Performance Comparison of DHT based Peer-to-Peer Full-Text Search Systems

Diploma Thesis

Christian J. Schwan

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science

Databases and Distributed Systems
Prof. Alejandro P. Buchmann

Supervisor: Dipl.-Inform. Christof Leng

Abstract

While Peer-to-Peer systems have their advantages in terms of availability and load balancing, it is not clear if they can compete with the speed and result quality of server based full-text search engines. As of yet it is even unclear which Peer-to-Peer full-text search method performs best, or at least has the most advantages in terms of latency, throughput, bandwidth consumption and result quality.
This diploma thesis analyzes two different approaches to Distributed Hash Table based Peer-to-Peer full-text search: parallel search with local result intersection and iterative implementation of recursive search with remote result intersection.
A typical file sharing scenario with files published and queried by tokens, generated from their filenames, is used in combination with the PeerfactSim Peer-to-Peer network simulator to obtain measurements of query processing in terms of traffic, throughput and response time.
The results show that, while not matching the execution time of server based full-text search, Peer-to-Peer algorithms are well able to achieve an execution time fast enough for most applications. However, they strongly depend on the amount of shared data. Providing a data set as big as Wikipedia for full-text search might already be beyond the limits of feasibility on current Peer-to-Peer networks.

Contents

1 Introduction

Purely server based applications are pretty common in networks. While a single server can be easy to implement, the availability and integrity of the service it provides stand and fall with the server itself. Furthermore, it produces running costs due to the necessary maintenance, hardware and power consumption. To avoid the single point of failure problem, and achieve some load balancing, data is usually spread and replicated across multiple servers, which in turn results in even more operating costs. Peer-to-Peer networks spread data across the whole network, thus eliminating the need for a server, and providing availability and resilience against central manipulation. While sending and receiving data from a Peer-to-Peer network is working quite well, the quality of its full text search algorithms is yet to be determined. There are papers on the feasibility of using common Peer-to-Peer full text search algorithms for different types of content, and approaches to calculate or approximate their performance [1][2][3][4], but no measurements under realistic conditions have been made so far. This diploma thesis focuses on comparing the performance of existing Peer-to-Peer full text search algorithms in real world scenarios.
Structured Peer-to-Peer networks are regarded as an improvement over unstructured Peer-to-Peer networks. They use consistent hashing to assign ownership of data to particular peers, while in unstructured networks data is arbitrarily assigned to peers. Implementing a straight forward search is easier on an unstructured network still making use of servers and their local indexing and search functions, but it is difficult to achieve scalability, as it would require multiple servers working together on searches, thus limiting the capacity of unstructured Peer-to-Peer networks on the quality of their servers. A serverless full text search on unstructured networks would require flooding the query through the network, which is suboptimal regarding performance and traffic.
Developing efficient full text search algorithms on Distributed Hash Table (DHT) based structured Peer-to-Peer networks is still an interesting and promising topic. This far there have been several suggestions for improvements [5][2], but not much has been implemented, and even less has been implemented into existing, commonly available Peer-to-Peer applications. This diploma thesis examines the following full text search algorithms, implemented on a Kademlia network, using the PeerfactSim Peer-to-Peer network simulator:

- parallel search with local result intersection
- iterative implementation of recursive search with remote result intersection

Those two algorithms present two opposing types of full text search. The parallel search mainly focuses on a short delay between sending the query and displaying the results, while the recursive search aims to save bandwidth by reducing the transmitted query results.
The algorithms are examined with the use of realistic data. For that purpose a list of files is published across the simulated network and searched for by keywords extracted from their filenames. The time, traffic and bandwidth it takes to perform searches are first analyzed in theory, then implemented and measured.
It is the goal of this diploma thesis to show the performance of currently used Peer-to-Peer full text search algorithms. In Chapter 2 related work is presented by describing the state of the

art on currently used algorithms in detail and giving a brief estimated performance evaluation based on their design. Chapter 3 describes the implementations of the two aforementioned algorithms on PeerfactSim. The framework used to perform the measurements is described in Chapter 4. Chapter 5 then presents the measurements of the simulation and an evaluation on the algorithms' performance.

2 Related Work

To evaluate full-text search algorithms a basis on which they are run has to be chosen. This basis consists of two parts, the overlay network and the simulator on which the evaluation is performed. The network type chosen for this thesis is the Distributed Hash Table, of which multiple real world applications exist. The following subchapters provide information on them, full-text search in general and the PeerfactSim network simulator.

2.1 DHT Networks

In its early days, though being far more versatile, Peer-to-Peer networking was mainly used for file sharing. After several legal issues with copyright violations through those networks its usage was shifted towards less complicated legal grounds, like instant messaging, Internet telephony and online gaming. These partly proprietary Peer-to-Peer networks are still mostly unstructured and make use of servers.

Structured Peer-to-Peer DHT networks are a more modern approach, eliminating the need for servers and running entirely on the network's peers. They however need to maintain a list of known peers to be able to rejoin the network. This list is stored when the client application is shut down. If a peer has not yet been part of the network it needs to perform a bootstrap where one known peer's contact information has to be entered manually. Real world applications either make use of bootstrap contacts and even complete known peers' lists, maintained by some enthusiastic users or provide the needed information through one or several fixed contacts.

2.1.1 Idea behind DHT

The idea behind Distributed Hash Hables is to provide a simple and fast way for retrieving data previously stored into the network.

In a DHT network every peer has an unique identifier, which is used as reference to distribute a hash table across the network. Each peer non-exclusively maintains a subsection of the hash table with hashes similar to its identifier. It is up to the implementation to determine similarity and therefore responsibility for keys. This Distributed Hash Table is available to every peer in the network and can be used like any other hash table.

2.1.2 Churn

In a real world Peer-to-Peer network the number of nodes is never fixed. It is in a permanent state of peers joining and leaving the network. This is called churn. By design DHT networks do not use constantly available supernodes or servers, this causes multiple problems. For once data stored into the network may get lost if all its responsible peers leave the network. This is commonly circumvented through publishing it to multiple peers, which in turn republish it

hourly. If the number of peers is too low or the republish interval is too high the data may still be lost, while contacting too many nodes or republishing too often generates much traffic and negatively affects the performance of the whole network. Another problem arises when a node tries to join the network and all its stored known contacts are no longer available, in which case the same steps as for joining the network for the first time need to be taken. This cannot be circumvented, but a sufficient sized list assures that the client application may be disconnected from the network for several weeks and still be able to rejoin the network. The list's size is only limited by the number of peers contacted during a session and the available storage space on the client computer, but for real world applications such as KAD a list of 200 contacts has proven to be enough.
Churn is commonly measured by either the churn rate, which describes the rate at which peers join and leave the network during a certain period of time in percent, or the mean session length, which describes the average time peers stay connected to the network.

2.1.3 Real World Applications

The most notable categorization of DHTs is the geometrical shape, which interpreting their nodes as points and the connections between two nodes as a line results in. Analyzing this shape gives a quick estimate of the needed maintenance's complexity.

CAN (content adressable network) [6] describes a decentralized DHT Peer-to-Peer infrastructure running on a dynamically positioned virtual n-dimensional Cartesian coordinate space. Every node is responsible for a distinct zone of the coordinate space and together the nodes cover the entire space. When joining the network a node chooses a random position, dividing this zone equally with the node previously responsible for it. Fairness is not guaranteed in this model, as it would require complete knowledge of the network on join.

In Chord [7] nodes form a virtual ring with a 160 bit address space. The SHA-1 hash function is used for object and node identifiers. Nodes keep track of their predecessor and successor, and are responsible for objects between their predecessor and themselves. The so called finger-list contains additional nodes with their distance doubling for each entry, therefore, insertion and lookup costs scale logarithmically with the number of nodes.

Pastry [8] is similar to Chord, its address space is circular forming a virtual ring. Each node keeps track of its immediate neighbors. The main advantage over Chord is an implementation of minimizing the distance messages travel according to a scalar proximity metric.

Tapestry [9] is similar to Pastry. It uses surrogate routing and does not maintain a direct neighbor-list. Instead, it keeps a multiple-level list, where on each level entries' IDs match on the i'th digit of the node's ID. Routing is done by incrementally selecting nodes that match the i'th digit of the desired ID.

In Kademlia [10] nodes are represented as leaves in a binary tree. They communicate using the UDP protocol, and each node is identified through an ID the size of the used hash values. Publishing a file is done by generating its hash and sending this hash and file location information to all nodes whose ID is similar to that hash. Similarity is determined using the XOR metric, which is also used to determine the distance between nodes. Every node maintains one list for every bit of the ID range, representing nodes with a specific distance. Those lists are also called

k-buckets due to their aim of storing up to *k* nodes. To lookup a file a certain amount of *k* nodes closest to the file's ID, typically ten, is determined and then queried simultaneously. They either return a list of their known sources for the file or the closest node they have in their routing table. This way the lookup gradually approaches the responsible nodes.

Overnet is a direct implementation of the Kademlia Peer-to-Peer protocol. Full-text search is not supported, so all file hashes have to be obtained using a server, or following links on web pages. In late 2006 all Overnet related sources were taken down, since then it is only used as communication protocol for decentralized botnets [11].

The Kademlia protocol is also used for trackerless torrents in various BitTorrent implementations. Again, as in Overnet there is no full-text search function.

The KAD protocol [12][13], an implementation based on the Kademlia Peer-to-Peer network protocol, is currently used by file sharing programs such as eMule v0.40+, MLDonkey v2.5-28+, Lphant v.3.50 beta 2+ and aMule. A full-text search using partitioning by keyword is implemented, but limited to one keyword with simple result-filtering using the remaining keywords and additional attributes, such as file size, extension and a variety of meta-data.

Kademlia is robust, scalable and efficient, providing low maintenance cost, thus granting a performant network. For this and its widespread usage in real world applications, it is chosen as platform on which this thesis' search algorithms are implemented and examined.

2.2 Full-Text Search

When using Peer-to-Peer networks there are several ways of implementing full-text search: using one or multiple indexing servers, flooding the search queries over the overlay network, and intersecting index lists stored in the Distributed Hash Table. A search involving servers defies the purpose of Peer-to-Peer networks and is therefore not considered at all by this thesis. Flooding the network with search queries is not effective enough as it causes too much traffic and does not provide a satisfying result quality, therefore it is not considered either. The most effective and scalable search algorithm would be to directly involve the Distributed Hash table for handling search queries. This can be done by distributing an inverted index across the network.

2.2.1 Inverted Index

An inverted index is used for mapping content, such as words, to its locations. For the purpose of full-text search in DHT networks it maps keyword hashes to multiple data hashes, each representing one data previously published to the DHT network. The inverted index is stored directly into the Distributed Hash Table, with the keyword hash representing the key and the list of data items stored as value. It is built and maintained through the publishing algorithm which extracts a list of keywords from a data item, where the method of extraction is up to the implementation, and then, for each keyword stores the file's location into the inverted index by sending it to the peers responsible for the keyword hash. This way every peer maintains an inverted index for all the keywords it is deemed responsible for.

To perform a full-text search query for a single keyword the entries for the corresponding keyword hash are retrieved from the inverted index by querying the peers responsible for the

keyword hash. Queries with multiple keywords are divided into sub-queries for every single keyword and their results are intersected.

As the inverted index can be fully integrated into the DHT network it can be used directly without additional steps. Any additional steps and proceedings can be seen as optimizations and only influence the data transfer between the peers, not the inverted index.

Additional full-text search functionality and features require storing more information for each data, such as the position of the keyword and the surrounding text for citation, a filename for file sharing or the title of the document for content search. It is up to the implementation to determine which additional information is required and whether it is published with the file's contact information or under the file's hash.

2.2.2 Publishing

To provide full-text search nodes need to take an additional step while publishing data to the network. The published data has to be assigned to keywords by its owner, which may be based on the content, the filename, meta-data stored in it or by querying external databases for additional information. Once a list of keywords for the data has been created those keywords are each sent to their respective peers. The amount of peers and how they are determined is up to the implementation. In common file sharing applications a hash for each keyword is generated which is then sent to peers with IDs similar to this keyword hash, together with the hash of the file itself and any desired amount of additional information. As an example the full-text search algorithm of the KAD network determines similarity by the first 8 of its 128 bit IDs, which means every node whose ID has the same first 8 bit as the keyword hash is determined responsible for said hash. When using an inverted index publishing already has a dominant influence on the search result quality, as this step determines how many peers get the file information. Considering the commonly used hourly republishing of data a lot more peers are contacted, and therefore, a lot more traffic and network load are generated for the publishing of one document than for one full-text search query.

It is crucial for a performant network to find a balance between network traffic for publishing and for full-text search. One of the factors involved is the amount of peers contacted while publishing. The more peers knowing a specific document, the easier it is to find. Ideally if every peer responsible for one keyword of the document knows it then the first full text-search query will already return a complete result thus limiting the needed traffic to a very low amount. This in turn would mean that for publishing every responsible peer has to be contacted, leading to a very high amount of traffic, generated for each publish. On the other hand, publishing the file only to some peers results in low traffic, while full-text searches either return only a small percentage of results or consume a lot of time and traffic to finish.

So depending on the publish and search load of a network, the constraints of how many peers to publish to and how many peers to send queries to have to be set. DHT Peer-to-Peer networks already have the infrastructure for handling file hashes and locating sources. Publishing files leads to proportional network load for both functions, publishing the file and publishing the keywords, while searching for resources and full-text search are independent from each other. Therefore no rule can be defined on how much traffic search and publish should take up proportionally to each other if both full-text search and file sharing are implemented on the same algorithms. If the sole purpose of the DHT network is to implement full-text search and key-

word publishing, it is best to use the functions intended for file sharing, and modify them for the whole system if needed.

2.2.3 Querying

Queries are used to retrieve information from a dataset, by filtering out the data matching a given query value. Full-text search uses a query string as filter value, which can contain one or multiple keywords. The implementation decides how a query is processed. To perform a full text query on an inverted index the query string has to be divided into its single keywords and the result lists for each keyword have to be retrieved from the index and intersected.

DHT networks maintain instances of each index entry on multiple peers and most of them are likely to be incomplete. Therefore retrieving multiple result lists for each keyword and merging them before intersecting the result lists for the different keywords is needed to increase the likelihood of a complete query result.

To send a query a lookup for responsible peers is invoked, which in turn updates the routing table and provides the multiple contacts for each keyword. The lookup process is crucial to the query time as it has several timeouts. The process waits for a defined time before returning the peers' contacts if less than the desired number has been found. A further timeout completely aborts a lookup if it has found no contact at all. A basic query terminates after receiving all replies for each keyword query, therefore the predictable execution time of the query is the time when the last reply arrives, unless the algorithm repeatedly invokes the lookup process until the desired number of contacts for each keyword is reached or a query timeout is triggered. The first result can be expected when the last keyword query without a reply receives its first result list. There is no simple way to predict at what point a result quality of 100% will be reached.

2.2.4 Optimizations

To avoid additional network load for multiple keywords KAD only queries the first keyword hash together with a list of additional information, e.g. other keywords to filter the filename with, file size, file type etc. The respective peer then performs a filter operation with the remaining keywords and conditions before returning the final result. This implementation saves bandwidth but reduces result quality – assuming the inverted indices are not complete. A query for each keyword could be sent in order to increase result quality for the trade-off of increasing the traffic again.

Another goal besides reducing the overall traffic needed for a query is to spread it amongst the participating nodes. Recursive search does this by peers forwarding their intersected result list together with the remaining keywords to the next set of responsible peers, and the final peer returning the result to the querie's origin.

