Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Tampering with Distributed Hash Tables

Master's Thesis

Michael Voser

vosermi@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Jochen Seidel, Christian Decker
Prof. Dr. Roger Wattenhofer

May 25, 2013

Acknowledgements

At first I would like to thank Christian Decker and Jochen Seidel for the pleasurable collaboration during my master's thesis. Their continuous feedback and valuable inputs encouraged me and opened my eyes for new methods. A special thanks is due to professor Roger Wattenhofer for the supervision and the opportunity to realize this thesis at the Distributed Computing Group.

Finally, I would like to express my gratitude to all of my friends and families for their moral support.

Abstract

BitTorrent is the most popular P2P protocol for file sharing on the Internet. Each BitTorrent client also participates in the DHT and acts as a distributed tracker. It therefore maintains a routing table to store contact information about other nodes in the DHT. In this thesis we propose a method to store data in the routing table of random nodes in the DHT. Thus we are able to store and share data on other computers without the users knowledge.

Contents

Chapter 1

Introduction

BitTorrent is the most used peer-to-peer (P2P) file sharing protocol on the Internet. It has millions of users and is responsible for approximately one third of the global internet traffic [1]. Unlike other P2P file sharing systems, BitTorrent does not provide a content search feature. Instead BitTorrent relies on torrent files that contain metadata about the file to be shared. Torrent files are published on websites for download and contain a reference to a tracker. A *tracker* is a centralized server that is used as the first meeting point for peers and provides a list of peers that are interested in the same file. The set of peers participating in the distribution of the file forms a *swarm*. After joining a swarm, the peers start to download pieces of the file from other peers in the swarm.

The use of trackers poses a single point of failure. If a server that is used as a tracker becomes unreachable for some reason, the peers are unable to join a swarm since the information of who already joined the swarm is unavailable as long as the tracker is unreachable. For this reason, it is important to come up with a solution to make the protocol more fault tolerant.

All popular BitTorrent clients also implement a “trackerless” system trough a distributed hash table (DHT). Trackerless does not mean that no tracker is used at all, but that the traditional centralized tracker is supplemented by a decentralized implementation of the tracker’s functionality. To that end each BitTorrent peer also acts as *node* participating in the DHT. DHTs have attracted a lot of attention due to their inherent scalability and reliability as generic data stores. The *BitTorrent DHT* is a highly specialized DHT for the BitTorrent use case. We take a step back and ask whether it is possible to store arbitrary data in this DHT. It is.

The method presented in Chapter 2 to store data in the BitTorrent DHT is based on a general framework for the BitTorrent DHT. This framework is presented in Section 2.1 of Chapter 2. It is also used in our proof-of-concept to show that storing data in the BitTorrent DHT is possible. We experimentally determine the performance of our methods as awell as the performance of the DHT in general in Chapter 3. Chapter 4 summarizes our findings and points out potential future work. We start by giving an overview of how the DHT operates in the following sections.

1.1 DHT Tracker

Like a traditional hash table, a distributed hash table allows to store and retrieve key/value pairs. The key/value pairs are distributed among a set of cooperating computers, which we refer to as *nodes*. The values are indexed by keys. The main service provided by a DHT is the look-up operation, which returns the value associated with a given key. The DHT used with BitTorrent is based on the Kademlia protocol [2]. The nodes exchange messages over UDP. Every node is responsible for the storage of a set of values whose keys are in a given range. Each node in the network has a unique node ID that is randomly chosen from a 160-bit key space. Every value that is stored in the DHT is also assigned a 160-bit key from the same key space as the node ID (e.g., the SHA-1 hash of some larger data). The value to this key is stored on nodes whose node ID is closest to that key. The distance between two points A and B in the key space is the bitwise exclusive or (XOR) interpreted as an unsigned integer:

$$\text{distance}(A, B) = |A \otimes B| \tag{1.1}$$

Smaller values are closer. Therefore the longer the prefix that two node IDs share, the closer are the nodes in the DHT. If we say that two nodes A and B *share x prefix bits*, then the first x bits of the IDs of nodes A and B starting from the most significant bit are equal. In that case $\text{distance}(A, B)$ must be less than 2^{160-x}. Table 1.1 shows the distance calculation for two IDs in the 4-bit key space. Nodes A and B share one prefix bit. If there is a node C that shares more than one prefix bit with Node A, then node C is closer to node A than node B.

In Kademlia, each node N maintains a *routing table*. The routing table consists of 160 lists. These lists are called *buckets* and are *indexed* from $1, .., 160$. Each bucket holds a maximum of k nodes (where $k = 8$ for the BitTorrent DHT). The nodes in bucket i share i prefix bits with N. The *contact information* for nodes consists of the triple $\langle$IP address, UDP Port, node ID$\rangle$. For each $0 \leq i < 160$, every node keeps a bucket of 8 nodes of distance between 2^i and

Node	ID (4-bit)	Decimal value
A	0101	5
B	0011	3
$(A \otimes B)$	0110	6

Table 1.1: Example of a distance (XOR) calculation between two 4-bit IDs.

2^{i+1} from N. This structure of the routing table can be illustrated by a binary tree (Figure 1.1). The bucket with *index* i contains a subset of the nodes in the i-th *sibling subtree*, i.e., the subtree rooted at the sibling of the node corresponding to the i-th bit in N's ID. The routing table for a 4-bit key space in Figure 1.1 is maintained by the node with ID 1001. Nodes in bucket 0 share no prefix bit with the node 1001, nodes in bucket 1 share one prefix bit with node 1001. The ID 1000 in bucket 3 is in brackets because there is no node with this ID. The node with ID 1000 would be the only node sharing 3 prefix bits with ID 1001 except node 1001 itself.

