, where each node sends at most $log_k N$ messages. This is an efficient broadcast algorithm.

Multicast groups form small DKS of their own. In order to multicast in a group, a node simply broadcasts in the DKS that the multicast group forms. In order to join a multicast group, the joining node must know at least one node in the multicast DKS. This is why all nodes of a multicast group are additionally stored in the hash table TOPDHT under the key of the multicast group's name.

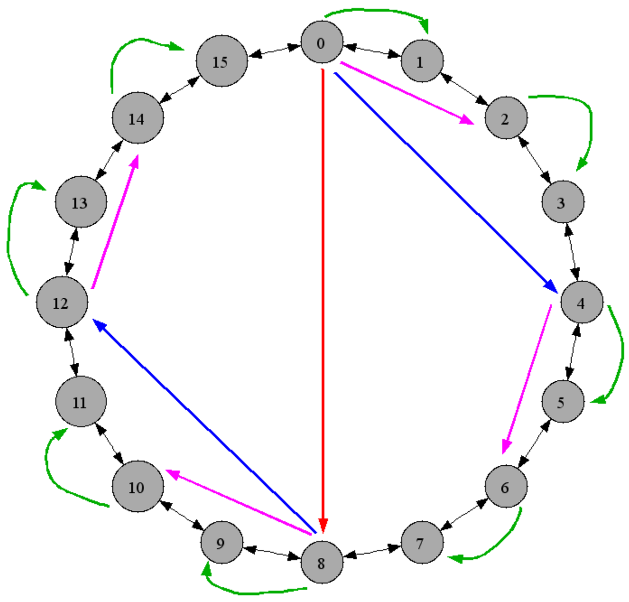

Figure 4: Node 0 broadcasts in a DKS with N=16, k=2

4 Handling Dynamism

So far, the basic network structure has been presented. However, there was no mention of network dynamism yet, which is an important aspect of every peer to peer network. New nodes need to be able to join the network and existing nodes need to be able to leave the network. Furthermore, nodes may crash or lose their internet connection. Whatever the cause, a node that leaves the network without utilizing the proper leaving procedure is a node that *fails*.

4.1 Locking and concurrency

The network should be able to handle multiple joining and nodes, such that all next successor and predecessor routes are still form a valid ring after any number of nodes joined, left or failed concurrently. In order to do so, a locking mechanism is introduced: Each node *hosts* a lock. A node can either be locked by itself, or by his next successor or its predecessor. In order to avoid resource starvation, each lock is released after a given timeout, while each message that relates to the process for which the lock was acquired renews the timeout to its original value. Note that a locking a node has no other effect than guaranteeing that no other node can also lock this node.

4.2 Joining

There are two cases when a node wants to join a DKS: He either wants to join an empty network, or he wants to join a network which contains at least one node. The former case is trivial: the node simply needs to set all routes either to himself (if applicable) or to *nil*. In the latter case, the joining node needs to know at least one node already existing in the network. An existing node could be advertised using DNS or similar means. This existing node would generate an node identifier n, $n \notin \mathcal{P}$, for the joining node. Note that there is no algorithm given in order to find n, but it is assumed that all node identifiers are distributed evenly across $[0..N[$. The contacted node then sends the identifier n and the next successor of n, to the joining node. We call n's next successor e. Inserting a node in the DKS is similar to inserting an object into a linked list.

First of all, node n locks itself, so that no joining node can use n to join the network until the joining of node n is finished. Then, node n sets its next successor route to e and sends a message to e, in which it requests a to lock e, and informs e that it wants to be e's new predecessor. If node e is already locked, it will tell this to node n, which will then wait an arbitrary time and try again. However, if node e is not locked, it will lock itself and set its predecessor pointer to n. From now on, it will forward all messages from its previous predecessor p to n, so that lookups stay consistent. Now e sends message back to n, informing n that is predecessor is p, and starts transferring all identifiers that n is responsible for to n. Furthermore, e tries to assemble a routing table RT_n that fits node n using only local information in node e. n now sets its predecessor pointer to p and sends a message to p, informing it that n is now p's next successor. p then sends a message to e, telling it accepted n as its new successor. e then releases its lock and sends a message to n that the join is completed. n then also releases its lock and is completely integrated into the network. Node n now can do further lookups in order to complete its routing table. Furthermore, node n