Bloom filters can be used to further reduce traffic. The first step, like recursive search, populates the m bit bloom filter with a set of k hash functions for each peer's results, forwarding it again with the remaining keywords. Instead of returning its result to the querie's origin, the final peer starts adding its results to a list which travels back the way it came while it is filled with the peers results. While requiring more CPU time on the peers this algorithm reduces traffic significantly, as in its first round only the Bloom filter of constant size is sent, while in the second round the

search results are added to, not removed from the result list. The only obvious trade-off, asides the CPU time, is the increased minimum traffic for a query.
Asides from optimizing the search algorithm itself, network traffic can be compressed using GZIP or similar performant compression algorithms. Once again CPU time is the trade-off, and with growing network bandwidth even modern CPUs might come to a point where compression shows no performance increase due to the network being faster than the compression/decompression throughput of the system.

2.3 PeerfactSim

PeerfactSim [14] is a discrete event based Peer-to-Peer network simulator written in Java and designed with the requirements for benchmarking platforms in mind. These requirements are modularity, underlay network model, user behavior, resource model, service model, easy experiment setup, scalability and documentation.

The simulator consists of five layers (see Table 2.1), with the first layer being the simulation engine, followed by the network wrapper, the overlay layer, the application layer and the user layer.

User
Application
Overlay
Network Wrapper
Simulation Engine

Table 2.1: PeerfactSim functionality layers

The network wrapper is a simplified combination of the underlying OSI layers and is able to simulate message delivery times according to a latency model. It uses the geographical distance between the start and end point as well as the available bandwidth to calculate the latency, taking into account additional delay from processing through intermediate systems. It is able to model network characteristics like retransmissions due to package loss, package damage or out-of-order packages and congestion.

The overlay layer encapsulates details of overlay communication protocols and specific overlay related algorithms and operations such as message routing and network maintenance. PeerfactSim comes with a multitude of fully implemented, interchangeable overlay networks to choose from such as Chord, CAN and Kademlia. Depending on the overlay network each peer can be assigned different roles such as router, maintainer, indexer and cacher.

The application layer holds the code for the application running on each peer. PeerfactSim comes with a simple file sharing application.

The user layer is responsible for creating nodes according to the configuration files, with all crucial configurations being stored in a XML document. All nodes are located on an Euclidean plane with their coordinates generated either randomly all over the plane or in predefined zones. The nodes furthermore make use of a churn generator, deciding the user-controlled participation dynamics in the network. The version used for this thesis came with two churn generators, the uniform-random and the mixed log-normal churn generator.

PeerfactSim comes with multiple predefined benchmarking sets, usefull when evaluating overlay network implementations. The *ideal* set takes place after nodes have succesfully joined the network, they run multiple operations and wait for the appropriate stabilization phase to finish after each operation. In the *scaling* scenario groups of nodes join the network one group at a time, publish their data and start random *get(key)* operations. This is done until all groups have joined the network. In the *unstable* scenario a significant number of peers leaves the network in a short time interval or performs a large number of overlay operations. In the *failures* scenario peers randomly fail leading to message loss.

3 Implementation

This thesis' implementation starts on top of the Kademlia implementation for PeerfactSim and adds full-text search capability to the existing Kademlia overlay network providing parallel and recursive search algorithms.
As a first step the SHA-1 algorithm is included for generating hashes used by the query and publish algorithms. Furthermore the option to add multiple message handlers is added, and a new Kademlia full-text search message type is defined and redirected exclusively to a new full-text search message handler, while the original message handler receives all other message types. This way Kademlia full-text search messages are excluded from the automated message replication and re-send routines.
The static class *DocumentProvider()* with a list of strings representing filenames is used for providing publish and query data in case no data is given as parameter. When publishing *getPublish()* is called and returns one of a defined amount of variations for each filename, all its keywords with three or more letters and a file hash generated from the filename using the SHA-1 algorithm. The variations are created by adding a numeral at the end of the filename which is not returned as a keyword but results in a different file hash. This step is later used to analyze the result quality as every query should return the same number of variations.
Published data is marked by a counter, which in turn allows the *getQuery()* function to only return keywords from filenames for which all variations have been published. A second counter assures that the query data is provided in the same order it is stored in the list, and once all published files have been queried it wraps around and returns keywords for the first filename.
Basically no further modifications on either the network layer or Kademlia are needed at this point.

Listing 3.1: DocumentProvider()

```
1  static class DocumentProvider(){
2
3    publishes   = 0;
4    queries     = 0;
5    variation   = 0;
6    documents = ['filename1',
7                 'filename2',
8                    ...
9                 'filenameX'];
10
11   public list getPublish(){
12     returnList = [];
13     if (publishes < documents.length);
14       fileName   = documents[publishes] + variation;
15       fileHash   = SHA-1(fileName);
16       keywords[] = filename.split('␣');
17       for (x = 0 to keywords.length){
```

```
18          if (keywords[x].length < 3 ||
19              keywords.countEntries(keywords[x]) > 1){
20             keywords[x].remove;
21           }
22        }
23        variation++;
24        if (variation >= 10){
25          variation = 0;
26          publishes++;
27        }
28        returnList = [fileName,
29                      fileHash,
30                      keywords];
31      }
32      return returnList;
33    }
34
35    public list getQuery(){
36        keywords[] = documents[queries].split('␣');
37        for (x = 0 to keywords.length){
38          if (keywords[x].length < 3 ||
39              keywords.countEntries(keywords[x]) > 1){
40             keywords[x].remove;
41           }
42        }
43        queries++;
44        if (queries > publishes) queries = 0;
45        return keywords;
46    }
47
48  }
```

3.1 Publish

The publish algorithm is designed to process one single keyword with its respective file hash at a time, though it can be called with a list of keyword and file hash pairs. It makes use of a queue in form of the *PublishQueue()* class and a periodical processing function *processQueue()*. It exactly limits the number of puplishes according to the desired publish limit provided through the simulation setup. On execution without parameters the publish data is obtained from the *DocumentProvider()* class. Each file hash is stored into the queue with all of its keywords through the queue's *add()* function and if it isn't already running *processQueue()* is activated. The processing function periodically extracts lists of all unfinished publish commands from the queue by calling the *get()* function. The queue stores and returns the publishes in the form of pairs of one filehash and one keyword. The *processQueue()* function processes the lists by sending

each publish through an ID lookup to its respective peers. This is done by using the preexisting Kademlia *send()* function. On receiving the publish message an automated reply is sent directly through the network layer with the *send()* function and the received data is stored into an inverted index each node maintains. When a certain amount of publish replies is received and stored into the queue using the *update()* function the publish terminates successfully. Once all publishes have terminated the query returns an empty list which in turn stops the processing function's periodical execution until it is triggered again by a new publish.
In common DHT networks all files are republished hourly to achieve high availability. With simulation time in mind this functionality is not implemented but instead the publish command is executed with churn disabled, guaranteeing an optimal replication of the published data. Furthermore the simulated realtime only covers four hours with churn disabled for the first hour. The sources deterioration in that time period is negligible thus making repeated publishing unnecessary.
The following pseudo-code provides a brief insight into the publishing algorithm and its components.

Listing 3.2: Publish – message handler

```
public void messageHandler(IncomingEvent message){
  if (message.type==publish){
    localIndex.storeSource(message.keywordHash, message.fileHash);
    NetLayer.send(message.sender,
                  message.content,
                  type=publishReply);
  } else if (message.type==publishReply){
    publishQueue.update(message);
  }
}
```

Listing 3.3: Publish – publish call

```
public void publish(String publishString){
  if (DocumentProvider.publishes < Setup.publishLimit){
    if (publishString == ''){
      publishQueue.add(DocumentProvider.getPublish());
    } else {
      publishQueue.add(publishString);
    }
    if (!periodic.isActive){
      periodic();
    }
  }
}

public void processQueue(){
  publishList[] = publishQueue.getPublishes();
  if (!publishList.isEmpty){
    for (x = 0 to publishList.length){
```

```
      Kademlia.send(publishList[x]);
    }
    Simulator.schedule(periodic(), 1000ms);
  }
}
```

Listing 3.4: Kademlia send()

```
public void send(String publish){
  nodes[] = overlay.lookup(publish.keywordHash);
  for (x = 0 to nodes.length){
    overlay.send(nodes[x],
                 publish.keywordHash,
                 publish.sourceHash,
                 type=publish);
  }
}
```

Listing 3.5: Publish – publish queue

```
class PublishQueue(){
  publishIndex = new Array[fileHash,
                           keywordHashes[],
                           contacts[][]];

  public void add(String publish){
    publishes[] = split(publish.keywords);
    for (x = 0 to publish.keywords.count){
      if (!publishIndex.exists(publish.fileHash, publishes[x])){
        publishIndex.add(publish.fileHash, publishes[x]);
      }
    }
  }

  public void update(IncomingEvent message){
    publishIndex.addPeer(message.fileHash,
                         message.keyword,
                         message.sender);
  }

  public list get(){
    publishList = [];
    for (x = 0 to publishIndex.count){
      if (!publishIndex.publish(x).minPeersPublishedTo){
        publishList.add(publishIndex.publish(x).publishData);
      }
    }
```

```
28      return publishList;
29    }
30  }
```

3.2 Search

The search algorithms are designed to use a list of keywords. If no list is given the *DocumentProvider()* is invoked to obtain one. These keywords are then stored into a queue and retrieved as individual single word queries by a periodic processing function.
The number of desired queries is forced by comparing the according value from the simulation setup file to the one from the *DocumentProvider()* class. Furthermore a query is only performed if the querying node is connected to the network which in turn means the execution of a query is aborted if the node itself is affected by churn and therefore not connected to the network.
This way the queries run on already churned nodes are ignored. The cases of a node loosing connection during a query, or a queried node being affected by churn are not influenced by this method.

3.2.1 Parallel Search with Local Result Intersection

The main functionality of the parallel search is to request the result lists for each keyword at the same time. The lists are then intersected so only results returned for every keyword end up in the final result.
For implementing a parallel search algorithm queries are separated into sub-queries of single keywords when added to the queue with the *add()* function. The original query string with all its keywords is stored as reference. When *getQueries()* is called by the periodic processing function the queue returns a list of all sub-queries' hashes with less than n replies whose query is still active. A query is determined inactive when a certain time has passed, a result limit has been reached or all sub-queries have received as many replies as set in the simulation setup.
Each sub-query hash in that list is then sent to its n respective peers using the Kademlia *send()* operation. On receiving a query a reply is sent through the network layer containing the corresponding data from the local inverse index. If no data for that query is found then an empty result is returned.
The reply is forwarded to the query queue, which in turn stores the new sources for each sub-query. When asked for queries the queue checks whether one of the conditions for terminating each query has been reached. Queries are not actively terminated, they are simply not included in the return statement of *getQuery()* anymore.
The following pseudocode provides a brief insight into the parallel query algorithm and its components.

Listing 3.6: Parallel search – message handler

```
1 public void messageHandler(IncomingEvent message){
2   if (message.type==query){
3     NetLayer.send(message.sender,
4                   message.query,
```

```
                    localIndex.getResults(message.query),
                    type=reply);
  } else if (message.type==reply){
    queryQueue.update(message);
  }
}
```

Listing 3.7: Parallel search – query call

```
public void query(String query){
  if (DocumentProvider.queries < Setup.queryLimit){
    if (query == ''){
      queryQueue.add(DocumentProvider.getQuery());
    } else {
      queryQueue.add(query);
    }
    if (!periodic.isActive){
      periodic();
    }
  }
}

public void periodic(){
  queryList[] = queryQueue.getQueries();
  if (!queryList.isEmpty){
    for (x = 0 to queryList.length){
      Kademlia.send(queryList[x]);
    }
    Simulator.schedule(periodic(), 1000ms);
  }
}
}
```

Listing 3.8: Parallel search – query queue

```
class QueryQueue(){
  public void add(String query){
    subQueries[] = split(query);
    queryIndex.add(query, subQueries);
  }

  public void update(IncomingEvent message){
    queryIndex.subQueryUpdate(message.query, message.results);
  }

  public list getQueries(){
    queryList = [];
```

```
13      for (x = 0 to queryIndex.count){
14        if (!queryIndex.query(x).desiredRepliesPerSubQuery &&
15            !queryIndex.query(x).runtimeExceeded &&
16            !queryIndex.query(x).resultLimitReached){
17          queryList.add(queryIndex.query(x).getOpenSubQueries);
18        }
19      }
20      return queryList;
21    }
22  }
```

3.2.2 Recursive Search with Remote Result Intersection

Traditional recursive search sends out its query string to the node responsible for the first keyword. This node then adds its results and passes it on to the node responsible for the next keyword which intersects the attached result list with its own. This goes on until the last keyword is reached. The nodes responsible for the last keyword then return their intersected result lists to the querying node where they are added to the final result list. As a slight optimization the keywords can be ordered ascending according to the responsible node's result list sizes for the keywords. This safes some traffic as the smallest result list gets passed on and is possibly even reduced through intersection.

As Kademlia is basically an iterative network the recursive search cannot be implemented straight forward. It would be possible to add the functionality to send out recursive queries for an incoming query, but with churn, lookup timeouts and querying the n closest nodes this would result in a rather slow algorithm, prone to returning empty results due to churn somewhere along the way or flooding the network with an exponential amount of messages due to the n replication. Therefore this algorithm is implemented in an iterative way.

The algorithm can be separated into four phases (see Figure 3.1). In phase 0 all sub-queries are retreived from the queue via the *getQueries()* function and sent out simultaneously to their n respective peers using the Kademlia *send()* operation. Instead of returning a result list the reply to a query simply includes the number of results the respective peer's local index holds for that hash value. From this point forward all communication is done through the network layer, therefore the only considerable delay will come from the initial ID lookups.

With a certain desired amount of replies per sub-query or with reaching a timeout and at least a minimal amount of replies per sub-query the *update()* function returns the boolean value *true*. As a result the algorithm enters phase 1 where the sub-queries are ordered in ascending order by the amount of results the respective peers hold, and the query is no longer only processed through a timed event but it is also triggered for each incoming event through the return value of the *update()* function. The main step of this phase is to retrieve the first sub-query from the queue with the *getIPQueries()* function and request all sources for it through the network *send()* operation.

Receiving the sources for the first sub-query the algorithm now enters phase 2, in which all following sub-queries will be sent out with a list of their previous sub-querie's results if there are any. Each incoming reply therefore triggers the sending of the sub-queries following this

reply's respective sub-query, with exception for those replies belonging to the last sub-query. On receiving a query the list of results sent alongside it is intersected with the local inverted index and a reply with the list of the remaining file hashes is sent.
When a time limit or a results limit has been reached or every node found in phase 0 has sent a reply the algorithm enters phase 3 for the corresponding query which results in this query not being processed anymore and therefore terminates it.
The following pseudocode provides a brief insight into the iterative implementation of the recursive query algorithm and its components.