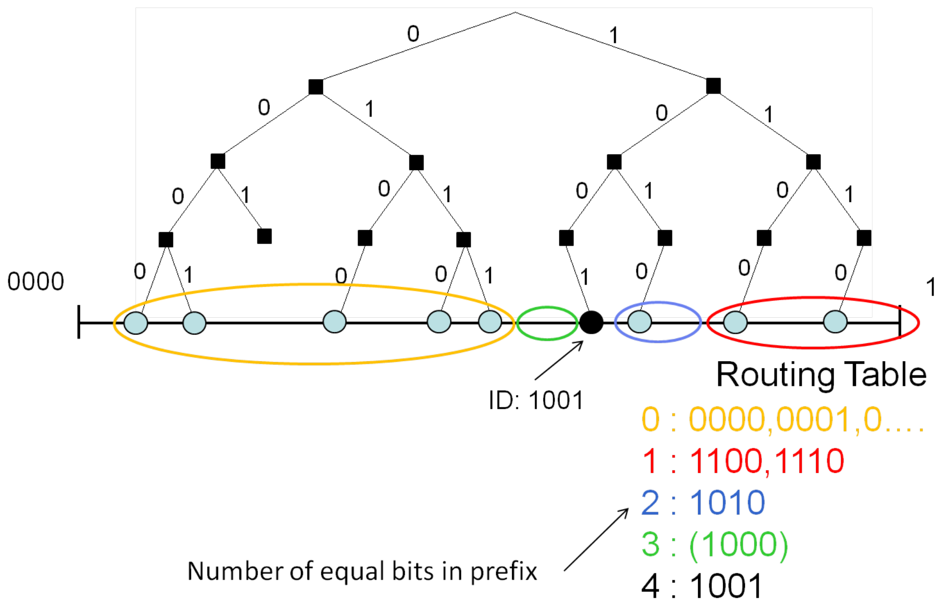


Figure 1.1: Tree topology of the routing table.

Kademlia defines two different message types: queries and responses. The protocol consists of four different queries and for each query there exists a corresponding response. Each query is parameterized by one or more *arguments* and each response contains one or more *return values*.

- The **PING** query has the querying node's ID as an argument. The response returns its own node ID. This query is used to see if a node is online.
- The **FIND_NODE** query takes two arguments: again the querying nodes ID and a search ID. The FIND_NODE response contains as return value a list with ⟨IP address, UDP Port, node ID⟩ triples for the 8 nodes from its routing table closest to the search ID.
- The **STORE** query has two arguments: the querying nodes ID and a

key/value pair. The STORE response returns its own node ID. The STORE message is used to store key/value pairs.

- The **FIND_VALUE** query takes two arguments: the querying nodes ID and a key. The FIND_VALUE query behaves like the FIND_NODE query, only the responses are different. If the queried node has a value stored with a key that equals the search ID, it returns the value and a nonce which provides some resistance to address forgery. If there is no value stored for the given key, the FIND_VALUE response is equal to the FIND_NODE response.

Kademlia uses an iterative look-up algorithm to locate nodes near a particular ID. We refer to this ID as the *search ID.* A node N_1 that performs a look up for a search ID N_F, it first consults its routing table to find the closest node N_2 to N_F it knows. A node N_A *knows* node N_B when it has its contact information stored. Node N_1 then queries N_2 with a finde_node message. N_2 answers to the find_node query with the closest nodes to N_F from its own routing table. If the response from N_2 contains a node N_3 that is closer to N_F than N_2 itself, then N_1 sends another find_node query, this time to node N_3. This procedure is repeated until no find_node response return value contains a closer node than the ones N_1 already knows. Each query brings N_1 at least one bit closer to the search ID. Therefore the iterative look-up procedure guarantees to terminate after $O(log(n))$ queries.

Figure 1.2 illustrates a look-up example. The colored dots represent the nodes in a 4-bit key space. The search ID is 0000. In our example the red node with ID 0000 is the closest node to the search ID. The green node with ID 1100 cannot directly contact the red node because it does not know which node is closest to the search ID. In its own routing table it finds ID 0111. The green node then queries this node. Node 0111 responds with the closest nodes to the search ID contained in its own routing table. The green node now has a closer node from the response of the first queried node. It then queries the new learned node 0011. This node also responds with a closer node, the node with ID 0001. After 4 find_node queries, the green node receives the contact information of the red node. The red node is also queried and responds to the find_node node query, but there will not be a closer node in this response. The black arrows show the sent queries and the red dashed arrows illustrate the progress of each query.

Two different DHT implementations that differ in features and message format are used in the context of BitTorrent, namely Azureus DHT and Mainline DHT. Azureus DHT is only implemented by one single client. The Mainline DHT is implemented across mulitple popular clients and has thus a larger user base. We therefore restrict ourselves to studying the Mainline DHT, and use terms Mainline DHT and BitTorrent DHT interchangeably.

In BitTorrent the DHT assumes the role of a tracker. It stores the contact

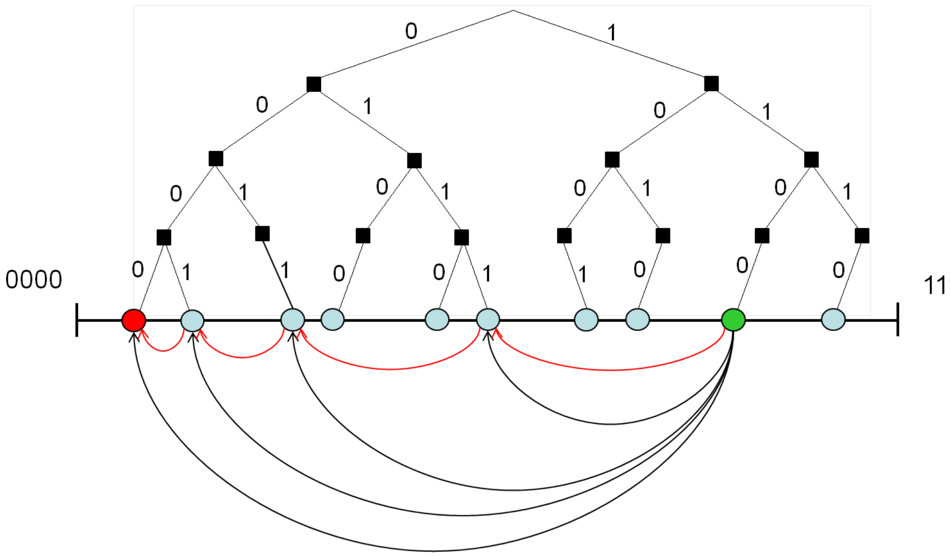


Figure 1.2: Example of an iterative lookup in a 4-bit key space.

information of peers that share a certain file. The contact information of peers contain the IP address and the TCP Port. Other then nodes, peers communicate over TCP. As described above the DHT stores key/value pairs. In BitTorrent the key is the *info_hash* of a file. The *info_hash* is a SHA-1 hash that uniquely indetifies content (single or multiple files) which is shared via BitTorrent. The value of the BitTorrent DHT is a *set of peers* that share this content. Each node has a unique identifier known as the node ID and is fully described by the triple ⟨IP address, UDP Port, node ID⟩. Node IDs are chosen arbitrarily from the 160-bit key space, i.e., the same key space as the *info_hash*. The node ID selection is supposed to be random, but as this choice is not verifiable a node may in fact choose any ID. We will show in Chapter 2 how this can be used for our benefit.