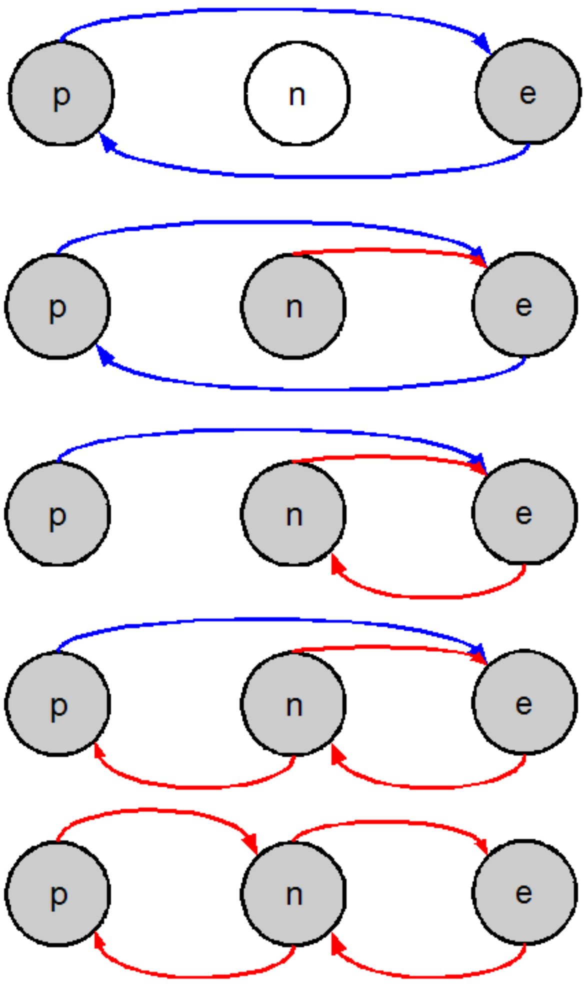

Figure 5: Node n joins the network

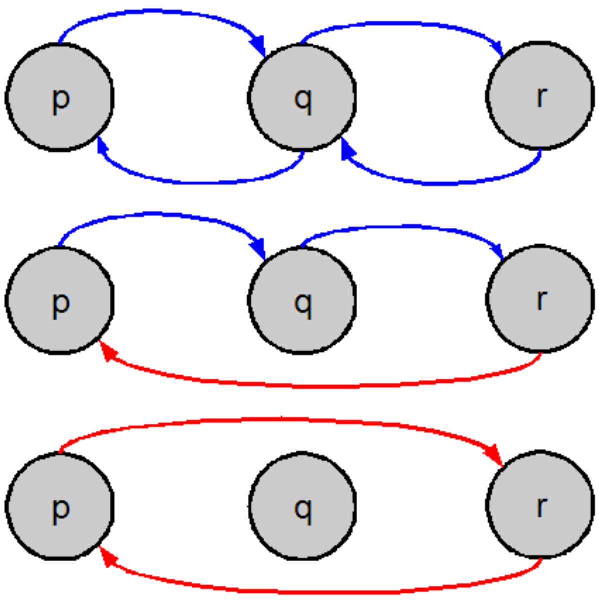

Figure 6: Node q leaves the network

contacts all nodes in its routing table that it has a route to them, so that they add node n to their backlist *bl*.

4.3 Leaving

The mechanism for leaving is also similar to removing an object from a double-linked list.

In order to keep the network in a consistent state, a node needs to lock itself and its next successor. If the algorithm would simple consist of the leaving node first locking itself and then its predecessor, it could run into a deadlock: if all nodes would try to leave the network at the same time, they would all lock themselves and then try to lock their next successor, which already locked itself. So in order to avoid deadlocks and livelocks, every node first locks itself and then tries to lock its next successor; except for the node with the highest identifier in the DKS, which first tries to lock its next successor and then itself. Note that a node has the highest identifier if his next successor's identifier is smaller than its identifier. With this algorithm, if all nodes want to leave the network simultaneously, they would all first lock themselves, except for the node with the highest identifier, which would try to lock its next successor, which already locked itself. Now the predecessor of the node with the highest identifier can lock the node with the highest identifier and leave the network.