Listing 3.9: Recursive search – message handler

```
public void messageHandler(IncomingEvent message){
  if (message.type==queryCount){
    Network.send(message.sender,
                 message.query,
                 localIndex.getResults(message.query).length,
                 type=reply);
  } else if (message.type==queryFirst){
    Network.send(message.sender,
                 message.query,
                 localIndex.getResults(message.query),
                 type=reply);
  } else if (message.type==query){
    Network.send(message.sender,
                 message.query,
                 intersect(localIndex.getResults(message.query),
                           message.sources),
                 type=reply);
  } else if (message.type==reply){
    if (queryQueue.update(message)){
      periodic();
    }
  }
}
```

Listing 3.10: Recursive search – query call

```
public void query(String query){
  queryQueue.add(query);
  if (!periodic.isActive){
    periodic();
  }
}

public void periodic(){
  queryList[]   = queryQueue.getQueries();
  queryIpList[] = queryQueue.getIpQueries();
  if (!queryList.isEmpty){
```

```
    for (x = 0 to queryList.length){
      Kademlia.send(queryList[x]);
    }
  }
  if (!queryIpList.isEmpty) {
    queryList[] = queryQueue.getIpQueries();
    if (!queryList.isEmpty){
      for (x = 0 to queryList.length){
        Network.send(queryList[x].IP,
                     queryList[x].query,
                     queryList[x].sources,
                     queryList[x].type);
      }
    }
  }
  if ((!queryList.isEmpty ||
      !queryIpList.isEmpty) &&
      periodic.isNotYetScheduled){
    Simulator.schedule(periodic(), 1000ms);
  }
}
```

Listing 3.11: Recursive search – query queue

```
class QueryQueue(){
  public void add(String query){
    subQueries[] = split(query);
    if (!queryIndex.exists(query)){
      queryIndex.add(query, subQueries);
    }
  }

  public boolean update(IncomingEvent message){
    if (message.sources.isNull &&
        !message.sourcesCount.isNull){
      queryIndex.subQueryUpdate(message.query,
                                message.sender,
                                message.sourcesCount,
                                type=contact);
    } if (!message.sourcs.isNull){
      queryIndex.subQueryUpdate(message.query,
                                message.sender,
                                message.sources,
                                type=sources);
    }
    x = queryIndex.queryContains(message.query).index;
    if (((queryIndex.query(x).enoughContacts &&
```

```
24          queryIndex.query(x).timeLimitReached) ||
25          queryIndex.query(x).desiredContacts) &&
26          queryIndex.query(x).phase = 0){
27        queryIndex.query(x).phase = 1;
28        queryIndex.query(x).sort();
29      }
30      return (queryIndex.query(x).phase > 0);
31    }
32
33    public list getQueries(){
34      queryList = [];
35      for (x = 0 to queryIndex.count){
36        if (queryIndex.get(x).phase == 0){
37          queryList.add(queryIndex.getOpenSubQueries);
38        }
39      }
40      return queryList;
41    }
42
43    public list getIPQueries(){
44      queryList = [];
45      for (x = 0 to queryIndex.count){
46        if (queryIndex.query(x).runtime > Setup.queryTimeLimit ||
47            queryIndex.query(x).results > Setup.queryResultLimit ||
48            queryIndex.query(x).allNodesReplied){
49          queryIndex.get(x).phase = 3;
50        } else if (queryIndex.get(x).phase == 1){
51          for (z = 0 to queryIndex.query(x).subQuery(0).contactsCount){
52            queryList.add(queryIndex.query(x).subQuery(0).contacts(z),
53                          queryIndex.query(x).subQuery(0).query,
54                          type=queryFirst);
55          }
56          queryIndex.query(x).phase = 2;
57        } else if (queryIndex.query(x).phase == 2){
58          for (y = 1 to queryIndex.query(x).subQuery.length){
59            if (queryIndex.query(x).subQuery(y-1).sources.length > 0 &&
60                !queryIndex.query(x).subQuery(y).contacts.allQueried){
61              for (z = 0 to queryIndex.query(x).subQuery(y).contacts.
62                            length){
63                queryList.add(queryIndex.query(x).subQuery(y).
64                              contacts(z));
65              }
66            }
67          }
68        }
69    }
```

```
70    }
71    return queryList;
72  }
```

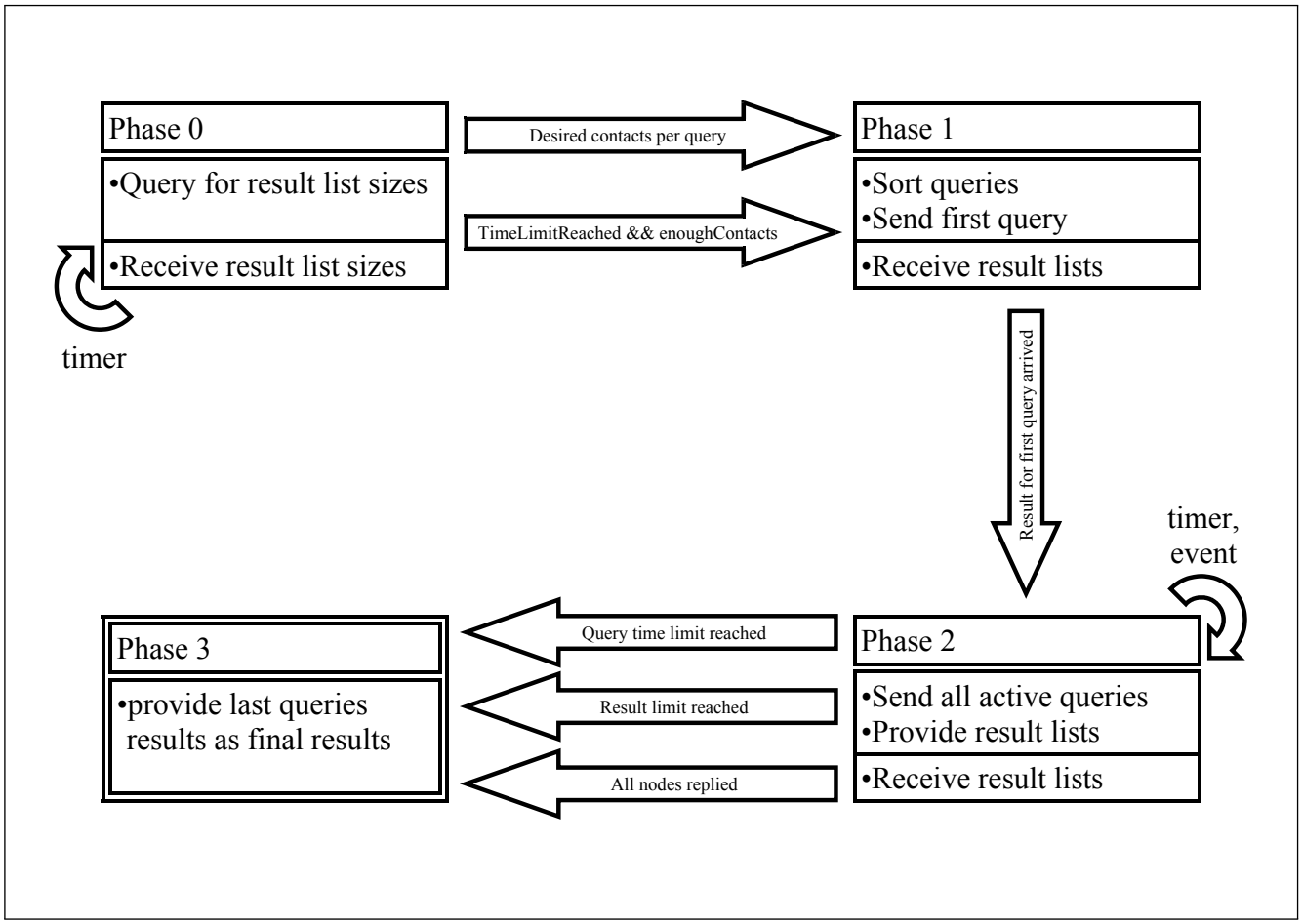

Figure 3.1: Recursive search – phases

4 Analysis Framework

The analysis framework consists of two parts, one being the simulator configuration and the other one the actual measurement of certain values.

To provide realistic measurements the configuration will be set to represent real world Kademlia networks as close as possible. While this way most settings are predetermined they will still be described in the following section to provide a deeper understanding of what they do and the effect changing them would have on the network.

4.1 Simulator Configuration

The course of the simulation itself during the simulated time period is controlled through a set of files consisting of *kadFTS.properties*, *kadFTS.XML* and *kadFTS-actions.dat*. Furthermore the used full-text search algorithm is configured through a separate setup file conveniently placed in a static *Setup()* class. This is done to keep this thesis work as far separated as possible from the original simulator source code.

The XML file contains the settings for the size of the network, the simulated time period and the simulated real times at which churn is enabled and measurement starts while the actions file calls node functions for subgroups at a defined time. The setup class for the full-text search algorithms determines the runtime of each query, a results limit, a query timeout, the time and contact limits for the recursive search algorithm, the amount of variations per file and the interval at which each nodes queues are processed.

The simulation can be divided into several chronological steps, with the first being the connection of the clients. This takes place in the first 30 minutes of simulated real time.

The next step, publishing of the full-text search data, takes place in the following 30 minutes.

At this point, 60 minutes into the simulated real time, churn is enabled for the whole network, and the measurement is started.

The size of the network will be set to different values for multiple simulation runs, to provide a view on the influence the network size has on the performance and an outlook on the scalability of the full-text search algorithms.

The most important part of this simulation, running queries, is performed from 80 minutes on with a rate of 1 query per node per minute, up until the desired total amount of queries has been finished. This rate is controlled using the network size and setting the time-range in the *actions.dat* accordingly. While in file sharing networks query response times of several seconds do not pose a problem, studies show [15] that most people will determine a regular web search to have returned no results after just under 6 seconds of no results coming in.

The total runtime of the search may take longer, and while file sharing networks run their queries for 45 seconds no user would wait that long for a web search. At this point the aforementioned 6 seconds would be a good time limit for the total runtime of a query, but in real life considerably longer response times are to be expected in the Kademlia network. Therefore a query runtime limit of 25 seconds is chosen.

The rate at which nodes repeat ID lookups for queries is set to one second. This increases the queries performance in case the already running lookups are steering into a timeout. According to later simulations this happens with a probability of 98% in at least one sub-query for every query performed, thus increasing the whole result time. Sending multiple lookups ensures that non responsive peers are dropped from the routing table according to the overlay network settings, which using this papers simulation settings is long before the upper tolerable timelimit of 6 seconds has been reached.
The iterative implementation of the recursive search algorithm needs two additional constraints. The first one is a timeout after which phase 1 of the query algorithm will be entered in case at least one results bearing contact per sub-query has been found. This timeout is however rendered obsolete if the query algorithms desired contacts per sub-query to enter phase 1 is set to 1. This can in theory dramatically reduce the result quality, but after some testing it turned out to work well in combination with requesting the full result list for new contacts belonging to the first sub-query. Therefore the only important value for the recursive search is *desiredContacts* located in the *Setup()* class.
The following tables provide a brief overview on the aforementioned files' settings.

size	1,000 peers 2,500 peers 5,000 peers 10,000 peers 20,000 peers
churnStart	60 minutes
meanSessionLength	60 minutes
measurementStart	60 minutes
finishTime	240 minutes

Table 4.1: kadFTS.XML

SimulatorQueryLimit	1,000 queries
fileVariations	10
resultsMax	300 results
queryRuntime	25 seconds
queueTimer	1 seconds
desiredContacts	1

Table 4.2: Setup() class

0m-29m	connect
30m-59m	publish
80m	query

Table 4.3: kadFTS-actions.dat

4.1.1 Physical Network Layer

The PeerfactSim Peer-to-Peer network simulator already implements a regular UDP/IP network protocol and does not allow direct manipulation of its properties. As any changes in the IP protocol would result in an incompatibility with existing UDP/IP networks they are of no interest to this simulation.

The IP network itself is simulated as a worldwide network. PeerfactSim takes into account the delay between subnetworks according to their geometric distance. It is possible to customize each subnetwork's node count thus allowing to simulate denser regions alongside regions with less infrastructure. The predefined network clusters in PeerfactSim are already very close to current real world network density charts [16] and will therefore not be modified.

North America	35.4%
East Asia	30.4%
Japan	15.6%
Europe	11.6%
Oceania	3.2%
Latin America	1.6%
Middle East	0.8%
Balkans	0.6%
South East Asia	0.4%
Africa	0.4%

Table 4.4: Network population

As most private Internet connections use asymmetric connections this simulation will do so as well. At the time of this thesis commonly available private Internet connections in Germany range roughly between 2,000 and 16,000 kilobits per second download and 200 and 1,000 kilobits per second upload bandwidth. Other countries may have slower connections available and widely used, so even if Kademlia has proven to be quite resilient against local bottlenecks by preferring the best performing known nodes in general a realistic connection speed has to include slower connections as well. To date the global average connection speed is approximately 1.7 Mbps [17]. Therefore and because of the fact that PeerfactSim only uses one bandwidth setting for all nodes a download bandwidth of 200 kilobytes per second paired with an upload bandwidth of 100 kilobytes per second is chosen and should provide a realistic environment for the simulation.

Churn is another important parameter for this simulation. The PeerfactSim version used in this thesis allows to choose between two churn models, the static and the exponential churn model. While static churn is easy to simulate, affecting whole subgroups of the network and disconnecting them from the network, its behavior is far from realistic. The exponential churn model affects individual nodes, making it the better choice for this simulation. PeerfactSim allows to setup churn through the mean session length of participating nodes, which will be set to 60 minutes.

4.1.2 Overlay Network Layer

The overlay network layer consists of three fundamental parts, its ID space, the overlay structure and the lookup algorithm making use of it.

The length of the n-bit identifier used in overlay networks is crucial to their usage, as it determines the upper limit of nodes in a network. Furthermore hashes, used to publish and query data, have to be of the same length. While on IPv4 networks a 32 bit ID would be enough to cover the whole IP range the needed 32 bit hash algorithm would be too prone to collisions. Yet the ability to cover the IP range wouldn't be enough, as with a dense population of the network this would consume a considerable amount of time and traffic just to find an unused ID, always risking that it is not unused but the current holder is offline at the time being. To allow as many nodes as IPv6 can manage an 128 bit identifier would be enough, and collisions in a 128 bit hash algorithm are already acceptably low. Nonetheless real world Kademlia applications use a 160 bit ID space, to reduce aforementioned time and traffic for finding a free ID. At an ID length of 32 bit more then the IP address space it can even be safely assumed that no lookup for a free ID is needed at all and it can simply be generated through hashing some system values. There is still the possibility to reassign a new ID if an ID collision occurs. As the 160 bit ID used in Kademlia works very well in real world applications it will be used for this simulation as well.

The virtual structure of the Kademlia overlay network is a binary tree with an order of 2. The hierarchy depth of the tree is 1, meaning that every node only knows his direct children. Increasing the hierarchy depth of the tree would have benefits for lookup operations because of the additional known nodes, but it would require considerably more maintenance as every node would need to keep updated on all of his children's children down to the level of hierarchy. This structure is the core of real world Kademlia applications and will therefore not be changed.

The lookup operation is set to locate the $k = 10$ nearest neighbors to a given 160 bit key. Accordingly the routing table stores 10 nodes per bucket with a replacement cache of an additional 10 nodes. Increasing k will result in increased traffic and memory consumption, while a lower k might reduce the result quality. Again the given value has proven itself as a good trade-off between traffic and result quality in real world applications and will be used in this simulation.

A stale counter is used to filter out non responsive nodes from the routing table after the second time of failing to respond. In this case the non responsive node will be replaced by the nearest node from the replacement cache. Lowering the stale counter to 1 would have the effect that a node experiencing churn would drop his entire routing table, putting him into a situation where it is hard to rejoin the network. Increasing it on the other hand would lead to more lookup timeouts. A stale counter of 2 has proven to work quite well, although real world Kademlia applications tend to drop their entire routing table when disconnected from the network for too long. As the routing table buckets are refreshed with an interval of 1 hour the offline time after which the entire routing table is empty can be determined to 2 hours for this setup, unless the client application detects its offline state and inhibits refreshing until it has rejoined the network. This could not be avoided by increasing the stale counter, only the necessary offline time would increase.