Each node in the BitTorrent DHT maintains a routing table with contact information of other nodes. If a node receives a message, the sender node may be added to the routing table of the receiving node. Therefore each message exchange reinforces the contact information. Nodes use the contact information from the routing table as a starting point for a node look up. Nodes are added to the routing table by their ID. Because of the 160-bit key space, the routing table is divided into 160 buckets. Each bucket contains a list of a maximum of 8 nodes. The bucket in which a node is inserted depends on the length of the prefix a node shares with the ID of the node to whose routing table the node is added. Figure 1.1 shows an example how the nodes are inserted.

Messages are exchanged between nodes with serialized datastructures. Compared to Kademlia, Mainline DHT has an additional message type, the error

Message	Query arguments	Response values
ping	• querying nodes ID	• queried nodes ID
find_node	• querying nodes ID • target ID	• queried nodes ID • list with 8 closest nodes to the target id
get_peers	• querying nodes ID • info_hash	• queried nodes ID • token • list of peers or • list with 8 closest nodes to the target id
announce_peer	• querying nodes ID • info_hash • port • token	• queried nodes ID

Table 1.2: An overview of the four queries and responses used in the BitTorrent DHT.

messages. Error messages are used to respond to queries if for example the message format is invalid. Mainline DHT also defines four queries and four responses, where the names and semantics are slightly different. Table 1.2 shows the queries and their corresponding responses that are defined for the BitTorrent DHT. The main difference from Kademlia is how a value is stored. In Kademlia, key/value pairs are stored with the STORE query whereas in the BitTorrent DHT, the corresponding query is called announce_peer. The announce_peer query does not really store a value, it merely adds a peer to the set of peers at the given key. Correspondingly, the get_peers query replaces the FIND_VALUE query and does not return all peers, i.e., the actual value, but only 50 random peers from the set of peers stored at that key.

1.2 Related Work

Crosby et al. [3] analyse the two DHTs that are used in BitTorrent which both are based on Kademlia. Mainly the characteristicts of the two DHTs are studied and compared but also security concerns and design issues are identified.

Lin et al. [4] propose a suite of security strategies for the DHT network in BitTorrent system to make it less vulnerable against attacks.

Arvid Norberg[1] proposes a DHT extension that combines the IP with the ID to make it harder to launch attacks against the BitTorrent DHT.

Ratnasamy et al. [5] introduce a Content-Addressable Network (CAN) that is fault-tolerant and completely self-organizing and demonstrate its scalability,

[1]http://www.rasterbar.com/products/libtorrent/dht_sec.html

robustness and low-latency properties through simulation. The CAN has a distributed infrastructure like DHTs.

Jin et al. [6] study malicious actions of nodes that can be detected through peer-based monitoring. The simulation results show that this scheme can efficiently detect malicious nodes with high accuracy, and that the dynamic redistribution method can achieve good load balancing among trustworthy nodes.

Varvello et al. [7] have monitored during four consecutive days the BitTorrent traffic of tracker-based and DHT-based traffic within a large ISP. They propose a design of a traffic localization mechanism for DHT-based BitTorrent networks.

Stutzbach et al. [8] study the performance of the key look-up in DHTs. They investigate how the efficiency and consistency of a look-up can be improved by performing parallel look-ups and maintaining multiple replicas.

CHAPTER 2

Implementation

In this chapter we describe the methods we used to crawl the DHT and write data into it. We also present how these methods have been implemented (in Java). The software consists of 3 main parts descpicted in figure 2.1. A BitTorrent framework provides the core functionality to operate a node participating in the DHT. Two prototypical applications are developed on top of this framework: (1) A *crawler* that allows us to explore the DHT overlay network and (2) a DHaTaStream that show-cases the ability to store arbitrary data in the routing table of remote nodes.

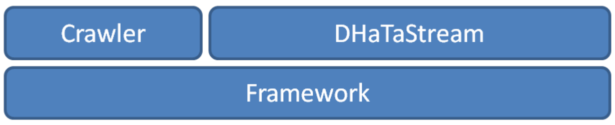


Figure 2.1: The 3 main software parts.

2.1 DHT Framework

The DHT framework provides the basic functionality a node requires to join the DHT and to interact with other nodes. The node needs to be able to send and receive queries and responses and has to maintain a routing table. Our framework has the ability to create and control multiple nodes. A node that is controlled by the framework is called *LocalNode* whereas other nodes participating in the DHT are called *RemoteNode* represented by RemoteNode objects. RemoteNodes are contolled by other clients than us, it is just a local placeholder object that represents another node. A RemoteNode holds contact information such as node ID, IP and Port. The number of LocalNodes that are created can be chosen by the user of the framework. The LocalNodes are distinguished by their user-defined node ID and port number. All LocalNodes share the same IP address. When a LocalNode first joins the DHT it contacts a *boostrap node* and looks up its own ID. A bootstrap node is a well known node that provides initial information to newly joining nodes. While performing this iterative look-up, the routing table gets filled with RemoteNodes that have answered to the find_node queries. A RemoteNode stays at least 15 minutes in the routing table. If a LocalNode has not received a message from a RemoteNode R contained in its routing table, then the RemoteNode may not be reachable anymore. In that case, the LocalNode pings R to see if R is still online. If there is no response from R within another 15 minutes, then R is deleted from the routing table. There

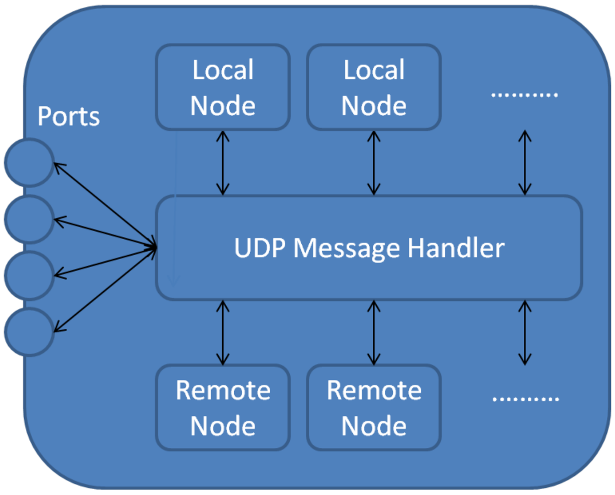


Figure 2.2: Overview of the framework architecture.

are several reasons why a RemoteNode may not responds to a query. Firstly, some do not answer at all depending on the client that is used. Secondly, some nodes have already left the DHT but are not yet removed from the routing table and therefore will not respond. Thirdly, some nodes have an anti *Denial-of-Service* (DOS) attack mechanism. A DOS attack is an attempt to make a node unavailable for other users. A common method of a DOS attack is to saturate a node with a huge number of queries so that it is unable to respond. A node may falsely detect a DOS attack and stop responding to queries that in fact were not sent with malicious intent. Lastly, messages can be lost because of the unreliable UDP protocol that is used for the message exchange. We thus cannot know if a query or its response was lost during transmission when no response is received. For that reason we have built in a retry functionality to be able to resend the message.