In the trivial case where the node is the last node in the DKS, it can simply leave the network. A node can tell if its the last node in the network if it predecessor and next successor both point to itself.

Otherwise, node q locks itself if it isn't the node with the highest identifier in the network. Now node q sends a message to its successor r asking for a lock. If r is already locked, node q unlocks itself and retries after an arbitrary time. However if node q is unlocked, it locks itself and sends an acknowledgment message back to node q. If q is the highest identifier in the network, it now locks itself.

From now on, node q forwards all messages it receives to node r, in order to achieve lockup consistency. Node q also starts transferring all identifiers it is responsible for to node r. Node q also contacts all nodes in its routing table *RT* and backlist *bl* that it leaves and that they should remove all routes to it. Node q then sends a message to r, telling it that its new predecessor is p. Node r updates its predecessor route and sends a message to p, telling it that it is its new next successor. Node p then updates its successor pointer accordingly and sends a message to q that it no longer points to it. Node q then sends a message to r to release the lock and releases its own lock. Node q can now safely leave the network.

4.4 Failures

4.4.1 Replacing Nodes in the Routing Table

In order to detect nodes failures, messages are sent periodically to all nodes in the routing table *RT* and to the *next successor* succ, if no regular messages are already sent to them. If the node does not respond within a given time, it is assumed that the node failed and is no longer part of the network. Note that the only guarantee of this failure detector is that it detects node failures eventually. It may detect node failures very late; it may also detect node failures where none exist: Assume node n detects that node s fails. One reason could be that there is no (underlay network) route from n to s while there are still valid routes from both n and s to other nodes in the network. Or node s could be very busy and have a very long message queue, and is unable to respond in time to node n's request. Or the network connection between node n and node s could be saturated. These are just some of various possible reasons for false failure detection.

When a node detects a node in its routing table has failed, it tries to replace the failed node. In order to do this, it sends a lookup message which searches for the successor of the interval that the failed node was responsible for.

4.4.2 Symmetric Replication

One of DKS parameters is f, which specifies the fault-tolerance of the network: every information is stored f times in the network, so that $f - 1$ nodes can fail and the network recovers gracefully. This is true for the next successor and predecessor lists, which each consist of f entries. Furthermore, DKS uses a technique called *symmetric replication* to store all identifiers n f times in the network, at nodes $n \oplus \frac{N}{f}$, $i \in N_0$.

Figure 7 shows a DKS with $N = 16$, $f = 4$. In this example node 0 is responsible for all key/value-pairs with identifier 0, and thus stores identifier 0 plus multiples of $\frac{N}{f} = 4$. This means that it is responsible for identifiers 0, 4, 8 and 12. Node 6 is responsible for identifiers $H(key) \in [4, 6]$, as $S(4) = 6$, and all multiples of $\frac{N}{f} = 4$ which add to these identifiers. Node 6 stores identifiers 4, 8, 12, 0 because it is responsible for identifier 4, identifiers 5, 9, 13, 1 as it