To avoid a contactless state at the beginning of the simulation, which would require the implementation of a bootstrap algorithm, each node joins the network with 100 nodes already in the routing table. This number is high enough to ensure every node manages to join the network, and by far lower than the number of contacts in the routing table after the first 60 minutes

of simulated realtime. It therefore can be safely assumed that it has no effect on the full-text search algorithms.
After retrieving the 10 closest nodes from the routing table the lookup operation contacts all of them to make sure they are alive. This is done for 3 nodes at a time, with a lookup timeout of 2 seconds and a timeout of 45 seconds for the entire lookup operation. Again the predefined values have proven to work well and are used in real world Kademlia applications where reducing the lookup timeout has a negative influence on the result quality, while a higher timeout only slows down the query algorithms. Due to the relatively small simulated network sizes used for this thesis simulations and the fact that the network is not used by any other application even a lookup timeout of 1 second would work well (see Figure 4.1 – simulated on 1,000 nodes with non mentioned settings according to this chapter), though it would not be a good representation of real world DHT networks.

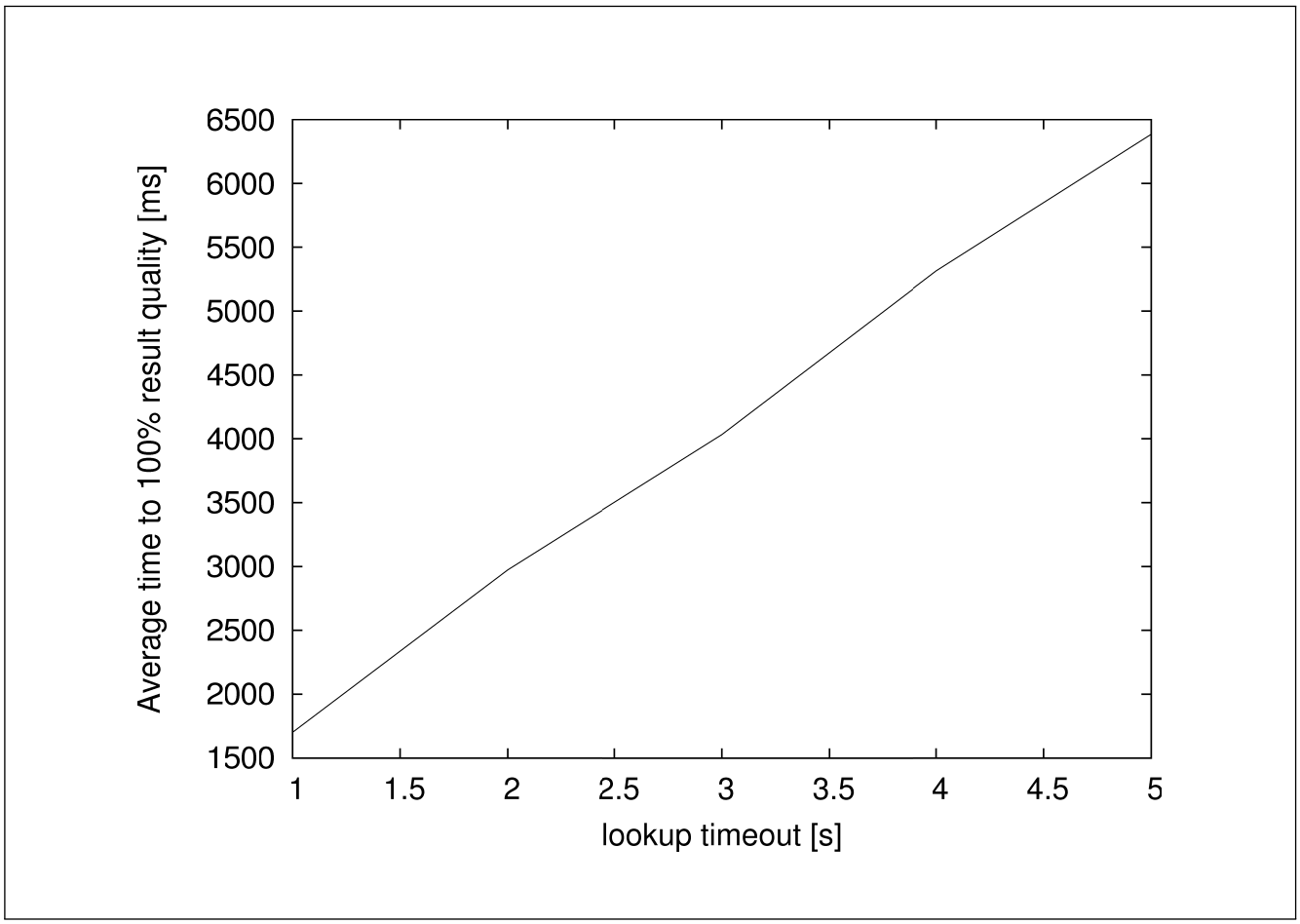

Figure 4.1: Overlay network layer – lookup timeout

The following table provides a brief overview on the overlay network settings used for this thesis.

ID_Length	160 bit
Btree	2
K	10 contacts
Replacement_Cache_Size	10 contacts
Stale_Counter	2
Refresh_Interval	1 hour
Hierarchy_Depth	1
Hierarchy_Btree	2
Initial_Routing_Table_Contacts	100
Alpha	3 seconds
Lookup_Message_Timeout	2 seconds
Lookup_Operation_Timeout	45 seconds

Table 4.5: Overlay network configuration

4.2 Measurement

The main attributes to be measured are the query response times and result quality. It is important to measure at which point the first results come in to determine if the algorithms would still be applicable for a regular full-text search. The result quality of each single query as well as the average result quality are important to determine the usefulness of Peer-to-Peer searches for different applications. While low result quality is acceptable for full-text search on file sharing networks other applications like web search require a higher – if not 100% – result quality.

Another important measurement value is the generated network traffic. This determines whether an algorithm would be fit for different kinds of applications. While on server based web search the sole limit is the server with usually a quite low amount of traffic per query, on Peer-to-peer networks the traffic is shared among the peers and queries take up considerably more traffic as multiple peers have to be contacted. From the traffic the overhead can be extracted. This again is an interesting value to measure as it gives an insight on the practicality of the used Peer-to-Peer search algorithms and the overlay network itself.

The network load during query operations is another important factor as it determines the maximum amount of peers that can send a query during a period of time before the networks performance collapses. For this purpose various simulation runs with different query rates will be evaluated. The main simulations will then be run with a non problematic query rate to rule out network congestion and focus on the algorithms performance and practicallity.

The PeerfactSim Kademlia package comes with a variety of predefined measurement analyzers. The one interesting for this simulation is the hourly overlay traffic analyzer.

In addition a set of new measurement analyzers is implemented for the query response time until the first reply arrives, until a result quality of 100% has been reached and until all K nodes per sub-query have sent a reply. Furthermore the in- and outgoing traffic from the view of the application for each query is stored and an average is calculated at the end of the simulation. The IP traffic and the overlay network traffic are measured separately and therefore can be used to calculate the generated overhead.

Multiple simulations will be run for different network sizes to gain an insight on scalability and an estimate of how larger networks would affect the simulation.

4.3 Shared Data

The data shared into the network, for the purpose of full text search, consists of keyword hash and file hash sets. To simplify the process simple strings are used as files (see Table 4.6). The keywords are the single words in that string, with three or more letters, and the file hash is generated by hashing the whole string. For each keyword, a keyword hash and the file hash are sent to the peers responsible for the keyword hash, and stored into the inverted indices of those peers.
To ensure an exact upper limit for the results of full text search queries, each string is used 10 times, with a numeral added at the end. This numeral is not extracted as a keyword, therefore all 10 files are published with the same keywords. However, they produce different file hashes, due to the variations in the string, which results in 10 different files being found by the queries. The following tables show the list of strings used for the *DocumentProvider()* and the result list sizes for single keyword queries with 10 variations of each file where one result takes up 160 bit.

1922 - Nosferatu	1987 - Robocop
1940 - Pinocchio	1987 - The Running Man
1942 - Bambi	1988 - A Fish Called Wanda
1946 - Song of the South	1988 - A Nightmare On Elm Street 4 - The Dream Master
1960 - The Little Shop Of Horrors	1988 - Beetlejuice - CD1
1966 - Batman	1988 - Beetlejuice - CD2
1970 - Aristocats	1988 - Die Hard
1971 - A Clockwork Orange - CD1	1988 - Halloween 4 - The Return Of Michael Myers
1971 - A Clockwork Orange - CD2	1988 - Hellraiser 2 - Hellbound
1972 - Fist Of Fury	1988 - The Blob
1973 - Enter The Dragon	1988 - Who Framed roger Rabbit
1977 - Pete's Dragon	1989 - A Nightmare On Elm Street 5 - The Dream Child
1977 - Star Wars Episode IV - A New Hope	1989 - Back To The Future II
1978 - Game Of Death	1989 - Batman - CD1
1978 - Halloween	1989 - Batman - CD2
1979 - Alien	1989 - Halloween 5
1979 - Stephen King - Salems Lot	1989 - Indiana Jones And The Last Crusade
1980 - Friday the 13th	1989 - Pet Sematary
1980 - Star Wars Episode V - The Empire Strikes Back	1989 - UHF
1981 - Game Of Death II - CD1	1990 - Back To The Future III
1981 - Game Of Death II - CD2	1990 - Die Hard 2 - Die Harder
1981 - Halloween 2	1990 - Gremlins 2 - The New Batch
1981 - Heavy Metal	1990 - Hardware
1981 - Indiana Jones And The Raiders Of The Lost Ark	1990 - Robocop 2
1982 - Halloween 3 - Season of the Witch	1991 - A Nightmare On Elm Street 6 - Freddys Dead - The Final Nightmare
1982 - The Thing	1991 - Highlander 2 - The Quickening
1983 - Star Wars Episode VI - Return of the Jedi	1991 - Hook
1984 - A Nightmare On Elm Street	1991 - Terminator 2 - CD1
1984 - Beverley Hills Cop	1991 - Terminator 2 - CD2
1984 - Gremlins	1992 - Aladdin
1984 - Purple Rain	1992 - Alien 3
1984 - Revenge of the Nerds	1992 - Batman Returns
1984 - Terminator	1992 - Dead Alive
1985 - A Nightmare On Elm Street 2 - Freddys Revenge	1992 - Hellraiser 3 - Hell On Earth
1985 - Back To The Future	1992 - Honey I Blew Up The Kid
1985 - Legend	1992 - Stephen King - The Lawnmower Man
1985 - Lifeforce	1993 - Jurassic Park
1986 - Aliens	1993 - Nightmare Before Christmas
1986 - Big Trouble In Little China	1993 - Robin Hood - Men In Tights
1986 - Ferris Bueller's Day Off	1993 - Robocop 3
1986 - Highlander	1993 - Twin Warriors
1986 - Jumpin Jack Flash	1994 - A Nightmare On Elm Street 7 - New Nightmare
1986 - Little Shop Of Horrors	1994 - Ace Ventura Pet Detective
1986 - The Fly	1994 - Beverley Hills Cop III
1987 - A Nightmare On Elm Street 3 - Dream Warriors	1994 - Highlander 3 - The Final Dimension
1987 - Beverley Hills Cop II	1994 - The Lion King
1987 - Hellraiser 1	1994 - Tiny Toons - How I Spent My Vacation
1987 - Honey I Shrunk The Kids	1995 - Ace Ventura - When Nature Calls
1987 - Lethal Weapon	1995 - Bad Boys
1987 - Revenge of the Nerds 2 - Nerds In Paradise	1995 - Batman Forever

Table 4.6: DocumentProvider() – strings

230 - The	20 - Pet	10 - Sematary	10 - Fury
90 - 1988	20 - Orange	10 - Season	10 - Friday
80 - Nightmare	20 - Nerds	10 - Salems	10 - Framed
80 - 1989	20 - Man	10 - Running	10 - Forever
80 - 1987	20 - Jones	10 - Robin	10 - Fly
70 - Street	20 - Indiana	10 - Returns	10 - Flash
70 - Elm	20 - III	10 - Rain	10 - Fist
70 - 1992	20 - Horrors	10 - Raiders	10 - Fish
70 - 1986	20 - Honey	10 - Rabbit	10 - Ferris
60 - the	20 - Hard	10 - Quickening	10 - Enter
60 - 1994	20 - Gremlins	10 - Purple	10 - Empire
60 - 1984	20 - Freddys	10 - Pinocchio	10 - Earth
50 - Halloween	20 - Final	10 - Pete's	10 - Dimension
50 - CD2	20 - Dragon	10 - Park	10 - Detective
50 - CD1	20 - Die	10 - Paradise	10 - Day
50 - Batman	20 - Dead	10 - Off	10 - Crusade
50 - 1993	20 - Clockwork	10 - Nosferatu	10 - Christmas
50 - 1991	20 - Beetlejuice	10 - Nature	10 - China
50 - 1990	20 - And	10 - Myers	10 - Child
50 - 1981	20 - Alien	10 - Michael	10 - Calls
40 - Back	20 - Ace	10 - Metal	10 - Called
40 - 1985	20 - 1982	10 - Men	10 - Bueller's
30 - Wars	20 - 1980	10 - Master	10 - Boys
30 - Terminator	20 - 1979	10 - Lot	10 - Blob
30 - Star	20 - 1978	10 - Lost	10 - Blew
30 - Robocop	20 - 1977	10 - Lion	10 - Big
30 - Revenge	20 - 1971	10 - Lifeforce	10 - Before
30 - New	10 - roger	10 - Lethal	10 - Batch
30 - Little	10 - Witch	10 - Legend	10 - Bambi
30 - King	10 - Who	10 - Lawnmower	10 - Bad
30 - Hills	10 - When	10 - Last	10 - Ark
30 - Highlander	10 - Weapon	10 - Kids	10 - Aristocats
30 - Hellraiser	10 - Wanda	10 - Kid	10 - Alive
30 - Game	10 - Vacation	10 - Jurassic	10 - Aliens
30 - Future	10 - UHF	10 - Jumpin	10 - Aladdin
30 - Episode	10 - Twin	10 - Jedi	10 - 1983
30 - Dream	10 - Trouble	10 - Jack	10 - 1973
30 - Death	10 - Toons	10 - How	10 - 1972
30 - Cop	10 - Tiny	10 - Hope	10 - 1970
30 - Beverley	10 - Tights	10 - Hook	10 - 1966
30 - 1995	10 - Thing	10 - Hood	10 - 1960
20 - Warriors	10 - Strikes	10 - Hellbound	10 - 1946
20 - Ventura	10 - Spent	10 - Hell	10 - 1942
20 - Stephen	10 - South	10 - Heavy	10 - 1940
20 - Shop	10 - Song	10 - Hardware	10 - 1922
20 - Return	10 - Shrunk	10 - Harder	10 - 13th

Table 4.7: DocumentProvider() – result lists

5 Measurements and Analysis

The PeerfactSim Peer-to-Peer network simulator uses a *seed* value with which its function for generating random values is initialized. This way all simulations, while using a certain degree of randomness are reconstructible. It is not possible however to provide all simulations with the same random pattern as besides the seed value it also depends on other simulation settings and values. To obtain comparable results multiple simulations with differing seed values are performed for each desired setting and an average is calculated.
This chapters results are obtained by calculating the average of five simulations with seed values ranging from 0 to 4. The version of PeerfactSim used for this thesis did not provide the possibility to run simulations with much more than 20,000 nodes. It also required disproportinal amounts of time if the number of published documents was increased noticeably. The simulation runtimes using the settings from chapter 3 were within an acceptable range on an Intel Core2 Quad with 12 MB cache running at 4 GHz and on 8 GB RAM (see Figure 5.1).