Because the framework is able to run several nodes, all incoming packets need to be routed to the targeted LocalNode. This is done by a multiplexer that handles the UDP packets (UDP Message Handler, c.f. Figure 2.2). It reads the port from which the packet has been received, looks up which LocalNode is identified by this port and forwards it correspondingly. The message is then processed by the LocalNode. Each message contains a 2 byte transaction ID which

N	S_3	No. of equal prefix bits
01001101...	010**1**1101...	3 (010........)

Table 2.1: Search ID calculation for $i = 3$ with a key space of 160-bit.

is included in the response to a query. This ensures that all incoming responses can be associated with the corresponding query. The number of messages that can be on the way is 2^{16} per RemoteNode. Our framework is able to operate multiple nodes in the DHT. Both the crawler and the DHaTaStream are built upon this framework. The framework will also be included into BitThief [9] to allow usage of the BitTorrent DHT for swarm discovery.

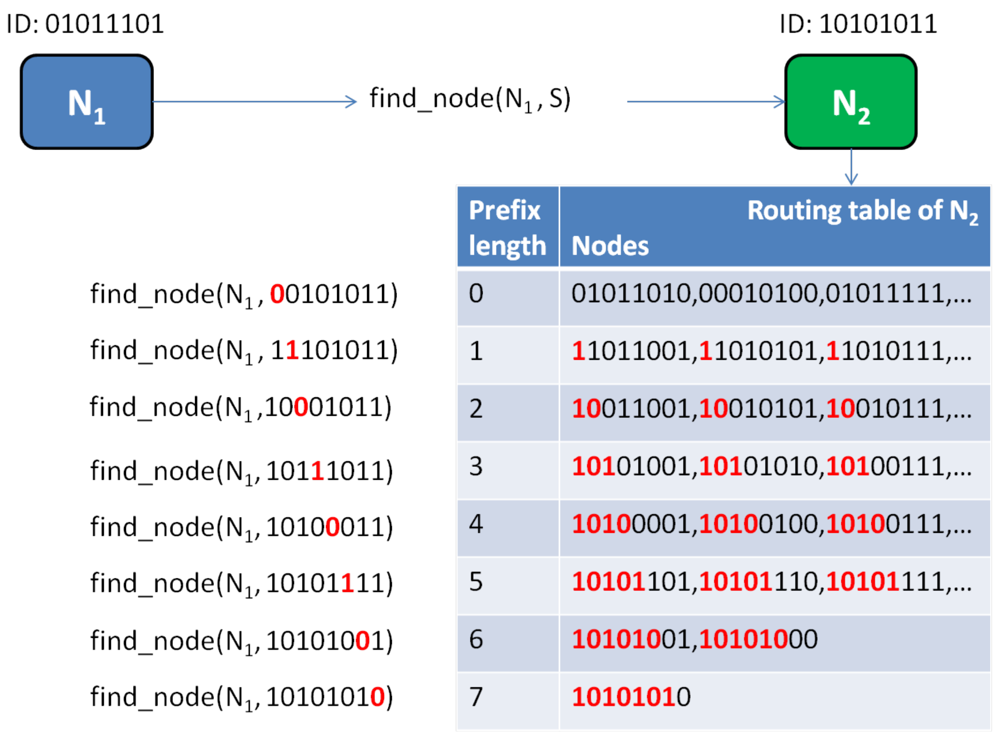
2.2 DHT Crawler

The goal of the crawler is to be able to contact as many nodes of the DHT as possible, preferably in a short time, to get a representative snap-shot of the DHT. With the collected information we want to reconstruct the DHT's overlay network and detect anomalies such as node IDs that are equal to an *info_hash* (indicates targeted positioning) or nodes with different IDs that are controlled by the same computer, i.e., IP address. The brute force method would be to send a message to every ID from the keyspace of 2^{160} IDs. This approach is of course practically infeasable.

Our approach is thus as follows. First a LocalNode N is created with the framework. Then we contact a bootsrap node and try to read the whole routing table of this node. To read the routing table, we have to send a find_node query for each bucket of the RemoteNode. A bucket can be read by taking the asked nodes ID and flip one bit at the appropriate position. We then send a find_node message to the asked node and take the ID with the flipped bit as the search ID. The search ID is calculated with the equation 2.1.

$$S_i = \text{flipbit}_{160-i}(N) \tag{2.1}$$

In order to get the search ID S to read bucket i we have to flip the bit $160 - i$ of the node N. Table 2.1 shows the search ID calculation to read the third bucket ($i = 3$). The bold bit was flipped. The search ID is an argument for the find_node query that the node N answers with the find_node response, i.e., the nodes it has stored in the third bucket of its routing table. If we want to read all buckets of a node's routing table we would have to send a find_node query for each search ID S_{ID_i} with $i = 0...160$. Since almost never all buckets are filled with nodes, we investigate the distance between the returned node IDs and the sender's ID. If the closest node of the returned nodes has a far smaller shared prefix than 160 bits with the asked node, then we can immediately jump to read the bucket where the closest node to the sender ID is stored. In the majority of cases the first 130 bits can be skipped and we only have to send 30 ($160 - 130 = 30$) queries to read the whole table. This trick reduces the time required to wait for responses and speeds up the crawler. After sending the queries, all returned nodes from the find_node responses are then added to a task queue. The task queue contains RemoteNodes we have not completely read yet. The node which sent the response is added to the set of good nodes. Good nodes are nodes that have responded to a find node query. It is important to know which nodes we have already in the task queue in order to not enqueue the same node multiple times. We now continue to read the routing table of nodes from the task queue until this queue is empty. The task queue does not only contain RemoteNodes, it stores tuples which consist of the RemoteNode and the current bucket index we are reading. If a RemoteNode responds to a query, the bucket index is increased by one and the tuple enqueued to the task queue. This procedure guarantees that only one query per RemoteNode is in flight at anytime. If too many queries are send to a RemoteNode, it may be detected as a DoS attack as explained earlier and the node would not respond to our queries. The problem with having only one active query is that it significantly slows down the crawler. To compensate this loss of speed we query multiple nodes in parallel.