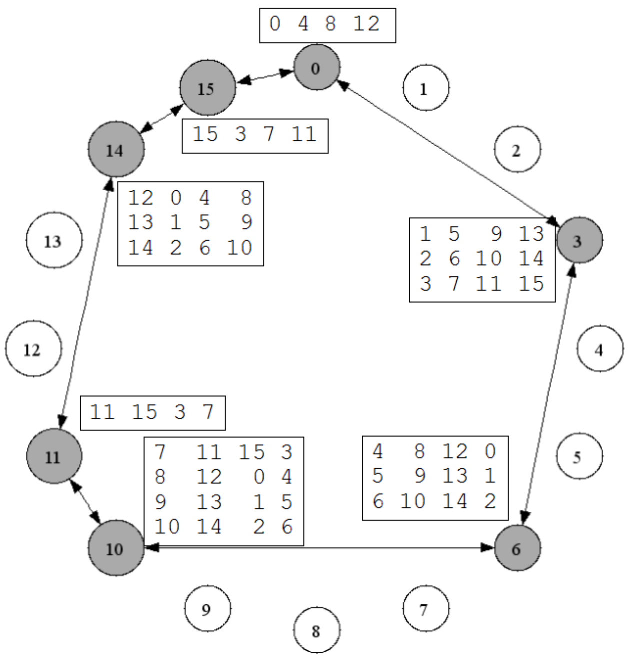

Figure 7: Nodes and the identifiers they are responsible for. DKS with N=16, f=4

is responsible for identifier 5 and identifiers 6, 10, 14, 2 as it is responsible for identifier 6.

Note that this divides the network into $\frac{N}{f}$ classes, which all store f identifiers. Thus, a node can lookup any of the f nodes in a class in order to retrieve a given key. On the downside, the network needs $O(f)$ messages to store an identifier. The algorithm presented in both [5] and [6] propose that a node simply sends a message to all f nodes that are storing the identifier. This however has the disadvantage that storing a key/value-pair is not atomic, as multiple nodes could concurrently try to store a value; as there is no guaruantee in which order the messages arrive, they could arrive in different order on different nodes. Each of the f nodes can only assume that the last received store message is the last one, which can differ from node to node. Thus, the network can remain in an inconsistent state after multiple nodes perform a store operation simultaneously. This is not to be desired. I propose two simple solutions to this problem: Either the node that wants to store a key/value-pair in the network sends it to the responsible node, which can then atomically distribute it to the $f - 1$ other nodes, or the node that wants to store a key/value-pair asks the responsible node for a ticket number and then stores the key/value-pair in all f nodes using this ticket number. The responsible node increases the ticket number by one after each request. Thus, storing operations can be ordered chronologically and a node simply ignores all storing operations for a given key if the ticket number is smaller than the ticket number of the currently stored key/value-pair.

When a node fails, this is detected by its successor. The successor then contacts the nodes in the class of its failed predecessor and fetches the identifiers in order to restore the replication.

The authors of [5] and [6] claim that symmetric replication combines the benefits of the replication schemes *successor lists* as used by the Chord system and multiple Hash functions as used by CAN or Tapestry.

5 Conclusion

In conclusion, the Distributed K-Ary System realizes a robust peer-to-peer network which can be used as multiple distributed hash tables and for group communication. Any node can be reached within $(log_k N) + 1$ hops. Broadcast and multicast are efficient. At any time, $f - 1$ nodes can fail and the network will recover gracefully. However, there is no discussion how N, k and f should be chosen for optimal results. Also, there is many research focusing on peer-to-peer networks like Chord, Pastry and Kademlia. As DKS claims that those networks can be seen as instantiations of DKS, it should be possible to port any improvement for these networks to DKS.

Finally, one can always download the Java DKS implementation [1] and try it out for oneself.

A Bibliography

References

[1] DKS CVS Repository http://calvin.sics.se/dks/cvsweb.cgi/.

[2] L.O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2003.

[3] S. El-Ansary. *A Framework For The Understanding, Optimization and Design Of Structured Peer-To-Peer Systems.* Ph. D. thesis, Swedish Institute of Computer Science (SICS) Kista, Sweden, 2003.

[4] A. Ghodsi, L.O. Alima, and S. Haridi. Low-Bandwidth Topology Maintenance for Robustness in Structured Overlay Networks. *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05)-Track 9-Volume 09*, 2005.

[5] A. Ghodsi, L.O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *The 3rd Int Workshop on Databases, Information Systems and Peer-to-Peer Computing, Trondheim, Norway*, 2005.

[6] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, December 2006.

[7] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, 258:263, 2002.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. *A scalable content-addressable network.* ACM Press New York, NY, USA, 2001.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218(0):329–350, 2001.

[10] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.

[11] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. *Computer*, 74, 2001.