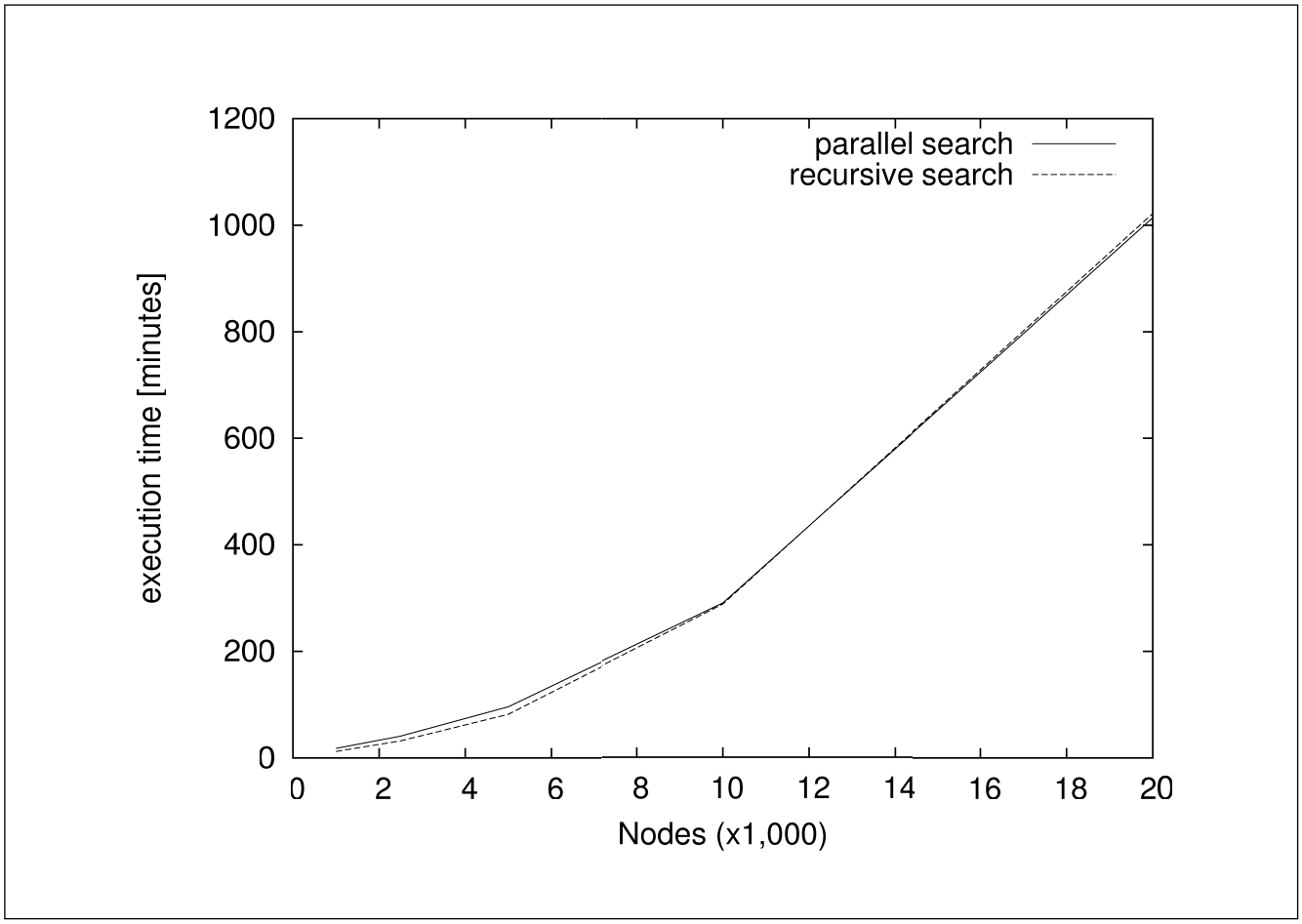

Figure 5.1: Simulation execution time

5.1 Result Quality

The result quality of the queries in this paper's evaluations is at a constant 100%. This is due to the publishing without churn, indexing nodes returning to the network after a disconnect and the relatively small size of the network.
Publishing to k nodes on a churned network can be achieved by repeating the ID lookup and publish algorithm, until the desired amount of nodes has been reached. This implementation, however, leads to a massive increase in the total simulation runtime, compared to simply publishing without churn, while leading to the same initial situation for the queries to be performed on. If queries were to be performed for not finished publishes and during the publishing phase, then there would be files published to $n < k$ nodes, and a higher probability for an ID lookup returning a set of k nodes that does not include any of the respective n nodes. This first factor for the high result quality is therefore depending on the test scenario.
In real world applications, the published file information deteriorates and needs regular republishing. This happens when nodes holding the information leave the network. On PeerfactSim a churned node is simply cut from the network and rejoins it after a period of time. Its inverted index is therefore never completely lost. This factor is mainly depending on the implementation, as data deterioration can be implemented. But as common implementations use republishing to circumvent this problem, this just means that it would require more simulation time to achieve the same results for the query algorithms. Furthermore, with the commonly used republish interval of 60 minutes being enough, and the queries in this simulation taking place in a time slot of 10 minutes, the expectable deterioration should not have any noticeable effect on the results. If so desired, it would be simpler to take a given average rate of deterioration and calculate a new result quality with it.
Finally, the network sizes used for this thesis' simulations are not big enough to cover the chance of finding an entirely different set of k nodes, when querying for a file, than the k nodes that file has been published to. This last factor is a limit resulting from the simulator and the hardware used for the simulations.

5.2 Network Load

The network load is directly influenced by the rate at which nodes send their queries. There is a point at which the result quality and performance suffer gravely due to congestion. The network load measurements are performed using the parallel query algorithm and a network size of 1,000 nodes. According to later measurements the rate at which congestion poses a problem when using the recursive query algorithm with this thesis test scenario is about 30% higher.

In regards to the time needed to complete a query the best rate is one query per node per minute (see Figure 5.2) while at rates up to 10 queries per node per minute all queries are processed completely (see Figure 5.3). However congestion is not the only factor influencing the performance. At rates lower than 0.1 queries per node per minute the query time rises significantly. This happens due to the fact that there is no traffic in the simulated network besides the queries and Kademlias periodic routing table refresh. In a churned network this hourly refresh makes sure all nodes stay connected, but it does not provide optimal conditions to perform lookups. If a node sends a query every minute this automatically updates parts of its routing table through usage and also provides all the contacted peers with this node's contact information, thus improving their routing tables in case they didn't know this node before.

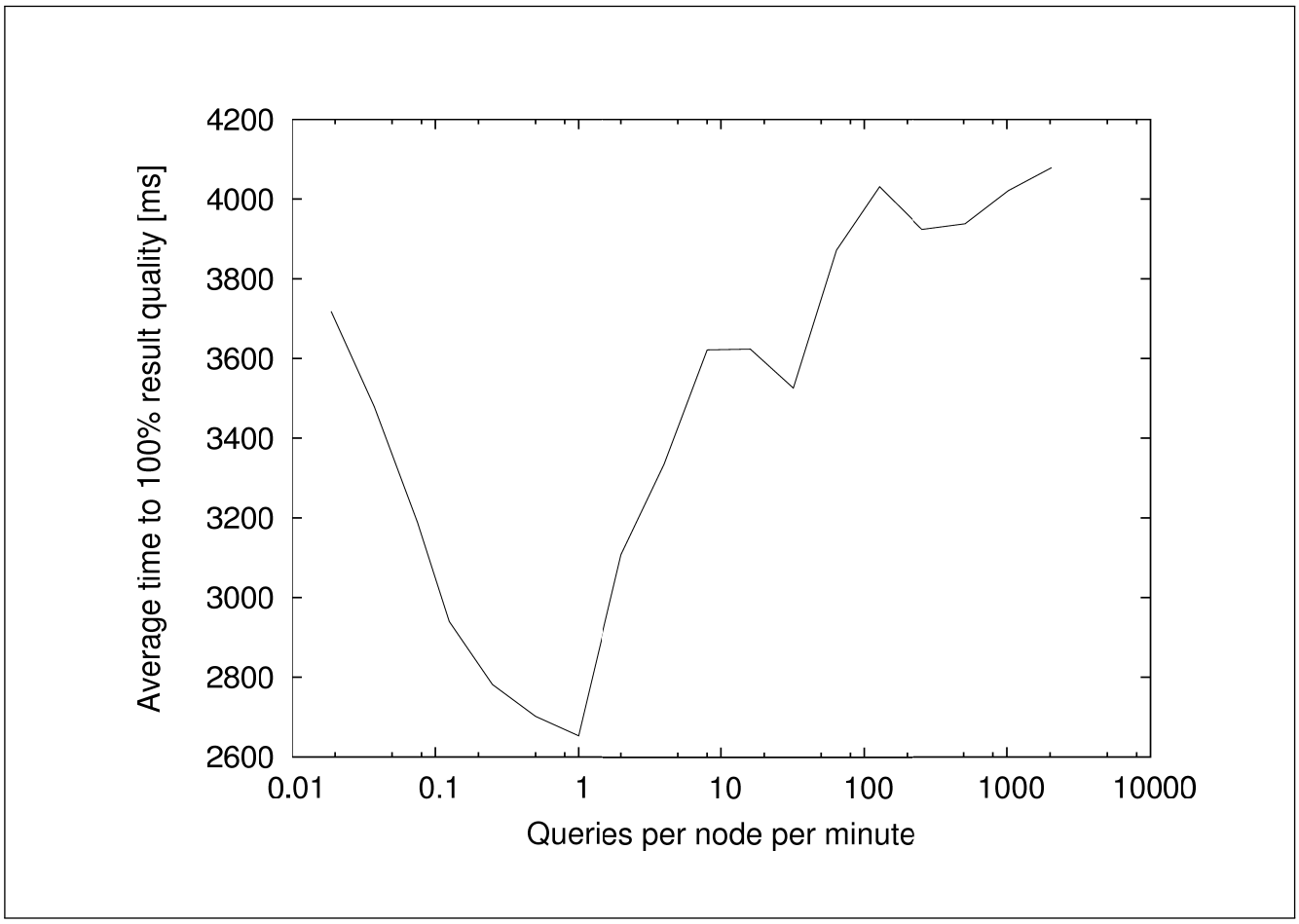

Figure 5.2: Network Load – query execution time

The drop in result quality at about 10 queries per node per minute happens due to network congestion. With less queries processed the network load decreases and therefore query times actually improve slightly at rates between 20 and 50 queries per node per minute. It would require many more simulations with query rates between 10 and 1,000 queries per node per minute to determine if this effect results in a wave patern or is completely random.
As Kademlia scales very well it is safe to assume an upper limit of 10 queries per node per minute even for bigger networks. It is also safe to assume that the average user will evaluate the results for one query before starting a new one, which in addition to the query execution time makes that rate very improbable in real world applications. On the contrary it very probable the average query rate will be way lower than even the 1 query per node per minute, but as the resulting increase in lookup time is related to each nodes local routing table this only means the first few queries will be slow, after that the routing table is up to date and query execution times will be similar to the ones measured in this thesis simulations.

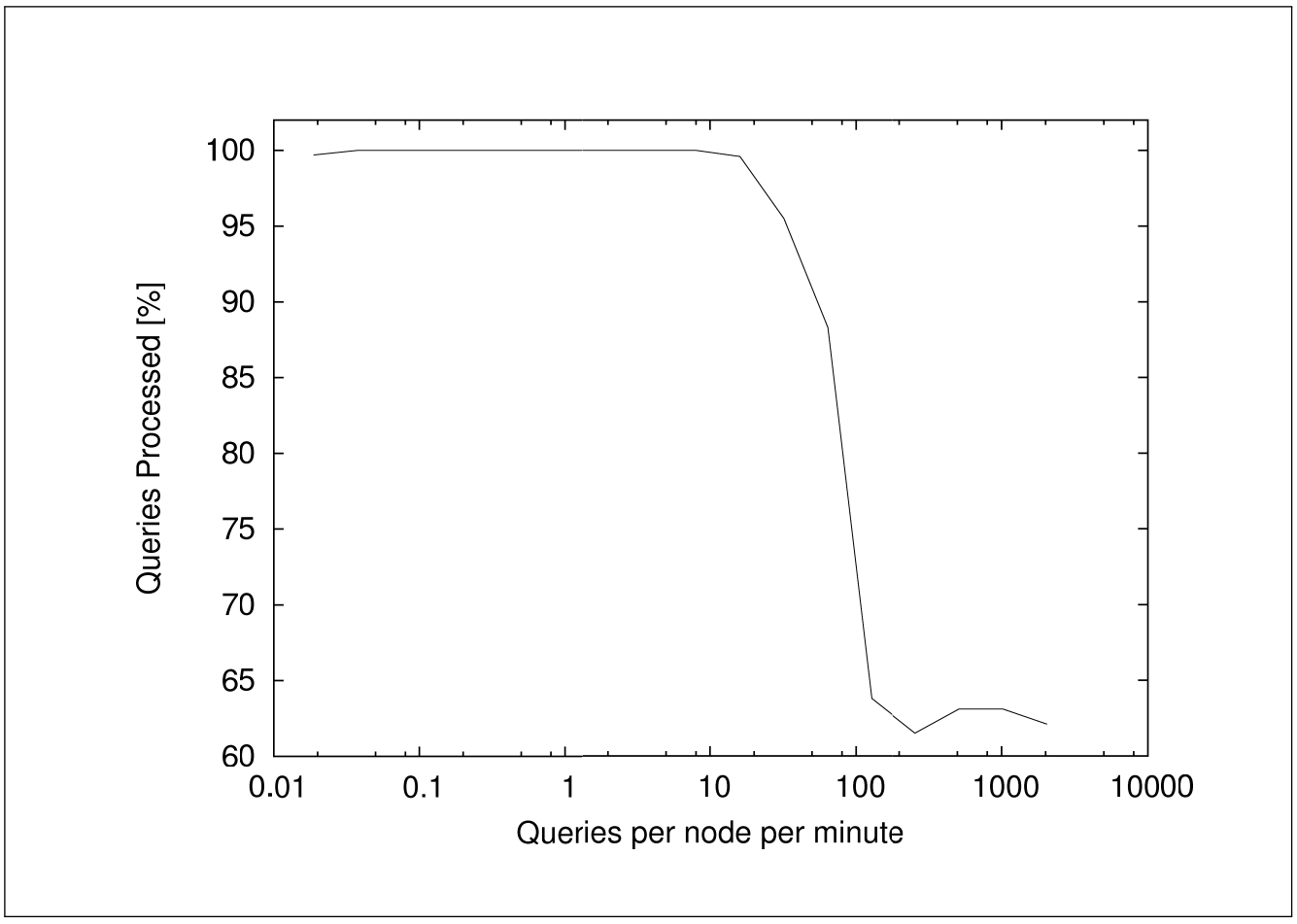

Figure 5.3: Network load – queries processed

The rate of 1 query per node per minute chosen for the following simulations turns out to be a good choice as lowering the result quality by loosing queries to congestion and beforehand increasing the result time are unwanted side effects when measuring the query algorithms performance.

5.3 Churn

Churn has a big impact on the overall performance and result quality and is more or less a random factor. PeerfactSim allows to set a mean session length for the nodes. This value describes the average time a node stays in the network before disconnecting for a period of time. Raising it has direct influence on the query time as it increases the possibility of finding all k nodes for a lookup while decreasing the possibility of not finding any node. Lookups that return all k nodes usually finish a lot faster than the lookup timeout, while in a highly churned network some lookups may take longer than the lookup timeout to return any result at all.
A good value for the mean session length should neither worsen the result time nor provide too optimistic conditions. The value of 60 minutes chosen for this evaluation leads to an average query time of about 2,800 ms (see Figure 5.4), which with a 2 second lookup timeout results in a realistic 800 ms for contacting the queried nodes and transferring the result lists.