	Prefix length	Nodes
find_node(N$_1$, 00101011)	0	01011010,00010100,01011111,...
find_node(N$_1$, 11101011)	1	11011001,11010101,11010111,...
find_node(N$_1$,10001011)	2	10011001,10010101,10010111,...
find_node(N$_1$, 10111011)	3	10101001,10101010,10100111,...
find_node(N$_1$, 10100011)	4	10100001,10100100,10100111,...
find_node(N$_1$, 10101111)	5	10101101,10101110,10101111,...
find_node(N$_1$, 10101001)	6	10101001,10101000
find_node(N$_1$, 10101010)	7	10101010

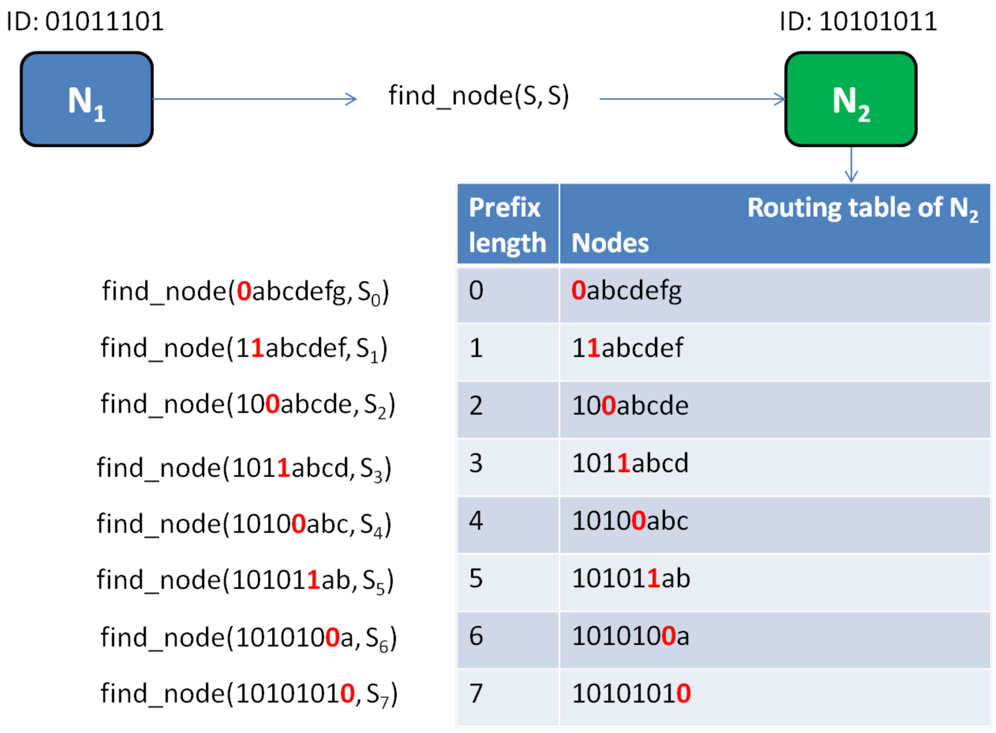
Figure 2.3: Node N_1 wants to read single buckets from node N_2 by flipping the corresponding bit in the search ID S.

2.3 DHaTaStream

The regular way to use the BitTorrent DHT is to add peer information (IP address, TCP Port) to a set of peers at a certain key. Recall that all the peers of this set share the same file with an *info_hash* that is equal to the key. Our goal is to use the the BitTorrent DHT to store *arbitrary* data. The first possibility would be to use the TCP Port field of the peer information to store data. The TCP Port field has a data size of 2 bytes. This is not much of storage space and we would not even be able to use the full 2 bytes for data because we would also have to include some sorting mechanism, for example including sequence numbers within the data. The sequence number can then be used when reading the data to put it in the correct order. Another issue with this approach is that when a node wants to get peers for a certain key (by issuing a get_peers query), the queried node returns a set of 50 random nodes contained in this set. Therefore it is not guaranteed that the set elements with the stored data are returned. Instead we investigate a more intricate way to store data in the BitTorrent DHT. The idea is to write and read data to and from the routing table of nodes in the DHT. A node N_1 adds node N_2 to its routing table when N_1 receives a message from N_2 and the corresponding bucket of the routing table of N_1 is not full yet. As we observed in Chapter 1, each node in the DHT can choose its own node ID.

If we wish that node N_1 is added to node N_2's routing table, then we have to know N_2's ID. Node N_1 can locate the bucket it wants to be added, by adjusting N_2's ID, but it can only be added when this bucket is not full yet, i.e., has not already stored 8 nodes. Adjusting the ID means that we can choose the number of equal prefix bits by flipping the bit at the proper position. How the position where the bit is flipped relates to the bucket that is aimed to be written is shown in equation 2.1. Figure 2.4 shows how N_1 has to adjusted N_2's ID to be added in the intended x-th bucket. The remainder of the the 160 bits, i.e., the bits on the right side of the flipped bit are not important in order to be added to bucket i of the routing table. We use these bits to store data. In order to utilize the full bucket, we send 8 find_node queries from 8 different ports, using the port as sequence number. This is important because when we read the data from the bucket we need to put the data in the correct order. Writing to a bucket of the routing table works simililar to reading a bucket from the routing table. Reading a bucket was already described in the previous section. The difference between reading and writing is that if we read a bucket we have to flip one bit of the search ID and when we want to write into a bucket we have to flip one bit of the sender node ID.

We cannot write into the routing table of every node. Some have more buckets that we can write into, some have only a few buckets we can use to store data and some do not add nodes at all. Some implementations do not actually initiate 160 buckets limiting the maximum prefix length to 30 bits.


ID: 01011101
ID: 10101011
N_1
find_node(S, S)
N_2
find_node(0abcdefg, S_0)
find_node(11abcdef, S_1)
find_node(100abcde, S_2)
find_node(1011abcd, S_3)
find_node(10100abc, S_4)
find_node(101011ab, S_5)
find_node(1010100a, S_6)
find_node(10101010, S_7)

Prefix length	Nodes (Routing table of N_2)
0	0abcdefg
1	11abcdef
2	100abcde
3	1011abcd
4	10100abc
5	101011ab
6	1010100a
7	10101010

Figure 2.4: Example of writing to a bucket with a node ID of 8-bit.

Our implementation of the described method, called the DHaTaStream, consists of four layers. Figure 2.5 displays the structure of the DHaTaStream.

- The first layer consists of *RemoteStorage* (RS) objets. These objects are an abstraction for a storage space in the DHT. It consists of a RemoteNode and the storage space that is available a this RemoteNode.
- The RS objects are controlled by the *RemoteStorageCombiner* (RSC) from the second layer. The RSC searches the closest RemoteNode to a given search ID and tries to write data into that RemoteNodes routing table. It builds on RS objects with variable size and opportunistically adds new RS objects as needed. In order to have one starting point and not lose track of the full storage path, the search ID is hashed for each new node. Thus a deterministic chain of keys can be built to know the path of storage. This is helpful for the reader if a node is not reachable anymore, it is still possible to continue with reading. The RSC looks up as many nodes as necessary to store all data it has to write. Because the look-up for RemoteNodes takes up some time, we decided to concurrently write data and search for the next RemoteNode storage. The RSC operates rather slowly.
- Writing only works sequentially because it is not possible to guess how

much can be stored in each RS object. Therefore we created a further abstraction, the *DataStriper* (DS). It represents the third layer of our software architecture. The DS, as the name implies, stripes the data into equal length stripes and writes it parallel to several RSCs. The purpose of the DS is to speed up writing.

- The fourth layer applies error correction coding to the data in order to correct missing data due to nodes that have left the DHT and cannot be read anymore.

We have implemented layer 1 to 3. Layer 4 is not in the scope of this thesis. An error correction code that is used for similar applications is the Reed-Solomon code [10] [11].

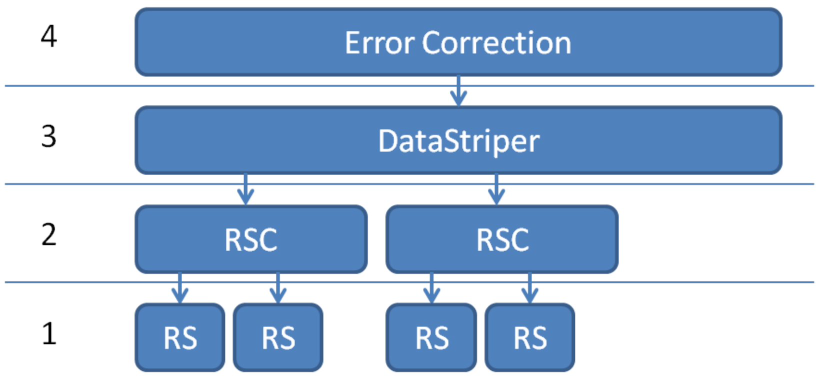


Figure 2.5: Architecture of the DHaTaStream.

Chapter 3

Evaluation

In order to measure various aspects of the DHT and estimate their impact on the presented software we designed the following experiments. Some experiments also help to determine parameters that are used in the software such as the *response timeout* and the *query retry count*. Those parameters are used in the node look-up. The response timeout is the time a LocalNode waits for a RemoteNode to respond to a query. The query retry count is the number of times a query is retransmitted, i.e., how many times we wait for an additional timeout. The reason why we use multiple retries is the unreliablility of the UDP Protocol that is used for the message exchange.

Node density. The purpose of the first experiment is to estimate the number of participating nodes in the DHT. This is accomplished by performing look-ups for multiple randomly chosen IDs. For each random ID we log the number of prefix bits which it shares with the node ID returned by the iterative look-up. Equation 3.1 is used to calculate the number of participating nodes n in the DHT.

$$E[L] = \sum_{i=0}^{N-1} 1 - \left(1 + \left(\frac{1}{2}\right)^{N} - \left(\frac{1}{2}\right)^{i}\right)^{n} \tag{3.1}$$

The random variable L is the measured quantity from the experiment, i.e., the maximum shared prefix length between a search ID and the closest node found by the iterative look-up. The expected value $E[L]$ is determined by taking the mean value of all measured values. N is the number bits in the ID ($N = 160$). The calculation of a look-up list for different values of $E[L]$ gives us the value for n. The resulting number of participating nodes in the DHT is around 6 millions. The Karlsruhe Institute of Technology (KIT) operates a DHT live monitoring[1]. The measured number by the KIT fluctuates between 6 and 10 million nodes.

[1]http://dsn.tm.kit.edu/english/2936.php

For further calculations we use the mean value of the KIT live monitoring (8 millions) because of its long term application.

Writable nodes and storage size per node. In this experiment we measure the number of nodes that are *writeable*, i.e., nodes that can be used store data. Furthermore we measure how much data can be stored in writeable nodes, in particular the *average storage capacity* of a node. Whether a node is writable depends on the node implementation since the routing table policy is left to the implementors. The mesurement starts with trying to store data in a random node. If data storing is successful, we try to store as much data as possible in this node. This procedure is repeated to get representative values. During this experiment we log the number of all queried nodes, the nodes that are writeable and how much data we are able to store. Combining the total DHT size and the number of writable nodes with the average storage capacity gives us the globally available storage of the BitTorrent DHT. Figure 3.1 shows the distribution of the nodes according to the number of bytes that could be stored in the routing table. Two peaks are noticeable. Nodes that store less than 500 bytes and nodes that store around 9000 bytes. The average storage size per node results in 4150 bytes. About 15.19% of all nodes in the DHT are writeable. With 8 million nodes in the BitTorrent DHT the total storage size is more than 5 gigabyte. Not all nodes that are writeable are also readable. A written node may simply leave the DHT or cannot be found anymore. Because of this reason we suggest adding an Error Correcting layer to gain redundancy. The next experiment investigates the probability to find the same node again after a certain amount of time.

Node search consistency. We want to measure how long the same node is found when searching for the same ID. The experiment starts with a look-up of a random ID. Every 2 minutes the look-up for the same ID is repeated for about the span of one hour. Before each look-up, the routing table is refreshed in order to avoid bias. The resulting node ID of each look-up with the search ID is logged for later comparison. This experiment gives us the probability of finding the same node again after up to one hour when performing a look-up for the same search ID. Figure 3.2 shows that after 2 minutes we find the same node again with a probability of 84%. After 25 minutes we still have a probability of 70%. An hour after the first look up the probabilty to find the same node is still over 60%. The node search consistency indicates how much redundancy is needed to compensate for data losses due to nodes that have become unavailable.