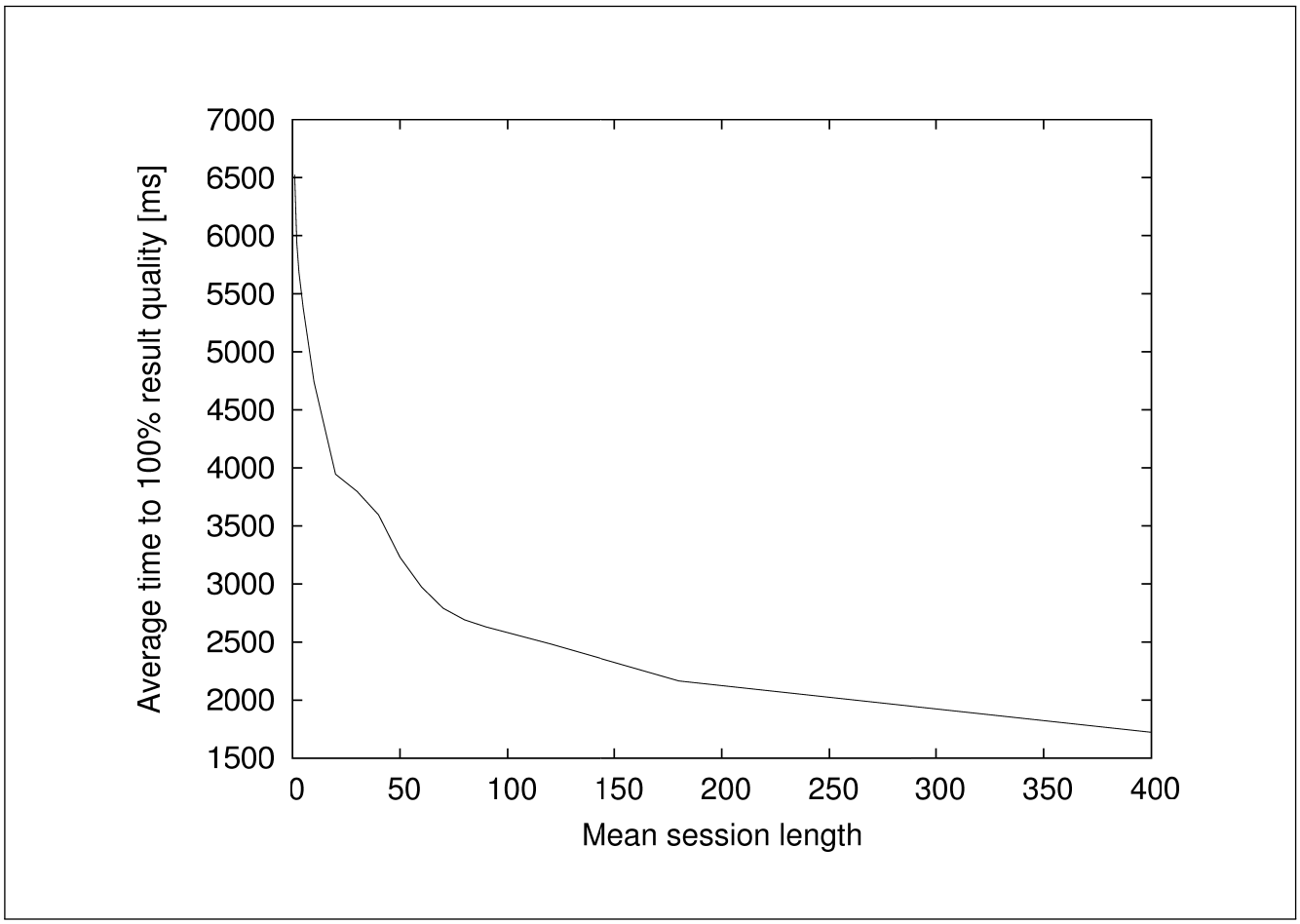

Figure 5.4: Churn – mean session length

Measuring the queries not executed due to churn at the querying node provides an insight on PeerfactSims churn algorithm which, as expected, delivers somewhat random results even when using the average of five runs with differing seed values (see Figure 5.5). This is important to correctly interpret later measurements.

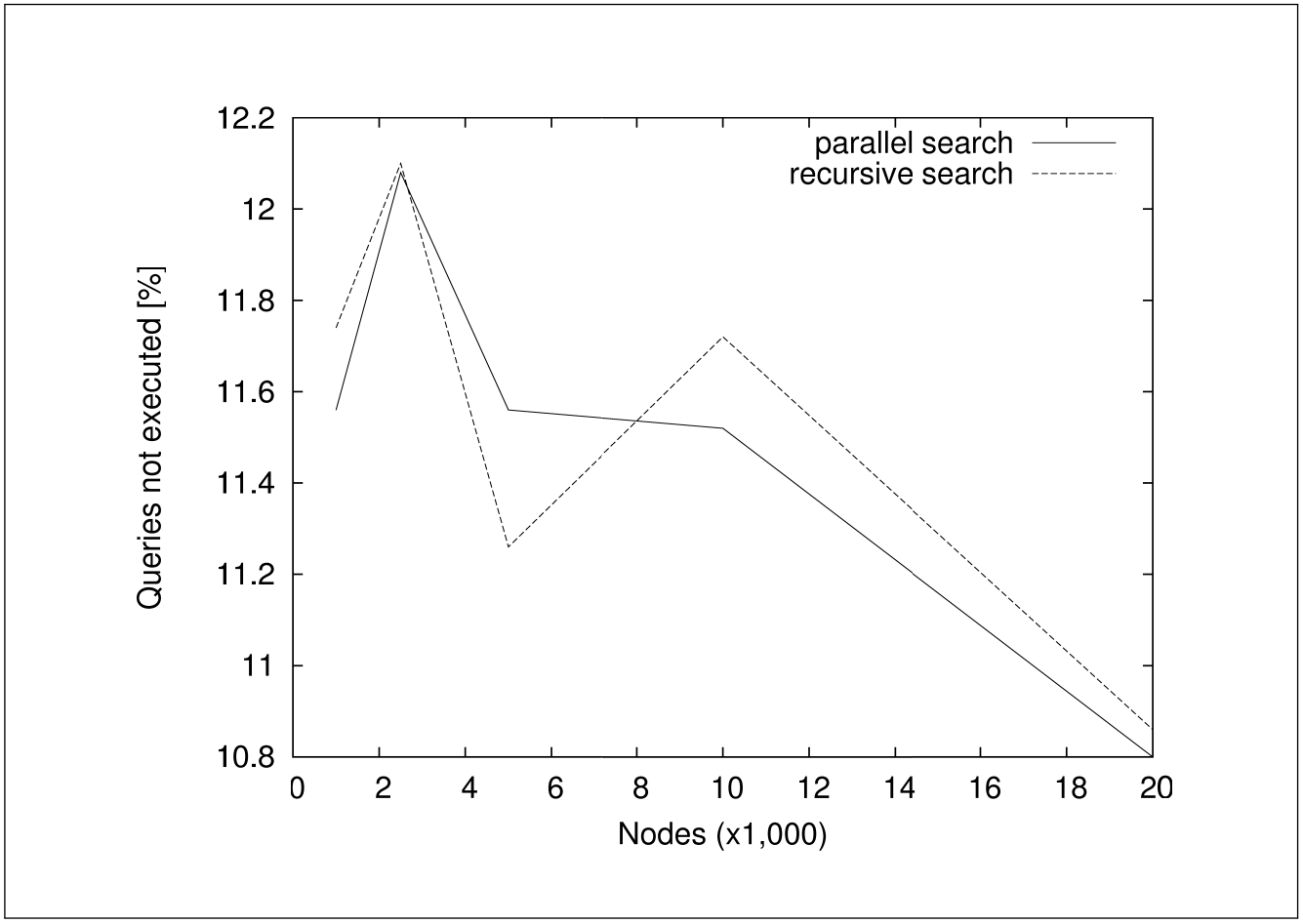

Figure 5.5: Churn – queries not executed

5.4 Lookups

In average the Kademlia ID lookups return about 8.9 contacts (see Figure 5.6). This result increases by about 0.07 contacts when using the recursive search algorithm. The cause for this slight difference is the way PeerfactSim calculates random values and uses them to generate network churn, as both algorithms perform the same initial lookup and only differ in the following IP traffic.
Both graphs rise proportionally to the network size but show a certain pattern instead of direct proportionality. This is again due to the way PeerfactSim generates churn, as both graphs rise indirectly proportional to their respective graphs from Figure 5.5. The slight rise is due to the effectiveness of Kademlia in larger networks. Lookups are more likely to find a full set of responsible nodes if there are more responsible nodes to pick from. The network sizes used for this simulation don't even come close to the size limits of real world Kademlia networks, therefore no negative effect was to be expected when increasing the network size.
In real world Peer-to-Peer applications network churn is not a fixed value, and the quality of ID lookups varies greatly. The overall ID lookup quality could be averaged to one single value, but it is more useful to define a range of values in which it will be with a high probability. As a result it can be stated that ID lookups in a Kademlia network with the dimensions and settings used for this evaluation return an average count of contacts ranging between 8.85 and 9.1.

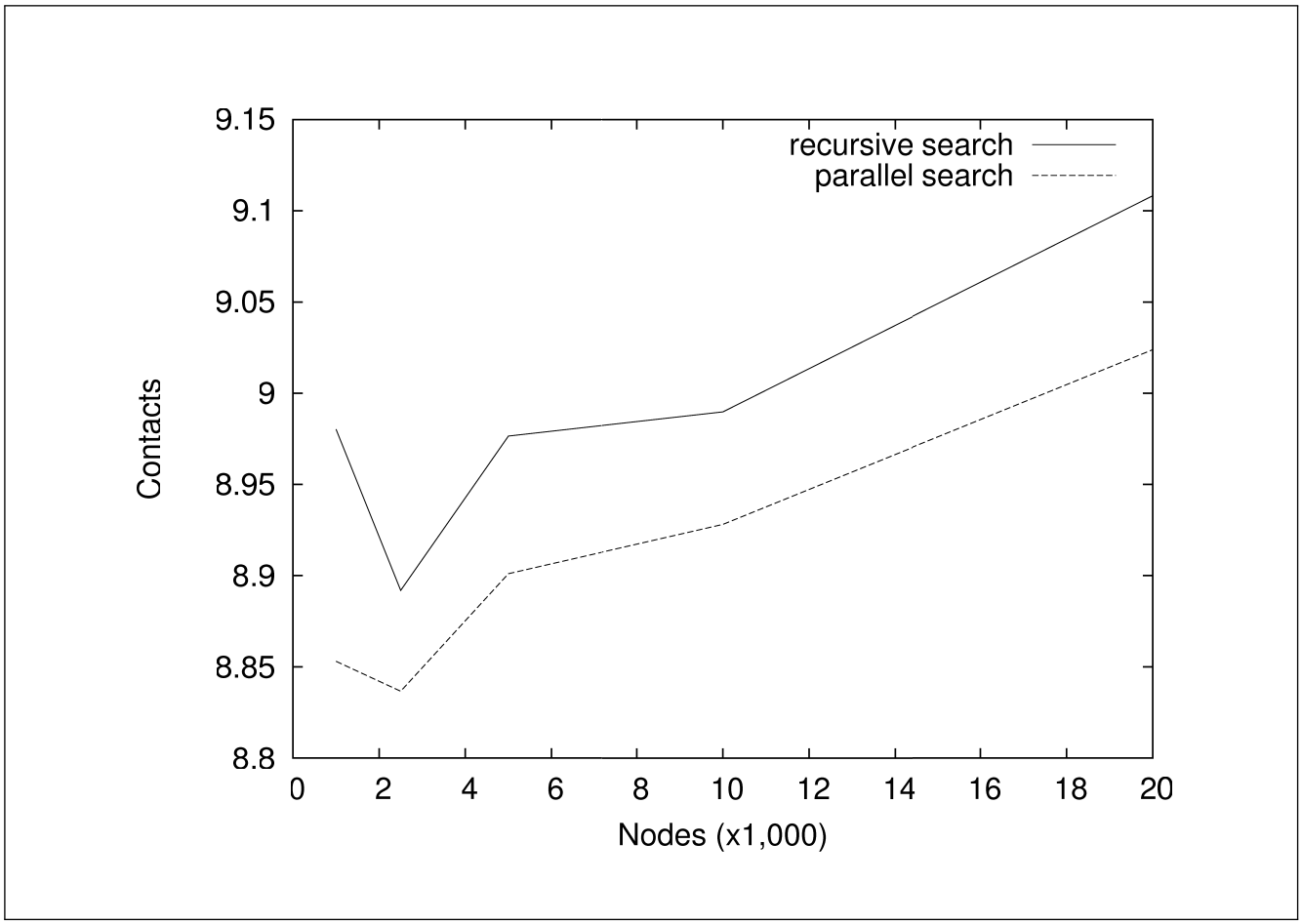

Figure 5.6: Lookup – contacts

5.5 Traffic

One major factor for a query algorithm's performance is the traffic it generates. With the relatively small dataset used for this evaluation the parallel query algorithm needs slightly less than 110 kbyte to execute, whereas the recursive query algorithm needs less than 85 kbyte, which is an improvement of about 25 kbyte (see Figure 5.7). The graphs representing both algorithms appear to be converging towards a certain value with rising node count.
It is necessary to further analyze the traffic in order to approximate the required traffic for bigger network sizes as it consists of overlay traffic generated through the ID lookups and IP traffic for transfering the result lists.