Timeout and retry sensitivity. In this experiment we measure the influence of two parameters on the distance between a random search ID and the closest node to this ID. The closest node to the search ID is determined using the iterative look-up. The parameters varied during the look-up are the response

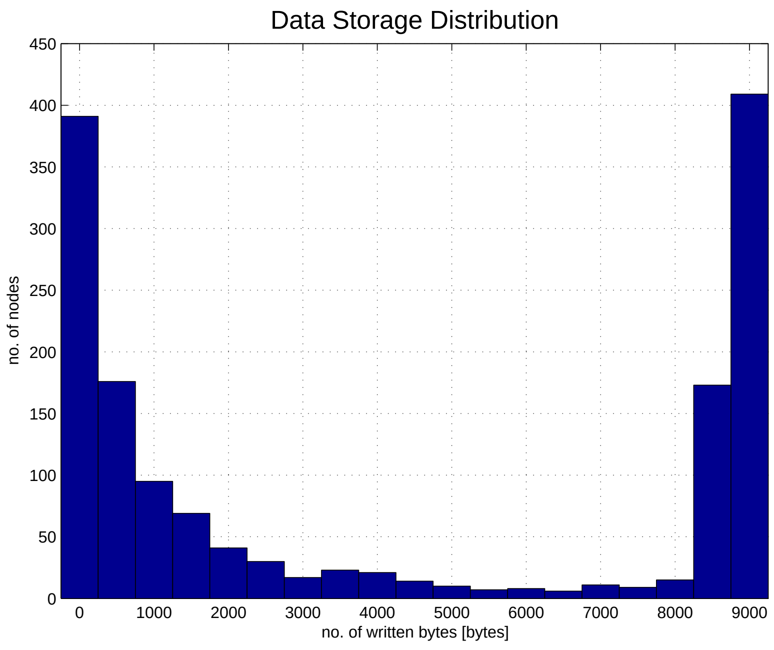


Figure 3.1: Writeables nodes either store around 5000 or 9000 bytes.

timeout and query retry count. The purpose of this experiment is to determine good values for response timeout and the query retry count, i.e., the values should be chosen so that the look-up is fast as well as robust. The look up is said to be robust when it finds the node that really is closest to the search ID. We define the following procedure: a node look-up is performed for a random ID to find the closest node for each combination of response timeout and query retry. The response timout is varied from 1 to 5 seconds in 1 second steps. The query retry count starts with 0 retries and ends with 3 retries. This leads to 20 measured configurations. For each measurement we perform several look-ups and log the number of equal prefix bits the found node shares with the search ID. The result of this experiment is depicted in Figure 3.3. The plot shows that the longer the chosen response timeout the closer is the found node to the search ID. However, a long response timeout alone does not guarantee that the closest node is found. The results for zero retries in Figure 3.3 indicates that the query has to be resent at least once. There is a big gap between no query retry and other query retry counts. The plot also shows that whatever parameter is chosen the found ID never has more than 24 equal prefix bits. This means that it does not help to further increase the response timeout or the query count. We are not able yet to chose the optimal parameters for the node look-up since we have not yet considered the time a look-up takes for the different parameters. But we can estimate that the longer the response timeout and the larger query retry count,

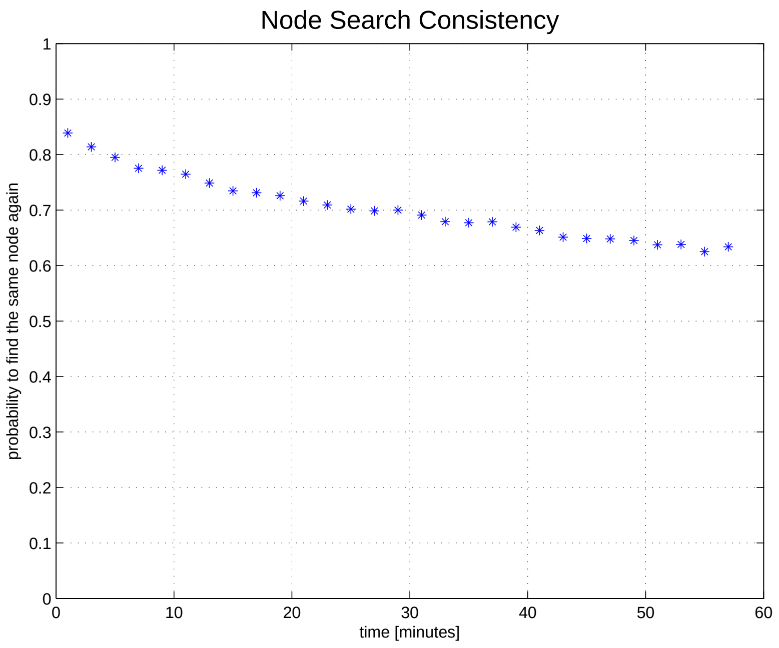


Figure 3.2: After 25 minutes, the probability to find the same node for a look-up of the same search ID is 70%.

the longer takes the look-up.

Node search time. The node search time is the mean amount of time we need to terminate the search a given ID. For all the look-ups we recorded during the timeout and retry sensitivity experiment, we also log the search time. Figure 3.4 shows the results of the of the experiments. It can be seen that the node search time increases with longer response timeout and larger query count. With the graphs in Figure 3.3 and 3.4 we are able to determine response timeout and query retry count to find the closest node to a search ID in the shortest amount of time. For example, if we choose a response timeout of 3 seconds with one query retry, the the look-up returns a node that has an equal prefix lenght of 23 with the search ID and a look-up time of about 17 seconds. The best choice for our parameters are 1 second response timeout with 2 query retries which results in a node search time of around 10 seconds.

Writing/reading speed. With this experiment we measured the writing and reading speed we have on average when writing to a writeable node. This experiment gives us the data size we were able to store and the amount of time it needs to store that data. After the writing has finished the data is read from the same node, and the time required to read the data is recorded. We are able to calculate

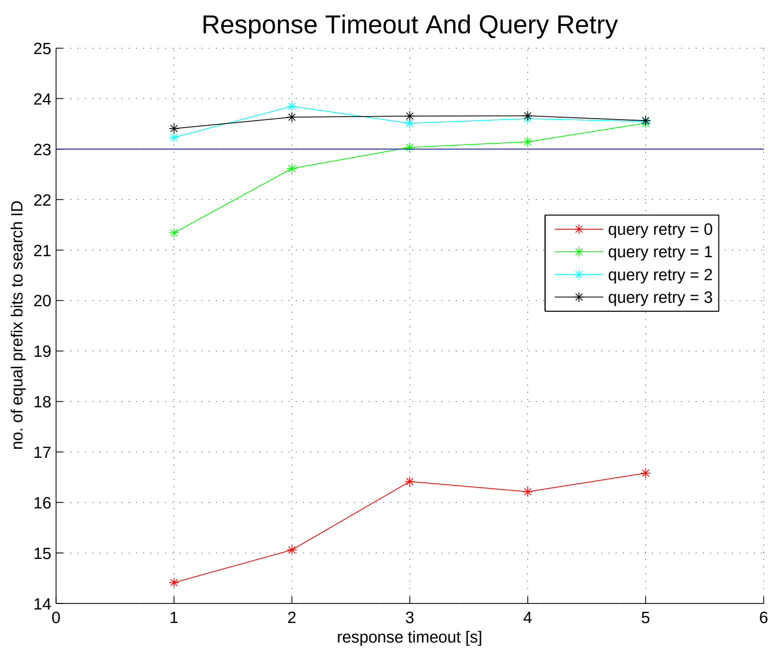


Figure 3.3: The query retry count has to be at least 1.

the writing speed with the amount of data that was stored and the time it took to store that data. The same is true for the reading speed. The resulting writing speed is 24.57 bit/s without the time for the look-up. The value is averaged over all nodes that were writeable. Nodes that can only store a small amount of data have a much lower average writing speed than nodes that store more data. We start to store data in buckets with a large index, i.e. many common prefix bits. Buckets with a large index have less space for storing data than buckets with a smaller index. But we have to send the same number of queries for all buckets, which means that it takes the same amount of time. Therefore storing data in a bucket with a small index is a lot faster. A way to speed up the writing process is to concurrently write to multiple nodes. The resulting reading speed is 95.39 bit/s. Reading is much faster than writing because we can read a full bucket with just a single query whereas for writing 8 messages are required to write a full bucket. One might wonder why reading is not around 8 times faster than writing. This is due to the reading procedure: If a node does not respond to all sent reading queries, the query is resent until the node returns at least one query for each bucket. With this procedure we are able to read 95% of the stored data.

Crawling the DHT. We mainly focused on the storing and reading data but we also did some testing with the crawler. After crawling almost a million nodes we analyzed the collected data. We scanned nodes with the same IP but a

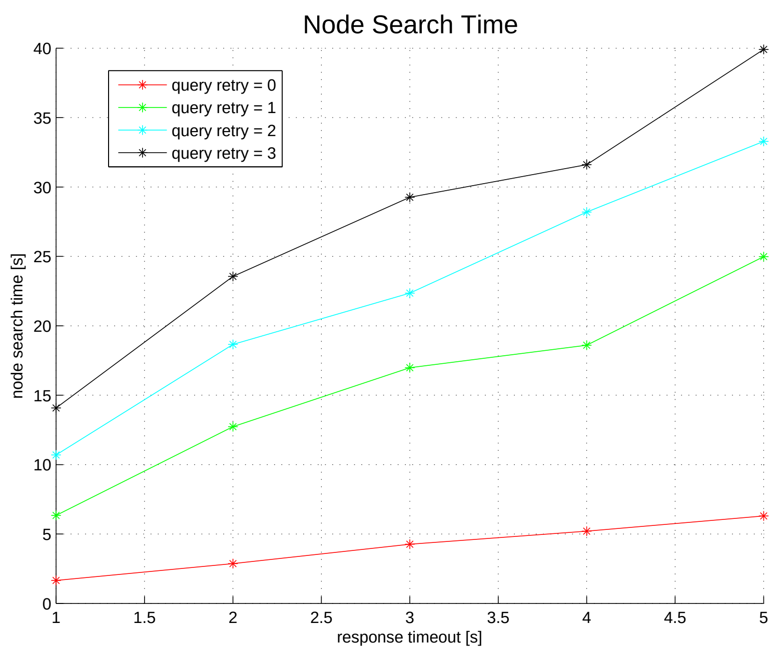


Figure 3.4: The node search time increases with the response timeout.

different port. We found three IPs with around 150 different ports and different node IDs each. It turns out that the found IPs belong to a company called Evidenzia[2]. Evidenzia claims to be one of the leading partners of the software, movie and music industry when it comes to tracing copyright infringements and illegal file sharing activities in peer-to-peer networks.

[2]www.evidenzia.de

Chapter 4

Conclusion and Future Work

In this thesis we have developed three different software tools for the BitTorrent DHT and implemented it in Java. The first one is a framework that can be used to control one or more nodes in the DHT. The framework will also be implemented into BitThief. The second tool is a crawler that is able to collect different kind of information about nodes in the DHT. With the collected information we are be able to reconstruct the DHT. With the third tool we have shown that it is possible to store any kind of data on a computer without the user's knowledge.

In the second part we have analyzed the perfomance of the built software tools and the DHT in general. We have investiagted how fast data can be stored and read from a node in the DHT. We further have calculated the storage size of the whole BitTorrent DHT. We have also determined the values for the response timeout and the query retry parameters to get a consitant and fast node look up.

- The framework can be included into BitThief to make it able to use BitTorrent for swarm discovery.
- A speed up of the data store and read algorithm can be achieved by more concurrent processes.
- The crawler can be used to reconstruct the DHT overlay network. All collected data may further be investigated to find some nodes that do not act according to the protocol. For example, a node may try to take control over certain files.
- The proposed Error Correcting Code may be implemented to add redundancy to make the system more robust if nodes with stored that leave the DHT and cannot be reed anymore.

Bibliography

[1] Mochalski, K., Schulz, H.: Internet study 2008/2009. (2009)

[2] Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the xor metric. (2002) 53–65

[3] Crosby, S.A., Wallach, D.S.: An analysis of bittorrents two kademlia-based dhts. (2007)

[4] Lin, H., Ma, R., Guo, L., Zhang, P., Chen, X.: Conducting routing table poisoning attack in DHT networks. In: International Conference on Communications, Circuits and Systems. (2010)

[5] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. SIGCOMM '01, New York, NY, USA, ACM (2001) 161–172

[6] Jin, X., Chan, S.H.G.: Detecting malicious nodes in peer-to-peer streaming by peer-based monitoring. ACM Trans. Multimedia Comput. Commun. Appl. **6**(2) (March 2010) 9:1–9:18

[7] Varvello, M., Steiner, M.: Traffic localization for dht-based bittorrent networks. In: Networking (2). (2011) 40–53

[8] Stutzbach, D., Rejaie, R.: Improving lookup performance over a widely-deployed dht. In: INFOCOM. (2006)

[9] Locher, T., Moor, P., Schmid, S., Wattenhofer, R.: Free Riding in BitTorrent is Cheap. In: 5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA. (November 2006)

[10] Plank, J.S.: A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. Number CS-96-332 (July 1996)

[11] Schmidt, G., Sidorenko, V.R., Bossert, M.: Collaborative decoding of interleaved reed-solomon codes and concatenated code designs. IEEE Trans. Inf. Theor. **55**(7) (July 2009) 2991–3012