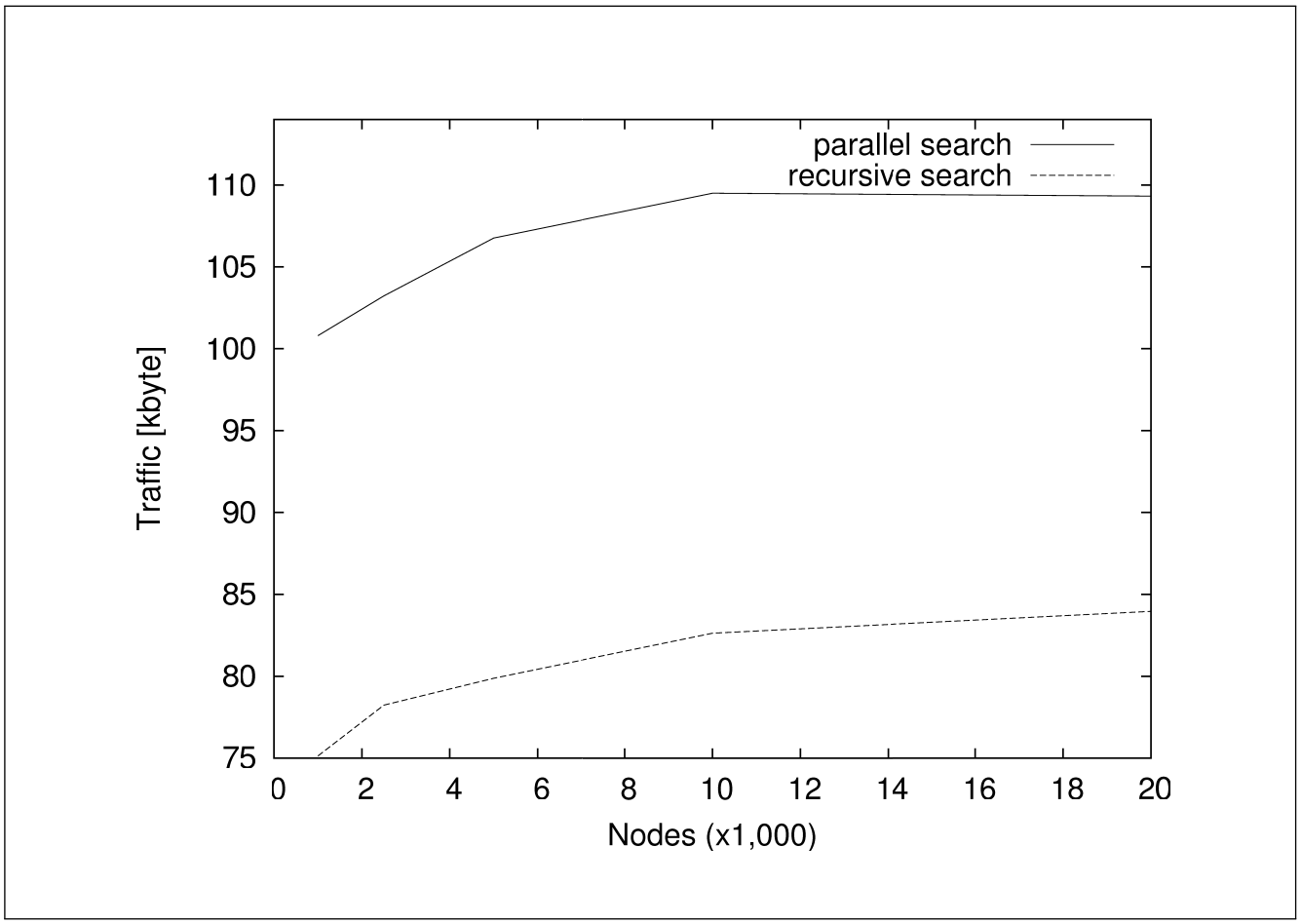

Figure 5.7: Traffic – combined traffic

The required overlay traffic is about identical for both query algorithms (see Figure 5.8). This is to be expected as they both start with ID lookups while everything after that is handled through the regular IP network protocol. Both algorithms repeat each of their ID lookups once every second until it returns at least one contact.
It is important to note that the required overlay traffic increases by roughly 6 kbyte when increasing the network size tenfold. With this it is possible to predict a required overlay traffic per query of about 70 kbyte for a network size of 1,000,000 nodes for both query algorithms.

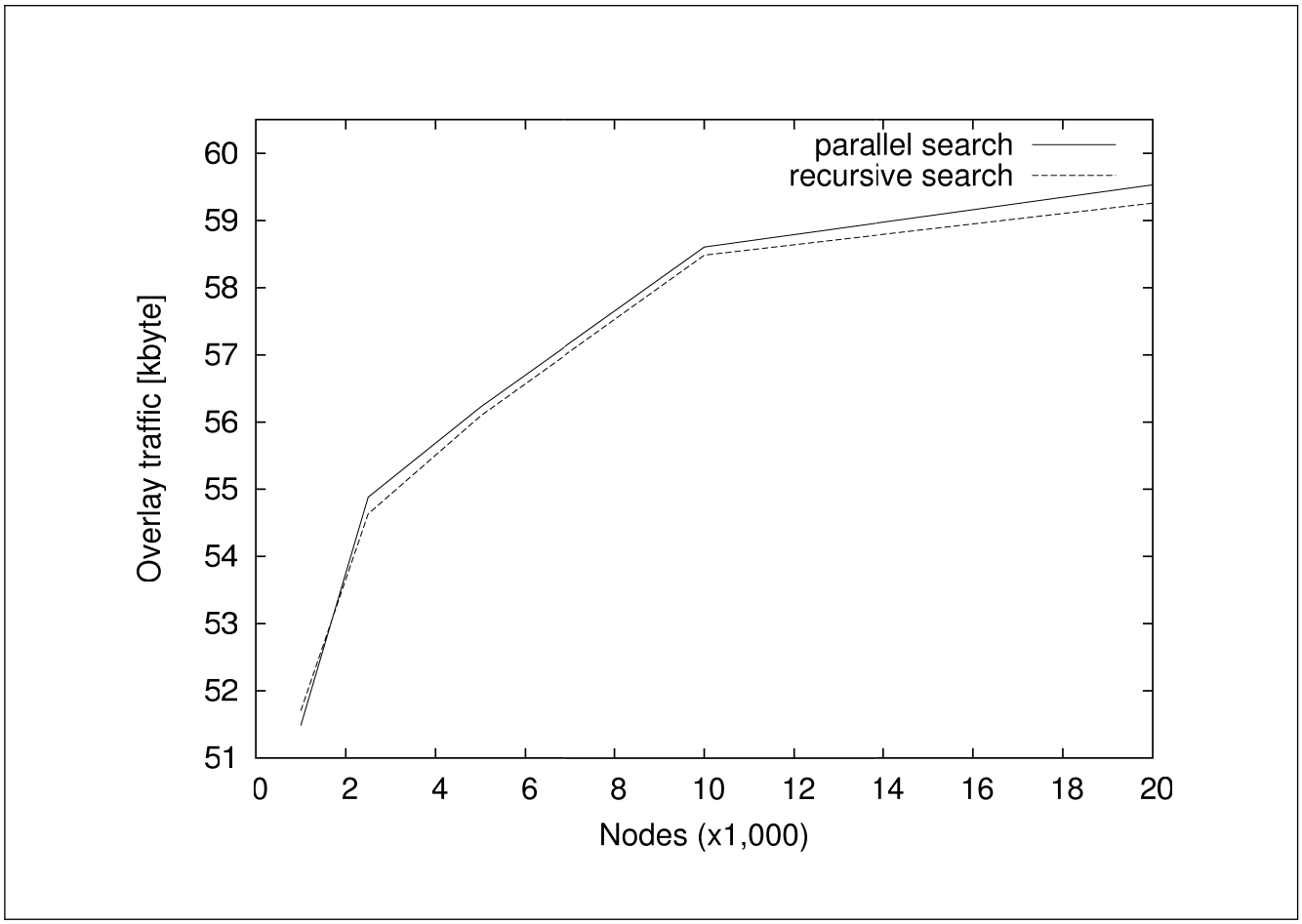

Figure 5.8: Traffic – overlay traffic

As expected, with about 25 kbyte the recursive query algorithm requires less traffic at transferring the result lists than the parallel query algorithm with about 50 kbyte (see Figure 5.9). This is because the biggest result list ever transfered in the recursive process is as big as the smallest one stored on the queried nodes. In this evaluation most queries have at least one sub-query with just 10 results and all queries should return exactly 10 results. Therefore in most cases the recursive query algorithm receives only result lists with the size of 10 whereas the parallel query algorithm receives all lists in their full size (see Table 4.7).
It is important to note that with a bigger index and therefore bigger result lists this difference might change drastically, depending on the difference between the average result list size and the average lowest result list size for all queries.
For this evaluations index the consumed traffic should not change noticeably when increasing the network size, as the result lists will not increase in size and the ID lookup still only contacts the same *k* nodes. The needed IP traffic for queries on a network size of 1,000,000 nodes will therefore be the same 50 kbyte for the parallel query algorithm and 25 kbyte for the recursive query algorithm. Together with the overlay traffic this leads to a predictable traffic of 120 kbyte for a parallel query and 95 kbyte for the recursive query, which saves about 20% of traffic on the recursive search compared to the parallel search.

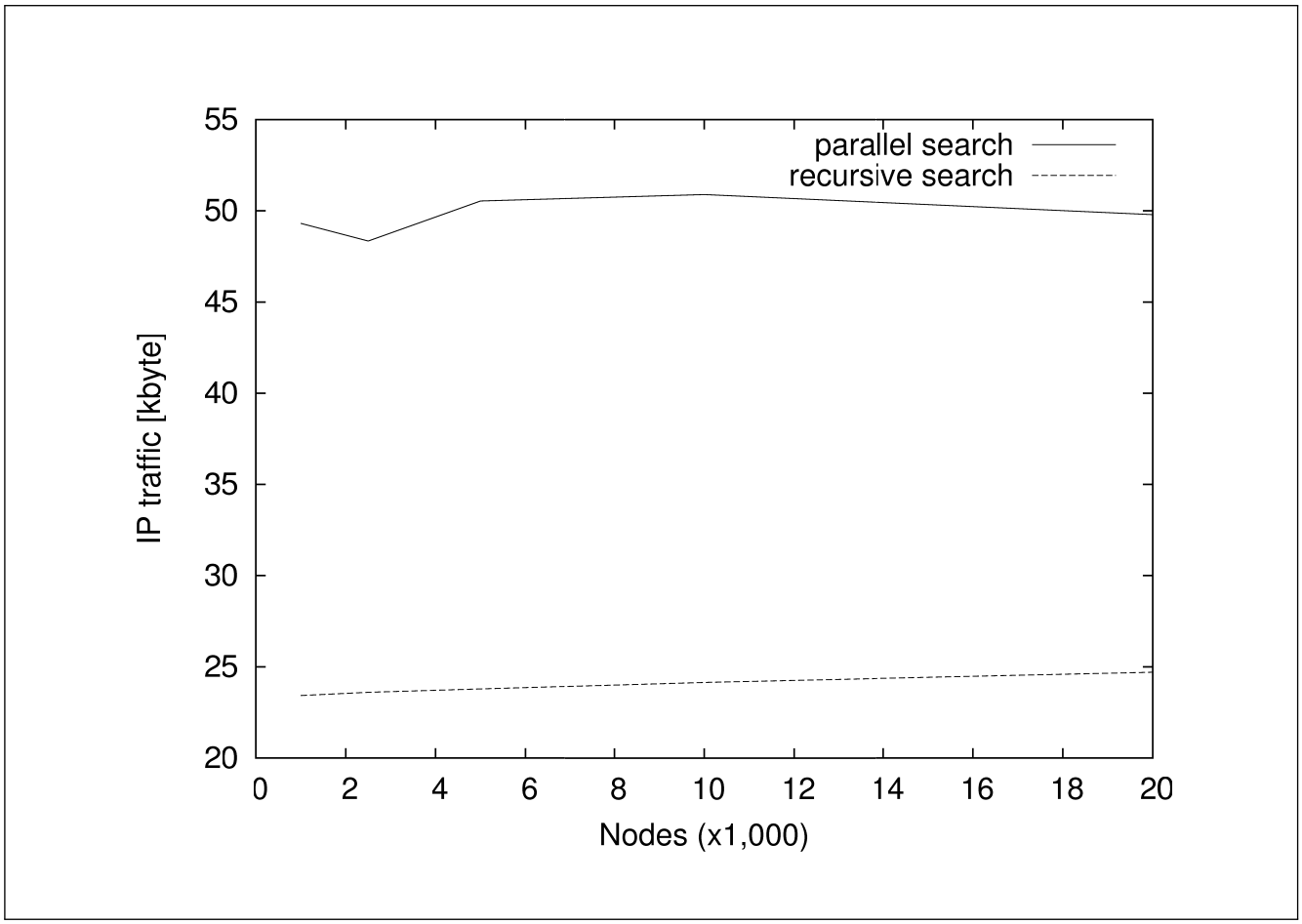

Figure 5.9: Traffic – IP traffic

However the recursive query algorithm needs to send out about as much traffic as it receives (see Figure 5.10), with just the first query not being sent with a result list. When factoring in the common asymmetric network connections and real world result lists which certainly are bigger than the ones used for this simulation the recursive query algorithms bottleneck seems to be the upload bandwidth of the querying node, which usually is by far less than the download bandwidth. This furthermore leads to an uneven network load where the queried nodes send far less than they could while the querying node could receive more but can not send any faster. In comparison the parallel search requires some traffic from the queried nodes while receiving as much as its download bandwidth allows.
There are examples in which either one of the algorithms performs better. The recursive query algorithm highly depends on the smallest result list, while the parallel query algorithm simply retrieves all available result lists.

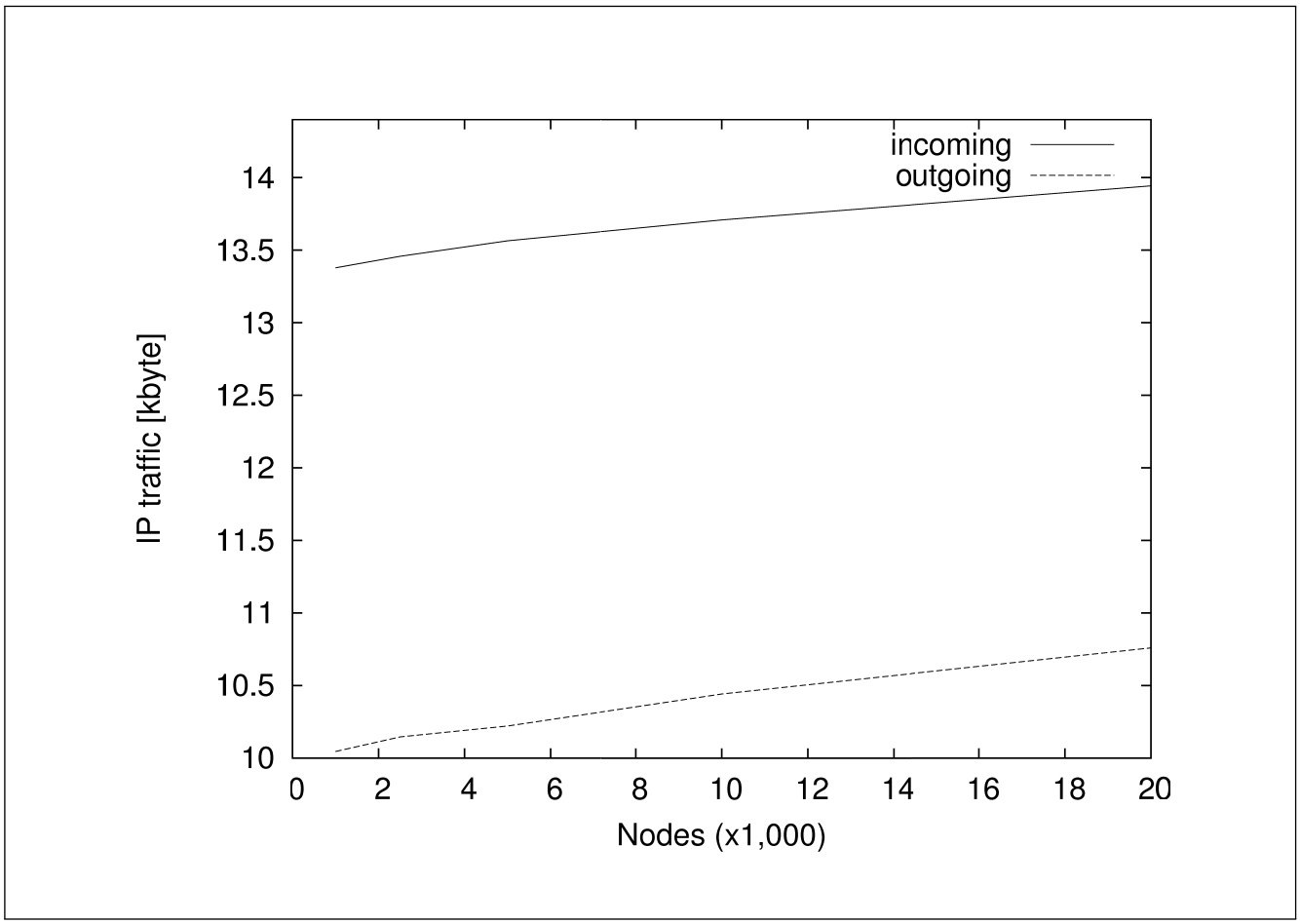

Figure 5.10: Traffic – recursive algorithm in/out

5.6 Query Execution Time

Next to the result quality the query execution time is the most important attribute of a search algorithm. It can be separated into the responsiveness of the query algorithm, which is the elapsed time until the first result is displayed, and the total query execution time which is reached once the last result has been determined.
With only a single exception for the recursive search on a network with 10,000 nodes the queries responsiveness and total execution time are nearly identical (see Figure 5.11). This is because for both algorithms there exists one incoming result list which changes the intersected result list size from no entries to almost all if not all possible results. Additionally there is a high probability that the other result lists for that sub-query arrive at the same time.
For the parallel search this happens when the longest result list has been received, as at this point all other sub-queries already have received most of their answers due to the shorter transmission times of their smaller result lists and thus the intersection with the newly received list delivers results.
The recursive search processes its sub-queries one after the other, and therefore also has a point at which the intersected result list grows from no results to almost all if not all results when the last sub-query is processed.

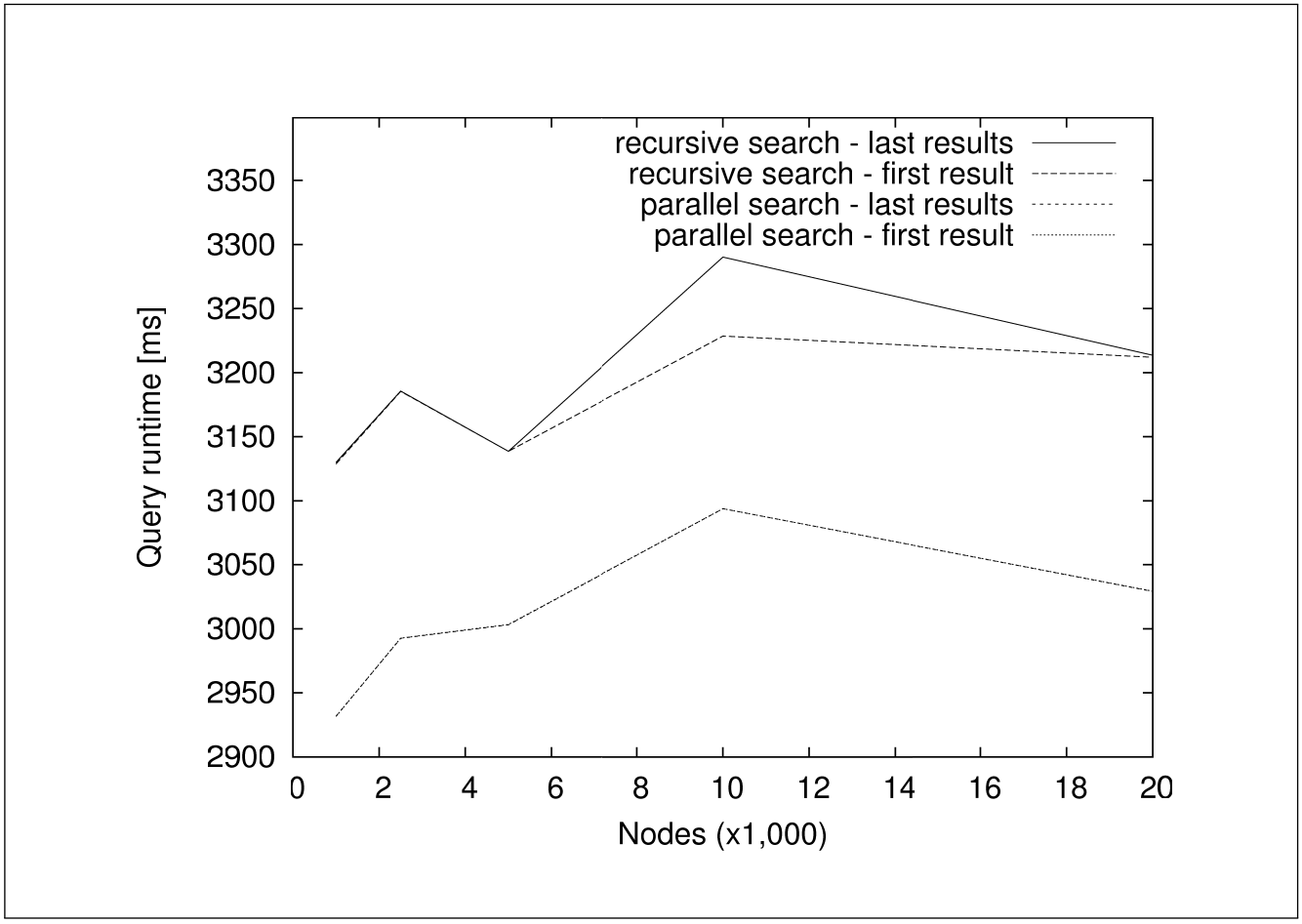

Figure 5.11: Query execution time

A comparison of simulation results with similar churn attributes (according to Figure 5.5) provides a better insight on the algorithms performances (see Figure 5.12). Due to the limited network size the parallel query algorithm still shows no sign of converging towards a certain time, while the recursive search algorithms query execution time increases less with a rising node count. This is due to the previously described increase in ID lookup traffic and IP traffic.
Taking previous results into account probable query execution times for a network size of 1,000,000 nodes would be 3,100ms for the parallel search and 3,300ms for the recursive search. This can be safely assumed as the ID lookup timeout limits the first step of both algorithms to 2,000ms and there will always be a maximum of $k = 10$ nodes to contact per sub-query.

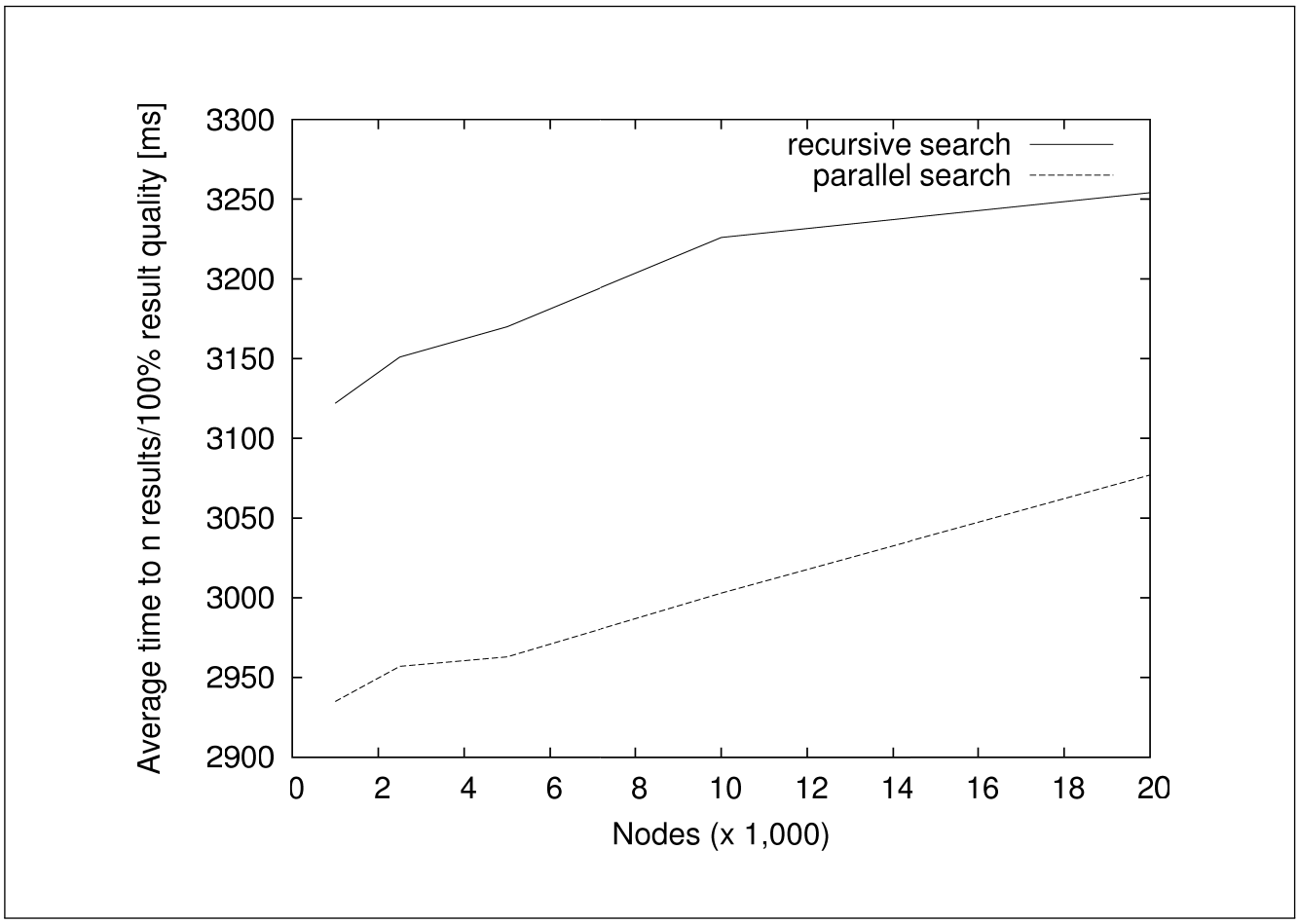

Figure 5.12: Query execution time with similar churn

6 Conclusion

This thesis is a first step towards simulating and evaluating full-text search algorithms on Peer-to-Peer networks. It uses the Kademlia network, which is one of the best performing DHT networks to date. Therefore the limits encountered can be seen as the upper limits of what is possible with current DHT solutions.

6.1 Discussion

Queries on a Peer-to-Peer network used exclusively for full-text search perform quite well. They are mainly limited by the size of the index as each keyword leads to the transfer of up to k result lists.
While the parallel search is a straight forward implementation the recursive search requires an iterative aproach to run on the Kademlia network. Both algorithms should perform well for small amounts of data, but would require some modifications to run on bigger data sets such as Wikipedia. With commonly used asymmetric Internet connections the recursive search is restricted by the upload bandwidth of the querying node, whereas the parallel search solely depends on its download bandwidth.
The query execution time and result quality do not depend much on the network size, meaning that even relatively small networks with sizes between 1,000 and 20,000 nodes are able to maintain and provide an inverted index of acceptable size. The average query execution time consists of a 2 second ID lookup timeout and the time needed to transfer the result lists. The recursive search algorithm needs another 200 ms to retrieve the result list sizes from the remote peers.
With some modifications it should be possible to replace the full-text search of pages such as Wikipedia with a DHT based solution. This would however require a service running for periods of time on the participating nodes instead of a simple Java applet which is terminated when the search page is closed or left, as the latter one would drastically reduce the mean session length and lead to a suboptimal network churn. It would also work with just some enthusiasts continuously running their clients and other users only joining the network to run a query as Kademlia is able to favor well running nodes. This would however be similar to setting up dedicated servers and goes against the philosophy of Peer-to-Peer networks.
It is pretty unrealistic to run an inverted index matching the size of the Google database anytime soon, as the needed ranking mechanisms might pose a problem when intersecting multiple ranked result lists, but also the required storage space on the client computers would exceed any currently acceptable limit due to the necessary replication. To distribute the data in smaller chunks across the network would require massive modifications to the whole network and does simply not work with a regular DHT. It might probably work with some other distributed concept, but it is safe to say that it is currently not possible to store this much data into a DHT network.
Finally it is safe to say that up to a certain amount of shared data the query execution time is well below the upper tolerable limit of 6 seconds [15], and the result quality will be 100%

as long as the data is stored well in the network. Publishing the data, maintaining it through republishes and minimizing the result lists is by far more crucial to the performance of full-text search on Peer-to-Peer networks than the actual algorithms.

6.2 Future Work

Small full-text search scenarios on the Kademlia network provide promising results and an insight to possible limits. This opens further work to analyze and possibly widen those limits:

- The biggest drawback of the parallel search algorithm is the amount of data the result lists take up. A remote intersection could help reduce those lists. One way to provide the possibility of remote intersection would be to publish data with a full list of all its keywords or keyword hashes. The responsible peers maintain additional entries in their inverted index for them even if they are not within their responsibility. A full-text search query then transfers a list of all its keywords with every sub-query so the remote peer can use this additional information to return an intersected list. It would be interesting to analyze how much the storage requirements on each peer increase through this step, the impact on publication costs and if it is actually practicable with today's average private storage space and Internet connection bandwidth.
- Modifying the recursive search to use a bloom filter before transferring the result lists may save bandwidth as the list grows with each step instead of shrinking. The query execution time might however suffer from the additional communication. It would be interesting to find out if the smaller result lists make up for the additional bloom filter processing and if this increases the amount of data manageable through the distributed inverted index.
- Another way to limit the result list sizes is to invoke a ranking mechanism and cropping the list after the top n results. This can be done by using various ranking algorithms in combination with a feedback mechanism where clicked results get some sort of ranking points. It would be interesting to see if this can be done or if by cropping multiple result lists the resulting intersected result list suffers too gravely.
- Up to a certain amount of published data recursive and parallel search perform pretty well. It would be interesting to find out at which scenarios either the recursive or parallel queries perform better and if switching to the parallel search after retrieving the remote peers' result list sizes and determining them to be too big for the available upload bandwidth improves the performance.
- To make full-text search result lists understandable for users a set of information is necessary such as describing meta-data and citations around the keywords. When using Peer-to-Peer full-text search in real world applications it is either necessary to store more than just a Kademlia key in the inverted index for each keyword or the data for each key in the result list has to be stored under that key and obtained through a lookup. However running an ID lookup for each result generates an unacceptable amount of traffic. Compression algorithms may help save storage space and bandwidth when storing additional data into the index. It would be an interesting work to evaluate several compression algorithms and the impact of features on the overall performance in real world scenarios.

List of Figures

List of Tables

Listings

Bibliography

[1] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. Lecture Notes in Computer Science, 2003.

[2] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. 23rd ICDE, 2007.

[3] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. Lecture Notes in Computer Science, 2003.

[4] Y. Yang, R. Dunlap, M. Rexroad, and B. F. Cooper. Performance of full text search in structured and unstructured peer-to-peer systems. INFOCOM, 2006.

[5] W. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top-k retrieval in peer-to-peer networks. 21st ICDE, 2005.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-adressable network. In *Proceedings of the 2001 SIGCOMM conference*, pages 161 – 172, 2001.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Lecture Notes in Computer Science, 2001.

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

[10] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Lecture Notes in Computer Science*, pages 53–65, 2002.

[11] Wikipedia. Overnet — Wikipedia, the free encyclopedia, 2009. [Online; accessed 11-March-2009].

[12] B. René. A performance evaluation of the kad-protocol. Master's thesis, Universität München, 2006.

[13] M. Steiner, D. Carra, and E. W. Biersack. Faster content access in kad. In *Eighth International Conference on Peer-to-Peer Computing*, pages 195 – 204, 2008.

[14] A. Kovacevic, S. Kaune, P. Mukherjee, N. Liebau, R. Steinmetz Benchmarking Platform for Peer-to-Peer Systems In *it - Information Technology*, pages 312–319, 5/2007.

[15] P.v. Schaik, J. Ling. The effect of system response time on visual search in Web pages. In *The Electronic Library*, pages 264 – 268, 2004.

[16] DIMES. The dimes project - a distributed scientific research project, aimed to study the structure and topology of the internet, 2009. [Online; accessed 12-July-2009].

[17] Akamai. First Quarter 2009 State of the Internet Report. [Online; accessed 16-June-2009].

Eidesstattliche Erklärung

zur Diplomarbeit

Hiermit versichere ich, dass ich die Diplomarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst habe.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

Ich bin damit einverstanden, dass ein Exemplar meiner Diplomarbeit in der Bibliothek ausgeliehen werden kann.